Proceedings of the Seminar

Machine Learning, Theory and Applications

University of Colorado, Colorado Springs

August 8, 2014

Editors: Jugal Kalita Qing Yi, Rory Lewis, Kristen Walcott, and Terrance Boult

Funded by

National Science Foundation

Personalized Learned Model to Predict Being Under the Influence

Miguel Alemán
Department of Electrical and Computer Engineering
University of Puerto Rico at Mayagüez

***Abstract*—This paper focuses on the personalization of a mobile application called *RU Influenced*. This personalization will allow us to measure whether the user is under the influence of alcohol or other drugs like marijuana. The Android platform provides a set of sensor technologies that we can use to estimate a blood-alcohol concentration equivalent influence factor. The structure of this application includes two mobile-based cognitive tests called the Digital Symbol Substitution Test (DSST) and STROOP Test and a reaction time test called the Stop Light Test. This application will provide a set of tools for self-monitoring where users can self-quantify their state and avoid being charged with Driving Under the Influence (DUI).**

***Index Terms*—DSST test, STROOP test, Stop Light Test, Machine Learning, Drunk Driving, driving under the influence.**

I. Introduction

This paper describes an effort intended to create a powerful tool for end-users to assess their level of impairment and avoid driving and other dangerous activities when they are under the influence. According to the National Highway Traffic Safety Administration (NHTSA), nearly 40% of the drivers killed in fatal crashes are under a high-level of influence [1]. Moreover, the socioeconomic impact of driving under the influence is staggering, with a 2006 study estimating it at $129.7 billion in the U.S. Currently, the most common methods used by police units include NHSTA-standardized field sobriety tests and a breath-alcohol test. However, breathalyzers are too expensive for most people to own. For these reasons, we want to provide tools that can decrease these numbers radically and can also easily be obtained by the users.

Currently, twenty-one states including Colorado and the District of Columbia have laws legalizing marijuana in some form. Drugs such as marijuana have no easy field test; most of these drugs require blood-based analysis. Therefore, it is necessary to come up with new methods to identify if an individual is under the influence of these drugs. With a personalized model, we expect to increase the chance that impairment can be accurately determined.

II. Previous Work

Several studies have shown that the DSST and STROOP tests are measures of cognitive functions correlated with levels of impairments or intoxication [2] [3]. However, we are the first group to research the concept of individualized baselines for DSST and STROOP testing. To do this, we have personalized an application called *RU Influenced* that includes both of these cognitive tests. The Stop Light test, a test measuring reaction time, is also included in this application.

A. The Digital Symbol Substitution Test (DSST)

The DSST is frequently used to measure associative abilities [4]. This test is normally administered as a paper-and-pencil task where an individual is given numbers between one and nine and a symbol below each number. On the same page, the subject is given a series of random numbers from one to nine and below each number, there is a blank space where the subject draws the symbol appropriate for each digit, see Figure 1. The subject needs to correctly complete a fixed number of questions as fast as he/she can.

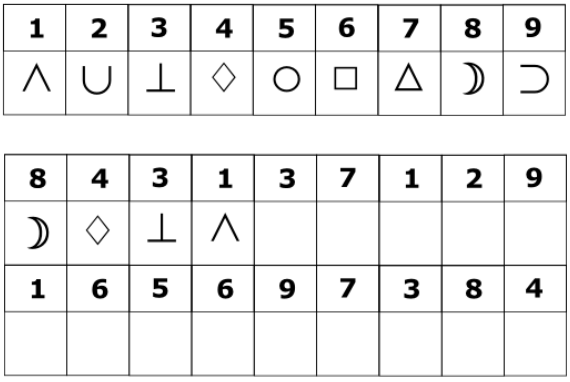

Fig. 1. DSST: Paper-and-pencil task

B. The STROOP Test

The STROOP effect is a demonstration of interference in the reaction time of a task [5]. This test consists of a list of colors printed with a different ink color not denoted by the name. For example, the word ‘pink’ is printed in blue ink, see Figure 2. It has been shown that naming the color of the word takes longer and is more prone to errors if the name of the color is not printed in the color denoted by the word [5]. Another study has shown that intoxicated people need more time to complete the whole series of words than people who are not under the influence [6].

Fig. 2. STROOP Test: List of colors

C. Stop Light Test

The Stop Light Test is a driving response test which measures response times to a range of driving-relevant signs, e.g., stop light colors, randomly appearing signs, and obstacles. There are no studies at the moment that show a relation between this test and level of impairment or intoxication. However, we decided to study and measure this test with respect to the others. For our implementation of this test, we are only considering change in stop light colors, see Figure 3.

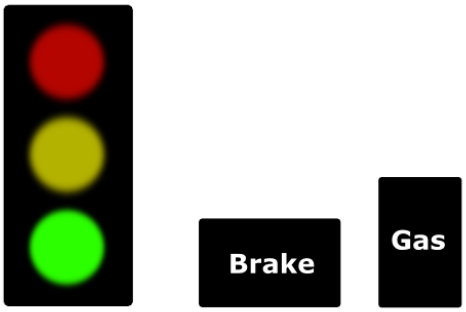

Fig. 3. Stop Light Test: Change in stop light colors

III. Problem Statement

The main goal of this project is to use the *RU Influenced* application to gather enough data to be able to build a personalized model that can predict the level of influence of a user. One of our long-term goals is to be able to correlate this level of influence with a blood alcohol level. There are many variables that we need to consider with respect to the data. For example, one day the subject might perform poorly on one of the tests, but on a subsequent day, he can get much better. This is called *learning effect*. On the other hand, let's say that we are testing the application with a 20-year-old man and a 60-year-old woman. There is a high probability that the 20-year-old man will exhibit a better reaction time than the 60-year-old woman, but that does not mean that the woman is under the influence. These are some difficulties that we need to overcome.

IV. Android Implementation

It is widely known fact that the number of Android users is growing exponentially [7]. The Android platform provides a set of features and technologies very useful for this study. For these reasons, we decided to implement this application using the Android platform. The general framework of *RU Influenced* consists of six main screens: add or select user, user state, start screen, DSST Implementation, STROOP Test Implementation and Stop Light Test Implementation, see Figure 4.

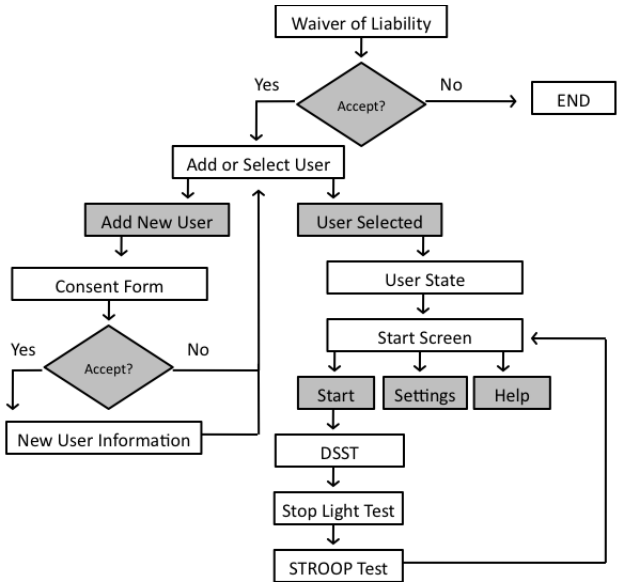

Fig. 4. General Framework

A. Add and Select User

After the user accepts a waiver of liability, a screen will appear asking to select an existing user account or add a new one, see Figure 5. If the user decides to add a new user account, a consent form will appear providing all the necessary information and purposes of this study. The users can either accept or decline this form. If the consent form is accepted, a new screen will appear with a few text fields asking for a username, age and gender. The username is asked to provide the users with a list of user accounts and will not be used for data collection, as all data will be completely anonymous. Users can also delete user accounts from this screen. If the user selects an existing user account, the *user state screen* will appear.

B. User State

In this particular screen, the user is asked a few questions to determine what sort of state the user's mind is in. This helps associate the data with a particular state. For the purposes of this study, we expect the subjects to be sober, but we cannot guarantee that they will in fact be sober. For these reasons, we decided to include an area where the user can specify if he/she has been drinking or if he/she has been using marijuana, see Figure 6. After the user provides all the necessary information the *start screen* will appear.

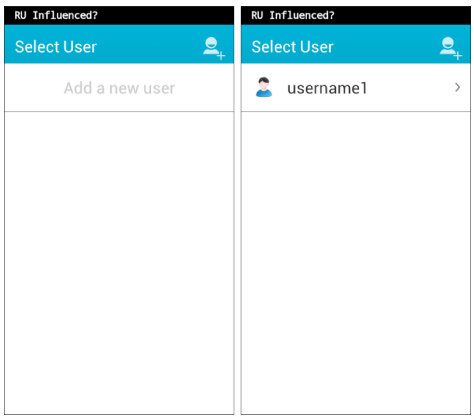

Fig. 5. Add and Select User Screen

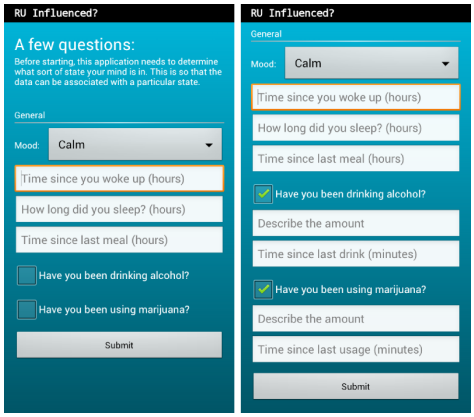

Fig. 6. User State Screen

C. Start Screen

The start screen contains a few options: *start*, *settings* and *help*. If the user selects start, the Digital Symbol Substitution Test will begin, followed by the Stop Light Test and the STROOP Test. On the other hand, if the user selects help, an HTML view containing instructions on how to perform the tests will appear, see Figure 7.

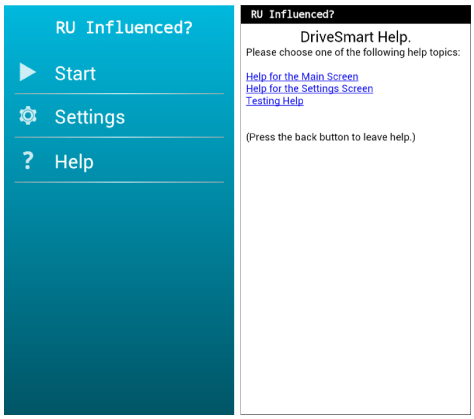

Fig. 7. Start and Help Screens

D. DSST Implementation

As mentioned above, this test is normally used as a paper-and-pencil task; however, there are many ways of implementing a mobile-based version of this test. For this study, we decided to use the *match-nomatch* approach. The program presents the user with two rows of random symbols. Each column of symbols are "matching." Two symbols from that set will appear in the center of the screen. If they are in the same column, the user has to press the match button; otherwise the user has to press the "no match" button. The objective is to press the correct button as fast as possible, see Figure 8. After this test is completed, the Stop Light Test will begin.

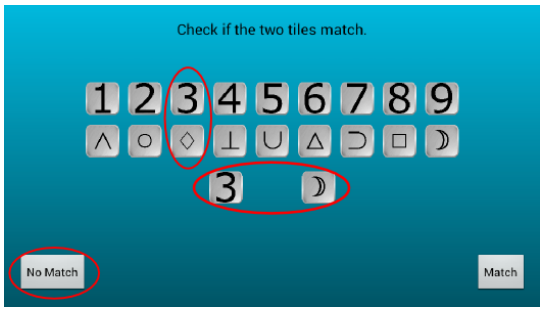

Fig. 8. DSST Implementation

E. Stop Light Test Implementation

In this screen, a stop light will change color each time the user presses a button. The yellow light will be lit when changing from the green color, but it will not necessarily change from yellow to red because this will be predictable for users. Instead, the light colors change randomly. The subject needs to press the accelerate button when the green light is on and the brake button when the red light is on, see Figure 9. After this test is completed, the STROOP Test will begin.

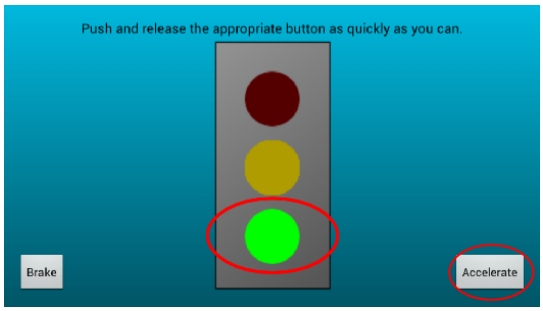

Fig. 9. Stop Light Test Implementation

F. STROOP Test Implementation

For this test, we were able to implement a mobile-based system where the user hears the words/colors and has to select them on the screen, see Figure 10. After this test is completed, the application proceeds to store the data and goes back to the start screen, where users can perform the tests again or simply close the application.

Fig. 10. STROOP Test Implementation

V. Data Storage

To be able to collect the data, we need to use a database system. The Android platform does not interact directly with remote databases but instead uses an SQLite database unique to each application and which can only be accessed inside that application. For the purpose of this study, this will not be very useful since we need to access the collected data outside the application to be able to apply machine learning algorithms to it. Therefore, it is necessary to use a remote database. To achieve this, we use PHP scripts. Through PHP scripts, it is possible to establish a connection between an Android application and a remote MySQL database. The Android platform uses HTTP requests to connect with the PHP scripts, and the PHP scripts are able to send and receive data from the remote database.

VI. Experiments

For this study, we were able to collect data from participants between the ages of 19 and 22. The *RU Influenced* application was downloaded into their Android devices in order to collect data for a period of two weeks. As mentioned above, all participants were encouraged to be sober when taking the tests, this is because one of our goals is to build a personalized model that can learn a user's characteristics when he/she is sober. For the data collection, we used the process described in the *Data Storage* section. The measured time and accuracy of the results (among other features such as age, gender, mood, last time he/she ate, last time he/she slept, how long did he/she sleep, time of the day, date, trial number, etc.) were stored in a remote database. Four more questions were asked to the participants in case they were drinking alcohol or using marijuana, however, since they were all sober, all these questions had a value of zero. All results were stored anonymously.

VII. Technical Approach

The essential function of this application is to predict a level of influence of a user. In other words, we need a numeric value that can be correlated with the level of influence that we want to find. For this reason, a supervised learning algorithm such as multiple linear regression [8] seems like a good approach for this problem.

A. Multiple Linear Regression

In order to perform multiple linear regression in our data, we first need to establish our attributes and our dependent variable. Since the data that we have is not completely numeric, we first changed some attributes from nominal to numeric, e.g., gender, and mood. For our dependent variable we are using a combination of the results denoted as *score*.

$$\text{score} = 100\frac{(\alpha_1+\alpha_2+\alpha_3)}{3} - 10\frac{(\tau_1+\tau_2+\tau_3)}{3}$$

The α- and τ-values are the accuracy and time average of each test, respectively. Using multiple linear regression will allow us to obtain a predicted score value of a new instance.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k$$

So far we are assuming that the relation between the attributes and the dependent variable is completely linear. In real practice this is not true. One of this research questions is how the results change with respect to time. We will see these effects in the *Results* section. However, we first need to find an implementation that can produce the weight of each attribute in order to predict a new score value.

B. Implementation

It is known for a fact that regression coefficients can be obtained from the equation

$$b = (X'X)^{-1}X'Y$$

where X, Y, and b are matrices of the attributes values, class values, and the coefficients values, respectively. However, there are different ways or methods that can be used to obtain these coefficients [9] [10]. For the purposes of this project and taking into account that the models will be build on an Android device, we decided to use Singular Value Decompostion (SVD). It has been proven that SVD is a very useful technique for a number of applications including regression [10]. To simplify this problem and take advantage of the technologies that are available, we are using OpenCV (Open Source Computer Vision) for Android [11]. This library contains different functions of performing SVD that allow us to obtain the regression coefficients of a model very quickly.

C. Main Idea

After we have our model, we are going to be able to predict the score of a new instance. Now, when a user performs the tests we will obtain two different scores, the real score and the score predicted by the model. The next stage of this project is to observe the difference between these two values and relate this difference with a level of influence. To be able to do this we will need to collect data from participants that are under

the influence. This stage of the project will be carried out as future work.

The goal of building a individualized model is to be able to apply *domain adaptation* [12]. This technique consists of taking a learned model that was trained with a lot of data from one source domain and adapt it to a different target domain. In other words, we want to analyze smaller groups of people, calculate separate models for each group or clusters, and then map new users into an appropriate cluster.

VIII. Results

There are two main questions that we expect to answer with these results. The first is something that we mentioned earlier, the *learning effect*. As expected, we can observe that this effect exists in our data, see Figure 11.

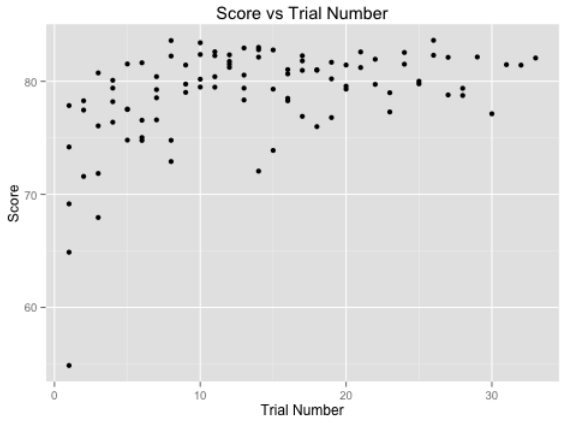

Fig. 11. Learning Effect

From this figure we can infer that the relation between the trial number and the score is not linear but logarithmic. This means that our model is no longer linear regression, we need to add some non-linear elements, making it a non-linear regression model. However, after a certain number of attempts the data behaves constant.

The second questions that we expect to answer is how these results vary according to time of the day. Again, by plotting the data, see Figure 12 we can see that in fact the results vary according to time of the day. For this reason, we are using a third order polynomial in our time data to take into account this daily variation.

It is important to mention that according to the data that we have so far, we cannot infer that the results vary according to the mood of a person, see Figure 13. From this figure, we can observe that there is no significant difference in the results between people who are tired or sleepy and people who are happy or calm. However, we will need more data to conclude that this feature is not relevant for our study.

By adding non-linear elements to our data we have improved our model; however, we still need more data for training and testing. In terms of performance, the SVD libraries of OpenCV seem like the best approach to solve this kind of regression problems on an Android device.

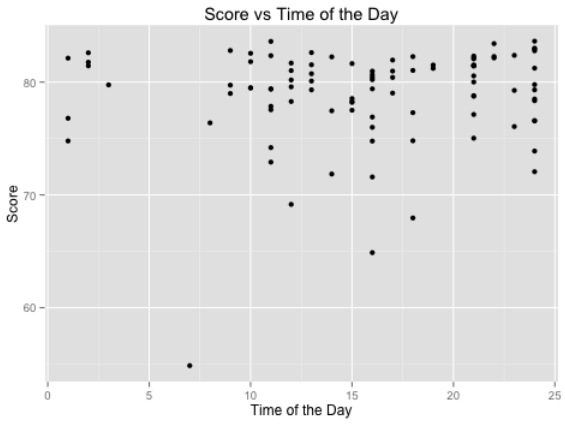

Fig. 12. Daily Variation

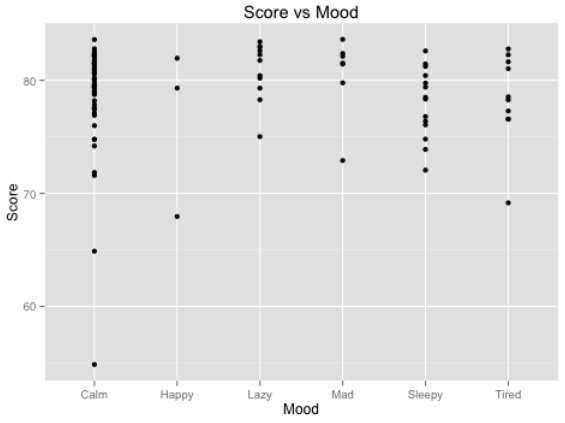

Fig. 13. Mood Variation

IX. Future Work

This application is intended to have a fourth test. This test will be an automated gaze analysis: specifically, the well-known and widely used Horizontal Gaze Nystagmus (HGN) [13]. Irregular motion while tracking at off-angle gazes is an uncontrollable motor reflex when someone is intoxicated, causing the eye to jitter when tracking an object. For this reason, we strongly believe that a phone-based version of this test will be very useful for end-users. One longer-term goal of this application is to be potentially used by third parties, e.g. police officials.

X. Conclusion

In general, *RU Influenced* will allow users to have an idea of their level of influence. After we collect data from users that are under the influence we will be able to give the user an estimate of their level of influence. When we collect this additional data, we expect to observe a significant difference between users who are sober and users who are not. However, from the results obtained so far we can infer many things. As we mentioned in the *Results* section we were able to prove that the *learning effect* exists in our data as well as daily variation of the scores obtained by the users.

ACKNOWLEDGMENT

We would like to thank the National Science Foundation (NFS) REU Grant 1359275 for sponsoring this research.

REFERENCES

[1] R. S. Kennedy, J. J. Turnage, G. G. Rugotzke, and W. P. Dunlap, "Indexing cognitive tests to alcohol dosage and comparison to standardized field sobriety tests," *Journal of Studies on Alcohol and Drugs*, vol. 55, no. 5, p. 615, 1994.

[2] S. J. Heishman, K. Arasteh, and M. L. Stitzer, "Comparative effects of alcohol and marijuana on mood, memory, and performance," *Pharmacology Biochemistry and Behavior*, vol. 58, no. 1, pp. 93–101, 1997.

[3] T. Brumback, D. Cao, and A. King, "Effects of alcohol on psychomotor performance and perceived impairment in heavy binge social drinkers," *Drug and alcohol dependence*, vol. 91, no. 1, pp. 10–17, 2007.

[4] D. R. McLeod, R. R. Griffiths, G. E. Bigelow, and J. Yingling, "An automated version of the digit symbol substitution test (dsst)," *Behavior Research Methods & Instrumentation*, vol. 14, no. 5, pp. 463–466, 1982.

[5] C. M. MacLeod and P. A. MacDonald, "Interdimensional interference in the stroop effect: Uncovering the cognitive and neural anatomy of attention," *Trends in cognitive sciences*, vol. 4, no. 10, pp. 383–391, 2000.

[6] R. Gustafson and H. Kallmen, "Effects of alcohol on cognitive performance measured with stroop's color word test," *Perceptual and motor skills*, vol. 71, no. 1, pp. 99–105, 1990.

[7] N. Gandhewar and R. Sheikh, "Google android: An emerging software platform for mobile devices," *International Journal on Computer Science and Engineering*, vol. 1, no. 1, pp. 12–17, 2010.

[8] L. S. Aiken, S. G. West, and S. C. Pitts, "Multiple linear regression," *Handbook of psychology*, 2003.

[9] W. Gander, "Algorithms for the qr decomposition," in *Seminar für Angewandte Mathematik: Research report*, 1980.

[10] J. Mandel, "Use of the singular value decomposition in regression analysis," *The American Statistician*, vol. 36, no. 1, pp. 15–24, 1982.

[11] G. Bradski, "Opencv," *Dr. Dobb's Journal of Software Tools*, 2000.

[12] S. Ben-David, J. Blitzer, K. Crammer, F. Pereira *et al.*, "Analysis of representations for domain adaptation," *Advances in neural information processing systems*, vol. 19, p. 137, 2007.

[13] S. E. Busloff, "Can your eyes be used against you? the use of the horizontal gaze nystagmus test in the courtroom," *Journal of Criminal Law and Criminology*, pp. 203–238, 1993.

Personalized Learned Model to Predict Being Under the Influence

Miguel Alemán
Department of Electrical and Computer Engineering
University of Puerto Rico at Mayagüez

***Abstract*—This paper focuses on the personalization of a mobile application called *RU Influenced*. This personalization will allow us to measure whether the user is under the influence of alcohol or other drugs like marijuana. The Android platform provides a set of sensor technologies that we can use to estimate a blood-alcohol concentration equivalent influence factor. The structure of this application includes two mobile-based cognitive tests called the Digital Symbol Substitution Test (DSST) and STROOP Test and a reaction time test called the Stop Light Test. This application will provide a set of tools for self-monitoring where users can self-quantify their state and avoid being charged with Driving Under the Influence (DUI).**

***Index Terms*—DSST test, STROOP test, Stop Light Test, Machine Learning, Drunk Driving, driving under the influence.**

I. Introduction

This paper describes an effort intended to create a powerful tool for end-users to assess their level of impairment and avoid driving and other dangerous activities when they are under the influence. According to the National Highway Traffic Safety Administration (NHTSA), nearly 40% of the drivers killed in fatal crashes are under a high-level of influence [1]. Moreover, the socioeconomic impact of driving under the influence is staggering, with a 2006 study estimating it at $129.7 billion in the U.S. Currently, the most common methods used by police units include NHSTA-standardized field sobriety tests and a breath-alcohol test. However, breathalyzers are too expensive for most people to own. For these reasons, we want to provide tools that can decrease these numbers radically and can also easily be obtained by the users.

Currently, twenty-one states including Colorado and the District of Columbia have laws legalizing marijuana in some form. Drugs such as marijuana have no easy field test; most of these drugs require blood-based analysis. Therefore, it is necessary to come up with new methods to identify if an individual is under the influence of these drugs. With a personalized model, we expect to increase the chance that impairment can be accurately determined.

II. Previous Work

Several studies have shown that the DSST and STROOP tests are measures of cognitive functions correlated with levels of impairments or intoxication [2] [3]. However, we are the first group to research the concept of individualized baselines for DSST and STROOP testing. To do this, we have personalized an application called *RU Influenced* that includes both of these cognitive tests. The Stop Light test, a test measuring reaction time, is also included in this application.

A. The Digital Symbol Substitution Test (DSST)

The DSST is frequently used to measure associative abilities [4]. This test is normally administered as a paper-and-pencil task where an individual is given numbers between one and nine and a symbol below each number. On the same page, the subject is given a series of random numbers from one to nine and below each number, there is a blank space where the subject draws the symbol appropriate for each digit, see Figure 1. The subject needs to correctly complete a fixed number of questions as fast as he/she can.

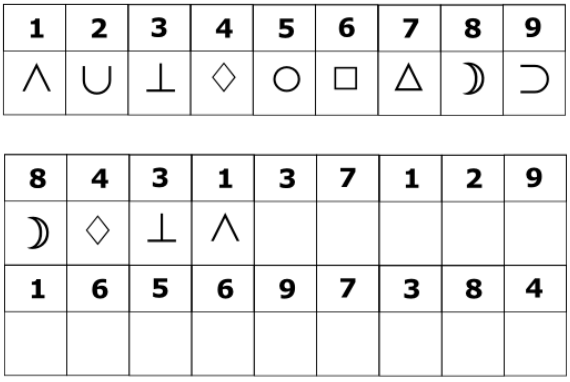

Fig. 1. DSST: Paper-and-pencil task

B. The STROOP Test

The STROOP effect is a demonstration of interference in the reaction time of a task [5]. This test consists of a list of colors printed with a different ink color not denoted by the name. For example, the word ‘pink’ is printed in blue ink, see Figure 2. It has been shown that naming the color of the word takes longer and is more prone to errors if the name of the color is not printed in the color denoted by the word [5]. Another study has shown that intoxicated people need more time to complete the whole series of words than people who are not under the influence [6].

Fig. 2. STROOP Test: List of colors

C. Stop Light Test

The Stop Light Test is a driving response test which measures response times to a range of driving-relevant signs, e.g., stop light colors, randomly appearing signs, and obstacles. There are no studies at the moment that show a relation between this test and level of impairment or intoxication. However, we decided to study and measure this test with respect to the others. For our implementation of this test, we are only considering change in stop light colors, see Figure 3.

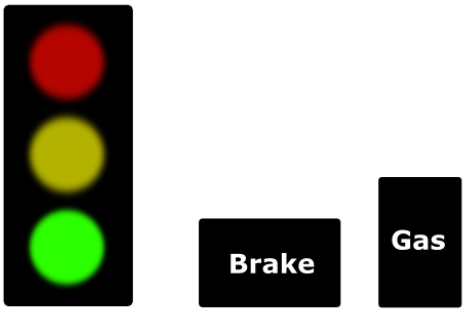

Fig. 3. Stop Light Test: Change in stop light colors

III. Problem Statement

The main goal of this project is to use the *RU Influenced* application to gather enough data to be able to build a personalized model that can predict the level of influence of a user. One of our long-term goals is to be able to correlate this level of influence with a blood alcohol level. There are many variables that we need to consider with respect to the data. For example, one day the subject might perform poorly on one of the tests, but on a subsequent day, he can get much better. This is called *learning effect*. On the other hand, let's say that we are testing the application with a 20-year-old man and a 60-year-old woman. There is a high probability that the 20-year-old man will exhibit a better reaction time than the 60-year-old woman, but that does not mean that the woman is under the influence. These are some difficulties that we need to overcome.

IV. Android Implementation

It is widely known fact that the number of Android users is growing exponentially [7]. The Android platform provides a set of features and technologies very useful for this study. For these reasons, we decided to implement this application using the Android platform. The general framework of *RU Influenced* consists of six main screens: add or select user, user state, start screen, DSST Implementation, STROOP Test Implementation and Stop Light Test Implementation, see Figure 4.

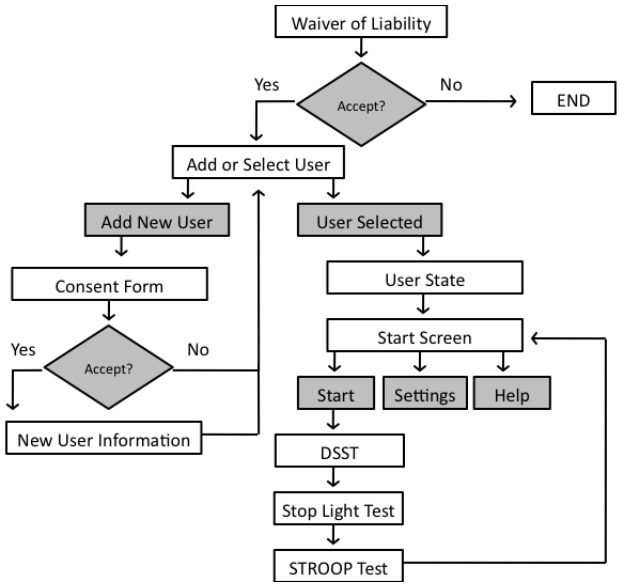

Fig. 4. General Framework

A. Add and Select User

After the user accepts a waiver of liability, a screen will appear asking to select an existing user account or add a new one, see Figure 5. If the user decides to add a new user account, a consent form will appear providing all the necessary information and purposes of this study. The users can either accept or decline this form. If the consent form is accepted, a new screen will appear with a few text fields asking for a username, age and gender. The username is asked to provide the users with a list of user accounts and will not be used for data collection, as all data will be completely anonymous. Users can also delete user accounts from this screen. If the user selects an existing user account, the *user state screen* will appear.

B. User State

In this particular screen, the user is asked a few questions to determine what sort of state the user's mind is in. This helps associate the data with a particular state. For the purposes of this study, we expect the subjects to be sober, but we cannot guarantee that they will in fact be sober. For these reasons, we decided to include an area where the user can specify if he/she has been drinking or if he/she has been using marijuana, see Figure 6. After the user provides all the necessary information the *start screen* will appear.

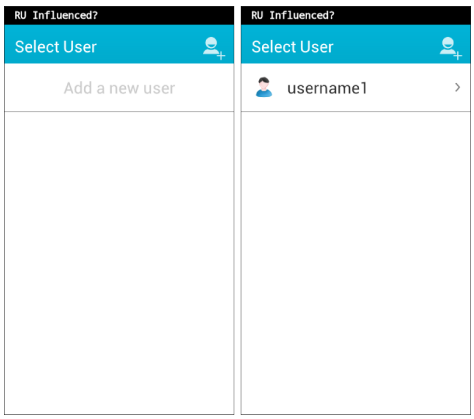

Fig. 5. Add and Select User Screen

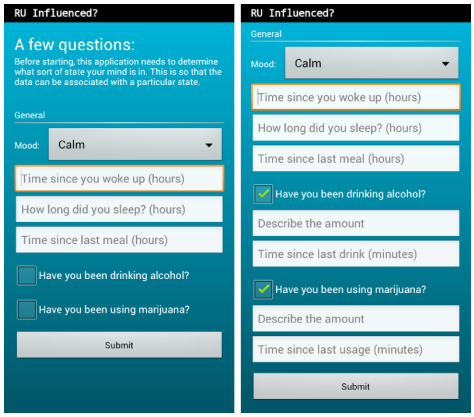

Fig. 6. User State Screen

C. Start Screen

The start screen contains a few options: *start*, *settings* and *help*. If the user selects start, the Digital Symbol Substitution Test will begin, followed by the Stop Light Test and the STROOP Test. On the other hand, if the user selects help, an HTML view containing instructions on how to perform the tests will appear, see Figure 7.

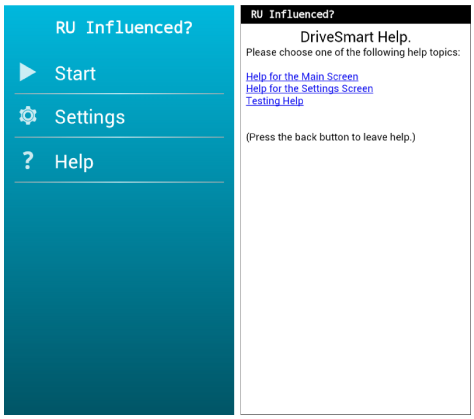

Fig. 7. Start and Help Screens

D. DSST Implementation

As mentioned above, this test is normally used as a paper-and-pencil task; however, there are many ways of implementing a mobile-based version of this test. For this study, we decided to use the *match-nomatch* approach. The program presents the user with two rows of random symbols. Each column of symbols are "matching." Two symbols from that set will appear in the center of the screen. If they are in the same column, the user has to press the match button; otherwise the user has to press the "no match" button. The objective is to press the correct button as fast as possible, see Figure 8. After this test is completed, the Stop Light Test will begin.

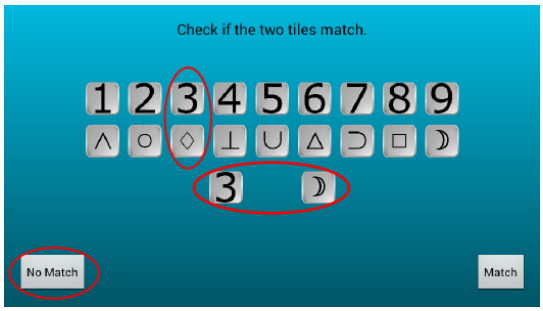

Fig. 8. DSST Implementation

E. Stop Light Test Implementation

In this screen, a stop light will change color each time the user presses a button. The yellow light will be lit when changing from the green color, but it will not necessarily change from yellow to red because this will be predictable for users. Instead, the light colors change randomly. The subject needs to press the accelerate button when the green light is on and the brake button when the red light is on, see Figure 9. After this test is completed, the STROOP Test will begin.

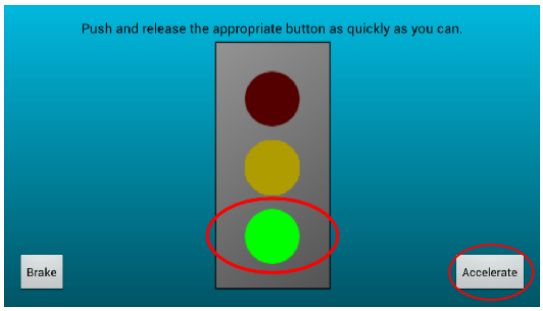

Fig. 9. Stop Light Test Implementation

F. STROOP Test Implementation

For this test, we were able to implement a mobile-based system where the user hears the words/colors and has to select them on the screen, see Figure 10. After this test is completed, the application proceeds to store the data and goes back to the start screen, where users can perform the tests again or simply close the application.

Fig. 10. STROOP Test Implementation

V. Data Storage

To be able to collect the data, we need to use a database system. The Android platform does not interact directly with remote databases but instead uses an SQLite database unique to each application and which can only be accessed inside that application. For the purpose of this study, this will not be very useful since we need to access the collected data outside the application to be able to apply machine learning algorithms to it. Therefore, it is necessary to use a remote database. To achieve this, we use PHP scripts. Through PHP scripts, it is possible to establish a connection between an Android application and a remote MySQL database. The Android platform uses HTTP requests to connect with the PHP scripts, and the PHP scripts are able to send and receive data from the remote database.

VI. Experiments

For this study, we were able to collect data from participants between the ages of 19 and 22. The *RU Influenced* application was downloaded into their Android devices in order to collect data for a period of two weeks. As mentioned above, all participants were encouraged to be sober when taking the tests, this is because one of our goals is to build a personalized model that can learn a user's characteristics when he/she is sober. For the data collection, we used the process described in the *Data Storage* section. The measured time and accuracy of the results (among other features such as age, gender, mood, last time he/she ate, last time he/she slept, how long did he/she sleep, time of the day, date, trial number, etc.) were stored in a remote database. Four more questions were asked to the participants in case they were drinking alcohol or using marijuana, however, since they were all sober, all these questions had a value of zero. All results were stored anonymously.

VII. Technical Approach

The essential function of this application is to predict a level of influence of a user. In other words, we need a numeric value that can be correlated with the level of influence that we want to find. For this reason, a supervised learning algorithm such as multiple linear regression [8] seems like a good approach for this problem.

A. Multiple Linear Regression

In order to perform multiple linear regression in our data, we first need to establish our attributes and our dependent variable. Since the data that we have is not completely numeric, we first changed some attributes from nominal to numeric, e.g., gender, and mood. For our dependent variable we are using a combination of the results denoted as *score*.

$$\text{score} = 100\frac{(\alpha_1+\alpha_2+\alpha_3)}{3} - 10\frac{(\tau_1+\tau_2+\tau_3)}{3}$$

The α- and τ-values are the accuracy and time average of each test, respectively. Using multiple linear regression will allow us to obtain a predicted score value of a new instance.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k$$

So far we are assuming that the relation between the attributes and the dependent variable is completely linear. In real practice this is not true. One of this research questions is how the results change with respect to time. We will see these effects in the *Results* section. However, we first need to find an implementation that can produce the weight of each attribute in order to predict a new score value.

B. Implementation

It is known for a fact that regression coefficients can be obtained from the equation

$$b = (X'X)^{-1}X'Y$$

where X, Y, and b are matrices of the attributes values, class values, and the coefficients values, respectively. However, there are different ways or methods that can be used to obtain these coefficients [9] [10]. For the purposes of this project and taking into account that the models will be build on an Android device, we decided to use Singular Value Decompostion (SVD). It has been proven that SVD is a very useful technique for a number of applications including regression [10]. To simplify this problem and take advantage of the technologies that are available, we are using OpenCV (Open Source Computer Vision) for Android [11]. This library contains different functions of performing SVD that allow us to obtain the regression coefficients of a model very quickly.

C. Main Idea

After we have our model, we are going to be able to predict the score of a new instance. Now, when a user performs the tests we will obtain two different scores, the real score and the score predicted by the model. The next stage of this project is to observe the difference between these two values and relate this difference with a level of influence. To be able to do this we will need to collect data from participants that are under

the influence. This stage of the project will be carried out as future work.

The goal of building a individualized model is to be able to apply *domain adaptation* [12]. This technique consists of taking a learned model that was trained with a lot of data from one source domain and adapt it to a different target domain. In other words, we want to analyze smaller groups of people, calculate separate models for each group or clusters, and then map new users into an appropriate cluster.

VIII. Results

There are two main questions that we expect to answer with these results. The first is something that we mentioned earlier, the *learning effect*. As expected, we can observe that this effect exists in our data, see Figure 11.

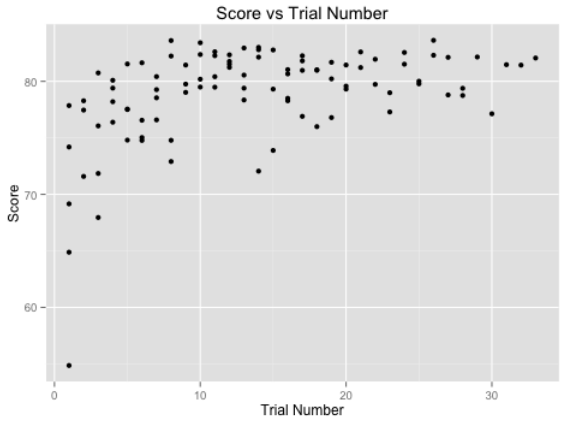

Fig. 11. Learning Effect

From this figure we can infer that the relation between the trial number and the score is not linear but logarithmic. This means that our model is no longer linear regression, we need to add some non-linear elements, making it a non-linear regression model. However, after a certain number of attempts the data behaves constant.

The second questions that we expect to answer is how these results vary according to time of the day. Again, by plotting the data, see Figure 12 we can see that in fact the results vary according to time of the day. For this reason, we are using a third order polynomial in our time data to take into account this daily variation.

It is important to mention that according to the data that we have so far, we cannot infer that the results vary according to the mood of a person, see Figure 13. From this figure, we can observe that there is no significant difference in the results between people who are tired or sleepy and people who are happy or calm. However, we will need more data to conclude that this feature is not relevant for our study.

By adding non-linear elements to our data we have improved our model; however, we still need more data for training and testing. In terms of performance, the SVD libraries of OpenCV seem like the best approach to solve this kind of regression problems on an Android device.

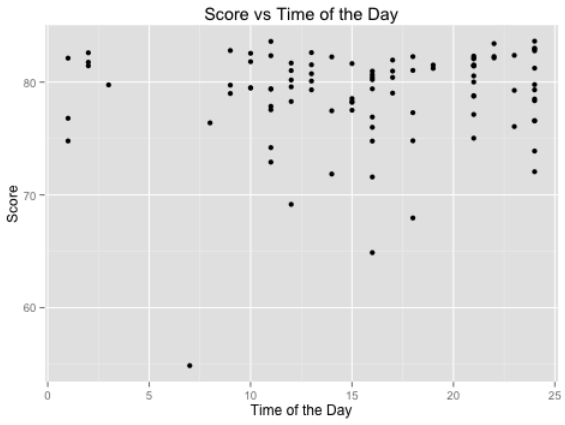

Fig. 12. Daily Variation

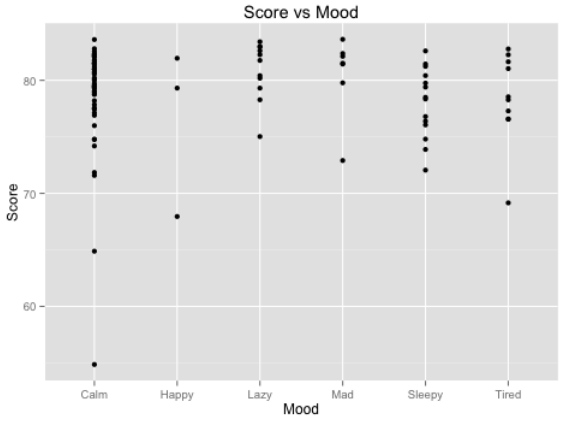

Fig. 13. Mood Variation

IX. Future Work

This application is intended to have a fourth test. This test will be an automated gaze analysis: specifically, the well-known and widely used Horizontal Gaze Nystagmus (HGN) [13]. Irregular motion while tracking at off-angle gazes is an uncontrollable motor reflex when someone is intoxicated, causing the eye to jitter when tracking an object. For this reason, we strongly believe that a phone-based version of this test will be very useful for end-users. One longer-term goal of this application is to be potentially used by third parties, e.g. police officials.

X. Conclusion

In general, *RU Influenced* will allow users to have an idea of their level of influence. After we collect data from users that are under the influence we will be able to give the user an estimate of their level of influence. When we collect this additional data, we expect to observe a significant difference between users who are sober and users who are not. However, from the results obtained so far we can infer many things. As we mentioned in the *Results* section we were able to prove that the *learning effect* exists in our data as well as daily variation of the scores obtained by the users.

ACKNOWLEDGMENT

We would like to thank the National Science Foundation (NFS) REU Grant 1359275 for sponsoring this research.

REFERENCES

[1] R. S. Kennedy, J. J. Turnage, G. G. Rugotzke, and W. P. Dunlap, "Indexing cognitive tests to alcohol dosage and comparison to standardized field sobriety tests," *Journal of Studies on Alcohol and Drugs*, vol. 55, no. 5, p. 615, 1994.

[2] S. J. Heishman, K. Arasteh, and M. L. Stitzer, "Comparative effects of alcohol and marijuana on mood, memory, and performance," *Pharmacology Biochemistry and Behavior*, vol. 58, no. 1, pp. 93–101, 1997.

[3] T. Brumback, D. Cao, and A. King, "Effects of alcohol on psychomotor performance and perceived impairment in heavy binge social drinkers," *Drug and alcohol dependence*, vol. 91, no. 1, pp. 10–17, 2007.

[4] D. R. McLeod, R. R. Griffiths, G. E. Bigelow, and J. Yingling, "An automated version of the digit symbol substitution test (dsst)," *Behavior Research Methods & Instrumentation*, vol. 14, no. 5, pp. 463–466, 1982.

[5] C. M. MacLeod and P. A. MacDonald, "Interdimensional interference in the stroop effect: Uncovering the cognitive and neural anatomy of attention," *Trends in cognitive sciences*, vol. 4, no. 10, pp. 383–391, 2000.

[6] R. Gustafson and H. Kallmen, "Effects of alcohol on cognitive performance measured with stroop's color word test," *Perceptual and motor skills*, vol. 71, no. 1, pp. 99–105, 1990.

[7] N. Gandhewar and R. Sheikh, "Google android: An emerging software platform for mobile devices," *International Journal on Computer Science and Engineering*, vol. 1, no. 1, pp. 12–17, 2010.

[8] L. S. Aiken, S. G. West, and S. C. Pitts, "Multiple linear regression," *Handbook of psychology*, 2003.

[9] W. Gander, "Algorithms for the qr decomposition," in *Seminar für Angewandte Mathematik: Research report*, 1980.

[10] J. Mandel, "Use of the singular value decomposition in regression analysis," *The American Statistician*, vol. 36, no. 1, pp. 15–24, 1982.

[11] G. Bradski, "Opencv," *Dr. Dobb's Journal of Software Tools*, 2000.

[12] S. Ben-David, J. Blitzer, K. Crammer, F. Pereira *et al.*, "Analysis of representations for domain adaptation," *Advances in neural information processing systems*, vol. 19, p. 137, 2007.

[13] S. E. Busloff, "Can your eyes be used against you? the use of the horizontal gaze nystagmus test in the courtroom," *Journal of Criminal Law and Criminology*, pp. 203–238, 1993.

Stencil Code Optimization for GPUs Through Machine Learning

Adam Barker
University of Colorado at Colorado Springs
abarker2@uccs.edu

***Abstract*—The microprocessor field today has begun to reach its limits as power and thermal constraints have been met and no longer can much leverage of increasing the processor's clock speed be achieved. Thus, much of the scientific and engineering community has shifted to using many-core architectures, such as GPUs, in order to do parallel computations. This paper focuses on the use of genetic algorithms to guide the optimization of stencil codes on NVIDIA's Compute Unified Device Architecture (CUDA) based GPUs and GPGPUs. In particular, we have implemented two separate stencil kernels (Jacobi 7 point and 27 point) in CUDA with each implementation parameterized for several optimiation parameters (thread blocking and loop unrolling factors). We then used a genetic algorithm to find optimal configurations for each kernel. This genetic algorithm is one part of our proposed solution of using an optimization framework incorporating the genetic algorithm to auto-tune automatically optimized stencil codes. Our results show that using a genetic algorithm to auto-tune stencil code optimizations is a valid approach of generating near-optimal configurations in a much more timely fashion than an exhaustive search.**

I. Introduction

As microprocessors reach the power wall, benefits of increasing the clock frequency are no longer achieveable as the cost to system stability and cooling is too much to warrant the increase in performance [1]. This has shifted the focus of the parallel community to many-core architectures, such as those found in Graphical Processing Units (GPUs), as they are comprised of a few hundred or thousand simple cores that are capable of performing highly-parallel computations with much more throughput than a typical multi-core system. However, developing parallel algorithms for GPUs can be no simple task for developers as developers must have a firm understanding of the underlying architecture and hardware properties in order to correctly write programs that correctly take advantage of these properties. Thus, there is a desire to develop a method to automatically apply optimizations to GPU programs in order to avoid the necessity of understanding the complexities of the hardware and architecture of the system.

Recently, in order to meet this desire, researchers have devloped several methods in order to automatically tune or automatically generate optimized codes for both GPUs and multi-core systems. However, as more optimizations are discovered, the search space the auto-tuner must search through grows to an amount where auto-tuning is no longer viable as the number of possible combinations of parameters becomes too large to effectively search through. This then sets the perfect stage for a machine learning application to predict the optimal code instead as it does not have to go through the entire search space, but rather make predictions based on previous results.

This work presents a method to use genetic algorithms in order to discover optimized configurations of parameterized CUDA stencil (nearest-neighbor) codes – a class of algorithms that typically work in structured grids to perform computations, such as finite-difference methods for solving parital differential equations, on a node within the grid by doing computations on the neighbors around the given node. Our work focuses on a simple 3D heat equation using two different stencil codes as the training set for a genetic algorithm to search through a search space of several thousand combinations of possible optimization parameters. Although stencil codes are important as scientific computations, they also provide a unique opportunity for hardware benchmarking as they are computationally simple and require a large use of memory, allowing for benchmarking of instruction-level and data-level parallelism [3]. These codes greatly benefit being run on GPUs as the parallel forms of these codes contain a great deal of instruction level parallelism which translates well to SIMD architectures, which are present on GPUs.

This research is the development of the optimal configuration generator portion of the framework detailed in Figure 1. The auto-optimization framework will be used to optimize existing stencil codes using machine learning in order to predict optimal tuning parameters that will be given to the optimizer which will apply these optimizations to the given stencil code and then output the optimized version of the given code. This is done so that developers can easily write unoptimized code for use in their programs and then run this auto-tuning framework on their code in order to use optimized code that correctly fits within their existing program.

In order to train the machine learning portion of the configuration generator, we implemented two stencil kernels to be used across three separate GPUs. The stencil codes we implemented were a 7 point and 27 point Jacobi iterative stencil codes and then parameterized the relevant optimizations that the genetic algorithms would find configurations for so that the fitness test for the genetic algorithm could change these parameters easily before compiling and running.

The optimization parameters considered for the generation of the search space that we used were the number of threads to use in the computation, and the distance to unroll the inner loop in our code. This inner-loop arises from our use

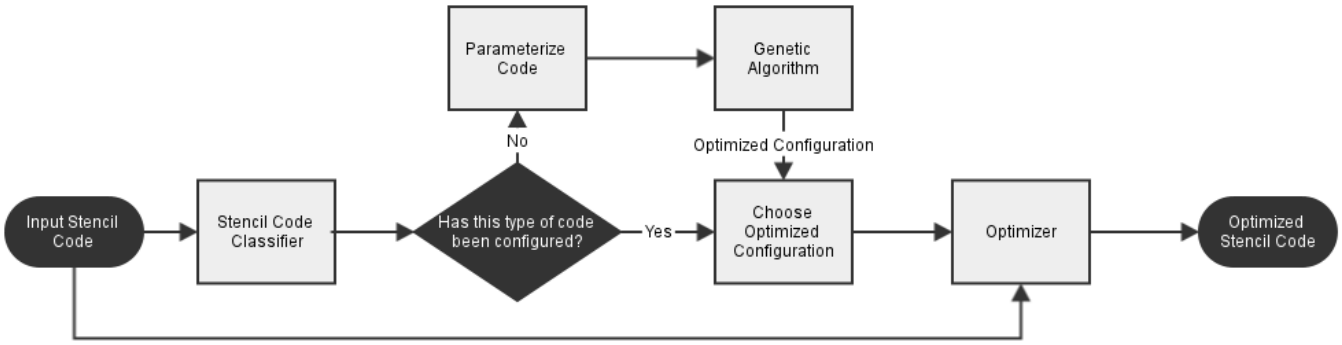

Fig. 1. Overview of auto-optimization framework

of 2.5D blocking, a thread blocking optimization that allows for threads to only be launched in a single plane of the 3D data, and then stream through the remaining axis as the computation goes on. This search space consists of 60 different thread configurations and 192 different loop unrolling configurations, giving us a search space of 11,520 possible combinations. Although the size of this search space is relatively small for most machine learning applications, one must consider the time it takes to compile the code as on our system, typical complilation time is 3 seconds, meaning that an exhaustive search through the search space would take more than 9 hours, whereas our use of a genetic algorithm took on average 8 minutes to find an optimized configuration.

Our contribution is a genetic algorithm that is capable of tuning optimizations on parameterized stencil codes. This genetic algorithm can effectively tune these codes to find near-optimal configurations for the applied optimizations in a very short amount of time, making it an effective method to use for auto-tuning stencil code optimizations.

The rest of the paper is organized into four sections: related work, tuning framework, experimental results, and conclusions and future work. In related work, other research that has been done in the field is presented and summarized along with how it is utilized in this research. The tuning framework section goes into more detail of stencil codes, optimizations, and the genetic algorithm that we used. Experimental results includes the experimental setup and the results we obtained from running our implementation on three different systems as well as a discussion of these results. Conclusions and future work summarizes this research and presents the outlook of incorporating it into future work.

II. Related Work

There exists significant research to automatically tune optimized stencil codes in order to find the best configuration of parameters for such optimizations [1], [3], [8], [11]. Datta et all have demonstrated the usefulness of optimizations with auto-tuning techniques as a means to effectively optimize stencil codes on both CPUs and GPUs [3]. Their work provides an effective base for the challenges of optimizing and auto-tuning stencil codes. Gana et all cite this work as their basis for using machine learning to optimize CPU stencil codes. In their research, they used a genetic algorithm in combination with the KCCA algorithm to perform quick searches through the parameter space of 4×10^7 different combinations. They managed to effectively auto-tune stencil codes on CPUs in two hours using their method [5]. Zhang and Mueller also researched auto-tuning and auto-generation of optimized stencil codes specifically for GPUs and GPU clusters which provides a more specific list of optimizations that are specifically used for GPU stencil code optimizations that were used in this research. In particular, their descriptions of 7-point and 27-point stencils, along with shared memory and register allocation for optimization were used throughout our research. [11].

Many optimizations have been developed over the years for stencil codes [2], [6], [7], [9], [10]. Nguyen et all provided a state-of-the art stencil code optimization that uses a combination of 2.5D thread blocking combined with 1d temporal blocking to create what they have called 3.5D blocking which provides throughput increases on GPUs of about two times what prior research had claimed [10]. In our research, we used their excellent description of 2.5D blocking as one of our optimizations for the genetic algorithm to automatically tune. Nguyen et all's research can also be parameterized by changing the amount of temporal blocking to perform, thus allowing a search space to be created for this optimization which was incorporated into this research.

III. Optimization Framework

A. Stencil Codes

Stencil codes are primarily used to solve partial differential equations in order to perform simulations such as heat flow or electromagnetic field propogation [3]. Most methods for solving these partial differential equations use iterative sweeps through spatial data, performing nearest-neighbor computations which are called stencils. Each node in the computation is weighted based on distance from the central node, which allows for the solving partial differential equations by switching these weights for the coefficients used in the solver. Using this structure, methods are created for different types of partial

differential equation solvers such as Jacobi iterative methods, which are the stencil codes we used in this research.

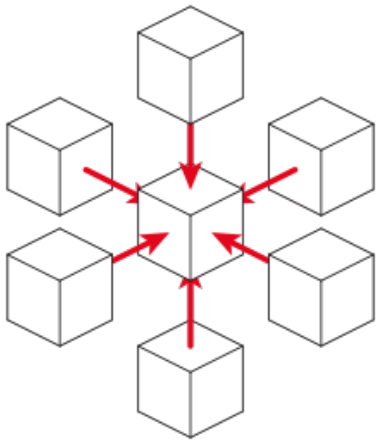

Fig. 2. A 6-point Von-Neuman stencil (credit: wikipedia.org)

As the data sizes used for stencil computations typically range outside the size of available cache memory, there is a large emphasis on data reuse and data-level parallelism in order to fully optimize stencil codes. This can cause portability issues as memory speeds and sizes can differ widely system to system, causing the need to use different parameters for optimizations on different architectures. This then produces a demand for a method to automatically tune stencil code optimizations on each architecture in order to enhance portability of the codes.

Auto-tuning of stencil kernels has become a fairly large area of study in order to work around the necessity of knowledge of the low-level specifications of the architecture in order to optimize the kernel. However, these auto-tuners may have to look in a parameter space that is upwards of 40 million combinations that may take months to fully check every single one for optimal performance [3]. This then creates a demand for a faster optimization process that is still automated in order to create a process that is viable for industry use. Thus, machine learning may be a good option for automatic optimization as it can use reinforcement learning paired with statistical machine learning and genetic algorithms in order to explore the parameter space much faster. Using machine learning may also overcome another downfall of auto-tuning in that each auto-tuner is generally programmed for one architecture, whereas a learner can learn architectures as well and correctly optimize for them.

B. Optimizations

In this research, we applied two types of optimizations to our stencil codes to be used in the tuning phase. The first optimization we used was 2.5D blocking. 2.5D blocking is an optimization for thread blocking of 3D stencil codes that only blocks in the x and y axes of the structured grid. Each thread then streams through the remaining z-axis, allowing for data-reuse of data already fetched by the thread earlier to fulfill data requirements. This optimization reduces the amount of global store and load instructions as threads can keep some data in the registers for quick access for several computations instead of fetching data from global memory each time a node must be calculated. The second optimization used is loop unrolling. Due to the nature of 2.5D blocking in that it must stream through the z-axis via a loop, this loop can be unrolled in order to provide more data-level parallelism and keep threads from becoming idle. These optimizations must be tuned in order to be fully optimized. 2.5D blocking takes two parameters: an x-axis blocking dimension and a y-axis blocking dimension. For a 256^3 grid, there are 60 different configurations of 2.5D blocking. For loop unrolling, the maximum unroll length allowed by the compiler is 192 iterations. By combining these two search spaces, the genetic algorithm used for tuning these optimizations searches through a search space containing 11,520 different configurations.

C. Genetic Algorithms

Genetic algorithms are a set of algorithms that mimic the natural selection process in order to find solutions to problems. Genetic algorithms do this by generating an initial population that generally consists of randomly generated individuals that contain randomly generated values for each parameter that will be searched. Each individual in the population then undergoes a selection process by which the fitness of their parameters that they contain is evaluated. The most fit individuals are then selected to be mutated and mated with each other in order to generate the next generation of individuals. This then continues until the population either converges to a singular value or the number of set generations is reached.

In this research, we used an initial population of ten individuals each with a chromosome (parameter set) containing three parameters – thread blocking for the x and y axes and loop unrolling factor. This population then underwent ten generations in order to get the individuals to converge on one value. The best performing individual was saved and then returned at the end of the generation process as the best configuration for the given optimizations. All of this was done using the Distributed Evolutionary Algorithms in Python (DEAP) project [4]. It allowed for use of built-in algorithms for the mating, mutating, and selection processes.

IV. Experimental Results

A. Goal

The goal of this experiment is to determine if genetic algorithms are a viable approach to tuning stencil code optimizations faster than other methods of tuning. The genetic algorithm used must be able to produce an optimal or near-optimal configuration for the optimized stencil code in a reasonable amount of time in contrast to the time it takes for an exhaustive search method to find the optimal configuration.

B. Setup

The setup we used to perform the experiments on consisted of three GPUs: one GPGPU (Tesla C2050) and two standard

GPUs (GTX 480, 680). Figure 3 details the theoretical peak FLoating-point OPeration (FLOP) rate determined by the number of cores (α) multiplied by the clock rate of each core (δ) multiplied by the number of FLOPs that can be performed each clock cycle (γ).

$$\alpha \times \delta \times \gamma = GFLOPS/sec$$

Fig. 3. Theoretical peak FLOP rate equation.

GPU	Architecture	Peak FLOP rate
GTX 480	Fermi	1344 GFLOP/s
Tesla C2050	Fermi	1030 GFLOP/s
GTX 680	Kepler	3250 GFLOP/s

Fig. 4. Peak GFLOP rate of GPUs (single precision)

The genetic algorithm was run on two stencil kernels: a 27 point Jacobi stencil and a 7 point Jacobi stencil. This genetic algorithm was used on each of the GPUs and was trained on the GTX 480 using the 7 point stencil. After the initial training, no values of the genetic algorithm were changed in order to generate the final results. The genetic algorithm used a three-gene chromosome to find configurations. The first two genes were for thread blocking dimensions along the x and y axes each being a power of two and their combined product could not exceed 2^{10} (60 combinations for 256^3 grid size). The third gene was for loop unrolling which was an integer from 1-192 for unroll length. The combined search space consisted of 11,520 different combinations the algorithm could possibly generate. This genetic algorithm was then run to create ten generations based on an initial population of ten individuals in order to find a configuration for each optimized stencil code that was close to the optimal value that was found through an exhaustive search of the search space.

$$a^{n+1}_{i,j,k} = \alpha(a^n_{i,j,k} + a^n_{i\pm1,j,k} + a^n_{i,j\pm1,k} + a^n_{i,j,k\pm1})$$

Fig. 5. 7-point Jacobi stencil equation.

$$a^{n+1}_{i,j,l} = \alpha(a^n_{i,j,k}) + \beta(a^n_{i\pm1,j,k} + a^n_{i,j\pm1,k} + a^n_{i,j,k\pm1}) + \gamma(a^n_{i\pm1,j\pm1,k} + a^n_{i,j\pm1,k\pm1} + a^n_{i\pm1,j,k\pm1}) + \epsilon(a^n_{i\pm1,j\pm1,k\pm1})$$

Fig. 6. 27-point Jacobi stencil equation.

The equations in figures 5 and 6 detail a typical 7-point and 27-point Jacobi stencil where a is the input grid, n is the iteration, and $\alpha, \beta, \gamma, \epsilon$ are coefficients multiplied upon the neighborhood sum. The ±1 symbols are used to save space in writing out each $i+1$ and $i-1$ for each i, j, k within the array of nodes.

C. Results

The graphs in Figure 7 are of the average performance in GFLOPS/sec of the population per generation. The red line is of the performance of the 7 point stencil code and the blue line is of the 27 point stencil code. The two dashed lines in each graph show the optimal configuration performance for each stencil code. The optimal configuration was found by performing an exhaustive search through the parameter search space.

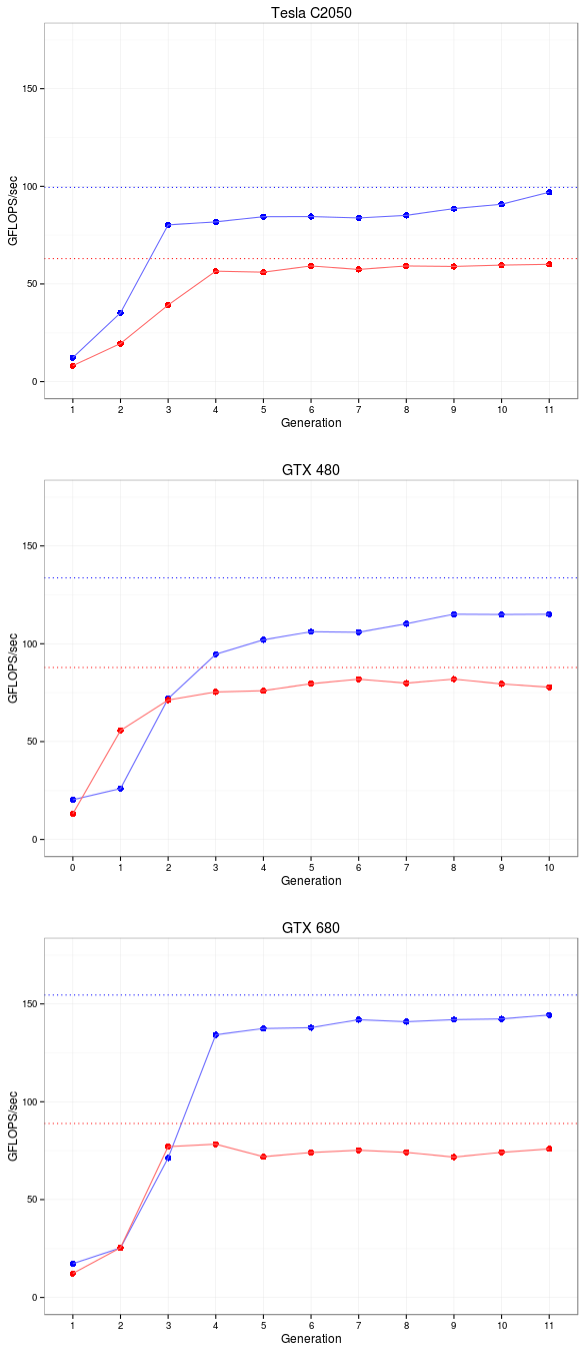

Fig. 7. Genetic algorithm average fitness of each generation for the three GPUs on both stencil kernels. The solid lines are for the average population fitness by generation for the 27-point stencil (blue) and 7-point stencil (red). The dashed lines show the optimal configuration throughput rate.

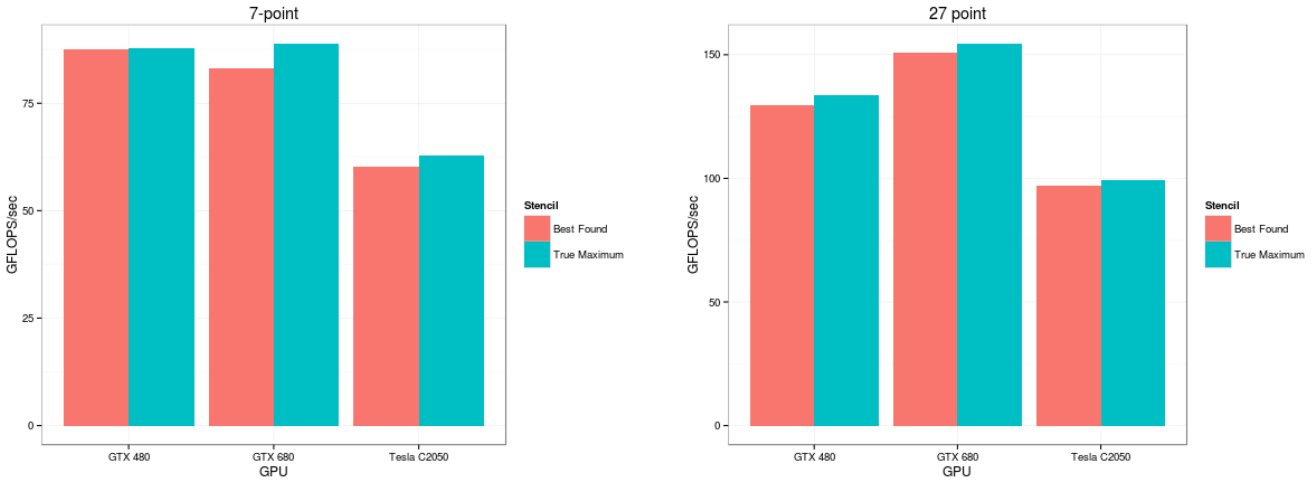

Fig. 8. Best configurations found by genetic algorithm vs the absolute best configuration found by exhaustive search for each stencil and GPU.

These results show the effectiveness of a genetic algorithm approach to auto-tuning stencil code optimizations as it generally only took 3–4 generations for each stencil code to be near-optimal. It should also be noted that these results only show the average performance of the entire population per generation, not the best candidates. The best candidates shown in Figure 8 of the population were typically within 3% of the optimal performance found for each stencil kernel by the 10^{th} population. The initial population for the genetic algorithm consisted of only ten members. Due to the small search space size, this small number of members was still able to quickly converge to a near-optimal configuration for each kernel. The small search size also allowed for us to check our results through exhaustive search as doing so took about 4–5 hours per kernel for each GPU. This speed is in contrast to the average eight minute execution time for the genetic algorithm to generate all ten generations and find a near-optimal configuration. These speeds differ in terms of which CPU is used to compile each code, but the large gap in performance still persists for each CPU, regardless of its speed.

However, these results show that each kernel could only reach up to a maximum of 100 GFLOPS/sec for the 27 point stencil on the Tesla C2050, which is far lower than the 450 GFLOPS/sec produced by Bergstra et all [2] on the same model of GPU. This is due to the optimizations that were used in our stencil codes as they are the main bottleneck of performance for the stencil code. Our optimizations still contain thread divergence in the code, and is thus less optimized compared to Bergstra et all's kernel which contains no thread divergence. For future work on this research, more optimizations will be considered so that the results will be closer to current stencil code performance.

V. Conclusions and Future Work

This work demonstrates the effectiveness of using a genetic algorithm in order to find near-optimal configurations for stencil code optimizations across multiple GPUs with differing architectures. This result allows for enhanced portability of stencil code optimizations to differing architectures in a timely fashion as the tuning phase was demonstrated to be much faster than exhaustive search alternatives as the genetic algorithm took, on average, eight minutes to generate all ten generations of the population in contrast to the 4–5 hour run time of the exhaustive search.

In the future, we would incorporate more parameters for use in the chromosome for the genetic algorithm in order to generate a search space worthy of using a machine learning technique to traverse it instead of exhaustive search being a viable method to use. We will also develop more parts to the auto-optimization framework from figure 1 such as the optimizer and stencil code classifier. This is in the hopes that a functional framework can be created such that it may be used to optimize existing stencil codes that are in use today and be continually optimized as more stencil code optimizations are found.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. 2012.

[3] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[5] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar09)*, 2009.

[6] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[7] Julien Jaeger and Denis Barthou. Automatic efficient data layout for multithreaded stencil codes on cpu sand gpus. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[8] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, page 256265. ACM, 2009.

[9] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[10] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM.

ACKNOWLEDGEMENTS

This research is supported by NSF grant 1359275.

RSSE: A New Method of Distributing Datasets and Machine Learning Software

Michael Gohde
Vision and Security Technology Lab
University of Colorado at Colorado Springs
Colorado Springs, Colorado
mgohde@uccs.edu

***Abstract*—Among the challenges faced by machine learning researchers today is that of distributing the datasets and algorithms used in their research. This problem arises mostly from the limitations involved in hosting datasets on servers outside of their origin. RSSE (Really Simple Syndication for Experiments) is intended to provide a message-based system with which researchers can share their data and algorithms.**

I. Introduction

RSSE draws on existing standards, namely XML and RSS, to facilitate easier communication and distribution of data among researchers. While RSSE is not an extension to the existing RSS standard[6], it is intended to be similar in general conventions and syntax to RSS. As such, it extends the concept of RSS, which is that of message-based syndication involving a client "reader" application and a message server established by an institution. RSSE is designed so that messages can be written either by researchers themselves or by automated tools.

Due to current copyright and IP law, it is often difficult or impossible for institutions and researchers to directly distribute external datasets used during computation[4]. Some large dataset providers, such as Yahoo, explicitly prohibit the redistribution of the dataset itself, instead allowing the dataset to be distributed in the form of links[5]. Such datasets usually allow users to cache the data locally. By providing a consistent system by which data and software can be distributed using messages containing URLs and checksums, RSSE should enable institutions to easily monitor and expand on the work supplied by other institutions. Furthermore, by passing URLs to datasets rather than the datasets themselves, experiments can be run by other institutions while still respecting the Intellectual Property rights of the dataset's source.

RSSE will work in a similar fashion to RSS (Really Simple Syndication), with some exceptions. As such, a researcher or automated utility would generate an XML file using RSSE tags and serve it over the HyperText Transfer Protocol (HTTP). Such an XML file would contain the project's title, a brief description of the project itself or changes made recently, several URLs, and checksums for the relevant URLs. Each URL should usually refer to a datasets involved during the course of research. One possibility, however, is that some of the URLs could refer to source code or compiled Java classes, which would, in turn, be executed locally to verify the results of the computation. When an XML file containing RSSE data is posted, client programs could proceed to download the file, parse its contents, then perform a predetermined set of actions, such as downloading all of the datasets and source code involved in the remote experiment. For a graphical representation of the data transfers involved, see figure 1.
(figure 1)

II. Previous Work

RSSE draws primarily off of the existing RSS standard[6]. Due to its flexible nature and widespread use, RSS has already utilized as a basis for distributing information to research librarians in an organized fashion[2]. A similar system featuring a dedicated message and client-based infastructure was implemented to distribute climactic data, however it did not explicitly use RSS, rather it directly exposed a database to a network[3].

III. Implementation

A. Implementation History

For this project, a reference implementation of the RSSE reader and file generation utility were written. As RSSE will be an open standard, the implementation discussed here exists solely as a reference for other future implementers to follow.

The first stage of the implementation was considering what tags would be acceptable for each RSSE file. These tags are listed in Table 1. The tags were determined after considering the minimum set of data necessary to represent a message. The most important tags involved are the dataset tag, the checksum tag, and the checksumtype tag. The dataset and checksum tags are self explanatory. The checksumtype tag will contain a string value representing the hashing algorithm used to generate the checksum on the server's side.

The second stage was the implementation of the RSSE reader program. This application was implemented before the RSSE message generation program because of the relative ease in manually writing RSSE files as opposed to manually reading RSSE files. Upon starting, a command line specification was drawn up based on all of the tags and operations expected of

the program. The command line interface was implemented first, due to the ease in doing so. Command line options are not included here as it should only be necessary for future implementations to utilize the base set of RSSE tags as opposed to implementing full compatibility. This was followed by the implementation of a GUI, which extended the features of the command line interface.

Once the graphical component of the RSSE Reader was complete, work started on the RSSE Generator. Because all of the features of RSSE were fairly stable by this point, the RSSE generator was far easier to implement. Unfortunately, due to time constraints, the generator does not yet have scripting support, however that has become a priority in the near future.

While each program is currently stable enough for general use, there are some UI elements that do need improvement due to their obtuse or erratic behavior. The most obvious of which is the difference in graphical styles between the reader and file generation utility.

Upon completing the first few revisions of the RSSE reference implementation, the project was demonstrated to a group of machine learning researchers. Based on their suggestions, the RSSE version 0.03 specification was drawn up with several enhancements in the form of four new tags and three new sets of attributes. These new tags should enable both researchers and end users to benefit from increased tailoring to various conditions and systems. The new features are mentioned in separate tables. Upon starting work on this version of the specification, it became very clear that each version of RSSE would in the future cause rendering and generation problems on prior and future versions of the RSSE reference implementation. Because of this, the <rsse> tag now carries an attribute specifying the expected minimum version code necessary to render a given RSSE message. The version encoding scheme is elaborated in Table IV.

B. Reasons For Using The Technologies Used

While doing the project, it became clear that it may be necessary at some point to justify the use of Java and XML as the primary language and data framework of the application, respectively.

Firstly, Java was selected as the primary implementation language due to its feature-set. Java has extensive support for reading and parsing XML files, which proved invaluable for the project as a whole. Furthermore, Java provides easy to use networking and graphical user interface APIs, which contributed to the quick implementation of the project. Finally, the project's code should be easily readable to a wide array of programmers due to the similarities in syntax and usage between Java and other programming languages.

XML was selected mostly because it is very easy for humans and computers alike to read and parse. By using plain-text instead of binary for messages, it allowed the developer to write test messages to pass to the reader before the file generation utility was complete. Furthermore, while it was not a clear focus in the beginning of the project, XML allows for a significant level of complexity and flexibility, which allows the RSSE standard to expand easily in the future. In the future, it is likely that future implementers will write their own messages in order to test their RSSE reader applications.

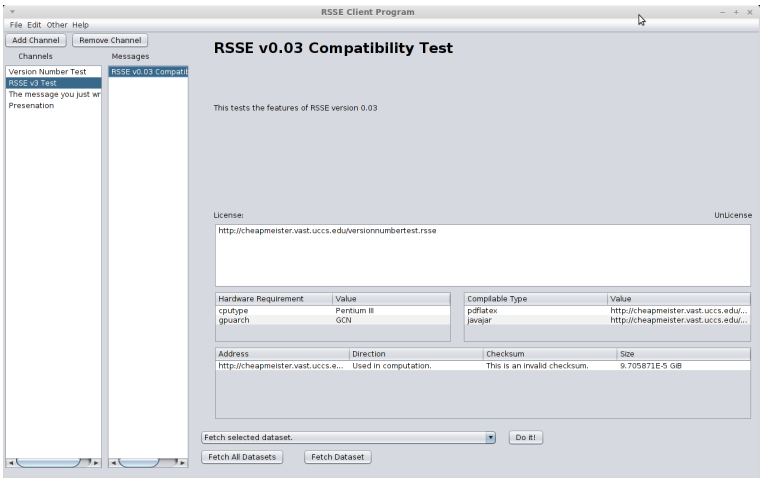

Fig. 1. The RSSE Reader Program

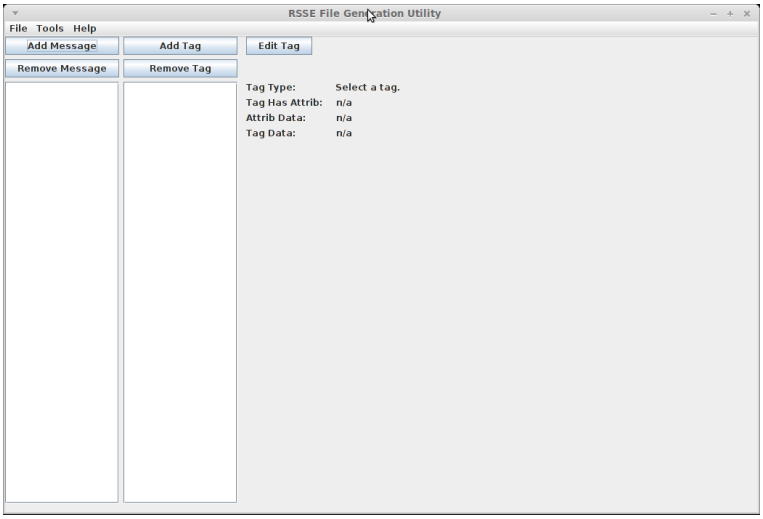

Fig. 2. The RSSE File Generation Program

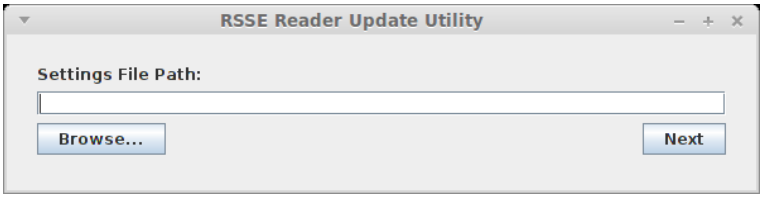

Fig. 3. An early build of the RSSE Updater

C. Interface and Design

While the graphical user interface is in a very early development stage, it is complete enough to be shown here. Please refer to Fig. 1 for an example configuration of the Reader, and Figure 2 for an example configuration of the Generator. In both programs, there are clearly defined lists of values to be modified and modifier buttons either on top of the lists or to

their side. This was done to associate the various modifiers with the data involved, which should hopefully lend itself to usability. The menu layout is very sparse, as most of the functions encountered in the program are represented by the buttons present. There are very few dynamic UI elements in order to promote portability to low-power devices and older operating systems. By providing unambiguous functions, the RSSE reference applications should be very easy to learn. Overall, this design aesthetic should prove helpful for the purposes of providing a reference implementation from which other implementations of the RSSE standard can be derived.

Tag	Tag Value
<rsse>	Tag used to denote an RSSE file.
<message>	Tag used to mark the start of a message. This allows for there to be more than one message in each file.
<title>	The title of the project.
<description>	The project's description.
<link>	A website to be visited by the user. Could be used to direct clients to more information.
<dataset direction="in">	Represents a dataset. The direction attribute is used to inform the user as to whether the dataset was used in computation (value "in") or generated as the result of computation (value "out").
<checksumtype>	Represents the type of checksum to generate and check.
<checksum>	Represents an individual checksum. Will be associated in the order of appearance of datasets.

TABLE I
TAGS IMPLEMENTED BY THE RSSE REFERENCE SOFTWARE

Tag	Tag Value
<update url="url">	Tag that could be used to send updates to the reader. Url attribute is used to mark the URL of the update. The tag will contain the
<license>	Tag that could be used to distribute executable code if implemented.
<minspec>	Tag representing the minimum specifications to run software bundled with a message.
<compilable type="type">	Tag that could be used to distribute code with special compilation requirements.

TABLE II
TAGS ADDED IN RSSE VERSION 0.03

Attribute	Tag	Description
pdflatex	<compilable>	Allows for the inclusion of LaTeX documents.
makefile	<compilable>	Allows for the distribution of projects utilizing makefiles.
javajar	<compilable>	Allows for the distribution of Java Jar files.
executable	<compilable>	Allows for the direct distribution of executable files. The reference implementation will never allow these files to execute without a prompt.
gpuarch	<minspec>	Allows for researchers to specify different GPU architectures, such as Kepler or GCN.
cpuarch	<minspec>	Allows for researcers to specify a CPU architecture for specific optimizations.
cputype	<minspec>	Allows for researchers to specify a specific type of CPU to be used. Only for heavy optimizations.
corecount	<minspec>	Allows for researchers to specify a minimum number of cores to comfortably run multithreaded software.
minram	<minspec>	Allows for researchers to specify a minimum amount of RAM as a floating point number of gigabytes.
minstorage	<minspec>	Allows for researchers to specify a minimum amount of free hard drive space as a floating point number of gigabytes.
osfamily	<minspec>	Allows for researchers to specity the intended operating system family for their software. This could be used to prevent non-POSIX operating systems from attempting to compile the software included.

TABLE III
ADDITIONAL ATTRIBUTES IMPLEMENTED IN VERSION 0.03

RSSE version	RSSE Tag
v0.01 (Depreciated)	<rsse>
v0.02	<rsse>
v0.03	<rsse version="3">
(Future releases)	<rsse version="(Version number*100)">

TABLE IV
THE RSSE VERSION ENCODING SCHEME.

IV. CHALLENGES

While writing the checksumming portion of the program, it became very clear that Java's default IO functions were too slow for the task. While it has not yet been implemented, the project will eventually add support for Java's NIO (Non-blocking IO)[1], which should provide a high performance framework for checksumming operations. Another challenge encountered was that of determining how checksums should be transmitted, as it is difficult to parse multiple attributes in each tag. This problem was solved by associating each checksum tag with dataset tags in the order that they appeared, however this is more of a short-term solution that may require the implementation of a complete XML parser within the code. One of the clearest challenges is deciding on which tags should be added to each version of the RSSE specification, as it involves several decisions as to which features would be easiest to implement, as well as which features would be best for end-users. Overall, accepting suggestions from others proved to be very beneficial to the project as a whole.

V. APPLICATIONS

RSSE has the potential to become a very valuable tool for researchers, especially those who wish to use commercial or otherwise difficult to distribute datasets. While RSSE is intended to be used primarily by a machine learning and computer vision audience, it has the potential to be used for scientific research, namely in peer-review. As such, RSSE need not be constrained to just distributing datasets. It has the potential to distribute papers, code, or even precompiled binaries to remote computers for independent verification of results. While it is not the intent of the project, it can be used to assist with distributed computing with only minimal modifications.

Feature or Tag	Information
Automatic Updates	The reference RSSE implementation currently includes a very basic update utility. In the future this utility will be improved and expanded.
Compilation and Execution Support	In its current state, the RSSE reader can only download executable or compilable objects from remote servers. Future revisions will allow for proper compilation and execution of RSSE messages.
System Requirement Checking	The RSSE reader is currently incapable of checking system requirements. In the future, support for this will be added.
Merge Checksum and Dataset	Merging the dataset tag and the checksum tag would streamline the distribution of datasets.
Improving Checksum Performance	Checksumming in the file generation program is unacceptably slow.
Implementing more Minspec Tags	The minspec tag list is currently somewhat incomplete.
Implementing Local Caching	One of the long-term goals of this project is to develop a caching system to avoid several of the problems in distributing datasets.
Implementing the RSSE Executor	RSSE will be functionally divided between the Manager (what the reader is now) and the Executor, which will fetch cached data and try to process it.
Splitting the Reader	The current RSSE Reader application is currently insufficient for caching and the constant update cycle needed by real-world researchers. As such, it will be split into two programs: The RSSE Reader and the RSSE Manager. The RSSE Reader will act as a graphical configuration utility for the RSSE Generator. As such, most of the features of the Reader will be merged into the Manager.
Updating Scripting Support	The RSSe file generator has great potential to be scripted, especially in providing automatic rapid updates to end users.

TABLE V
FEATURES TO BE IMPLEMENTED IN FUTURE RELEASES OF RSSE

Some clear distinctions need to be drawn between the RSSE project and RSS, however. Its focus on academic pursuits should be preserved and remain a primary goal. As such, other applications, such as distributing newsfeeds or non-research related data should be discouraged to avoid feature bloat. Such feature bloat would make implementing additional readers and file generators significantly more difficult than the standard is designed to allow. However, other implementations should be encouraged to deviate somewhat so that additional features can later be brought into the mainstream RSSE specification.

VI. Summary and Conclusions

Provided that the RSSE project becomes widely adopted, it will provide a clean, easy to use, and rapid means by which researchers can share data. It has been designed with a clear emphasis on having low barriers to entry. These low barriers to entry should allow RSSE to become a *de facto* standard in research and communication. Given such status, other implementations of the reader and file generation programs would likely be written with more features than could be implemented here. Such implementations can be expected to include features specific to various fields, such as some peer review system for scientific research.

VII. Acknowledgements

I would like to thank Dr. Terrance Boult for proposing the original idea for RSSE, as well as the machine learning researchers who demonstrated a need for this software. I would like to thank the NSF for providing funding for the REU research program.

References

[1] File i/o (featuring nio.2). http://docs.oracle.com/javase/tutorial/essential/io/fileio.html. Accessed: 2014-07-03.

[2] Alexia D. Estabrook and David L. Rothma. Applications of rss in health sciences libraries. *Medical Reference Services Quarterly*, 26(sup1):51–68, 2007. PMID: 17210549.

[3] Hannes Grobe, Michael Diepenbroek, Nicolas Dittert, Manfred Reinke, and Rainer Sieger. Archiving and distributing earth-science data with the pangaea information system. In DieterKarl Ftterer, Detlef Damaske, Georg Kleinschmidt, Hubert Miller, and Franz Tessensohn, editors, *Antarctica*, pages 403–406. Springer Berlin Heidelberg, 2006.

[4] Gerald Schaefer and Michal Stich. Ucid: an uncompressed color image database, 2003.

[5] David A. Shamma. News: One hundred million creative commons flickr images for research. http://labs.yahoo.com/news/yfcc100m/. Accessed: 2014-07-03.

[6] UserLand Software. Rss 2.0 specification, 2002.

Learning Patterns of Mobile Interface Design

George C. GUVERNATOR V
The College of William and Mary
Williamsburg, Virginia
gcguvernator@email.wm.edu

***Abstract*—Nothing for now. We'll write our abstract last.**

I. INTRODUCTION

Interface design is a crucial element in any software project. Graphical interfaces allow for more intuitive human-computer interaction but can challenge even the most skilled developers as they take on the additional responsibilities of a designer. In the world of mobile applications, where many competing implementations of an idea are available to users, design can play a key role in a user's choice of application. Additionally, limited and varying screen sizes, touch-based interfaces, and limited resources all challenge the mobile interface designer. It is our observation that, unfortunately, some developers fail to spend ample time designing Graphical User Interfaces (GUIs). This is especially true in academia, where many of the most technically correct and well-implemented software projects falter in this area, considering GUI design an overly costly afterthought. In mobile applications, this can mean that not all screen sizes, input methods, accessibility features, and other mobile-exclusive considerations are accounted for. As a result, developers lose many potential users from otherwise well-written and well-executed projects.

Mobile application development presents unique challenges beyond those faced when developing software with more traditional keyboard and mouse interfaces. On mobile platforms, user interface design poses the unique challenge of restricted screen space with respect to more conventional desktop or console platforms and therefore has a greater influence on the overall usability of the application. Additionally, developers must keep in mind the variety of devices of different size that their applications will run on. An interface built for a 15 by 10 cm tablet, for example, may not scale well to a 9 by 5 cm smartphone. Elements designed to be tapped on the larger screen by human fingers or styli would become more difficult to accurately activate on the smaller screen. Another complication to developers are extensions to the platform's standard user interface, such as those used for accessibility or universal access by users with physical disabilities. Finally, fragmentation on the Android platform causes design and development bugs, both from devices running different versions of the Android platform which support different graphical layout components, and from a variety of vendors building their devices differently. The problem of Android device and version fragmentation are discussed by Han et. al. [1] and Degusta [2].

This research funded by a grant from the National Science Foundation (1359275).

Facing these additional challenges of mobile development, it is therefore important to understand both how mobile applications are designed and which identifiable design patterns users prefer. The former can be accomplished given access to an application's source code, and the latter is theoretically possible by scraping user ratings (both long-form text reviews and one- to five-star numeric ratings) from Google Play, the officially supported Android application repository. Given the unique challenges of the mobile environment discussed above, we assert that design is a significant factor reviewers consider when rating an application. While many reviews center around program functionality and stability, we believe there is enough weight placed on design to show clear trends and allow for correlative measurement.

In this research, we analyze the GUI design of a variety of Android applications, allowing us to gather data and create a characteristic model. We evaluate correlations between the gathered design data and reported user experiences, such as comments and ratings, as well as other potentially confounding variables found in the application's metadata from the application repository.

To accomplish this, we scrape Android source code from the F-Droid Web repository[1] using a tool we developed called *fdscrape*.[2] Package names found with the source code on F-Droid are also searched on Google Play, a non-free Android application repository, and metadata such as user ratings, comments, popularity, and size are scraped and associated with each application. The combined source code and metadata, collectively the *F-Droid corpus*, is run through a program we have developed called AGUILLE.[3] This tool analyzes the structure of GUI markup language in the source code and extracts and counts the individual elements used to construct the interface, analyzing and combining data points to prepare for machine learning analysis.

Finally, the analysis of AGUILLE and metadata found with fdscrape are combined in a machine learning workflow in Weka. The workflow leverages the power of the M5 model tree to generate explainable branches and decision points that are applied in an ordered hierarchical structure to determine a potential rating for future applications. This is improved by analyzing each application category separately and building separate models for those categories with enough applications to make valid predictions. This categorical discretization is a

[1] F-Droid can be accessed on the Web at `https://f-droid.org/`.

[2] Fdscrape is licensed under the GNU General Public License (version 3) and is available on the Web at `https://github.com/qguv/fdscrape`.

[3] AGUILLE is licensed by the GNU General Public License (Version 3) and is available on the Web at `https://github.com/qguv/aguille`.

key part of our experiment, see section IV on the following page.

A summary of specific results should go here. It will mirror the summary that will end up in the conclusion.

Further development could turn the predictive model into a suggestive one. The models generated by the framework described in this research could be a crucial addition to "Interface Builders," [3] graphical applications designed to help developers create graphical interfaces. Developers would have a new, powerful tool suggesting subtle changes to their design in order to better emulate the most popular and successful graphical interfaces available today.

The main contributions of this research are as follows:

- Development of a framework of tools to gather layout information of Android applications (section III)
- An empirical study correlating Android GUI design patterns and the reported quality user experiences (section IV on the following page)
- Analysis of recurring design patterns and trends in Android GUI design (section V on page 4)
- Development of a predictive model for mobile GUI design (section IV-B on page 4)

II. Background

This section will be expanded to better explain where our research fits in the field of related work discussed in section VI on page 5.

Research on learning design patterns [4], [5] proves useful when designing a predictive machine learning workflow. Both Neural Networks and vanilla Decision Trees are discussed and implemented in [5].

We began by correlating specific features and ratings manually using straightforward linear regression in order to test different weightings of features. Next, random forests were used to gain insight on the sorts of decision points that regression and model trees produce. We eventually settled on to the M5 model tree to firm up final results and to allow trends to be explained and described in human-friendly decision points rather than difficult-to-describe coëfficient models or more opaque random models.

A. Choosing Android

Though the concepts presented in this paper are applicable to any graphical environment, we have chosen to work with the Android mobile platform due to both the unique challenges of a mobile environment discussed in section I on the preceding page and the uniformity and availability of application source code.

We feel the concepts presented in this paper would be most advantageous to mobile developers, as user ratings, our evaluative metric, can directly influence an application's ultimate success or failure. Android users must often choose between similar implementations of the same tool. The Google Play store in turn provides a system with which users can rate applications and post feedback for developers and other potential users. These user ratings help Android users to narrow down their choices in a vastly competitive market.

Android is also ubiquitous among mobile device users. The popularity of the platform continues to grow as new users and developers adopt Android as their primary mobile platform. With over 1.3 million[4] Android applications on the Google Play store at time of writing, the popularity of the Android platform has provided us and will continue to provide other research teams with ample data to search for significant correlations and generalizations.

Finally, the somewhat constrained GUI design framework in the Android platform (Android XML) allows for relatively straightforward parsing of the application's graphical layout without needing to peek into the application's logic.

Because of these unique properties of the Android platform, we believe mobile applications will benefit most from the preliminary results presented in this paper.

III. Extracting Design Elements

Before we can begin evaluating and correlating patterns, we must first collect information on Android GUI design. Since Android developers define graphical layouts in source code, we decided to gather and interpret source code in order to gather data about Android GUIs. We chose the free and open-source Android software repository F-Droid as a source for Android source code.

After gathering source data, we must extract the parts of the source code pertaining to graphical layouts. We then analyze those layouts to determine what built-in graphical elements the developer chose to use in designing the application and in what quantity and proportion.

After all layouts of all available applications in the repository have been analyzed, the results are fed to a machine learning algorithm to make generalizations about which elements affect others, which best predict ratings, and which carry little meaning in the context of this study.

Finally, the performance of this machine learning system is analyzed, and the workflow is tweaked to attempt to improve prediction and correlation both between elements and against user ratings.

A. Dataset Acquisition with fdscrape

We have developed a program (in Python) to enable mass retrieval of Android source code to mine. We use F-Droid, a software repository containing binaries and source code for 1,145 free and open-source Android applications. The majority of these applications are also available on the official Android application repository, the Google Play store.[5] Because of this, we have downloaded all available applications and their source code from F-Droid as well as Google Play ratings and metadata for the same applications, storing the data for analysis in the machine learning step.[6] This data is stored with the source code of each application.

[4] According to *Appbrain Stats*, a Google Play metrics service. Visit `http://www.appbrain.com/stats/number-of-android-apps` for the latest statistic.

[5] The Google Play store can be accessed on the Web at `https://play.google.com/store`.

[6] To accomplish this, we have cross-checked Java package names against both F-Droid and the Google Play store.

We originally chose to scrape only rating information from Google Play. It was decided, however, that by saving more of the metadata provided by Google Play and developers, better predictions could be made by accounting for variables beyond the scope of design. This permits meta-analysis of our hypothesis, i.e. we can decide how much design affects ratings and how much predictive accuracy to expect from our model. We gather this data in order to compensate for any confounding correlation that Google Play metadata may have on determining rating.

Specifically, the developer-chosen application category (e.g. Weather Application, Productivity Application, Action Game, Puzzle Game, and others in table I) provides an effective way to analyze groups of applications at a time. It is our hypothesis in RQ 1 that separate analysis within application categories will yield more meaningful results. This has the potential to greatly improve the accuracy of our predictive algorithm and allows us to better understand what "good design" entails in certain domains. For example, the same elements that constitute good design for an action game might exemplify bad design for a news application.

After omitting applications that were not on the Google Play store, had no ratings, or did not host source on the main F-Droid website, we collected the source code of 894 applications to build our dataset.

B. Tag Lexing & Extraction with AGUILLE

We have developed AGUILLE, the Android Graphical User Interface Lazy LExer, to perform the Android source analysis we originally hoped GUITAR would accomplish. The tool takes in an application's source code, finds the relevant Android XML structure, and parses that structure into native Python objects. Using these objects, AGUILLE calculates the frequency with which each XML tag, or element, occurs in the application. The graphical design of the application, therefore, is reflected in the developer's choice of graphical elements.

These frequencies are collected in a CSV file, along with the metadata gathered with fdscrape. Lots of the scraped information can be cached to speed up parses of entire repositories.

The tool is designed such that, should more sophisticated calculations prove necessary, separate sub-commands could easily be added. AGUILLE is open-source; anyone may extend it by adding further subcommands or modifying its current behavior.

C. Machine Learning with Weka

We make use of the Weka 3.7 *Knowledge Flow* environment to create a machine learning workflow. Data from AGUILLE is loaded separately by category. We drop categories which contain less than one percent of all applications mined, leaving the 20 categories described in table I.

	Applications	
Category Name	Number	Percent
'Tools'	278	33.3%
'Productivity'	88	10.5%
'Communication'	67	8.0%
'Personalization'	34	4.1%
'Books and Reference'	32	3.8%
'Game Puzzle'	30	3.6%
'Education'	29	3.5%
'Media and Video'	29	3.5%
'Music and Audio'	25	3.0%
'Entertainment'	24	2.9%
'Transportation'	18	2.2%
'Travel and Local'	18	2.2%
'Finance'	17	2.0%
'Game Arcade'	17	2.0%
'Health and Fitness'	17	2.0%
'Lifestyle'	15	1.8%
'News and Magazines'	15	1.8%
'Social'	15	1.8%
'Photography'	13	1.6%
'Libraries and Demo'	11	1.3%
Total: 20 categories	792	94.7%

TABLE I. APPLICATIONS IN EACH MINED CATEGORY

IV. EMPIRICAL EVALUATION

The design of our experiment is such that two chief research questions (RQs) may be addressed:

1) What sort of design do applications have in common? What sort of trends emerge when analyzing entire repositories of applications?
2) Does separate analysis of applications by Google Play category improve the quality of our predictive algorithm? Does such separation give more meaningful generalizations when explaining the output of our decision tree?

Our goal in evaluating the accuracy and quality of our predictions is twofold: to attempt to improve the ability of our algorithm to predict user ratings and to make conclusions how about individual elements affect user perception of an application.

When evaluating the performance of our machine learning workflow, we are looking for statistically significant correlations with performance better than the 5–10% correlation we see with naïve sample algorithms.

We cannot expect anything near perfect prediction, as more goes into a user's choice of rating than design. We must therefore focus on explainable output, such as that from a decision tree, to try to explain the influence of design on ratings.

A. Experiment Design

To be able to learn which design elements lead to the best applications, we need a group of factors, or *heuristic*,

to evaluate, as well as a metric for determining what constitutes a "good" application. This study uses the frequency of use of Android XML tags in graphical layout source code as a heuristic. Because the Android framework comes with a rigidly-defined set of elements, XML tag (and therefore element) frequency allows us to point to very specific design choices to explain findings when learning correlations.

Android also allows developers to define their own tag elements, but because scraping these developer-defined, application-specific tags and properties involves parsing Java logic and rendering the elements, this added complexity is somewhat beyond the scope of this project. The application-specific nature of these tags also means that they will most likely not be of use when attempting to make correlations between applications.

Although it may be possible to learn a more complex heuristic over time, it would increase overhead and likely introduce excessive complexity into our system. At this stage of research, it is wiser to rely on the pre-designed elements available to all Android applications to potentially determine the quality of GUI design. We have found that element frequency is sufficient to establish a statistically significant correlation between the elements themselves and the evaluative metric, Google Play ratings.

Potential alternative evaluative metrics could include uninstall rate and frequency, certain statistical functions on the cumulative body of Google Play user ratings, or cross-referenced reviews from established news sources. Google Play ratings were trivial to scrape and analyze, as the data is publicly available, so these public ratings serve as an initial metric we use to establish the overall quality of an application. In our algorithm, we can weigh the rating's relevance depending on how many users rated the application, a metric we also obtained from the Google Play store. We might be more confident in a metric to which many users contributed.

In early models, we found that the M5 model tree first branched based on the amount of user ratings on Google Play. The sub-trees after this split did not clearly resemble each other; the branch reached by applications with few ratings gave a counter-intuitive model, while the branch for applications with a more substantial amount of ratings weighted elements as we would expect. This suggests that the model's predictive accuracy increases substantially when more rating data is available, as we would expect.

Our group of independent variables, the frequency of each Android graphical element, will reflect the different proportions of interactive Android *widgets* with respect to each other. These interactive widgets are built-in to the Android platform and include buttons, check-boxes, radio buttons, images, and text. Additionally, these widgets can appear in different *views*, all of which have different ways to specify how the widgets will be laid out on the screen. All of these views and widgets are part of our element count.

Of course, GUI design is hardly the only factor users consider when rating a program. It is important to consider major confounding variables (viz. quality of functionality, stability, the ability of the program to solve a real problem) and integrate Google Play's qualitative long-form paragraph review system. A naïve but effective way to acknowledge applications with known performance issues (and therefore identify those applications whose low ratings have little to do with design) involves searching for key terms occurring abnormally frequently in text reviews. See table II for a list of key words and phrases that could indicate poor performance rather than poor design. Such key words and phrases are likely to indicate that low ratings are due to factors outside the realm of interface design. After gathering other metadata, fdscrape counts the frequency of these key words and phrases with respect to the available body of reviews. The calculated frequencies of these words are fed into the machine learning algorithm along with the tag frequency count from AGUILLE and the metadata from fdscrape.

If our algorithm were to put significant weight on these terms rather than the intended features in our heuristic, we can safely chalk these up to poor application performance. This gives a decision tree the option to discard obviously poor-quality applications in an early decision node in order to focus on the design factors we are interested in analyzing. If a more sophisticated method than calculating tag frequency proves necessary, we could take a naïve Baysean approach, analyzing the probability rather than the frequency of key phrases in known or exemplary good and bad applications.

By acknowledging applications which have known issues unrelated to design, observed ratings of the remaining applications in our dataset will better reflect design quality.

Key Phrase	Possible Conclusion
'incompatible'	Could indicate versioning or device compatibility issues for certain users.
'uninstall'	Could indicate frustration with the application or the inability for certain users to un-install pre-installed software.
'crash'	Could indicate stability issues.
'slow', 'lag'	Could indicate resource overloading, frequent Internet requests, or poor data structure implementation.
'black screen', 'white screen', 'blank screen'	Could indicate initialization problems.

TABLE II. KEY WORDS AND PHRASES SUGGESTING POOR PERFORMANCE

B. Experiment Results

Results would go here, once we decide what output of which machine learning workflows to include.

V. DISCUSSION & CHALLENGES

The single most significant setback to this project has been the failure of the Android fork of the GUITAR tool. We have been forced to develop an in-house tool from scratch in its place. It has taken weeks to develop AGUILLE to a usable and dependable state. While new developments such as those detailed in section IV-B show promising correlation

and prediction, preliminary results with primitive data did not show expected trends. Specifically, before AGUILLE and our machine-learning workflow became capable of more sophisticated data transformations, the sample heuristic (viz. mean amount of buttons per layout in each application) correlated against average rating showed no statistically significant results.

After improvement to AGUILLE and the addition of metadata and tag phrases, statistically significant results surfaced. Category discretization provided even better results, as discussed in section IV-B on the preceding page. We believe further probing into consequential design decisions and further sophistication of AGUILLE and the design heuristic will continue to render reportable, statistically significant results.

For example, a further step up in sophistication involves a report of what percentage of all available screen space is occupied by widgets.

VI. RELATED WORK

Available research into the overlap of the machine learning and user experience fields tends to concentrate on either GUI testing or programming interface (API) design rather than using machine learning to gain insight on the GUI design patterns users favor. Much available research that does indeed combine machine learning and user interface design aims to design front-end applications for the non-statistician that enable powerful data mining with little knowledge of the implementation of machine learning algorithms.

Arlt et. al. [6] have written a chapter on various different methods of parsing and testing GUIs. The research of Nguyen et. al. [7] presents a tool called GUITAR to parse the structure of an application's GUI in order to generated automated tests for that application. We were originally hopeful that GUITAR or one of its derivatives could prove invaluable in gathering GUI data to mine. Unfortunately, the Android-specific fork of GUITAR has not been updated since the release of Android 2.2 and is therefore not compatible with the majority of applications on F-Droid. Although much existing research [6], [8], [9] makes use of GUITAR, our GUI-parsing tool had to be developed from scratch as discussed in section III-B on page 3.

The approach posited by Yang et. al. [9] to programmatically generate GUI models in mobile applications does not use GUITAR in its entirely; rather, it analyzes GUI events using GUITAR and calls these events directly on the application. Similarly, Amalfitano et. al. [10] showcase "an automated technique that tests Android apps via their [GUI]." Although writing a GUI parser from scratch may have slowed down development, using just one tool to rip the GUI of an Android application where other teams have used many has helped to simplify the process of gathering GUI data.

The research of Shi et. al. [4] and Ferenc et. al. [5] discusses ways to better understand source code design patterns in Java and C++, respectively. Our research aims to discover design patterns in graphical interfaces, not implementation patterns in source code, setting our research apart from other pattern-based learning research.

Lieberman's research [3] discusses the concept of an "Interface Builder," a graphical tool assisting a developer in designing a user interface. He discusses *Programming by Example,* where the developer "takes on the role of operating the user interface in the same manner as the intended end-user would, interacting with the on-screen interface components to demonstrate concrete examples of how to use the interface." The application then learns and generalizes the developer's input to guess at the desired functionality. This research may prove invaluable to future research where the "smart interface builder" discussed above is being designed.

Papatheodorou's research [11] focuses chiefly on using machine learning to learn over time the sort of interface the user may expect and to adapt to that knowledge. It may be possible to use aspects of [11] to generalize the expectations of a range of users or potential users during the design process rather than after deployment, saving developers valuable time to test and improve their software. Our work, however, proposes a different approach to interface learning, requiring no such lengthy data collection process from users. We learn from freely available data, speeding up empirical research and eschewing the technical challenges and privacy concerns inherent in collecting data directly from users.

A promising, unique opportunity to combine the machine learning and user experience fields is missing in available research. We hope to open the door for future research to use machine learning and data mining to analyze a wealth of existing information about user interfaces. This will help developers of all platforms to better understand their users' preferences and peeves not only in graphical or mobile environment design but also in the design of overall user experience. With more intuition on user propensity and preference, developers can design more natural, intuitive software.

VII. FUTURE WORK

Our model would determine the likely rating for a graphical interface given the analyzed source of the application. Given a prototype GUI designed by a developer in an interface builder, the model could guess at a rating for the design based on trends it found in other applications in the same category.

It could also be possible to use a genetic algorithm to determine better positions and attributes for the interface elements in the GUI. The algorithm would weigh a high rating (as determined by the model) against changing the developer's design as little as possible, creating incremental changes to generate potentially optimized versions of the developer's original layout.

VIII. CONCLUSION

We have discussed a method for learning the correlation between certain elements of GUI design, which we will analyze with AGUILLE, and Google Play ratings, which we have mined from the Web. This is improved when factoring in Google Play metadata and discretizing applications by category. Because of the importance of optimized, intuitive interface on smaller devices, developers will benefit from insight from a model that attempts to explain user behavior and preference.

A summary of specific results should go here. It will mirror the summary that will end up in the introduction.

We are confident that further (perhaps automated) probing will continue to reveal interesting relationships between different elements. After learning our model, we could predict the quality of future interfaces. With future refinement, the algorithm could suggest interface improvements by means of a genetic process in an interactive interface builder.

REFERENCES

[1] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 83–92.

[2] M. DeGusta, "Android orphans: Visualizing a sad history of support," 2011.

[3] H. Lieberman, "Computer-aided design of user interfaces by example," in *Computer-Aided Design of User Interfaces III*. Springer, 2002, pp. 1–12.

[4] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 123–134.

[5] R. Ferenc, A. Beszédes, L. Fülöp, and J. Lele, "Design pattern mining enhanced by machine learning," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 295–304.

[6] S. Arlt, S. Pahl, C. Bertolini, and M. Schäf, "Trends in model-based gui testing," *Advances in Computers*, vol. 86, pp. 183–222, 2012.

[7] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[8] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 77–83.

[9] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[11] C. Papatheodorou, "Machine learning in user modeling," in *Machine Learning and Its Applications*. Springer, 2001, pp. 286–294.

Learning User Behavior for Mobile Test Suite Adequacy

Cody Kinneer
Allegheny College
Email: kinneerc@allegheny.edu

***Abstract*—Software development for mobile devices proceeds at a rapid pace. Software as a service, rapid development, and agile programming means that mobile applications are released and updated quickly. As a result, developers have less time to test their applications and cannot completely know the effects of a change. Existing test suite adequacy criteria are insufficient in this quickly changing environment.**

In this paper, we develop a new behavior based test suite adequacy criterion that adapts to user interactions with an application in the wild. We evaluate the time and space overhead of they system and perform an empirical study analyzing existing Android test suites according to our behavior driven criterion. Our analysis reveals that the two test suites focused testing on infrequently used contexts, achieving behavioral adequacy scores from 12 to 33 percent less than the probabilistic calling context coverage. This shows the potential for substantial improvement in the development of test suites for mobile applications.

I. Introduction

Developers frequently use test suites, a collection of test cases, to ensure that the component under test performs according to specification, or to ensure that accuracy does not change over time. A test suite's usefulness lies in its ability to detect problems. With the rise of software as a service and rapid development practices, test suites must be effective in detecting important problems quickly. However, since it is not known beforehand where a problem will occur, determining the adequacy of a test suite is a challenging problem.

The most common test suite adequacy criterion is structural coverage. These criteria, such as line or block coverage seek to maximize the amount of code exercised by a test. Since a bug needs to be executed to be exposed by a test, maximizing structural coverage is a reasonable strategy. However, this definition of test suite adequacy suffers from not taking into account the importance of the structure covered. Achieving complete test coverage in practice is most often wishful thinking, and structural adequacy fails to provide insight into what areas of the application are more important to test. Furthermore, a good test should resemble the conditions under which the application will actually be used, but structural techniques say nothing about the realism of a test.

Another approach to test adequacy is fault-finding. This consists of introducing faults into an application, and then determining which tests tend to find the greatest number of faults. Mutation testing is one such technique. However, in practice, mutation testing speaks to the ability of a test to find faults in a certain structure. It cannot tell us what structures are more important to search for faults in.

With the rise of new software engineering paradigms such as software as a service, agile, and rapid development, these criteria fail to keep up with the pace of software development. This is particularly true of Android applications. According to Android's website, there are 7 Android API's in use [1]. The rate of change of the Android OS itself is a testament to the rapid development of Android applications.

These adequacy measures could be improved by taking into account the way users interact with the application after it is deployed. A behavior driven adequacy criterion confers two benefits. Firstly, if the purpose of application is to be used by a user base, then more frequently utilized components are more important than those that are less frequently used. A problem that occurs in a more frequent use area will affect more users. Additionally, a test suite's similarity to observed user behavior favors tests that are more similar to the conditions that the application will be exposed to in the wild.

Previous work in model-based software testing applied Markov chain models to software testing [2], [3], [4]. These works discuss how a Markov chain used to model software usage could be useful for input generation, software specification, and statistical software testing. However, they do not address the issue of how the model should be generated, and do not focus on test suite adequacy.

In this paper, we present a new test suite adequacy criterion that takes into account learned user behavior in the wild. By collecting data from users actually interacting with an application, we learn a Markov chain that models user behavior. This model can be continuously updated to respond to changes in the users' interactions. We then determine a test suites adequacy by its similarity to the constructed user behavior model.

We seek to determine how well test suites for Android applications reflect real user behavior. We evaluate our technique in terms of time and space overhead for a collection of Android applications, and evaluate the test suite adequacy of the applications using our proposed criterion.

The contributions of this paper are therefore as follows,

- A new behavior driven test suite adequacy criterion (section III).
- An implementation of the criterion for Android applications (section III and section IV).
- An empirical study evaluating the overhead of the implementation (section IV).

- An evaluation of several Android applications' test suites using the new criterion (section IV).

II. Background

To calculate behavioral adequacy, a technique called probabilistic calling context [5] is used for profiling and a Markov chain is used for modeling.

Profiling

Calculating a behavioral criterion requires that an application's behaviors be profiled. A program's behavior can be thought of as the collection of its function calls, which makes profiling based on these calls a reasonable choice for modeling application behavior.

Probabilistic calling context (PCC) is a profiling strategy developed by Bond and McKinley [5]. PCC attempts to assign an integer to every unique stack state. This system is useful because it can be computed efficiently, only 3% overhead is reported in the literature. An example of a stack state that could be represented by PCC is shown in Figure 1. This figure shows an example stack state inspired by the K-9 Mail email application. First, MAIN is called, which is assigned a PCC value of zero. Then, MAIN calls CHECKEMAIL, (hereafter, we will use $\rightarrow$ to signify a function call). The next PCC is calculated from the last PCC and the name of the CHECKEMAIL function. Thus, the next PCC value is meant to represent the sequence MAIN $\rightarrow$ CHECKEMAIL. If CHECKEMAIL were to return to MAIN, then the PCC value would also return to zero. Instead however, CHECKEMAIL calls ITERATEACCTS, so the next PCC value is calcuated from the previous PCC value and the next function name. When a function is called, the next PCC value is given by

$$nextPCC = currentPCC * 3 + currentContext$$

where $currentContext$ is an integer that represents the current context, such as a hash value of the called function name.

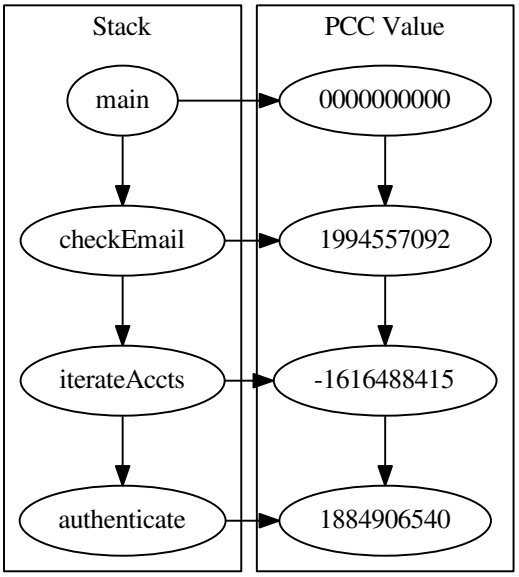

Fig. 1. PCC value updating as methods are added to the stack.

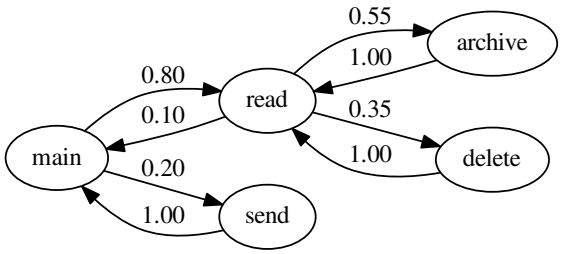

Fig. 2. An example of a Markov chain behavior model inspired by K-9 Mail.

Markov Chains

A Markov chain is a state based system where the next state depends only upon the current state [6]. An example is shown in Figure 2. The nodes in the graph represent states, and the edges represent the transition probabilities. Starting at the MAIN state, there is an 80% chance of transitioning to the READ state and a 20% chance of transitioning to the SEND state.

Markov chains have been used to model expected user behavior in model based testing [2], [4], [3]. However, these techniques generally do not learn models from user behavior, but reflect how the developer expects users to behave.

III. Behavior Driven Test Suite Adequacy

Using PCC and Markov chains, we present a technique for assessing a test suite's adequacy based on how users interact with the application being tested. Our technique for calculating behavioral adequacy is shown in Figure 3.

First, the application is instrumented to collect data for profiling user behavior. Because the application will be used while this information is collected, the overhead incurred by the user must be acceptably small. Additionally, the information gathered must be useful in modeling user behavior. PCC was chosen because it satisfies both of these requirements. A program's behavior can be thought of as the sequence of functions that it calls, which makes profiling based on function calls an appealing strategy for profiling behavior. Since PCC takes into account the functions on the stack, it provides more information than simply profiling based on function frequency, while still maintaining low overhead. The instrumentation calculates the current PCC value from the calling context, and records each transition between PCCs. The application is then released for use by the user base, and behavioral data is collected in the form of these PCC transitions.

The developer then runs the application's test suites on the instrumented application. The transitions between PCCs

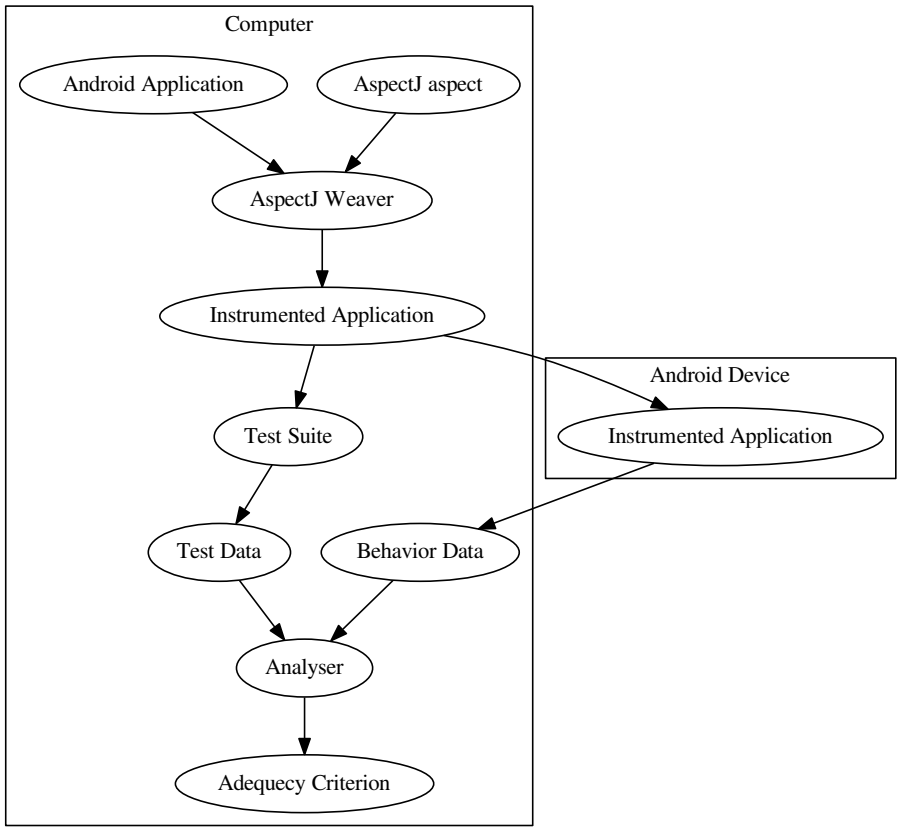

Fig. 3. Framework for a behavior driven test suite adequacy criterion.

observed during testing are recorded as well, giving test data that characterizes the behavior of the test suite in terms of the observed PCC transitions.

The user data is then aggregated and used to construct a Markov chain where the nodes are PCC values and the edges are the probability that a PCC value will transition into another PCC value.

For example, suppose the data in the table below was collected from users interacting with a simple email application. Caller PCC represents the current PCC value, and callee PCC represents the next PCC value. Rather than integer representations, the names of functions on the stack are given for explanatory purposes.

Caller PCC	Callee PCC	Frequency
MAIN	MAIN → READ	80
MAIN	MAIN → SEND	20
MAIN → SEND	MAIN	20
MAIN → READ	MAIN	8
MAIN → READ	MAIN → READ → ARCHIVE	44
MAIN → READ	MAIN → READ → DELETE	28
MAIN → READ → DELETE	MAIN → READ	28
MAIN → READ → ARCHIVE	MAIN → READ	44

This data would be converted to a Markov chain similar to what is shown in Figure 2. In the example, the nodes are function calls rather than PCC values for the sake of explanation. For example, the ARCHIVE node represents the context MAIN → READ → ARCHIVE. If the archive function could be called in a different context, for example,MAIN → SEND → ARCHIVE, that context would be represented by a different PCC value. However, in this simplified example, each function can only be called in one context.

For every caller PCC, there is an edge to each callee PCC. The transition probabilities can be found by taking the frequency of a given callee PCC divided by the sum of the frequencies for the corresponding caller PCC. For example, for the caller PCC MAIN, there are two possible callee PCCs. The probability of transitioning to MAIN → READ is $\frac{80}{80+20} = 0.80$. Alternatively, the probability of transitioning to MAIN → SEND is $\frac{20}{80+20} = 0.20$.

To calculate test suite adequacy, the sum of edges in the model observed during testing is divided by the sum of all edges in the model. For example, consider the example Markov chain shown in Figure 2.

If this Markov chain were constructed from user behavior, then when the application was in the hands of users, READ is called from MAIN 80% of the time while SEND is called 20% of the time. If during testing we exercised MAIN → READ,

then we would have a behavioral adequacy of:

$$\frac{.8}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .16$$

If instead, we tested MAIN → READ and READ → DELETE, then our adequacy increases because we are covering more code.

$$\frac{.8 + .35}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .23$$

However, if we test MAIN → READ and READ → ARCHIVE instead, then our adequacy increases further because ARCHIVE is more likely to be used than DELETE.

$$\frac{.8 + .55}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .27$$

This makes sense intuitively since testing more behaviors increases the score, and testing more frequently used behaviors further increases the score. Using this criterion, 100% adequacy is achieved when every behavior observed by the user-base is tested, and 0% is achieved when no behavior observed in the user-base is tested.

IV. Empirical Evaluation

To evaluate our proposed test suite adequacy criterion, we implemented a system for calculating behavioral adequacy for Android applications. The goals of the evaluation are as follows.

1) Determine the time and space overhead of the online behavioral profiling.
2) Determine the overhead associated with calculating behavioral adequacy offline.
3) Evaluate the behavioral adequacy of existing Android applications.

A. Experiment Design

To instrument applications, we used AspectJ because it provides a way to quickly instrument Java and Android applications. An AspectJ aspect was written to calculate the PCC value of the application at function calls. Only application defined functions were considered, so Android system calls and Java library calls were ignored. At each call, the current PCC, the caller, and the next PCC, the callee, were stored as a 64 bit value identifying a transition between PCC values. The frequency of these transitions were recorded, and written to a file upon an activity being paused, stopped, or destroyed. This data was then sent to a desktop PC for processing. A Java program was implemented to construct a Markov chain from the PCC edges. The test suite under study was then executed and the PCC edges collected on a per test basis. For structural coverage, Android's included EMMA tool was used.

Offline tasks were completed using a desktop running Centos 6.5 with a quad-core 1.6GHz CPU and 16MB of memory. User data was collected on an Asus Nexus 7 tablet running Android 4.3 and a Samsung Galaxy SIII smartphone running Android 4.1.1.

B. Case Studies

To conduct our evaluation, we selected several applications from the F-Droid open source appstore. We attempted to select well known applications with large test suites, however this was difficult since few applications contained test suites. The applications selected were K-9 Mail, and Github. K-9 Mail is an email application that can connect to IMAP, POP3, and SMTP servers to manage a user's email accounts. Github is an application that allows a Github user to interact with Github on an Android device. It supports browsing repositories, commenting, and creating Gists and issues.

Application	Files	Classes	Methods	Lines
K-9 Mail	230	806	5671	35410
Github	?	?	?	?

C. Metrics

We evaluate runtime overhead in terms of percent change in time. Space overhead in terms of percent change in source code occupied space on disk. Structural coverage is given in percent of code covered. Behavioral coverage is given as the sum of exercised edges over the sum of all edges.

D. Experimental Results

To evaluate online runtime overhead, the benchmarks' test suites were executed five times, and the execution time was measured with and without profiling instrumentation. The average of the five trials was taken. The table below shows the results, time is given in seconds.

Application	Time Uninstrumented	Time Instrumented	Percent Change
K-9 Mail	4.482	10.385	132
Github	66.006	70.445	7

The large difference in percent change between the applications warrants additional investigation. A possible explanation is that K-9's test suite primarily tests backend code that tends to complete very quickly, whereas Github's test suite involves testing UI elements, such as creating activities that is less sensitive to the instrumentation.

To evaluate space overhead, the benchmarks binary size was measured before and after instrumentation. Size is given in megabytes.

Application	Size Uninstrumented	Size Instrumented	Percent Change
K-9 Mail	2.91	3.35	15
Github	1.76	1.99	13

The size overhead between the two applications was about the same, with both applications using around 14% more disk space when instrumented.

To evaluate offline overhead, we measured the time needed to build the model from user data, and determine behavioral adequacy from the model.

To evaluate Android application test suites, we instrumented the benchmarks and allowed two users to interact with the applications for one day. Afterwards, we profiled the benchmarks' test suites, and constructed a model from the user data. We then calculated behavioral adequacy from the model and test data. For comparison, we determined the structural

coverage of the benchmarks' test suites using EMMA, and the PCC coverage by dividing the number of PCC transitions exercised by both the users and the tests and the number of PCC transitions exercised by the users.

Application	Behavioral Coverage	PCC Coverage	Method Coverage
K-9 Mail	0.00016	0.00024	7
Github	0.03824	0.04324	?

E. Threats to Validity

The most significant thread to the validity of our evaluation is the limited number of applications tested. The applications selected for the evaluation may not represent all Android applications. This problem can be alleviated by conducting a larger study on a wider range of applications. Another threat is the limited number of users participating in the study. The users participating in the study may not be representative of the rest of the user-base. The more the users interact with an application, the more likely they are to exercise PCC values not seen during testing, and thus decrease the score. This means users interacting with an application longer than normal will likely cause the behavioral adequacy to decrease. Additionally, users interacting with the application for less time than normal could cause the behavioral adequacy results to be too optimistic. Alternatively, since behaviors exercised by the test suite but not by the users are given a score of zero, users exercising very little of the application may cause the score to decrease. This issue can also be mitigated by a larger experiment with many users to increase the chance that the users represent an accurate sample of the larger user-base.

V. Related Works

Relative coverage is an alternative to traditional coverage that takes context into account when determining coverage. This is useful in software as a service systems where only a portion of a larger service is used by an application. From the perspective of the smaller application, some features the larger service provides are not used, and thus, irrelevant. These features do not need to be tested, and therefore should not be considered when determining coverage. Relative coverage excludes these unused feature from the coverage equation.

Miranda and Bertolino's work [7] on Social Coverage is the most similar to our work. They propose a system inspired by relative coverage that determines coverage according to context. Relative coverage systems rely on the developer to select which features are relevant, while social coverage, like us, uses observed user behavior to determine what features are important. Social coverage collects user data and can find similar users. The features used by these users might be relevant to the application. These features are taken into account when calculating social coverage.

The Synoptic system [8] also has similarities to our work. Synoptic is a tool that can infer finite state models from reading execution logs. Like us, they construct a model based on user behavior. However, synoptic requires the application log states, and is thus more suited to a high level model, whereas we model based on calling context. Additionally, we apply the learned model to test suite adequacy while Synoptic focuses on analysing logs.

The Gamma system presented by Orso et al. [9] attempts to enable remote monitoring of software after its deployment. Gamma does attempt to address the issue of runtime overhead, and allows for the costs of instrumentation to be shared among users. The developer can specify what type of information they are interested in, and the Gamma system divides the task of collecting this information among the userbase. Additionally, Gamma allows for its instrumentation to be modified by an update.

Bond and McKinley [5] introduced a technique for decreasing the costs of tracking a programs calling context called probabilistic calling context. This system allows a calling context to be represented as an integer that is easy to calculate and well suited to anomaly detection applications. The technique consists of a function that takes as input the current probabilistic calling context and an integer representation of the current context. It then outputs an integer representing the current probabilistic calling context. The function produces outputs that are uniformly distributed, so that the chance of conflict is low, and the order of the contexts is taken into account.

In a later work, Bond et al. [10] present a technique for calculating the entire calling context from the probabilistic calling context. This technique has the advantage of being able to reconstruct the calling context offline, however, some dynamic information is needed make a search of the context space feasible, which requires additional overhead of 10-20%.

Elbaum et al. [11] present a study showing how software evolution affects code coverage. The study shows how even small changes can have a large impact. This work is similar to ours in that we are concerned with the performance of structural adequacy criteria in evolving software environments.

Whittaker presented a series of papers [2], [3], [4] on using markov chains in software testing, including input generation, software specification, and statistical software testing.

Andrews et al. [12] present a paper analyzing the usefulness of mutation testing. The study shows that mutation testing creates faults similar to real faults. The abc compiler provides a way to perform AspectJ instrumentation on Android bytecode without access to the source code [13].

VI. Conclusion

Android applications exist in a rapidly changing environment. Traditional test suite adequacy criteria such as structural coverage and fault finding adequacy provide insufficient guidance to developers in such a rapid development cycle. As an alternative, we propose a behavior driven test suite adequacy criterion that can adapt to changes in the environment when assessing an applications test suite. By instrumenting behavior on applications running in the wild, a markov chain is constructed that models user behavior. This behavioral data is then compared with data obtained during the execution of a test suite to determine the test suites adequacy. A case study of two applications suggests that there is potential for major improvement in the quality of test cases for mobile applications. A more comprehensive empirical study is needed to explore the technique's run-time overhead and evaluate the adequacy of additional application's test suites.

ACKNOWLEDGMENT

This work is supported by NSF REU Grant 1359275.

REFERENCES

[1] Android, "Dashboards," http://developer.android.com/about/dashboards/index.html, Jul. 2014.

[2] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 1, pp. 93–106, Jan. 1993.

[3] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 10, pp. 812–824, Oct. 1994.

[4] J. Whittaker, "Stochastic software testing," *Annals of Software Engineering*, vol. 4, no. 1, pp. 115–131, 1997.

[5] M. D. Bond and K. S. McKinley, "Probabilistic calling context," *SIGPLAN Not.*, vol. 42, no. 10, pp. 97–112, Oct. 2007.

[6] J. G. Kemeny and J. L. Snell, *Finite markov chains*. van Nostrand Princeton, NJ, 1960, vol. 356.

[7] B. Miranda and A. Bertolino, "Social coverage for customized test adequacy and selection criteria," in *Proceedings of the 9th International Workshop on Automation of Software Test*, ser. AST 2014. New York, NY, USA: ACM, 2014, pp. 22–28.

[8] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, "Synoptic: Studying logged behavior with inferred models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 448–451.

[9] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: Continuous evolution of software after deployment," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 65–69.

[10] M. D. Bond, G. Z. Baker, and S. Z. Guyer, "Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses," *SIGPLAN Not.*, vol. 45, no. 6, pp. 13–24, Jun. 2010.

[11] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 170–.

[12] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 402–411.

[13] S. Arzt, S. Rasthofer, and E. Bodden, "Instrumenting android and java applications as easy as abc," in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Springer Berlin Heidelberg, 2013, vol. 8174, pp. 364–381.

Diagnosis of Autism Spectrum Disorders Using an Interactive Diagnosis Program

Tate Krejci, *Student, UCCS*

***Abstract*—Asperger Spectrum Disorders (ASD) affect a relatively large portion of the population, causing difficulty in learning appropriate behaviors for various social situations. Tests to diagnose ASD require an expert, and symptoms can often be mistaken for other mental disorders leading to under-diagnosis. Therefore, the application of a machine learning algorithm in an interactive environment such as a program will potentially increase the amount of successful diagnoses of ASD. The successful implementation of such a program will not only increase the likelihood of successfully diagnosing ASD, but also increase our understanding of ASD.**

I. Introduction

Autism Spectrum Disorders affect approximately one in two hundred and fifty people, causing difficulty in the acquisition and understanding of normal social protocols [1]. Many cases go unnoticed or misdiagnosed as there is no definitive way to diagnose an ASD over other mental disorders without excessive trial and error [2]. Furthermore, early detection of ASD in children requires expert evaluation, and cannot easily be carried out by parents or teachers [3]. Therefore, an easily and cheaply distributable application for the detection of ASD using a machine learning algorithm has the potential to detect more cases of ASD at earlier ages, and potentially provide further insights into symptoms of ASD. A program can yield potentially greater results as a program can be tailored to be interactive and captivating to its target age group. This allows for greater amounts of data to be collected than if the application was of a less interactive nature.

The latest edition of the DSM (Diagnostic and Statistical Manual of Mental Disorders), the DSM 5 has grouped the previously distinct disorders of Asperger Syndrome and Autism into the same disorder, known as an Autism Spectrum Disorder (ASD). Because of this, it is vital to determine where on the spectrum an afflicted person lies to ensure they receive the help that they specifically need. A person with severe ASD will demonstrate symptoms commonly associated with Autism, while a person with mild ASD will have symptoms similar to Asperger Syndrome. Differentiating the patients based on severity will ensure the correct type of help is provided. Thus an interactive program that can not only differentiate between a person with ASD and one without it, but can also provide insight into the severity of a patients disorder will prove to be a valuable tool.

II. Previous Work

The detection of mental disorders through the use of programs has been considered before and effectively applied to children with ADHD, exhibiting a success rate of approximately seventy-five percent with the use of a machine learning algorithm [4]. The bulk of the work done on ASD has been to identify the symptoms of ASD, the predominant one being inability to learn proper social protocol through normal social interactions [5]. However mild ASD remains harder to diagnose than severe ASD as the signs are far more subtle, especially for those with high functioning Autism [6]. Furthermore, symptoms of ASD can have multiple implications, making determining if a disorder is in fact ASD difficult [7]. More symptoms of ASD include unusual patterns of interest and behavior often leading to children with ASD seeming distant or inattentive [8]. While the symptoms of ASD are well known and progress has been made in its diagnosis, there still exists no definitive way to determine if a disorder is within the Autism spectrum of disorders or something else entirely.

III. Social Learning Theory and ASD

The primary function of ASD is to impair the ability of those affected to learn appropriate social behaviors the way unaffected individuals learn. Social Learning Theory is the theory that explains by what methods people learn what social behaviors are acceptable and what behaviors are not, though currently little is actually known about how this process actually occurs. Examples of these social behaviors include knowing to look somebody in the eye when they are speaking to you or knowing not to talk over someone else [9]. Detection techniques today involve qualitative question and answer sessions, with little in the way of quantitative data to support one diagnosis over another, often leading to misdiagnosis [2]. This is exacerbated by the fact that there is no medication to treat ASD, so doctors cannot try varying medications to determine the true disorder, as is often the case when diagnosing ADHD. With the spectrum of high IQ ASD to low IQ ASD, doctors and psychologists find it difficult to create a definitive list of symptoms [9], meaning a machine learning algorithm has the potential to discover new patterns to assist in the diagnosis of ASD.

IV. Methodology

A. The program and Machine Learning Algorithms

To identify quantitative indicators of ASD, it will be necessary to use a supervised machine learning algorithm to group data collected during the program. The type of algorithm used will depend on what kind of data the program will collect, although it is likely that a type of clustering algorithm will help group ASD users together and help identify them. This will

hopefully allow the algorithm to discover new data sets that group together people suffering from ASD. Because of this, the testing phase of the program will be of utmost importance to ensure a large enough data set is collected to effectively predict whether a person has an ASD. To collect data, the program can measure variables such as answers provided, response time and mouse movements. The program will collect this data when the user is presented with social situations in which the correct response will not be readily evident to a person with an ASD. During early trials, the program can be tuned to give the maximum amount of data per encounter, and additional features can be added as needed. Programs of this style have already been implemented for the diagnosis of ADHD, with a success rate of approximately seventy-five percent [4].

B. Creating the Tests and Data Sets

Unlike some machine learning projects, total control over the data collection will be possible in this project. This means that designing the tests in the program will be as important, if not more important than fine tuning a machine learning algorithm. If the tests do not collect pertinent data that distinguishes people suffering from ASD, it is unlikely that even a well tuned algorithm could give a meaningful prediction. Therefore, extensive testing of the tests themselves will be a vital portion of creating a program for the diagnosis of ASD. To ensure the questions are well tuned, initial testing will occur only on people who do not show symptoms of ASD. With this data, it will be possible to determine what questions are effective because people without ASD should answer well written questions in the same way as other people without ASD. Once questions and scenarios have been verified through this method, they can be tested on people with ASD to determine if they can separate them out from regular people.

To create tests that capture relevant data, it has been necessary to partner with experts in psychology, specifically ASD and social learning theory. With their help, it has been possible to create scenarios in the program where the user is presented with options that indicate if they suffer from ASD or not. For example, the user could be presented with a situation where they will make differing choices based on their empathy for the characters in the program. People with ASD will likely show less empathy than those without it, as a lack of empathy is one of the characteristics of ASD [10]. Experts have expedited the process of creating tests that capture data relevant to the diagnosis of ASD. There is also the possibility to test some of the non-social aspects of ASD in a program such as the abnormal ways in which a person with ASD will focus on different tasks. Expert advice has been used to ensure that all the tests that have been implemented so far are true measures of ASD.

There are many symptoms that can indicate ASD such as deficits in executive functions [9]. This makes it possible to assess the severity of ASD in a given person by determining how impaired their executive functions are. Somebody with mild symptoms will likely show less impairment than somebody with severe symptoms [9]. To determine executive function impairment, an ordering section has been implemented where the user is shown multiple pictures of a scene and asked to select them in the order they think is best. This test can be developed to assess ASD specifically by using pictures portraying social interactions. Another test currently in the program shows the user a picture of a social interaction and asks them questions about it. Specifically, it probes the user's knowledge of what the various people in the picture think about one another, which is generally a challenge for people with ASD. The program also keeps track of the time taken to complete each individual question, meaning there is a possibility to filter out people with ASD based on the time taken to complete the various tests. Currently the program also includes the Coolidge Autism Symptoms Survey (CASS) which has already proven to be effective at diagnosing ASD [11]. This means that the program can build off of an already successful tool while adding new methods of diagnosis which are capable of measuring metrics that a pen and paper survey cannot.

C. Targets for the program

ASD manifests in different ways based on the person's age making it important to target a specific age group initially to develop both the program and the algorithm [12]. This will simplify the initial design of the program as it will only have to include tests for that specific age group, and not all possible age groups. For this reason the final program will be targeted toward 3rd grade age students. This is the age when most students have gained sufficient experience reading to take text based tests, and is regarded as the age when the symptoms of mild ASD first become visible [9]. This also means that the program will help those with ASD get help as soon as possible, greatly increasing their quality of life in later years.

An important note is that children in this age group are just beginning to read, so its important that any test targeted at them not accidentally test reading ability and comprehension. To do this it will be necessary to keep the amount of reading needed to a minimum, have a parent help the child, or implement a voice-over feature. For initial testing, a researcher will likely be present to answer any questions about the application, meaning that in initial phases of testing, the amount of reading is not a major concern. This is especially important when analyzing the amount of time it took to complete each question. A voice-over function that could read aloud questions and answers would serve to make the time to read prompts constant, so time taken to complete a question would be due primarily to thinking time. Another viable option is to make the tests use pictures for both prompts and answers, although doing so affects the kinds of data that can be collected. The final program will feature both voice overs and picture based tests to ensure reading ability is not an aspect of the data that is collected.

Due to the difficulty of conducting trials on children, initial tests have been conducted on a number of colleagues (8) to determine if questions are answered consistently by people who can be considered to be free of ASD. This assumption can be made because those undergoing initial testing are using

an application designed for young children, and have a high probability of selecting answers that indicate they do not have ASD. This means that if a specific question is not answered consistently, it is likely a confusing or ambiguous question and should be rethought. Early testing also provides feedback on the general design of the application, all of which will lead to a far more refined application when official trials do begin and gives a partial data set to begin training the learning algorithm. Early testing has also provided valuable insight into how often 'normal' people make mistakes on the tests, which will be valuable information when training the algorithm.

D. Implementation

To make the program easily distributable and appealing to the largest audience, the program has been developed for PCs using a Windows environment. Windows has many well established frameworks for creating interactive programs such as Unity and XNA Studio, simplifying the development of a program for a Windows platform. The program will primarily perform data collection, and if successful diagnosis, but it will prove unwieldy for the program to also store and process data at larger scales. Initially the program will store data locally for simplicity, but later can send data to a data storage system. When the program is completed with the testing phase, it can be deployed to a web based player. This will allow the program to be accessed by a website, removing the need for users to install a piece of software to use it.

The program will consist of two main parts, the test portion and the data processing portion. The testing portion will consist of all the tests designed to gather data to determine a diagnosis. This portion will also include the CASS which can be treated separately internally as a cross check within the program the validity of the results for unknown cases. The second portion will be the data analysis portion which will handle all of the data processing. While the program is in the testing phase, the program will be separate from the testing portion so it can be fine tuned without having use the testing portion. Later, the data processing portion will be added at the end of the testing portion so it can give users an immediate result as well as integrate the data provided by them.

An important feature of the program will be the artwork used, as many of the tests will require specific ideas to be conveyed through pictures. At the start of the project, it is impractical to use custom artwork so images found on the Internet will be used in the early iterations of the program. While these may not convey a message perfectly, early testing should determine their efficacy. From early testing it will be evident what types of image question combinations work the best, and early testing can be based on these findings. If the results of the testing are positive, later versions of the program can include custom artwork. This will allow greater control of the messages conveyed by the pictures and will allow for more customization in the scenarios that are presented. Currently, some scenarios can not be implemented due to restrictions in the types of art available, so attaining custom art will likely increase the performance of the program once it is acquired.

E. Current Tests

Currently two different types of tests have been implemented in the application which each have multiple individual questions. The first type of test presents the user with multiple pictures that together depict a task or social interaction from beginning to end. The user clicks on each tile in the order that they think is best. This test serves to test the user's executive functions, which is lacking in many people with ASD, as well as their understanding of social situations [9]. Both the answers and the time taken to complete the question are stored by the application. Fig. 1 on the following page shows a screen shot of one of the questions in the ordering portion of the application. The second type of test implemented shows the user a picture of some social interaction and gives them a prompt and a series of answers. Some of the answers delve into what the people in the picture are actually thinking, which is how people without ASD should answer. The other answers have less to do with what the people in the pictures are actually thinking, and are likely to be chosen by people with ASD. Fig. 2 on the next page shows a screen shot of one of the questions in the intentions portion of the application. It should be noted that the artwork in the screen shots is not the artwork that will be included in the final iteration of the application. The current artwork was chosen based on availability so that prototyping and initial testing could begin rapidly.

V. Learning Algorithm

The essential function of the program is to determine whether a person has ASD. This means that there are two classes to map results to: ASD or non-ASD. Because of this, it seems a classification algorithm lends itself to the problem. Since classification generally follows a supervised structure, example data from people with and without ASD is necessary. This works out well as determining the efficacy of the various tests will mean testing the program on people with ASD before the learning algorithm is even implemented.

To first determine an effective learning algorithm, it is necessary to consider how the data collected should be processed. The program will consist of various tests with multiple parts, many of which will feature multiple choice questions. It is likely that scoring each different test and using a score from each test as the parameters for classification will be most effective. This way the number of variables is limited. However, if this proves ineffective it may also be possible to use every answer for the classification or perhaps use a regression type algorithm on the results pertaining to individual tests. The key issue will be making the number of parameters small enough for efficiency, while retaining meaning.

A Naive Bayes classifier will be used as the algorithm as it is effective at taking many parameters and calculating the probability of a class based on those. Because it functions by summing the probabilities that each individual attribute leads to a specific class, it will automatically give each answer a weight based on how effective it is at classifying ASD or non-ASD. This will hopefully add even more value to the program, as each test will affect the final outcome based on its efficacy. Another function of a Naive Bayes algorithm is

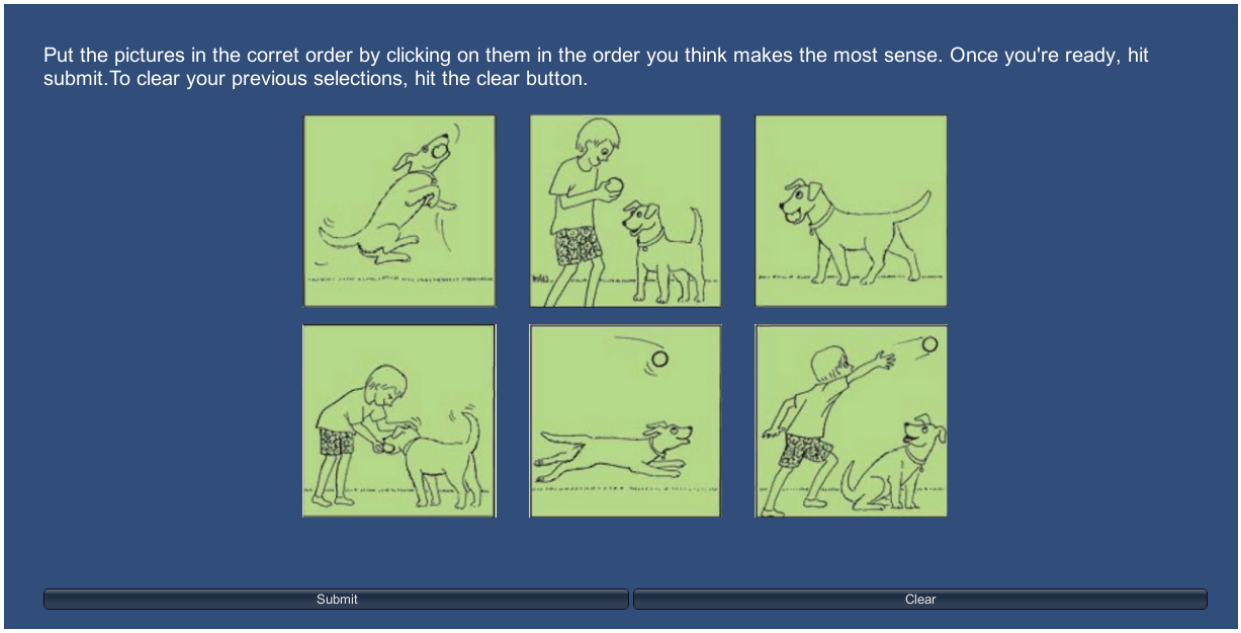

Fig. 1. Ordering Test

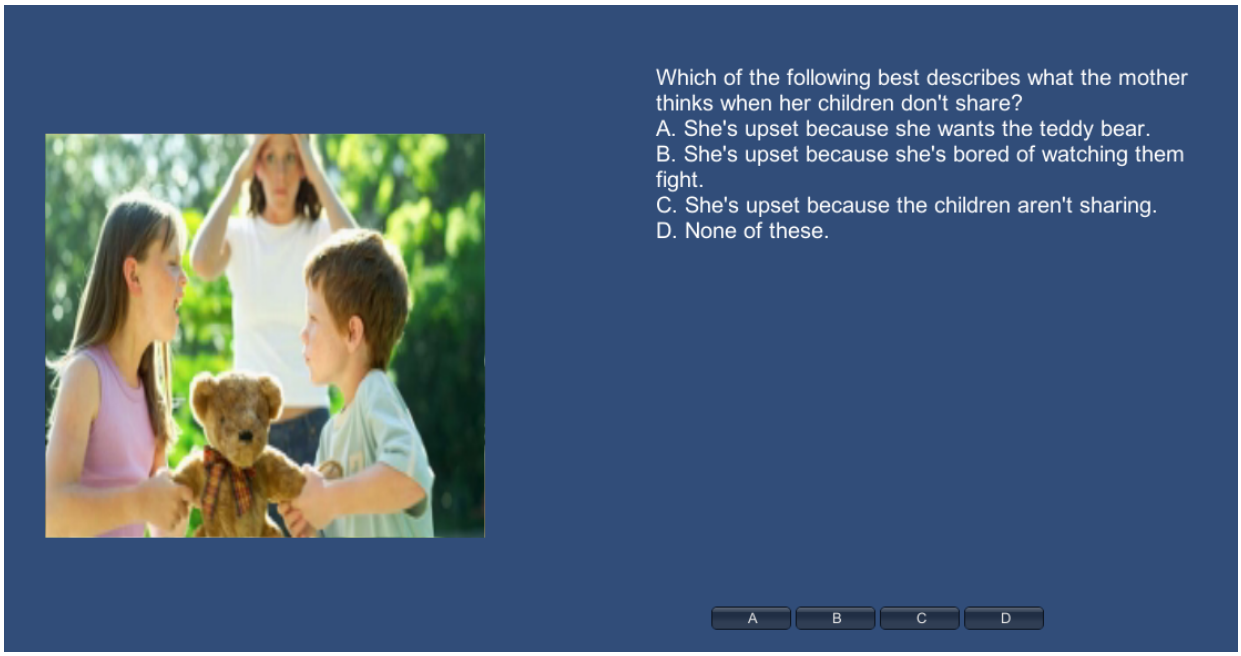

Fig. 2. Intentions Test

that the probability of a given parameter resulting in a specific class is not dependent on previous parameters. This may be seen as a hindrance in some cases however, the answer to one question does not have any impact on the answer to another. In this case, the disregard of previous answers serves to simplify the algorithm, making it more efficient. The fact that prior probabilities are disregarded likely will have no effect on overall results. The equation below shows the Bayes rule, off of which the Naive Bayes algorithm is based.

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

To use this algorithm, you sum the probabilities of each element giving a certain class based on weight. Thus it is possible for it to be effective with large amounts of data and will hopefully give good results when implemented. As mentioned before, it is important not only to classify users as ASD positive or negative, but also give a measure of the severity of their affliction.

The naive assumption of the Bayes algorithm removes the denominator of the equation, representing the assumption that the individual probabilities of each element of the classifier are independent of each other. As the answers to a given question

TABLE I
INITIAL RESULTS FROM NON-ASD SUBJECTS

Question Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	ASD
Participant 0	B	C	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 1	C	C	A	A	B	B	B	B	C	C	A	C	A	A	C	A	C	N
Participant 2	B	C	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 3	B	C	A	A	B	B	B	B	C	C	B	C	A	C	C	A	C	N
Participant 4	C	C	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 5	B	C	A	A	C	B	B	B	C	C	A	C	A	A	C	A	C	N
Participant 6	C	C	A	A	B	B	B	B	C	C	A	C	A	A	C	A	C	N
Participant 7	C	D	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 8	C	C	A	A	B	B	B	B	C	C	A	C	A	A	C	A	C	N

are unlikely to affect one another, the naive assumption seems a safe assumption to make for this application. To determine the probability of a given class, the probabilities that each individual property lead to a given class are multiplied together and this is then multiplied by the probability of a given class and is shown in the formula below where S is a selection of n attributes.

$$p(A|S) = \sum_{n=1}^{n} p(S_n|A) * p(B)$$

VI. INITIAL RESULTS

At this time, the main tests that have been conducted have been to assess the efficacy of various styles of questions and tests. Currently, children with a diagnosis of ASD are unavailable for testing, so tests have been conducted with colleagues. Because they do not have diagnoses of ASD, testing them provides an insight into the responses of non impaired persons. If the answers provided by them generally match for a specific question, the question is at least effective at grouping users without ASD together. This testing will help filter out questions that are ambiguous and give inconsistent data. When a test has been verified as consistent, more tests can be created based on those. The next step will be to give the refined tests to people with ASD to determine that they are also effective at distinguishing them from non-impaired people. TABLE I shows some of the preliminary data collected from users who do not have ASD using the intentions test. In this test, the user is presented with a picture of people performing various actions. The user is provided a prompt and a selection of answers. The answers each present a different level of social awareness, so people with ASD will likely pick the answers that show less social awareness.

Here the rows represent the answers submitted by each participant and the columns represent each question in one of the tests. For the most part, the answers are the same as expected. Differing answers to the same question indicate an ambiguous question that should be rethought so it is consistently answered the same for all people without ASD. The questions whose answers are all the same represent good questions with a style that should be repeated when adding new questions to this test. It should be noted that isolating good styles is accomplished by using the same prompt and image for a question and just varying the answer choices. The focus for this particular test was on creating distinguishing answers, which is why they were the factors that changed to determine the efficacy.

To ensure that a naive Bayes algorithm is an appropriate choice for the algorithm, test data was generated to represent the answers of users who were suffering from ASD. This data was generated pseudo-randomly with the goal of testing the classifier in mind. These are not results from real people with ASD. When this data was used with the data obtained from colleagues, the algorithm classified eighty-seven percent of the cases correctly. This means that as long as ASD users answer questions in the predicted manner, a naive Bayes classifier will successfully distinguish between ASD and non-ASD users. With the implementation of further tests, and the collection of more data, hopefully this number can be further increased in the future.

VII. TIME LINE

At this point the core functionality of the program has been implemented. Multiple tests are completed and a Naive Bayes classifier has been added to provide in application results for users. Future work will involve collecting real data to train the classifier using children of an appropriate age. Another important task will be to improve the existing tests, and add new tests. More tests will make the application a more comprehensive test of ASD, likely increasing the accuracy of the diagnosis. Existing tests can also be expanded as limited art assets have made the test sections relatively short. With the acquisition of custom art, more tests can be devised and existing tests will have greatly increased accuracy. This is because with custom art, variables that can affect a person's answer can be easily eliminated. Once the application has reached a polished state, and preliminary data has been gathered, the program can be exported to a web player. This will means that more people will have access to the program which will provide more diverse training data and hopefully further improve the algorithm's accuracy.

VIII. GOALS

The completion of the prototype of the program will allow for small scale data collection and testing of the program. This will involve identifying a test group, and determining what member of that group suffer from ASD. From there, a naive Bayes algorithm can be applied to the data and used to identify patterns indicative of ASD. If the algorithm can successfully predict ASD in the targeted age group, it can be deployed on

a larger scale to collect more data to improve the algorithm and it can be modified to support various target age groups. With enough success, the program could be further modified for use as not only a diagnostic tool, but as a treatment for patients with ASD.

Deployment to a larger scale will involve extensive testing and polishing of the existing program to come up with a complete suite of tests measuring many different aspects of ASD in distinct ways. Once the program reaches this point, and with university approval, the program can be exported to a web browser so it can be taken online and collect data online. It can then be modified to also give a suggestion of a diagnosis of ASD so it can be used by real people online. If the online program proves successful, it may be possible to create more diagnostic tools for the various mental illnesses that exist.

IX. Conclusion

The successful implementation of a machine learning algorithm to diagnose ASD will provide parents and doctors with a more effective means of diagnosing and helping those suffering from ASD. It also has the potential to vastly increase our understanding of the symptoms of ASD and perhaps provide clues to its causes and increase our understanding of social learning. The use of a program as the primary platform for the test will increase patient interaction with the application, leading to a greater quality and quantity of the data collected.

References

[1] B. J. Tonge, A. V. Brereton, K. M. Gray, and S. L. Einfeld, "Behavioural and emotional disturbance in high-functioning autism and asperger syndrome," *Autism*, vol. 3, no. 2, pp. 117–130, 1999.

[2] B. G. Haskins and J. A. Silva, "Asperger's disorder and criminal behavior: forensic-psychiatric considerations," *Journal of the American Academy of Psychiatry and the Law Online*, vol. 34, no. 3, pp. 374–384, 2006.

[3] O. Teitelbaum, T. Benton, P. K. Shah, A. Prince, J. L. Kelly, and P. Teitelbaum, "Eshkol–wachman movement notation in diagnosis: The early detection of asperger's syndrome," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. 32, pp. 11 909–11 914, 2004.

[4] S. Srivastava, M. Heller, J. Srivastava, M. Kurt Roots, and J. Schumann, "Tangible games for diagnosing adhd–clinical trial results."

[5] L. Wing, "Asperger's syndrome: a clinical account." *Psychological medicine*, 1981.

[6] P. Howlin and A. Asgharian, "The diagnosis of autism and asperger syndrome: findings from a survey of 770 families," *Developmental Medicine & Child Neurology*, vol. 41, no. 12, pp. 834–839, 1999.

[7] D. V. Bishop, "Autism, asperger's syndrome and semantic-pragmatic disorder: Where are the boundaries?" *International Journal of Language & Communication Disorders*, vol. 24, no. 2, pp. 107–121, 1989.

[8] A. Klin and F. R. Volkmar, "Asperger's syndrome," *Handbook of autism and pervasive developmental disorders*, vol. 2, pp. 88–125, 1997.

[9] M. G. Winner, *Thinking About You Thinking About Me*. Think Social Pub, 2007.

[10] I. Dziobek, K. Rogers, S. Fleck, M. Bahnemann, H. R. Heekeren, O. T. Wolf, and A. Convit, "Dissociation of cognitive and emotional empathy in adults with asperger syndrome using the multifaceted empathy test (met)," *Journal of autism and developmental disorders*, vol. 38, no. 3, pp. 464–473, 2008.

[11] C. S. R. D. L. S. Frederick L. Coolidge, Peter D. Marle and P. Monaghan, "Psychometric properties of a new measure to assess autism spectrum disorder in dsm-5," *American Journal of Orthopsychiatry*, vol. 83, no. 1, p. 126–130, 2013.

[12] C. Koning and J. Magill-Evans, "Social and language skills in adolescent boys with asperger syndrome," *Autism*, vol. 5, no. 1, pp. 23–36, 2001.

Simplified Statement Extraction Using Machine Learning Techniques

Conor McGrory, Princeton University

***Abstract*—The automatic generation of basic, factual questions from a single sentence of text is a problem in the field of natural language processing (NLP) that has received a considerable amount of attention in the past five years. Some studies have suggested splitting this problem into two parts: first, decomposing the source sentence into a set of smaller, simple sentences, and then transforming each of these sentences into a question. This paper outlines a novel method for the first part, combining two techniques recently developed for related NLP problems. Our method uses a trained classifier to determine which phrases of the source sentence are potential answers to questions, and then creates different compressions of the sentence for each one.**

I. Introduction

Asking questions is one of the most fundamental ways that human beings use natural language. When someone studies a foreign language, many of the first utterances they learn are basic questions. The ability of a speaker to form a grammatical question — to request a specific piece of information from another party — is indispensable in most practical situations involving basic communication. Over the past five years, there has been a significant amount of new research towards developing computer systems that can automatically generate basic questions from input text. This is referred to in the literature as the problem of Question Generation (QG), and it has many potential applications in education, including the development of computerized tutoring systems and the generation of basic reading comprehension questions for elementary-level students. Although some studies in the past have tried to generate questions based on whole blocks of text [1], the majority of recent work done on QG has focused on the problem of generating factual questions from a single sentence of input.

Early attempts to solve this problem used complicated sets of grammatical rules to transform the input sentence directly into a question [2]. However, in 2010, Heilman and Smith [3] suggested separating the problem into two steps: first, simplifying the source sentence, and then transforming it into a question. The advantage of this approach is that grammatical rules are much better at transforming simple sentences into questions than they are at transforming complex ones. Our paper outlines a method for preforming the first step in this process, which we refer to as the problem of Simplified Statement Extraction (SSE).

Conor McGrory is participating in a National Science Foundation REU at the University of Colorado at Colorado Springs, Colorado Springs, CO, 80918. e-mail: cmcgrory@princeton.edu

II. Prior Work

In a paper also published in 2010 [4], Heilman and Smith developed a rule-based SSE algorithm that extracted multiple simple sentences from a source sentence. This algorithm recursively applied a set of transformations to the a phrase structure tree representation of the input sentence to generate the simple statements. By extracting multiple simplified statements from the source sentence, they greatly increased the number of possible questions that could be generated and the percentage of words from the input sentence that appeared in one of the output statements [4].

Two problems in NLP that are related to QG are cloze question generation and sentence compression. A cloze question is a type of question commonly used to test a student's comprehension of a text, where the student is asked, after reading the text, to complete a given sentence by filling in a blank with the correct word. One example could be the question

A _____ is a conceptual device used in computer science as a universal model of computing processes.

In this case, the answer would be *Turing machine*. Because these questions are commonly used in testing, and require no syntactic rearrangement of the source sentence (just deletion of a specific phrase), they seem like an easy place to apply QG techniques. However, selecting which phrase or phrases in the sentence to delete is somewhat difficult. A question like

A Turing Machine ____ a conceptual device used in computer science as a universal model of computing processes.

with the verb *is* as the answer would be completely useless to a student interested in testing their knowledge of basic computer science. An automatic cloze question generator needs to have some way of distinguishing informative questions from extraneous ones. Because the quality of a cloze question can depend on complicated relationships between a large number of factors (syntax, semantics, etc.), distinguishing quality of a question is a good task for a machine learning system. Becker et al.[5] did this by training a logistic regression classifier on a corpus of questions paired with human judgements of their quality. The classifier was able to identify 83 percent of the high-quality sentences correctly and only misidentified 19 percent of low-quality questions as high quality[5].

Sentence compression is the problem of transforming an input sentence into a shorter version that is grammatical and retains the most important semantic elements of the original. This can be used to generate summaries or headlines for large blocks of text. Various methods have been developed to attack this problem. Knight and Marcu [6] used a statistical language

model where the input sentence is treated as a noisy channel and the compression is the signal, while Clarke and Lapata [7] used a large set of constituency parse tree manipulation rules to generate compressions.

Filippova and Strube [8] developed a sentence compression system where the compressed sentence is generated by pruning the dependency parse tree of the input sentence. Using the Tipster corpus, they calculated the conditional probabilities of specific dependencies occurring after a given head word. These were used, in combination with data on the frequencies of the words themselves, to calculate a score for each dependency in the tree. They then formulated the problem of compressing the sentence as an integer linear program. Each variable corresponded to a dependency in the tree. A value of 1 meant the dependent word of that dependency would be preserved in the compression, and a value of 0 meant that it would be deleted. Constraints were added to the linear program to restrict the structure and length of the compression, and the objective function set to be maximized was the sum of the scores of the preserved dependencies.

The central assumption made by Fillippova and Strube's method is that the frequency with which a particular dependency occurs after a given word is a good indicator of its grammatical necessity. For example, transitive verbs like *chase* require direct objects, so the frequency of the *dobj* dependency after the head word *chase* in the corpus is very high. Although *chase* can also be the governor of a prepositional phrase, this is not grammatically necessary, so there will be many more instances in the corpus where *chase* does not govern a prepositional phrase, resulting in the frequency of the *prep* dependency after *chase* to be lower.

III. Problem Definition

In explaining our system, it will help to have a formal definition of the problem. We will define the problem of simplified statement extraction as follows:

For a source sentence S, create a set of simplified statements $\{s_i ... s_n\}$ that are semantic entailments of S. A sentence is considered to be a *simplified statement* if it is a declarative sentence (a statement) that can be directly transformed into a question-answer pair (QA pair) without any compression. Ideally, the interrogative transformations of the generated $\{s_i\}$ should include as many as possible of the set of QA pairs a human being could generate given S. We will call the ratio of computer-generated, grammatical QA pairs to human-generated QA pairs the *coverage* of the system.

IV. Solution

As Becker et al. [5] showed with their work on cloze questions, there are certain phrases in S that make sense as answers to questions and others that do not. The fundamental idea behind our SSE system is that knowledge of which phrases in S are good answers can inform the compression process, preventing us from missing important information and thereby maximizing coverage. We divide the SSE problem into two parts: first identifying potential answers, and then generating for each of these answers a compression of S where that answer is preserved. These compressions form the set $\{s_i\}$ of simplified statements. Because each one of these statements will ultimately be transformed into a question with the given answer, our goal when compressing for a particular answer is to find the *shortest grammatical compression* of S that contains the given answer. This will ensure that each selected answer is preserved in at least one of the simplified statements and that these statements will contain minimal amounts of extraneous information.

To select potential answers from the input sentence, we use a slightly modified version of Becker et al.'s cloze question generation system [5]. Because all questions are essentially requests for specific pieces of information, determining which phrases in S make good answers to a standard grammatical question is very similar to determining which phrases make good blank spaces for a cloze question. Once we have the set of possible answers, we use a more substantially modified version of Filippova and Strube's dependency tree pruning method [8] to generate the set of shortest grammatical compressions of S that contain each of the answers.

V. Answer Selection

We designed and implemented the answer selection system using the Stanford NLP Toolkit [9] and the Weka machine learning software [10]. It uses the corpus of sentences, QA pairs, and human judgments developed by Becker et al.[5] to train a classifier to find the nodes in the parse tree of the input sentence that are most likely viable answers to questions. Our implementation performs two basic functions. First, it has the ability to read in the corpus, calculate a set of features and determine a final classification for each potential answer, and output this data set as an .arff file (the standard file format used by Weka). When the program needs to find the good answers in an input sentence, it loads the classifier from the file, determines all grammatically possible answer phrases in the input sentence (this is based on a set of constraints given by Becker et al. [5]), and uses the classifier to determine which of these phrases are good answers.

A. Feature Set

The Stanford NLP Toolkit [9] provides us with two very useful tools for describing the grammatical structure of a sentence: a Penn Treebank style constituency parse tree and the Stanford dependency relations [11]. The Stanford dependency relations are a set of grammatical relations between governor and dependent words in a sentence. Some examples include verb-subject, verb-indirect object, noun-modifier, and noun-determiner. Essentially, it is a dependency grammar with more specific information than which words a given word governs and which words it depends on. The relations also have a set hierarchy. For example, the verb-subject, verb-object, and verb-adverbial modifier relations are all instances of the parent relation predicate-argument. This enables the user to work at different levels of detail. For our purposes, we used the 56 basic relations defined in the Stanford library to categorize all of our dependencies.

We used many of the same features as Becker et al.[5] did, but because we used a different NLP package to implement our system (we used Stanford's, they used a toolkit developed by Microsoft), some of our features were significantly different. At this point, we have also implemented far fewer features than they did. Our features can be divided into three basic categories: token count features, syntactic features, and semantic features.

The token count features we used were the exact same as those used by Becker et al. This category contained 5 features which had to do with the length of the answer in comparison to the length of the sentence, like the raw lengths of both and the length of the answer as a percentage of the length of the question.

The syntactic features were calculated using the constituency parse tree. Currently, our system uses three syntactic features: the Penn part-of-speech tag of the word that comes immediately before the answer in the sentence, the tag of the word that comes immediately after, and the set of tags of words contained in the answer phrase.

The semantic features use the Stanford dependencies system and are completely different than the semantic features used by Becker et al. The purpose of these is to determine the grammatical role the answer phrase plays within the sentence. We currently have four semantic features implemented: the dependency relation between the head of the answer phrase and its governor in the sentence, the set of relations between governors in the answer and dependents not in the answer, the set of relations with both governors and dependents in the answer, and the distance in the constituency tree between the answer node and its maximal projection.

B. Classifier

The classifier used in our system is the Weka Logistic classifier [12]. Because each instance is classified as either "Good" or "Bad", this is a binary logistic regression classifier, similar to the one used by Becker et al. However, Becker et. al also used L2 regularization (adding a constant multiple of the L2 norm of the regression coefficients to the error function as a penalty for overfitting), which we have not yet implemented.

C. Human Judgments

The corpus provided by Becker et al. consists of slightly over 2,000 sentences, each with a selected answer phrase and four human judgments of the quality of the answer. Human judges could rate answers as either "Good", "Okay", or "Bad". Because the classifier requires that each instance be classified in only one category, we had our program use the four judgments to calculate a score for each answer, which we then used to determine how to classify it in the data set. A "Good" rating added 0.25 to the score, an "Okay" added 0.125, and a "Bad" rating added nothing. This score is then compared to the threshold value (a pre-set constant in the program). If the score is greater than or equal to this value, the answer is classified in the data set as "Good". Otherwise, it is classified as "Bad".

VI. Results

We used the program to produce a data set from the Becker et al. corpus [5]. This data set was created using a threshold value of 1.0 (all four human judges have to rate the sentence as "Good"). Then, using Weka, a random sample of the sentences was drawn from this data to produce a subset with a comprable amount of "Good" and "Bad" sentences. This set contained a total of 582 instances, 278 of which were "Good" and 304 of which were "Bad". We tested both the Weka Logistic classifier [12] and the Weka Simple Logistic classifier on the data using 10-fold cross-validation.

The statistics we were most concerned with were the correct classification rate (the number of correctly classified instances divided by the total number of classified instances), the true positive rate (the number of correctly classified "Good" instances divided by the total number of "Good" instances), and the false positive rate (the number of incorrectly classified "Bad" instances divided by the total number of "Bad" instances). We also looked at the Weka-generated "confusion matrix," which summarizes the classifications.

For the Logistic classifier, the correct classification rate was 72.3%, the true positive rate was 78.4%, and the false positive rate was 33.2%. For the confusion matrix (which is normalized), we have:

	Classified "Good"	Classified "Bad"
"Good"	218	60
"Bad"	101	203

In total, 54.8% of the instances were labeled "Good" and 45.2% were labeled "Bad".

For the Simple Logistic classifier, the correct classification rate was 74.2%, the true positive rate was 81.3%, and the false positive rate was 32.2%. For the confusion matrix, we have:

	Classified "Good"	Classified "Bad"
"Good"	226	52
"Bad"	98	206

In total, 55.7% of the instances were labeled "Good" and 44.3% were labeled "Bad".

Becker et. al were able to get a true positive rate of 83% and a false positive rate of 19% at the equal error rate [5]. Although their false positive rate is lower, the true positive rate of our system is definitely comparable to theirs.

VII. Sentence Compression

To compress S into the different simplified statements, we used a modified version of the integer linear programming (ILP) model described by Filippova and Strube [8]. We first calculated probabilities of dependencies occurring after head words and used this as an estimate of the grammatical necessity of different dependencies given the presence of a head word. Along with all of the constraints placed on the ILP in the original model, we added an extra constraint that ensures the preservation of the answer phrase in the compression. We then used a linear program solver to solve the ILP for all length values between 0 and the length of S, generating a set of compressions of S with all possible lengths. From these compressions, we used a 3-gram model to calculate the Mean First Quartile (MFQ) grammaticality metric described by Clark et al. [13]. Compressions with an MFQ value lower

than a threshold were deemed grammatical, and the shortest of these was selected as the final compression of S for the given answer.

A. Dependency Probabilities

In order to be more precise, we used a larger set of Stanford dependencies to calculate the conditional probabilities than we did for the feature set in the selection part of the system. The extra dependencies included in this set were collapsed dependencies [11], which are created when closed-class words like *and*, *of*, or *by* are made part of the grammatical relation, producing dependencies like *conj_and*, *prep_of*, and *prep_by*.

To calculate the frequencies of dependencies after certain head words, we used a pre-parsed section of the Open American National Corpus [14]. Filippova and Strube [8] used part of the TIPSTER corpus to calculate their frequencies, but we lacked the computational resources to parse the data ourselves, so we used the pre-parsed data. The frequency of a dependency in our system is defined as the the number of words in the document that are governors of at least one of these dependencies. If a dependency appears more than once for a given governor word (e.g. if a noun is modified by two prepositional phrases), our program will only increase its count by one. This prevents the frequency of a dependency following a given head word from ever exceeding the frequency of the head word itself.

To prevent rounding errors, we used a smoothing function when calculating the probabilities from the frequency data. If we let $f_{(\ell|h)}$ be the frequency with which dependency of type ℓ occurs with head word h in the corpus, and let f_h be the frequency of word h in the corpus, then we define the smoothed probability $P_{(\ell|h)}$ to be

$$P_{(\ell|h)} = \log_2(\frac{f_{(\ell|h)}}{f_h} + 1)$$

Because $f_h \geq f_{(\ell|h)}$ and $f_h, f_{(\ell|h)} \geq 0$, $\frac{f_{(\ell|h)}}{f_h} \in [0, 1]$. Therefore, because

$$\log_2(x + 1) \in [0, 1] \forall x \in [0, 1]$$

we know that $P_{(\ell|h)} \in [0, 1]$ for all possible ℓ and h.

Finally, to avoid problems that come with probability values of zero, our system linearly maps the $P_{(\ell|h)}$ values from $[0, 1]$ to $[10^{-4}, 1]$.

B. Integer Linear Program

Like Filippova and Strube [8], we formulate the compression problem as an ILP. For each dependency in the parse tree (say, the dependency with the Stanford type ℓ, holding between head word h and dependent word w), we create a variable $x^{\ell}_{h,w}$. These variables must each take on a value of 0 or 1 in the solution, where dependencies whose variables are equal to 1 are preserved in the resulting compression and dependencies whose variables are equal to 0 are deleted, along with their dependent words. The ILP maximizes the objective function

$$f(X) = \sum_x x^{\ell}_{h,w} \cdot P_{(\ell|h)} \cdot t(\ell, P_{(\ell|h)})$$

where t is the *tweak function*, which corrects discrepancies between frequency and grammatical necessity that occur with some specific types of dependencies. For example, conjunctions (*conj*) occur very frequently in written English, but they are generally not necessary for the grammaticality of a sentence. Often, deleting parts of conjunctions can actually be an effective way to compress a sentence. Multiplying a particular probability by t linearly maps the range of that value from $[0, 1]$ to $[min_\ell, max_\ell]$. The tweak function is defined as

$$t(\ell, P_{(\ell,h)}) = max_\ell - min_\ell(1 + \frac{1}{P_{(\ell,h)}})$$

where

$$min_{conj} = 0.0, max_{conj} = 0.4$$

,

$$min_{det} = 0.4, max_{det} = 1.0$$

,

$$min_{poss} = 0.5, maxposs = 1.0$$

, and

$$min_\ell = 0.0, max_\ell = 1.0$$

for all other dependencies, which means that $t(\ell, P_{(\ell,h)}) = 1$ for all dependencies besides conjunctions, determiners and possessives. Our tweak function replaces the importance function used in Filippova and Strube's objective function [8].

Filippova and Strube also used two constraints in their model to preserve tree structure and connectedness in the compression:

$$\forall w \in W, \sum_{h,\ell} x^{\ell}_{h,w} \leq 1$$

$$\forall w \in W, \sum_{h,\ell} x^{\ell}_{h,w} - \frac{1}{|W|} \sum_{u,\ell} x^{\ell}_{w,u} \geq 0$$

and one to restrict the length of the final compression to α:

$$\sum_x x^{\ell}_{h,w} \leq \alpha$$

To ensure that all of the words in the pre-selected answer A are also preserved, we include in our model the extra constraint

$$\forall w \in A, \sum_{h,\ell} x^{\ell}_{h,w} \geq 1$$

We solved these integer linear programs using lp_solve [15], an open-source LP and ILP solver.

C. Shortest Grammatical Compression

In order to find the shortest grammatical compression of S, our system first finds a solution to the ILP for S and A for every value of α (the maximum length constraint parameter) between the length of S and the length of A. Because the constraints also specify that every word in A is preserved in the compression, any model where α is less than the length of A would have no solution.

Although all solutions to the ILP are connected dependency trees, some of the actual sentences created by linearizing these trees will not be grammatical. To determine the grammaticality

```
Sentence: Bill drives his car to the park every morning.
Answer: the park
Compressed Sentences:
Compressions:
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
drives his car to the park every morning .      S = 1.2192966633804272
Bill drives to the park every morning .          S = 1.1363540097281664
drives to the park every morning .      S = 1.2192966633804272
Bill drives to the park .       S = 1.06102567648667
drives to the park .    S = 1.133694190282954
Best Compression: Bill drives to the park .
```

Fig. 1. Simulation Results

```
Sentence: Bill drives his car to the park every morning.
Answer: every morning
Compressed Sentences:
Compressions:
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
drives his car to the park every morning .      S = 1.2192966633804272
Bill drives to the park every morning .          S = 1.1363540097281664
drives to the park every morning .      S = 1.2192966633804272
drives to park every morning .  S = 1.1429945444805045
Best Compression: Bill drives his car to the park every morning .
```

Fig. 2. Simulation Results

of the compressions, we use the MFQ metric, which is based on a 3-gram model created using the Berkeley Language Model Toolkit [16] and trained on the OANC text. This metric was shown to work well at distinguishing grammatically well-formed sentences from ungrammatical ones by Clarke et al. [13]. It considers the log-probabilities of all of the n-grams in the given sentence, selects the first quartile (25% with the lowest values), and calculates the mean of the ratios of each n-gram log-probability over the unigram log-probability of that n-gram's head word. The larger the MFQ value is, the less likely the sentence is to be grammatical.

Our system looks through the list of different length compressions and selects the shortest compression with an MFQ value less than a specified threshold (for our 3-gram model, we used a threshold of 1.14). This compression is returned as the simplified statement extracted from S for the answer A.

D. Results

We have not yet been able to conduct a test of the compression system, because testing the grammaticality of the generated compressions and their coverage of the set of possible simplified statements requires the use of a large number of human judges. However, the basic functionality of the compression system can at least be demonstrated with some sample outputs from the compressor. In each of the outputs, the sentence and answer are specified at the top, and then each row contains a potential compression and its MFQ value (labeled as 'S' on the readout).

Figure 1 shows a perfect compression of the sentence *Bill drives his car to the park every morning*. In the list of generated compressions, the one ultimately selected is clearly the shortest grammatical compression of the input sentence.

The output in Figure 2 is still grammatical, but there is one shorter compression in the list that is also grammatical, but was not identified by the program. This is because the MFQ value for *Bill drives to the park every morning* was 1.136, which is slightly less than the threshold of 1.14. Examples like this make it clear that tuning the gramamticality threshold is very important.

Figure 3 is not grammatical, but there is a grammatical sentence in the compression list only one word longer than the

```
Sentence: Bill drives his car to the park every morning.
Answer: Bill
Compressed Sentences:
Compressions:
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives car to the park every morning .      S = 1.1363540097281664
Bill drives to the park every morning .          S = 1.1363540097281664
Bill drives car to the park .    S = 1.1363540097281664
Bill drives his car .    S = 1.0533242202705644
Bill drives car .        S = 1.1043456643485705
Bill drives .    S = 1.1702534936017774
Best Compression: Bill drives car .
```

Fig. 3. Simulation Results

compression that was selected. The chosen compression had a higher MFQ score than the true shortest grammatical sentence, but because it was shorter, it was chosen nonetheless.

VIII. Conclusion

The key principle around which our system is built is that selecting the answer at the beginning of the QG process and using them to guide SSE can improve the coverage of the system. We implemented the machine learning-based approach for answer selection used by Becker et al. [5] and developed a way to compress a sentence while leaving a specified answer phrase intact. Although we have not yet been able to perform large scale tests on this system where the output is rated by human judges, we have generated some good output sentences. Once our implementation is perfected and tuned, we will perform more powerful and complete tests.

This system will soon be integrated with Jacob Zerr's Part-of-Speech Pattern Matching system for direct declarative-to-interrogative transformation to produce a full, functional, QG system.

References

[1] Kunichika, Hidenobu, Tomoki Katayama, Tsukasa Hirashima, and Akira Takeuchi. "Automated question generation methods for intelligent English learning systems and its evaluation." In Proceedings of ICCE2004, pp. 2-5. 2003.

[2] Wolfe, John H. "Automatic question generation from text-an aid to independent study." In ACM SIGCUE Outlook, vol. 10, no. SI, pp. 104-112. ACM, 1976.

[3] Heilman, Michael, and Noah A. Smith. "Good question! statistical ranking for question generation." In Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 609-617. Association for Computational Linguistics, 2010.

[4] Heilman, Michael, and Noah A. Smith. "Extracting simplified statements for factual question generation." In Proceedings of QG2010: The Third Workshop on Ques-tion Generation, p. 11. 2010.

[5] Becker, Lee, Sumit Basu, and Lucy Vanderwende. "Mind the gap: learning to choose gaps for question generation." In Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 742-751. Association for Computational Linguistics, 2012.

[6] Knight, Kevin, and Daniel Marcu. "Statistics-based summarization-step one: Sentence compression." In AAAI/IAAI, pp. 703-710. 2000.

[7] Cohn, Trevor, and Mirella Lapata. "Sentence Compression as Tree Transduction." Journal of Artificial Intelligence Research 34 (2009): 637-674.

[8] Filippova, Katja, and Michael Strube. "Dependency tree based sentence compression." In Proceedings of the Fifth International Natural Language Generation Conference, pp. 25-32. Association for Computational Linguistics, 2008.

[9] Stanford NLP Toolkits, http://nlp.stanford.edu/software.

[10] Holmes, Geoffrey, Andrew Donkin, and Ian H. Witten. "Weka: A machine learning workbench." In Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on, pp. 357-361. IEEE, 1994.

[11] De Marneffe, Marie-Catherine, and Christopher D. Manning. "Stanford typed dependencies manual." URL http://nlp. stanford. edu/software/dependencies manual. pdf (2008).
[12] Le Cessie, Saskia, and J. C. Van Houwelingen. "Ridge estimators in logistic regression." Applied statistics (1992): 191-201.
[13] Clark, Alexander, Gianluca Giorgolo, and Shalom Lappin. "Statistical representation of grammaticality judgements: the limits of n-gram models." CMCL 2013 (2013): 28.
[14] Ide, Nancy, and Catherine Macleod. "The american national corpus: A standardized resource of american english." In Proceedings of Corpus Linguistics 2001, vol. 3. 2001.
[15] Berkelaar, Michel. "lpSolve: Interface to Lp solve v. 5.5 to solve linear/integer programs." R package version 5, no. 4 (2008).
[16] Pauls, Adam, and Dan Klein. "Faster and smaller n-gram language models." In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1, pp. 258-267. Association for Computational Linguistics, 2011.

Extreme Value Theory and Visual Recognition

Rachel Moore
Department of Computer Science
University of Colorado, Colorado Springs

***Abstract* – The fields of machine learning and psychology have begun to merge, particularly in the subject of vision and recognition. This paper proposes an experiment on human recognition and categorization, using arbitrary images as stimuli. The data will be fitted to an Extreme Value Theory based model, which we hope will give clearer incite into the ways humans categorize novel information.**

***Index Terms* – Recognition, Category Learning, Machine Learning, Extreme Value Theory, Cognitive Psychology.**

I. Introduction

Training set selection is one of the most crucial steps in machine learning. If one wishes for a machine to identify images of apples, only providing images of oranges during training is, in most cases, counter productive. Traditionally, training sets have been selected so that was a wide array of data, so that the training set would closely match a gaussian, or normal, distribution [1]. However, this can become expensive, particularly in tasks that require labeled data for supervised learning. An extensive amount of data is also needed, as smaller sets are less likely to have a normal distribution, which can cause high variance responses and inconclusive results [1].

There have been many advances in the area of training data selection, but there is still a need for methods that select the best training data from small datasets. Simple methods like bootstapping can be used to generate new data in cases of small data sets. Many of these methods do not work with certain learning models [1] [2]. To understand what makes an effective training set, it is important to study how training affects the ultimate categorization. One way of doing this is to study recognition and categorization in humans.

The ability to categorize is one of the most crucial skills we develop as children. Despite its importance, the way we organize information is still a mystery. There are many models of categorical learning in psychology, and more are in development. Studies, such as the one done by Hsu and Griffiths [3] (discussed in Section II), have given some insight into the category learning process, and have yielded interesting results. However, the Gaussian models currently being used on these types of experiments are not capturing the extremes in the data, or the participants' bias towards one category or another. From our research, Extreme Value based models in machine learning have been shown to be a better predictor of human response frequency than Gaussian models. In summary, there are 4 main contributions of this paper:

- Extreme Value Theory and its application.
- Empirical evaluations and metrics for this research.
- Experimental results
- The future of this work.

II. Background

In this section, we discuss Extreme Value Theory (EVT) and its applications, as well as studies involving categorization and EVT modeling. We will also explore psychological research involving categorical learning, which will be the basis for our experiments.

A. Extreme Value Theory

The extreme value theorem states that a function with a continuous and closed interval will have a minimum and maximum value [4] [5]. EVT has been implemented as a statistical model in many different fields of research. Hugueny, Clifton, and Tarassenko [6] used EVT as the basis to create a new model for intelligent patient monitors. The current monitors they reviewed set off false alarms constantly, to the point that hospital staff ignored them. The model they proposed would be less likely to do this, as the EVT-based model would be able to differentiate between truly non-extreme changes in vitals and clear abnormality. EVT has also been used in machine learning to normalize recognition scores [7], which may skew distributions due to outliers.

This research seeks to establish a new EVT-based model of visual recognition and categorization. Particularly, this model may be instrumental for tasks that wish to replicate human information processing. There are three types of extreme value distributions:

Type 1, Gumbel-type distribution:

$$PR[X \leq x] = exp[-e^{x-\mu/\sigma}]. \tag{1}$$

Type 2, Fréchet-type distribution:

$$PR[X \leq x] = \begin{cases} 0, & x < \mu, \\ exp\left\{-\frac{x-\mu}{\sigma}^{-\xi}\right\} & x \geq \mu. \end{cases} \tag{2}$$

Type 3, Weibull-type distribution:

$$PR[X \leq x] = \begin{cases} exp\left\{-\frac{x-\mu}{\sigma}^{\xi}\right\}, & x \leq \mu \\ 0 & x > \mu \end{cases} \tag{3}$$

where μ, $\sigma(> 0)$ and $\xi(>$s $0)$ are the parameters [5].

EVT-based models can be used as replacements for Binary and Gaussian models, as EVT-models are able to include multiple classes, and do not rely heavily on norms (see Fig. 1). This can also be helpful in the case of training set selection. For example, say there is a set images of apples that need to be categorized into 2 groups: green granny smith and red

delicious. While the first and last apple groups have green and red skin tones, respectively, with slight variations in color. However, in this set of apples are a few fuji apples, whose colors range from ruddy green to orange red, and might be categorized into either of the other apple groups. To make the best predictions on which category each apple belongs to, we can use the EVT to find the apples at the groups' decision boundaries, i.e. the most and least red red delicious apples, and the most and least green granny smith apples. From this we can create a training data set. When these clear decision boundaries are known, anything that lies outside of them, say a greenish red fuji apple, can be categorized as a true outlier or part of a third class in the data.

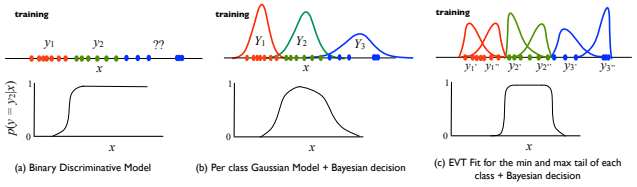

Fig. 1. Example of data selected with EVT. Courtesy of Boult

B. Prior Research in Human Visual Recognition

In a two part study, Cohen, Nosofsky, and Zaki [8] examined the effects of class variability on categorization. They hypothesized that the generalized context model (GCM), used to calculate the probability that an item will be categorized into one class or another, would substantially underestimate the degree to which participants would classify stimuli into the categories of high variance (we discuss GCM in greater detail in Section V). They found that the middle stimuli (items that were in between the low variance and high variance classes) were classified into the higher variance category, with the probability of up to .73. The GCM estimated the probability to be as low as .35, significantly below what was indicated by the data.

Hsu and Griffins [3] conducted a study in which the participants were taught two alien "languages", consisting of simple images of line segments. Class A had short, low variance line segments, which only differed slightly from one another. Class B had much longer, high variance line segments, in which each line's length was very different from the others. Participants were put into either a generative learning condition or a discriminative learning condition, which varied by the way the training images were presented. In the generative condition, two different cartoon aliens would appear on the screen to indicate which line belonged to which tribe's language. In the discriminative condition, one cartoon alien appeared as a single translator, indicating which language was language was on the screen. After training, participants were shown line segments that were between the lengths of the low and high variance classes and asked to categorize them.

As with Cohen, Nosofsky, and Zaki's [8] study, the results showed that the participants had a strong bias toward the high variance class (Class B), clustering the middle stimuli with the more diverse lines. They found that their Gaussian-based model did not fit their data accurately, and therefore wondered if the Gaussian assumption did not reflect this type of human recognition.

III. Empirical Evaluations

In this section, we discuss our experimental designs, as well as the metrics and technical approach of our study.

In a pilot study, Boult et. al [1] analyzed the data from Hsu and Griffins' [3] study using an EVT model. Because of the bias toward the high variance class, they believed that an EVT-based model would match human data in a more concise way (see Fig. 2) than Gaussian models.

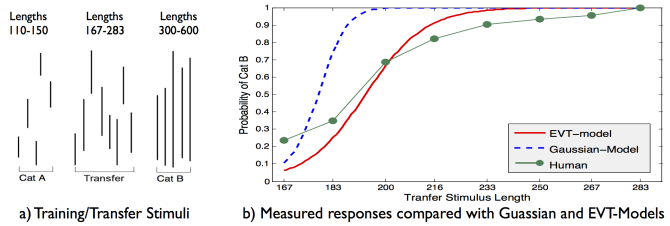

Fig. 2. Comparison of Gaussian and EVT-based models with human data. Courtesy of Boult.

For the second half of this pilot study, we have collected our own data. Our experiment expanded on Hsu and Griffins' [3] study, but used EVT-based models. We hope our model will paint a clearer and more accurate picture of the way humans categorize unfamiliar stimuli. Another possible extraneous factor in Hsu and Griffiths' [3] study is the way the alien interpreters (the categories) were presented. Those in their generative group were clearly shown when the category had changed, as the aliens changed depending on the sign. In the discriminative group, there was a single alien which never left the screen, and so the participants may not have noticed the sign change. We have duplicated some of their stimuli to test for this factor (see Fig. 3).

A. Metrics and Design

Our research uses models based on the extreme value distributions. Scheirer et al. [7] define extreme value distributions as "... limiting distributions that occur for the maximum (or minimum, depending on the data) of a large collection of random observations from an arbitrary distribution." In the case of visual recognition and categorization in humans, instead of removing the outliers or having them skew the results, one can normalize them, possibly allowing for a better fitted prediction.

For our experiment, we referred to the generalized extreme value (GEV) distribution, or the combined Gumbel, Frechet, and Weibull distributions. GEV is defined as

$$GEV(t) = \begin{cases} \frac{1}{\lambda} e^{-v^{-1/k}} v^{-(1/k+1)} & k \neq 0 \\ \frac{1}{\lambda} e^{-\left(x+e^{-x}\right)} & k = 0 \end{cases} \quad (4)$$

[1]Personal Communication

where x is equal to $\frac{t-\tau}{\lambda}$, v is equal to $(1+k\frac{t-\tau}{\lambda})$, and k,λ, and τ are the shape, scale, and location parameters.

For stimuli, we created a set of 2 dimensional Non-uniform rational B-spline (NURBS) shapes. NURBS are mathematically based shapes, and can be manipulated through functions and interpolation. In a NURBS parametric form, "... each of the coordinates of a point on a curve is represented separately as an explicit function of an independent parameter" [9]

$$C(u)=(x(u),y(u)) \qquad a\leq u\leq b \tag{5}$$

Where "$C(u)$ is a vector-valued function of the independent variable u", which is within the interval [a,b] (usually normalized to [0,1]) [9]. The NURBS we created look similar to ink blots. Each group of images had points that were interpolated to create a set with two clear classes, and another that was somewhere between those two classes (see Fig. 3 and 4). Four groups of shapes were used, and each group contained 17 images. These stimuli acted as distractor tasks. The other

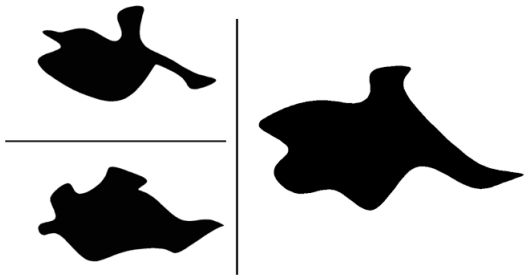

Fig. 3. Example of images duplicated from our NURBs stimuli.

stimuli were white lines of varying lengths, place inside of a black circle. Each set had a total of 18 images. These were based on the stimuli in Hsu and Griffiths' [3] study (see Fig. 5). These stimuli were placed into 3 conditions: generative, discriminative, and enhanced tails.

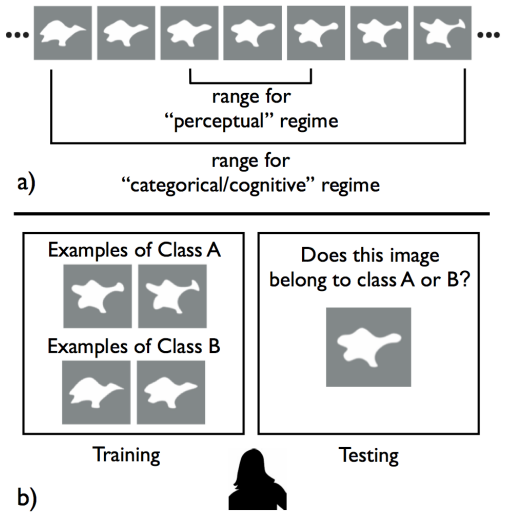

Fig. 4. Example of images from training classes A and B, and a testing image. Courtesy of Boult.

For the experiment itself, we used a program called PsychoPy, version 1.80 [2]. PsychoPy is open source psychophysics software, developed by Piece [10].

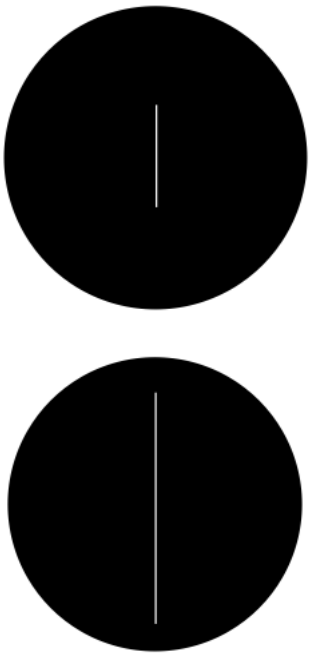

Fig. 5. Example of images duplicated from Hsu and Griffiths' [3].

There were 8 participants total, some of whom took the experiment on multiple occasions. From them, we gathered 30 trials for each of the 3 conditons. Our participants were asked to categorize a series of images into one of two groups, Group 0 or Group 1, and told that there was to be a training component where they would be shown the images and their respective categories, and a testing component where they would label the images themselves. The groups had a separate training set or training style, and testing set. For every group, the participants were trained on the 10 shapes at extrema, 5 from each tail. They were then tested on those same shapes, along with the shapes from the middle of the set, some of which were repeated to make up a total of 20 shapes per training. All trials were repeated, for a total of 20 trial blocks.

IV. Data Analysis

We recorded which middle stimuli were categorized into Group 0 or 1, and the frequency to which these stimuli were placed in these groups (see Fig. 9). The EVT-based model was fitted to the data. It reflected the biases the participants have in categorization, as it did in the pilot study.

A. Factor 1: The Generative Condition

In the generative condition, participants were shown the training stimuli. The training set consisted of lines that were in high variance and low variance categories. The low variance lines were 110, 120, 130, 140, and 150 pixels in length, while

[2]Accessed here: http://www.psychopy.org

the lines in the high variance category were 300, 375, 450, 525, and 600 pixels in length. During training, a box appeared 0.5 sec before before the stimuli, indicating which group the image belonged to. After the stimulus appeared, both the box and the image remained on the screen for 1.5 sec. This was repeated for all 10 stimuli in the training set.

The testing set was comprised of the training set, as well as a set of "middle" stimuli with line lengths of 167, 183, 200, 216, 233, 250, 267, and 283 pixels. The probability that each of these lines would be categorized into the high variance category was 0.17, 0.20, 0.33, 0.4, 0.53, 0.70, 0.77, 0.90, respectfully. The data fit our model well, with the main deviation being at line length 267 (see Fig. 6). Out of the three conditions, this condition was the closest fit to our EVT-based model.

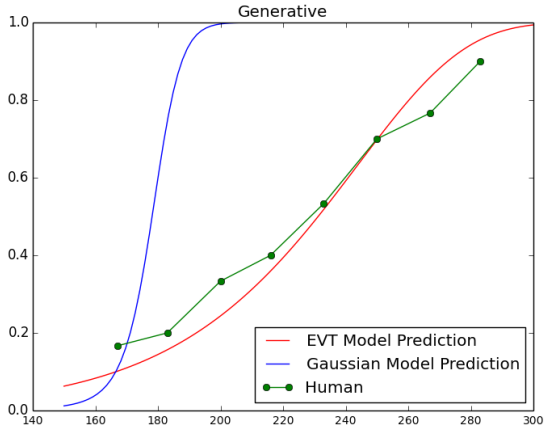

Fig. 6. Comparison of our EVT model with human data for the generative condition.

B. Factor 2: The Discriminative Condition

In the discriminative condition, participants were shown the same training and testing stimuli as the generative condition. However, the indicator box remained on the screen throughout the training session, with only the text changing. Each image still remained on the screen for 1.5 sec. For this condition, the probability that each of the middle stimuli would be categorized into the high variance category was 0.13,0.23, 0.30, 0.37, 0.63, 0.67, 0.83, and 0.90, respectively. The data for this condition also fit our model well, with the main deviation being at line length 233 (see Fig. 7).

C. Factor 3: The Enhanced Tails Condition

The final training set of lines contained the set of low variance lines as the generative and discriminative conditions, but the high variance lines had an elongated tail, with pixel lengths of 300, 375, 450, 600, and 800. The training set up was identical to that of the generative condition. For this condition, the probability that each of the middle stimuli would be categorized into the high variance category was 0.00,0.10, 0.20, 0.20, 0.43, 0.53, 0.60, 0.77, respectively. This shows

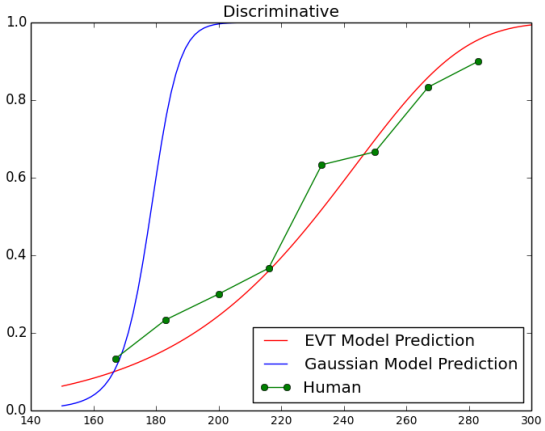

Fig. 7. Comparison of our EVT model with human data for the discriminative condition.

a shift towards the low variance category. We trained the model on the same set of training data used for the previous conditions for a better visual representation of the bias towards the low variance category (see Fig. 8).

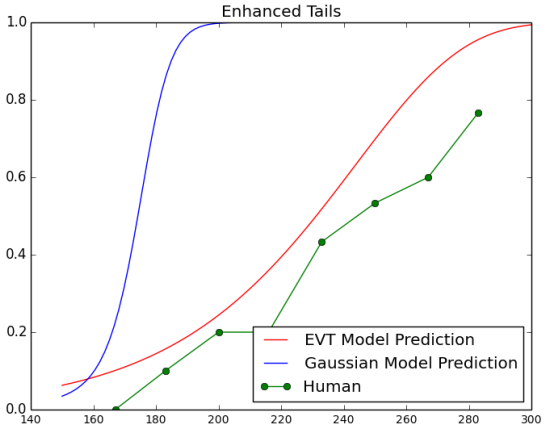

Fig. 8. Comparison of our EVT model with human data for the enhanced tails condition.

D. Comparison

In this section, we will compare the generative, discriminative, and the enhanced tails conditions, and discuss the statistical analysis for the experiment. Fig. 9 is a summary of the probabilities of each condition. The error bars indicate the variance of each line lengths probability. Both the generative and discriminative categories had similar trends. The variance of the generative, discriminative, and enhanced tails conditions were 0.073, 0.083, and 0.072, respectively.

V. Future Work

For our future research, we will incorporate measurements from the NURBS shapes. Because these shapes are mathematically based, the stimuli's dimensions can be easily applied to

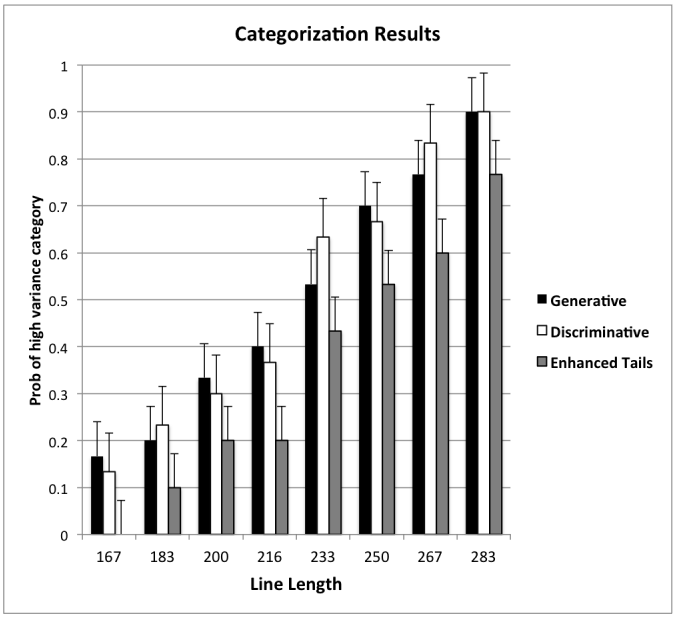

Fig. 9. Probability of categorization of middle stimuli into the high variance category for the generative, discriminative, and enhanced tails conditions.

probability models. One such model, which was mentioned in Section II, is the generalized context model (GCM), which states that "For the case of two categories A and B, the probability that a given stimulus X is classified in category A is given by

$$P(A|X) = \frac{\beta_a \eta_{XA}^{\alpha}}{\beta_A \eta_{XA}^{\alpha} + (1-\beta_A)\eta_{XB}^{\alpha}} \tag{6}$$

where β_A is a response bias toward category A and η_{XA} and η_{XB} are similarity measures of stimulus X toward all stored exemplars of categories A and B, respectively" [11].

Because our stimuli are so diverse, we plan to make at least one other variation on the current experiment. This may involve changing the task difficulty, the time length, or varying the amount of stimuli in the training sessions.

VI. Conclusion

This paper proposed a new EVT based model for visual recognition. For our purposes, we hope our model will prove to be consistent and accurate in predicting human recognition and categorization. If it is shown to be both of these things, the model could be used to select training sets for machine learning more efficiently, as EVT-based models focus on training data at the extremes, which may cut down on costs of supervised learning. We have seen that EVT-based models can be applied to both generative and discriminative learning situations. We believe that EVT-based models should also be insensitive to the difference between categorical and perceptual learning. With more research, our model may be applied to other human learning tasks, not just visual recognition.

Acknowledgement

I would like to acknowledge Dr. Walter Scheirer, Dr. David Cox, and the team of scientists at Harvard University, who have greatly contributed to this project. I would also like to thank Dr. Terrance Boult, Dr. Lori James, Dr. Kristen Walcott-Justice, and Dr. Jugal Kalita for their invaluable guidance. This project is being supported by NSF REU Grant 1359275.

References

[1] E. Alpaydin, *Introduction to machine learning*. MIT press, 2004.
[2] I. H. Witten, E. Frank, and A. Mark, "Hall (2011)." data mining: Practical machine learning tools and techniques," 2011.
[3] A. S. Hsu, T. L. Griffiths *et al.*, "Effects of generative and discriminative learning on use of category variability," in *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, 2010, pp. 242–247.
[4] M. K. Nasution, "The ontology of knowledge based optimization," *arXiv preprint arXiv:1207.5130*, 2012.
[5] S. Kotz and S. Nadarajah, *Extreme value distributions: Theory and applications*. World Scientific, 2000, vol. 31.
[6] S. Hugueny, D. A. Clifton, and L. Tarassenko, "Probabilistic patient monitoring with multivariate, multimodal extreme value theory," in *Biomedical Engineering Systems and Technologies*. Springer, 2011, pp. 199–211.
[7] W. Scheirer, A. Rocha, R. Micheals, and T. Boult, "Robust fusion: extreme value theory for recognition score normalization," in *Computer Vision–ECCV 2010*. Springer, 2010, pp. 481–495.
[8] A. L. Cohen, R. M. Nosofsky, and S. R. Zaki, "Category variability, exemplar similarity, and perceptual classification," *Memory & Cognition*, vol. 29, no. 8, pp. 1165–1175, 2001.
[9] L. Piegl and W. Tiller, "The nurbs book," *Monographs in Visual Communication*, 1997.
[10] "Psychopy—psychophysics software in python," *Journal of Neuroscience Methods*, vol. 162, no. 1–2, pp. 8 – 13, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165027006005772
[11] T. Smits, G. Storms, Y. Rosseel, and P. De Boeck, "Fruits and vegetables categorized: An application of the generalized context model," *Psychonomic Bulletin & Review*, vol. 9, no. 4, pp. 836–844, 2002.

Using Hidden Markov Models and Spark to Mine ECG Data

Jamie O'Brien
Saint Mary's College of Maryland
St. Mary's City, Maryland
Email: jcobrien@smcm.edu

***Abstract*—New potential risk factors for cardioembolic strokes are being considered in the medical community. The presence of these factors can be determined by reading an electrocradiogram (ECG). Manual ECG analysis can take hours. We propose combining accurate Hidden Markov Model (HMM) techniques with Apache Spark to improve the speed of ECG analysis. The potential exists for developing a fast classifer for these risk factors.**

I. Introduction

The proliferation of medical data in modern hospitals provides a rich environment for data mining. Electrocardiograms (ECGs) provide a wealth of information that can be used to diagnose cardiovascular diseases (CVDs). In Agarwal and Soliman [1], it is suggested that the ECG can be used to detect cardioembolic stroke risk factors. Aside from those factors included in the Framingham Risk Score, emerging factors include:

1) cardiac electrical/structural remodeling,
2) higher automaticity,
3) heart rate & heart rate variability.

Currently, the manual analysis of ECG patterns is time-consuming. It can take several hours to complete Acharya et al [2].

II. Problem Statement

We want to find a better method of detecting the emerging risk factors listed in Section I. We want to combine an effective Hidden Markov Model (HMM) classifier for ECGs with the fast, distributed processing power of Apache Spark.

A. Atrial Fibrillation—A Verified Stroke Risk

In atrial fibrillation (AF), the heart's atrial walls do not produce an organized contraction—instead, they quiver [3]. Even though AF is a component of the Framininghamm Stroke Risk Score [4], it is often undetected; the condition has evaded detection even in patients known to have paroxysmal atrial fibrilation. The detection rates may vary depending on the algorithms used, but seem to improve with longer monitoring times [5]. The difficulty of accurately detecting AF motivates the search for additional stroke risk factors.

B. Hidden Markov Models

Hidden Markov Models (HMMs) have been used with great effect in classifying ECGs. Andereão et al were able to demonstrate an accuracy of 99.97% in detecting the QRS complex of the heartbeat [6]. Their approach was to create a general model of the heartbeat, and then tune the model to each individual by using data from the first 20 seconds of their ECG. The general model of the heartbeat was composed of discrete states representing the P, Q, R, S, and T waves, the PQ and ST intervals, and the isoline. Andreão et al's work was able to detect premature ventricular contractions (PVCs). We hope to use a similar model for detecting ectopic beats and bundle blocks.

C. Apache Hadoop

Hadoop pairs a high-bandwidth distrubted file system with MapReduce programming Svachko et al [7]. This allows for a task to be broken up across many computers, the components calculated independently, and the results collected. In this way, Hadoop may improve the performance of signal processing tasks. This performance improvement is the core of the Cloudwave system described in Jayapandian et al [8]. The authors of that work used Hadoop to process multimodal bioinformatic data. A stand-alone machine was able to process 10 signals in 22-36 minutes. Their Hadoop cluster was able to process the same data in 4-6 minutes.

D. Apache Spark as a replacement for Hadoop MapReduce

While the Cloudwave system described in Jayapandian et al [8] is impressive, the highly iterative nature of data mining tasks may cause significant overhead under Hadoop's MapReduce architecture. Apache Spark avoids this issue by using the concept of resilient distributed datasets (RDDs). These RDDs can be cached in memory. This make the data available for iterative and parallel programming alike without having to be constantly reloaded Zaharia et al [9].

III. Method

The in-progress research explores the applicability of Hidden Markov Models on ECG readings, with the goal of detecting the emerging factors mentioned in [1]. Here we note the strategy for constructing our system.

We obtained ECG signals from the QT Database (QTDB), using the WaveForm Database application suite. We also

obtained two sets of annotations: one, marked `atr`, contains annotations that marks beats as normal, or as having some abnormality (preventricular contraction, for instance); the second set of annotations, marked `pu0`, contains waveform markers, such as p, t, and N (for normal qrs complex). Any records from the QTDB that did not contain annotations from `atr` were excluded, as we would not be able to verify our results against them.

We transformed the `pu0` annotations to provide clearer information. The standard for annotating waves is to open a wave with a paren, note the wave, and then close it with a paren. For instance, the p wave would be marked by the annotations (, p,). We wrote a script to process these annotations, and change them to the form pBegin, p, pEnd, so that all parenthesis were removed. This meant that the annotations themselves could now become a set of states for use in a Hidden Markov Model. The states derived from the annotations were: pBegin, p, pEnd, q, r, q, tBegin, t, tEnd, unknownBegin, and unknownEnd.

However, we found that it was not practical to simply map the states annotated in `pu0` to the beat classifications annotated in `atr`. When attempting to map the state sequence to PVC, for instance, no significant correlation could be found in a sample of PVC beats. We hypothesized that the duration of the states was also significant. It may be necessary to mark states as being faster or slower than normal. The duration between, for instance, pBegin and pEnd could tell us if the p wave were of normal duration.

With this in mind, we are determining a way to map the ECG signal itself to states. In [10], we find an algorithm for decomposing ECG signals into line segments. This algorithm moves a dynamically-sized window along the ECG signal. The window checks the distance between the endpoints and every point in-between, using normalized distances where needed. We can adjust the allowed error to accomodate noisy signals.

We modify this algorithm to output a list of 4-tuples of the form (starting point, length, mean of segment, standard deviation of segment). This converts the continuous ECG signal into a set of data points. We must then convert this set of data points into states that correspond with the waveforms of the heart beat: the p wave, qrs complex, t wave, and the intervals between them.

IV. The Classification Process

We begin by slicing an ECG signal between its R-R intervals. We then take a slice and segment it using the algorithm described in [10]. These segments are then labeled by the state they most match, using a decision tree. The progression of states is treated as an observation, and fed into the HMM to determine which beat type most accurately matches the observation.

V. Further Work

This work will not be complete until the HMM itself is built and can be tested. In anticipation of this, we have separated the QTDB into a training set comprising approximately 80% of the annotated data, and a testing set with the remaining approximately 20%. The training set is composed of five sub-groups, each approximately 20% of the size of the training set. We intend to use these sub-groups for cross-validation.

After the model is built and its performance is evaluated, we can begin the construction of the Apache Spark implementation of the model. The purpose of this will be to compare the performance of the Spark implementation against the locally-run implementation. The parameters for this experiment will be determined when the HMM itself is complete.

VI. Conclusion

This research may provide a effective method for detecting the emerging risk factors for a cardioembolic stroke mentioned in section I. This would assist researchers who are investigating these risk factors.

Acknowledgment

We would like to thank the National Science Foundation (NSF) for their generous grant, and the University of Colorado, Colorado Springs for hosting the Research Experience for Undergrads (REU) program.

References

[1] S. Argwal and E. Soliman, "Ecg abnormalities and stroke incidence," 2013. [Online]. Available: http://www.medscape.com/viewarticle/808752

[2] R. Acharya, A. Kumar, P. Bhat, C. Lim, N. Kannathal, and S. Krishnan, "Classification of cardiac abnormalities using heart rate signals," *Medical and Biological Engineering and Computing*, vol. 42, no. 3, pp. 288–293, 2004.

[3] F. H. Martini, J. L. Nath, and E. F. Bartholomew, *Fundamentals of Anatomy and Physiology (9th Edition)*. Benjamin Cummings, 1 2011.

[4] F. H. Study, "Stroke," https://www.framinghamheartstudy.org/risk-functions/stroke/stroke.php, (Visited on 07/14/2014).

[5] M. A. Rosenberg, M. Samuel, A. Thosani, and P. J. Zimetbaum, "Use of a noninvasive continuous monitoring device in the management of atrial fibrillation: a pilot study," *Pacing and Clinical Electrophysiology*, vol. 36, no. 3, pp. 328–333, 2013.

[6] R. V. Andreão, B. Dorizzi, and J. Boudy, "Ecg signal analysis through hidden markov models," *Biomedical Engineering, IEEE Transactions on*, vol. 53, no. 8, pp. 1541–1549, 2006.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[8] C. P. Jayapandian, C.-H. Chen, A. Bozorgi, S. D. Lhatoo, G.-Q. Zhang, and S. S. Sahoo, "Cloudwave: Distributed processing of big data from electrophysiological recordings for epilepsy clinical research using hadoop," in *AMIA Annual Symposium Proceedings*, vol. 2013. American Medical Informatics Association, 2013, p. 691.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[10] A. Koski, "Modelling ecg signals with hidden markov models," *Artificial intelligence in medicine*, vol. 8, no. 5, pp. 453–471, 1996.

Question Generation using Part of Speech Information

Jacob Zerr, Texas A&M University

***Abstract*—When testing students on knowledge from a story or article, a human must interpret the text to generate English questions. The difficulty in automating this process is producing a computational algorithm that can fully account for the syntactic and semantic complexities of human languages. Most approaches use big, costly semantic tools such as WordNets to achieve their semantic accuracy and rule-based approaches to achieve their syntactic accuracy. We propose an approach for generating knowledge-testing questions from textual English using machine learning to use part of speech pattern matching without using any large semantic tools.**

I. Introduction

Many attempts have been made to automate interpreting natural human languages, most of which have taken some small sub-problem and attempted to solve it. One such sub-problem is manipulating sentences to create question-answer pairs from a sentence, which we will be addressing. The main difficulty of question generation is that the method must maintain both semantic and syntactic accuracy. When formatting a question, we will need to change the structure of the sentence, add and remove words, change the tense or part of speech of words, or other complex operations. Moreover, through these operations we must keep the semantic integrity of the statement and select the correct answer to the resultant question. However, the applications of a proficient question generator could span domains from automated education tools to better AI conversation generation. We propose a new method of question generation that uses part of speech (POS) pattern matching based off of Inversion Transduction Grammars (ITG) from a sentence and question-answer pair corpus. We restrict our input to sentences containing one independent clause with the thought that this approach would work on any input if compressed first.

II. Problem Definition

Our input will be any collection of English sentences containing one independent clause. The sentences should be well formed and in correct English grammar for best results. The output will be a set of question-answer pairs that should be asking about the contextual knowledge of the original text. The output questions should also be grammatically correct. Here are a couple examples.

Jacob Zerr is participating in a National Science Foundation REU at the University of Colorado at Colorado Springs, Colorado Springs, CO 80918.
email: zerr2@tamu.edu

- John drove the car to work. → Who drove the car to work? John
- The pump is now operational. → Is the pump operational? Yes
- He waters the garden every day. → What does he do every day? waters the garden

III. Related Work

Question generation was brought to the attention of the natural language processing community by Wolfe [4] in 1976. He outlined the purpose and applications of a question generator and the potential challenges. Since then, many have produced question generators of a limited focus. Papasalouros [1] creates only multiple choice questions by producing a set of similar sentences where a key word has been replaced in the wrong selections. This reduces the complexity of the problem by avoiding interrogative sentence structure. Brown [3] focuses only on questions that test vocabulary and uses a WordNet to increase their question complexity without losing semantic accuracy. They also use part of speech (POS) tagging to maintain the syntactic accuracy of the question. Kunichika [2] provides the most general approach of all by dissecting both the syntactic and semantic structure of the original sentence before producing the question. After looking at both of these, their algorithm has a broad spectrum of questions it can generate about the original declarative sentence. However, this approach relies heavily on the accuracy of the interpretation of the sentence using tools like WordNets that may not be accurate in all cases. These are three representations of the current best solutions, none of which use machine learning. Our approach will rely heavily on a POS tagger for which we will be using the Stanford Parser outlined in Toutanove [9]. On a different note, Heilman [5] ranks generated questions which may be considered as a useful addition to our question generation process later on in our development.

IV. Inversion Transduction Grammars

Inversion Transduction Grammars are grammars that map two languages simultaneously and generally follow the format of a context free grammar. The main difference is the angle brackets in the grammar denote that the symbols should be read in left-to-right order for the first language and right-to-left for the second. This allows for the grammar to successfully map two languages with different part of speech orderings like SOV, SVO, or VSO languages. From there the lexicon has word pairs, one from each of the two languages, that should be direct translations of each other. This method uses basic

word-to-word translations and the fact that most languages use similar part of speech models, just in a different order, to achieve an accurate machine translation. Wu [6] explains these grammars in detail and shows how they can be used as an accurate form of machine translation. Both Goto [7] and Neubig [8] use these techniques to successfully perform machine translations between complex languages.

V. Our Approach

A. Producing POS Pattern Templates from the Corpus

The main approach that we will be pursuing to convert our declarative sentences to questions is through a POS pattern matching approach based off of ITGs. Though ITGs have mainly been used to convert a parsed sentence into another language, we will be using it to convert between declarative English and interrogative English. The difficulty in this process is that ITGs rely on the structure of the two sentences to be similar in all but ordering. However, there are structural parts of interrogative English that are not in declarative English and vice verse. Our approach will avoid this by ignoring the tree structure of the grammar and just map the movement of different phrases from the sentence to the question.

The first major step of processing our corpus instances is to identify the phrases that stay consistent in the transition from declarative sentence to question-answer pair. We do this by searching the instance for phrases of the exact same wording starting from the largest possible phrases and then incrementally decreasing the size until all of the common phrases have been identified. We call this process chunking. Figure 1 shows such an instance and the phrases that have been identified after chunking has been completed.

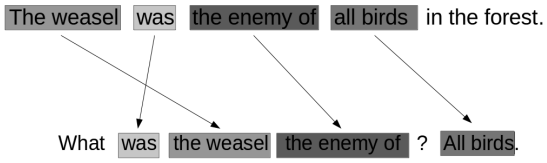

Figure 1. Sample of chunking the common phrases from an instance in our corpus.

Notice that there may be phrases in the sentence, question, or answer that are not in any of the other parts of the instance; in this case *in the forest* and *What*. These are kept and used by the algorithm in the process of finalizing the template; this will be explained later.

Our algorithm also can identify phrases that appear in all three parts of the instance as a part of the chunking process. This helps create templates for questions that quiz on adjectives of the sentence while still maintaining accuracy. Figure 2 is an example of such an instance where *plate* is repeated in all three parts of the instance to ensure that the answer makes sense.

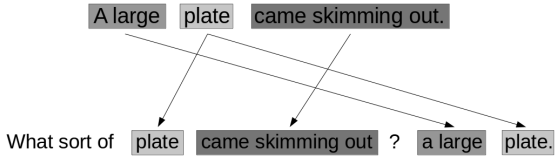

Figure 2. Sample of chunking an instance where a phrase is repeated in all three parts of an instance. This helps produce templates of questions that quiz on adjectives in the input sentence while still keeping accuracy.

Now that we have chunked our instances, we need to determine the part of speech of each of the phrases included in the sentence portion of the instance. For this we will be using the Stanford POS tagger [9]. There are two main approaches to attempting to tag these phrases with a POS: parsing it within the original context of the sentence or parsing it out of context. When parsing it out of context, we can conveniently get a single POS for the phrase. However, you forfeit accuracy with this method because the Stanford Parser solves ambiguities internally and it may return the wrong POS in an ambiguous case. For this reason, we chose to parse the phrase within the context of the original sentence. However, this is slightly more difficult, because we will now have to search the grammar parse tree of the whole sentence produced by the Stanford Parser. Our method for this was to search for the node of the tree that was the deepest ancestor of all of the words in the phrase. An example of this is shown in Figure 3. Here we can see that we identify the POS for *the enemy of* as a Noun Phrase in the context of the sentence *The weasel was the enemy of all birds in the forest*.

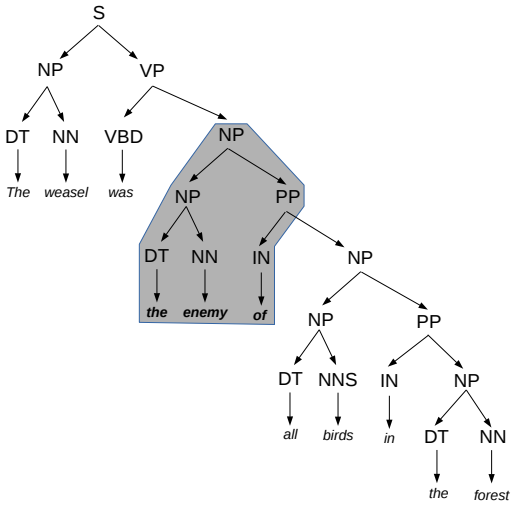

Figure 3. Our method for finding the POS of a phrase involves finding the deepest common ancestor of the words of the phrase. Here we can see *the enemy of* is being labeled as a Noun Phrase.

With this method of POS tagging, we then will label every phrase in the original sentence. This includes any phrases that

were not repeated in the question or the answer. The result of this process from the example used in Figure 1 is shown in Figure 4.

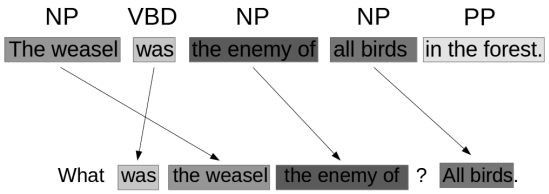

Figure 4. The example from Figure 1 with its sentence chunks labeled with their POS.

After the sentence chunks have been labeled, we drop all of the phrases that appeared in the sentence part of the instance. The phrases that only appeared in the question or answer are left as a part of the template. This is the final step of producing our POS template from an instance in our corpus. This process is completed for every instance in the corpus before we start trying to use these templates on our input sentences. Figure 5 shows this last step on our example.

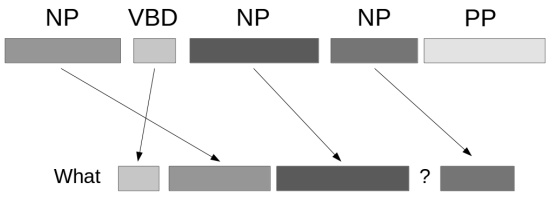

Figure 5. The final step of preparing the templates is drop all of the phrases that appeared in the sentence. Phrases that were just in the question or answer remain.

B. Generating Questions

Once we have converted the instances of our corpus into POS pattern matching templates, we can begin to try to fit input sentences into our templates. We do this by simply seeing if the input sentence can be divided into phrases that, when tagged with a POS, match the template. If we do find a match, we reorder the phrases by using the template's question-answer ordering to produce our question-answer pair. An example of a sentence fitting the template we produced above is in Figure 6.

An interesting question that arose from this pattern matching method is which type of POS tagging we would use for this part of the algorithm, in-context or out-of-context. Initially, it seemed clear that we should follow the same method we did in producing our corpus and use in-context. However,

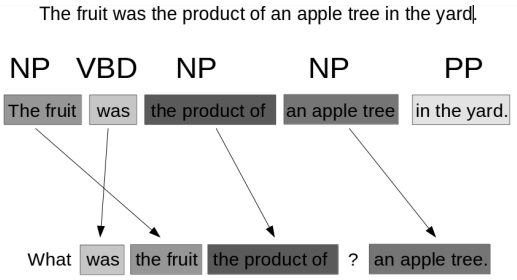

Figure 6. A sentence being matched to our example template and the question-answer pair it produced.

when experimenting with out-of-context we sometimes would produce a wrongful tag to a phrase that would fit a template. The expectation was that from an erroneous matching we would produce an inaccurate question, but this was not always the case. Figure 7 shows an input sentence matching to the template we produced above with *render* erroneously being labeled a Noun Phrase, however the produced question is accurate. We explored this question and our answer is discussed later in the results section.

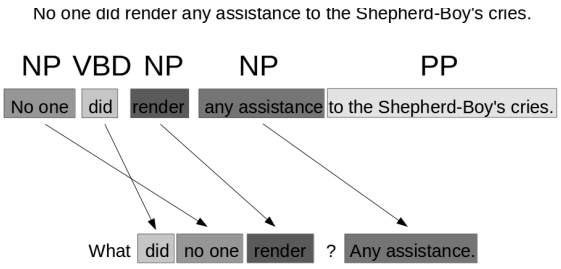

Figure 7. A sentence being wrongfully tagged and matched to our example template may still produce an accurate question-answer pair.

As a last note, if our algorithm can divide an input sentence to match a template more than one way, then it will produce a different question for each different legitimate divisions. An example of this is shown below in Figure 8.

An interesting observation on our method is that because we are simply reordering phrases, we keep the same vernacular of the original sentence. We have been operating in the domain of children's stories for this project and often times children stories will have odd wording that is not common vernacular anymore. These odd phrases will always be reflected in our output. The example in Figure 7 uses phrases like *render assistance* whereas most people would simply say *help*. This can be both a good and bad attribute of our approach. The good part is that our questions may contain slang or improper words of spoken English that takes our questions to a semantic level not normally achievable by a computer. However, it also can sometimes cause problems if these words are wrongfully

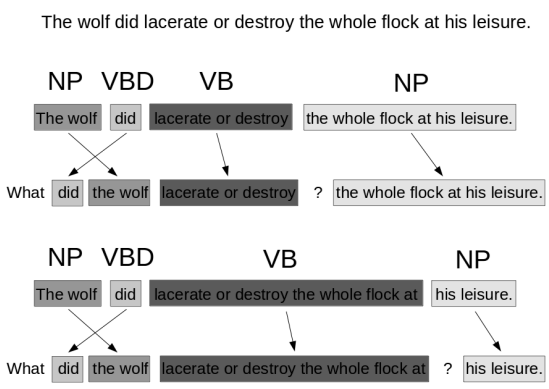

Figure 8. If a sentence can be divided in more than one way to match a template then the algorithm will produce a different question for each way.

tagged and will miss the factual tone of SAT-style questions that a user may want because of odd diction in the original sentence.

Also, it is important to note that we restricted our input to sentences containing one independent clause. This is necessary, because with additional clauses the accuracy of our POS tagging for phrases goes drastically down. For instance, if we divide our input in a way that a phrase is an entire clause it may be be given a POS label of Sentence, which is too general for our templates to produce good results. Typically, the larger the phrases become, the higher up the parse tree you will have to go for a deepest common ancestor, and the less specificity of POS tags we will have. Thus, in order to maintain reasonable levels of accuracy, we must limit our input to one clause sentences.

VI. Datasets

A large part of our work was producing and manipulating the corpus that defined our POS matching templates. This corpus is a collection of instances that map a sentence to a question-answer pair. Initially our corpus had 254 instances. From initial testing using this corpus we observed several things; a small corpus could produce an ample number of questions even with just one sentence inputted, often the same questions were produced more than once, and some instances in our corpus were better at producing accurate sentences than others. From these observations we decided to stop expanding the corpus and to actually start eliminating some instances.

Firstly, we had learned that some of the instances in our corpus were producing the same templates. Thus we went through and found the instances producing the duplicate templates and deleted them. An example of this was the template below had been produced 34 times. After 93 deleting instances that were producing duplicate templates our corpus had been reduced down to 161 instances.

$$\textbf{NP} \quad \textbf{VP} \quad \rightarrow \quad Who \quad \textbf{VP} \quad ? \quad \textbf{NP}$$

Secondly, we observed that some templates produced by our instances were much better at producing successful question-answer pairs than others. To test this theory we ran our corpus against some preliminary testing examples and confirmed this. Some templates were producing many consistently accurate questions, some produced very few questions, and others produced many inaccurate questions. Based on these results we eliminated any instance from our corpus that was producing questions at a twenty percent accuracy level or worse. This reduced our corpus down to just 129 instances.

Contrary to most forms of corpus-based machine learning, we found this corpus to be more than enough to produce a high number of different questions and a wide breadth of different types of questions. This is one of the largest advantages of our approach; it takes a comparatively tiny amount of data to get good results especially when compared to most of the other approaches that use large WordNets or other large semantic tools.

VII. Results

We analyzed and made improvements based off the syntactic and semantic accuracy of the output of our approach. The proportion of output instances that are grammatically correct, accurately quizzes the reader on the original knowledge, and has the corresponding answer will be our main metric of success. We used unbiased volunteer evaluators that judged each produced question-answer pair on whether they were syntactic and semantic accurate or not. Our evaluators are native English speakers that are in the process of attaining a Bachelors Degree, thus they have a firm knowledge of the English language. Our input were single independent clause sentences from children's stories such as *The Princess and the Pea*, *The Boy Who Cried Wolf*, and other such children stories.

A. POS Tagging Methods

We would first like to address the question we presented earlier on whether POS tagging on our input sentences should be done in-context or out-of-context. We would first like to note that we used only in-context tagging for creating our templates so that we could create accurately tagged templates. However, as we noted before, we produced accurate questions using both methods when tagging the input sentences. Based off of this we decided to experiment using four different methods for determining the POS to tag the sentence phrases: using the in-context tag (IC), using the out-of-context tag (OC), using a POS tag only if the two methods agreed ($IC \ \&\& \ OC$), and using either method to try to fit a sentence into a template ($IC \ || \ OC$). We tried these four methods on a 20 sentence input children's story.

Based off of the above results we chose to use the $IC \ || \ OC$ method for the rest of our work because of the greatly increased total solution production despite a very similar accuracy rate. Based off of the numbers above, this method was producing, on average, 7.25 accurate questions per inputted sentence.

Table I
POS TAGGING METHODS

Method	Accurate	Total	Percent
IC	94	160	58.75
OC	87	150	58.00
IC && OC	58	90	60.00
IC || OC	145	243	59.67

B. Overall Accuracy

For our final accuracy test, we used an input 48 sentences long from children's stories. We produced an output of 435 question-answer pairs. This means that we averaged 9.06 question-answer pairs per inputted sentence. This puts into perspective that our corpus, at 129 instances, really can perform like a large semantic tool despite its small size. The produced question-answer pairs were assessed by 4 evaluators that ranged the accuracy from 57.01% to 59.67% with an average of 58.36%. This result is also encouraging considering the previous work in this area. Brown [3] produced an accuracy rate from 52.86% to 64.52% and Papasalourous [1] produced an accuracy of 75% from his best strategy, but averaged an accuracy of 47.55% between all of their strategies. It is also interesting to note that both of these approaches were slightly more restricted in domain than our approach and they both relied on advanced wordnets in order to maintain semantic accuracy.

VIII. POSSIBLE FUTURE WORK

A possible extension of this work would be automatically analyzing the questions produced and ranking them in some way. Depending on the accuracy of the rankings we may be able to achieve a higher accuracy of the questions that are ranked in some top fraction of the produced questions.

IX. CONCLUSION

By using a relatively simple machine learning method with a small dataset, we were able to out-perform previous rule-based methods that used large semantic tools. If used with an accurate sentence compressor, we believe this method for generating questions would be extremely accurate and convenient. Our approach is also not domain-specific and thus can be used in anything from automated education tools to better AI conversation generation.

REFERENCES

[1] A. Papasalouros, K. Kanaris, and K. Kotis. "Automatic Generation Of Multiple Choice Questions From Domain Ontologies." In *e-Learning, pp. 427-434. 2008*.

[2] H. Kunichika, T. Katayama, T. Hirashima, and A. Takeuchi. "Automated question generation methods for intelligent English learning systems and its evaluation." In *Proceedings of International Conference of Computers in Education 2004, pp. 2-5, Hong Kong, China, 2003*.

[3] J. Brown, G. Frishkoff, and M. Eskenazi. "Automatic question generation for vocabulary assessment." In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing, pp. 819-826, Vancouver, Canada, Association for Computational Linguistics, 2005*.

[4] J. Wolfe "Automatic question generation from text-an aid to independent study." In *ACM SIGCUE Outlook, vol. 10, no. SI, pp. 104-112, ACM, 1976*.

[5] M. Heilman, and N. Smith. "Good question! statistical ranking for question generation." In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 609-617, Los Angeles, USA, Association for Computational Linguistics, 2010*.

[6] D. Wu. "Stochastic inversion transduction grammars and bilingual parsing of parallel corpora." In *Computational Linguistics 23, pp 377-403, 1997*.

[7] I. Goto, M. Utiyama, and E. Sumita. "Post-Ordering by Parsing with ITG for Japanese-English Statistical Machine Translation." In *ACM Transactions on Asian Language Information Processing (TALIP) 12, no. 4, 2013*.

[8] G. Neubig, T. Watanabe, S. Mori, and T. Kawahara. "Substring-based machine translation." In *Machine Translation 27, no. 2, pp 139-166, 2013*.

[9] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. "Feature-rich part-of-speech tagging with a cyclic dependency network." In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, Volume 1, pp 173-180, Edmonton, Canada 2003*.

Stencil Code Optimization for GPUs Through Machine Learning

Adam Barker
University of Colorado at Colorado Springs
abarker2@uccs.edu

***Abstract*—The microprocessor field today has begun to reach its limits as power and thermal constraints have been met and no longer can much leverage of increasing the processor's clock speed be achieved. Thus, much of the scientific and engineering community has shifted to using many-core architectures, such as GPUs, in order to do parallel computations. This paper focuses on the use of genetic algorithms to guide the optimization of stencil codes on NVIDIA's Compute Unified Device Architecture (CUDA) based GPUs and GPGPUs. In particular, we have implemented two separate stencil kernels (Jacobi 7 point and 27 point) in CUDA with each implementation parameterized for several optimiation parameters (thread blocking and loop unrolling factors). We then used a genetic algorithm to find optimal configurations for each kernel. This genetic algorithm is one part of our proposed solution of using an optimization framework incorporating the genetic algorithm to auto-tune automatically optimized stencil codes. Our results show that using a genetic algorithm to auto-tune stencil code optimizations is a valid approach of generating near-optimal configurations in a much more timely fashion than an exhaustive search.**

I. Introduction

As microprocessors reach the power wall, benefits of increasing the clock frequency are no longer achieveable as the cost to system stability and cooling is too much to warrant the increase in performance [1]. This has shifted the focus of the parallel community to many-core architectures, such as those found in Graphical Processing Units (GPUs), as they are comprised of a few hundred or thousand simple cores that are capable of performing highly-parallel computations with much more throughput than a typical multi-core system. However, developing parallel algorithms for GPUs can be no simple task for developers as developers must have a firm understanding of the underlying architecture and hardware properties in order to correctly write programs that correctly take advantage of these properties. Thus, there is a desire to develop a method to automatically apply optimizations to GPU programs in order to avoid the necessity of understanding the complexities of the hardware and architecture of the system.

Recently, in order to meet this desire, researchers have devloped several methods in order to automatically tune or automatically generate optimized codes for both GPUs and multi-core systems. However, as more optimizations are discovered, the search space the auto-tuner must search through grows to an amount where auto-tuning is no longer viable as the number of possible combinations of parameters becomes too large to effectively search through. This then sets the perfect stage for a machine learning application to predict the optimal code instead as it does not have to go through the entire search space, but rather make predictions based on previous results.

This work presents a method to use genetic algorithms in order to discover optimized configurations of parameterized CUDA stencil (nearest-neighbor) codes – a class of algorithms that typically work in structured grids to perform computations, such as finite-difference methods for solving parital differential equations, on a node within the grid by doing computations on the neighbors around the given node. Our work focuses on a simple 3D heat equation using two different stencil codes as the training set for a genetic algorithm to search through a search space of several thousand combinations of possible optimization parameters. Although stencil codes are important as scientific computations, they also provide a unique opportunity for hardware benchmarking as they are computationally simple and require a large use of memory, allowing for benchmarking of instruction-level and data-level parallelism [3]. These codes greatly benefit being run on GPUs as the parallel forms of these codes contain a great deal of instruction level parallelism which translates well to SIMD architectures, which are present on GPUs.

This research is the development of the optimal configuration generator portion of the framework detailed in Figure 1. The auto-optimization framework will be used to optimize existing stencil codes using machine learning in order to predict optimal tuning parameters that will be given to the optimizer which will apply these optimizations to the given stencil code and then output the optimized version of the given code. This is done so that developers can easily write unoptimized code for use in their programs and then run this auto-tuning framework on their code in order to use optimized code that correctly fits within their existing program.

In order to train the machine learning portion of the configuration generator, we implemented two stencil kernels to be used across three separate GPUs. The stencil codes we implemented were a 7 point and 27 point Jacobi iterative stencil codes and then parameterized the relevant optimizations that the genetic algorithms would find configurations for so that the fitness test for the genetic algorithm could change these parameters easily before compiling and running.

The optimization parameters considered for the generation of the search space that we used were the number of threads to use in the computation, and the distance to unroll the inner loop in our code. This inner-loop arises from our use

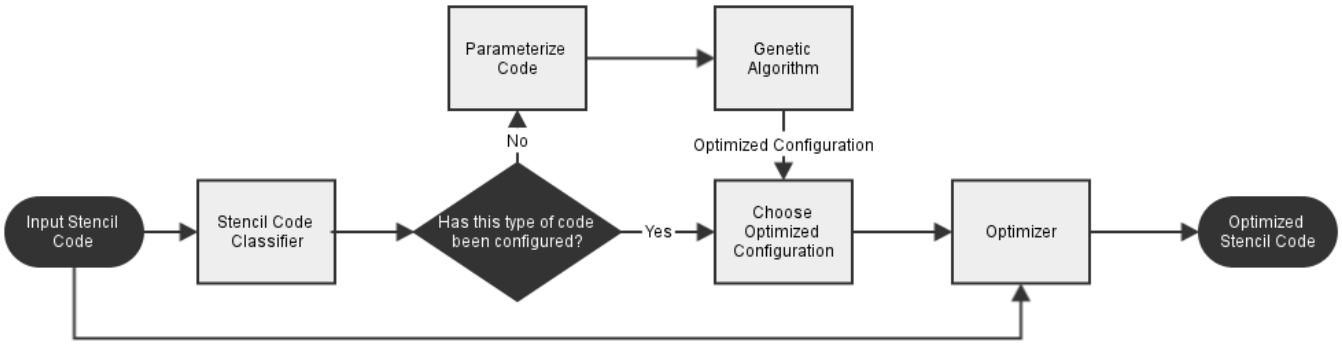

Fig. 1. Overview of auto-optimization framework

of 2.5D blocking, a thread blocking optimization that allows for threads to only be launched in a single plane of the 3D data, and then stream through the remaining axis as the computation goes on. This search space consists of 60 different thread configurations and 192 different loop unrolling configurations, giving us a search space of 11,520 possible combinations. Although the size of this search space is relatively small for most machine learning applications, one must consider the time it takes to compile the code as on our system, typical complilation time is 3 seconds, meaning that an exhaustive search through the search space would take more than 9 hours, whereas our use of a genetic algorithm took on average 8 minutes to find an optimized configuration.

Our contribution is a genetic algorithm that is capable of tuning optimizations on parameterized stencil codes. This genetic algorithm can effectively tune these codes to find near-optimal configurations for the applied optimizations in a very short amount of time, making it an effective method to use for auto-tuning stencil code optimizations.

The rest of the paper is organized into four sections: related work, tuning framework, experimental results, and conclusions and future work. In related work, other research that has been done in the field is presented and summarized along with how it is utilized in this research. The tuning framework section goes into more detail of stencil codes, optimizations, and the genetic algorithm that we used. Experimental results includes the experimental setup and the results we obtained from running our implementation on three different systems as well as a discussion of these results. Conclusions and future work summarizes this research and presents the outlook of incorporating it into future work.

II. Related Work

There exists significant research to automatically tune optimized stencil codes in order to find the best configuration of parameters for such optimizations [1], [3], [8], [11]. Datta et all have demonstrated the usefulness of optimizations with auto-tuning techniques as a means to effectively optimize stencil codes on both CPUs and GPUs [3]. Their work provides an effective base for the challenges of optimizing and auto-tuning stencil codes. Gana et all cite this work as their basis for using machine learning to optimize CPU stencil codes. In their research, they used a genetic algorithm in combination with the KCCA algorithm to perform quick searches through the parameter space of 4×10^7 different combinations. They managed to effectively auto-tune stencil codes on CPUs in two hours using their method [5]. Zhang and Mueller also researched auto-tuning and auto-generation of optimized stencil codes specifically for GPUs and GPU clusters which provides a more specific list of optimizations that are specifically used for GPU stencil code optimizations that were used in this research. In particular, their descriptions of 7-point and 27-point stencils, along with shared memory and register allocation for optimization were used throughout our research. [11].

Many optimizations have been developed over the years for stencil codes [2], [6], [7], [9], [10]. Nguyen et all provided a state-of-the art stencil code optimization that uses a combination of 2.5D thread blocking combined with 1d temporal blocking to create what they have called 3.5D blocking which provides throughput increases on GPUs of about two times what prior research had claimed [10]. In our research, we used their excellent description of 2.5D blocking as one of our optimizations for the genetic algorithm to automatically tune. Nguyen et all's research can also be parameterized by changing the amount of temporal blocking to perform, thus allowing a search space to be created for this optimization which was incorporated into this research.

III. Optimization Framework

A. Stencil Codes

Stencil codes are primarily used to solve partial differential equations in order to perform simulations such as heat flow or electromagnetic field propogation [3]. Most methods for solving these partial differential equations use iterative sweeps through spatial data, performing nearest-neighbor computations which are called stencils. Each node in the computation is weighted based on distance from the central node, which allows for the solving partial differential equations by switching these weights for the coefficients used in the solver. Using this structure, methods are created for different types of partial

differential equation solvers such as Jacobi iterative methods, which are the stencil codes we used in this research.

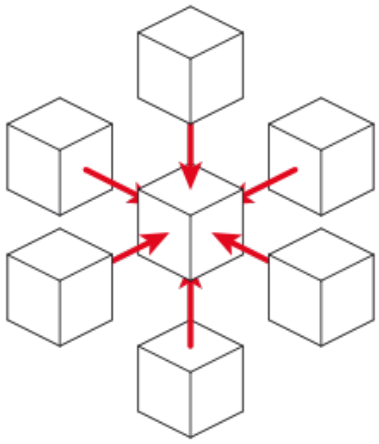

Fig. 2. A 6-point Von-Neuman stencil (credit: wikipedia.org)

As the data sizes used for stencil computations typically range outside the size of available cache memory, there is a large emphasis on data reuse and data-level parallelism in order to fully optimize stencil codes. This can cause portability issues as memory speeds and sizes can differ widely system to system, causing the need to use different parameters for optimizations on different architectures. This then produces a demand for a method to automatically tune stencil code optimizations on each architecture in order to enhance portability of the codes.

Auto-tuning of stencil kernels has become a fairly large area of study in order to work around the necessity of knowledge of the low-level specifications of the architecture in order to optimize the kernel. However, these auto-tuners may have to look in a parameter space that is upwards of 40 million combinations that may take months to fully check every single one for optimal performance [3]. This then creates a demand for a faster optimization process that is still automated in order to create a process that is viable for industry use. Thus, machine learning may be a good option for automatic optimization as it can use reinforcement learning paired with statistical machine learning and genetic algorithms in order to explore the parameter space much faster. Using machine learning may also overcome another downfall of auto-tuning in that each auto-tuner is generally programmed for one architecture, whereas a learner can learn architectures as well and correctly optimize for them.

B. Optimizations

In this research, we applied two types of optimizations to our stencil codes to be used in the tuning phase. The first optimization we used was 2.5D blocking. 2.5D blocking is an optimization for thread blocking of 3D stencil codes that only blocks in the x and y axes of the structured grid. Each thread then streams through the remaining z-axis, allowing for data-reuse of data already fetched by the thread earlier to fulfill data requirements. This optimization reduces the amount of global store and load instructions as threads can keep some data in the registers for quick access for several computations instead of fetching data from global memory each time a node must be calculated. The second optimization used is loop unrolling. Due to the nature of 2.5D blocking in that it must stream through the z-axis via a loop, this loop can be unrolled in order to provide more data-level parallelism and keep threads from becoming idle. These optimizations must be tuned in order to be fully optimized. 2.5D blocking takes two parameters: an x-axis blocking dimension and a y-axis blocking dimension. For a 256^3 grid, there are 60 different configurations of 2.5D blocking. For loop unrolling, the maximum unroll length allowed by the compiler is 192 iterations. By combining these two search spaces, the genetic algorithm used for tuning these optimizations searches through a search space containing 11,520 different configurations.

C. Genetic Algorithms

Genetic algorithms are a set of algorithms that mimic the natural selection process in order to find solutions to problems. Genetic algorithms do this by generating an initial population that generally consists of randomly generated individuals that contain randomly generated values for each parameter that will be searched. Each individual in the population then undergoes a selection process by which the fitness of their parameters that they contain is evaluated. The most fit individuals are then selected to be mutated and mated with each other in order to generate the next generation of individuals. This then continues until the population either converges to a singular value or the number of set generations is reached.

In this research, we used an initial population of ten individuals each with a chromosome (parameter set) containing three parameters – thread blocking for the x and y axes and loop unrolling factor. This population then underwent ten generations in order to get the individuals to converge on one value. The best performing individual was saved and then returned at the end of the generation process as the best configuration for the given optimizations. All of this was done using the Distributed Evolutionary Algorithms in Python (DEAP) project [4]. It allowed for use of built-in algorithms for the mating, mutating, and selection processes.

IV. Experimental Results

A. Goal

The goal of this experiment is to determine if genetic algorithms are a viable approach to tuning stencil code optimizations faster than other methods of tuning. The genetic algorithm used must be able to produce an optimal or near-optimal configuration for the optimized stencil code in a reasonable amount of time in contrast to the time it takes for an exhaustive search method to find the optimal configuration.

B. Setup

The setup we used to perform the experiments on consisted of three GPUs: one GPGPU (Tesla C2050) and two standard

GPUs (GTX 480, 680). Figure 3 details the theoretical peak FLoating-point OPeration (FLOP) rate determined by the number of cores (α) multiplied by the clock rate of each core (δ) multiplied by the number of FLOPs that can be performed each clock cycle (γ).

$$\alpha \times \delta \times \gamma = GFLOPS/sec$$

Fig. 3. Theoretical peak FLOP rate equation.

GPU	Architecture	Peak FLOP rate
GTX 480	Fermi	1344 GFLOP/s
Tesla C2050	Fermi	1030 GFLOP/s
GTX 680	Kepler	3250 GFLOP/s

Fig. 4. Peak GFLOP rate of GPUs (single precision)

The genetic algorithm was run on two stencil kernels: a 27 point Jacobi stencil and a 7 point Jacobi stencil. This genetic algorithm was used on each of the GPUs and was trained on the GTX 480 using the 7 point stencil. After the initial training, no values of the genetic algorithm were changed in order to generate the final results. The genetic algorithm used a three-gene chromosome to find configurations. The first two genes were for thread blocking dimensions along the x and y axes each being a power of two and their combined product could not exceed 2^{10} (60 combinations for 256^3 grid size). The third gene was for loop unrolling which was an integer from 1-192 for unroll length. The combined search space consisted of 11,520 different combinations the algorithm could possibly generate. This genetic algorithm was then run to create ten generations based on an initial population of ten individuals in order to find a configuration for each optimized stencil code that was close to the optimal value that was found through an exhaustive search of the search space.

$$a^{n+1}_{i,j,k} = \alpha(a^n_{i,j,k} + a^n_{i\pm1,j,k} + a^n_{i,j\pm1,k} + a^n_{i,j,k\pm1})$$

Fig. 5. 7-point Jacobi stencil equation.

$$a^{n+1}_{i,j,l} = \alpha(a^n_{i,j,k}) + \beta(a^n_{i\pm1,j,k} + a^n_{i,j\pm1,k} + a^n_{i,j,k\pm1}) + \gamma(a^n_{i\pm1,j\pm1,k} + a^n_{i,j\pm1,k\pm1} + a^n_{i\pm1,j,k\pm1}) + \epsilon(a^n_{i\pm1,j\pm1,k\pm1})$$

Fig. 6. 27-point Jacobi stencil equation.

The equations in figures 5 and 6 detail a typical 7-point and 27-point Jacobi stencil where a is the input grid, n is the iteration, and $\alpha, \beta, \gamma, \epsilon$ are coefficients multiplied upon the neighborhood sum. The ±1 symbols are used to save space in writing out each $i+1$ and $i-1$ for each i, j, k within the array of nodes.

C. Results

The graphs in Figure 7 are of the average performance in GFLOPS/sec of the population per generation. The red line is of the performance of the 7 point stencil code and the blue line is of the 27 point stencil code. The two dashed lines in each graph show the optimal configuration performance for each stencil code. The optimal configuration was found by performing an exhaustive search through the parameter search space.

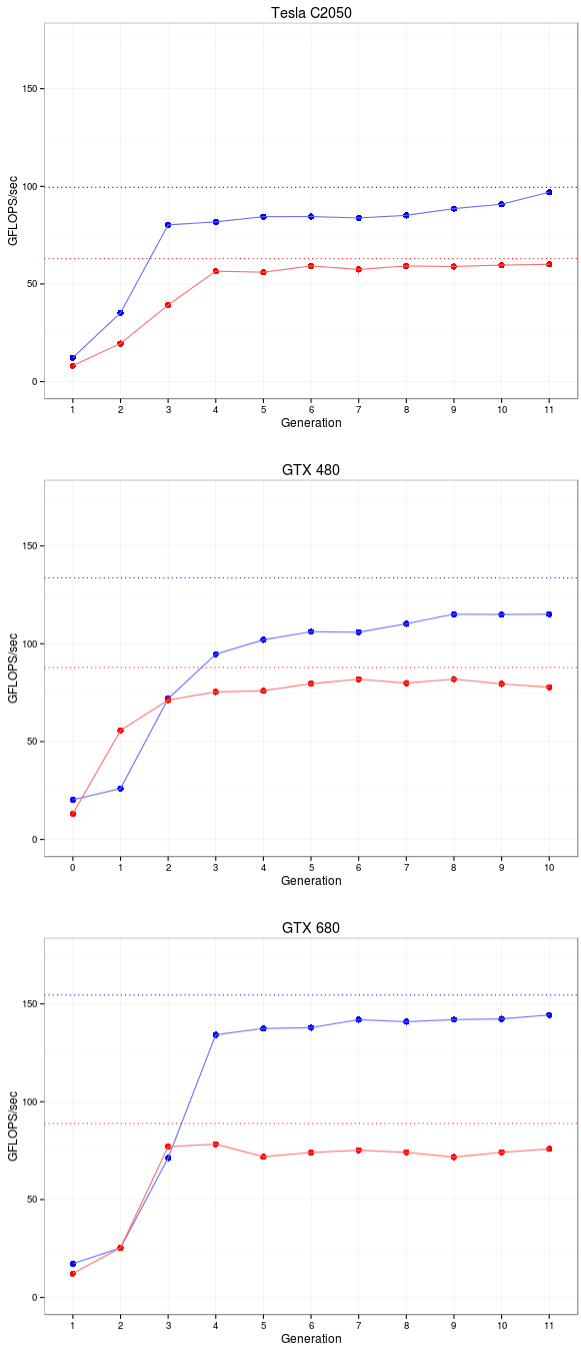

Fig. 7. Genetic algorithm average fitness of each generation for the three GPUs on both stencil kernels. The solid lines are for the average population fitness by generation for the 27-point stencil (blue) and 7-point stencil (red). The dashed lines show the optimal configuration throughput rate.

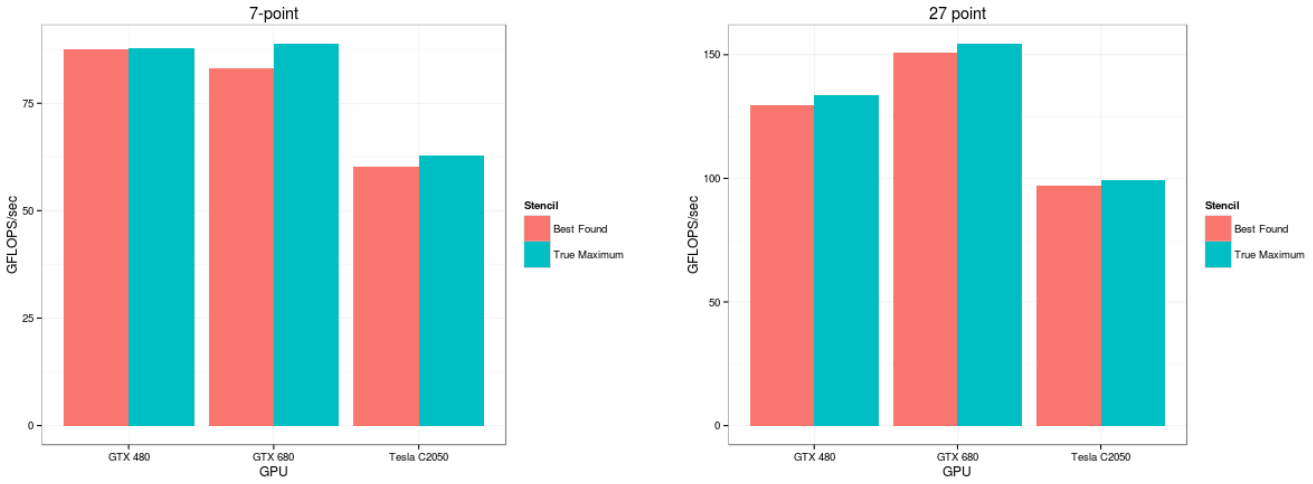

Fig. 8. Best configurations found by genetic algorithm vs the absolute best configuration found by exhaustive search for each stencil and GPU.

These results show the effectiveness of a genetic algorithm approach to auto-tuning stencil code optimizations as it generally only took 3–4 generations for each stencil code to be near-optimal. It should also be noted that these results only show the average performance of the entire population per generation, not the best candidates. The best candidates shown in Figure 8 of the population were typically within 3% of the optimal performance found for each stencil kernel by the 10^{th} population. The initial population for the genetic algorithm consisted of only ten members. Due to the small search space size, this small number of members was still able to quickly converge to a near-optimal configuration for each kernel. The small search size also allowed for us to check our results through exhaustive search as doing so took about 4–5 hours per kernel for each GPU. This speed is in contrast to the average eight minute execution time for the genetic algorithm to generate all ten generations and find a near-optimal configuration. These speeds differ in terms of which CPU is used to compile each code, but the large gap in performance still persists for each CPU, regardless of its speed.

However, these results show that each kernel could only reach up to a maximum of 100 GFLOPS/sec for the 27 point stencil on the Tesla C2050, which is far lower than the 450 GFLOPS/sec produced by Bergstra et all [2] on the same model of GPU. This is due to the optimizations that were used in our stencil codes as they are the main bottleneck of performance for the stencil code. Our optimizations still contain thread divergence in the code, and is thus less optimized compared to Bergstra et all's kernel which contains no thread divergence. For future work on this research, more optimizations will be considered so that the results will be closer to current stencil code performance.

V. Conclusions and Future Work

This work demonstrates the effectiveness of using a genetic algorithm in order to find near-optimal configurations for stencil code optimizations across multiple GPUs with differing architectures. This result allows for enhanced portability of stencil code optimizations to differing architectures in a timely fashion as the tuning phase was demonstrated to be much faster than exhaustive search alternatives as the genetic algorithm took, on average, eight minutes to generate all ten generations of the population in contrast to the 4–5 hour run time of the exhaustive search.

In the future, we would incorporate more parameters for use in the chromosome for the genetic algorithm in order to generate a search space worthy of using a machine learning technique to traverse it instead of exhaustive search being a viable method to use. We will also develop more parts to the auto-optimization framework from figure 1 such as the optimizer and stencil code classifier. This is in the hopes that a functional framework can be created such that it may be used to optimize existing stencil codes that are in use today and be continually optimized as more stencil code optimizations are found.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. 2012.

[3] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[5] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar09)*, 2009.

[6] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[7] Julien Jaeger and Denis Barthou. Automatic efficient data layout for multithreaded stencil codes on cpu sand gpus. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[8] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, page 256265. ACM, 2009.

[9] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[10] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM.

ACKNOWLEDGEMENTS

This research is supported by NSF grant 1359275.

RSSE: A New Method of Distributing Datasets and Machine Learning Software

Michael Gohde
Vision and Security Technology Lab
University of Colorado at Colorado Springs
Colorado Springs, Colorado
mgohde@uccs.edu

***Abstract*—Among the challenges faced by machine learning researchers today is that of distributing the datasets and algorithms used in their research. This problem arises mostly from the limitations involved in hosting datasets on servers outside of their origin. RSSE (Really Simple Syndication for Experiments) is intended to provide a message-based system with which researchers can share their data and algorithms.**

I. Introduction

RSSE draws on existing standards, namely XML and RSS, to facilitate easier communication and distribution of data among researchers. While RSSE is not an extension to the existing RSS standard[6], it is intended to be similar in general conventions and syntax to RSS. As such, it extends the concept of RSS, which is that of message-based syndication involving a client "reader" application and a message server established by an institution. RSSE is designed so that messages can be written either by researchers themselves or by automated tools.

Due to current copyright and IP law, it is often difficult or impossible for institutions and researchers to directly distribute external datasets used during computation[4]. Some large dataset providers, such as Yahoo, explicitly prohibit the redistribution of the dataset itself, instead allowing the dataset to be distributed in the form of links[5]. Such datasets usually allow users to cache the data locally. By providing a consistent system by which data and software can be distributed using messages containing URLs and checksums, RSSE should enable institutions to easily monitor and expand on the work supplied by other institutions. Furthermore, by passing URLs to datasets rather than the datasets themselves, experiments can be run by other institutions while still respecting the Intellectual Property rights of the dataset's source.

RSSE will work in a similar fashion to RSS (Really Simple Syndication), with some exceptions. As such, a researcher or automated utility would generate an XML file using RSSE tags and serve it over the HyperText Transfer Protocol (HTTP). Such an XML file would contain the project's title, a brief description of the project itself or changes made recently, several URLs, and checksums for the relevant URLs. Each URL should usually refer to a datasets involved during the course of research. One possibility, however, is that some of the URLs could refer to source code or compiled Java classes, which would, in turn, be executed locally to verify the results of the computation. When an XML file containing RSSE data is posted, client programs could proceed to download the file, parse its contents, then perform a predetermined set of actions, such as downloading all of the datasets and source code involved in the remote experiment. For a graphical representation of the data transfers involved, see figure 1.
(figure 1)

II. Previous Work

RSSE draws primarily off of the existing RSS standard[6]. Due to its flexible nature and widespread use, RSS has already utilized as a basis for distributing information to research librarians in an organized fashion[2]. A similar system featuring a dedicated message and client-based infastructure was implemented to distribute climactic data, however it did not explicitly use RSS, rather it directly exposed a database to a network[3].

III. Implementation

A. Implementation History

For this project, a reference implementation of the RSSE reader and file generation utility were written. As RSSE will be an open standard, the implementation discussed here exists solely as a reference for other future implementers to follow.

The first stage of the implementation was considering what tags would be acceptable for each RSSE file. These tags are listed in Table 1. The tags were determined after considering the minimum set of data necessary to represent a message. The most important tags involved are the dataset tag, the checksum tag, and the checksumtype tag. The dataset and checksum tags are self explanatory. The checksumtype tag will contain a string value representing the hashing algorithm used to generate the checksum on the server's side.

The second stage was the implementation of the RSSE reader program. This application was implemented before the RSSE message generation program because of the relative ease in manually writing RSSE files as opposed to manually reading RSSE files. Upon starting, a command line specification was drawn up based on all of the tags and operations expected of

the program. The command line interface was implemented first, due to the ease in doing so. Command line options are not included here as it should only be necessary for future implementations to utilize the base set of RSSE tags as opposed to implementing full compatibility. This was followed by the implementation of a GUI, which extended the features of the command line interface.

Once the graphical component of the RSSE Reader was complete, work started on the RSSE Generator. Because all of the features of RSSE were fairly stable by this point, the RSSE generator was far easier to implement. Unfortunately, due to time constraints, the generator does not yet have scripting support, however that has become a priority in the near future.

While each program is currently stable enough for general use, there are some UI elements that do need improvement due to their obtuse or erratic behavior. The most obvious of which is the difference in graphical styles between the reader and file generation utility.

Upon completing the first few revisions of the RSSE reference implementation, the project was demonstrated to a group of machine learning researchers. Based on their suggestions, the RSSE version 0.03 specification was drawn up with several enhancements in the form of four new tags and three new sets of attributes. These new tags should enable both researchers and end users to benefit from increased tailoring to various conditions and systems. The new features are mentioned in separate tables. Upon starting work on this version of the specification, it became very clear that each version of RSSE would in the future cause rendering and generation problems on prior and future versions of the RSSE reference implementation. Because of this, the <rsse> tag now carries an attribute specifying the expected minimum version code necessary to render a given RSSE message. The version encoding scheme is elaborated in Table IV.

B. Reasons For Using The Technologies Used

While doing the project, it became clear that it may be necessary at some point to justify the use of Java and XML as the primary language and data framework of the application, respectively.

Firstly, Java was selected as the primary implementation language due to its feature-set. Java has extensive support for reading and parsing XML files, which proved invaluable for the project as a whole. Furthermore, Java provides easy to use networking and graphical user interface APIs, which contributed to the quick implementation of the project. Finally, the project's code should be easily readable to a wide array of programmers due to the similarities in syntax and usage between Java and other programming languages.

XML was selected mostly because it is very easy for humans and computers alike to read and parse. By using plain-text instead of binary for messages, it allowed the developer to write test messages to pass to the reader before the file generation utility was complete. Furthermore, while it was not a clear focus in the beginning of the project, XML allows for a significant level of complexity and flexibility, which allows the RSSE standard to expand easily in the future. In the future, it is likely that future implementers will write their own messages in order to test their RSSE reader applications.

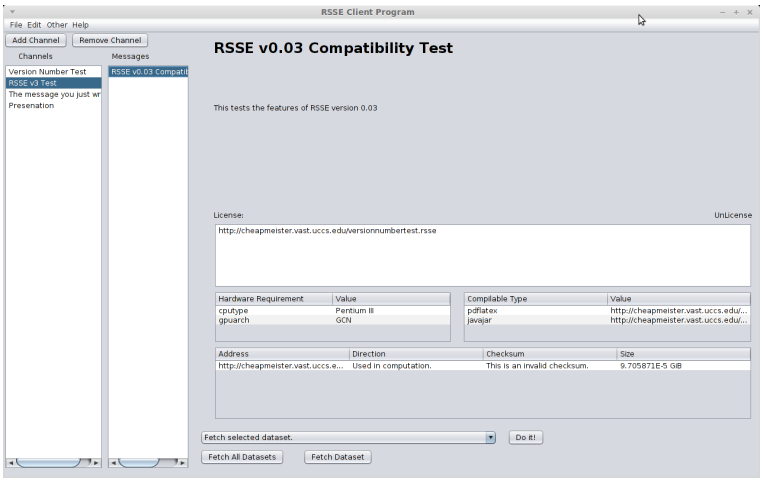

Fig. 1. The RSSE Reader Program

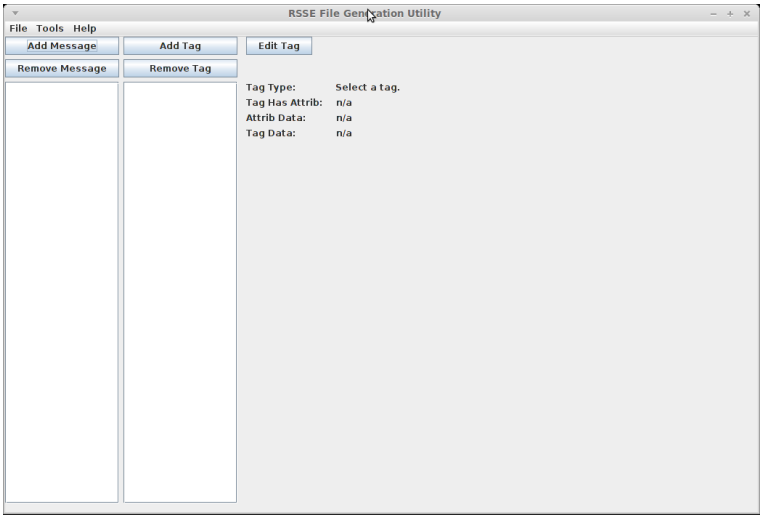

Fig. 2. The RSSE File Generation Program

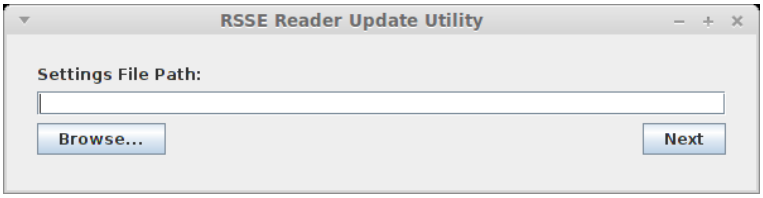

Fig. 3. An early build of the RSSE Updater

C. Interface and Design

While the graphical user interface is in a very early development stage, it is complete enough to be shown here. Please refer to Fig. 1 for an example configuration of the Reader, and Figure 2 for an example configuration of the Generator. In both programs, there are clearly defined lists of values to be modified and modifier buttons either on top of the lists or to

their side. This was done to associate the various modifiers with the data involved, which should hopefully lend itself to usability. The menu layout is very sparse, as most of the functions encountered in the program are represented by the buttons present. There are very few dynamic UI elements in order to promote portability to low-power devices and older operating systems. By providing unambiguous functions, the RSSE reference applications should be very easy to learn. Overall, this design aesthetic should prove helpful for the purposes of providing a reference implementation from which other implementations of the RSSE standard can be derived.

Tag	Tag Value
<rsse>	Tag used to denote an RSSE file.
<message>	Tag used to mark the start of a message. This allows for there to be more than one message in each file.
<title>	The title of the project.
<description>	The project's description.
<link>	A website to be visited by the user. Could be used to direct clients to more information.
<dataset direction="in">	Represents a dataset. The direction attribute is used to inform the user as to whether the dataset was used in computation (value "in") or generated as the result of computation (value "out").
<checksumtype>	Represents the type of checksum to generate and check.
<checksum>	Represents an individual checksum. Will be associated in the order of appearance of datasets.

TABLE I
TAGS IMPLEMENTED BY THE RSSE REFERENCE SOFTWARE

Tag	Tag Value
<update url="url">	Tag that could be used to send updates to the reader. Url attribute is used to mark the URL of the update. The tag will contain the
<license>	Tag that could be used to distribute executable code if implemented.
<minspec>	Tag representing the minimum specifications to run software bundled with a message.
<compilable type="type">	Tag that could be used to distribute code with special compilation requirements.

TABLE II
TAGS ADDED IN RSSE VERSION 0.03

Attribute	Tag	Description
pdflatex	<compilable>	Allows for the inclusion of LaTeX documents.
makefile	<compilable>	Allows for the distribution of projects utilizing makefiles.
javajar	<compilable>	Allows for the distribution of Java Jar files.
executable	<compilable>	Allows for the direct distribution of executable files. The reference implementation will never allow these files to execute without a prompt.
gpuarch	<minspec>	Allows for researchers to specify different GPU architectures, such as Kepler or GCN.
cpuarch	<minspec>	Allows for researcers to specify a CPU architecture for specific optimizations.
cputype	<minspec>	Allows for researchers to specify a specific type of CPU to be used. Only for heavy optimizations.
corecount	<minspec>	Allows for researchers to specify a minimum number of cores to comfortably run multithreaded software.
minram	<minspec>	Allows for researchers to specify a minimum amount of RAM as a floating point number of gigabytes.
minstorage	<minspec>	Allows for researchers to specify a minimum amount of free hard drive space as a floating point number of gigabytes.
osfamily	<minspec>	Allows for researchers to specity the intended operating system family for their software. This could be used to prevent non-POSIX operating systems from attempting to compile the software included.

TABLE III
ADDITIONAL ATTRIBUTES IMPLEMENTED IN VERSION 0.03

RSSE version	RSSE Tag
v0.01 (Depreciated)	<rsse>
v0.02	<rsse>
v0.03	<rsse version="3">
(Future releases)	<rsse version="(Version number*100)">

TABLE IV
THE RSSE VERSION ENCODING SCHEME.

IV. CHALLENGES

While writing the checksumming portion of the program, it became very clear that Java's default IO functions were too slow for the task. While it has not yet been implemented, the project will eventually add support for Java's NIO (Non-blocking IO)[1], which should provide a high performance framework for checksumming operations. Another challenge encountered was that of determining how checksums should be transmitted, as it is difficult to parse multiple attributes in each tag. This problem was solved by associating each checksum tag with dataset tags in the order that they appeared, however this is more of a short-term solution that may require the implementation of a complete XML parser within the code. One of the clearest challenges is deciding on which tags should be added to each version of the RSSE specification, as it involves several decisions as to which features would be easiest to implement, as well as which features would be best for end-users. Overall, accepting suggestions from others proved to be very beneficial to the project as a whole.

V. APPLICATIONS

RSSE has the potential to become a very valuable tool for researchers, especially those who wish to use commercial or otherwise difficult to distribute datasets. While RSSE is intended to be used primarily by a machine learning and computer vision audience, it has the potential to be used for scientific research, namely in peer-review. As such, RSSE need not be constrained to just distributing datasets. It has the potential to distribute papers, code, or even precompiled binaries to remote computers for independent verification of results. While it is not the intent of the project, it can be used to assist with distributed computing with only minimal modifications.

Feature or Tag	Information
Automatic Updates	The reference RSSE implementation currently includes a very basic update utility. In the future this utility will be improved and expanded.
Compilation and Execution Support	In its current state, the RSSE reader can only download executable or compilable objects from remote servers. Future revisions will allow for proper compilation and execution of RSSE messages.
System Requirement Checking	The RSSE reader is currently incapable of checking system requirements. In the future, support for this will be added.
Merge Checksum and Dataset	Merging the dataset tag and the checksum tag would streamline the distribution of datasets.
Improving Checksum Performance	Checksumming in the file generation program is unacceptably slow.
Implementing more Minspec Tags	The minspec tag list is currently somewhat incomplete.
Implementing Local Caching	One of the long-term goals of this project is to develop a caching system to avoid several of the problems in distributing datasets.
Implementing the RSSE Executor	RSSE will be functionally divided between the Manager (what the reader is now) and the Executor, which will fetch cached data and try to process it.
Splitting the Reader	The current RSSE Reader application is currently insufficient for caching and the constant update cycle needed by real-world researchers. As such, it will be split into two programs: The RSSE Reader and the RSSE Manager. The RSSE Reader will act as a graphical configuration utility for the RSSE Generator. As such, most of the features of the Reader will be merged into the Manager.
Updating Scripting Support	The RSSe file generator has great potential to be scripted, especially in providing automatic rapid updates to end users.

TABLE V
FEATURES TO BE IMPLEMENTED IN FUTURE RELEASES OF RSSE

Some clear distinctions need to be drawn between the RSSE project and RSS, however. Its focus on academic pursuits should be preserved and remain a primary goal. As such, other applications, such as distributing newsfeeds or non-research related data should be discouraged to avoid feature bloat. Such feature bloat would make implementing additional readers and file generators significantly more difficult than the standard is designed to allow. However, other implementations should be encouraged to deviate somewhat so that additional features can later be brought into the mainstream RSSE specification.

VI. Summary and Conclusions

Provided that the RSSE project becomes widely adopted, it will provide a clean, easy to use, and rapid means by which researchers can share data. It has been designed with a clear emphasis on having low barriers to entry. These low barriers to entry should allow RSSE to become a *de facto* standard in research and communication. Given such status, other implementations of the reader and file generation programs would likely be written with more features than could be implemented here. Such implementations can be expected to include features specific to various fields, such as some peer review system for scientific research.

VII. Acknowledgements

I would like to thank Dr. Terrance Boult for proposing the original idea for RSSE, as well as the machine learning researchers who demonstrated a need for this software. I would like to thank the NSF for providing funding for the REU research program.

References

[1] File i/o (featuring nio.2). http://docs.oracle.com/javase/tutorial/essential/io/fileio.html. Accessed: 2014-07-03.

[2] Alexia D. Estabrook and David L. Rothma. Applications of rss in health sciences libraries. *Medical Reference Services Quarterly*, 26(sup1):51–68, 2007. PMID: 17210549.

[3] Hannes Grobe, Michael Diepenbroek, Nicolas Dittert, Manfred Reinke, and Rainer Sieger. Archiving and distributing earth-science data with the pangaea information system. In DieterKarl Ftterer, Detlef Damaske, Georg Kleinschmidt, Hubert Miller, and Franz Tessensohn, editors, *Antarctica*, pages 403–406. Springer Berlin Heidelberg, 2006.

[4] Gerald Schaefer and Michal Stich. Ucid: an uncompressed color image database, 2003.

[5] David A. Shamma. News: One hundred million creative commons flickr images for research. http://labs.yahoo.com/news/yfcc100m/. Accessed: 2014-07-03.

[6] UserLand Software. Rss 2.0 specification, 2002.

Learning Patterns of Mobile Interface Design

George C. GUVERNATOR V
The College of William and Mary
Williamsburg, Virginia
gcguvernator@email.wm.edu

***Abstract*—Nothing for now. We'll write our abstract last.**

I. INTRODUCTION

Interface design is a crucial element in any software project. Graphical interfaces allow for more intuitive human-computer interaction but can challenge even the most skilled developers as they take on the additional responsibilities of a designer. In the world of mobile applications, where many competing implementations of an idea are available to users, design can play a key role in a user's choice of application. Additionally, limited and varying screen sizes, touch-based interfaces, and limited resources all challenge the mobile interface designer. It is our observation that, unfortunately, some developers fail to spend ample time designing Graphical User Interfaces (GUIs). This is especially true in academia, where many of the most technically correct and well-implemented software projects falter in this area, considering GUI design an overly costly afterthought. In mobile applications, this can mean that not all screen sizes, input methods, accessibility features, and other mobile-exclusive considerations are accounted for. As a result, developers lose many potential users from otherwise well-written and well-executed projects.

Mobile application development presents unique challenges beyond those faced when developing software with more traditional keyboard and mouse interfaces. On mobile platforms, user interface design poses the unique challenge of restricted screen space with respect to more conventional desktop or console platforms and therefore has a greater influence on the overall usability of the application. Additionally, developers must keep in mind the variety of devices of different size that their applications will run on. An interface built for a 15 by 10 cm tablet, for example, may not scale well to a 9 by 5 cm smartphone. Elements designed to be tapped on the larger screen by human fingers or styli would become more difficult to accurately activate on the smaller screen. Another complication to developers are extensions to the platform's standard user interface, such as those used for accessibility or universal access by users with physical disabilities. Finally, fragmentation on the Android platform causes design and development bugs, both from devices running different versions of the Android platform which support different graphical layout components, and from a variety of vendors building their devices differently. The problem of Android device and version fragmentation are discussed by Han et. al. [1] and Degusta [2].

This research funded by a grant from the National Science Foundation (1359275).

Facing these additional challenges of mobile development, it is therefore important to understand both how mobile applications are designed and which identifiable design patterns users prefer. The former can be accomplished given access to an application's source code, and the latter is theoretically possible by scraping user ratings (both long-form text reviews and one- to five-star numeric ratings) from Google Play, the officially supported Android application repository. Given the unique challenges of the mobile environment discussed above, we assert that design is a significant factor reviewers consider when rating an application. While many reviews center around program functionality and stability, we believe there is enough weight placed on design to show clear trends and allow for correlative measurement.

In this research, we analyze the GUI design of a variety of Android applications, allowing us to gather data and create a characteristic model. We evaluate correlations between the gathered design data and reported user experiences, such as comments and ratings, as well as other potentially confounding variables found in the application's metadata from the application repository.

To accomplish this, we scrape Android source code from the F-Droid Web repository[1] using a tool we developed called *fdscrape*.[2] Package names found with the source code on F-Droid are also searched on Google Play, a non-free Android application repository, and metadata such as user ratings, comments, popularity, and size are scraped and associated with each application. The combined source code and metadata, collectively the *F-Droid corpus*, is run through a program we have developed called AGUILLE.[3] This tool analyzes the structure of GUI markup language in the source code and extracts and counts the individual elements used to construct the interface, analyzing and combining data points to prepare for machine learning analysis.

Finally, the analysis of AGUILLE and metadata found with fdscrape are combined in a machine learning workflow in Weka. The workflow leverages the power of the M5 model tree to generate explainable branches and decision points that are applied in an ordered hierarchical structure to determine a potential rating for future applications. This is improved by analyzing each application category separately and building separate models for those categories with enough applications to make valid predictions. This categorical discretization is a

[1] F-Droid can be accessed on the Web at `https://f-droid.org/`.

[2] Fdscrape is licensed under the GNU General Public License (version 3) and is available on the Web at `https://github.com/qguv/fdscrape`.

[3] AGUILLE is licensed by the GNU General Public License (Version 3) and is available on the Web at `https://github.com/qguv/aguille`.

key part of our experiment, see section IV on the following page.

A summary of specific results should go here. It will mirror the summary that will end up in the conclusion.

Further development could turn the predictive model into a suggestive one. The models generated by the framework described in this research could be a crucial addition to "Interface Builders," [3] graphical applications designed to help developers create graphical interfaces. Developers would have a new, powerful tool suggesting subtle changes to their design in order to better emulate the most popular and successful graphical interfaces available today.

The main contributions of this research are as follows:

- Development of a framework of tools to gather layout information of Android applications (section III)
- An empirical study correlating Android GUI design patterns and the reported quality user experiences (section IV on the following page)
- Analysis of recurring design patterns and trends in Android GUI design (section V on page 4)
- Development of a predictive model for mobile GUI design (section IV-B on page 4)

II. Background

This section will be expanded to better explain where our research fits in the field of related work discussed in section VI on page 5.

Research on learning design patterns [4], [5] proves useful when designing a predictive machine learning workflow. Both Neural Networks and vanilla Decision Trees are discussed and implemented in [5].

We began by correlating specific features and ratings manually using straightforward linear regression in order to test different weightings of features. Next, random forests were used to gain insight on the sorts of decision points that regression and model trees produce. We eventually settled on to the M5 model tree to firm up final results and to allow trends to be explained and described in human-friendly decision points rather than difficult-to-describe coëfficient models or more opaque random models.

A. Choosing Android

Though the concepts presented in this paper are applicable to any graphical environment, we have chosen to work with the Android mobile platform due to both the unique challenges of a mobile environment discussed in section I on the preceding page and the uniformity and availability of application source code.

We feel the concepts presented in this paper would be most advantageous to mobile developers, as user ratings, our evaluative metric, can directly influence an application's ultimate success or failure. Android users must often choose between similar implementations of the same tool. The Google Play store in turn provides a system with which users can rate applications and post feedback for developers and other potential users. These user ratings help Android users to narrow down their choices in a vastly competitive market.

Android is also ubiquitous among mobile device users. The popularity of the platform continues to grow as new users and developers adopt Android as their primary mobile platform. With over 1.3 million[4] Android applications on the Google Play store at time of writing, the popularity of the Android platform has provided us and will continue to provide other research teams with ample data to search for significant correlations and generalizations.

Finally, the somewhat constrained GUI design framework in the Android platform (Android XML) allows for relatively straightforward parsing of the application's graphical layout without needing to peek into the application's logic.

Because of these unique properties of the Android platform, we believe mobile applications will benefit most from the preliminary results presented in this paper.

III. Extracting Design Elements

Before we can begin evaluating and correlating patterns, we must first collect information on Android GUI design. Since Android developers define graphical layouts in source code, we decided to gather and interpret source code in order to gather data about Android GUIs. We chose the free and open-source Android software repository F-Droid as a source for Android source code.

After gathering source data, we must extract the parts of the source code pertaining to graphical layouts. We then analyze those layouts to determine what built-in graphical elements the developer chose to use in designing the application and in what quantity and proportion.

After all layouts of all available applications in the repository have been analyzed, the results are fed to a machine learning algorithm to make generalizations about which elements affect others, which best predict ratings, and which carry little meaning in the context of this study.

Finally, the performance of this machine learning system is analyzed, and the workflow is tweaked to attempt to improve prediction and correlation both between elements and against user ratings.

A. Dataset Acquisition with fdscrape

We have developed a program (in Python) to enable mass retrieval of Android source code to mine. We use F-Droid, a software repository containing binaries and source code for 1,145 free and open-source Android applications. The majority of these applications are also available on the official Android application repository, the Google Play store.[5] Because of this, we have downloaded all available applications and their source code from F-Droid as well as Google Play ratings and metadata for the same applications, storing the data for analysis in the machine learning step.[6] This data is stored with the source code of each application.

[4] According to *Appbrain Stats*, a Google Play metrics service. Visit `http://www.appbrain.com/stats/number-of-android-apps` for the latest statistic.

[5] The Google Play store can be accessed on the Web at `https://play.google.com/store`.

[6] To accomplish this, we have cross-checked Java package names against both F-Droid and the Google Play store.

We originally chose to scrape only rating information from Google Play. It was decided, however, that by saving more of the metadata provided by Google Play and developers, better predictions could be made by accounting for variables beyond the scope of design. This permits meta-analysis of our hypothesis, i.e. we can decide how much design affects ratings and how much predictive accuracy to expect from our model. We gather this data in order to compensate for any confounding correlation that Google Play metadata may have on determining rating.

Specifically, the developer-chosen application category (e.g. Weather Application, Productivity Application, Action Game, Puzzle Game, and others in table I) provides an effective way to analyze groups of applications at a time. It is our hypothesis in RQ 1 that separate analysis within application categories will yield more meaningful results. This has the potential to greatly improve the accuracy of our predictive algorithm and allows us to better understand what "good design" entails in certain domains. For example, the same elements that constitute good design for an action game might exemplify bad design for a news application.

After omitting applications that were not on the Google Play store, had no ratings, or did not host source on the main F-Droid website, we collected the source code of 894 applications to build our dataset.

B. Tag Lexing & Extraction with AGUILLE

We have developed AGUILLE, the Android Graphical User Interface Lazy LExer, to perform the Android source analysis we originally hoped GUITAR would accomplish. The tool takes in an application's source code, finds the relevant Android XML structure, and parses that structure into native Python objects. Using these objects, AGUILLE calculates the frequency with which each XML tag, or element, occurs in the application. The graphical design of the application, therefore, is reflected in the developer's choice of graphical elements.

These frequencies are collected in a CSV file, along with the metadata gathered with fdscrape. Lots of the scraped information can be cached to speed up parses of entire repositories.

The tool is designed such that, should more sophisticated calculations prove necessary, separate sub-commands could easily be added. AGUILLE is open-source; anyone may extend it by adding further subcommands or modifying its current behavior.

C. Machine Learning with Weka

We make use of the Weka 3.7 *Knowledge Flow* environment to create a machine learning workflow. Data from AGUILLE is loaded separately by category. We drop categories which contain less than one percent of all applications mined, leaving the 20 categories described in table I.

	Applications	
Category Name	Number	Percent
'Tools'	278	33.3%
'Productivity'	88	10.5%
'Communication'	67	8.0%
'Personalization'	34	4.1%
'Books and Reference'	32	3.8%
'Game Puzzle'	30	3.6%
'Education'	29	3.5%
'Media and Video'	29	3.5%
'Music and Audio'	25	3.0%
'Entertainment'	24	2.9%
'Transportation'	18	2.2%
'Travel and Local'	18	2.2%
'Finance'	17	2.0%
'Game Arcade'	17	2.0%
'Health and Fitness'	17	2.0%
'Lifestyle'	15	1.8%
'News and Magazines'	15	1.8%
'Social'	15	1.8%
'Photography'	13	1.6%
'Libraries and Demo'	11	1.3%
Total: 20 categories	792	94.7%

TABLE I. APPLICATIONS IN EACH MINED CATEGORY

IV. EMPIRICAL EVALUATION

The design of our experiment is such that two chief research questions (RQs) may be addressed:

1) What sort of design do applications have in common? What sort of trends emerge when analyzing entire repositories of applications?
2) Does separate analysis of applications by Google Play category improve the quality of our predictive algorithm? Does such separation give more meaningful generalizations when explaining the output of our decision tree?

Our goal in evaluating the accuracy and quality of our predictions is twofold: to attempt to improve the ability of our algorithm to predict user ratings and to make conclusions how about individual elements affect user perception of an application.

When evaluating the performance of our machine learning workflow, we are looking for statistically significant correlations with performance better than the 5–10% correlation we see with naïve sample algorithms.

We cannot expect anything near perfect prediction, as more goes into a user's choice of rating than design. We must therefore focus on explainable output, such as that from a decision tree, to try to explain the influence of design on ratings.

A. Experiment Design

To be able to learn which design elements lead to the best applications, we need a group of factors, or *heuristic*,

to evaluate, as well as a metric for determining what constitutes a "good" application. This study uses the frequency of use of Android XML tags in graphical layout source code as a heuristic. Because the Android framework comes with a rigidly-defined set of elements, XML tag (and therefore element) frequency allows us to point to very specific design choices to explain findings when learning correlations.

Android also allows developers to define their own tag elements, but because scraping these developer-defined, application-specific tags and properties involves parsing Java logic and rendering the elements, this added complexity is somewhat beyond the scope of this project. The application-specific nature of these tags also means that they will most likely not be of use when attempting to make correlations between applications.

Although it may be possible to learn a more complex heuristic over time, it would increase overhead and likely introduce excessive complexity into our system. At this stage of research, it is wiser to rely on the pre-designed elements available to all Android applications to potentially determine the quality of GUI design. We have found that element frequency is sufficient to establish a statistically significant correlation between the elements themselves and the evaluative metric, Google Play ratings.

Potential alternative evaluative metrics could include uninstall rate and frequency, certain statistical functions on the cumulative body of Google Play user ratings, or cross-referenced reviews from established news sources. Google Play ratings were trivial to scrape and analyze, as the data is publicly available, so these public ratings serve as an initial metric we use to establish the overall quality of an application. In our algorithm, we can weigh the rating's relevance depending on how many users rated the application, a metric we also obtained from the Google Play store. We might be more confident in a metric to which many users contributed.

In early models, we found that the M5 model tree first branched based on the amount of user ratings on Google Play. The sub-trees after this split did not clearly resemble each other; the branch reached by applications with few ratings gave a counter-intuitive model, while the branch for applications with a more substantial amount of ratings weighted elements as we would expect. This suggests that the model's predictive accuracy increases substantially when more rating data is available, as we would expect.

Our group of independent variables, the frequency of each Android graphical element, will reflect the different proportions of interactive Android *widgets* with respect to each other. These interactive widgets are built-in to the Android platform and include buttons, check-boxes, radio buttons, images, and text. Additionally, these widgets can appear in different *views*, all of which have different ways to specify how the widgets will be laid out on the screen. All of these views and widgets are part of our element count.

Of course, GUI design is hardly the only factor users consider when rating a program. It is important to consider major confounding variables (viz. quality of functionality, stability, the ability of the program to solve a real problem) and integrate Google Play's qualitative long-form paragraph review system. A naïve but effective way to acknowledge applications with known performance issues (and therefore identify those applications whose low ratings have little to do with design) involves searching for key terms occurring abnormally frequently in text reviews. See table II for a list of key words and phrases that could indicate poor performance rather than poor design. Such key words and phrases are likely to indicate that low ratings are due to factors outside the realm of interface design. After gathering other metadata, fdscrape counts the frequency of these key words and phrases with respect to the available body of reviews. The calculated frequencies of these words are fed into the machine learning algorithm along with the tag frequency count from AGUILLE and the metadata from fdscrape.

If our algorithm were to put significant weight on these terms rather than the intended features in our heuristic, we can safely chalk these up to poor application performance. This gives a decision tree the option to discard obviously poor-quality applications in an early decision node in order to focus on the design factors we are interested in analyzing. If a more sophisticated method than calculating tag frequency proves necessary, we could take a naïve Baysean approach, analyzing the probability rather than the frequency of key phrases in known or exemplary good and bad applications.

By acknowledging applications which have known issues unrelated to design, observed ratings of the remaining applications in our dataset will better reflect design quality.

Key Phrase	Possible Conclusion
'incompatible'	Could indicate versioning or device compatibility issues for certain users.
'uninstall'	Could indicate frustration with the application or the inability for certain users to un-install pre-installed software.
'crash'	Could indicate stability issues.
'slow', 'lag'	Could indicate resource overloading, frequent Internet requests, or poor data structure implementation.
'black screen', 'white screen', 'blank screen'	Could indicate initialization problems.

TABLE II. KEY WORDS AND PHRASES SUGGESTING POOR PERFORMANCE

B. Experiment Results

Results would go here, once we decide what output of which machine learning workflows to include.

V. DISCUSSION & CHALLENGES

The single most significant setback to this project has been the failure of the Android fork of the GUITAR tool. We have been forced to develop an in-house tool from scratch in its place. It has taken weeks to develop AGUILLE to a usable and dependable state. While new developments such as those detailed in section IV-B show promising correlation

and prediction, preliminary results with primitive data did not show expected trends. Specifically, before AGUILLE and our machine-learning workflow became capable of more sophisticated data transformations, the sample heuristic (viz. mean amount of buttons per layout in each application) correlated against average rating showed no statistically significant results.

After improvement to AGUILLE and the addition of metadata and tag phrases, statistically significant results surfaced. Category discretization provided even better results, as discussed in section IV-B on the preceding page. We believe further probing into consequential design decisions and further sophistication of AGUILLE and the design heuristic will continue to render reportable, statistically significant results.

For example, a further step up in sophistication involves a report of what percentage of all available screen space is occupied by widgets.

VI. RELATED WORK

Available research into the overlap of the machine learning and user experience fields tends to concentrate on either GUI testing or programming interface (API) design rather than using machine learning to gain insight on the GUI design patterns users favor. Much available research that does indeed combine machine learning and user interface design aims to design front-end applications for the non-statistician that enable powerful data mining with little knowledge of the implementation of machine learning algorithms.

Arlt et. al. [6] have written a chapter on various different methods of parsing and testing GUIs. The research of Nguyen et. al. [7] presents a tool called GUITAR to parse the structure of an application's GUI in order to generated automated tests for that application. We were originally hopeful that GUITAR or one of its derivatives could prove invaluable in gathering GUI data to mine. Unfortunately, the Android-specific fork of GUITAR has not been updated since the release of Android 2.2 and is therefore not compatible with the majority of applications on F-Droid. Although much existing research [6], [8], [9] makes use of GUITAR, our GUI-parsing tool had to be developed from scratch as discussed in section III-B on page 3.

The approach posited by Yang et. al. [9] to programmatically generate GUI models in mobile applications does not use GUITAR in its entirely; rather, it analyzes GUI events using GUITAR and calls these events directly on the application. Similarly, Amalfitano et. al. [10] showcase "an automated technique that tests Android apps via their [GUI]." Although writing a GUI parser from scratch may have slowed down development, using just one tool to rip the GUI of an Android application where other teams have used many has helped to simplify the process of gathering GUI data.

The research of Shi et. al. [4] and Ferenc et. al. [5] discusses ways to better understand source code design patterns in Java and C++, respectively. Our research aims to discover design patterns in graphical interfaces, not implementation patterns in source code, setting our research apart from other pattern-based learning research.

Lieberman's research [3] discusses the concept of an "Interface Builder," a graphical tool assisting a developer in designing a user interface. He discusses *Programming by Example,* where the developer "takes on the role of operating the user interface in the same manner as the intended end-user would, interacting with the on-screen interface components to demonstrate concrete examples of how to use the interface." The application then learns and generalizes the developer's input to guess at the desired functionality. This research may prove invaluable to future research where the "smart interface builder" discussed above is being designed.

Papatheodorou's research [11] focuses chiefly on using machine learning to learn over time the sort of interface the user may expect and to adapt to that knowledge. It may be possible to use aspects of [11] to generalize the expectations of a range of users or potential users during the design process rather than after deployment, saving developers valuable time to test and improve their software. Our work, however, proposes a different approach to interface learning, requiring no such lengthy data collection process from users. We learn from freely available data, speeding up empirical research and eschewing the technical challenges and privacy concerns inherent in collecting data directly from users.

A promising, unique opportunity to combine the machine learning and user experience fields is missing in available research. We hope to open the door for future research to use machine learning and data mining to analyze a wealth of existing information about user interfaces. This will help developers of all platforms to better understand their users' preferences and peeves not only in graphical or mobile environment design but also in the design of overall user experience. With more intuition on user propensity and preference, developers can design more natural, intuitive software.

VII. FUTURE WORK

Our model would determine the likely rating for a graphical interface given the analyzed source of the application. Given a prototype GUI designed by a developer in an interface builder, the model could guess at a rating for the design based on trends it found in other applications in the same category.

It could also be possible to use a genetic algorithm to determine better positions and attributes for the interface elements in the GUI. The algorithm would weigh a high rating (as determined by the model) against changing the developer's design as little as possible, creating incremental changes to generate potentially optimized versions of the developer's original layout.

VIII. CONCLUSION

We have discussed a method for learning the correlation between certain elements of GUI design, which we will analyze with AGUILLE, and Google Play ratings, which we have mined from the Web. This is improved when factoring in Google Play metadata and discretizing applications by category. Because of the importance of optimized, intuitive interface on smaller devices, developers will benefit from insight from a model that attempts to explain user behavior and preference.

A summary of specific results should go here. It will mirror the summary that will end up in the introduction.

We are confident that further (perhaps automated) probing will continue to reveal interesting relationships between different elements. After learning our model, we could predict the quality of future interfaces. With future refinement, the algorithm could suggest interface improvements by means of a genetic process in an interactive interface builder.

REFERENCES

[1] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 83–92.

[2] M. DeGusta, "Android orphans: Visualizing a sad history of support," 2011.

[3] H. Lieberman, "Computer-aided design of user interfaces by example," in *Computer-Aided Design of User Interfaces III*. Springer, 2002, pp. 1–12.

[4] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 123–134.

[5] R. Ferenc, A. Beszédes, L. Fülöp, and J. Lele, "Design pattern mining enhanced by machine learning," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 295–304.

[6] S. Arlt, S. Pahl, C. Bertolini, and M. Schäf, "Trends in model-based gui testing," *Advances in Computers*, vol. 86, pp. 183–222, 2012.

[7] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[8] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 77–83.

[9] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[11] C. Papatheodorou, "Machine learning in user modeling," in *Machine Learning and Its Applications*. Springer, 2001, pp. 286–294.

Learning User Behavior for Mobile Test Suite Adequacy

Cody Kinneer
Allegheny College
Email: kinneerc@allegheny.edu

***Abstract*—Software development for mobile devices proceeds at a rapid pace. Software as a service, rapid development, and agile programming means that mobile applications are released and updated quickly. As a result, developers have less time to test their applications and cannot completely know the effects of a change. Existing test suite adequacy criteria are insufficient in this quickly changing environment.**

In this paper, we develop a new behavior based test suite adequacy criterion that adapts to user interactions with an application in the wild. We evaluate the time and space overhead of they system and perform an empirical study analyzing existing Android test suites according to our behavior driven criterion. Our analysis reveals that the two test suites focused testing on infrequently used contexts, achieving behavioral adequacy scores from 12 to 33 percent less than the probabilistic calling context coverage. This shows the potential for substantial improvement in the development of test suites for mobile applications.

I. Introduction

Developers frequently use test suites, a collection of test cases, to ensure that the component under test performs according to specification, or to ensure that accuracy does not change over time. A test suite's usefulness lies in its ability to detect problems. With the rise of software as a service and rapid development practices, test suites must be effective in detecting important problems quickly. However, since it is not known beforehand where a problem will occur, determining the adequacy of a test suite is a challenging problem.

The most common test suite adequacy criterion is structural coverage. These criteria, such as line or block coverage seek to maximize the amount of code exercised by a test. Since a bug needs to be executed to be exposed by a test, maximizing structural coverage is a reasonable strategy. However, this definition of test suite adequacy suffers from not taking into account the importance of the structure covered. Achieving complete test coverage in practice is most often wishful thinking, and structural adequacy fails to provide insight into what areas of the application are more important to test. Furthermore, a good test should resemble the conditions under which the application will actually be used, but structural techniques say nothing about the realism of a test.

Another approach to test adequacy is fault-finding. This consists of introducing faults into an application, and then determining which tests tend to find the greatest number of faults. Mutation testing is one such technique. However, in practice, mutation testing speaks to the ability of a test to find faults in a certain structure. It cannot tell us what structures are more important to search for faults in.

With the rise of new software engineering paradigms such as software as a service, agile, and rapid development, these criteria fail to keep up with the pace of software development. This is particularly true of Android applications. According to Android's website, there are 7 Android API's in use [1]. The rate of change of the Android OS itself is a testament to the rapid development of Android applications.

These adequacy measures could be improved by taking into account the way users interact with the application after it is deployed. A behavior driven adequacy criterion confers two benefits. Firstly, if the purpose of application is to be used by a user base, then more frequently utilized components are more important than those that are less frequently used. A problem that occurs in a more frequent use area will affect more users. Additionally, a test suite's similarity to observed user behavior favors tests that are more similar to the conditions that the application will be exposed to in the wild.

Previous work in model-based software testing applied Markov chain models to software testing [2], [3], [4]. These works discuss how a Markov chain used to model software usage could be useful for input generation, software specification, and statistical software testing. However, they do not address the issue of how the model should be generated, and do not focus on test suite adequacy.

In this paper, we present a new test suite adequacy criterion that takes into account learned user behavior in the wild. By collecting data from users actually interacting with an application, we learn a Markov chain that models user behavior. This model can be continuously updated to respond to changes in the users' interactions. We then determine a test suites adequacy by its similarity to the constructed user behavior model.

We seek to determine how well test suites for Android applications reflect real user behavior. We evaluate our technique in terms of time and space overhead for a collection of Android applications, and evaluate the test suite adequacy of the applications using our proposed criterion.

The contributions of this paper are therefore as follows,

- A new behavior driven test suite adequacy criterion (section III).
- An implementation of the criterion for Android applications (section III and section IV).
- An empirical study evaluating the overhead of the implementation (section IV).

- An evaluation of several Android applications' test suites using the new criterion (section IV).

II. Background

To calculate behavioral adequacy, a technique called probabilistic calling context [5] is used for profiling and a Markov chain is used for modeling.

Profiling

Calculating a behavioral criterion requires that an application's behaviors be profiled. A program's behavior can be thought of as the collection of its function calls, which makes profiling based on these calls a reasonable choice for modeling application behavior.

Probabilistic calling context (PCC) is a profiling strategy developed by Bond and McKinley [5]. PCC attempts to assign an integer to every unique stack state. This system is useful because it can be computed efficiently, only 3% overhead is reported in the literature. An example of a stack state that could be represented by PCC is shown in Figure 1. This figure shows an example stack state inspired by the K-9 Mail email application. First, MAIN is called, which is assigned a PCC value of zero. Then, MAIN calls CHECKEMAIL, (hereafter, we will use $\rightarrow$ to signify a function call). The next PCC is calculated from the last PCC and the name of the CHECKEMAIL function. Thus, the next PCC value is meant to represent the sequence MAIN $\rightarrow$ CHECKEMAIL. If CHECKEMAIL were to return to MAIN, then the PCC value would also return to zero. Instead however, CHECKEMAIL calls ITERATEACCTS, so the next PCC value is calcuated from the previous PCC value and the next function name. When a function is called, the next PCC value is given by

$$nextPCC = currentPCC * 3 + currentContext$$

where $currentContext$ is an integer that represents the current context, such as a hash value of the called function name.

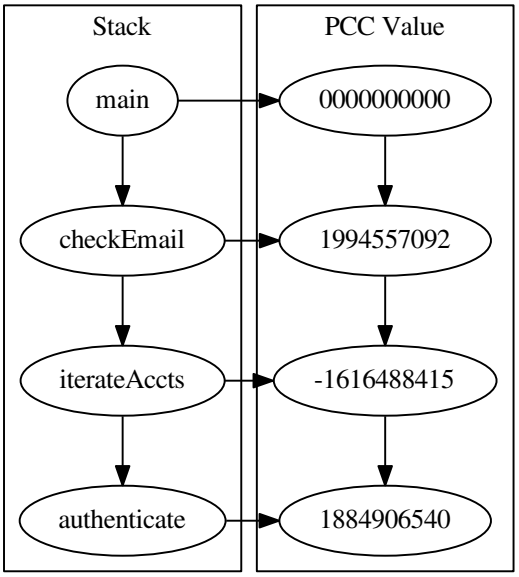

Fig. 1. PCC value updating as methods are added to the stack.

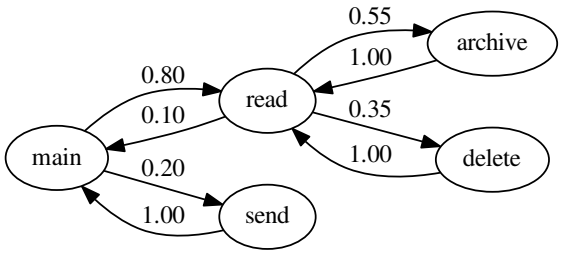

Fig. 2. An example of a Markov chain behavior model inspired by K-9 Mail.

Markov Chains

A Markov chain is a state based system where the next state depends only upon the current state [6]. An example is shown in Figure 2. The nodes in the graph represent states, and the edges represent the transition probabilities. Starting at the MAIN state, there is an 80% chance of transitioning to the READ state and a 20% chance of transitioning to the SEND state.

Markov chains have been used to model expected user behavior in model based testing [2], [4], [3]. However, these techniques generally do not learn models from user behavior, but reflect how the developer expects users to behave.

III. Behavior Driven Test Suite Adequacy

Using PCC and Markov chains, we present a technique for assessing a test suite's adequacy based on how users interact with the application being tested. Our technique for calculating behavioral adequacy is shown in Figure 3.

First, the application is instrumented to collect data for profiling user behavior. Because the application will be used while this information is collected, the overhead incurred by the user must be acceptably small. Additionally, the information gathered must be useful in modeling user behavior. PCC was chosen because it satisfies both of these requirements. A program's behavior can be thought of as the sequence of functions that it calls, which makes profiling based on function calls an appealing strategy for profiling behavior. Since PCC takes into account the functions on the stack, it provides more information than simply profiling based on function frequency, while still maintaining low overhead. The instrumentation calculates the current PCC value from the calling context, and records each transition between PCCs. The application is then released for use by the user base, and behavioral data is collected in the form of these PCC transitions.

The developer then runs the application's test suites on the instrumented application. The transitions between PCCs

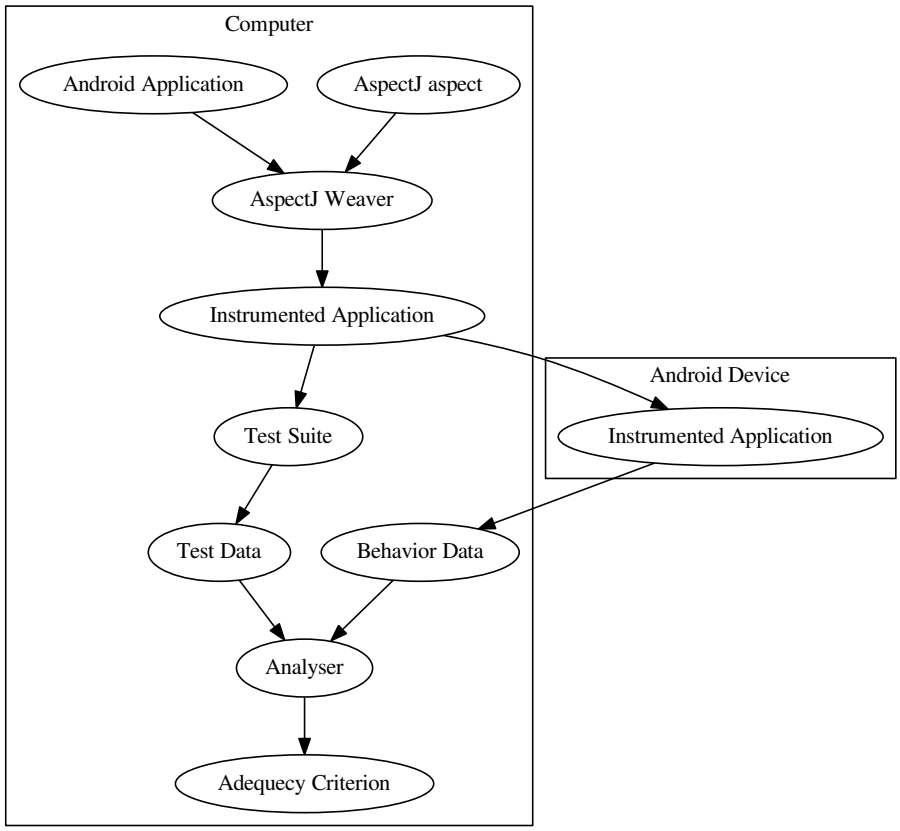

Fig. 3. Framework for a behavior driven test suite adequacy criterion.

observed during testing are recorded as well, giving test data that characterizes the behavior of the test suite in terms of the observed PCC transitions.

The user data is then aggregated and used to construct a Markov chain where the nodes are PCC values and the edges are the probability that a PCC value will transition into another PCC value.

For example, suppose the data in the table below was collected from users interacting with a simple email application. Caller PCC represents the current PCC value, and callee PCC represents the next PCC value. Rather than integer representations, the names of functions on the stack are given for explanatory purposes.

Caller PCC	Callee PCC	Frequency
MAIN	MAIN → READ	80
MAIN	MAIN → SEND	20
MAIN → SEND	MAIN	20
MAIN → READ	MAIN	8
MAIN → READ	MAIN → READ → ARCHIVE	44
MAIN → READ	MAIN → READ → DELETE	28
MAIN → READ → DELETE	MAIN → READ	28
MAIN → READ → ARCHIVE	MAIN → READ	44

This data would be converted to a Markov chain similar to what is shown in Figure 2. In the example, the nodes are function calls rather than PCC values for the sake of explanation. For example, the ARCHIVE node represents the context MAIN → READ → ARCHIVE. If the archive function could be called in a different context, for example,MAIN → SEND → ARCHIVE, that context would be represented by a different PCC value. However, in this simplified example, each function can only be called in one context.

For every caller PCC, there is an edge to each callee PCC. The transition probabilities can be found by taking the frequency of a given callee PCC divided by the sum of the frequencies for the corresponding caller PCC. For example, for the caller PCC MAIN, there are two possible callee PCCs. The probability of transitioning to MAIN → READ is $\frac{80}{80+20} = 0.80$. Alternatively, the probability of transitioning to MAIN → SEND is $\frac{20}{80+20} = 0.20$.

To calculate test suite adequacy, the sum of edges in the model observed during testing is divided by the sum of all edges in the model. For example, consider the example Markov chain shown in Figure 2.

If this Markov chain were constructed from user behavior, then when the application was in the hands of users, READ is called from MAIN 80% of the time while SEND is called 20% of the time. If during testing we exercised MAIN → READ,

then we would have a behavioral adequacy of:

$$\frac{.8}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .16$$

If instead, we tested MAIN → READ and READ → DELETE, then our adequacy increases because we are covering more code.

$$\frac{.8 + .35}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .23$$

However, if we test MAIN → READ and READ → ARCHIVE instead, then our adequacy increases further because ARCHIVE is more likely to be used than DELETE.

$$\frac{.8 + .55}{.8 + .55 + .35 + .2 + .1 + 1 + 1 + 1} = .27$$

This makes sense intuitively since testing more behaviors increases the score, and testing more frequently used behaviors further increases the score. Using this criterion, 100% adequacy is achieved when every behavior observed by the user-base is tested, and 0% is achieved when no behavior observed in the user-base is tested.

IV. Empirical Evaluation

To evaluate our proposed test suite adequacy criterion, we implemented a system for calculating behavioral adequacy for Android applications. The goals of the evaluation are as follows.

1) Determine the time and space overhead of the online behavioral profiling.
2) Determine the overhead associated with calculating behavioral adequacy offline.
3) Evaluate the behavioral adequacy of existing Android applications.

A. Experiment Design

To instrument applications, we used AspectJ because it provides a way to quickly instrument Java and Android applications. An AspectJ aspect was written to calculate the PCC value of the application at function calls. Only application defined functions were considered, so Android system calls and Java library calls were ignored. At each call, the current PCC, the caller, and the next PCC, the callee, were stored as a 64 bit value identifying a transition between PCC values. The frequency of these transitions were recorded, and written to a file upon an activity being paused, stopped, or destroyed. This data was then sent to a desktop PC for processing. A Java program was implemented to construct a Markov chain from the PCC edges. The test suite under study was then executed and the PCC edges collected on a per test basis. For structural coverage, Android's included EMMA tool was used.

Offline tasks were completed using a desktop running Centos 6.5 with a quad-core 1.6GHz CPU and 16MB of memory. User data was collected on an Asus Nexus 7 tablet running Android 4.3 and a Samsung Galaxy SIII smartphone running Android 4.1.1.

B. Case Studies

To conduct our evaluation, we selected several applications from the F-Droid open source appstore. We attempted to select well known applications with large test suites, however this was difficult since few applications contained test suites. The applications selected were K-9 Mail, and Github. K-9 Mail is an email application that can connect to IMAP, POP3, and SMTP servers to manage a user's email accounts. Github is an application that allows a Github user to interact with Github on an Android device. It supports browsing repositories, commenting, and creating Gists and issues.

Application	Files	Classes	Methods	Lines
K-9 Mail	230	806	5671	35410
Github	?	?	?	?

C. Metrics

We evaluate runtime overhead in terms of percent change in time. Space overhead in terms of percent change in source code occupied space on disk. Structural coverage is given in percent of code covered. Behavioral coverage is given as the sum of exercised edges over the sum of all edges.

D. Experimental Results

To evaluate online runtime overhead, the benchmarks' test suites were executed five times, and the execution time was measured with and without profiling instrumentation. The average of the five trials was taken. The table below shows the results, time is given in seconds.

Application	Time Uninstrumented	Time Instrumented	Percent Change
K-9 Mail	4.482	10.385	132
Github	66.006	70.445	7

The large difference in percent change between the applications warrants additional investigation. A possible explanation is that K-9's test suite primarily tests backend code that tends to complete very quickly, whereas Github's test suite involves testing UI elements, such as creating activities that is less sensitive to the instrumentation.

To evaluate space overhead, the benchmarks binary size was measured before and after instrumentation. Size is given in megabytes.

Application	Size Uninstrumented	Size Instrumented	Percent Change
K-9 Mail	2.91	3.35	15
Github	1.76	1.99	13

The size overhead between the two applications was about the same, with both applications using around 14% more disk space when instrumented.

To evaluate offline overhead, we measured the time needed to build the model from user data, and determine behavioral adequacy from the model.

To evaluate Android application test suites, we instrumented the benchmarks and allowed two users to interact with the applications for one day. Afterwards, we profiled the benchmarks' test suites, and constructed a model from the user data. We then calculated behavioral adequacy from the model and test data. For comparison, we determined the structural

coverage of the benchmarks' test suites using EMMA, and the PCC coverage by dividing the number of PCC transitions exercised by both the users and the tests and the number of PCC transitions exercised by the users.

Application	Behavioral Coverage	PCC Coverage	Method Coverage
K-9 Mail	0.00016	0.00024	7
Github	0.03824	0.04324	?

E. Threats to Validity

The most significant thread to the validity of our evaluation is the limited number of applications tested. The applications selected for the evaluation may not represent all Android applications. This problem can be alleviated by conducting a larger study on a wider range of applications. Another threat is the limited number of users participating in the study. The users participating in the study may not be representative of the rest of the user-base. The more the users interact with an application, the more likely they are to exercise PCC values not seen during testing, and thus decrease the score. This means users interacting with an application longer than normal will likely cause the behavioral adequacy to decrease. Additionally, users interacting with the application for less time than normal could cause the behavioral adequacy results to be too optimistic. Alternatively, since behaviors exercised by the test suite but not by the users are given a score of zero, users exercising very little of the application may cause the score to decrease. This issue can also be mitigated by a larger experiment with many users to increase the chance that the users represent an accurate sample of the larger user-base.

V. Related Works

Relative coverage is an alternative to traditional coverage that takes context into account when determining coverage. This is useful in software as a service systems where only a portion of a larger service is used by an application. From the perspective of the smaller application, some features the larger service provides are not used, and thus, irrelevant. These features do not need to be tested, and therefore should not be considered when determining coverage. Relative coverage excludes these unused feature from the coverage equation.

Miranda and Bertolino's work [7] on Social Coverage is the most similar to our work. They propose a system inspired by relative coverage that determines coverage according to context. Relative coverage systems rely on the developer to select which features are relevant, while social coverage, like us, uses observed user behavior to determine what features are important. Social coverage collects user data and can find similar users. The features used by these users might be relevant to the application. These features are taken into account when calculating social coverage.

The Synoptic system [8] also has similarities to our work. Synoptic is a tool that can infer finite state models from reading execution logs. Like us, they construct a model based on user behavior. However, synoptic requires the application log states, and is thus more suited to a high level model, whereas we model based on calling context. Additionally, we apply the learned model to test suite adequacy while Synoptic focuses on analysing logs.

The Gamma system presented by Orso et al. [9] attempts to enable remote monitoring of software after its deployment. Gamma does attempt to address the issue of runtime overhead, and allows for the costs of instrumentation to be shared among users. The developer can specify what type of information they are interested in, and the Gamma system divides the task of collecting this information among the userbase. Additionally, Gamma allows for its instrumentation to be modified by an update.

Bond and McKinley [5] introduced a technique for decreasing the costs of tracking a programs calling context called probabilistic calling context. This system allows a calling context to be represented as an integer that is easy to calculate and well suited to anomaly detection applications. The technique consists of a function that takes as input the current probabilistic calling context and an integer representation of the current context. It then outputs an integer representing the current probabilistic calling context. The function produces outputs that are uniformly distributed, so that the chance of conflict is low, and the order of the contexts is taken into account.

In a later work, Bond et al. [10] present a technique for calculating the entire calling context from the probabilistic calling context. This technique has the advantage of being able to reconstruct the calling context offline, however, some dynamic information is needed make a search of the context space feasible, which requires additional overhead of 10-20%.

Elbaum et al. [11] present a study showing how software evolution affects code coverage. The study shows how even small changes can have a large impact. This work is similar to ours in that we are concerned with the performance of structural adequacy criteria in evolving software environments.

Whittaker presented a series of papers [2], [3], [4] on using markov chains in software testing, including input generation, software specification, and statistical software testing.

Andrews et al. [12] present a paper analyzing the usefulness of mutation testing. The study shows that mutation testing creates faults similar to real faults. The abc compiler provides a way to perform AspectJ instrumentation on Android bytecode without access to the source code [13].

VI. Conclusion

Android applications exist in a rapidly changing environment. Traditional test suite adequacy criteria such as structural coverage and fault finding adequacy provide insufficient guidance to developers in such a rapid development cycle. As an alternative, we propose a behavior driven test suite adequacy criterion that can adapt to changes in the environment when assessing an applications test suite. By instrumenting behavior on applications running in the wild, a markov chain is constructed that models user behavior. This behavioral data is then compared with data obtained during the execution of a test suite to determine the test suites adequacy. A case study of two applications suggests that there is potential for major improvement in the quality of test cases for mobile applications. A more comprehensive empirical study is needed to explore the technique's run-time overhead and evaluate the adequacy of additional application's test suites.

ACKNOWLEDGMENT

This work is supported by NSF REU Grant 1359275.

REFERENCES

[1] Android, "Dashboards," http://developer.android.com/about/dashboards/index.html, Jul. 2014.

[2] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 1, pp. 93–106, Jan. 1993.

[3] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 10, pp. 812–824, Oct. 1994.

[4] J. Whittaker, "Stochastic software testing," *Annals of Software Engineering*, vol. 4, no. 1, pp. 115–131, 1997.

[5] M. D. Bond and K. S. McKinley, "Probabilistic calling context," *SIGPLAN Not.*, vol. 42, no. 10, pp. 97–112, Oct. 2007.

[6] J. G. Kemeny and J. L. Snell, *Finite markov chains*. van Nostrand Princeton, NJ, 1960, vol. 356.

[7] B. Miranda and A. Bertolino, "Social coverage for customized test adequacy and selection criteria," in *Proceedings of the 9th International Workshop on Automation of Software Test*, ser. AST 2014. New York, NY, USA: ACM, 2014, pp. 22–28.

[8] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, "Synoptic: Studying logged behavior with inferred models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 448–451.

[9] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: Continuous evolution of software after deployment," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 65–69.

[10] M. D. Bond, G. Z. Baker, and S. Z. Guyer, "Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses," *SIGPLAN Not.*, vol. 45, no. 6, pp. 13–24, Jun. 2010.

[11] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 170–.

[12] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 402–411.

[13] S. Arzt, S. Rasthofer, and E. Bodden, "Instrumenting android and java applications as easy as abc," in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Springer Berlin Heidelberg, 2013, vol. 8174, pp. 364–381.

Diagnosis of Autism Spectrum Disorders Using an Interactive Diagnosis Program

Tate Krejci, *Student, UCCS*

***Abstract*—Asperger Spectrum Disorders (ASD) affect a relatively large portion of the population, causing difficulty in learning appropriate behaviors for various social situations. Tests to diagnose ASD require an expert, and symptoms can often be mistaken for other mental disorders leading to under-diagnosis. Therefore, the application of a machine learning algorithm in an interactive environment such as a program will potentially increase the amount of successful diagnoses of ASD. The successful implementation of such a program will not only increase the likelihood of successfully diagnosing ASD, but also increase our understanding of ASD.**

I. Introduction

Autism Spectrum Disorders affect approximately one in two hundred and fifty people, causing difficulty in the acquisition and understanding of normal social protocols [1]. Many cases go unnoticed or misdiagnosed as there is no definitive way to diagnose an ASD over other mental disorders without excessive trial and error [2]. Furthermore, early detection of ASD in children requires expert evaluation, and cannot easily be carried out by parents or teachers [3]. Therefore, an easily and cheaply distributable application for the detection of ASD using a machine learning algorithm has the potential to detect more cases of ASD at earlier ages, and potentially provide further insights into symptoms of ASD. A program can yield potentially greater results as a program can be tailored to be interactive and captivating to its target age group. This allows for greater amounts of data to be collected than if the application was of a less interactive nature.

The latest edition of the DSM (Diagnostic and Statistical Manual of Mental Disorders), the DSM 5 has grouped the previously distinct disorders of Asperger Syndrome and Autism into the same disorder, known as an Autism Spectrum Disorder (ASD). Because of this, it is vital to determine where on the spectrum an afflicted person lies to ensure they receive the help that they specifically need. A person with severe ASD will demonstrate symptoms commonly associated with Autism, while a person with mild ASD will have symptoms similar to Asperger Syndrome. Differentiating the patients based on severity will ensure the correct type of help is provided. Thus an interactive program that can not only differentiate between a person with ASD and one without it, but can also provide insight into the severity of a patients disorder will prove to be a valuable tool.

II. Previous Work

The detection of mental disorders through the use of programs has been considered before and effectively applied to children with ADHD, exhibiting a success rate of approximately seventy-five percent with the use of a machine learning algorithm [4]. The bulk of the work done on ASD has been to identify the symptoms of ASD, the predominant one being inability to learn proper social protocol through normal social interactions [5]. However mild ASD remains harder to diagnose than severe ASD as the signs are far more subtle, especially for those with high functioning Autism [6]. Furthermore, symptoms of ASD can have multiple implications, making determining if a disorder is in fact ASD difficult [7]. More symptoms of ASD include unusual patterns of interest and behavior often leading to children with ASD seeming distant or inattentive [8]. While the symptoms of ASD are well known and progress has been made in its diagnosis, there still exists no definitive way to determine if a disorder is within the Autism spectrum of disorders or something else entirely.

III. Social Learning Theory and ASD

The primary function of ASD is to impair the ability of those affected to learn appropriate social behaviors the way unaffected individuals learn. Social Learning Theory is the theory that explains by what methods people learn what social behaviors are acceptable and what behaviors are not, though currently little is actually known about how this process actually occurs. Examples of these social behaviors include knowing to look somebody in the eye when they are speaking to you or knowing not to talk over someone else [9]. Detection techniques today involve qualitative question and answer sessions, with little in the way of quantitative data to support one diagnosis over another, often leading to misdiagnosis [2]. This is exacerbated by the fact that there is no medication to treat ASD, so doctors cannot try varying medications to determine the true disorder, as is often the case when diagnosing ADHD. With the spectrum of high IQ ASD to low IQ ASD, doctors and psychologists find it difficult to create a definitive list of symptoms [9], meaning a machine learning algorithm has the potential to discover new patterns to assist in the diagnosis of ASD.

IV. Methodology

A. The program and Machine Learning Algorithms

To identify quantitative indicators of ASD, it will be necessary to use a supervised machine learning algorithm to group data collected during the program. The type of algorithm used will depend on what kind of data the program will collect, although it is likely that a type of clustering algorithm will help group ASD users together and help identify them. This will

hopefully allow the algorithm to discover new data sets that group together people suffering from ASD. Because of this, the testing phase of the program will be of utmost importance to ensure a large enough data set is collected to effectively predict whether a person has an ASD. To collect data, the program can measure variables such as answers provided, response time and mouse movements. The program will collect this data when the user is presented with social situations in which the correct response will not be readily evident to a person with an ASD. During early trials, the program can be tuned to give the maximum amount of data per encounter, and additional features can be added as needed. Programs of this style have already been implemented for the diagnosis of ADHD, with a success rate of approximately seventy-five percent [4].

B. Creating the Tests and Data Sets

Unlike some machine learning projects, total control over the data collection will be possible in this project. This means that designing the tests in the program will be as important, if not more important than fine tuning a machine learning algorithm. If the tests do not collect pertinent data that distinguishes people suffering from ASD, it is unlikely that even a well tuned algorithm could give a meaningful prediction. Therefore, extensive testing of the tests themselves will be a vital portion of creating a program for the diagnosis of ASD. To ensure the questions are well tuned, initial testing will occur only on people who do not show symptoms of ASD. With this data, it will be possible to determine what questions are effective because people without ASD should answer well written questions in the same way as other people without ASD. Once questions and scenarios have been verified through this method, they can be tested on people with ASD to determine if they can separate them out from regular people.

To create tests that capture relevant data, it has been necessary to partner with experts in psychology, specifically ASD and social learning theory. With their help, it has been possible to create scenarios in the program where the user is presented with options that indicate if they suffer from ASD or not. For example, the user could be presented with a situation where they will make differing choices based on their empathy for the characters in the program. People with ASD will likely show less empathy than those without it, as a lack of empathy is one of the characteristics of ASD [10]. Experts have expedited the process of creating tests that capture data relevant to the diagnosis of ASD. There is also the possibility to test some of the non-social aspects of ASD in a program such as the abnormal ways in which a person with ASD will focus on different tasks. Expert advice has been used to ensure that all the tests that have been implemented so far are true measures of ASD.

There are many symptoms that can indicate ASD such as deficits in executive functions [9]. This makes it possible to assess the severity of ASD in a given person by determining how impaired their executive functions are. Somebody with mild symptoms will likely show less impairment than somebody with severe symptoms [9]. To determine executive function impairment, an ordering section has been implemented where the user is shown multiple pictures of a scene and asked to select them in the order they think is best. This test can be developed to assess ASD specifically by using pictures portraying social interactions. Another test currently in the program shows the user a picture of a social interaction and asks them questions about it. Specifically, it probes the user's knowledge of what the various people in the picture think about one another, which is generally a challenge for people with ASD. The program also keeps track of the time taken to complete each individual question, meaning there is a possibility to filter out people with ASD based on the time taken to complete the various tests. Currently the program also includes the Coolidge Autism Symptoms Survey (CASS) which has already proven to be effective at diagnosing ASD [11]. This means that the program can build off of an already successful tool while adding new methods of diagnosis which are capable of measuring metrics that a pen and paper survey cannot.

C. Targets for the program

ASD manifests in different ways based on the person's age making it important to target a specific age group initially to develop both the program and the algorithm [12]. This will simplify the initial design of the program as it will only have to include tests for that specific age group, and not all possible age groups. For this reason the final program will be targeted toward 3rd grade age students. This is the age when most students have gained sufficient experience reading to take text based tests, and is regarded as the age when the symptoms of mild ASD first become visible [9]. This also means that the program will help those with ASD get help as soon as possible, greatly increasing their quality of life in later years.

An important note is that children in this age group are just beginning to read, so its important that any test targeted at them not accidentally test reading ability and comprehension. To do this it will be necessary to keep the amount of reading needed to a minimum, have a parent help the child, or implement a voice-over feature. For initial testing, a researcher will likely be present to answer any questions about the application, meaning that in initial phases of testing, the amount of reading is not a major concern. This is especially important when analyzing the amount of time it took to complete each question. A voice-over function that could read aloud questions and answers would serve to make the time to read prompts constant, so time taken to complete a question would be due primarily to thinking time. Another viable option is to make the tests use pictures for both prompts and answers, although doing so affects the kinds of data that can be collected. The final program will feature both voice overs and picture based tests to ensure reading ability is not an aspect of the data that is collected.

Due to the difficulty of conducting trials on children, initial tests have been conducted on a number of colleagues (8) to determine if questions are answered consistently by people who can be considered to be free of ASD. This assumption can be made because those undergoing initial testing are using

an application designed for young children, and have a high probability of selecting answers that indicate they do not have ASD. This means that if a specific question is not answered consistently, it is likely a confusing or ambiguous question and should be rethought. Early testing also provides feedback on the general design of the application, all of which will lead to a far more refined application when official trials do begin and gives a partial data set to begin training the learning algorithm. Early testing has also provided valuable insight into how often 'normal' people make mistakes on the tests, which will be valuable information when training the algorithm.

D. Implementation

To make the program easily distributable and appealing to the largest audience, the program has been developed for PCs using a Windows environment. Windows has many well established frameworks for creating interactive programs such as Unity and XNA Studio, simplifying the development of a program for a Windows platform. The program will primarily perform data collection, and if successful diagnosis, but it will prove unwieldy for the program to also store and process data at larger scales. Initially the program will store data locally for simplicity, but later can send data to a data storage system. When the program is completed with the testing phase, it can be deployed to a web based player. This will allow the program to be accessed by a website, removing the need for users to install a piece of software to use it.

The program will consist of two main parts, the test portion and the data processing portion. The testing portion will consist of all the tests designed to gather data to determine a diagnosis. This portion will also include the CASS which can be treated separately internally as a cross check within the program the validity of the results for unknown cases. The second portion will be the data analysis portion which will handle all of the data processing. While the program is in the testing phase, the program will be separate from the testing portion so it can be fine tuned without having use the testing portion. Later, the data processing portion will be added at the end of the testing portion so it can give users an immediate result as well as integrate the data provided by them.

An important feature of the program will be the artwork used, as many of the tests will require specific ideas to be conveyed through pictures. At the start of the project, it is impractical to use custom artwork so images found on the Internet will be used in the early iterations of the program. While these may not convey a message perfectly, early testing should determine their efficacy. From early testing it will be evident what types of image question combinations work the best, and early testing can be based on these findings. If the results of the testing are positive, later versions of the program can include custom artwork. This will allow greater control of the messages conveyed by the pictures and will allow for more customization in the scenarios that are presented. Currently, some scenarios can not be implemented due to restrictions in the types of art available, so attaining custom art will likely increase the performance of the program once it is acquired.

E. Current Tests

Currently two different types of tests have been implemented in the application which each have multiple individual questions. The first type of test presents the user with multiple pictures that together depict a task or social interaction from beginning to end. The user clicks on each tile in the order that they think is best. This test serves to test the user's executive functions, which is lacking in many people with ASD, as well as their understanding of social situations [9]. Both the answers and the time taken to complete the question are stored by the application. Fig. 1 on the following page shows a screen shot of one of the questions in the ordering portion of the application. The second type of test implemented shows the user a picture of some social interaction and gives them a prompt and a series of answers. Some of the answers delve into what the people in the picture are actually thinking, which is how people without ASD should answer. The other answers have less to do with what the people in the pictures are actually thinking, and are likely to be chosen by people with ASD. Fig. 2 on the next page shows a screen shot of one of the questions in the intentions portion of the application. It should be noted that the artwork in the screen shots is not the artwork that will be included in the final iteration of the application. The current artwork was chosen based on availability so that prototyping and initial testing could begin rapidly.

V. Learning Algorithm

The essential function of the program is to determine whether a person has ASD. This means that there are two classes to map results to: ASD or non-ASD. Because of this, it seems a classification algorithm lends itself to the problem. Since classification generally follows a supervised structure, example data from people with and without ASD is necessary. This works out well as determining the efficacy of the various tests will mean testing the program on people with ASD before the learning algorithm is even implemented.

To first determine an effective learning algorithm, it is necessary to consider how the data collected should be processed. The program will consist of various tests with multiple parts, many of which will feature multiple choice questions. It is likely that scoring each different test and using a score from each test as the parameters for classification will be most effective. This way the number of variables is limited. However, if this proves ineffective it may also be possible to use every answer for the classification or perhaps use a regression type algorithm on the results pertaining to individual tests. The key issue will be making the number of parameters small enough for efficiency, while retaining meaning.

A Naive Bayes classifier will be used as the algorithm as it is effective at taking many parameters and calculating the probability of a class based on those. Because it functions by summing the probabilities that each individual attribute leads to a specific class, it will automatically give each answer a weight based on how effective it is at classifying ASD or non-ASD. This will hopefully add even more value to the program, as each test will affect the final outcome based on its efficacy. Another function of a Naive Bayes algorithm is

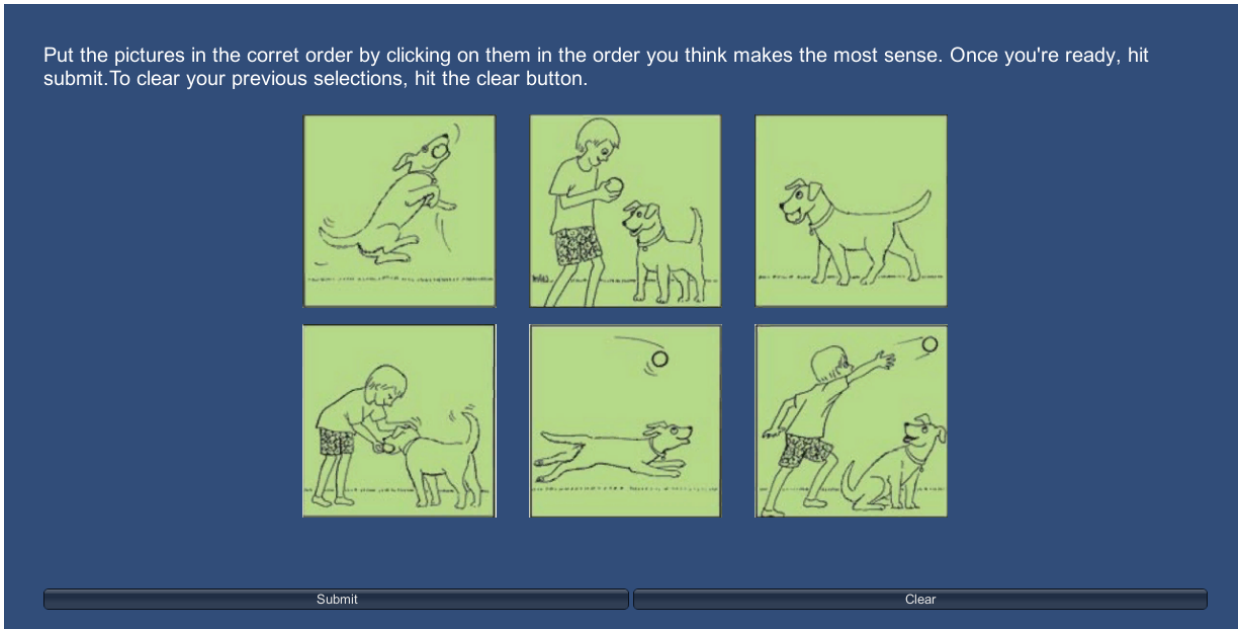

Fig. 1. Ordering Test

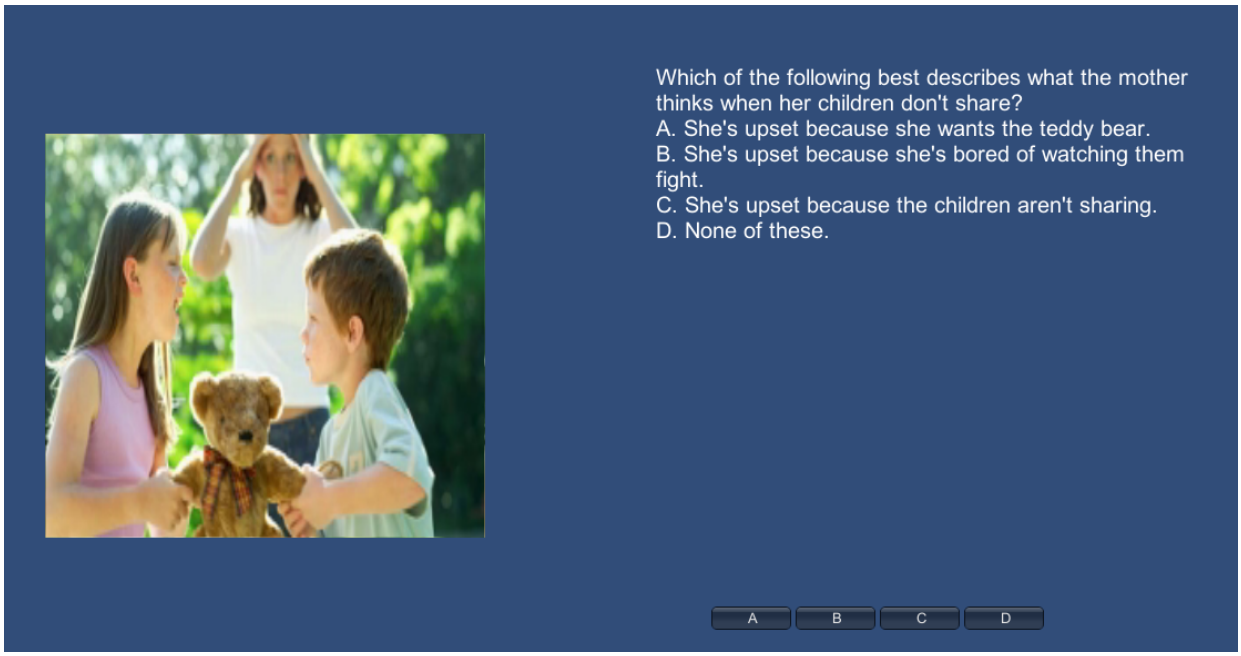

Fig. 2. Intentions Test

that the probability of a given parameter resulting in a specific class is not dependent on previous parameters. This may be seen as a hindrance in some cases however, the answer to one question does not have any impact on the answer to another. In this case, the disregard of previous answers serves to simplify the algorithm, making it more efficient. The fact that prior probabilities are disregarded likely will have no effect on overall results. The equation below shows the Bayes rule, off of which the Naive Bayes algorithm is based.

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

To use this algorithm, you sum the probabilities of each element giving a certain class based on weight. Thus it is possible for it to be effective with large amounts of data and will hopefully give good results when implemented. As mentioned before, it is important not only to classify users as ASD positive or negative, but also give a measure of the severity of their affliction.

The naive assumption of the Bayes algorithm removes the denominator of the equation, representing the assumption that the individual probabilities of each element of the classifier are independent of each other. As the answers to a given question

TABLE I
INITIAL RESULTS FROM NON-ASD SUBJECTS

Question Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	ASD
Participant 0	B	C	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 1	C	C	A	A	B	B	B	B	C	C	A	C	A	A	C	A	C	N
Participant 2	B	C	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 3	B	C	A	A	B	B	B	B	C	C	B	C	A	C	C	A	C	N
Participant 4	C	C	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 5	B	C	A	A	C	B	B	B	C	C	A	C	A	A	C	A	C	N
Participant 6	C	C	A	A	B	B	B	B	C	C	A	C	A	A	C	A	C	N
Participant 7	C	D	A	A	B	B	B	B	C	C	B	C	A	A	C	A	C	N
Participant 8	C	C	A	A	B	B	B	B	C	C	A	C	A	A	C	A	C	N

are unlikely to affect one another, the naive assumption seems a safe assumption to make for this application. To determine the probability of a given class, the probabilities that each individual property lead to a given class are multiplied together and this is then multiplied by the probability of a given class and is shown in the formula below where S is a selection of n attributes.

$$p(A|S) = \sum_{n=1}^{n} p(S_n|A) * p(B)$$

VI. INITIAL RESULTS

At this time, the main tests that have been conducted have been to assess the efficacy of various styles of questions and tests. Currently, children with a diagnosis of ASD are unavailable for testing, so tests have been conducted with colleagues. Because they do not have diagnoses of ASD, testing them provides an insight into the responses of non impaired persons. If the answers provided by them generally match for a specific question, the question is at least effective at grouping users without ASD together. This testing will help filter out questions that are ambiguous and give inconsistent data. When a test has been verified as consistent, more tests can be created based on those. The next step will be to give the refined tests to people with ASD to determine that they are also effective at distinguishing them from non-impaired people. TABLE I shows some of the preliminary data collected from users who do not have ASD using the intentions test. In this test, the user is presented with a picture of people performing various actions. The user is provided a prompt and a selection of answers. The answers each present a different level of social awareness, so people with ASD will likely pick the answers that show less social awareness.

Here the rows represent the answers submitted by each participant and the columns represent each question in one of the tests. For the most part, the answers are the same as expected. Differing answers to the same question indicate an ambiguous question that should be rethought so it is consistently answered the same for all people without ASD. The questions whose answers are all the same represent good questions with a style that should be repeated when adding new questions to this test. It should be noted that isolating good styles is accomplished by using the same prompt and image for a question and just varying the answer choices. The focus for this particular test was on creating distinguishing answers, which is why they were the factors that changed to determine the efficacy.

To ensure that a naive Bayes algorithm is an appropriate choice for the algorithm, test data was generated to represent the answers of users who were suffering from ASD. This data was generated pseudo-randomly with the goal of testing the classifier in mind. These are not results from real people with ASD. When this data was used with the data obtained from colleagues, the algorithm classified eighty-seven percent of the cases correctly. This means that as long as ASD users answer questions in the predicted manner, a naive Bayes classifier will successfully distinguish between ASD and non-ASD users. With the implementation of further tests, and the collection of more data, hopefully this number can be further increased in the future.

VII. TIME LINE

At this point the core functionality of the program has been implemented. Multiple tests are completed and a Naive Bayes classifier has been added to provide in application results for users. Future work will involve collecting real data to train the classifier using children of an appropriate age. Another important task will be to improve the existing tests, and add new tests. More tests will make the application a more comprehensive test of ASD, likely increasing the accuracy of the diagnosis. Existing tests can also be expanded as limited art assets have made the test sections relatively short. With the acquisition of custom art, more tests can be devised and existing tests will have greatly increased accuracy. This is because with custom art, variables that can affect a person's answer can be easily eliminated. Once the application has reached a polished state, and preliminary data has been gathered, the program can be exported to a web player. This will means that more people will have access to the program which will provide more diverse training data and hopefully further improve the algorithm's accuracy.

VIII. GOALS

The completion of the prototype of the program will allow for small scale data collection and testing of the program. This will involve identifying a test group, and determining what member of that group suffer from ASD. From there, a naive Bayes algorithm can be applied to the data and used to identify patterns indicative of ASD. If the algorithm can successfully predict ASD in the targeted age group, it can be deployed on

a larger scale to collect more data to improve the algorithm and it can be modified to support various target age groups. With enough success, the program could be further modified for use as not only a diagnostic tool, but as a treatment for patients with ASD.

Deployment to a larger scale will involve extensive testing and polishing of the existing program to come up with a complete suite of tests measuring many different aspects of ASD in distinct ways. Once the program reaches this point, and with university approval, the program can be exported to a web browser so it can be taken online and collect data online. It can then be modified to also give a suggestion of a diagnosis of ASD so it can be used by real people online. If the online program proves successful, it may be possible to create more diagnostic tools for the various mental illnesses that exist.

IX. Conclusion

The successful implementation of a machine learning algorithm to diagnose ASD will provide parents and doctors with a more effective means of diagnosing and helping those suffering from ASD. It also has the potential to vastly increase our understanding of the symptoms of ASD and perhaps provide clues to its causes and increase our understanding of social learning. The use of a program as the primary platform for the test will increase patient interaction with the application, leading to a greater quality and quantity of the data collected.

References

[1] B. J. Tonge, A. V. Brereton, K. M. Gray, and S. L. Einfeld, "Behavioural and emotional disturbance in high-functioning autism and asperger syndrome," *Autism*, vol. 3, no. 2, pp. 117–130, 1999.

[2] B. G. Haskins and J. A. Silva, "Asperger's disorder and criminal behavior: forensic-psychiatric considerations," *Journal of the American Academy of Psychiatry and the Law Online*, vol. 34, no. 3, pp. 374–384, 2006.

[3] O. Teitelbaum, T. Benton, P. K. Shah, A. Prince, J. L. Kelly, and P. Teitelbaum, "Eshkol–wachman movement notation in diagnosis: The early detection of asperger's syndrome," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. 32, pp. 11 909–11 914, 2004.

[4] S. Srivastava, M. Heller, J. Srivastava, M. Kurt Roots, and J. Schumann, "Tangible games for diagnosing adhd–clinical trial results."

[5] L. Wing, "Asperger's syndrome: a clinical account." *Psychological medicine*, 1981.

[6] P. Howlin and A. Asgharian, "The diagnosis of autism and asperger syndrome: findings from a survey of 770 families," *Developmental Medicine & Child Neurology*, vol. 41, no. 12, pp. 834–839, 1999.

[7] D. V. Bishop, "Autism, asperger's syndrome and semantic-pragmatic disorder: Where are the boundaries?" *International Journal of Language & Communication Disorders*, vol. 24, no. 2, pp. 107–121, 1989.

[8] A. Klin and F. R. Volkmar, "Asperger's syndrome," *Handbook of autism and pervasive developmental disorders*, vol. 2, pp. 88–125, 1997.

[9] M. G. Winner, *Thinking About You Thinking About Me*. Think Social Pub, 2007.

[10] I. Dziobek, K. Rogers, S. Fleck, M. Bahnemann, H. R. Heekeren, O. T. Wolf, and A. Convit, "Dissociation of cognitive and emotional empathy in adults with asperger syndrome using the multifaceted empathy test (met)," *Journal of autism and developmental disorders*, vol. 38, no. 3, pp. 464–473, 2008.

[11] C. S. R. D. L. S. Frederick L. Coolidge, Peter D. Marle and P. Monaghan, "Psychometric properties of a new measure to assess autism spectrum disorder in dsm-5," *American Journal of Orthopsychiatry*, vol. 83, no. 1, p. 126–130, 2013.

[12] C. Koning and J. Magill-Evans, "Social and language skills in adolescent boys with asperger syndrome," *Autism*, vol. 5, no. 1, pp. 23–36, 2001.

Simplified Statement Extraction Using Machine Learning Techniques

Conor McGrory, Princeton University

***Abstract*—The automatic generation of basic, factual questions from a single sentence of text is a problem in the field of natural language processing (NLP) that has received a considerable amount of attention in the past five years. Some studies have suggested splitting this problem into two parts: first, decomposing the source sentence into a set of smaller, simple sentences, and then transforming each of these sentences into a question. This paper outlines a novel method for the first part, combining two techniques recently developed for related NLP problems. Our method uses a trained classifier to determine which phrases of the source sentence are potential answers to questions, and then creates different compressions of the sentence for each one.**

I. Introduction

Asking questions is one of the most fundamental ways that human beings use natural language. When someone studies a foreign language, many of the first utterances they learn are basic questions. The ability of a speaker to form a grammatical question — to request a specific piece of information from another party — is indispensable in most practical situations involving basic communication. Over the past five years, there has been a significant amount of new research towards developing computer systems that can automatically generate basic questions from input text. This is referred to in the literature as the problem of Question Generation (QG), and it has many potential applications in education, including the development of computerized tutoring systems and the generation of basic reading comprehension questions for elementary-level students. Although some studies in the past have tried to generate questions based on whole blocks of text [1], the majority of recent work done on QG has focused on the problem of generating factual questions from a single sentence of input.

Early attempts to solve this problem used complicated sets of grammatical rules to transform the input sentence directly into a question [2]. However, in 2010, Heilman and Smith [3] suggested separating the problem into two steps: first, simplifying the source sentence, and then transforming it into a question. The advantage of this approach is that grammatical rules are much better at transforming simple sentences into questions than they are at transforming complex ones. Our paper outlines a method for preforming the first step in this process, which we refer to as the problem of Simplified Statement Extraction (SSE).

Conor McGrory is participating in a National Science Foundation REU at the University of Colorado at Colorado Springs, Colorado Springs, CO, 80918. e-mail: cmcgrory@princeton.edu

II. Prior Work

In a paper also published in 2010 [4], Heilman and Smith developed a rule-based SSE algorithm that extracted multiple simple sentences from a source sentence. This algorithm recursively applied a set of transformations to the a phrase structure tree representation of the input sentence to generate the simple statements. By extracting multiple simplified statements from the source sentence, they greatly increased the number of possible questions that could be generated and the percentage of words from the input sentence that appeared in one of the output statements [4].

Two problems in NLP that are related to QG are cloze question generation and sentence compression. A cloze question is a type of question commonly used to test a student's comprehension of a text, where the student is asked, after reading the text, to complete a given sentence by filling in a blank with the correct word. One example could be the question

A _____ is a conceptual device used in computer science as a universal model of computing processes.

In this case, the answer would be *Turing machine*. Because these questions are commonly used in testing, and require no syntactic rearrangement of the source sentence (just deletion of a specific phrase), they seem like an easy place to apply QG techniques. However, selecting which phrase or phrases in the sentence to delete is somewhat difficult. A question like

A Turing Machine ____ a conceptual device used in computer science as a universal model of computing processes.

with the verb *is* as the answer would be completely useless to a student interested in testing their knowledge of basic computer science. An automatic cloze question generator needs to have some way of distinguishing informative questions from extraneous ones. Because the quality of a cloze question can depend on complicated relationships between a large number of factors (syntax, semantics, etc.), distinguishing quality of a question is a good task for a machine learning system. Becker et al.[5] did this by training a logistic regression classifier on a corpus of questions paired with human judgements of their quality. The classifier was able to identify 83 percent of the high-quality sentences correctly and only misidentified 19 percent of low-quality questions as high quality[5].

Sentence compression is the problem of transforming an input sentence into a shorter version that is grammatical and retains the most important semantic elements of the original. This can be used to generate summaries or headlines for large blocks of text. Various methods have been developed to attack this problem. Knight and Marcu [6] used a statistical language

model where the input sentence is treated as a noisy channel and the compression is the signal, while Clarke and Lapata [7] used a large set of constituency parse tree manipulation rules to generate compressions.

Filippova and Strube [8] developed a sentence compression system where the compressed sentence is generated by pruning the dependency parse tree of the input sentence. Using the Tipster corpus, they calculated the conditional probabilities of specific dependencies occurring after a given head word. These were used, in combination with data on the frequencies of the words themselves, to calculate a score for each dependency in the tree. They then formulated the problem of compressing the sentence as an integer linear program. Each variable corresponded to a dependency in the tree. A value of 1 meant the dependent word of that dependency would be preserved in the compression, and a value of 0 meant that it would be deleted. Constraints were added to the linear program to restrict the structure and length of the compression, and the objective function set to be maximized was the sum of the scores of the preserved dependencies.

The central assumption made by Fillippova and Strube's method is that the frequency with which a particular dependency occurs after a given word is a good indicator of its grammatical necessity. For example, transitive verbs like *chase* require direct objects, so the frequency of the *dobj* dependency after the head word *chase* in the corpus is very high. Although *chase* can also be the governor of a prepositional phrase, this is not grammatically necessary, so there will be many more instances in the corpus where *chase* does not govern a prepositional phrase, resulting in the frequency of the *prep* dependency after *chase* to be lower.

III. Problem Definition

In explaining our system, it will help to have a formal definition of the problem. We will define the problem of simplified statement extraction as follows:

For a source sentence S, create a set of simplified statements $\{s_i ... s_n\}$ that are semantic entailments of S. A sentence is considered to be a *simplified statement* if it is a declarative sentence (a statement) that can be directly transformed into a question-answer pair (QA pair) without any compression. Ideally, the interrogative transformations of the generated $\{s_i\}$ should include as many as possible of the set of QA pairs a human being could generate given S. We will call the ratio of computer-generated, grammatical QA pairs to human-generated QA pairs the *coverage* of the system.

IV. Solution

As Becker et al. [5] showed with their work on cloze questions, there are certain phrases in S that make sense as answers to questions and others that do not. The fundamental idea behind our SSE system is that knowledge of which phrases in S are good answers can inform the compression process, preventing us from missing important information and thereby maximizing coverage. We divide the SSE problem into two parts: first identifying potential answers, and then generating for each of these answers a compression of S where that answer is preserved. These compressions form the set $\{s_i\}$ of simplified statements. Because each one of these statements will ultimately be transformed into a question with the given answer, our goal when compressing for a particular answer is to find the *shortest grammatical compression* of S that contains the given answer. This will ensure that each selected answer is preserved in at least one of the simplified statements and that these statements will contain minimal amounts of extraneous information.

To select potential answers from the input sentence, we use a slightly modified version of Becker et al.'s cloze question generation system [5]. Because all questions are essentially requests for specific pieces of information, determining which phrases in S make good answers to a standard grammatical question is very similar to determining which phrases make good blank spaces for a cloze question. Once we have the set of possible answers, we use a more substantially modified version of Filippova and Strube's dependency tree pruning method [8] to generate the set of shortest grammatical compressions of S that contain each of the answers.

V. Answer Selection

We designed and implemented the answer selection system using the Stanford NLP Toolkit [9] and the Weka machine learning software [10]. It uses the corpus of sentences, QA pairs, and human judgments developed by Becker et al.[5] to train a classifier to find the nodes in the parse tree of the input sentence that are most likely viable answers to questions. Our implementation performs two basic functions. First, it has the ability to read in the corpus, calculate a set of features and determine a final classification for each potential answer, and output this data set as an .arff file (the standard file format used by Weka). When the program needs to find the good answers in an input sentence, it loads the classifier from the file, determines all grammatically possible answer phrases in the input sentence (this is based on a set of constraints given by Becker et al. [5]), and uses the classifier to determine which of these phrases are good answers.

A. Feature Set

The Stanford NLP Toolkit [9] provides us with two very useful tools for describing the grammatical structure of a sentence: a Penn Treebank style constituency parse tree and the Stanford dependency relations [11]. The Stanford dependency relations are a set of grammatical relations between governor and dependent words in a sentence. Some examples include verb-subject, verb-indirect object, noun-modifier, and noun-determiner. Essentially, it is a dependency grammar with more specific information than which words a given word governs and which words it depends on. The relations also have a set hierarchy. For example, the verb-subject, verb-object, and verb-adverbial modifier relations are all instances of the parent relation predicate-argument. This enables the user to work at different levels of detail. For our purposes, we used the 56 basic relations defined in the Stanford library to categorize all of our dependencies.

We used many of the same features as Becker et al.[5] did, but because we used a different NLP package to implement our system (we used Stanford's, they used a toolkit developed by Microsoft), some of our features were significantly different. At this point, we have also implemented far fewer features than they did. Our features can be divided into three basic categories: token count features, syntactic features, and semantic features.

The token count features we used were the exact same as those used by Becker et al. This category contained 5 features which had to do with the length of the answer in comparison to the length of the sentence, like the raw lengths of both and the length of the answer as a percentage of the length of the question.

The syntactic features were calculated using the constituency parse tree. Currently, our system uses three syntactic features: the Penn part-of-speech tag of the word that comes immediately before the answer in the sentence, the tag of the word that comes immediately after, and the set of tags of words contained in the answer phrase.

The semantic features use the Stanford dependencies system and are completely different than the semantic features used by Becker et al. The purpose of these is to determine the grammatical role the answer phrase plays within the sentence. We currently have four semantic features implemented: the dependency relation between the head of the answer phrase and its governor in the sentence, the set of relations between governors in the answer and dependents not in the answer, the set of relations with both governors and dependents in the answer, and the distance in the constituency tree between the answer node and its maximal projection.

B. Classifier

The classifier used in our system is the Weka Logistic classifier [12]. Because each instance is classified as either "Good" or "Bad", this is a binary logistic regression classifier, similar to the one used by Becker et al. However, Becker et. al also used L2 regularization (adding a constant multiple of the L2 norm of the regression coefficients to the error function as a penalty for overfitting), which we have not yet implemented.

C. Human Judgments

The corpus provided by Becker et al. consists of slightly over 2,000 sentences, each with a selected answer phrase and four human judgments of the quality of the answer. Human judges could rate answers as either "Good", "Okay", or "Bad". Because the classifier requires that each instance be classified in only one category, we had our program use the four judgments to calculate a score for each answer, which we then used to determine how to classify it in the data set. A "Good" rating added 0.25 to the score, an "Okay" added 0.125, and a "Bad" rating added nothing. This score is then compared to the threshold value (a pre-set constant in the program). If the score is greater than or equal to this value, the answer is classified in the data set as "Good". Otherwise, it is classified as "Bad".

VI. Results

We used the program to produce a data set from the Becker et al. corpus [5]. This data set was created using a threshold value of 1.0 (all four human judges have to rate the sentence as "Good"). Then, using Weka, a random sample of the sentences was drawn from this data to produce a subset with a comprable amount of "Good" and "Bad" sentences. This set contained a total of 582 instances, 278 of which were "Good" and 304 of which were "Bad". We tested both the Weka Logistic classifier [12] and the Weka Simple Logistic classifier on the data using 10-fold cross-validation.

The statistics we were most concerned with were the correct classification rate (the number of correctly classified instances divided by the total number of classified instances), the true positive rate (the number of correctly classified "Good" instances divided by the total number of "Good" instances), and the false positive rate (the number of incorrectly classified "Bad" instances divided by the total number of "Bad" instances). We also looked at the Weka-generated "confusion matrix," which summarizes the classifications.

For the Logistic classifier, the correct classification rate was 72.3%, the true positive rate was 78.4%, and the false positive rate was 33.2%. For the confusion matrix (which is normalized), we have:

	Classified "Good"	Classified "Bad"
"Good"	218	60
"Bad"	101	203

In total, 54.8% of the instances were labeled "Good" and 45.2% were labeled "Bad".

For the Simple Logistic classifier, the correct classification rate was 74.2%, the true positive rate was 81.3%, and the false positive rate was 32.2%. For the confusion matrix, we have:

	Classified "Good"	Classified "Bad"
"Good"	226	52
"Bad"	98	206

In total, 55.7% of the instances were labeled "Good" and 44.3% were labeled "Bad".

Becker et. al were able to get a true positive rate of 83% and a false positive rate of 19% at the equal error rate [5]. Although their false positive rate is lower, the true positive rate of our system is definitely comparable to theirs.

VII. Sentence Compression

To compress S into the different simplified statements, we used a modified version of the integer linear programming (ILP) model described by Filippova and Strube [8]. We first calculated probabilities of dependencies occurring after head words and used this as an estimate of the grammatical necessity of different dependencies given the presence of a head word. Along with all of the constraints placed on the ILP in the original model, we added an extra constraint that ensures the preservation of the answer phrase in the compression. We then used a linear program solver to solve the ILP for all length values between 0 and the length of S, generating a set of compressions of S with all possible lengths. From these compressions, we used a 3-gram model to calculate the Mean First Quartile (MFQ) grammaticality metric described by Clark et al. [13]. Compressions with an MFQ value lower

than a threshold were deemed grammatical, and the shortest of these was selected as the final compression of S for the given answer.

A. Dependency Probabilities

In order to be more precise, we used a larger set of Stanford dependencies to calculate the conditional probabilities than we did for the feature set in the selection part of the system. The extra dependencies included in this set were collapsed dependencies [11], which are created when closed-class words like *and*, *of*, or *by* are made part of the grammatical relation, producing dependencies like *conj_and*, *prep_of*, and *prep_by*.

To calculate the frequencies of dependencies after certain head words, we used a pre-parsed section of the Open American National Corpus [14]. Filippova and Strube [8] used part of the TIPSTER corpus to calculate their frequencies, but we lacked the computational resources to parse the data ourselves, so we used the pre-parsed data. The frequency of a dependency in our system is defined as the the number of words in the document that are governors of at least one of these dependencies. If a dependency appears more than once for a given governor word (e.g. if a noun is modified by two prepositional phrases), our program will only increase its count by one. This prevents the frequency of a dependency following a given head word from ever exceeding the frequency of the head word itself.

To prevent rounding errors, we used a smoothing function when calculating the probabilities from the frequency data. If we let $f_{(\ell|h)}$ be the frequency with which dependency of type ℓ occurs with head word h in the corpus, and let f_h be the frequency of word h in the corpus, then we define the smoothed probability $P_{(\ell|h)}$ to be

$$P_{(\ell|h)} = \log_2(\frac{f_{(\ell|h)}}{f_h} + 1)$$

Because $f_h \geq f_{(\ell|h)}$ and $f_h, f_{(\ell|h)} \geq 0$, $\frac{f_{(\ell|h)}}{f_h} \in [0, 1]$. Therefore, because

$$\log_2(x + 1) \in [0, 1] \forall x \in [0, 1]$$

we know that $P_{(\ell|h)} \in [0, 1]$ for all possible ℓ and h.

Finally, to avoid problems that come with probability values of zero, our system linearly maps the $P_{(\ell|h)}$ values from $[0, 1]$ to $[10^{-4}, 1]$.

B. Integer Linear Program

Like Filippova and Strube [8], we formulate the compression problem as an ILP. For each dependency in the parse tree (say, the dependency with the Stanford type ℓ, holding between head word h and dependent word w), we create a variable $x^{\ell}_{h,w}$. These variables must each take on a value of 0 or 1 in the solution, where dependencies whose variables are equal to 1 are preserved in the resulting compression and dependencies whose variables are equal to 0 are deleted, along with their dependent words. The ILP maximizes the objective function

$$f(X) = \sum_x x^{\ell}_{h,w} \cdot P_{(\ell|h)} \cdot t(\ell, P_{(\ell|h)})$$

where t is the *tweak function*, which corrects discrepancies between frequency and grammatical necessity that occur with some specific types of dependencies. For example, conjunctions (*conj*) occur very frequently in written English, but they are generally not necessary for the grammaticality of a sentence. Often, deleting parts of conjunctions can actually be an effective way to compress a sentence. Multiplying a particular probability by t linearly maps the range of that value from $[0, 1]$ to $[min_\ell, max_\ell]$. The tweak function is defined as

$$t(\ell, P_{(\ell,h)}) = max_\ell - min_\ell(1 + \frac{1}{P_{(\ell,h)}})$$

where

$$min_{conj} = 0.0, max_{conj} = 0.4$$

,

$$min_{det} = 0.4, max_{det} = 1.0$$

,

$$min_{poss} = 0.5, maxposs = 1.0$$

, and

$$min_\ell = 0.0, max_\ell = 1.0$$

for all other dependencies, which means that $t(\ell, P_{(\ell,h)}) = 1$ for all dependencies besides conjunctions, determiners and possessives. Our tweak function replaces the importance function used in Filippova and Strube's objective function [8].

Filippova and Strube also used two constraints in their model to preserve tree structure and connectedness in the compression:

$$\forall w \in W, \sum_{h,\ell} x^{\ell}_{h,w} \leq 1$$

$$\forall w \in W, \sum_{h,\ell} x^{\ell}_{h,w} - \frac{1}{|W|} \sum_{u,\ell} x^{\ell}_{w,u} \geq 0$$

and one to restrict the length of the final compression to α:

$$\sum_x x^{\ell}_{h,w} \leq \alpha$$

To ensure that all of the words in the pre-selected answer A are also preserved, we include in our model the extra constraint

$$\forall w \in A, \sum_{h,\ell} x^{\ell}_{h,w} \geq 1$$

We solved these integer linear programs using lp_solve [15], an open-source LP and ILP solver.

C. Shortest Grammatical Compression

In order to find the shortest grammatical compression of S, our system first finds a solution to the ILP for S and A for every value of α (the maximum length constraint parameter) between the length of S and the length of A. Because the constraints also specify that every word in A is preserved in the compression, any model where α is less than the length of A would have no solution.

Although all solutions to the ILP are connected dependency trees, some of the actual sentences created by linearizing these trees will not be grammatical. To determine the grammaticality

```
Sentence: Bill drives his car to the park every morning.
Answer: the park
Compressed Sentences:
Compressions:
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
drives his car to the park every morning .      S = 1.2192966633804272
Bill drives to the park every morning .          S = 1.1363540097281664
drives to the park every morning .      S = 1.2192966633804272
Bill drives to the park .       S = 1.06102567648667
drives to the park .    S = 1.133694190282954
Best Compression: Bill drives to the park .
```

Fig. 1. Simulation Results

```
Sentence: Bill drives his car to the park every morning.
Answer: every morning
Compressed Sentences:
Compressions:
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
drives his car to the park every morning .      S = 1.2192966633804272
Bill drives to the park every morning .          S = 1.1363540097281664
drives to the park every morning .      S = 1.2192966633804272
drives to park every morning .  S = 1.1429945444805045
Best Compression: Bill drives his car to the park every morning .
```

Fig. 2. Simulation Results

of the compressions, we use the MFQ metric, which is based on a 3-gram model created using the Berkeley Language Model Toolkit [16] and trained on the OANC text. This metric was shown to work well at distinguishing grammatically well-formed sentences from ungrammatical ones by Clarke et al. [13]. It considers the log-probabilities of all of the n-grams in the given sentence, selects the first quartile (25% with the lowest values), and calculates the mean of the ratios of each n-gram log-probability over the unigram log-probability of that n-gram's head word. The larger the MFQ value is, the less likely the sentence is to be grammatical.

Our system looks through the list of different length compressions and selects the shortest compression with an MFQ value less than a specified threshold (for our 3-gram model, we used a threshold of 1.14). This compression is returned as the simplified statement extracted from S for the answer A.

D. Results

We have not yet been able to conduct a test of the compression system, because testing the grammaticality of the generated compressions and their coverage of the set of possible simplified statements requires the use of a large number of human judges. However, the basic functionality of the compression system can at least be demonstrated with some sample outputs from the compressor. In each of the outputs, the sentence and answer are specified at the top, and then each row contains a potential compression and its MFQ value (labeled as 'S' on the readout).

Figure 1 shows a perfect compression of the sentence *Bill drives his car to the park every morning*. In the list of generated compressions, the one ultimately selected is clearly the shortest grammatical compression of the input sentence.

The output in Figure 2 is still grammatical, but there is one shorter compression in the list that is also grammatical, but was not identified by the program. This is because the MFQ value for *Bill drives to the park every morning* was 1.136, which is slightly less than the threshold of 1.14. Examples like this make it clear that tuning the gramamticality threshold is very important.

Figure 3 is not grammatical, but there is a grammatical sentence in the compression list only one word longer than the

```
Sentence: Bill drives his car to the park every morning.
Answer: Bill
Compressed Sentences:
Compressions:
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives his car to the park every morning .          S = 1.1363540097281664
Bill drives car to the park every morning .      S = 1.1363540097281664
Bill drives to the park every morning .          S = 1.1363540097281664
Bill drives car to the park .    S = 1.1363540097281664
Bill drives his car .    S = 1.0533242202705644
Bill drives car .        S = 1.1043456643485705
Bill drives .    S = 1.1702534936017774
Best Compression: Bill drives car .
```

Fig. 3. Simulation Results

compression that was selected. The chosen compression had a higher MFQ score than the true shortest grammatical sentence, but because it was shorter, it was chosen nonetheless.

VIII. Conclusion

The key principle around which our system is built is that selecting the answer at the beginning of the QG process and using them to guide SSE can improve the coverage of the system. We implemented the machine learning-based approach for answer selection used by Becker et al. [5] and developed a way to compress a sentence while leaving a specified answer phrase intact. Although we have not yet been able to perform large scale tests on this system where the output is rated by human judges, we have generated some good output sentences. Once our implementation is perfected and tuned, we will perform more powerful and complete tests.

This system will soon be integrated with Jacob Zerr's Part-of-Speech Pattern Matching system for direct declarative-to-interrogative transformation to produce a full, functional, QG system.

References

[1] Kunichika, Hidenobu, Tomoki Katayama, Tsukasa Hirashima, and Akira Takeuchi. "Automated question generation methods for intelligent English learning systems and its evaluation." In Proceedings of ICCE2004, pp. 2-5. 2003.

[2] Wolfe, John H. "Automatic question generation from text-an aid to independent study." In ACM SIGCUE Outlook, vol. 10, no. SI, pp. 104-112. ACM, 1976.

[3] Heilman, Michael, and Noah A. Smith. "Good question! statistical ranking for question generation." In Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 609-617. Association for Computational Linguistics, 2010.

[4] Heilman, Michael, and Noah A. Smith. "Extracting simplified statements for factual question generation." In Proceedings of QG2010: The Third Workshop on Ques-tion Generation, p. 11. 2010.

[5] Becker, Lee, Sumit Basu, and Lucy Vanderwende. "Mind the gap: learning to choose gaps for question generation." In Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 742-751. Association for Computational Linguistics, 2012.

[6] Knight, Kevin, and Daniel Marcu. "Statistics-based summarization-step one: Sentence compression." In AAAI/IAAI, pp. 703-710. 2000.

[7] Cohn, Trevor, and Mirella Lapata. "Sentence Compression as Tree Transduction." Journal of Artificial Intelligence Research 34 (2009): 637-674.

[8] Filippova, Katja, and Michael Strube. "Dependency tree based sentence compression." In Proceedings of the Fifth International Natural Language Generation Conference, pp. 25-32. Association for Computational Linguistics, 2008.

[9] Stanford NLP Toolkits, http://nlp.stanford.edu/software.

[10] Holmes, Geoffrey, Andrew Donkin, and Ian H. Witten. "Weka: A machine learning workbench." In Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on, pp. 357-361. IEEE, 1994.

[11] De Marneffe, Marie-Catherine, and Christopher D. Manning. "Stanford typed dependencies manual." URL http://nlp. stanford. edu/software/dependencies manual. pdf (2008).
[12] Le Cessie, Saskia, and J. C. Van Houwelingen. "Ridge estimators in logistic regression." Applied statistics (1992): 191-201.
[13] Clark, Alexander, Gianluca Giorgolo, and Shalom Lappin. "Statistical representation of grammaticality judgements: the limits of n-gram models." CMCL 2013 (2013): 28.
[14] Ide, Nancy, and Catherine Macleod. "The american national corpus: A standardized resource of american english." In Proceedings of Corpus Linguistics 2001, vol. 3. 2001.
[15] Berkelaar, Michel. "lpSolve: Interface to Lp solve v. 5.5 to solve linear/integer programs." R package version 5, no. 4 (2008).
[16] Pauls, Adam, and Dan Klein. "Faster and smaller n-gram language models." In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1, pp. 258-267. Association for Computational Linguistics, 2011.

Extreme Value Theory and Visual Recognition

Rachel Moore
Department of Computer Science
University of Colorado, Colorado Springs

***Abstract* – The fields of machine learning and psychology have begun to merge, particularly in the subject of vision and recognition. This paper proposes an experiment on human recognition and categorization, using arbitrary images as stimuli. The data will be fitted to an Extreme Value Theory based model, which we hope will give clearer incite into the ways humans categorize novel information.**

***Index Terms* – Recognition, Category Learning, Machine Learning, Extreme Value Theory, Cognitive Psychology.**

I. Introduction

Training set selection is one of the most crucial steps in machine learning. If one wishes for a machine to identify images of apples, only providing images of oranges during training is, in most cases, counter productive. Traditionally, training sets have been selected so that was a wide array of data, so that the training set would closely match a gaussian, or normal, distribution [1]. However, this can become expensive, particularly in tasks that require labeled data for supervised learning. An extensive amount of data is also needed, as smaller sets are less likely to have a normal distribution, which can cause high variance responses and inconclusive results [1].

There have been many advances in the area of training data selection, but there is still a need for methods that select the best training data from small datasets. Simple methods like bootstapping can be used to generate new data in cases of small data sets. Many of these methods do not work with certain learning models [1] [2]. To understand what makes an effective training set, it is important to study how training affects the ultimate categorization. One way of doing this is to study recognition and categorization in humans.

The ability to categorize is one of the most crucial skills we develop as children. Despite its importance, the way we organize information is still a mystery. There are many models of categorical learning in psychology, and more are in development. Studies, such as the one done by Hsu and Griffiths [3] (discussed in Section II), have given some insight into the category learning process, and have yielded interesting results. However, the Gaussian models currently being used on these types of experiments are not capturing the extremes in the data, or the participants' bias towards one category or another. From our research, Extreme Value based models in machine learning have been shown to be a better predictor of human response frequency than Gaussian models. In summary, there are 4 main contributions of this paper:

- Extreme Value Theory and its application.
- Empirical evaluations and metrics for this research.
- Experimental results
- The future of this work.

II. Background

In this section, we discuss Extreme Value Theory (EVT) and its applications, as well as studies involving categorization and EVT modeling. We will also explore psychological research involving categorical learning, which will be the basis for our experiments.

A. Extreme Value Theory

The extreme value theorem states that a function with a continuous and closed interval will have a minimum and maximum value [4] [5]. EVT has been implemented as a statistical model in many different fields of research. Hugueny, Clifton, and Tarassenko [6] used EVT as the basis to create a new model for intelligent patient monitors. The current monitors they reviewed set off false alarms constantly, to the point that hospital staff ignored them. The model they proposed would be less likely to do this, as the EVT-based model would be able to differentiate between truly non-extreme changes in vitals and clear abnormality. EVT has also been used in machine learning to normalize recognition scores [7], which may skew distributions due to outliers.

This research seeks to establish a new EVT-based model of visual recognition and categorization. Particularly, this model may be instrumental for tasks that wish to replicate human information processing. There are three types of extreme value distributions:

Type 1, Gumbel-type distribution:

$$PR[X \leq x] = exp[-e^{x-\mu/\sigma}]. \tag{1}$$

Type 2, Fréchet-type distribution:

$$PR[X \leq x] = \begin{cases} 0, & x < \mu, \\ exp\left\{-\frac{x-\mu}{\sigma}^{-\xi}\right\} & x \geq \mu. \end{cases} \tag{2}$$

Type 3, Weibull-type distribution:

$$PR[X \leq x] = \begin{cases} exp\left\{-\frac{x-\mu}{\sigma}^{\xi}\right\}, & x \leq \mu \\ 0 & x > \mu \end{cases} \tag{3}$$

where μ, $\sigma(> 0)$ and $\xi(>$s $0)$ are the parameters [5].

EVT-based models can be used as replacements for Binary and Gaussian models, as EVT-models are able to include multiple classes, and do not rely heavily on norms (see Fig. 1). This can also be helpful in the case of training set selection. For example, say there is a set images of apples that need to be categorized into 2 groups: green granny smith and red

delicious. While the first and last apple groups have green and red skin tones, respectively, with slight variations in color. However, in this set of apples are a few fuji apples, whose colors range from ruddy green to orange red, and might be categorized into either of the other apple groups. To make the best predictions on which category each apple belongs to, we can use the EVT to find the apples at the groups' decision boundaries, i.e. the most and least red red delicious apples, and the most and least green granny smith apples. From this we can create a training data set. When these clear decision boundaries are known, anything that lies outside of them, say a greenish red fuji apple, can be categorized as a true outlier or part of a third class in the data.

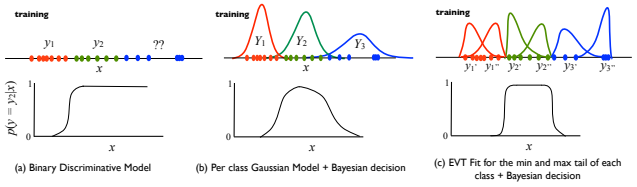

Fig. 1. Example of data selected with EVT. Courtesy of Boult

B. Prior Research in Human Visual Recognition

In a two part study, Cohen, Nosofsky, and Zaki [8] examined the effects of class variability on categorization. They hypothesized that the generalized context model (GCM), used to calculate the probability that an item will be categorized into one class or another, would substantially underestimate the degree to which participants would classify stimuli into the categories of high variance (we discuss GCM in greater detail in Section V). They found that the middle stimuli (items that were in between the low variance and high variance classes) were classified into the higher variance category, with the probability of up to .73. The GCM estimated the probability to be as low as .35, significantly below what was indicated by the data.

Hsu and Griffins [3] conducted a study in which the participants were taught two alien "languages", consisting of simple images of line segments. Class A had short, low variance line segments, which only differed slightly from one another. Class B had much longer, high variance line segments, in which each line's length was very different from the others. Participants were put into either a generative learning condition or a discriminative learning condition, which varied by the way the training images were presented. In the generative condition, two different cartoon aliens would appear on the screen to indicate which line belonged to which tribe's language. In the discriminative condition, one cartoon alien appeared as a single translator, indicating which language was language was on the screen. After training, participants were shown line segments that were between the lengths of the low and high variance classes and asked to categorize them.

As with Cohen, Nosofsky, and Zaki's [8] study, the results showed that the participants had a strong bias toward the high variance class (Class B), clustering the middle stimuli with the more diverse lines. They found that their Gaussian-based model did not fit their data accurately, and therefore wondered if the Gaussian assumption did not reflect this type of human recognition.

III. Empirical Evaluations

In this section, we discuss our experimental designs, as well as the metrics and technical approach of our study.

In a pilot study, Boult et. al [1] analyzed the data from Hsu and Griffins' [3] study using an EVT model. Because of the bias toward the high variance class, they believed that an EVT-based model would match human data in a more concise way (see Fig. 2) than Gaussian models.

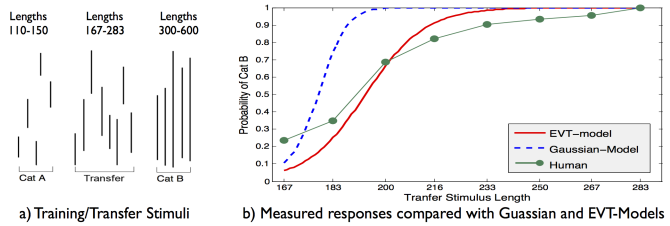

Fig. 2. Comparison of Gaussian and EVT-based models with human data. Courtesy of Boult.

For the second half of this pilot study, we have collected our own data. Our experiment expanded on Hsu and Griffins' [3] study, but used EVT-based models. We hope our model will paint a clearer and more accurate picture of the way humans categorize unfamiliar stimuli. Another possible extraneous factor in Hsu and Griffiths' [3] study is the way the alien interpreters (the categories) were presented. Those in their generative group were clearly shown when the category had changed, as the aliens changed depending on the sign. In the discriminative group, there was a single alien which never left the screen, and so the participants may not have noticed the sign change. We have duplicated some of their stimuli to test for this factor (see Fig. 3).

A. Metrics and Design

Our research uses models based on the extreme value distributions. Scheirer et al. [7] define extreme value distributions as "... limiting distributions that occur for the maximum (or minimum, depending on the data) of a large collection of random observations from an arbitrary distribution." In the case of visual recognition and categorization in humans, instead of removing the outliers or having them skew the results, one can normalize them, possibly allowing for a better fitted prediction.

For our experiment, we referred to the generalized extreme value (GEV) distribution, or the combined Gumbel, Frechet, and Weibull distributions. GEV is defined as

$$GEV(t) = \begin{cases} \frac{1}{\lambda} e^{-v^{-1/k}} v^{-(1/k+1)} & k \neq 0 \\ \frac{1}{\lambda} e^{-\left(x+e^{-x}\right)} & k = 0 \end{cases} \quad (4)$$

[1]Personal Communication

where x is equal to $\frac{t-\tau}{\lambda}$, v is equal to $(1+k\frac{t-\tau}{\lambda})$, and k,λ, and τ are the shape, scale, and location parameters.

For stimuli, we created a set of 2 dimensional Non-uniform rational B-spline (NURBS) shapes. NURBS are mathematically based shapes, and can be manipulated through functions and interpolation. In a NURBS parametric form, "... each of the coordinates of a point on a curve is represented separately as an explicit function of an independent parameter" [9]

$$C(u)=(x(u),y(u)) \qquad a\leq u\leq b \tag{5}$$

Where "$C(u)$ is a vector-valued function of the independent variable u", which is within the interval [a,b] (usually normalized to [0,1]) [9]. The NURBS we created look similar to ink blots. Each group of images had points that were interpolated to create a set with two clear classes, and another that was somewhere between those two classes (see Fig. 3 and 4). Four groups of shapes were used, and each group contained 17 images. These stimuli acted as distractor tasks. The other

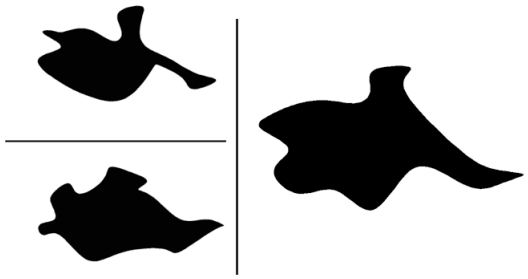

Fig. 3. Example of images duplicated from our NURBs stimuli.

stimuli were white lines of varying lengths, place inside of a black circle. Each set had a total of 18 images. These were based on the stimuli in Hsu and Griffiths' [3] study (see Fig. 5). These stimuli were placed into 3 conditions: generative, discriminative, and enhanced tails.

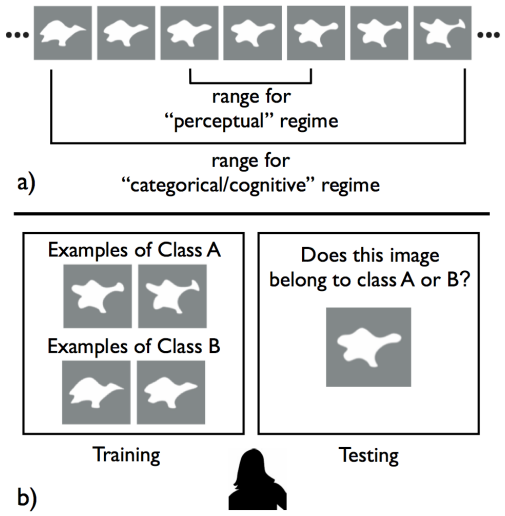

Fig. 4. Example of images from training classes A and B, and a testing image. Courtesy of Boult.

For the experiment itself, we used a program called PsychoPy, version 1.80 [2]. PsychoPy is open source psychophysics software, developed by Piece [10].

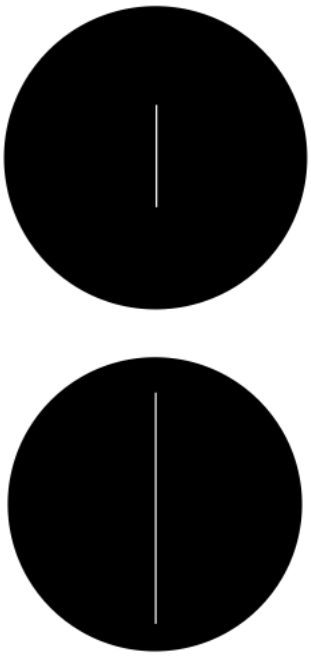

Fig. 5. Example of images duplicated from Hsu and Griffiths' [3].

There were 8 participants total, some of whom took the experiment on multiple occasions. From them, we gathered 30 trials for each of the 3 conditons. Our participants were asked to categorize a series of images into one of two groups, Group 0 or Group 1, and told that there was to be a training component where they would be shown the images and their respective categories, and a testing component where they would label the images themselves. The groups had a separate training set or training style, and testing set. For every group, the participants were trained on the 10 shapes at extrema, 5 from each tail. They were then tested on those same shapes, along with the shapes from the middle of the set, some of which were repeated to make up a total of 20 shapes per training. All trials were repeated, for a total of 20 trial blocks.

IV. Data Analysis

We recorded which middle stimuli were categorized into Group 0 or 1, and the frequency to which these stimuli were placed in these groups (see Fig. 9). The EVT-based model was fitted to the data. It reflected the biases the participants have in categorization, as it did in the pilot study.

A. Factor 1: The Generative Condition

In the generative condition, participants were shown the training stimuli. The training set consisted of lines that were in high variance and low variance categories. The low variance lines were 110, 120, 130, 140, and 150 pixels in length, while

[2]Accessed here: http://www.psychopy.org

the lines in the high variance category were 300, 375, 450, 525, and 600 pixels in length. During training, a box appeared 0.5 sec before before the stimuli, indicating which group the image belonged to. After the stimulus appeared, both the box and the image remained on the screen for 1.5 sec. This was repeated for all 10 stimuli in the training set.

The testing set was comprised of the training set, as well as a set of "middle" stimuli with line lengths of 167, 183, 200, 216, 233, 250, 267, and 283 pixels. The probability that each of these lines would be categorized into the high variance category was 0.17, 0.20, 0.33, 0.4, 0.53, 0.70, 0.77, 0.90, respectfully. The data fit our model well, with the main deviation being at line length 267 (see Fig. 6). Out of the three conditions, this condition was the closest fit to our EVT-based model.

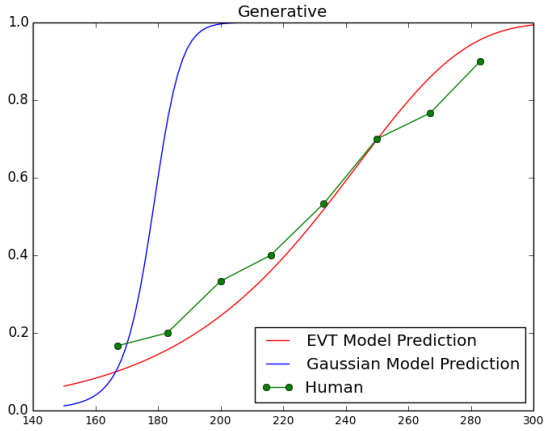

Fig. 6. Comparison of our EVT model with human data for the generative condition.

B. Factor 2: The Discriminative Condition

In the discriminative condition, participants were shown the same training and testing stimuli as the generative condition. However, the indicator box remained on the screen throughout the training session, with only the text changing. Each image still remained on the screen for 1.5 sec. For this condition, the probability that each of the middle stimuli would be categorized into the high variance category was 0.13,0.23, 0.30, 0.37, 0.63, 0.67, 0.83, and 0.90, respectively. The data for this condition also fit our model well, with the main deviation being at line length 233 (see Fig. 7).

C. Factor 3: The Enhanced Tails Condition

The final training set of lines contained the set of low variance lines as the generative and discriminative conditions, but the high variance lines had an elongated tail, with pixel lengths of 300, 375, 450, 600, and 800. The training set up was identical to that of the generative condition. For this condition, the probability that each of the middle stimuli would be categorized into the high variance category was 0.00,0.10, 0.20, 0.20, 0.43, 0.53, 0.60, 0.77, respectively. This shows

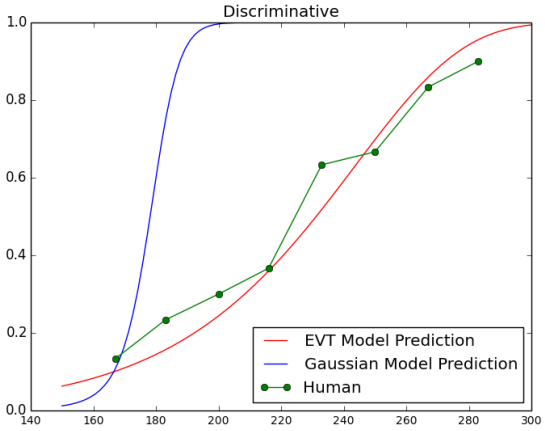

Fig. 7. Comparison of our EVT model with human data for the discriminative condition.

a shift towards the low variance category. We trained the model on the same set of training data used for the previous conditions for a better visual representation of the bias towards the low variance category (see Fig. 8).

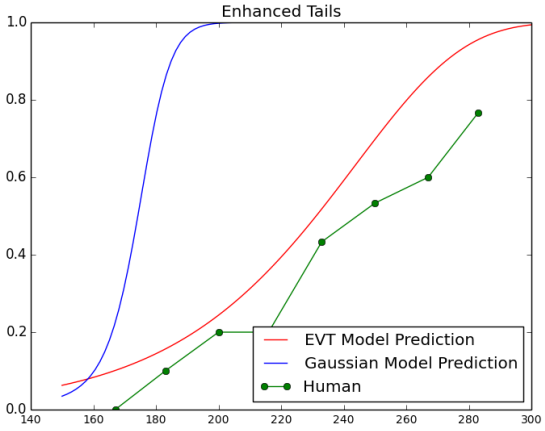

Fig. 8. Comparison of our EVT model with human data for the enhanced tails condition.

D. Comparison

In this section, we will compare the generative, discriminative, and the enhanced tails conditions, and discuss the statistical analysis for the experiment. Fig. 9 is a summary of the probabilities of each condition. The error bars indicate the variance of each line lengths probability. Both the generative and discriminative categories had similar trends. The variance of the generative, discriminative, and enhanced tails conditions were 0.073, 0.083, and 0.072, respectively.

V. Future Work

For our future research, we will incorporate measurements from the NURBS shapes. Because these shapes are mathematically based, the stimuli's dimensions can be easily applied to

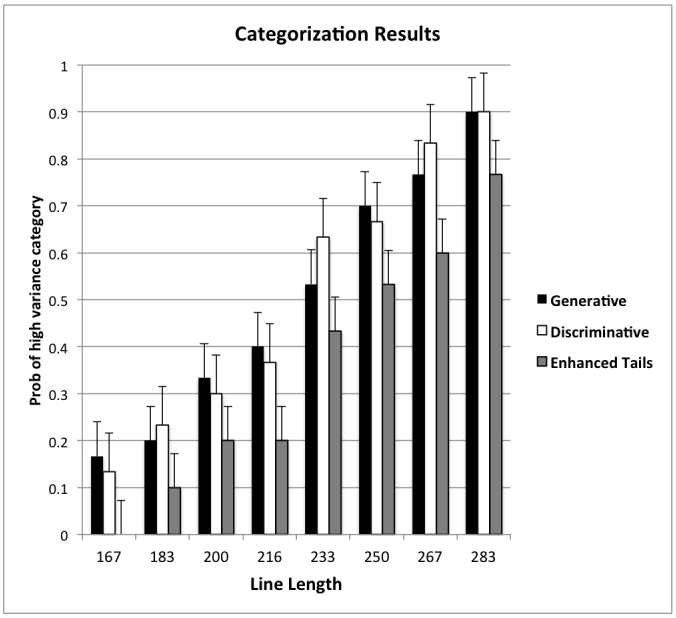

Fig. 9. Probability of categorization of middle stimuli into the high variance category for the generative, discriminative, and enhanced tails conditions.

probability models. One such model, which was mentioned in Section II, is the generalized context model (GCM), which states that "For the case of two categories A and B, the probability that a given stimulus X is classified in category A is given by

$$P(A|X) = \frac{\beta_a \eta_{XA}^{\alpha}}{\beta_A \eta_{XA}^{\alpha} + (1-\beta_A)\eta_{XB}^{\alpha}} \tag{6}$$

where β_A is a response bias toward category A and η_{XA} and η_{XB} are similarity measures of stimulus X toward all stored exemplars of categories A and B, respectively" [11].

Because our stimuli are so diverse, we plan to make at least one other variation on the current experiment. This may involve changing the task difficulty, the time length, or varying the amount of stimuli in the training sessions.

VI. Conclusion

This paper proposed a new EVT based model for visual recognition. For our purposes, we hope our model will prove to be consistent and accurate in predicting human recognition and categorization. If it is shown to be both of these things, the model could be used to select training sets for machine learning more efficiently, as EVT-based models focus on training data at the extremes, which may cut down on costs of supervised learning. We have seen that EVT-based models can be applied to both generative and discriminative learning situations. We believe that EVT-based models should also be insensitive to the difference between categorical and perceptual learning. With more research, our model may be applied to other human learning tasks, not just visual recognition.

Acknowledgement

I would like to acknowledge Dr. Walter Scheirer, Dr. David Cox, and the team of scientists at Harvard University, who have greatly contributed to this project. I would also like to thank Dr. Terrance Boult, Dr. Lori James, Dr. Kristen Walcott-Justice, and Dr. Jugal Kalita for their invaluable guidance. This project is being supported by NSF REU Grant 1359275.

References

[1] E. Alpaydin, *Introduction to machine learning*. MIT press, 2004.
[2] I. H. Witten, E. Frank, and A. Mark, "Hall (2011)." data mining: Practical machine learning tools and techniques," 2011.
[3] A. S. Hsu, T. L. Griffiths *et al.*, "Effects of generative and discriminative learning on use of category variability," in *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, 2010, pp. 242–247.
[4] M. K. Nasution, "The ontology of knowledge based optimization," *arXiv preprint arXiv:1207.5130*, 2012.
[5] S. Kotz and S. Nadarajah, *Extreme value distributions: Theory and applications*. World Scientific, 2000, vol. 31.
[6] S. Hugueny, D. A. Clifton, and L. Tarassenko, "Probabilistic patient monitoring with multivariate, multimodal extreme value theory," in *Biomedical Engineering Systems and Technologies*. Springer, 2011, pp. 199–211.
[7] W. Scheirer, A. Rocha, R. Micheals, and T. Boult, "Robust fusion: extreme value theory for recognition score normalization," in *Computer Vision–ECCV 2010*. Springer, 2010, pp. 481–495.
[8] A. L. Cohen, R. M. Nosofsky, and S. R. Zaki, "Category variability, exemplar similarity, and perceptual classification," *Memory & Cognition*, vol. 29, no. 8, pp. 1165–1175, 2001.
[9] L. Piegl and W. Tiller, "The nurbs book," *Monographs in Visual Communication*, 1997.
[10] "Psychopy—psychophysics software in python," *Journal of Neuroscience Methods*, vol. 162, no. 1–2, pp. 8 – 13, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165027006005772
[11] T. Smits, G. Storms, Y. Rosseel, and P. De Boeck, "Fruits and vegetables categorized: An application of the generalized context model," *Psychonomic Bulletin & Review*, vol. 9, no. 4, pp. 836–844, 2002.

Using Hidden Markov Models and Spark to Mine ECG Data

Jamie O'Brien
Saint Mary's College of Maryland
St. Mary's City, Maryland
Email: jcobrien@smcm.edu

***Abstract*—New potential risk factors for cardioembolic strokes are being considered in the medical community. The presence of these factors can be determined by reading an electrocradiogram (ECG). Manual ECG analysis can take hours. We propose combining accurate Hidden Markov Model (HMM) techniques with Apache Spark to improve the speed of ECG analysis. The potential exists for developing a fast classifer for these risk factors.**

I. Introduction

The proliferation of medical data in modern hospitals provides a rich environment for data mining. Electrocardiograms (ECGs) provide a wealth of information that can be used to diagnose cardiovascular diseases (CVDs). In Agarwal and Soliman [1], it is suggested that the ECG can be used to detect cardioembolic stroke risk factors. Aside from those factors included in the Framingham Risk Score, emerging factors include:

1) cardiac electrical/structural remodeling,
2) higher automaticity,
3) heart rate & heart rate variability.

Currently, the manual analysis of ECG patterns is time-consuming. It can take several hours to complete Acharya et al [2].

II. Problem Statement

We want to find a better method of detecting the emerging risk factors listed in Section I. We want to combine an effective Hidden Markov Model (HMM) classifier for ECGs with the fast, distributed processing power of Apache Spark.

A. Atrial Fibrillation—A Verified Stroke Risk

In atrial fibrillation (AF), the heart's atrial walls do not produce an organized contraction—instead, they quiver [3]. Even though AF is a component of the Framininghamm Stroke Risk Score [4], it is often undetected; the condition has evaded detection even in patients known to have paroxysmal atrial fibrilation. The detection rates may vary depending on the algorithms used, but seem to improve with longer monitoring times [5]. The difficulty of accurately detecting AF motivates the search for additional stroke risk factors.

B. Hidden Markov Models

Hidden Markov Models (HMMs) have been used with great effect in classifying ECGs. Andereão et al were able to demonstrate an accuracy of 99.97% in detecting the QRS complex of the heartbeat [6]. Their approach was to create a general model of the heartbeat, and then tune the model to each individual by using data from the first 20 seconds of their ECG. The general model of the heartbeat was composed of discrete states representing the P, Q, R, S, and T waves, the PQ and ST intervals, and the isoline. Andreão et al's work was able to detect premature ventricular contractions (PVCs). We hope to use a similar model for detecting ectopic beats and bundle blocks.

C. Apache Hadoop

Hadoop pairs a high-bandwidth distrubted file system with MapReduce programming Svachko et al [7]. This allows for a task to be broken up across many computers, the components calculated independently, and the results collected. In this way, Hadoop may improve the performance of signal processing tasks. This performance improvement is the core of the Cloudwave system described in Jayapandian et al [8]. The authors of that work used Hadoop to process multimodal bioinformatic data. A stand-alone machine was able to process 10 signals in 22-36 minutes. Their Hadoop cluster was able to process the same data in 4-6 minutes.

D. Apache Spark as a replacement for Hadoop MapReduce

While the Cloudwave system described in Jayapandian et al [8] is impressive, the highly iterative nature of data mining tasks may cause significant overhead under Hadoop's MapReduce architecture. Apache Spark avoids this issue by using the concept of resilient distributed datasets (RDDs). These RDDs can be cached in memory. This make the data available for iterative and parallel programming alike without having to be constantly reloaded Zaharia et al [9].

III. Method

The in-progress research explores the applicability of Hidden Markov Models on ECG readings, with the goal of detecting the emerging factors mentioned in [1]. Here we note the strategy for constructing our system.

We obtained ECG signals from the QT Database (QTDB), using the WaveForm Database application suite. We also

obtained two sets of annotations: one, marked `atr`, contains annotations that marks beats as normal, or as having some abnormality (preventricular contraction, for instance); the second set of annotations, marked `pu0`, contains waveform markers, such as p, t, and N (for normal qrs complex). Any records from the QTDB that did not contain annotations from `atr` were excluded, as we would not be able to verify our results against them.

We transformed the `pu0` annotations to provide clearer information. The standard for annotating waves is to open a wave with a paren, note the wave, and then close it with a paren. For instance, the p wave would be marked by the annotations (, p,). We wrote a script to process these annotations, and change them to the form pBegin, p, pEnd, so that all parenthesis were removed. This meant that the annotations themselves could now become a set of states for use in a Hidden Markov Model. The states derived from the annotations were: pBegin, p, pEnd, q, r, q, tBegin, t, tEnd, unknownBegin, and unknownEnd.

However, we found that it was not practical to simply map the states annotated in `pu0` to the beat classifications annotated in `atr`. When attempting to map the state sequence to PVC, for instance, no significant correlation could be found in a sample of PVC beats. We hypothesized that the duration of the states was also significant. It may be necessary to mark states as being faster or slower than normal. The duration between, for instance, pBegin and pEnd could tell us if the p wave were of normal duration.

With this in mind, we are determining a way to map the ECG signal itself to states. In [10], we find an algorithm for decomposing ECG signals into line segments. This algorithm moves a dynamically-sized window along the ECG signal. The window checks the distance between the endpoints and every point in-between, using normalized distances where needed. We can adjust the allowed error to accomodate noisy signals.

We modify this algorithm to output a list of 4-tuples of the form (starting point, length, mean of segment, standard deviation of segment). This converts the continuous ECG signal into a set of data points. We must then convert this set of data points into states that correspond with the waveforms of the heart beat: the p wave, qrs complex, t wave, and the intervals between them.

IV. The Classification Process

We begin by slicing an ECG signal between its R-R intervals. We then take a slice and segment it using the algorithm described in [10]. These segments are then labeled by the state they most match, using a decision tree. The progression of states is treated as an observation, and fed into the HMM to determine which beat type most accurately matches the observation.

V. Further Work

This work will not be complete until the HMM itself is built and can be tested. In anticipation of this, we have separated the QTDB into a training set comprising approximately 80% of the annotated data, and a testing set with the remaining approximately 20%. The training set is composed of five sub-groups, each approximately 20% of the size of the training set. We intend to use these sub-groups for cross-validation.

After the model is built and its performance is evaluated, we can begin the construction of the Apache Spark implementation of the model. The purpose of this will be to compare the performance of the Spark implementation against the locally-run implementation. The parameters for this experiment will be determined when the HMM itself is complete.

VI. Conclusion

This research may provide a effective method for detecting the emerging risk factors for a cardioembolic stroke mentioned in section I. This would assist researchers who are investigating these risk factors.

Acknowledgment

We would like to thank the National Science Foundation (NSF) for their generous grant, and the University of Colorado, Colorado Springs for hosting the Research Experience for Undergrads (REU) program.

References

[1] S. Argwal and E. Soliman, "Ecg abnormalities and stroke incidence," 2013. [Online]. Available: http://www.medscape.com/viewarticle/808752

[2] R. Acharya, A. Kumar, P. Bhat, C. Lim, N. Kannathal, and S. Krishnan, "Classification of cardiac abnormalities using heart rate signals," *Medical and Biological Engineering and Computing*, vol. 42, no. 3, pp. 288–293, 2004.

[3] F. H. Martini, J. L. Nath, and E. F. Bartholomew, *Fundamentals of Anatomy and Physiology (9th Edition)*. Benjamin Cummings, 1 2011.

[4] F. H. Study, "Stroke," https://www.framinghamheartstudy.org/risk-functions/stroke/stroke.php, (Visited on 07/14/2014).

[5] M. A. Rosenberg, M. Samuel, A. Thosani, and P. J. Zimetbaum, "Use of a noninvasive continuous monitoring device in the management of atrial fibrillation: a pilot study," *Pacing and Clinical Electrophysiology*, vol. 36, no. 3, pp. 328–333, 2013.

[6] R. V. Andreão, B. Dorizzi, and J. Boudy, "Ecg signal analysis through hidden markov models," *Biomedical Engineering, IEEE Transactions on*, vol. 53, no. 8, pp. 1541–1549, 2006.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[8] C. P. Jayapandian, C.-H. Chen, A. Bozorgi, S. D. Lhatoo, G.-Q. Zhang, and S. S. Sahoo, "Cloudwave: Distributed processing of big data from electrophysiological recordings for epilepsy clinical research using hadoop," in *AMIA Annual Symposium Proceedings*, vol. 2013. American Medical Informatics Association, 2013, p. 691.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[10] A. Koski, "Modelling ecg signals with hidden markov models," *Artificial intelligence in medicine*, vol. 8, no. 5, pp. 453–471, 1996.

Question Generation using Part of Speech Information

Jacob Zerr, Texas A&M University

***Abstract*—When testing students on knowledge from a story or article, a human must interpret the text to generate English questions. The difficulty in automating this process is producing a computational algorithm that can fully account for the syntactic and semantic complexities of human languages. Most approaches use big, costly semantic tools such as WordNets to achieve their semantic accuracy and rule-based approaches to achieve their syntactic accuracy. We propose an approach for generating knowledge-testing questions from textual English using machine learning to use part of speech pattern matching without using any large semantic tools.**

I. Introduction

Many attempts have been made to automate interpreting natural human languages, most of which have taken some small sub-problem and attempted to solve it. One such sub-problem is manipulating sentences to create question-answer pairs from a sentence, which we will be addressing. The main difficulty of question generation is that the method must maintain both semantic and syntactic accuracy. When formatting a question, we will need to change the structure of the sentence, add and remove words, change the tense or part of speech of words, or other complex operations. Moreover, through these operations we must keep the semantic integrity of the statement and select the correct answer to the resultant question. However, the applications of a proficient question generator could span domains from automated education tools to better AI conversation generation. We propose a new method of question generation that uses part of speech (POS) pattern matching based off of Inversion Transduction Grammars (ITG) from a sentence and question-answer pair corpus. We restrict our input to sentences containing one independent clause with the thought that this approach would work on any input if compressed first.

II. Problem Definition

Our input will be any collection of English sentences containing one independent clause. The sentences should be well formed and in correct English grammar for best results. The output will be a set of question-answer pairs that should be asking about the contextual knowledge of the original text. The output questions should also be grammatically correct. Here are a couple examples.

Jacob Zerr is participating in a National Science Foundation REU at the University of Colorado at Colorado Springs, Colorado Springs, CO 80918.
email: zerr2@tamu.edu

- John drove the car to work. → Who drove the car to work? John
- The pump is now operational. → Is the pump operational? Yes
- He waters the garden every day. → What does he do every day? waters the garden

III. Related Work

Question generation was brought to the attention of the natural language processing community by Wolfe [4] in 1976. He outlined the purpose and applications of a question generator and the potential challenges. Since then, many have produced question generators of a limited focus. Papasalouros [1] creates only multiple choice questions by producing a set of similar sentences where a key word has been replaced in the wrong selections. This reduces the complexity of the problem by avoiding interrogative sentence structure. Brown [3] focuses only on questions that test vocabulary and uses a WordNet to increase their question complexity without losing semantic accuracy. They also use part of speech (POS) tagging to maintain the syntactic accuracy of the question. Kunichika [2] provides the most general approach of all by dissecting both the syntactic and semantic structure of the original sentence before producing the question. After looking at both of these, their algorithm has a broad spectrum of questions it can generate about the original declarative sentence. However, this approach relies heavily on the accuracy of the interpretation of the sentence using tools like WordNets that may not be accurate in all cases. These are three representations of the current best solutions, none of which use machine learning. Our approach will rely heavily on a POS tagger for which we will be using the Stanford Parser outlined in Toutanove [9]. On a different note, Heilman [5] ranks generated questions which may be considered as a useful addition to our question generation process later on in our development.

IV. Inversion Transduction Grammars

Inversion Transduction Grammars are grammars that map two languages simultaneously and generally follow the format of a context free grammar. The main difference is the angle brackets in the grammar denote that the symbols should be read in left-to-right order for the first language and right-to-left for the second. This allows for the grammar to successfully map two languages with different part of speech orderings like SOV, SVO, or VSO languages. From there the lexicon has word pairs, one from each of the two languages, that should be direct translations of each other. This method uses basic

word-to-word translations and the fact that most languages use similar part of speech models, just in a different order, to achieve an accurate machine translation. Wu [6] explains these grammars in detail and shows how they can be used as an accurate form of machine translation. Both Goto [7] and Neubig [8] use these techniques to successfully perform machine translations between complex languages.

V. Our Approach

A. Producing POS Pattern Templates from the Corpus

The main approach that we will be pursuing to convert our declarative sentences to questions is through a POS pattern matching approach based off of ITGs. Though ITGs have mainly been used to convert a parsed sentence into another language, we will be using it to convert between declarative English and interrogative English. The difficulty in this process is that ITGs rely on the structure of the two sentences to be similar in all but ordering. However, there are structural parts of interrogative English that are not in declarative English and vice verse. Our approach will avoid this by ignoring the tree structure of the grammar and just map the movement of different phrases from the sentence to the question.

The first major step of processing our corpus instances is to identify the phrases that stay consistent in the transition from declarative sentence to question-answer pair. We do this by searching the instance for phrases of the exact same wording starting from the largest possible phrases and then incrementally decreasing the size until all of the common phrases have been identified. We call this process chunking. Figure 1 shows such an instance and the phrases that have been identified after chunking has been completed.

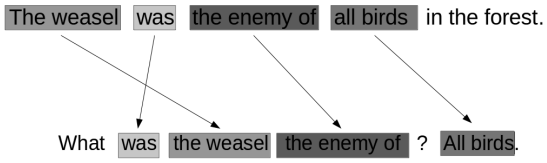

Figure 1. Sample of chunking the common phrases from an instance in our corpus.

Notice that there may be phrases in the sentence, question, or answer that are not in any of the other parts of the instance; in this case *in the forest* and *What*. These are kept and used by the algorithm in the process of finalizing the template; this will be explained later.

Our algorithm also can identify phrases that appear in all three parts of the instance as a part of the chunking process. This helps create templates for questions that quiz on adjectives of the sentence while still maintaining accuracy. Figure 2 is an example of such an instance where *plate* is repeated in all three parts of the instance to ensure that the answer makes sense.

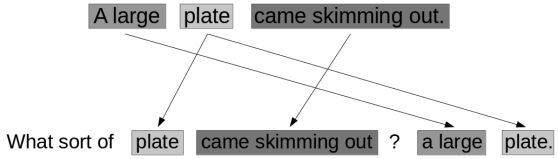

Figure 2. Sample of chunking an instance where a phrase is repeated in all three parts of an instance. This helps produce templates of questions that quiz on adjectives in the input sentence while still keeping accuracy.

Now that we have chunked our instances, we need to determine the part of speech of each of the phrases included in the sentence portion of the instance. For this we will be using the Stanford POS tagger [9]. There are two main approaches to attempting to tag these phrases with a POS: parsing it within the original context of the sentence or parsing it out of context. When parsing it out of context, we can conveniently get a single POS for the phrase. However, you forfeit accuracy with this method because the Stanford Parser solves ambiguities internally and it may return the wrong POS in an ambiguous case. For this reason, we chose to parse the phrase within the context of the original sentence. However, this is slightly more difficult, because we will now have to search the grammar parse tree of the whole sentence produced by the Stanford Parser. Our method for this was to search for the node of the tree that was the deepest ancestor of all of the words in the phrase. An example of this is shown in Figure 3. Here we can see that we identify the POS for *the enemy of* as a Noun Phrase in the context of the sentence *The weasel was the enemy of all birds in the forest*.

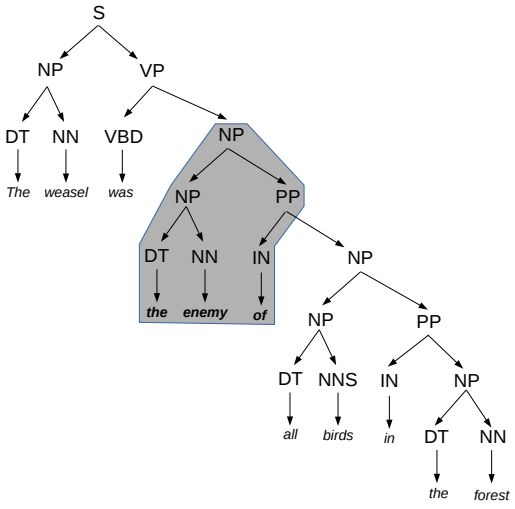

Figure 3. Our method for finding the POS of a phrase involves finding the deepest common ancestor of the words of the phrase. Here we can see *the enemy of* is being labeled as a Noun Phrase.

With this method of POS tagging, we then will label every phrase in the original sentence. This includes any phrases that

were not repeated in the question or the answer. The result of this process from the example used in Figure 1 is shown in Figure 4.

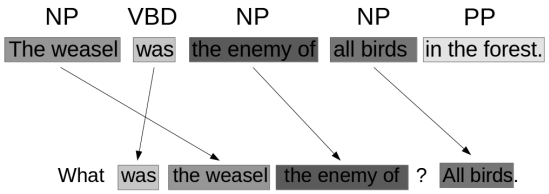

Figure 4. The example from Figure 1 with its sentence chunks labeled with their POS.

After the sentence chunks have been labeled, we drop all of the phrases that appeared in the sentence part of the instance. The phrases that only appeared in the question or answer are left as a part of the template. This is the final step of producing our POS template from an instance in our corpus. This process is completed for every instance in the corpus before we start trying to use these templates on our input sentences. Figure 5 shows this last step on our example.

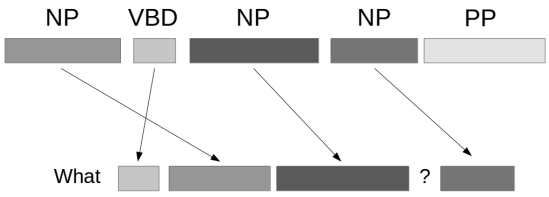

Figure 5. The final step of preparing the templates is drop all of the phrases that appeared in the sentence. Phrases that were just in the question or answer remain.

B. Generating Questions

Once we have converted the instances of our corpus into POS pattern matching templates, we can begin to try to fit input sentences into our templates. We do this by simply seeing if the input sentence can be divided into phrases that, when tagged with a POS, match the template. If we do find a match, we reorder the phrases by using the template's question-answer ordering to produce our question-answer pair. An example of a sentence fitting the template we produced above is in Figure 6.

An interesting question that arose from this pattern matching method is which type of POS tagging we would use for this part of the algorithm, in-context or out-of-context. Initially, it seemed clear that we should follow the same method we did in producing our corpus and use in-context. However,

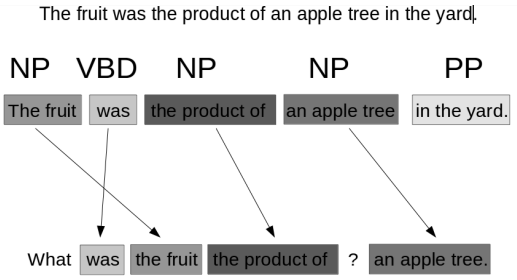

Figure 6. A sentence being matched to our example template and the question-answer pair it produced.

when experimenting with out-of-context we sometimes would produce a wrongful tag to a phrase that would fit a template. The expectation was that from an erroneous matching we would produce an inaccurate question, but this was not always the case. Figure 7 shows an input sentence matching to the template we produced above with *render* erroneously being labeled a Noun Phrase, however the produced question is accurate. We explored this question and our answer is discussed later in the results section.

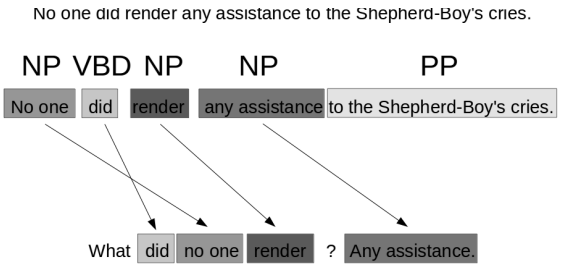

Figure 7. A sentence being wrongfully tagged and matched to our example template may still produce an accurate question-answer pair.

As a last note, if our algorithm can divide an input sentence to match a template more than one way, then it will produce a different question for each different legitimate divisions. An example of this is shown below in Figure 8.

An interesting observation on our method is that because we are simply reordering phrases, we keep the same vernacular of the original sentence. We have been operating in the domain of children's stories for this project and often times children stories will have odd wording that is not common vernacular anymore. These odd phrases will always be reflected in our output. The example in Figure 7 uses phrases like *render assistance* whereas most people would simply say *help*. This can be both a good and bad attribute of our approach. The good part is that our questions may contain slang or improper words of spoken English that takes our questions to a semantic level not normally achievable by a computer. However, it also can sometimes cause problems if these words are wrongfully

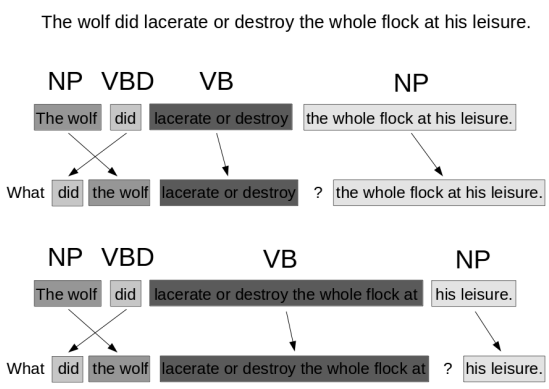

Figure 8. If a sentence can be divided in more than one way to match a template then the algorithm will produce a different question for each way.

tagged and will miss the factual tone of SAT-style questions that a user may want because of odd diction in the original sentence.

Also, it is important to note that we restricted our input to sentences containing one independent clause. This is necessary, because with additional clauses the accuracy of our POS tagging for phrases goes drastically down. For instance, if we divide our input in a way that a phrase is an entire clause it may be be given a POS label of Sentence, which is too general for our templates to produce good results. Typically, the larger the phrases become, the higher up the parse tree you will have to go for a deepest common ancestor, and the less specificity of POS tags we will have. Thus, in order to maintain reasonable levels of accuracy, we must limit our input to one clause sentences.

VI. Datasets

A large part of our work was producing and manipulating the corpus that defined our POS matching templates. This corpus is a collection of instances that map a sentence to a question-answer pair. Initially our corpus had 254 instances. From initial testing using this corpus we observed several things; a small corpus could produce an ample number of questions even with just one sentence inputted, often the same questions were produced more than once, and some instances in our corpus were better at producing accurate sentences than others. From these observations we decided to stop expanding the corpus and to actually start eliminating some instances.

Firstly, we had learned that some of the instances in our corpus were producing the same templates. Thus we went through and found the instances producing the duplicate templates and deleted them. An example of this was the template below had been produced 34 times. After 93 deleting instances that were producing duplicate templates our corpus had been reduced down to 161 instances.

$$\textbf{NP} \quad \textbf{VP} \quad \rightarrow \quad Who \quad \textbf{VP} \quad ? \quad \textbf{NP}$$

Secondly, we observed that some templates produced by our instances were much better at producing successful question-answer pairs than others. To test this theory we ran our corpus against some preliminary testing examples and confirmed this. Some templates were producing many consistently accurate questions, some produced very few questions, and others produced many inaccurate questions. Based on these results we eliminated any instance from our corpus that was producing questions at a twenty percent accuracy level or worse. This reduced our corpus down to just 129 instances.

Contrary to most forms of corpus-based machine learning, we found this corpus to be more than enough to produce a high number of different questions and a wide breadth of different types of questions. This is one of the largest advantages of our approach; it takes a comparatively tiny amount of data to get good results especially when compared to most of the other approaches that use large WordNets or other large semantic tools.

VII. Results

We analyzed and made improvements based off the syntactic and semantic accuracy of the output of our approach. The proportion of output instances that are grammatically correct, accurately quizzes the reader on the original knowledge, and has the corresponding answer will be our main metric of success. We used unbiased volunteer evaluators that judged each produced question-answer pair on whether they were syntactic and semantic accurate or not. Our evaluators are native English speakers that are in the process of attaining a Bachelors Degree, thus they have a firm knowledge of the English language. Our input were single independent clause sentences from children's stories such as *The Princess and the Pea*, *The Boy Who Cried Wolf*, and other such children stories.

A. POS Tagging Methods

We would first like to address the question we presented earlier on whether POS tagging on our input sentences should be done in-context or out-of-context. We would first like to note that we used only in-context tagging for creating our templates so that we could create accurately tagged templates. However, as we noted before, we produced accurate questions using both methods when tagging the input sentences. Based off of this we decided to experiment using four different methods for determining the POS to tag the sentence phrases: using the in-context tag (IC), using the out-of-context tag (OC), using a POS tag only if the two methods agreed ($IC \ \&\& \ OC$), and using either method to try to fit a sentence into a template ($IC \ || \ OC$). We tried these four methods on a 20 sentence input children's story.

Based off of the above results we chose to use the $IC \ || \ OC$ method for the rest of our work because of the greatly increased total solution production despite a very similar accuracy rate. Based off of the numbers above, this method was producing, on average, 7.25 accurate questions per inputted sentence.

Table I
POS TAGGING METHODS

Method	Accurate	Total	Percent
IC	94	160	58.75
OC	87	150	58.00
IC && OC	58	90	60.00
IC || OC	145	243	59.67

B. Overall Accuracy

For our final accuracy test, we used an input 48 sentences long from children's stories. We produced an output of 435 question-answer pairs. This means that we averaged 9.06 question-answer pairs per inputted sentence. This puts into perspective that our corpus, at 129 instances, really can perform like a large semantic tool despite its small size. The produced question-answer pairs were assessed by 4 evaluators that ranged the accuracy from 57.01% to 59.67% with an average of 58.36%. This result is also encouraging considering the previous work in this area. Brown [3] produced an accuracy rate from 52.86% to 64.52% and Papasalourous [1] produced an accuracy of 75% from his best strategy, but averaged an accuracy of 47.55% between all of their strategies. It is also interesting to note that both of these approaches were slightly more restricted in domain than our approach and they both relied on advanced wordnets in order to maintain semantic accuracy.

VIII. POSSIBLE FUTURE WORK

A possible extension of this work would be automatically analyzing the questions produced and ranking them in some way. Depending on the accuracy of the rankings we may be able to achieve a higher accuracy of the questions that are ranked in some top fraction of the produced questions.

IX. CONCLUSION

By using a relatively simple machine learning method with a small dataset, we were able to out-perform previous rule-based methods that used large semantic tools. If used with an accurate sentence compressor, we believe this method for generating questions would be extremely accurate and convenient. Our approach is also not domain-specific and thus can be used in anything from automated education tools to better AI conversation generation.

REFERENCES

[1] A. Papasalouros, K. Kanaris, and K. Kotis. "Automatic Generation Of Multiple Choice Questions From Domain Ontologies." In *e-Learning, pp. 427-434. 2008*.

[2] H. Kunichika, T. Katayama, T. Hirashima, and A. Takeuchi. "Automated question generation methods for intelligent English learning systems and its evaluation." In *Proceedings of International Conference of Computers in Education 2004, pp. 2-5, Hong Kong, China, 2003*.

[3] J. Brown, G. Frishkoff, and M. Eskenazi. "Automatic question generation for vocabulary assessment." In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing, pp. 819-826, Vancouver, Canada, Association for Computational Linguistics, 2005*.

[4] J. Wolfe "Automatic question generation from text-an aid to independent study." In *ACM SIGCUE Outlook, vol. 10, no. SI, pp. 104-112, ACM, 1976*.

[5] M. Heilman, and N. Smith. "Good question! statistical ranking for question generation." In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 609-617, Los Angeles, USA, Association for Computational Linguistics, 2010*.

[6] D. Wu. "Stochastic inversion transduction grammars and bilingual parsing of parallel corpora." In *Computational Linguistics 23, pp 377-403, 1997*.

[7] I. Goto, M. Utiyama, and E. Sumita. "Post-Ordering by Parsing with ITG for Japanese-English Statistical Machine Translation." In *ACM Transactions on Asian Language Information Processing (TALIP) 12, no. 4, 2013*.

[8] G. Neubig, T. Watanabe, S. Mori, and T. Kawahara. "Substring-based machine translation." In *Machine Translation 27, no. 2, pp 139-166, 2013*.

[9] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. "Feature-rich part-of-speech tagging with a cyclic dependency network." In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, Volume 1, pp 173-180, Edmonton, Canada 2003*.