Proceedings of the Seminar

# Machine Learning: Theory and Applications

University of Colorado, Colorado Springs

August 7, 2015

**Editor: Jugal K. Kalita**

Funded by

National Science Foundation

## Preface

It is with great pleasure, we present to you papers describing the research performed by the NSF-funded Research Experience for Undergraduates (REU) students who spent 10 weeks during the summer of 2015 at the University of Colorado, Colorado Springs. Within a very short period of time, the students were able to choose cutting-edge projects involving machine learning, write proposals, design interesting algorithms and approaches, develop code, and write papers describing their work. We hope that the students will continue working on these projects and submit papers to conferences and journals within the next few months. We also hope that it is the beginning of a fruitful career in research and innovation for all our participants.

We thank the National Science Foundation for funding our REU project. We also thank the University of Colorado, Colorado Springs, for providing an intellectually stimulating environment for research. In particular, we thank Drs. Kristen Walcott-Justice, Qing Yi and Terrance Boult, who were faculty advisors for the REU students. We also thank Ali Langfels for working out all the financial details. We also thank our graduate students, in particular, Tri Doan, Abhijit Bendale and Ethan Rudd, for helping the students with any systems and programming issues. Francisco Torres-Reyes and his team also deserve our sincere gratitude for making sure that the computing systems performed reliably during the summer.

Sincerely,

Jugal Kalita
jkalita@uccs.edu
Professor

# NSF REU Seminar on Machine Learning

Department of Computer Science

University of Colorado, Colorado Springs

Osborne Center, A-343, Engineering Building

August 7, 2015: Friday

10*:30-10:35 AM:* Welcome Remarks by Dr. Kelli Klebe, Professor of Psychology, Dean of the Graduate School and Associate Vice Chancellor for Research and Faculty Development, University of Colorado, Colorado Springs

10*:35-11:50 AM Session Chair: Lisa Jesse, Co-founder, Intelligent Software Solutions, Inc., Colorado Springs, CO*

- 10:35-11:00 *Steve Cruz,* University of Colorado, Colorado Springs, CO: Multi-slab Models
- 11:00-11:25 *Ben Steele*, Colorado College, Colorado Springs, CO: Open Set Forests
- 11:25-11:50 *Chantz Large*, University of Colorado, Colorado Springs, C*O:* Direction-Boundary Set Reduction

11:50-12:45 PM: Lunch Break

12:45-2:00 PM *Session Chair: Dr. Suzette Stoutenburg, Principal Software Systems Engineer, MITRE Corporation, Colorado Springs, CO*

- 12:45-1:10 *Noah Weber*, Winthrop University, Rock Hill, SC: Correcting Verb Related Errors
- 1:10-1:35 *Jack Reuter,* Wesleyan University, Middletown, CT: Twitter Hashtag Segmentation
- 1:35-2:00 *Allen Burgett*, Normandale College, Bloomington, MN: Grouping and Testing Methods with Clustering Algorithms

2:00-2:15 Snack Break

2*:15-3:30 PM Session Chair: Dr. Qing Yi, Associate Professor, Computer Science, University of Colorado, Colorado Springs*

- 2:15-2:40 *SJ Guillaume*, Allegheny College, Meadville, PA: Mutant Selection Using Machine Learning Techniques
- 2:40-3:05 *Tiffany Connors*, Texas State University, San Marcos, TX: Modeling the Impact of Thread Configuration on Power and Performance of GPUs
- 3:05-3:30 *Michael Dennis*, DePaul University, Chicago, IL: Static Performance Prediction of Compiler Optimization

3:30 PM: Closing Remarks

7:00 PM: Farewell Dinner, Restaurant to be announced

# Table of Contents

# Multi-Slab Models

Steve Cruz
University of Colorado Colorado Springs

***Abstract*—In today's world of computer vision there is the problem of "open set", things that are unknown at training time. Optimization is key when dealing with open set problems. Ideally there is always positive and negative space that is taken into account. This article extends the idea of open set recognition to accommodate, or classify, for both positive and negative space, but also for the unknown space. We further extend a slab seen in previous work and modify it to incorporate a negative slab and have different labels when classifying. The goal of this paper is to be able to classify something as unknown instead of "I don't know".**

## I. Introduction

Open set recognition has become a common phrase in computer vision the last couple of years. Almost all problems have been closed set recognition, meaning that all classes were known during training. When dealing with open set, there are classes in testing that were not present when training. Recognition gives the assumption that some classes are recognized in a much larger space of classes that are not recognized. When dealing with the open set recognition problem, the data will have multiple known classes and many unknown classes. The unknowns are what make the problem open set. In this case, those unknowns would be the collection of data not recognized.

Every open set problem has some "openness" to it. A problem becomes more open when there are more classes of interest. As seen in Fig. 1, we will introduce unknown classes during testing, thus making the problem more open and moving away from the closed side of the spectrum. Facing an open set recognition problem introduces open space. Open space can be thought of as the region far away from any known data. A sample that was not seen during training, but was introduced during testing, would have no class nearby to support classification. In other words, the sample would not be labeled class 1, class 2, or class 3 if those were the only classes present. Also in open space, there is no clear answer as to how far is far. The idea of being close to a class still has the possibility of being far. According to Scheirer et al. [1], open set recognition is not well addressed by existing algorithms, as it has strong generalization.

Differentiating between the known and unknown classes introduces misclassification risk, which can cause a sample to be misclassified. An unknown class may be classified as part of another class, but be completely wrong. Another factor to consider is open space risk. Labeling space as positive that rarely has any data in the region is considered to be overgeneralized. That overgeneralization can be thought of as open space risk.

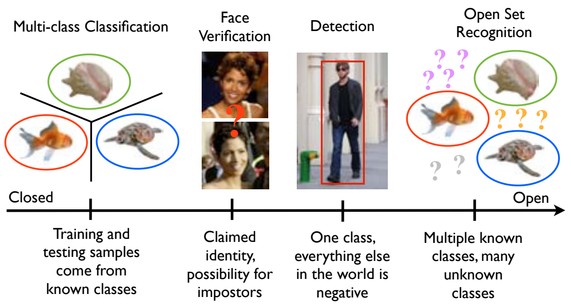

Multi-class Classification
Face Verification
Detection
Open Set Recognition
Closed
Open
Training and testing samples come from known classes
Claimed identity, possibility for impostors
One class, everything else in the world is negative
Multiple known classes, many unknown classes

Fig. 1. Shown are some of the vision problems faced today. The work in this article is to the right of the spectrum. The plan is to further the knowledge for the open set problem. Figure courtesy of Boult.

There are still problems that are present when dealing with open set recognition. A problem that still is somewhat unanswered is dealing with the multi-class setting. There is no clear basis that labels something as positive, negative, or unknown. When introducing a class at testing, there should be some sort of procedure that aides in labeling something as not having been seen before and not giving it the wrong label.

The goal of this work is to further develop the open set setting. We construct a formalization that expands the existing 1-vs-Set Machine [1]. More in depth, we extend the machine to have a positive and negative slab model that bounds the risk for each respective class. The resulting Multi-Slab Models give a way of voting for classification. Instead of having only one choice, we introduce the positive region, negative region, and the unknown region.

## II. Related Work

Open set recognition has been researched by Scheirer et al before [1] [2] [3]. A solution that they formalized for some of the open set problem was a new variation of a Support Vector Machine (SVM) [4], called the 1-vs-Set Machine [1], as seen in Fig. 2.

The idea of the 1-vs-Set Machine is to minimize the positive labeled space to address open space risk combined with constraints to the margin to minimize known risk. In other words, positive space was reduced down to only a specific region to reduce risk. The constrained positive region takes the form of a slab as seen in the shaded region labeled positive in Fig. 2. The slab risk model utilized empathized on having continuous positive only space. The slab in [1] only bounds the risk for the positive class, also known as the class of interest. It is referred to as a slab because it is rectangular in shape. The space between the two parallel hyperplanes A and Ω can be

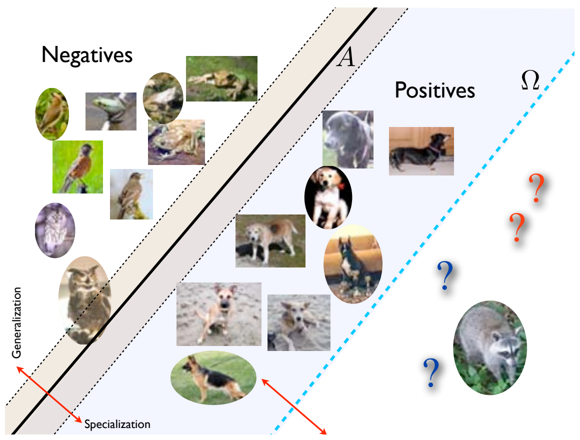


Fig. 2. The square images are from training and the oval images are from testing. The 1-vs-Set Machine adds a second plane Ω and defines an optimization to adjust A and Ω to balance empirical and open space risk. Figure courtesy of Boult.

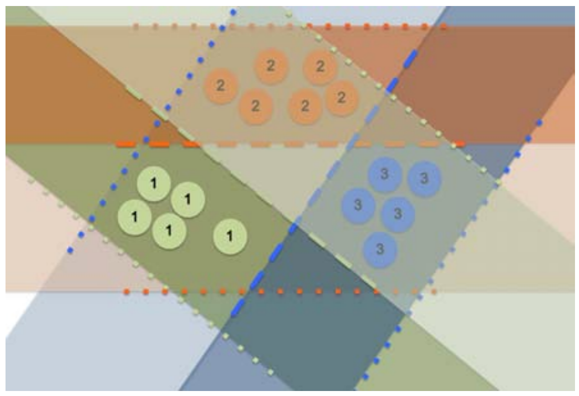


Fig. 3. The square images are from training and the oval images are from testing. The 1-vs-Set Machine adds a second plane Ω and defines an optimization to adjust A and Ω to balance empirical and open space risk. Figure courtesy of Boult.

referred to as the positive slab. The planes act as a boundary to avoid misclassification. Positive is what is of interest, while negative is perceived as an unsuccessful match or outside the boundary.The slab accounts for risk by way of its thickness. Depending upon the position of the classes, the slab can be moved in order to account for more classes.

The solution to the problem in [1], is addressed as 1-vs-set because only the closest data is being used. This solution, however, is still problematic because it can be thought of as 1-vs-all. The experiments and results never actually did anything that was multi-class. The name, 1-vs-set, is only used because 1-vs-all is more along the lines of saying that the machine has seen everything in the world, which is not the case. The 1-vs-Set Machine talks about the unknowns, but never actually labels anything as unknown. The raccoon seen in Fig. 2 should be classified as unknown because it is not part of the dog class or the bird class. Based on the current solution, however, the raccoon would be classified as negative and not as an unknown. This showcases that there is still room for improvement.

## III. Method

Since the 1-vs-Set Machine defined a slab for the positive region, we plan to use the current library of that work and extend it based on our criteria. The current library has an implementation of LIBLINEAR [5] and LIBSVM [4]. It extends the LIBSVM library to calculate the margin that separates the data and uses LIBLINEAR to classify the data linearly. The plane is calculated using parameters A and Ω. In this case A is what we call the near plane, sometimes referred to as the SVM margin, and Ω as the far plane. The space between the near plane and the far plane is what is considered the slab. This slab is what bounds the risk for a respective class.

Another extension for the 1-vs-Set Machine is to make the problem multi-class instead of 1-vs-all. For example, let's say we currently have 3 classes and we introduce a random sample at testing. Each of the three classes would have a positive slab and a negative slab as seen in Fig. 3. If the sample falls within a positive slab then it would get a vote for that particular class. If the sample, however, gets more votes for unknown than it does for positive, then the sample would be classified as unknown. This is because the majority of the data has not seen it before. Getting a negative vote, a positive vote, and a unknown vote for the 3 class example would mean that the sample would be classified positive for that particular positive vote. This is because a class can at least differentiate between not knowing what the sample was and that the sample is just not part of the class. Also if the sample fell into the negative slab for all three classes then it would be classified as unknown because it is not part of any classes present.

## IV. Evaluation

The data sets used for these experiments are the re-casted versions of the LETTER [6] and MNIST [7] data sets that are seen in [3] and [2]. The LETTER data set was re-casted and named OLETTER to better fit the open set problem. It has 15,000 points, 26 classes, and 16 features with 15 random distinct labels as known. Openness varied by adding subsets of the remaining 11 labels. We chose to use this data set because although it was thought to be solved with previous algorithms, it is a significant challenge for the current state of the art.

The MNIST data set was re-casted and named OMNIST to also better fit the open set problem. It has 60,000 points, 10 classes, and 778 features with 6 random distinct labels as known. Openness varied by adding subsets of the remaining 4 classes.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Most would assume that accuracy would be the optimal evaluation, however it is not. Accuracy cannot be used with open set recognition because the total number of classes is always undefined. Eq. 1 can be used, but it does not provide sufficient evidence between correct positive and negative classifications. The goal is to point of the positive samples that are within the mass region of the negatives. An evaluation that is optimal for open set recognition is f-measure because it provides a consistent comparison from both precision and recall numbers. F-measure can be thought at the combination of both precision and recall.

$$\text{F-measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \qquad (2)$$

When training with OLETTER and OMNIST, the precision was about the same value as the 1-vs-Set Machine. However once more and more unknown classes started to be introduced, f-measure started increasing. The slabs classified some of the points as unknowns, which were misclassified with the 1-vs-Set Machine. The algorithm performs much better with larger data, however, it slowly starts to diminish over time.

## V. Possible Extensions

Based on the optimization of the Multi-Slab Models, this current solution still has room for improvement. We plan to implement Extreme Value Theory (EVT) [8] to further improve the models. Using EVT we can normalize the distances of a point to the plane as a way of estimating probabilities. The idea would be to use an arbitrary number of values from the tail of the data. This is an improvement to a solution because instead of votes, we will have 2 possible probabilities. One is the probability of being on one side of the plane, which will be referred to as confidence. Also the other is the probability of being within the known data space, which will be referred to as pertinence.

Extreme Value Theory provides a way to determine probabilities regardless of the distribution of data. The extreme value scores of any distribution coming from a recognition algorithm can be modeled by EVT distribution [3] [9]. This is good for open space recognition because some data be out in the extremes of open space.

## VI. Conclusion

This article provides a path on improving open set recognition. F-measure was on par with the 1-vs-Set Machine and even classified things that should have been labeled as unknown, unknown. With the extensions added to the current implementation, we hope to come up with a state of the art for all open set problems.

## References

[1] W. J. Scheirer, A. Rocha, A. Sapkota, and T. E. Boult, "Towards open set recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence (T-PAMI)*, vol. 36, July 2013.

[2] L. P. Jain, W. J. Scheirer, and T. E. Boult, "Multi-class open set recognition using probability of inclusion," in *Computer Vision – ECCV 2014*, ser. Lecture Notes in Computer Science, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Springer International Publishing, 2014, vol. 8691, pp. 393–409.

[3] W. J. Scheirer, L. P. Jain, and T. E. Boult, "Probability models for open set recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence (T-PAMI)*, vol. 36, November 2014.

[4] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, pp. 27:1–27:27, April 2011.

[5] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wan, and C.-J. Lin, "Liblinear: A library for large linear classification," *Journal of Machine Learning Research 9*, pp. 1871–1874, 2008.

[6] P. W. Frey and D. J. Slate, "Letter recognition using holland-style adaptive classifiers," *Machine Learning*, vol. 6, no. 2, pp. 161–182, 1991.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278 – 2324, November 1998.

[8] S. Kotz and S. Nadarajah, *Extreme Value Distributions: Theory and Applications*. Imperial College Press, 2000.

[9] W. J. Scheirer, A. Rocha, R. Michaels, and T. E. Boult, "Meta-recognition: The theory and practice of recognition score analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 33, pp. 1689–1695, 2011.

# Open Set Forests

Ben Steele
UCCS Machine Learning REU
Benjamin.Steele@ColoradoCollege.edu

***Abstract*—Current ensemble classifiers use closed set decisions for each individual classifier in the ensemble. This applies to random forests, where each decision tree is a closed set classifier. In this article we propose using statistical extreme value theory to determine the relevance of the input to each decision tree in a random forest. this allows us to make open set decisions for the forest as a whole by looking at the average similarity between each tree and the given datapoint. We found that it is difficult to preserve the closed set accuracy when working with open set data in a random forest. However, it is possible to preserve the open set accuracy as unknown classes are added to testing.**

## I. Introduction

Classification problems are currently set up in a closed set world. In closed set, any input to a model is assumed to belong to one the the classes it was trained on. This works well when this assumption holds true, which is often the case for specific classification problems. However, when we do classification of real world problems, we often cannot account for all of the classes we may see. Open set is a form of modeling in which the model is able to determine whether an input is similar to the data it was trained on.

We call this similarity to the training data pertinence. If the pertinence for a point is too low, the model cannot accurately predict the point because the point is likely not in any of the classes it was trained on. A classic example of an open set problem is object recognition in images. There is no way to account for every image the model might see during training, so it is beneficial for a model to be able to determine if an input is similar enough to the training data to be considered a known object.

Random forests are among the highest performing machine learning techniques. Because they are comprised of decision trees, they inherently have very low bias. By averaging the output of each tree they reduce variance which results in a relatively low bias, low variance model. The original implementation of random forests took the majority vote from all the trees to decide which class to output [3]. This means each tree is given equal weight in the forest's decision.

Probability estimation trees (PETs) are able to estimate the probability that a given data point is in a class. PETs give an estimation of the probability that the class they have chosen is correct, which allows the tree to weigh its decision depending on whether it has a high or low probability. However, these PETs are very poor estimators in general. Bagging, the basis of random forests, substantially improves PETs classification accuracy [5]. While bagging is accepted to improve classification accuracy in PETs, the best algorithm to do so is still an open question [2]. We believe the same concepts used in PETs can be used to create an open set tree.

Instead of predicting the probability that a point is in a class, we will predict the probability that a point is in any of the classes. Most implementations of PETs only compute probability at the leaf node, but this will not be sufficient for our algorithm. We will be using decision trees in a random forest, so the branching decisions will be limited to linear boundaries parallel to the axes. If we only check at the leaf we will only be checking pertinence along one axis, meaning we only get the pertinence of one feature. Because of this we will calculate pertinence at each node in the tree. We can then combine the pertinence of each node a point goes through to determine the overall pertinence.

Extreme value theory (EVT) [6] has been used for open set implementations before, so we will be using the libMR library to create EVT models for each tree. Decision trees are well suited for EVT because for data to be fit to an EVT model, it must first be represented in 1 dimension. In the case of a decision tree, each decision boundary is parallel to an axis, so only one feature is used in the decision. Because only one feature is used, each decision down the tree reduces the problem to a single dimension. This means the tree is already constructed in a way that EVT can easily be implemented at each node.

## II. Method

We will be modifying the random forest from the Sci Kit learn library [4] to produce an open set forest. A decision tree in a random forest produces disjoint classes with linear boundaries parallel to the axes. At each node of a decision tree a linear boundary called the decision boundary is created, separating the data points of that node. Our modification adds EVT models to each of these decision boundaries in order to compute pertinence.

### A. Node Pertinence

To create the model we must first reduce the points at a node to 1 dimension. This is done by taking the distance from the points to the decision boundary. Because the decision boundaries are parallel to the axis, the boundary is really just a threshold for a single feature. To calculate distance we only need to subtract the threshold from the point's value for that feature. We consider points greater than the decision boundary to be positive and points less than the decision boundary to be negative. Once all of the points are represented in a single dimension, we can fit an EVT model to them. We fit two models at each node, one to fit the high end (greatest points) and one to fit the low end (smallest points). The EVT models are monotonic, so a single model can only describe one side of the dataset. These two models allow us to compute the pertinence of any point that reaches the node.

To compute the pertinence of a point for a single node we first find its distance from the decision boundary. We can then plug that distance into each EVT model. We use the minimum of the two models as the final pertinence. The minimum is used because the model on the opposite side of the data from where the point is will give the point a very high pertinence (because the model is monotonic). The minimum of the two models is the pertinence that we are interested in because it will belong to the model that is on the same side of the data as the point.

### B. *Tree Pertinence*

To find the pertinence of a point using the entire tree we combine the pertinence of each node that the point passes through in a few ways. Choosing a method of aggregating the pertinences from each node a point passes through was nontrivial and we chose 3 to determine experimentally which is best. One method was simply taking the minimum of all the pertinences. This would accentuate a single feature that is significantly not pertinent but would not be good at distinguishing points that are only slightly not pertinent for many features.

Our second method takes the average of all the nodes. This is better when many features are slightly not pertinent, but greatly softens the impact a single variable can have on the overall pertinence.

The third method is taking the product of each node's pertinence. This should allow a single extreme feature to have a larger impact than using the average, while maintaining the ability to find many features that are slightly not pertinent. this is also a logical way to combine the nodes because when you find the probability of multiple independent things happening together, you multiply their individual probabilities. This is essentially what we are doing.

Not every leaf in a decision tree is at the same level in the tree. This complicates the product because points with more nodes will generally have a smaller product. To correct for this we use a log function.

$N$ = number of nodes

$P$ = product of pertinences of each node

$$\log_{10^N} P$$

### C. *Forest Pertinence*

The forest averages the pertinences from each of its trees to determine the overall pertinence. Once we calculated the overall pertinence, we needed a way to decide what is pertinent and what not. For this we trained a threshold. Using linear regression, we trained the threshold to maximize accuracy on a validation set. Once the threshold is trained the model is completely trained and can be tested.

## III. Evaluation

Our primary focus in this study is to create a foundation for open set in random forests. To evaluate the model we use the accuracy score of the original forest against the open set forest using min pertinence, average pertinence, and product

| classes | original | min | average | product |
|---|---|---|---|---|
| 0 | 0.96 | 0.71 | 0.78 | 0.71 |
| 1 | 0.76 | 0.69 | 0.72 | 0.68 |
| 2 | 0.64 | 0.69 | 0.68 | 0.67 |
| 3 | 0.55 | 0.67 | 0.64 | 0.65 |

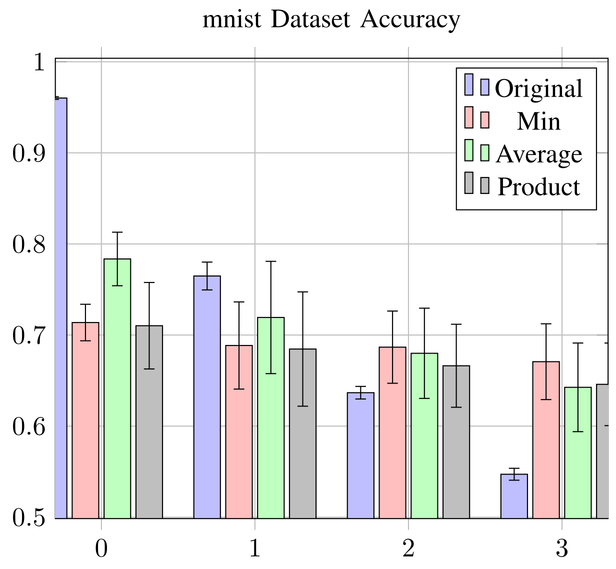


Fig. 1. A graph of the accuracy of all 4 algorithms with different numbers of unknown classes. The x-axis is the number of unknown classes being tested. At 0 on the x-axis it is using the 4 classes the model was trained on, but no unknown classes. The y-axis is the % accuracy of the model.

pertinence. We found during testing that the more continuous features the data has, the better our algorithm will perform. Because many datasets have large numbers of non continuous data we had to choose specific datasets and generate our own. We used the mnist dataset from the UCI Repository [1] as well as generated datasets. The mnist dataset has 70,000 data points, 780 features, and 10 non separable classes.

The generated datasets were non separable and used either normal distributions or gamma distributions for each class. They contained 1,000 points per class, 50 features, and 100 classes.

## IV. Results

In testing we found that our algorithm is extremely variable depending on the dataset that is used. When using the mnist dataset we constructed a forest with 40 trees and used 4 classes to train and 3 classes to validate. we tested using the remaining 3 classes in 5 trials. When the model was tested using only the classes used during training the original forest was able to get 96% accuracy while the open set forest got at best 78% with average pertinence, as seen in figure 1.

As unknown classes were added, the accuracy of the original forest dropped much faster than the open set implementations and once two classes had been added all of the open set implementations outperformed the original forest

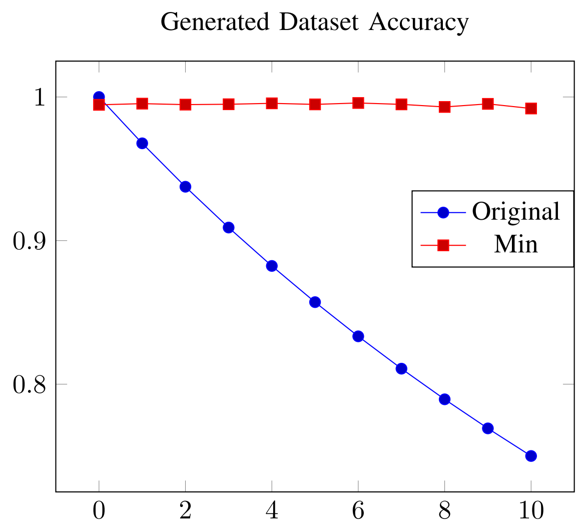


Fig. 2. A graph of the accuracy of the original forest and the min pertinence random forest. The x-axis is the number of unknown classes being tested. At 0 on the x-axis it is using the 30 classes the model was trained on, but no unknown classes.

in accuracy. when 2 and 3 unknown classes are added to the 4 known classes min pertinence has the best accuracy while the original forest does the worst. The min pertinence implementation did drop in accuracy as unknown classes were added, but no more than 2% for each unknown class.

using the generated data we only tested the original forest against the minimum pertinence open set implementation. We used forests with 20 trees, training on 30 classes and validating with 8 classes. When testing on only the training classes, the original forest had 100% accuracy while the open set forest had 99.4% accuracy as seen in figure 2. This is a far more acceptable loss of accuracy compared to the mnist dataset. As unknown data is added, we see a similar trend as in the mnist dataset. The original forest drops in accuracy far faster than the open set forest. after adding 29 unknown classes, the original dataset performed at 43.3% accuracy while the open set forest was still at 98.1% accuracy.

While the accuracy of the open set forest is far greater in the generated data than in the mnist data, the behavior when unknown data is added is similar. In both cases the accuracy falls slowly, showing that accuracy can be preserved as unknown data is added, even if that accuracy is far less than in a closed set implementation.

## V. Conclusions

Open set forests succeed in maintaining accuracy in the presence of unknown classes, but cause a loss in their base accuracy (accuracy on closed set test) to do so. It is promising that our implementations are able to maintain a relatively stable accuracy. In both datasets, the accuracy does not significantly decrease as more unknown classes are added. In the generated dataset we show that even with very high base accuracy, adding unknown classes will not cause it to drop.

The 3 implementations we used for open set forests all performed similarly compared to the original forest. Of the three, min pertinence seems most stable because it has the smallest decrease in accuracy as unknown data is added. However, with small amounts of unknown data the average pertinence performs better. Depending on the nature of the data, one may be better than the other. Product pertinence consistently performs poorly in these tests, making it the worst of the 3 implementations.

The generated dataset's results are far different from the mnist dataset. We believe the generated data is probably very close to separable because the high number of dimensions creates a large space even when the range in each dimension is small. This does not make the results insignificant. the generated dataset shows that our algorithm can work very effectively with medium size dimensionalty and many classes when the classes are near separable.

The mnist dataset had much poorer results than the generated data. The base accuracy dropped over 20% on known classes. even though the accuracy was better when a few unknown classes were added, this is a tradeoff most users would opt not to take. We believe this drop in accuracy is caused because the trees are unable to produce good estimations of pertinence.

The decision trees only see one feature at a time, one feature at each node. This is because only looking at a single feature for each node makes a forest much faster than if multiple features could be considered at the same time. This means the algorithm is most fit at finding outliers that have one or more features that are grossly different than those features in the training data. This may be why min pertinence seems to be the best metric, it captures the single feature that separates a datapoint from the rest of the data.

This is more of a problem in higher dimensional spaces because a point can easily have similar feature values to the training points when looking at the features individually, but that points euclidean distance from the training points can be very large at the same time. A simple example of this in 2d is if you have a circle inside a square, both with the same width. take a point on the corner of the square. if you look at that point from either axis it looks like it is right on the edge of the circle, but in 2d it is far from the edge of the circle. In much higher dimensions this error space becomes very large and this hinders the ability of these trees to predict pertinence. This phenomena is somewhat mitigated because as the tree gets deeper, the dataset is cut into smaller and smaller sections and the error space becomes increasingly small. However, this is not the only issue in higher dimensions.

With higher dimensions, the tree is going to make decisions on only a small portion of the features before it can determine a class for any point. This means some features that obviously make a point not pertinent may not be the feature that the model is using to determine that point's class. These issues can be fixed by allowing the use of any hyperplane at each node, not only planes parallel to the axes.

A significant limitation to open set forests is that they perform extremely poorly without a majority of continuous features. This occurs because only one feature can be evaluated at a time. A binary feature cannot give much information on pertinence on its own, and similarly any feature with a low number of possible values becomes a very poor indicator of pertinence. This also means any non numeric feature is insignificant in determining pertinence. This would not be as big an issue if the decision boundaries were not parallel to the axis, so future work could be done on decision trees that can use any hyperplane as a decision boundary. This would be far more computationally intensive, but is likely to be much more accurate and robust.

## VI. Acknowledgments

We would like to thank the University of Colorado at Colorado Springs and the National Science Foundation's Research Experiences for Undergraduates program, who's Grant 1359275 funded this research.

## References

[1] CL Blake and C Merz. Uci repository of machine learning databases. *University of California, Department of Information and Computer Science*, 1998.

[2] Henrik Bostrom. Estimating class probabilities in random forests. In *Sixth International Conference on Machine Learning and Applications*, pages 211–216. IEEE, 2007.

[3] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] Foster Provost and Pedro Domingos. Tree induction for probability-based ranking. *Machine Learning*, 52(3):199–215, 2003.

[6] Walter J Scheirer, Anderson Rocha, Ross J Micheals, and Terrance E Boult. Meta-recognition: The theory and practice of recognition score analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1689–1695, 2011.

# Direction-Boundary Set Reduction

Chantz T. Large
University of Colorado – Colorado Springs

***Abstract*—Reducing the size of the dataset is of interest to those working with resource constrained devices or performing incremental learning. This paper introduces a new direction-boundary Set Reduction strategy for reducing the size of the dataset. The algorithm in its current state performs well for problems involving large-scale sparse datasets which are reasonably linearly separable. Executed on the popular letter scale data set, it has been shown to reduce the size of the dataset by 37% while maintaining its model generating integrity (2% accuracy reduction).**

## I. Introduction

The direction-boundary algorithm works well for point-preservation between phases of incremental learning, as well as, dataset maintenance on resource constrained devices. The algorithm is designed to be simple, lightweight and easily-extensible. At its root, the algorithm is design to identify the relevant boundary points of the dataset; preserving only the points necessary for defining the shape of the set itself. In its current state, the algorithm has only been developed with datasets involving convex classes which are generally linearly separable.

Applications for this algorithm extend to those involving; incremental learning where maintenance of the training batch between training phases is unreasonable, resource constrained devices where storage capacity or available memory is scarce, or where reducing the training time of the dataset itself is of concern.

Many current machine learning algorithms share much consideration for the amount of resources necessary to conduct learning, or are constructed on the premise that the training batch will be maintained between training phases. While concurrent algorithms continue to elevate the computational bounds for unprecedented modeling of large datasets, many of the previously mentioned applications are unable to capitalize on these innovations. This work in particular was motivated by the incremental learning library, LIBLINEAR.

## II. method

The method for point-selection involves the following steps; projection and fitting of the direction vectors to the original dataset, identifying points residing nearest to the projected boundary, Fig. 1.

Projection of the boundary involves calculating the direction vectors defined by the mean center of the dataset and a corresponding point within the dataset. After all points have been projected the boundary is then fitted to the original dataset by scaling by the maximum and minimum values in all dimensions. Finally, distance is measured and points nearest to the projected boundary are retained.

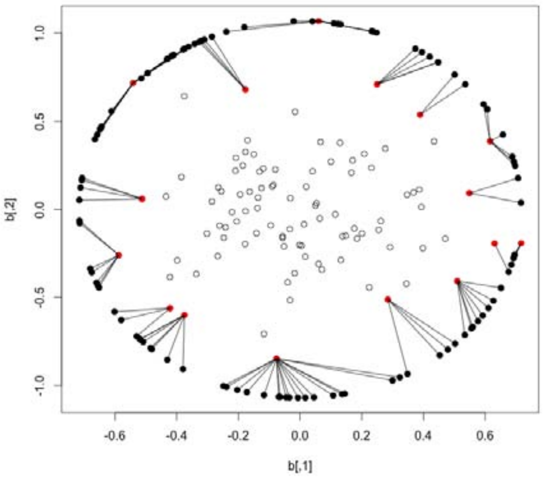


Fig. 1. 2-dimensional plot illustrating point-identification; empty circles represent original dataset, filled circles represent projected boundary, filled-red circles represent selected points. Lines to and from boundary points to selected points illustrate voting.

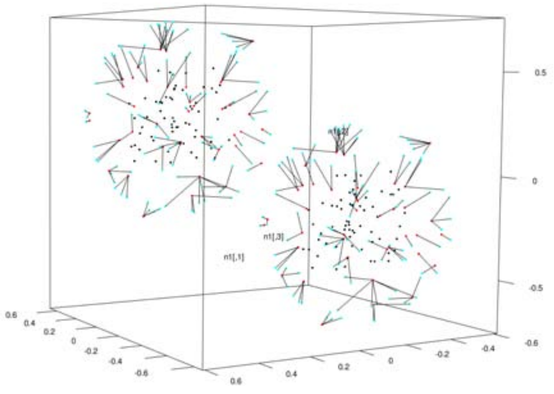


Fig. 2.

## III. evaluation

Early evaluation of the direction-boundary algorithm demonstrates promising results. Preprocessing of the letter training set effectively reduced the size of the set by 37% and while retaining acceptable model generating integrity (2% accuracy loss). As the scale of the data set increases, the performance of the algorithm improves, Fig. 5.

For illustrative purposes, the algorithm has been demonstrated to be effective on skewed datasets as well, Fig. 4.

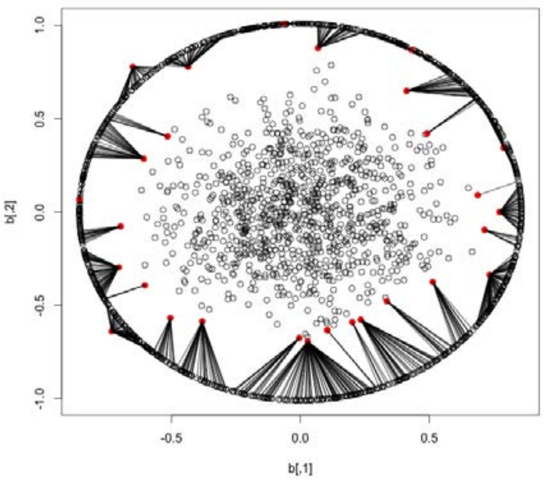


Fig. 3.

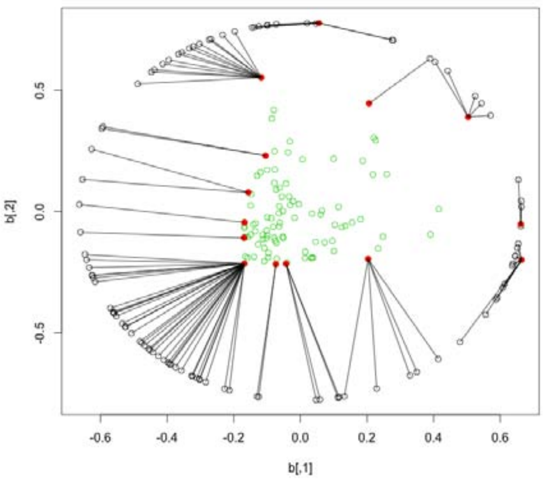


Fig. 4.

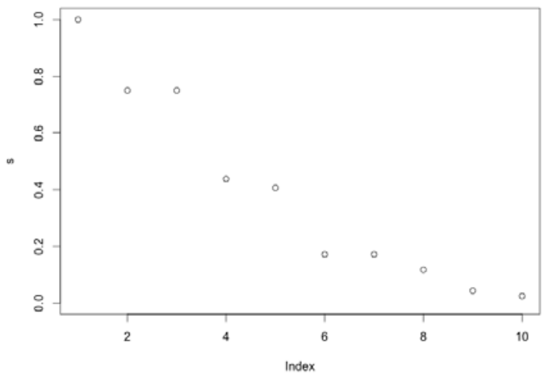


Fig. 5.

# Correcting Verb Related Errors

Noah Weber

***Abstract*—Verb related errors are common in the writings of those learning English as a second language (ESL), due to the various uses and forms of verbs. Though much of the research in automatic ESL error correction has focused solely on fixing either determiner or preposition errors, there has been recent interest in developing methods for automatic verb error correction. Previous approaches to this problem have typically relied on data from error annotated learner's corpora to learn common types of verb errors and the situations in which they are most likely to occur in. In this paper, we compare the results of two models for estimating the probability that a certain verb tense and aspect is correct given the sentence context and the original verb tense and aspect. The first model is a generative model which estimates this value by modeling the prior probability distribution as well the probability distribution for a instance of a verb tense and aspect being an error given the intended tense aspect and the sentence context. From these two distributions, we can generate a model for the posterior probability distribution. The second model is a discriminative model which models the posterior distribution directly.**

## I. Introduction

Due to the increase of English as a second language (ESL) students, the interest in automated grammar checking systems has increased in the past decade. Much of the previous work in this field has looked at the correction of article and preposition errors [2][7][8], both respectively being the two most common errors in the writings of ESL students [6]. Though verb related errors are the third most common type of error in ESL writings, relatively little research has been done on the topic.

The problem of correcting verb errors also presents several additional difficulties that separate it from article or preposition correction. Due to the various ways in which verbs can be used, identifying verbs in a text is typically a more involved process when compared to article and preposition detection. Automatic verb correction may also involve detecting missing verbs, in addition to correcting the verbs present in the sentence. For example, in the example sentence:

> *The problem is that the same step being repeat for every interval.

the error to be fixed involves both inserting the verb *is* and changing the verb *repeat* from its base form to its past participle form. The correct choice of verb also typically depends on the context in which it is used which further complicates the task at hand. In addition, the form and tense of verbs usually depends not only on the surrounding sentence context, but also on the form and tense of surrounding verbs. This means that verb correction has an additional complication in that correcting a single verb might have an effect on what other verbs in the sentence should be corrected to, if they need to be corrected at all. In this paper we plan to build upon previous research on this topic and develop a new method for correcting verb errors that utilizes both machine learning and linguistic theory.

## II. Related Work

Previous research in the field of automated grammar checking has mostly focused on preposition and article related errors. Because prepositions and articles are both closed word classes, solutions to the problem have typically involved the use of multiclass classifiers, with each individual preposition or article making up a single class [2]. Some approaches treat grammar checking as a machine translation problem, and simply use standard statistical machine translation techniques to translate an ungrammatical sentence to a grammatical one [4]. Some recent work has looked into the problem of correcting several types of errors simultaneously. Dahlmeir and Ng (2012) proposed a model that takes corrects article, preposition, and noun count errors using a beam search decoder. Their model consists of proposers for each type of error, as well as expert models for each type of error. The proposers generate new sentences by making small edits to the current sentence. The expert models are used to score the grammaticality of the sentence. Similar to the decoding step done in statistical machine translation, a beam search is done in order to find the sentence that results from the highest rated series of edits. Work done by Wu and Ng (2013) similarly tries to fix article, preposition, and noun count errors simultaneously, but reformulates the problem from a searching problem to an integer programming problem.

Due to many difficulties involved with verbs, the verb correction problem must be approached in a different manner. The existing research on verb correction use several different approaches. The work of Lee and Seneff (2008) deals with verb form errors by looking for parse trees that are likely to be produced from a verb error, along with probabilities derived from n-gram counts to identify and correct errors. While this method worked well for verb agreement and form mistakes, it did not account for verb tense errors. The work of Tajiri et al. (2012) aims to correct verb tense errors by treating the problem as a sequence classifying problem where each verb is labeled with a tense. The label chosen for an individual verb depends on the labels chosen for surrounding verbs. They use conditional random fields for sequence labeling, allowing them to use both syntactic and semantic features to aid in labeling. Recent work by Rozovskaya et al. (2014) looks into correcting tense, agreement, and form errors. Their work takes advantage of several linguistic properties of verbs, most notably verb finiteness, in order to guide their statistical learning method. Their learning method classifies verbs in a text as either having an aspect, tense, form, or no errors. Error correction is also handled using a multiclass classifier, with each class of verb error having a unique error correction model.

## III. Methodology

Our proposed method will build upon previous work and utilize linguistic features in tandem with machine learning methods. The method we propose closely resembles the Bayesian noisy channel model used in tasks such as speech recognition and statistical machine translation. In this model, we treat the possibly incorrect sequence of verbs as a corrupted version of some correct sequence of verbs. The goal of this method is to model the probability distribution:

$$P(C|O, S)$$

Where $C$ is a proposed correct tense aspect for a verb instance. $O$ is the tense aspect of the verb instance originally put down by the writer and $S$ is the sentence context. We try and compare two different models to estimate $P(C|O, S)$. The first way is using a generative model. For the generative model we rewrite the distribution $P(C|O, S)$ using Bayes Theorem as:

$$P(C|O, S) = \frac{P(O, S|C)P(C)}{P(O, S)}$$

For our purposes we rewrite this as:

$$\begin{aligned} P(C|O, S) &= \frac{P(O, S, C)}{P(O, S)} \\ &= \frac{P(S)P(C|S)P(O|S, C)}{P(O|S)P(S)} \\ &= \frac{P(C|S)P(O|S, C)}{P(O|S)} \end{aligned}$$

Since the bottom $P(O|S)$ term remains constant we can simply ignore it. Our goal for the generative model is thus to model the distribution of $P(C|S)$ and $P(O|S, C)$. The seeming advantage of using a generative model in this instance is that is allows one to utilize data from both well-formed corpora in the estimation of $P(C|S)$, as well as data from error annotated learner corpora for the estimation of $P(O|S, C)$. However, as the results show this actually may not be the case. To estimate both $P(C|S)$ and $P(O|S, C)$ we treat the problem as a classification problem, as traditionally done in prior research in automated grammar correction systems. For our classifier we use a Maximum Entropy (MaxEnt) classifier. The reason being its use as a classifier in previous studies done in automated grammatical error checking [2]. However, we plan to use other classifiers as well in the future.

Our second model is a discriminative model which estimates the distribution $P(C|O, S)$ directly using a MaxEnt classifier. The data used in the training and testing of this model as well as the model for the $P(O|S, C)$ distribution comes from the error annotated FCE corpus [9]. The data used for the model the $P(C|S)$ distribution comes from several different sources. The data for this model comprises of sections from the Brown corpus, the MASC section of the Open American National Corpus, as well as a fully corrected version of the FCE corpus. Many of the features we utilize come from features used by both Tajiri et al. (2012) as well as Rozovskaya et al. (2014).

| Feature | Description |
|---|---|
| verb lemma | The lemma of the current verb |
| left/right lemma | The lemma of the words to the right/left of the current verb |
| subject | The word, pos, person, and number of the sentence subject |
| determiner | The determiner for the sentence subject |
| left/right noun | The word, pos, and person of nouns to the left and right of the current verb |
| first | Whether the verb is the first in a chain of verbs |
| last | Whether the verb is the last in a chain of verbs |
| governor | The governor of the verb and the relation type between them |
| governee | The governee of the verb and the relation type between them |
| left/right time adverbs | Time adverbs to the left/right of current verb |

Table 1: Description of features used in model

## IV. Results

To test the models we split off a section of the FCE corpus solely for use in testing. The section contained around 7052 instances of verb sequences, with around 200 of these sequences having some type of error. To test, we used our trained classifier to classify each unlabeled verb instance into a tense and a aspect. We then compared the tense aspects generated by the model with the actual tense aspects put down by the annotator. As our evaluation criteria we use both precision and recall. As our results show, the discriminative model outperforms the generative model by a large margin.

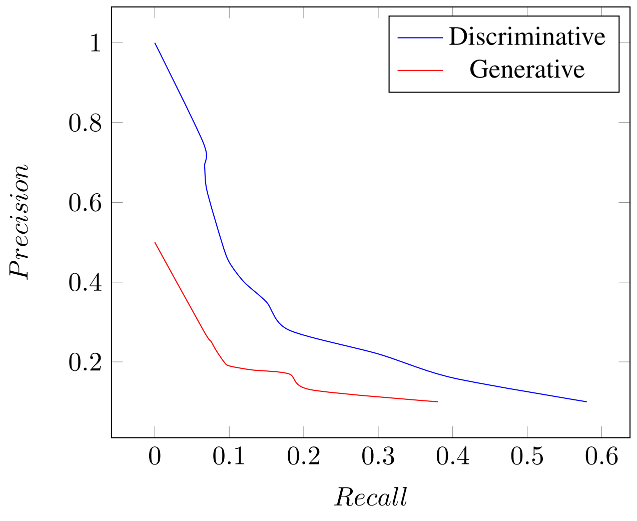


Figure 1: Precision and Recall Curve for both models

## References

[1] J. Lee and S. Seneff. 2008. "Correcting misuse of verb forms". *Proceedings of the ACL-08*, pg 174−182, Association for Computational Linguistics, Columbus, Ohio, June.

[2] J. Tetreault and M. Chodorow. 2008. “The ups and downs of preposition error detection in ESL writing”. *In Proceedings of the 22nd International Conference on Computational Linguistics”*, pg. 865−872, Manchester, UK, August.

[3] T. Tajiri, M. Komachi, and Y. Matsumoto 2012. “Tense and aspect error correction for esl learners using global context”. *In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics* (Volume 2: Short Papers), pg 198−202, Association for Computational Linguistics. Jeju Island, Korea, July.

[4] C. Brockett, W. Dolan, and M. Gamon. 2006. “Correcting ESL Errors Using Phrasal SMT Techniques”. *In Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pg 249256, Association for Computational Linguistics. Sydney, Australia, July.

[5] A. Rozovskaya, D. Roth, and V Srikumar. 2014. “Correcting grammatical verb errors”. *In Proceedings of EACL*, pg 358−357.

[6] C. Leacock et al. 2010. “Automated grammatical error detection for language learners.” *Synthesis Lectures on Human Language Technologies*, pg. 15−27

[7] D. Dahlmeier and H. Ng. 2012. “A beamsearch decoder for grammatical error correction”. *In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pg 568−578.

[8] Y. Wu and H. Ng. 2013. “Grammatical error correction using integer linear programming”. *In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pg 1456−1465.

[9] H. Yannakoudakis, T. Briscoe, and B. Medlock. 2011. *A new dataset and method for automatically grading esol texts. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pg 180−189, Association for Computational Linguistics. Portland, Oregon, USA, June.

# Twitter Hashtag Segmentation

Jack Reuter — Wesleyan University — jreuter@wesleyan.edu

*Abstract*—**This paper describes an effort to segment non-delimited strings of English text, specifically hashtags. Extensive research has been done in word segmentation, particularly in languages like Chinese, for its lack of spacing between words, and German, for its extensive use of compounds. English, however, has been relatively untouched. The goal of this research is to adapt and extend methods used in prior research to fit the demands of modern English.**

## I. Introduction

#wordsoftheday ... word soft he day?
#statefarmisthere ... state far mist here?
#brainstorm ... bra in storm?
#doubledown ... do u bled own?
#votedems ... voted ems?

WORD segmentation is an important first step in natural language processing (NLP). It is difficult to derive accurate meaning from a piece of text without first determining the words that it comprises. Twitter, for those unfamiliar, is a social media site that allows users to share brief (less than 140 character) "tweets" with their followers. In their tweets, users have the option of including hashtags, a form of metadata labeling. Typically, hashtags contain no delimiters between words. From these hashtags, Twitter compiles lists of the most noteworthy topics of the day, month, year.

Despite the abundance of garbage inherent in social media, trending hashtags and their related posts often follow relevant topics and provide insight into public opinion. Twitter mining has applications in public health [1], political sentiment [2], and emergency coordination [3]. The analysis of hashtags and the relationships between them is a bountiful area of research. The goal of this project is to further the reach of such research by enabling machines to process individual hashtags into understandable chunks of data.

## II. Problem Definition

Formally, the problem is as follows: given a string $a_1a_2...a_k$, where each $a_i$ is a meaningful substring (i.e. a word, name, abbreviation, etc.), determine where the boundaries lie between each $a_i$ and $a_{i+1}$. For example, given a string such as "randompicoftheday" return "random pic of the day". Initially it may seem like a simple problem. Simply loop through all substrings of the input, looking for matches in a dictionary. Once all matches have been found, segment the string accordingly.

The real problem, however, lies in the phrase "then segment accordingly." From Xue 2003, "The key to accurate automatic word identification...lies in the successful resolution of these ambiguities and a proper way to handle out-of-vocabulary words" [5]. Although referring to the segmentation of Chinese characters, the sentiment is still very much appropriate. In our case, ambiguities occur when a string has more than one meaningful segmentation, out-of-vocabulary words when our dictionary fails us. Both scenarios occur frequently, and the success of our methods depends on the handling of such situations.

Consider, for example, the string "brainstorm". Using a table lookup, a machine could read this as either "bra in storm", "brain storm", or the correct, untouched "brainstorm". Adopting the jargon of Webster and Kit [4], we will refer to this as *conjunctive ambiguity*, i.e. when a meaningful string can be broken up into $n$ meaningful substrings. The natural solution of course is to take the segmentation with the largest matched word. This maximum matching approach handles conjuctive ambiguity very well, for it is unlikely that a syntactically sound clause happens to merge into a larger word, yet it is quite common for a larger word to break up into dictionary-matchable pieces.

Maximum matching fails, however, in cases of *disjunctive ambiguity*—the situation when a string "ABC" can be broken up meaningfully as either "AB C" or "A BC". "doubledown", for example, could be interpreted as either "doubled own" or the correct "double down" (or even "do u bled own" which exhibits both conjunctive and disjunctive ambiguity). Taking the maximum matching here would result in the incorrect segmentation "doubled own". Disjunctive ambiguity, it appears, requires a bit more syntactic knowledge to resolve.

The other main issue is the handling of strings outside of out dictionary. The string "votedems", for instance, should be returned as "vote dems" and not "voted ems", while our dictionary may only contain the abbreviation "ems" and not "dems". With typos, abbreviations, online slang, and just a general abundance of linguistic rule-breaking in hashtags, these situations are bound to occur, and occur frequently. For this project to succeed, such unknowns must be recognized and handled appropriately.

## III. Related Work

Since English text is almost always whitespace delimited, little to no work has been done in English word segmentation (excluding morphological segmentation). Such research, however, has been widespread in other languages where segmentation is an important first step for NLP. German and Chinese, for example, due to their large compounds and lack of delimitation, respectively, have been researched for years, yielding successful results. The following is a brief compilation prior methods which have been both succesful in their own rights and adaptable, in parts, for English usage.

1) *Parallel Corpora:* In [7], Koehn and Knight use data from monolingual and parallel corpora to learn splitting

rules for German compounds, achieving 99.1% accuracy. After first obtaining all possible segmentations of a compound into known words based on a German corpus, they choose most-likely segmentations by examining word frequency in that same corpus (favoring more common words), as well as word occurrence in a parallel English corpus, (favoring segmentations whose translations contain the same words as the translation of the original string). Part-of-speech (POS) tags are also taken into account to avoid splitting off suffixes and prefixes from root lemmas.

2) *Unknown Handling:* In [8], Nie, Hannan, and Jin focus on the problem of unknown word detection in Chinese. By first segmenting heuristically, then statistically analyzing unknown strings to determine their likelihood of being real words, they are able to automatically update their dictionary and achieve, at best, a success ratio of 95.66%.
3) *Maximum Entropy:* In [5], Xue approached Chinese word segmentation as a character tagging problem. By training a MaxEnt model using features involving neighboring characters, Xue tags each character in a string by its most likely position; left, right, middle, or alone. From that information, a most likely segmentation is returned, yielding 94.96% f-score.
4) *Conditional Random Fields:* In [9], Peng, Feng, and McCallum, take a similar approach, but use CRFs as opposed to MaxEnt models to tag characters as either START or NONSTART. Using POS tags and neighboring characters as features, as well as an N-best system to process and accept probable unknown words, they achieve as high as 95.7% f-score.

## IV. Disambiguation methods

Each of the following methods defines a scoring function whose aim is to assign top scores to correct segmentations, thus turning the process of segmentation into a search for the highest score.

### A. Maximum Known Matching (MKM)

To begin, we try a simple maximum matching approach—i.e. given a string $s$, get all possible segmentations of $s$ into dictionary words, then return the "longest" segmentation.

The question then becomes how to define the length of a segmentation. Should we prefer the segmentation containing the longest words? That which has the largest average word length?

We want to consider both. We first want to consider just the segmentations with the largest average word length, then take that which contains the longest word (if the longest words are equal then compare the second longest, third, etc). In other words we need a function f that fulfills the following two conditions:

1) $size(s_1) > size(s_2) \implies f(s_1) < f(s_2)$
2) $size(s_1) = size(s_2) \wedge s_1 > s_2 \implies f(s_1) > f(s_2)$

Where $s_1$ and $s_2$ are segmentations, $size$ a function that returns the number of words in a segmentation, and $s_1 > s_2$ meaning $s_1$ contains longer words than $s_2$. (Note that average word length is inversely proportional to the number of words in a segmentation).

We could of course write out the logic above, i.e. define a function that takes the segmentation of shortest length containing the largest individual words, but it may be useful later on to be able to assign a numeric score which models the same choice. Thus, we define the length score of a segmentation as follows:

$$score(s) = \sqrt[i]{\sum_{k=1}^{i} len(w_k)^2}$$

Where $len(w)$ returns the length of a word $w$, and $s$ is a segmentation into $i$ words.

For example, the score of "bra in storm" would be $\sqrt[3]{3^2 + 2^2 + 5^2} \approx 3.36$ whereas the score of "brainstorm" would be $10^2 = 100$. This scoring function indeed satisfies condition 2, and, for the values we are working with, very closely approximates condition 1. Hypothetically, it could fail on say a segmentation of length ten with words of lengths 11,1,1,1,1,1,1,1,1,1, versus a segmentation of length nine with words of lengths 2,2,2,2,2,2,2,3,3, "incorrectly" preferring the former despite the greater average word length of the latter, but segmentations like these are highly improbable and will be easily outscored by less extreme matchings.

### B. Maximum Unknown Matching (MUM)

The problem with the previous aptly named approach, however, is that it accepts no unknown words. To amend this, we expand our algorithm in the following way: given a string $s$, rather than looking at all segmentations into known words, consider all segmentations of $s$ where each division point borders at least one known word. Then return the segmentation with the highest length score.

But now we must amend our definition of length score, for as it stands it will simply return $s$ itself (or, if we exclude s, a segmentation with a very large unknown word). The previous definition of length score has no way of weighing known versus unknown words, and places high value on the average word length of a segmentation. To adjust we redefine the score as follows:

$$score(s) = \frac{\sum_{k=1}^{i} len(known_k)^2 + \sum_{k=1}^{j} len(unknown_k)}{i+j}$$

Where $s$ is a segmentation into $i$ known words and $j$ unknown words.

### C. Two grams (2GM)

These simple methods produce fairly effective results (see evaluation section), but, as discussed earlier, successful disambiguation cannot rely merely on length—some syntactic data must be incorporated. Using a database of 2-gram occurences in a corpus, we define the 2-gram score of a segmentation $s$, simply as the number of recognized 2-grams in $s$, divided by the total number of 2-grams in $s$. A segmentation of length 1, thus containing no possible 2-grams, receives a score of -1.

The final scores are then normalized to fall between 0 and 1, and a score of -1 is set by default to 0.5.

We ignore the actual occurence count of the recognized 2-grams in order to avoid over-segmentation. The string “d at a mining”, for example, would return a much higher score than the correct “data mining”, due to the frequency of the 2-gram “at a”, were occurence count taken into consideration. Without it, the latter outscores the former.

To account for numbers and ordinals, which are not included in our 2-gram datasets, as well as acronyms and contractions, we translate each unrecognized word into a set of possible words, via either a translation dictionary—the contents of which are detailed below—or, in the case of numbers and ordinals, a predefined ruleset. Out of these possibile translations, that with the highest 2-gram score is assumed to be correct.

### D. POS tagging

2GM is flawed, however, in the same way that MKM is flawed; it lacks strength in handling unknown words. 2-grams which lie outside of the database return a count of 0.

As an attempt to correct this, we approach the problem as a POS tagging problem. I.e. given a segmentation, tag each word with the appropriate POS, then assign it a score based on the probability of a given sequence of POS tags.

This approach poses two new problems:

1) Tag appropriately.
2) Score tag sequences.

And for each we pose two solutions, one using the ARK POS tagger for Twitter [14], and one using Hidden Markov Models (HMM) implemented with the MALLET toolkit [13]. Using ARK, we can tag segmenations by POS, then, using their POS n-gram data, repeat 2GM using POS tags rather than words themselves. Similarly, with a trained HMM, we can tag segmentations by POS, then score a tag sequence by taking its average edge weight in the model. This leads us to four new possible strategies:

1) Tag with ARK, assign probabilities with ARK (AA).
2) Tag with ARK, assign probabilities with HMM (AH).
3) Tag with HMM, assign probabilities with ARK (HA).
4) Tag with HMM, assign probabilities with HMM (HH).

## V. Algorithms

We now have seven scoring methods for disambiguation—MKM, MUM, 2GM, plus the four listed above. When used in overall segmentation algorithms, these scores are normalized on each new input to fall between 0 and 1, based on the highest scoring segmentation for the current input.

### A. Pipeline

These scores, plus some simple heuristics, leave us with the pipeline-based segmentation algorithm as detailed in the next column.

```
if s is known then
    return s;
end
if s is already delimited then
    return segmentByDelimiters(s);
else
    possible = allPossibleSegmentations(s);
    probable = prune(possible);
    return highestScoringSeg(probable);
end
```

In the pipeline algorithm, $highestScoringSeg()$ searches by brute force using a scoring function that is some convex combination of previously mentioned scoring functions (MUM, 2GM, AH, etc.); $prune()$ heuristically filters out unlikely segmentations; $allPossibleSegmentations()$ returns every possible segmentation of s—unless s exceeds a length threshold, in which case at least one of the k-longest words in s is made to be present in every segmentation (to limit search space size); and $segmentByDelimeters()$ segments based on punctuation and capitalization, returning, for example “Club Rio - June 19 th - 8 - 12 am” when given “ClubRio-June19th-8-12am”. Note that this will fail on confusing inputs. “LadiesoftheDMV”, for example, looks delimited but is in fact only partially, and would thus be under-segmented. Only those rule-based segmentations which meet a certain length score threshold, or are composed entirely of known words, therefore, are returned. The rest have their unknown sections fed back into the segmenter to be treated by the normal algorithm.

### B. Hill-climbing

Based on the work of Zhang et. al. [11], we also consider a greedy hill-climbing algorithm. I.e., rather than pruning down to a set of probable segmentations and then searching by brute force, start with a random segmentation, calculate the scores of all “nearby” segmentations, then climb to the one with the highest score and recurse. The algorithm terminates once no further upward steps are possible. $k$ random restarts are allowed to minimize the chance of getting stuck in local maxima. In our implementation, segmentations of a string of length $n$ string are considered as binary strings of length $n-1$, where a “1” indicates that the corresponding character in the original string is followed by a splitting point. “Nearby” segmentations are then simply defined as the set of binary strings obtained by flipping a bit in the original, i.e. either adding or removing a splitting point.

In its initial implementations, hill-climbing, although more elegant than the pipeline approach, proved to be slightly less successful (at best yielding an f-score of 76.5%, whereas, at the time, the pipeline approach peaked at 82.2%). Likely, the definition of nearby segmentations was too narrow, creating too many local maxima to avoid. Due to time constraints, however, the hill-climbing approach was dropped in order to devote more time towards improving the pipeline method.

## VI. Unknown handling

In [8], Nie et. al. define the following process for unknown handling:

1) Perform a maximum matching.
2) Gather remaining unknowns.

3) Remove unlikely candidates based on a predefined ruleset.
4) Add those which occur most frequently to the dictionary and repeat.

This method enables the segmenter to improve with repeated applications, and it is, generally, the method we use to accomodate unknown words. Rather than defining a ruleset for how words should look, however, we train a Markov chain on a list of English words. The states of the chain correspond letters of the alphabet, and transitions between states model letter sequence probabilities. The average transition probability of a given string should, then, theoretically mirror its likelihood of being a real word. The removal of unlikely candidates then comes down to choosing a threshold value. As with Nie's method, the frequency of a given unknown is also factored into its estimated probability of being a real word. Because of this, the size of the test set will directly affect the appropriate threshold value.

In addition to this algorithm, unknown handling has also been attempted via the use of spell-checking resources. Spelling errors are the root cause of many unknowns, and handling them effectively would have significant effects on performance. Several strategies were tested—treating unknown words within some edit distance threshold of known words as known words themselves; treating such words as "semi-known" words, and adjusting the length score function to handle them; including spell-check suggestions as translations for 2GM scoring—but each led to disjunctive ambiguity errors; words would be segmented with extra letters when they shouldn't have. Rules were devised as an attempt to exclude such occurrences, but still without improving results. The one method which did prove to be useful was the inclusion of common misspellings and their root words in our translation dictionary. As with numbers, ordinals, acronyms, and contractions, common spelling corrections have thus been included in translations for 2GM scoring.

## VII. Data

The dictionary used for matching includes

- $\sim$100,000 english words,
- $\sim$4,000 abbreviations,
- $\sim$200 slang words,
- $\sim$330 corporations,
- $\sim$24,000 names (both first and last), and
- $\sim$18,000 place names.

The dictionary used for translation includes

- $\sim$500 acronyms,
- $\sim$100 contractions, and
- $\sim$5,000 common misspellings.

All dictionary entries were collected from freely available online resources. For testing, $\sim$400,000 hashtags were pulled from Rovereto's Twitter 1-gram data [17]. 2-gram data contains the 1,000,000 most common English 2-grams based on the 450 million word Corpus of Contemporary American English [18]. HMMs used in conjunction with the ARK POS tagger were trained on 547 POS tagged tweets (7707 tagged words) from data used by ARK [14]. HMMs used alone were tried first on the ARK dataset, then on a manually curated supervised dataset of 1,000 hashtags using the following alternative tagset:

- NO: noun
- VE: verb
- AD: adjective
- DE: determiner
- PR: preposition
- CO: conjunction
- NU: number
- OR: ordinal
- AB: abbreviation
- FN: first name
- LN: last name
- PL: place
- MO: month
- DA: day of the week
- TE: Twitter team (e.g. "Team Iphone")
- TI: title (e.g. "mr", "mrs", "lord")
- PU: punctuation
- O: other

Dictionaries of names and places were also added as length-1 hashtags to increase word recognition by the HMM. The HH method using this training proved more effective than using ARK data, though still not up to par with the 2GM method. In the end, due to time constraints, HH work, as with work on the hill-climbing algorithm, was stymied in order to focus on maximizing 2GM results.

An attempt was also made to include a lexical normalization dictionary, taken from Han, Cook, and Baldwin's research on normalization for microblogs [12], as part of the translation dictionary, but the data proved too noisy to be useful.

## VIII. Evaluation

Performance is rated in terms of precision, recall, and f-score. For a single segmentation, precision is defined as the number of correctly segmented words divided by the total number of words *in the proposed segmentation*, and recall as the number of correctly segmented words divided by the total number of words *in the correct segmentation*. This leads to overall precision and recall scores for a list of $k$ hashtags defined simply as the average scores for all segmentations, i.e.

$$P = \frac{1}{k}\sum_{i=1}^{k} p_k$$
$$R = \frac{1}{k}\sum_{i=1}^{k} r_k$$

And f-score is calculated as the harmonic mean of the two:

$$F = \frac{2PR}{P+R}$$

[6]. Additionally, two new metrics are provided, *known-precision* and *known-recall*. These are simply precision and recall measured in terms of *known* words in a segmentation, rather than *all* words in a segmentation. Generally, known-precision scores are higher than precision scores, and known-recall lower than recall. The purpose of these measures is meant to be more pragmatic, e.g. someone requiring higher

precision at the expense of lower recall could choose to only consider the known words in a proposed segmentation.

Correct segmentations are based on a manually segmented list of 1129 hashtags. In the list, each hashtag corresponds to a set of all valid segmentations (typically just one, but in some cases alternate segmentations exist which are equally acceptable). During calculation of evaluation metrics on a proposed segmentation, scores are calculated for each possible answer and the highest results are returned.

The following table lists the results of methods that have been tested with the current dictionaries and heuristics, as tested on our manually curated answer set.

| Method | Prec | Rec | F-score | KPrec | KRec |
|---|---|---|---|---|---|
| MKM | 0.901 | 0.909 | 0.905 | 0.912 | 0.835 |
| MUM | 0.916 | 0.918 | 0.917 | 0.935 | 0.825 |
| 2GM | 0.921 | 0.924 | 0.923 | 0.942 | 0.829 |
| AH | 0.921 | 0.923 | 0.922 | 0.938 | 0.830 |

In preliminary testing AH outscored the other three POS-based methods. This makes sense, as ARK's successful POS tagger should easily trump a our HMMs in terms of tagging accuracy, whereas the edge weights of the HMM should provide comparable or better transition probabilities. As such, AH was the first (and as of yet, only) of the four to be tested with the updated heuristics and newly introduced translation scheme.

For efficiency, possible segmentations had to be pruned twice before actually taking AH score into consideration. First, heuristically. Second, by a combination of length and 2GM score. This second pruning was based on an optimal convex combination of length and 2GM scores, the values of which are depicted in figure 1. The remaining segmentations were then disambiguated by a convex combination of all three scores, length, 2-gram, and AH. Figure 2 displays the relationship between possible combinations of the three and their effects on f-score. Although the success of the AH method is largely due to the success of the pipeline system, figure 2 shows that it can perform as well as the 2GM method in disambiguation.

In figue 1, L refers to the weight assigned to the length score, and T, the weight assigned to the 2GM score, can be simply calculated as 1-L. Similarly, in figure 2, L represents the weight of the length score, A the weight of the AH score, and T is left to be calculated as 1-(L+A). Performance evaluation in figure 2 is based on the segmentation of 232 hashtags chosen such that they could not be handled by simple heuristics. Physically, they were chosen by running a heuristic segmenter on the manually curated collection and gathering those on which the machine failed. Performance in figure 1 is evaluated based on the full answer set.

Fig. 1.

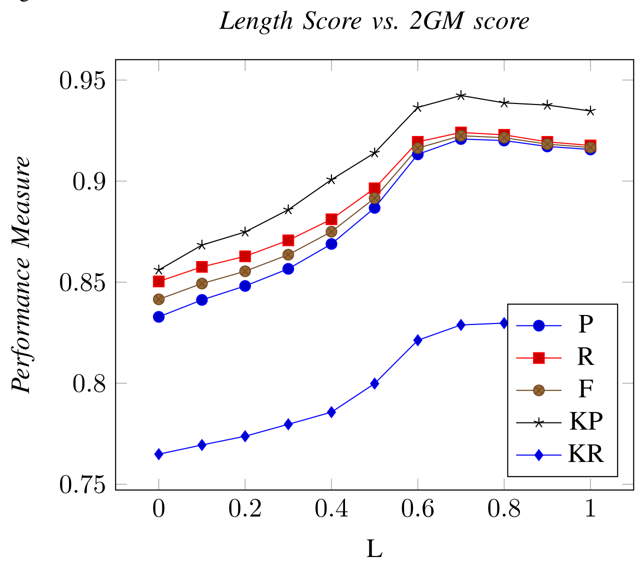
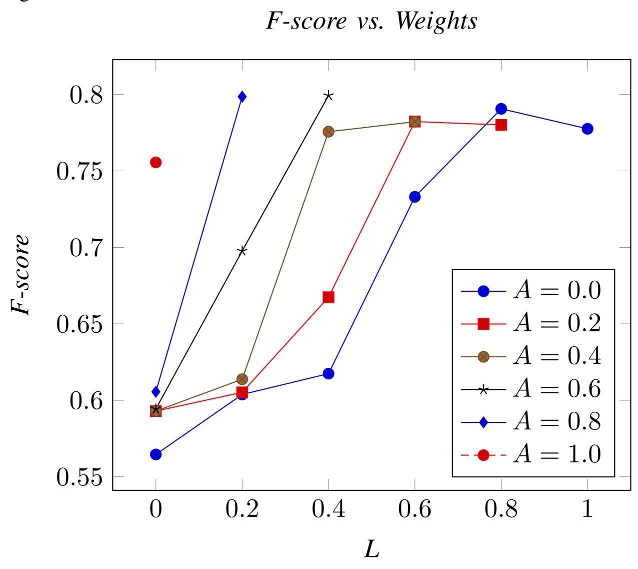


Fig. 2.



## IX. Possible Improvements

Areas to be improved upon are varied. With the right tagset and enough supervised learning data, there is still hope for the success of the HH method. Alternatively, CRFs, as shown in [9], have been used successfully as segmentation tools, and moving from HMMs to CRFs, thus allowing arbitrary feature inclusion, could be a more effective option. Larger n-gram data has not yet been tried—3-grams, 4-grams, etc.—to extend 2GM, and neither have alternative lexical normalization strategies. With different definitions of distance, the hill-climbing algorithm could also prove superior to the pipeline approach.

Much work is also left to be done in the task of unknown handling. In our research, the character HMM method has not been extensively developed. Different training sets could be explored, as well as more sophisticated methods for learning threshold values. Additionally, as far as spelling correction goes, our solution is far from perfect. A distance measure that balances spelling error correction against the creation of erroneous disjunctive ambiguities would likely improve performance greatly. Physical keyboard proximity may even be useful to consider.

## X. Conclusion

Hashtags typify a significant chunk of conversational language online. They have spread beyond Twitter and into most popular social media sites. This project provides the foundations of a tool to accurately segment hashtags into meaningful subdivisions. This will hopefully extend the ability of machines to understand the enormous amount of data produced on the Internet constantly, primarily by social and other informal media outlets.

One immediate extension of such a tool would be to use a resource like WordNet to create a graph of relationships among hashtags, allowing machines to first process a hashtag, then not only explore related topics within that tag, but other topics within related tags. Relationships could also be estimated by viewing hashtags from the "bag of words" perspective, and then applying machine learning techniques such as clustering to group them appropriately.

The same segmentation methods could also be applied to similar strings. Urls, for instance, are the first that come to mind.

## References

[1] Michael J. Paul and Mark Dredze, "You Are What You Tweet: Analyzing Twitter for Public Health", *Proceedings of the Fifth International AAAI Conference on Weblogs and Social Media*, Association for the Advancement of Artificial Intelligence, 2011.

[2] Michael D. Conover et. al., "Predicting the Political Alignment of Twitter Users", *IEEE International Conference on Privacy, Security, Risk, and Trust, and IEEE International Conference on Social Computing*, pp. 192-199, 2011.

[3] Hemant Purohit, "What kind of #conversation is Twitter? Mining #psycholinguistic cues for emergency coordination", *Computers in Human Behavior*, Vol. 29, Issue 6, pp. 2438-2447, 2013.

[4] Jonathan J. Webster and Chunyu Kit, "Tokenization as the initial phase in NLP", *Proceedings of the 14th conference on Computational linguistics*, Vol. 4, pp. 1106-1110, Association for Computational Linguistics Stroudsburg, PA, USA, 1992.

[5] Ninanwen Xue, "Chinese Word Segmentation as Character Tagging", *Computational Linguistics and Chinese Language Processing*, Vol. 8, No.1, pp. 29-48, 2003.

[6] Sebastian Spiegler and Christian Monson, "EMMA: A Novel Evaluation Metric for Morphological Analysis", *Proceedings of the 23rd International Conference on Computational Linguistics*, pp. 1029-1037, 2010.

[7] Phillipp Koehn and Kevin Knight, "Empirical methods for compound splitting", *EACL '03 Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, Vol 1, pp. 187-193, Association for Computational Linguistics Stroudsburg, PA, USA, 2003.

[8] Jian-Yun Nie, Marie-Louise Hannan, and Wanying Jin, "Unknown word detection and segmentation of chinese using statistical and heuristic knowledge", *Communications of COLIPS 5.1*, pp. 47-57, 1995.

[9] Fuchun Peng, Fangfang Feng, and Andrew McCallum, "Chinese segmentation and new word detection using conditional random fields", *COLING '04 Proceedings of the 20th international conference on Computational Linguistics*, Artical No. 562, Association for Computational Linguistics Stroudsburg, PA, USA, 2004.

[10] Hoifung Poon, Colin Cherry, and Kristina Toutanova, "Unsupervised morphological segmentation with log-linear models", *NAACL '09 Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 209-217, Assocation for Computational Linguistics, Stroudsburg, PA, USA, 2009.

[11] Yuan Zhang, Chengtao Li, Regina Barzilay, and Kareem Darwish, "Randomized Greedy Inference for Joint Segmentation, POS Tagging and Dependency Parsing", *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computations Linguistics: Human Language Technologies* pp. 42-52, Association for Computational Linguistics, Denver, CO, 2015.

[12] Bo Han, Paul Cook, and Timothy Baldwin, "Automatically constructing a normalisation dictionary for microblogs", *Proceedings of EMNLP-CoNLL 2012*, pp 421-432, Kora, 2012.

[13] Andrew Kachites McCallum, "MALLET: A Machine Learning for Language Toolkit", "http://mallet.cs.umass.edu", 2002.

[14] Olutobi Owoputi, et. al., "Improved part-of-speech tagging for online conversational text with word clusters", *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 380-391, Association for Computational Linguistics, 2013.

[15] John Goldsmith et. al., "The Linguistica Project", "http://linguistica.uchicago.edu".

[16] Alec Go, Richa Bhayani, and Lei Huang, "Twitter sentiment classification using distant supervision", *CS224N Project Report, Stanford*, Vol. 1, pp. 1-12, 2009.

[17] Ama Herdadelen, "Rovereto Twitter N-Gram Corpus: An n-gram dataset of Twitter messages with gender labels and time of posting", http://clic.cimec.unitn.it/amac/twitter_ngram/,

[18] "N-grams data", http://www.ngrams.info/

# Grouping and Testing Methods with Clustering Algorithms

Allen Burgett
Normandale College
burgetta@my.normandale.edu

***Abstract*—There are many tools, and in some cases code, available to developers on Android platforms. However, finding the most desirable code and more specifically the most efficient code can be difficult. This paper presents the framework for a technique, EffMethod (Efficient Method), by which a developer could quickly find and incorporate existing efficient code into their project. EffMethod utilizes a modified version of the k-means clustering algorithm to identify similar methods and outputs those methods for testing. After a battery of tests to determine efficiency, the developer is provided with a much smaller amount of methods to choose from.**

***Index Terms*—Program Analysis, Static Analysis, K-Means, Efficient Code, Edit Distance**

## I. Introduction

WHILE application developers have access to large stockpiles of open source code, finding the right code snippets for a project can be time consuming. At times, this could lead to in-house development of code that has already been openly shared. These in-house developments might be riddled with errors and inefficient code, leading to further patching and development, and wasting more time and money. These kinds of setbacks can be costly for any organization, but especially costly to small organizations.

To combat this problem, we gauge the feasibility of comparing open source methods from a variety of applications. Utilizing our EffMethod tool, we will allow the developer to quickly compare several open source applications. The result will give the developer a list of methods and an efficiency rating of those methods. This efficiency will rely on testing for energy and memory usage as well as run-time. This will allow the developer to import pre-made efficient methods into their programs, saving cost and time. Allowing developers to spend more time on the more proprietary aspects of their applications.

To gain access to large repositories of open source android code, we've chosen F-Droid. F-Droid allows us to search for specific types of applications and download the entire program as source code and separate from the APK package found on Google Play. This is particularly helpful as we need examples of a variety of applications as well as access to their source code and .class files.

While this work is currently being tested on the Android platform, we believe that with modification, this technique could be applied to every form of development.

The contributions of this paper are therefore as follows,

- A method for quickly gathering java bytecode on a compiled Android program.
- A technique for identifying similar methods through k-means and edit distance.
- An integrated test suite, that focuses on energy efficiency, memory usage, and runtime.

## II. Background

APIs (Application Program Interface) are used to allow methods to communicate with each other and perform certain pre-defined tasks. API packages contain a library of methods and classes from which a developer would call to perform a certain task within their method. Furthermore, work done on MAPO [10] and SAMOA [6], utilizes API calls along side other properties to determine what those methods do.

MAPO utilizes a two part process for their clustering. First, hierarchical clustering in which the API calls and their sequences are taken from individual methods and are used like letters in a word and grouped according to "family". The results of this are then cross referenced with a similar hierarchical clustering technique, which splits the names of the methods into families as well. The resulting data is a fairly accurate clustering of methods of similar type. Using these ideas, we can extrapolate that methods that make certain specific API calls, especially if those calls are in the same order or similar order, are doing similar things. By clustering those methods that have API call similarity, we can begin to categorize the methods and make predictions as to what they're actually doing.

The final product of the above step in our process will be similar to that of MAPO's. However, where they used a two part hierarchical clustering process, which has a fairly high time complexity, we will attempt to gain similar results with k-means which has a dramtically lower time complexity than MAPO's clustering process. This would allow our process to integrate other testing models to conclude efficiency, without sacrificing speed in the process. The principal question this paper generates is whether this process can actually be accomplished with a non-hierarchical clustering technique, namely k-means.

Based on the research conducted by Pathak et al while developing Eprof [8], we can conclude that some methods are just developed better. Where better is defined as less energy usage. Better can also be defined in terms of user experience. The immediate example of this is, if an application comsumes a dramatic amount of energy in comparison to its competitors, then the user is less likely to use it. A different example of efficiency, which may also effect user experience is, if a method is utilizing a large amount of memory, the user will experience delays in that program and other parts of their of device. In general, users have little patience for slow tasks. While stack memory usage has been analized for quite some time, there have been many fewer breakthroughs in analyizing heap memory. Specifically sampling heap memory during application execution. Brenschede's [2] work in this area combined with current stack analyzing techniques, will be paramount to analyzing an application or individual method's memory usuage within this project's scope. Therefore, developers have to take energy and memory usage, as well as runtime into account when writing or selecting code.

## III. Technique

The project comprises of two separate parts. First, an effective tool like MAPO [10], will need to be developed, that performs a static analysis of each selected method's API calls [1]. This static analysis will identify the function and similarity of each method in comparison to the others. Our version of MAPO's core research utilizes a variation of k-means explained later in this section, our tool would then categorize each of the selected methods, based on the number of shared API calls and their sequence within the method. The clustering algorithm will make predictions on what the methods do, based on their API calls. This will translate into separate categories that the developer can pick.

To do this applications have to be broken down into methods, with a focus on the calls invoked by the method. Java gives us an easy process by which this can be done: the commandline function 'javap -c'. This breaks down a .class file, which is a compiled .java file, into java bytecode and comments on the java bytecode. By building a script that executes commandline functions, javap can be invoked on an entire application recursively. A simple java program can then parse the comments and extract only the method names and calls. This information is then dumped into a .csv file and provides a summary of the application, its methods, and calls. All of this gathered data, for all applications, needs to be concatendated together so that pre-clustering information can be gathered from it in bulk.

Quailty assurance needs to be performed on the data given. Things like single call methods are excluded, because they're too small to be classified accurately and possibly of no use to the developer (who might find it easier to build the small method themselves, than to search for it). During this quality assurance process, two other important factors must be taken into account. First, if a method has a call that references another part of the program or as we refer to it a "local call", the method being invoked should be broken down into calls and replace the local call with the sequence of calls from the invoked method. This process is handled recursively, with checks in place that the process does not encounter an infinite loop, where a local call is invoked which invokes another local call, which in turn invokes the orignal local call. Once the local calls are replaced with sequences of real API calls, the quailty assurance program then removes any call that is invoked only once across all programs. This is done to insure that those singular calls do not disrupt the mean generation or actual clustering itself.

Our problem presents a unique structural difference to the basic k-means architecture. Traditonally k-means is used with euclidean distance, which cannot be performed on our data set. This distance metric is fundamental to k-mean's inital mean generation as well as its recentering process. To combat this distinction, we utilize a different method to generate our inital means. After we've collected the most appropriate data, we generate means using the calls we've gathered post-quailty assurance. The size of these means are based on the number of overall calls and the amount of means being generated. The number of means being generated varies based on the number of programs that are incorporated into the experiment. Within our version of k-means, our randomly generated means utilize edit distance to calculate their distance from the imported data. Our version of edit distance is based on how many removes and adds are required to make our random mean the same as the original method. Where the calls act as characters within a string and certain edits are required to make the strings the same, as shown in Figure 1.

Our current configuration of k-means has foundational similarities to k-means as well as some drastically different modifications. As it exists, our k-means takes in the randomly generated means as well as the full real-data set. During the first pass, each real-method is assigned to a mean based on its edit distance from each mean. If the lowest distance is split between more than one mean, a mean is randomly picked from that set and the method is assigned to it. Once all methods have been assigned to a cluster, the clusters are submitted to a recentering process. If any clusters are empty coming into this recentering process, then a poll is done within all of the clusters and the method with the largest overall distance from its mean is moved to the empty cluster. Next, each method inside of a cluster is polled regarding which call is contained within each index. This polling data is compared against the other methods' polling data and the call with a plurality of votes within each index is assigned as the new index for the mean. Any remaining empty indexes of the mean are filled with random calls from the post-quality assurance call set.

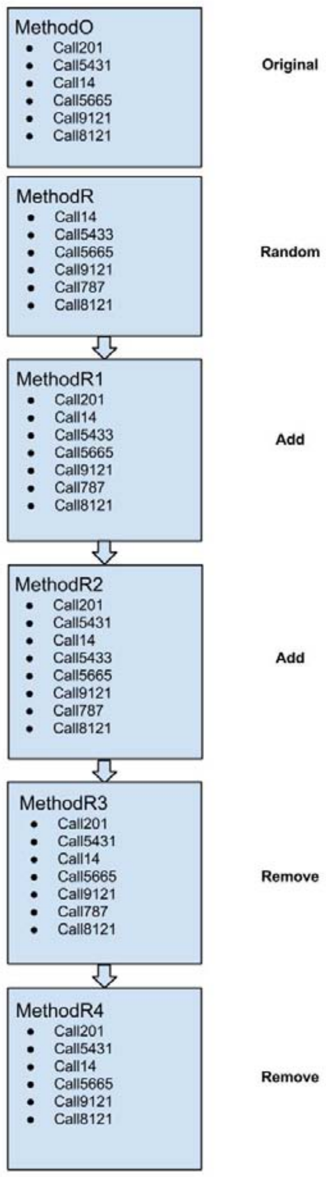


Figure 1. Our version of edit distance

The process starts again and all of the methods are compared to the new means and assigned as they were on the first iteration. The recentering process workes the same each iteration. As of now, iterations are chosen arbitrarily in an attempt to find the optimum amount.

The second part of this process identifies the most efficient methods out of those that have been clustered. Once the clustering is complete and the developer has decided which category of method to compare, a battery of tests will be executed based on: energy usage, memory usage, and method run-time. For energy usage, we will utilize a tool called Eprof, developed by Purdue and now licensed by a company called Microenergetics. Eprof breaks down applications' energy consumption by each method [8]. We will be testing memory usage by a combination of stack and heap sampling BHeapSampler [2], developed by Brenschede. Using an assortment of benchmarking tools or refactoring, we will compare the run-time of the individual methods. After testing is completed, the outputs of each method will be sorted based on probability distribution. This will give us a percentage based in comparison to the other methods. An overall rating will be assigned as an average of the method's percentages in each category. Results will be displayed to the user, divided into the four categories of overall, energy usuage, memory usage, and runtime. Those categories will be sorted most efficient to least, so that the developer can make an educated choice about the best method for their project.

## IV. Preliminary Results

Our current configuration of k-means has provided mixed results. Some methods have shown similarity within their respective cluster, while others within the same cluser have no apparent similarity with any other method. Further yet, at least 10% of these clusters have absolutely no similarity amongst member methods.

The more interesting analysis so far is, that iterations over 100 seem to have no possitive affect on overall clustering; with optimal iterations seeming to be between 40 and 80. In fact, iterations over 100 have a 50% chance of converging all of the methods into one singular cluster.

## V. Discussion of Results

Some inital explanations of our current results, have to do with how the data is being generated, refined, and processed within k-means. The smallest, but still important, flaw is the original capturing of data itself. When taking in method names, the parsing program does not strip their arguements as well. After reviewing the original data, we can see that about .5% of the original methods have overloaded counterparts. These are not distinguishable from one another and are therefore labeled as the same method, which correlates to at least .5% of our inital data being useless, but still used. The simpile soultion to this problem is to strip the arguements with the method names, which insures overloaded methods would not be rolled together.

The second problem originates from our current disregard for local calls. Where we define local calls as calls that are made to other parts of the program. This problem derives itself from the complicated task of recursively replacing local calls with a list of the external API calls of the invoked internal method. As our model currently stands, local calls are removed during the quality assurance process and disregarded overall. This ensures that the local call, which is unlikely to be found in methods from other programs, does not distort the overall data that is to be clustered. Longterm development will have to include a process for including the external calls of a local call. This process will also give us a better picture of the method and what it does.

The third problem comes from the randomness of our method placement during the cluster assginment phase of our k-means process. It is possible, with our current configuration, that methods with similar sequences of calls would be randomly placed into separate clusters. Then if they're not polled out of those clusters, they will never converge into a singular cluster. A better process needs to be designed in order to prevent this eventuality during the cluster assignment phase.

The final conceived problem with our current configuration is with the recentering process as a whole. As of now, clusters rely heavily upon the polling data collected from the methods. However, the current polling does not account for differences in size or locations of similar sequences. Two methods may be similar, for example a method with a size of nine might contain the same sequence as a method with a size of four. The method with a size of nine might invoke some common call five or six times before it invokes the similar sequence. While these two methods have commonality, they are unlikely to be clustered together, except by accident, because they would poll to change different indexes of their respective mean. A process needs to be developed to handle polling for sequences as a whole and not individual indexes.

## VI. Conclusion

Mobile applications are often run in low resource environments, making energy and memory huge concerns for both developers and users. These concerns, including run-time, are aspects of applications that will affect users and be the basis of their application download choices. This means that finding code that operates efficiently is a huge concern for developers. With EffMethod, we have laid out a framework in which a developer could easily identify and implement efficient code. This tool will save both time and money for organizations and individual programmers alike. We have also laid out an arguement for using k-means as a time-conscious alternative to MAPO's hierarchical approach to the initial clustering of this problem.

## VII. Further Considerations

The randomness of our mean generation presents its own unique problem: finding a comparable example of each unique type of method within the given set of applications. This inadequacy has led to the development of a possible alternative to API call sequence clustering. By classifying methods by the packages it invokes instead of the sequence of calls, one could cluster those methods that have an overall similarity. Many deviations of this could be tested and we offer this as an alternative branch to our current process.

## VIII. Future Work

Our long term development goals are as follows:

- Implement the energy and memory usage and run-time sorting process.
- Release EffMethod as an Eclipse plug-in.
- Generalize the tool to work on other languages. Currently this is being developed for Android, however it is not a strech to assume it will also work for standard Java programs. Which also points to the possibility of the overall idea being applied to other languages outside of the Java umbrella.
- Build-in a measure that will track user changes to code. This could be used to develope a genetic algorithm similar to GenProg [4] which has been used to patch legacy code. A modified version of GenProg could potentially modify the open source code selected by the developer, so that it fits easily within their current project.
- Build-in a measure that informs the developer of updates to the code they selected for their project.
- Add EffMethod to other major IDEs, like Visual Studio.

## IX. Related Work

MAPO, developed by Zhong et al, clusters methods based on their API calls, order of calls, and to some extent method name [10]. SAMOA, developed by Minelli et al, analyzes applications by development history, API calls, and source code [6]. The work done by Barstad et al. runs a static analysis on student's code to determine whether its "good code" or "bad code" [1]. Their tool was better at identifying "good code" than "bad code". Pathak et al developed the tool Eprof to measure energy usage of methods contained within applications [8]. Brenschede developed a tool called BHeapSampler, to analyze heap data in a java application [2]. Murphy et al dives deeper into refactoring, which could aid in our run-time tests [7]. The work provided by Hecht et al, resulted in an effective tool called Paprika, that can detect antipatterns in Android applications [3]. Similarly, Verloop et al, measured code smell in four Android applications [9]. Le Goues et al utilized genetic algorithms to patch legacy code with a 94% success rate [4]. Machiry et al developed a tool called Dynodroid, that generated input sequences and found bugs in the input sequences of 5 of the top 1000 Android applications [5].

## Acknowledgment

The author would like to thank the University of Colorado at Colorado Springs and the National Science Foundation's Research Experiences for Undergraduates program, who's Grant 1359275 funded this research.

## References

[1] V. Barstad, M. Goodwin, and T. Gjøsæter. Predicting source code quality with static analysis and machine learning. *Norsk Informatikkonferanse (NIK)*, 2014.

[2] A. Brenschede. Graph-based performance and heap memory analysis. In *Proceedings of the 15th Java Forum Stuttgart*, 2012.

[3] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. *Detecting Antipatterns in Android Apps*. PhD thesis, INRIA Lille, 2015.

[4] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[5] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[6] R. Minelli and M. Lanza. Software analytics for mobile applications–insights amp; lessons learned. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 144–153, March 2013.

[7] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *Software, IEEE*, 25(5):38–44, 2008.

[8] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.

[9] D. Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.

[10] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

# Mutant Selection Using Machine Learning Techniques

SJ Guillaume
Department of Computer Science
Allegheny College
guillaumes@allegheny.edu

***Abstract*—Mutation testing is an effective, but high cost approach to ensuring test suite adequacy. The issue of efficiency emerges. Thousands of mutants can be inserted in a program, and many tests may be run before the mutant is killed, if it is killed at all. This paper proposes that there is a way to reduce the cost of mutation testing employing data mining and machine learning algorithms to reduce the number of mutant operators run. Previous research in mutation testing proves that mutation test prioritization and reduction is possible without resulting in a significantly different mutation score. To our knowledge, no prior research techniques use machine learning models to perform mutant selection, specifically mutation operator selection.**

## I. Introduction

MUTATION testing is a method of measuring test suite adequacy. Having a high quality test suite is inherently valuable, if the test suite is reliable in catching mistakes it ensures that real issues can be detected and fixed in a program. This process is important in software development, specifically mobile software development, because with the rate at which program development and updates are demanded and required by the market, we wonder if we are asking too much of programmers. Speed and accuracy do not correlate, and catching errors before releasing a product into the market is essential. Mutation testing can make this process easier, as the purpose of mutation testing is to generate errors in code that can indicate where a program or test suite is weak or incorrect.

The drawback of creating a quality product is the time requirement. While any type of software testing is a time consuming process, mutation testing demands an unreasonably high cost in producing test cases, running time, and development effort. Faster testing is essential. Researchers have investigated ways to reduce the high cost of mutation testing for years. This paper proposes that an efficient, accurate mutation testing framework can be discovered through the application of data mining techniques and applying machine learning algorithms to determine which mutants are unnecessary and can be eliminated. Fewer generated mutants results in reduced time involved in inserting, compiling, and running the test suite.

The purpose of mutation testing is to check that a test suite detects small syntactic errors, similar to mistakes all programmers are susceptible to. The faults produced may not be syntactically similar to those produced by human error according to. [6] However, Just et al.'s work argues that the MAJOR framework does produce faults comparable to real faults. [7] Regardless, mutation testing continues as a highly regarded way to measure test suite adequacy.[16]. Mutation testing is an effective tool for testing software with known faults, and therefore is a useful tool on software for which faults are unknown [12]. An effective mutant is one which catches that there has been a change in the program which alters the result. This works because the test suite is designed to check that the program performs as expected. The test suite may have multiple tests, called test cases, for the same part of the program. If one test case compares the expected result, what the original program should produce, to what the section of the program that has been mutated, that result should differ from the expected result. When this discrepancy occurs, the mutant is killed.

For example, in the MAJOR mutation framework, the mutation operator AOR(Arithmetic operator replacement)will go into the system under test(SUT), and generate mutants. For every $+$, AOR will alter the SUT and insert a $-$ in place of the $+$. Test suite, T, will then run its test cases on SUT. For each segment of a code, there may be multiple test cases in T checking SUT. When a test case runs on SUT, and a difference is detected between what the original program produced, and what the mutated program produced, we consider the mutant k̈illed.̈ When none of the test cases in T detect the difference in SUT, the mutant is considered l̈ive.̈ There are several explanations for a l̈ivem̈utant, including equivalent, redundant, and quasi-mutants which will be discussed later. When T executes all test cases, the mutation score can be calculated. The unmutated version of the system under test(SUT) will be referred to as the original program in this paper. Equivalent mutants are those who do not produce a different result than the original program does. Thus, when T runs all of its test cases on SUT all of the test cases pass, and the mutant is not detected. Similarly, redundant mutants are those which are syntactically the same as parts of the program which are not mutated. Quasi-mutants are different in that they are syntactically incorrect, often referred to as s̈till bornb̈ecause they are not fully formed mutations, it is impossible to check these statements.

There is supporting research on the benefits of mutation selection techniques. Wright et al.'s work shows that the use of many operators is a disadvantage in mutation testing. Too many mutants may defeat the overall purpose because it may create equivalent or redundant mutants that make the testing less effective than desired.[14] The reduction technique developed in this paper suggests that by running mutation testing suites on a variety of programs, we may be able to utilize machine learning algorithms to produce a reduced set of mutation operators that are able to produce a mutation score comparable to the mutation score of the original set of mutation operators. The goal is for this method to determine set of operators that can be applied to many programs outside the training set, and have an accurate, comparable mutation score.

## II. Technique

We will be using the R-language for statistical computing, specifically the caret package for classification and regression testing. We aim to use these tools to perform Data Mining. Data mining involves going through a massive amount of data to identify patterns. [11] Once patterns are identified, we can perform machine learning to select a subset of mutation operators that will produce a mutation score comparable to the original mutation score.

The mutation frameworks we will use in this paper are MAJOR[3] and PIT[5], operators for each are found in Table I and Table II respectively. We will perform mutation testing on programs with automatically created test suites and one manually created test suite. The automatically created test suites will be generated using Codepro[1], EvoSuite[2], and Randoop[4].

Table I contains the Mutation Operators from MAJOR.

| Operator | Description |
|---|---|
| AOR | Arithmetic Operator Replacement |
| LOR | Logical Operator Replacement |
| COR | Conditional Operator Replacement |
| ROR | Relational Operator Replacement |
| SOR | Shift Operator Replacement |
| ORU | Operator Replacement Unary |
| STD | Statement Deletion Operator |
| LVR | Literal Value Replacement |

TABLE I
Table of Mutation Operators in MAJOR: adapted from $mutation - testing.org/doc/major.pdf$

Table II contains the Mutation Operators from PIT.

| Operator | Description |
|---|---|
| CBM | Conditionals Boundary Mutator |
| NCM | Negate Conditionals Mutator |
| RCM | Remove Conditionals Mutator |
| MM | Math Mutator |
| IM | Increments Mutator |
| INM | Invert Negatives Mutator |
| IC | Inline Constant Mutator |
| RVM | Return Values Mutator |
| VCM | Void Method Call Mutator |
| NVM | Non Void Method Call Mutator |
| CCM | Constructor Call Mutator |
| EMM | Experimental Member Variable Mutator |
| ESM | Experimental Switch Mutator |

TABLE II
Table of Mutation Operators in PIT: list from $pitest.org/quickstart/mutators/$

## III. Algorithms

### A. Basic Algorithms

There are many algorithms that have been developed to reduce the number of mutants generated. For our study, we use three of these techniques for baseline data to compare our machine learning algorithms against.

The first algorithm technique we use is the percentage reduction technique. This involves randomly selecting a set percentage of possible mutants to compile and execute[13].

The second algorithm technique we use is the percentage reduction by mutation operator technique. For each operator, we would reduce the number of mutants compiled and executed by a certain percentage[13]. This method ensures the set of mutants used for measuring the quality of the test suite is representative of the total set of possible mutants.

The third algorithm technique is an operator reduction technique. The idea of reducing an entire operator from testing is controversial, as that reduces all of a type of mutant that the test suite may or may not be capable of handling[16][10].

.

### B. Machine Learning Algorithms

This paper aims to perform machine learning on mutation selection using a k-fold cross validation approach and training models with each fold being a single program, thus the model would train on all but one program, and then it would test to see how accurate it is on the remaining program. Once the model has trained, we then combine the trained models to make one model for the set.

Machine learning we perform on this includes boosted class trees, greedy, lars, and clustering.

## IV. Evaluation

With the data collected, this paper aims to see if there is a subset of mutation operators that can be applied in any mutation testing, or if factors such as whether the test suite was created automatically or manually impacts the subset of mutation operators selected. Another factor we want to observe is if the subset of operators chosen from MAJOR or PIT are similar in what their functions are, also we can observe if the size of the program impacts the the subset of operators selected. All of these comparisons are important in using machine learning algorithms to select a subset of operators from the training set of programs that can create generalized rules for choosing a subset of mutation operators.

### A. Metrics

Mutation score is calculated by dividing the number of killed mutants over the number of total non-equivalent mutants. We do not consider equivalent, redundant, or quasi-mutants in the calculation because they are essentially immune to the test cases due to their similarities to the original program. The mutation score is a number between zero and one, numbers closer to one indicate a higher quality test suite.

### *B. Benchmarks*

To evaluate, ten programs were selected from the SF110 based on their size and associated test suites. These can be found in Table III. This set of programs provides a range of size. The largest program is Netweaver with 17,953 lines of code (LOC), and the smallest program is the Jni-inchi with 783 LOC.

Table III Benchmark Programs and Properties

| Program | LOC | Cyclomatic Complexity |
|---|---|---|
| Netweaver | 17953 | 2.82 |
| Inspirento | 1769 | 1.76 |
| Jsecurity | 9470 | 2.05 |
| Saxpath | 1441 | 2.10 |
| Jni-inchi | 783 | 2.05 |
| Xisemele | 1399 | 1.29 |
| Diebierse | 1539 | 1.74 |
| Lagoon | 6060 | 3.52 |
| Lavalamp | 1039 | 1.50 |
| Jnfe | 1294 | 1.38 |

TABLE III
TABLE OF BENCHMARK PROGRAMS

## V. RELATED WORK

Zhang et al. prove operator-based mutant selection is not superior to random mutant selection, where operator-based mutant selection only creates mutants on a subset of sufficient operators, and random mutant selection creates a subset of mutants from any mutation operator. The study in [16] proves that operator-based mutant selection and random mutant selection are competitive techniques, and random mutant selection is arguably superior because it chooses fewer mutants than any of the operator-based mutant selection techniques.

Research by Offutt et al. found that there are five operators, out of the 22 available in the MOTHRA mutation testing suite, that are sufficient for mutation testing, and considered better, or equal, to the full set of operators in mutation analysis [10].

Wong et al. show that random selection of mutants can effectively evaluate the quality of a test suite [13]. In this work, they learn that the use of only 10 percent of mutation operators yields a mutation score comparable to a full mutation set. The research shows that reduced mutant testing is a cost effective alternative to mutation testing.

Zhang et al. invent the FaMT technique to prioritize test cases such that ones most likely to kill the mutant are run earlier, and reduce test cases by running only a subset of all possible tests on a mutant in their paper [17]. The FaMT method reduced all executions for all mutants by about 50% but for some programs, it reduced the executions for all mutants by more than 63%. This resulted in a greatly reduced run time overhead.

Just et al. use a prioritization and reduction technique also. Their technique differs by evaluating redundancies and runtime differences of test cases to prioritize and reduce the cost of mutation analysis up to 65% [8].

In [14] removing equivalent, redundant, and quasi mutants leads to a more efficient, effective mutation analysis suite. Wright et al. propose that performing mutation analysis with so many mutants may defeat the overall purpose because it may create so many ineffective mutants. Ineffective mutants reduce accuracy and increase the cost of mutation analysis. Removing the mutants that do not make a valuable contribution to analysis proved to be an effective method for reducing mutation testing cost; there was a 56% decrease in time cost for the majority of schemas, time varied depending on the DBMS used. The mutation score increased for 75% of schemas after removing the ineffective mutants, and in 44% of cases it changed to 1, a perfect mutation score as all the mutants were killed. This is noted a statistically significant result.

Namin et al. use a mutation operator selection method on the Proteum system which generated less than 8% of mutants generated by the full set. They declare this subset is sufficient for determining test suite adequacy[12].

Size and structural coverage are important factors to consider when measuring test effectiveness. Coverage is sometimes correlated with effectiveness, but using both size and coverage gives a better prediction of effectiveness than size alone[9]. Coverage has been the most utilized way to measure test suite quality, it supports the belief that if you execute more of a program, you will be more likely to catch problematic elements. The study by Namin et al. determines that size and coverage independently influence test suite effectiveness, however, the relationship is not linear.

Offutt et. al completed an insightful study into sufficient mutant operators. The results indicate that certain operators, such as mutant operators that replace all operands with all syntactically legal operands and mutant operators that modify entire statements, do not contribute greatly to the effectiveness of mutation testing. Ultimately, this study determines that of the 22 mutant operators in Mothra, only five of these are necessary to effectively implement mutation testing[10]. These five mutation operators are ABS, AOR, LCR, ROR, and UOI.

Regression mutation testing is a technique created by Zhang et al. where they used previous tests on software to determine which mutants should be executed on subsequent tests on updated versions of the software based on kill ability and coverage of statement[18].

Research by Zhang et al. takes a different stance on mutant selection. Instead of choosing between the two types, combining mutation operator selection and random selection allows for even greater testing efficiency to be achieved at a low cost. This method applies random mutant selection on top of operator selection, to further reduce the number of mutants necessary to assert test suite adequacy. Only 5% of mutants are necessary to preserve the mutation score[15].

Our work is distinct from previous advances in mutation selection, reduction, and prioritization because the aim of this paper is to use sophisticated data mining methods to perform mutation operator selection. To our knowledge, no previous work has been done with taking a machine learning approach to mutant selection.

## VI. DISCUSSION

There were some difficulties hindering the success of this research. One is the difficulty we had with some programs and

MAJOR in producing the information needed to perform full analysis. Randoop generated test suites often did not produce accurate results, and thus is not suitable for use in comparison with other test suites.

## VII. Conclusion and Future Work

This research accomplishes the goal of determining if machine learning techniques produce better mutation subsets than algorithms that have been widely used and studied before this paper was written: random selection and operator selection. While the work done so far in this paper provides a base on which to continue research, implementing more machine learning approaches is a clear goal for future research. With the greedy algorithm as a good indicator of what could be accomplished with machine learning application to mutant selection approaches, we anticipate more advanced machine learning methods will produce further reduced sets of mutants.

## VIII. Acknowledgement

This work is supported by an NSF REU Grant.

## References

[1] Codepro analytix. https://developers.google.com/java-dev-tools/codepro/doc/?hl=en.

[2] Evosuite: Automatic test suite generation for java. http://www.evosuite.org/.

[3] The major mutation framework. http://www.mutation-testing.org/.

[4] Randoop: Automatic unit test generation for java. http://mernst.github.io/randoop/.

[5] Real world mutation testing. http://www.pitest.org/.

[6] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 189–200, Nov 2014.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

[8] R. Just, G. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 11–20, Nov 2012.

[9] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[10] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, Apr. 1996.

[11] B. Rajagopalan and R. Krovi. Benchmarking data mining algorithms. *Journal of Database Management*, 13(1):25–35, Jan 2002.

[12] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, New York, NY, USA, 2008. ACM.

[13] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[14] C. J. Wright, G. M. Kapfhammer, and P. McMinn. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 57–66. IEEE, 2014.

[15] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 92–102, Nov 2013.

[16] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM.

[17] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[18] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 331–341, New York, NY, USA, 2012. ACM.

# Modeling the Impact of Thread Configuration on Power and Performance of GPUs

Tiffany Connors
Texas State University
Email: tac115@txstate.edu

***Abstract*—The use of graphics processing units (GPUs) has become more widespread due to their high computational power. However, GPUs consume large amounts of power. Due to the associated energy costs, improving energy-efficiency has become a growing concern.**

**By evaluating the impact of thread configuration on performance and power trade-off, energy-efficient solutions can be identified. Using machine learning, the effect of applying a given thread configuration to a program can be predicted in terms of the relative change in performance and power trade-off of a GPU kernel. This enables us to establish which dynamic program features are used to predict the impact of a thread configuration on a program's performance and how these features are related to the overall effectiveness of an applied configuration. Using these program features, machine learning can be used to assist in determining the most effective thread configuration to be applied based on a given code.**

## I. Introduction

There has been an increasing demand for high performance systems for the processing of large sets of data and complex scientific computations. However, performance increase typically results in greater levels of power consumption. The consequence is increased energy costs. Through the modeling of performance and power consumption, it is possible to identify a correlation between the two and determine ways in which to make systems more energy-efficient while continuing to provide high levels of performance speedup.

Because GPUs are a low-cost option for achieving high computational power, they have become widely used in high-performance computing. Although able to provide high levels of performance speedup, GPUs can have a power consumption of up to 300W [1]. Due to the prevalence of GPUs in computational intensive work, there is a need for solutions that will decrease the associated energy costs of GPUs while continuing to provide performance speedup.

## II. Problem Definition

The main goal of this research is to improve both performance and power consumption of GPUs through the selection of optimal thread configurations by creating a predictive model using machine learning. This model can then be used to assist in the selection of thread configurations that will produce the greatest improvement in execution time while maintaining a minimal increase in power consumption. In order to achieve this, the impact of various commonly used thread configurations on power consumption and performance values will be observed. Specifically, the compiler optimizations which this work will concentrate on are thread configurations. Additionally, our work will focus on optimizing search algorithms for use on GPUs.

One problem which arises when attempting to optimize code through either the use of compiler optimizations or a change in thread configuration is that while an optimization may produce substantial improvements for one application, the same optimization may be detrimental for another application [2]. For this reason, it is important to create a model that, when provided with the code to be optimized, will predict the impact that an optimization will have on execution time and energy usage.

The framework and overview of the machine learning process proposed in this paper can be seen in Figure 1 and is explained in further detail in the following section.

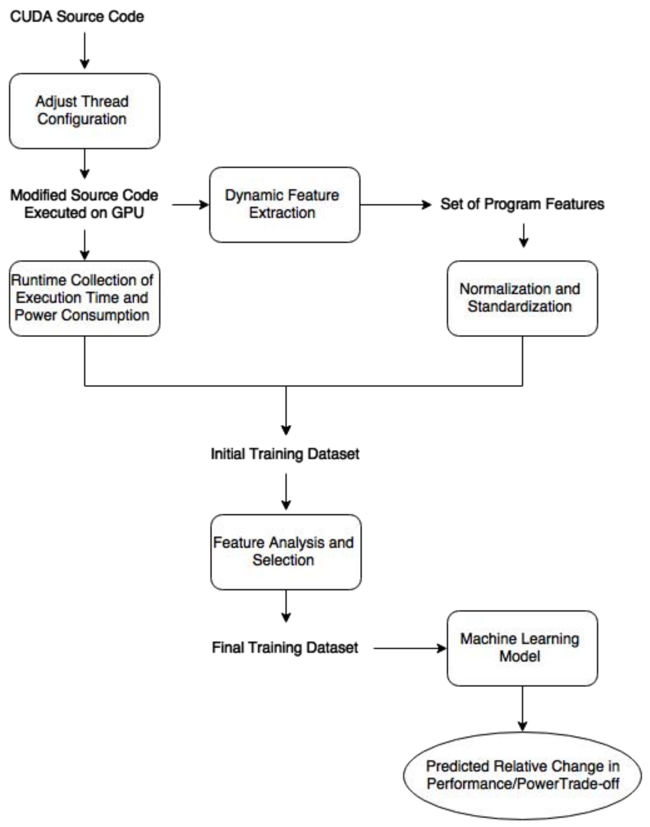

CUDA Source Code
Adjust Thread Configuration
Modified Source Code Executed on GPU
Dynamic Feature Extraction
Set of Program Features
Runtime Collection of Execution Time and Power Consumption
Normalization and Standardization
Initial Training Dataset
Feature Analysis and Selection
Final Training Dataset
Machine Learning Model
Predicted Relative Change in Performance/PowerTrade-off

Fig. 1. Overview of the machine learning process proposed in this paper.

## III. Technique

In CUDA, threads are organized into blocks, which are then organized into groupings called grids. Block size refers to the number of threads per thread block. Grid size is the number of thread blocks [3].

In this paper, the number of threads per block and number of block per grid is referred to as the thread configuration. A set of feasible thread configurations, shown below in Table III, is used and the impact of each thread configuration on performance and power is evaluated.

| | | | |
|---|---|---|---|
| 64x32 | 128x16 | 512x4 | 1024x2 |
| 64x64 | 128x32 | 512x8 | 1024x4 |
| 64x96 | 128x48 | 512x12 | 1024x6 |
| 64x128 | 128x64 | 512x20 | 1024x8 |

TABLE I
Set of thread configurations

In this work, we focus specifically on optimizing the performance and power trade-off of search algorithms for use on GPU systems. The programs included in our study as sample code are quadratic assignment problem (QAP) solvers implemented with tabu, simulated annealing, and three variations of 2opt. In addition to the varying algorithms, multiple input datasets from QAPLIB were used. The datasets which were selected are lipa20, lipa30, and tai25a. Each dataset varies in size and structure, and thus affects the program's behavior.

### A. Feature Extraction

In order to create a machine learning model that works with more than just one type of code, key features of the code which provide a good description of the program's characteristics must be determined. To do this, the source code is analyzed at runtime and a set of dynamic features is extracted. Each thread configuration from our set of commonly used configurations was applied to all of our sample source code.

The modified code was executed and the programs runtime behavior was recorded using NVIDIAs command-line GPU profiler, nvprof. It has been shown that by taking code features into account when selecting compiler optimizations, significant improvement in performance can be achieved [4]. The fifty-two features which were extracted are shown in Table II. To normalize the values collected, each feature was divided by the number of instructions executed.

### B. Collection of Execution Time and Power Consumption

In addition to dynamic features, the programs performance and energy usage were also collected. These values were obtained by using the built-in power sensor of the Tesla K20c GPU. During execution of each modified program, the average power consumption and the program's total execution time were recorded. Next, the performance and power values for each thread configuration were compared to all other thread configurations of the same program.

The increase in execution time and power consumption was calculated by dividing the original thread configuration performance and power values to those of the new target configurations. The performance and power trade-off was then computed by taking the ratio of the performance and power increases, as shown in the following equation:

$$trade-off = \frac{\triangle\, execution\ time}{\triangle\, power\ consumption}$$

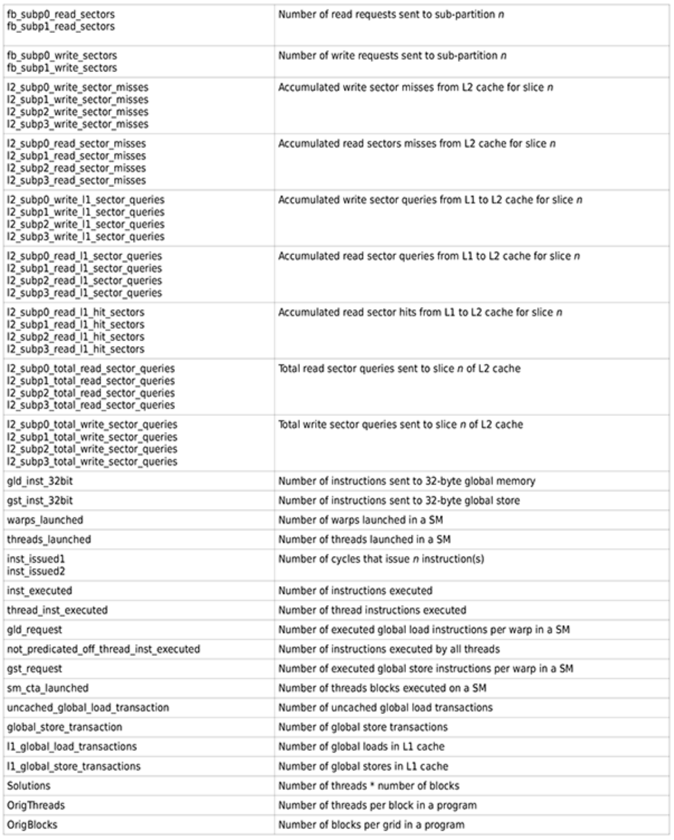
Based off this trade-off value, the row of data was assigned a class of either good or bad. These two classes describe the relative change in performance and power of the program. If a thread configuration produced a trade-off of 1.05 or greater in a program, it was classified as "good". Otherwise, the row of data was assigned to the "bad" class. The result is a file for each target thread configuration in which each row in the dataset contains a set of features, the corresponding original thread configuration, and a class label.

Next, all numeric values of the dataset were standardized by using the following formula to calculate the z-score:

$$z = \frac{X - \mu}{\sigma}$$

### C. Feature Selection

To reduce our set of features down to only those with the highest predicitive power, feature selection was performed. First, any features which had zero variance were removed. Machine learning models were then built and the variable importance of each model was calculated, Figure 6. Next, features were analyzed using correlation matrices, see in



| | |
|---|---|
| fb_subp0_read_sectors<br>fb_subp1_read_sectors | Number of read requests sent to sub-partition *n* |
| fb_subp0_write_sectors<br>fb_subp1_write_sectors | Number of write requests sent to sub-partition *n* |
| l2_subp0_write_sector_misses<br>l2_subp1_write_sector_misses<br>l2_subp2_write_sector_misses<br>l2_subp3_write_sector_misses | Accumulated write sector misses from L2 cache for slice *n* |
| l2_subp0_read_sector_misses<br>l2_subp1_read_sector_misses<br>l2_subp2_read_sector_misses<br>l2_subp3_read_sector_misses | Accumulated read sectors misses from L2 cache for slice *n* |
| l2_subp0_write_l1_sector_queries<br>l2_subp1_write_l1_sector_queries<br>l2_subp2_write_l1_sector_queries<br>l2_subp3_write_l1_sector_queries | Accumulated write sector queries from L1 to L2 cache for slice *n* |
| l2_subp0_read_l1_sector_queries<br>l2_subp1_read_l1_sector_queries<br>l2_subp2_read_l1_sector_queries<br>l2_subp3_read_l1_sector_queries | Accumulated read sector queries from L1 to L2 cache for slice *n* |
| l2_subp0_read_l1_hit_sectors<br>l2_subp1_read_l1_hit_sectors<br>l2_subp2_read_l1_hit_sectors<br>l2_subp3_read_l1_hit_sectors | Accumulated read sector hits from L1 to L2 cache for slice *n* |
| l2_subp0_total_read_sector_queries<br>l2_subp1_total_read_sector_queries<br>l2_subp2_total_read_sector_queries<br>l2_subp3_total_read_sector_queries | Total read sector queries sent to slice *n* of L2 cache |
| l2_subp0_total_write_sector_queries<br>l2_subp1_total_write_sector_queries<br>l2_subp2_total_write_sector_queries<br>l2_subp3_total_write_sector_queries | Total write sector queries sent to slice *n* of L2 cache |
| gld_inst_32bit | Number of instructions sent to 32-byte global memory |
| gst_inst_32bit | Number of instructions sent to 32-byte global store |
| warps_launched | Number of warps launched in a SM |
| threads_launched | Number of threads launched in a SM |
| inst_issued1<br>inst_issued2 | Number of cycles that issue *n* instruction(s) |
| inst_executed | Number of instructions executed |
| thread_inst_executed | Number of thread instructions executed |
| gld_request | Number of executed global load instructions per warp in a SM |
| not_predicated_off_thread_inst_executed | Number of instructions executed by all threads |
| gst_request | Number of executed global store instructions per warp in a SM |
| sm_cta_launched | Number of threads blocks executed on a SM |
| uncached_global_load_transaction | Number of uncached global load transactions |
| global_store_transaction | Number of global store transactions |
| l1_global_load_transactions | Number of global loads in L1 cache |
| l1_global_store_transactions | Number of global stores in L1 cache |
| Solutions | Number of threads * number of blocks |
| OrigThreads | Number of threads per block in a program |
| OrigBlocks | Number of blocks per grid in a program |

TABLE II
Initial set of fifty-two program features.

Figure 2, and those which were highly correlated to one another were identified.

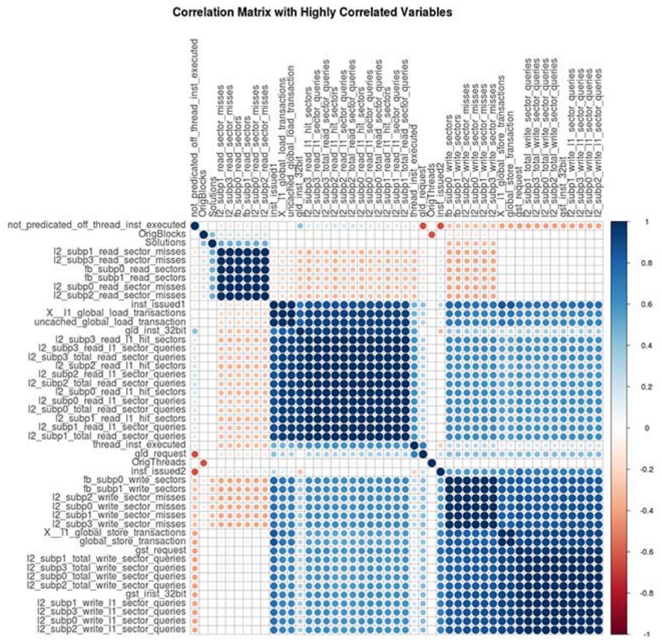


Fig. 2. Correlation matrix depicting the level of correlation between each of the 52 features.

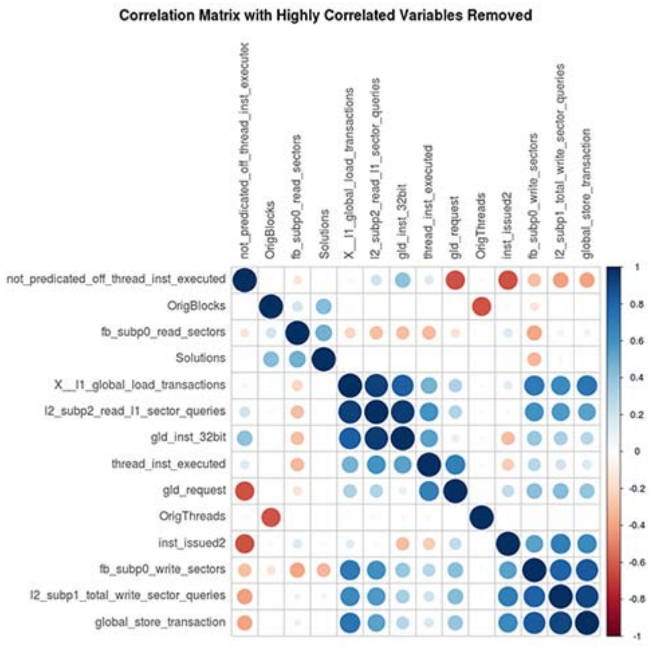


Fig. 3. Correlation matrix which illustrates the remaining 14 features once highly correlated variables were removed from the dataset.

Any features which had a correlation of 95% or higher were removed from the feature set and a new correlation matrix, Figure 3, was produced. Principle component analysis (PCA) was performed on the remaining features in order to evaluate the active variables' degree of correlation. Additionally, the features were compared with each of the models' variable importance values to determine if the features identified as significant factors in the models remained in the new set of features.

## D. Machine Learning Methods

Nine different machine learning algorithms, listed in Table 6, were used in this work. The algorithms selected were those which supported binary classification and are available in R's Caret package. The purpose of using varied machine learning methods was to determine if the same features were significant factors across all models, as well as identify which machine learning algorithms worked best with our data. Each target thread configuration was treated independently and separate models were built, trained, and tested for each of these configurations. The algorithm which performed the best was then selected to be used for building the final predicitive model.

| | | |
|---|---|---|
| boosted C5.0 | bagged CART | ctree |
| random forest | naive bayes | smvRadial |
| flexible discriminant analysis | neural network | k-nearest neighbors |

TABLE III
THE NINE MACHINE LEARNING ALGORITHMS USED IN THIS WORK.

## IV. Results

For each target thread configuration file, the data was randomly partition and 60% of the data was used for training and the remaining 40% was reserved for testing. Additionally, repeated k-fold cross-validation was used with k=10 and repeats set to 5. This partitioned data was then used in each of the nine models.

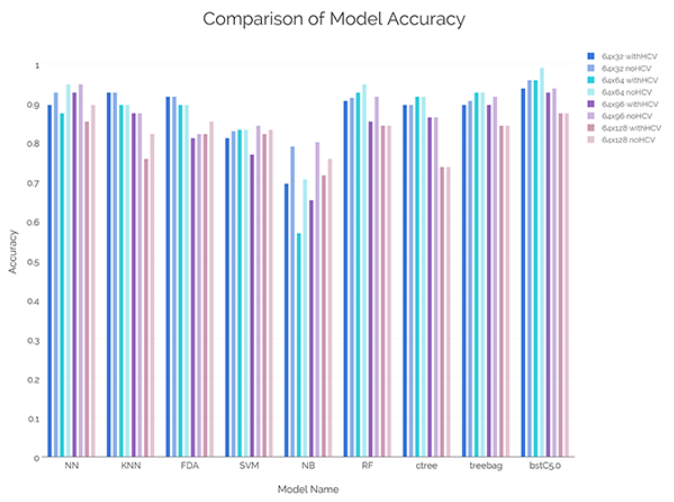


Fig. 4. A graph depicting the change in model accuracy before and after feature selection has been performed.

The machine learning models performed quite well, with most models having an accuracy in the high eighties to mid-nineties. The boosted C5.0 tree algorithm was selected as the final model for predicting the impact of modifying the thread configuration of a program. Models built using this algorithm achieved the highest levels of accuracy among the nine machine learning algorithms used, with an average accuracy rate of 94.9%.

Through feature selection, the initial set of features was reduced from fifty-two down to the fourteen features identified in Figure 3. When principle component analysis is performed on the remaining fourteen factors and the variables factor map is generated, Figure 5, we can see that OrigThreads, OrigBlocks, Solutions, and fb_subp0_read_sectors are highly correlated to each other. not_predicted_off_thread_inst_executed is not strongly correlated to anything else and has negative correlation to inst_issued2. In addition to having a correlation of less than 95% to one another, these fourteen features had also been identified as significant factors in all nine of the machine learning models, as seen in Figure 6. These results were the same across all target thread configuration files and models.

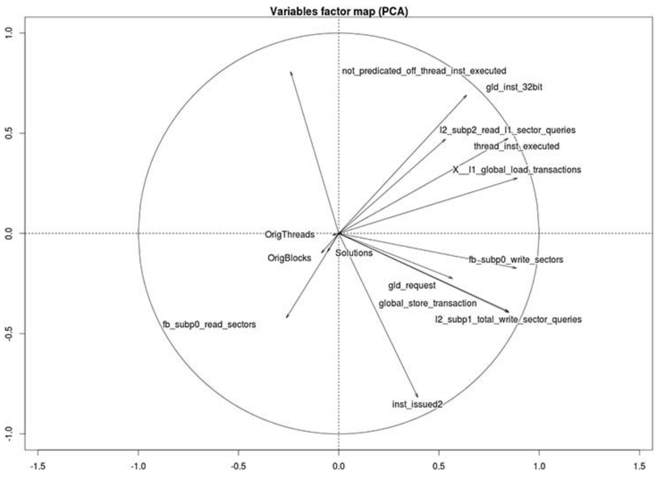


Fig. 5. In the variable's graph, the angle between two arrows depicts the correlation of the two variables. If the arrows are at a 90 degree angle, there is no correlation. Two arrows that are on a near linear line are negatively correlated.

As seen in Figure 4, limiting the feature set to only those features which were not highly correlated improved the accuracy of the models. The feature most frequently identified as the factor with the greatest significance was original threads. It appears that if the original thread count is large, then reducing the thread count to a lower number will be more beneficial.

The two thread configurations which had the greatest number of good instances were 64x128 and 128x16. Changing the thread configuration to 64x128 was good 57.5% of the time, while changing the thread configuration to 128x16 resulted in desirable tradeoff 89.58% of the time. The visualization of these two trees can be seen in Figure 7. Changing the thread configuration to 1024x8 was always bad, therefore we were unable to use C5.0 for this thread configuration since there was no variance in the response label.

The C5.0 tree model predicted that changing the thread configuration to 64x128 would result in improved performance/power trade-off when the program had a high original thread count in conjunction with a large number of thread instructions executed. If a lower amount of thread instructions are executed, then improvement will be seen if this is coupled with a smaller number of write requests sent to sub-partition 0.

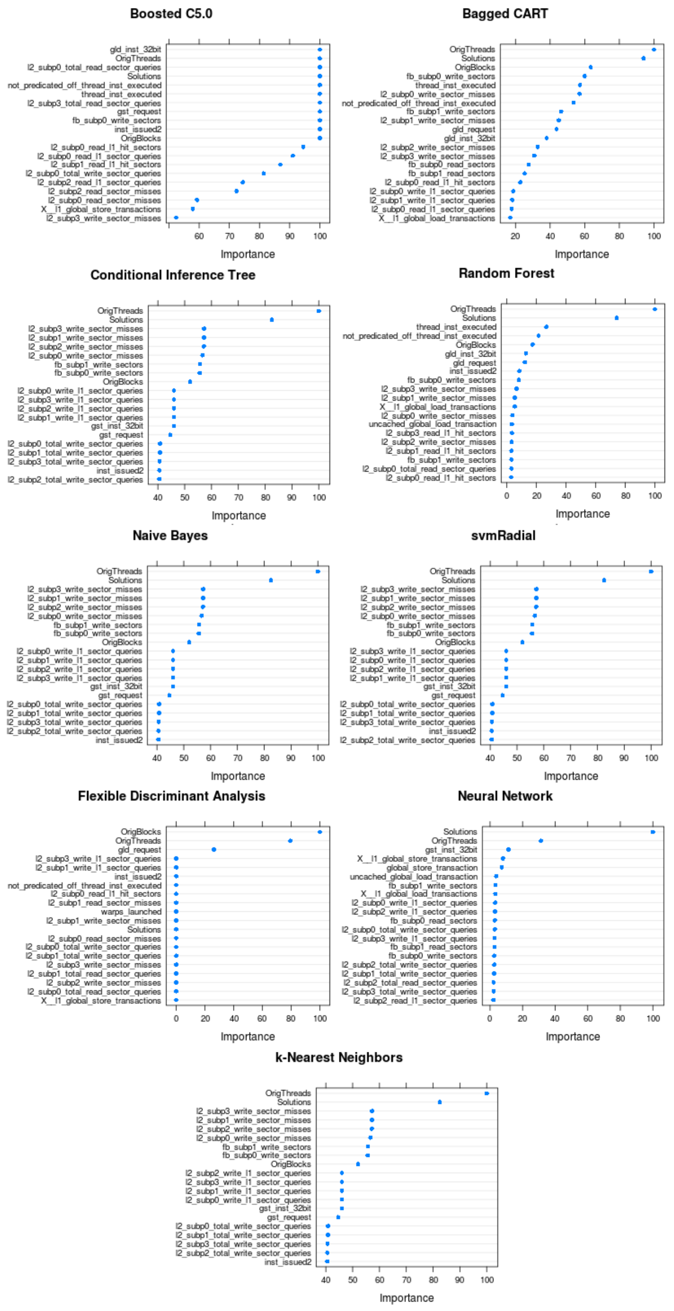


Fig. 6. Variable importance of the top 20 predicting factors for each of the nine machine learning models.

When the thread configuration is modified to 128x16, the C5.0 tree predicted that the change would have the highest probability of being bad if the original thread count is small, the number of instructions sent to 32-bit global memory is low, and a smaller number of read requests are sent to sub-partition 0. Otherwise, the change will likely result in improved performance/power trade-off.

## V. Related Work

Ukidave et al. studied the effects of optimizations and algorithm design on power/performance trade-offs of GPUs,

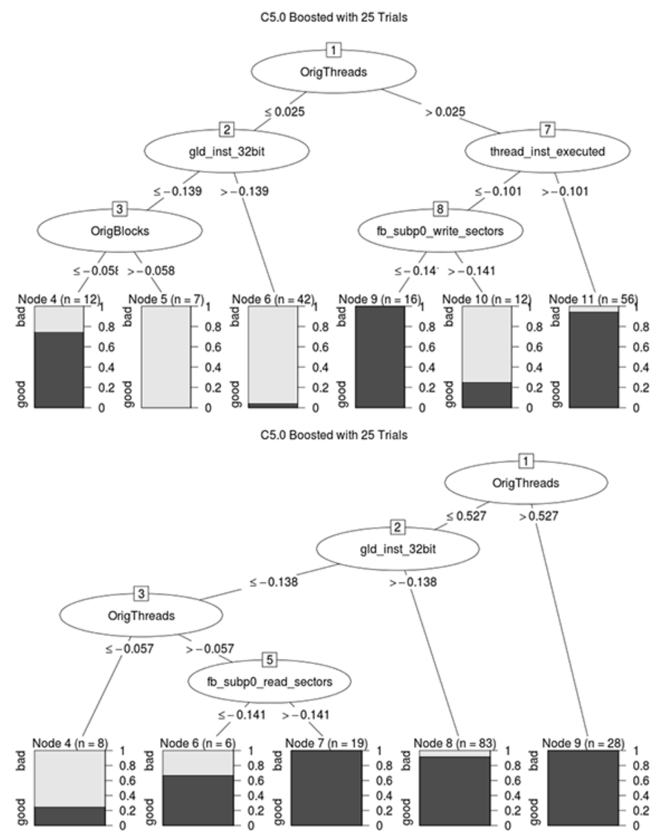


Fig. 7. Visualization of the 64x128 (top) and 128x16 (bottom) C5.0 trees.

APUs, and SoCs. The compiler optimizations investigated were loop unrolling, data transformation, and local memory optimizations [6]. In contrast, this research concentrates on thread configuration. Additionally, the paper by Ukidave et al. did not attempt to create a machine learning model for the implementation of these optimizations.

Agakov et al. used iterative search to select good compiler optimizations for increasing performance. Program features were taken into account and Agakov et al. identified thirty-six loop-level features that described a program's characteristics [7]. In the work presented in this paper, we did not limit ourselves to loop-level features. Another difference is that our work uses CUDA programs, while their work focused on C programs running on CPU systems.

There has been much work exploring the use of machine learning for selecting compiler optimizations. Research performed by Magni et al. aimed at building a machine learning model for automatic optimization for GPUs by using thread-coarsening [8]. Similarly, a technique has been proposed by Cavazos et al. which uses machine learning to automatically select the best optimizations to increase GPU performance [9]. Liang et al. has proposed a joint register and thread structure optimization framework that achieves considerable increase in speedup, showing that the impact of thread structure and register allocation on performance are related [10]. However, unlike the research proposed in this paper, the work mentioned above did not take power consumption into account.

## VI. Future Work

This work currently includes only two classes used in the machine learning models. It is our goal to expand these models to include up to eight different classes in order to give more detailed results of the relative change in performance/power trade-off.

We intend to expand this work by investigating performance and power consumption of stencil code on the GPU. By using a stencil code generator, a large number of programs can be produced in a short amount of time. Dynamic features, execution times, and power consumption will be collected using the same methods already outlined in this paper.

Including stencil code will enable us to create a larger training dataset. This will also allow us to see if the same dynamic features important for predicting the impact of a thread configuration on QAP algorithms are significant factors for stencil code.

In addition to using machine learning to predict the relative change in performance and power trade-off, we also intend to create a model that will predict which thread configuration should be applied in order to obtain optimal trade-off. Provided with a set of program features, the model will output the thread configuration predicted to result in the best performance and power trade-off.

## VII. Conclusion

Due to their high computational power, GPUs have become an increasingly popular choice for high performance computing. While GPUs provide exceptional performance speedup, they also consume a large amount of power, resulting in higher energy costs. In order to make these systems more energy-efficient, this paper proposes using machine learning to select the thread configuration which will provide greater speedup while maintaining minimal power consumption.

Using dynamic feature extraction on a given code and selecting the features which are most closely related to thread configuration, a machine learning model can accurately predict the impact that a new thread configuration will have on the performance/power trade-off of the program. In turn, this can be used to assist programers in the proper selection of thread configurations which will give increased performance while maintaining minimal increase in power consumption.

## VIII. Acknowledgment

This material is based upon work supported by the National Science Foundation Research Experiences for Undergraduates program under Grant No. 1359275.

## References

[1] S. Collange, D. Defour, and A. Tisserand, "Power consumption of gpus from a software perspective," in *Proceedings of the 9th International Conference on Computational Science: Part I*, ser. ICCS '09, Baton Rouge, LA, 2009, pp. 914–923.

[2] E. Granston and A. Holler, “Automatic recommendation of compiler options,” in *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*, Austin, TX, 2001.

[3] *Cuda c programming guide, version 7.0*, NVIDIA. [Online]. Available: http://docs.nvidia.com/cuda/index.html.

[4] J. Cavazos and M. O’Boyle, “Method-specific dynamic compilation using logistic regression,” in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ser. OOPSLA ’06, Portland, OR, 2006, pp. 229–240.

[5] R. Kohavi, “Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid,” in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, OR, 1996, pp. 202–207.

[6] Y. Ukidave, A. K. Ziabari, P. Mistry, G. Schirner, and D. Kaeli, “Analyzing power efficiency of optimization techniques and algorithm design methods for applications on heterogeneous platforms,” *The International Journal of High Performance Computing Applications*, vol. 28, no. 3, pp. 319–334, 2014.

[7] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G.Fursin, M. O’Boyle, J. Thomson, M. Toussaint, and C. Williams, “Using machine learning to focus iterative optimization,” in *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization*, ser. CGO ’06, New York, NY, 2006, pp. 295–305.

[8] A. Magni, C. Dubach, and M. O’Boyle, “Automatic optimization of thread-coarsening for graphics processors,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14, Edmonton, AB, Canada, 2014, pp. 455–466.

[9] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO ’07, San Jose, CA, 2007, pp. 185–197.

[10] Y. Liang, Z. Cui, K. Rupnow, and D. Chen, “Register and thread structure optimization for gpus,” in *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, Yokohama, Japan, 2013, pp. 461–466.

# Static Performance Prediction of Compiler Optimizations

Michael Dennis

***Abstract*—Tunning the interaction of various compiler optimizations to be optimal for a specific platform is a daunting task with complicated interacting considerations. The problem is exacerbated by the fact that what is optimal on one architecture likely performs terribly on another. Since manually tuning a compiler for each new architecture is infeasible, this project will use machine learning to create a model specific to each architecture that will accurately predict the runtime performance of code based on static features to better inform compiler optimizations. Once the predictive model is created, it can be quickly used to determine near optimal settings for compiler tunning parameters specific to each block of code. To give this research direction, we will begin our work with stencil code, moving on to dynamic programing after preliminary results. These classes of algorithms provides more structure to the problem by focusing on programs whose performance is minimally input dependent allowing our methods to achieve more accurate results.**

## I. Introduction

COMPILER optimization is a complex subject. The quality of any individual optimization often depends on fine details of the machine that vary widely from platform to platform. What could be a powerful optimization on one machine could slow down performance on another and it is not uncommon for a compiler to try a very complex series of optimizations only to find that performance has not changed or even decreased. Tuning parameters of compilers for the wide variety of systems by hand is a problem beyond the scale of practicality.

To handle the complexity of choosing the best configuration of optimizations many have used iterative compilation assisted by machine learning techniques to tune parameters to a specific block of code. Running an iterative machine learning algorithm with recompilation vastly slows down the process and ultimately results in less time for other optimizations to be fine tuned. Additionally, an iterative technique often finds local mixima, which offer less than optimal performance gains. More significantly, iterative compilation techniques fail to utilize the structure of the optimization itself, learning how the optimization effects the performance of this program but not programs in general. This project will move the iterative and expensive machine learning to compiler-tuning time by creating a predictive model of the performance of applications on the host architecture to use in later optimizations. The hope of this model is to learn in detail about how the machine performs and to use this model to predict how optimizations will effect the performance of yet unseen programs. Once this model is created for a particular architecture and reused for every subsequent compilation resulting in more efficient compilation and a more informed search for optimizations. This paper will focus on the creation of the predictive model on the host architecture.

To give this work direction we will compare models created from strictly stencil codes to models created with a larger set of codes that allows for more flexible iteration patterns and optional expressions at each level. These two sets will use the same sets of features, and will be compared by the predictive capabilities of their models. Since the second type of code is a superset of stencil code, it will necessarily show at least as much error as the stencil code predictive model. Seeing the extent of the decrease in accuracy as the code complexity increases will demonstrate the challenges of performance prediction from static features as well as the necessity for more quality code features to be used to counteract the decrease in performance.

## II. Technique

### A. Code Structure

We have limited the scope of our work to two specific code structures for purpose of comparison. The first code structure is simply stencil codes, a very well studied, well structured type of program. The other code structure is a slight generalization of stencil codes, adding the ability for each level to have memory accesses and calculations and allowing loops to iterate in more complex configurations. This code structure allows any loop structure that can be mode from a constant step, a lower bound starting at the outer loop's index variable, and a constant number of iterations. Since this second type of code structure is significantly more complicated, it will be possible to observe how predictive power decreases as complexity increases.

### B. Feature Selection

In order to create the predictive model it is important to select quality static features. In our work we have found that it is best to use features that are not calculated, rather observed. Since metrics such as working set size and memory reuse can be calculated from simple, observable static features such as loop factors, memory accesses, and distribution statistics describing the location of memory accesses, giving the model the strictly observable static features gives the feature vector at least as much discriminatory power as if we used to calculated features, but it does not bias the model to use known forms of performance prediction and it limits the chances of multiple features being closely related by their calculations.

For raw features we have chosen the following: Number of Floating Point Adds, Subtracts, Multiplications and divisions, Number of memory accesses, the minimum and maximum and average distance between consecutive memory accesses, Number of iterations of the loop and how much each iteration changes the centroid of higher loop levels. Each of these features are repeated for each level of the loop. For the purposes of these initial experiments we have limited ourselves to 5 nested loops with the understanding that this can later be extended to more loops for a production model. However, though the model in this form is limited to structures with a finite number of loops, this limitation will not decrease the applicability of the model as any optimization whose feature changes are within this constant number of loops will be fully described.

## *C. Feature Validation*

From feature selection we proceeded to determine the descriptive power of our features. Since we are grouping programs by a finite set of features, many programs are going to share the same feature set, and thus one would naturally expect there to be more variance in measurements of different programs with the same features than there would be in different measurements of the same program. To calculate this difference we selected 100 vectors at random and generated 30 stencils for each vector. These stencils where tested each 30 times shuffled with the tests for other stencils with the same features to amortize performance inconsistencies on the host platform. We use this data to calculate a mean percent error as well as a variance of sample means of percent errors for each vector. By the central limit theorem we can then treat each of these measurements for each vectors as a normal distribution and compute the probability density function of percent error as a mixture of these normal distributions. Again sampling from this set we can obtain a mean percent error across all vectors and a confidence interval for this percent error. Following this same methodology we can compute the average error in measurement for a specific code structure and feature set.

We followed the above process on both generated stencil codes and generated codes following the previously mentioned broader structure. The results suggest that, with this feature vector, stencil codes could be predicted with an average error of 5.529277% (sd 0.247) and the more general structure could be with an average error of 12.09777% (sd 0.460). These are both compared to an average measurement error of 2.78825 % (sd 0.416). The fact that stencil codes are predicted better than the more broad category speaks to greater variance in running times of the broader class do to increased complexity. While this feature vector seems adequate to predict the performance of stencil codes it is not well suited for the more complex category. To counteract this decrease in accuracy it is necessary to increase the predictive power of the feature vector in order to add features that can discriminate between the best and worst performers within a specified vector.

## *D. Model Training*

Using the identified features of the code and knowing they give a good basis for performance prediction we, collected data uniformly randomly inside the space of codes with this structure (subject to time and space limitations). A code generator was created that was able to generate code with specific features and test it on the host system. Each such test will represent a data point, giving a correlation between this list of features and performance. After collecting this data for some time a predictive model was generated using Multivariate Adaptive Regression Splines(MARS) [2]. Choosing MARS as our regression algorithm is important as it allows for a piecewise predictive model, which is representative of computing performance where sudden discontinuities can occur when predictive parameters hit certain values such as cache size or bandwidth of some hardware component. Additionally, MARS produces a regression model as a mathematical function that is continuous and differentiable which will be important for later mathematical analysis. The predictive model that has been generated is ready to be integrated into the compiler as a description of performance for the host architecture and will not change in future compilations.

## *E. Model Utilization*

During the compilation process, the compiler will consider the optimizations that it can make, and generate equations representing the space of possible optimizations and how they effect the aspects of the code used by our static predictive model. An equation will be given to the predictive model for each variable that it relies upon. These equations will contain variables that are meaningful tuning parameters to the compiler, but to the model they are simply values that are used to calculate important features of the code. For example, if a compiler does standard blocking of matrix code, it could determine an equation for working set size based on the blocking coefficients. These coefficients will be only variables to the predictive model, but, to the compiler, an assignment of these variables represents a combination of optimizations. The job of the performance model at runtime is to find the assignment to these tuning parameters which maximizes performance on the host system. By using MARS we have created a predictive model that is differentiable, and by restricting the compiler to send polynomial equations, which are both typical and expressive, we have a set of equations to calculate predictive parameters that are also differentiable. Since both the predictive model and the constraint equations are differentiable we can use Lagrange Multipliers to find a finite set of possible local maxima to test, making the process of finding the best assignment of the tunning parameters at compile time faster and more accurate than methods of iterative improvement. Importantly, this method will allow us to make ensure that we make the best choice of parameters that our model can predict and will avoid being stuck at a local maximum.

## III. Related Work

In the field of static performance prediction much has been attempted to mixed results. There have long been techniques for predicting cache miss rates [7], however, since these results were achieved, much has changed in computer architectures, and even with an accurate prediction of miss rates, the results cannot necessarily be generalized to an accurate prediction of performance. Others have tried to predict the performance directly, though their methods were only partly static, requiring expensive profiling at compile-time and were only able to achieve average error of around 20% [1]. Yet others have narrowed their focus to scientific applications and were able to create models that had good performance prediction across multiple architectures [4]. However, their methods were still dependent on dynamic analysis. A large variety of machine learning algorithms have been used at compile time with different heuristics to tune compiler optimizations with a 3 fold improvement from unoptimized code, but requiring 10 minute iterative compilation process [6]. This compile time overhead has been avoided by others, instead using machine learning off line to tune the search heuristics of the compiler, with positive, but less significant results, only achieving a 2 fold speedup over unoptimized code on standard benchmarks [5].

## IV. Future Work

Continued improvements will come in two forms. First, the current models could be made more precise. The error of our model is larger than the average error from the mean measured from samples within each of our vectors. This observation leads us to conclude that our model does not predict as well as our feature vector will allow, and thus there is room for improvement in the creation of the model. To realize this potential improvement is a matter of improving our model creation process. By changing machine learning parameters, changing code generation patterns to be more representative of real code, and choosing to get more data from specifically misclassified sections of the model, we can work to improve the accuracy of the model for a specified feature set. Fine tunning the initial suit of tests is essential to creating an accurate model. If irregularities are not found initially, they could be missed by later steps as well. To insure an initially good set of tests we can employ the standard Roof Line model [3] to generate initial areas of interest for the first round of data collection. This model is already used to great effect in performance analysis, and though it is not meant to guarantee the degree of precision we desire, it will provide a baseline that is close and will only improve after the tests have been performed.

To achieve improvements past the mean measured standard error of a specific feature set would require a change to the features them selfs. This option has the possibility of greatly improving the performance of the model, giving it the ability to distinguish different utilization patterns of the host machine that were previously indistinguishable. However, it is important that the features that were added are both useful, they provide new information, and raw, directly observable from the source code.

The second form of improvement is to begin to utilize the created models for actual optimization prediction. After the previously mentioned improvements produce a model capable of accurately predicting performance of a specific type of code we can imagine an optimization that transforms the code by keeps it within the set if codes capable of being predicted by the performance model. Given a mathematical description of the parameterized optimizations in the form of equations describing how changing these parameters changes the features of the code, it is possible to use the trained mathematical model to find the predicted optimum choice of parameters without ever running the program. If we accept as given that the model is accurately predicting changes in performance, such a system has the possibility of giving better and more reliable performance gains than current methods. These methods could be further improved by supporting more optimizations or supporting compositions and permutations of supported optimizations.

## V. Conclusion

Choosing the best or even choosing a good set of optimizations for an arbitrary program on an arbitrary architecture is a problem so difficult that standard manual analysis techniques prove impractical. Common methods for dealing with this complexity, such as iterative compilation, relearn with each new program certain key facts about the structure of not only the optimization but how the machine performs. It is our hope that creating a mathematical performance model at compiler-tune time will avoid this relearning process, creating a single reliable model that can be trained when time is less important.

It is clear from our initial work that creating mathematical models for significantly complex sets of codes will require larger sets of quality features. However, it's also clear that for well structured code and with enough features, the creation of mathematical models with moderate accuracy is possible. Continuing to improve the process of creating the prediction models, and increasing their accuracy has promise to greatly improve compiler optimizations. Utilizing the completeness of the mathematical description of the machine, it may be possible to provide greater confidence in the improvement of the chosen optimizations. Work will continues to expand model creation to be accurate for more types of codes, adding new features when necessary to counteract increases in complexity. As these results are improved, this work will allow compilers to utilize the structure of the machine and optimizations to perform more effective searches that may lead to quicker compilation of more efficient programs.

## References

[1] Calin Cascaval, Luiz De Rose, David A. Padua, and Daniel A. Reed. Compile-time based performance prediction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 365–379, London, UK, UK, 2000. Springer-Verlag.

[2] Jerome H Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67, 1991.

[3] YuJung Lo, Samuel Williams, Brian Van Straalen, TerryJ. Ligocki, MatthewJ. Cordery, NicholasJ. Wright, MaryW. Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 129–148. Springer International Publishing, 2015.

[4] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[5] Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In *Software Automatic Tuning*, pages 335–351. Springer, 2010.

[6] Keith Seymour, Haihang You, and Jack Dongarra. A comparison of search heuristics for empirical code optimization. In *Cluster Computing, 2008 IEEE International Conference on*, pages 421–429. IEEE, 2008.

[7] Y. Zhong, S.G. Dropsho, Xipeng Shen, A. Studer, and Chen Ding. Miss rate prediction across program inputs and cache configurations. *Computers, IEEE Transactions on*, 56(3):328–343, March 2007.

## Author Index