A Framework to Efficiently Predict Variable Rankings by Relative Importance

Patrick Pape
Mississippi State University
PO Box 9627
Mississippi State, MS 39762
(662) 325-2080
pape@dasi.msstate.edu

Christopher Ivancic
Stephen F. Austin State University
P.O. Box 13063, SFA Station
Nacogdoches, TX 75962
(936) 468-1461
ivanciccp@sfasu.edu

John A. Hamilton, Jr.
Mississippi State University
PO Box 9627
Mississippi State, MS 39762
(662) 325-3570
hamilton@research.msstate.edu

ABSTRACT
This paper describes a work-in-progress software framework for identifying the highest priority variables in a software sample, based on a relative importance metric. The framework utilizes a combination of static and dynamic analysis to gather features pertaining to each variable in the relevant functions of a software sample and then makes a prediction as to the priority ranking of each variable in that sample. This ranking is based on the likelihood of the variable to cause a fail state in the software when dealing with a faulty or unexpected data value and the magnitude of the failure. The magnitude of the failure is determined by how far reaching the impact of the data fault is and how long the fault persists in the software sample. An initial experiment is presented where two open-source software samples are used to get some initial data on the effectiveness of the framework. The samples are used in two ways: a training/test method and a cross-validation method. These two methods are used to test the learning algorithms used in the experiment against unseen data and against data that is familiar, respectively. The data indicates a strong potential to this line of research and once the framework is automated, a much larger sample size will be collected and evaluated. The key goal of the research at this stage is to determine if the features extracted from the software sample can be used to accurately predict the trend of the rankings of the variables in the top ten to thirty percent within a reasonable range. To reduce the time needed to bring an open-source software component to an acceptable range of reliability and security, only the most important variables are indicated for follow-up with error handling and recovery techniques. Any variables that fall below the ranking threshold, usually the top ten to twenty percent, have too low of an importance ranking to cause a lasting, long-reaching failure.

Categories and Subject Descriptors
D.4.5 [**Operating Systems**]: Reliability – *Fault-tolerance, Verification,* Security and Protection – *Verification*

D.2.5 [**Software Engineering**]: Testing and Debugging – *Diagnostics, Error handling and recovery*

General Terms
Algorithms, Measurement, Design, Reliability, Experimentation, Security, Verification.

Keywords
Variables, relative importance, efficient test cases, prediction algorithms, open-source, software framework

1. INTRODUCTION
The research presented in this paper seeks to address the rising trend of open-source software usage across all levels of software development. Open-source software in general uses a less regulated development process and requires a period of intensive testing and debugging before integrating a component into a current software project. This testing and debugging process can make the idea of utilizing open-source software less appealing than just developing the code in-house. We are developing a framework that combines the elements of: static analysis, dynamic analysis, fault tolerance and learning algorithms to predict the priority of variables in the target software. The primary focus of this paper is to highlight the improvements to the framework with respect to the learning algorithms used to predict variable importance.

The goal is to determine which variables, at what parts in the code, have the highest impact on the system when a data fault is introduce in the memory space belonging to that variable. Once these key variables are identified, error handling and recovery mechanisms can be put in place to prevent the system from entering a fail state or recovering from one gracefully. The framework is being tested on C source code, but we are working on expanding the functionality to include other languages, such as Python and Java. The primary focus of our effort now is the automation of a large portion of the framework to greatly speed up the testing process to create a larger sample set to train and test our data. Once the framework is automated and new data is published on the effectiveness of the work, a version of the framework will be released in order to aid in the recreation of the data by other researchers.

By utilizing relative use cases and test cast prioritization and minimization techniques in tandem with the results of the variable ranking algorithm, the testing time needed to identify key areas in the code that require error handling and recovery mechanisms is reduced [1]. This reduction in cost is key for justifying the use of open-source software in a project that is heavily scrutinized in terms of reliability and security. With the high priority variables identified, the software can be hardened to reduce the occurrence of system fail states. Hardening the code is vital when running in an environment where a fail state could allow the system running the code to be accessed, exposing sensitive data, or when the fail state itself is the concern. The framework allows open-source components to be used in high fidelity systems, by reducing the time and effort needed to bring the components up to the required reliability and security needs.

Assuming access to the source code, this framework is useful for helping to manage the increasing time and cost of completing full

testing suites on software in the late stages of development or during regression testing from a software patch. The framework when used along with statistical or historical data analysis tools can aid in creating a more efficient means of prioritizing test cases. As the framework learns from the development of the tool over time, it can make judgments about the code with previous knowledge gained from earlier releases. The framework design is moving towards reducing the amount of direct interaction and manipulation of the target software in order to collect the data and make the predictions about the relative importance of variables across the sample.

2. RELATED WORK

Utilizing software metrics as machine learning algorithm features to make decisions about software is not a new concept. Of particular importance to this work is the research done on selecting software metrics and other features used in determining fault-proneness of a software module and prediction models for determining reliability and faultiness of a software system.

A comparison of intermittent and transient hard faults on programs is done by Wei, et al. in [2]. The fault model used to represent the faults in this work is similar to our framework using transient fault injection simulation to measure the failure rate of variables under certain conditions. The differences between intermittent and transient fault injection is dependent on the length of the intermittent fault, the fault type, and the origin of the fault in the hardware.

A model based on support vector regression and an estimation distribution model is used in [3]. Estimation of distribution algorithms are used to optimize the parameters of the support vector regression model. This model is used to predict the reliability based on data acquired during the life-cycle of the project. The work showed that diverse population of training data can improve performance and that a hybrid model has better performance to the original model. Another hybrid approach is explored in [4] which seeks to avoid relying heavily on unrealistic assumptions about software usage and reliability models too dependent on the environment where the software is run. The work notes the inaccuracies of these types of reliability models when applied to a real-world case study. The hybrid model which utilized two prediction phases to find trends in the data came out ahead of the single phase prediction model using real word software failure data.

A neural network regression is used in [5] to estimate the failure rates of software. A Bayesian method is used to predict the reliability based around the available software metrics. The results of the work indicate performance issues with working with a generic implementation of a prediction model. The frameworks presented in [6] look at measuring reliability based on the observation of failures and the fault removal process. The framework watches for faults that are not completely removed and measures the complexity of a fault with regards to the delay between identifying a failure and removing the fault which caused the failure. A path-based prediction model is used in the adaptive framework from [7] which focuses on viewing sequences, branches, and loop structures. These metrics are used to determine path reliabilities in the software, which are then used to determine the reliability of the entire system. The resulting data indicated a high correlation between actual software reliability and simulated path reliability.

Deciding on which features to extract from the software is key when attempting to train and test a prediction model to maximize the accuracy of the model. Coupling between object classes and lines of code are investigated in [8] and [9] and found to be a strong indicator of fault-proneness of a software module. Regression and machine learning methods are reviewed in [10] and [11] along with object oriented software metrics to detect faults specific to classes in a system. The data found here suggests that predicting quality of software using machine learning is possible, especially in object-oriented languages. The work in [12] suggests that the accuracy of the prediction models using software metrics comes more from utilizing a combination of code and design metrics and not necessarily from the chosen machine learning algorithm.

3. FRAMEWORK OVERVIEW

The framework aims to complete reliability verification and risk assessment on software where the source code is available as efficiently as possible. The original design for the framework focused on using fault injection to acquire the data needed to make a determination about which variables where the highest priority in the sample and then placing error handling and recovery mechanisms around those variables. Changes to the framework have been made to make extracting the metric data and ranking the variables by relative importance less invasive and more efficient.

The overall objective remains the same, but there are additional steps involving the use of machine learning algorithms to predict priority functions and variables at two major steps in the process. In addition to the inclusion of machine learning elements, the static and dynamic analysis elements have been expanded to include a mixture of currently accepted metrics and some metric specific to this framework. The fault injection framework code was updated and expanded upon, refining the injection method and including a larger number of available injection types. The research goals are to expand the static and dynamic analysis portions of the process to increase the number of metrics that could be used to predict the priority of functions and variables. Machine learning is integrated into the framework to make identifying key locations for error handling more efficient. The fault injection framework is used to validate the predictions and increase the training data sets for teaching the model. Ideally, the process would not utilize fault injection, but would instead have enough data to predict the importance of the variables without needing to calculate relative importance manually using the failure rate, spatial impact and temporal impact. The first step of the research is to identify key pieces of open source software that can be easily obtained, compiled, and tested. At this stage of the research, the process does not consider the functionality of the target software or its general structure.

The research objective is to validate that it is possible to effectively utilize machine learning algorithms in the context of making reliability verification testing more efficient. Once the target software is obtained, the function metrics are gathered and are sent to the binary classification model which predicts whether the functions in each model are important, i.e. that they are part of the critical path. Once the predictions have been made, the priorities of the functions are calculated and the critical path is determined. The idea behind the critical path is discussed in more detail in [1], but it is essentially the key data flow of information throughout the system across modules and the functions within those modules. This is done by determining the highest priority function and passing that information along to the second stage of the process. This stage will determine the critical path using static analysis to map out the flow of data in the program to locate all the functions that have an effect on the variables in the priority function. Further analysis is done to complete the dataset for variable metrics in this stage. The metrics are used as features in the regression model to

predict the importance of each variable relative to the usage of the target software. Lastly, fault injection is completed on the target software to obtain the relative importance and the results of this testing are used to create a validation dataset. This dataset is tested against the previous predictions in order to validate the effectiveness of the prediction models. The following sections further detail the effort that goes into each stage of the process.

4. EXPERIMENT DESIGN

This section will detail the setup and methodology behind the initial test cases used as a proof of concept for the framework. This experiment uses only two open-source software samples for the proof of concept tests. A much larger sample set size will be used in further testing, now that promising results have been obtained.

4.1. Case Study

The case study for this experiment tests the entirety of the framework on a new dataset that had not been seen by the models and then combines a second dataset and first dataset for one larger dataset and to observe the improvements with an increase in data points. The dataset used in this experiment is taken from the testing and analysis of the Mv program from the coreutils. The Mv tests come from testing the prediction model by using the Mp3gain datasets as training data and the Mv datasets as test data. The second part of the validation testing combines these models and completes ten-fold cross validation in order to get a more accurate reading of the prediction potential of the models once more data points are collected.

4.2. Test Setup

The data used in the experiment is collected during the execution of the framework on each of the software samples. Once the framework has completed fault injection analysis and calculated the relative importance, the datasets were run through a series of machine learning algorithms of varying types. WEKA [13] is used to investigate the capability of the predictive models to predict on data that had not been seen before and then again using a method more relevant to smaller datasets. We are currently moving away from utilizing the WEKA tool for training and testing our predictive model and instead utilizing the sklearn package in python [14].

4.3. Variable Attribute Selection

To identify which attributes have the greatest merit when predicting the value of relative importance for each of the variables in the dataset, two attribute selection algorithms are tested. The full training set is used as an input to the attribute selection algorithms. In this case, not all of the algorithms worked utilizing ten-fold cross-validation and so the entire training set is used in order to maintain consistency between the algorithms.

4.3.1. Software Features

All of the metric values from the function pertaining to each individual variable are included in the dataset. At this point it was important to include as many features as possible in order to identify any possible trends between the software metrics in order to evaluate which metrics were the best for predicting the relative importance. The only new metric to add to this section is the inclusion of the probability of corruption for a single function, $P_{Cf,}$ which was added with the improvements to the metrics and was calculated for each function in the target software and used to recalculate the priority of each function accordingly. This metric is used to estimate the potential for a randomly generated data fault to occur in the memory space relating to a variable in a given function.

The new static metrics included in the framework analysis are: local connection (LC), intermodule connections (IMC), reads, writes, and variable type. The local connections are the number of variables that affect or are affected by the variable in question. LC was determined using the source code of the target software and manually investigating each line of code that the variable appeared in. This value can be determined during the dynamic code slicing stage of the framework. The intermodule connections refers to the number of the functions that the variable appears to effect. This includes all function calls within the function currently being investigated. All variables are given a starting value of one local and intermodule connection and for each new connection that value is increased by one. Each branch statement that relies on the variable value increases the local connections by one. In the case of global variables, the intermodule connections value increases by one. The reads and writes are the number of times in the function that the value of the variable is read from and written to. This is done because it is important for helping to predict the temporal impact and potentially the failure rate. Each write to the memory location for a variable is one potential overwrite of the faulty data. Each read is potentially a point where the data fault can affect the system. The variable type has proven to be an interesting metric to include in the datasets. The different variable types have a widely varying effect on the system when injected with faults. For example, with the implementation of Boolean variables the failure rate is particularly low because of the chances of flipping the single bit used to determine value. The failure rate for string variables tends to be higher than other variable types, especially with variables dealing with file paths and file names.

The new dynamic metrics include the frequency and size. The frequency of the variable, VarFreq, is calculated by using the results of the gcov tool. Each occurrence of the variable in the function currently being analyzed and the frequency of the lines of code that are executed with the variable in them are summed to give the frequency that the variable occurs throughout the execution of the program. This value serves as the variable equivalent to the frequency found for each function call using the gcov tool to determine the probability of execution of each function. The size metric is calculated by instrumenting the code with a probe that returns the runtime size of the variable in bytes. This is done because the size of variables is implementation specific and it is important to identify the size of the variables for the particular system being used. For this reason both the size and the variable type are included in the dataset. The size is used in order to calculate the probability of corruption for the specific variable.

The primary relative metrics included in the dataset are: P_{Ev}, P_{Cv}, P_{FI}, and relative importance. The probability of the randomly generated data fault occurring in memory belonging to a variable, $P_{Cv,}$ and then being accessed by the program, $P_{Ev,}$ are determined in order to add an element of relevance to the importance calculation. The probability of these two independent events occurring at the same time is the P_{FI} metric. This is the value that is multiplied by the importance of the variable to calculate the relative importance.

4.3.2. Merit and Rank of Variable Attributes

The principal component algorithm performs an analysis and transformation of the data. Dimensionality reduction is accomplished by choosing enough eigenvectors to account for some percentage of the variance in the original data, in this case a variance of ninety-five percent is used. Dimensionality reduction in this context is the process of reducing the number of random variables under consideration for feature selection. Essentially, each value of the attribute would be multiplied by its coefficient and added to the other attributes in order to obtain the class value.

The relief attribute evaluation algorithm evaluates the worth of an attribute by repeatedly sampling an instance and considering the value of the given attribute for the nearest instance of the same and different class. This algorithm detects the features which are statistically relevant to the desired class. The influence of each attribute indicates how much of an effect the attribute has on the prediction of the relative importance. Higher numbers indicate a greater influence on the prediction and the positive and negative values indicate a direct or indirect proportion to the class, respectively. More information can be found at [15].

4.3.3. Variable Classification

ZeroR This is the default algorithm for classifying the dataset. It serves as the starting point for validating the performance of other classification algorithms on a dataset. This classification has no rules, so it merely predicts the mean, for numeric classes, or the mode, for nominal classes. For the purposes of this framework, ZeroR and the majority of the rule based algorithms are not of much use because they create a series of rules that provide threshold values for the predictions, which is incompatible with the ranking system used to sort the highest priority variables.

Linear Regression A regression algorithm is included as another reference point to compare the other machine learning algorithms against. This is a simple classifier that uses linear regression for prediction. The linear regression model fits a straight line through the set of data points in order to make the sum of the squared values for the distance between the points of the data set as small as is possible. The resulting model is used to predict the desired class.

IBk A version of the K-nearest neighbor classifier, IBk [16], expands on the nearest neighbor algorithm to reduce the high storage requirements. A major drawback of this classification algorithm is that when one classification is significantly more common than another, it is more likely that the test data will be skewed towards the more popular classification. This algorithm was chosen for a similar reason to the others, it is easier to understand and allows for easy customizations in order to get some initial data about the trends in the data set. Another advantage to using this algorithm is that it is conducive to working as a weighted classifier, which will become important when doing cost-sensitive analysis.

KStar K* is an instance-based classifier, meaning the class of a test instance based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy-based distance function. Using the entropy as a distance measure provides this algorithm with benefits over other: consistent handling of symbolic attributes, real values and missing values. Instance based learners work by comparing the current instance, or data point, to a database of pre-classified examples, the training set. This explains the poor performance of this variable in the validation testing that tested the models using new unseen test data instead of cross-validation [17].

LWL stands for locally weighted learning. This algorithm uses an instance-based algorithm to assign instance weights which are then used by a specified WeightedInstancesHandler method in WEKA. This algorithm then utilizes another algorithm depending on the desired prediction values. For classification predictions Naive Bayes is used and linear regression is used for regression models, like this one. The results show that the lazy types of predictors do well with the cross-validation testing because the dataset is balanced be- tween training and test data. The exception to this is the validation testing using just the mv variable dataset that the predictor has not seen before. This problem is potentially solved by adding more data points over varying types to the dataset to give the algorithm a wider range of data points for comparing the test data. More information on the functionality of the algorithm can be found at [18].

Least Median Squared This algorithm implements a least median squared linear regression algorithm based on the original linear regression model discussed previously. Least squared regression functions are generated from random subsamples of the data. The least squared regression with the lowest median squared error is chosen as the final model. This model is an interesting addition to the experimental data, because even though it is based around the linear regression model that provides less desirable prediction rates, this algorithm is consistently a strong predictor. Particularly in dealing with the test data from mv that the model has not seen before in the next section. [19][20].

M5P This algorithm implements base routines for generating an M5 model and trees. The original algorithm M5 is defined in [21] and improvements are described in [22]. The algorithm deals with inducing trees of regression models. The algorithm works by creating a decision-tree induction algorithm to build a tree where each node minimizes the variation of the class values for the nodes beneath it on its branch. The primary difference between the old algorithm and this one is that when pruning an interior node the algorithm decides between replacing the node with a constant value or a regression plane. The attributes of the regression plane are those that are used in decisions in the tree nodes that are located lower in the branch than the current node. [22]

5. RESULTS

The attribute selection algorithms: principal component and relief information gain were run on the combined dataset to obtain the evaluation metrics. The principal component algorithm chose the P_{Ev}, function frequency, P_{FI}, and a few of the variable types as the attributes with the highest impact on the relative importance of the variable. This makes sense when you consider how the relative importance is calculated. The P_{FI} is directly proportional to the value of the relative importance, but the other attributes help to identify some trends in the metrics that can be used to determine the relationship with the other values that make up the relative importance. The type of variable, function frequency, probability that a variable will be executed after having a fault injected and the type of variable all appear to have a closer connection to the failure rate, spatial impact and temporal impact. The relief attribute evaluation provides a similar attribute selection, identifying P_{FI}, variable frequency, P_{Ev} and the read metric as being the strongest

indicators of predicting the correct priority of the variables. Just as before, these values make sense, as the variable frequency is tied to the probability of a fault occurring during the program executing a line of code involving the particular variable and the reads helps to show the likelihood that the data fault will have an effect on the system.

Given the dataset sizes used, it makes sense for there to be a drop in accuracy when dealing with unseen data as opposed to the validation dataset that uses the combined data from Mv and Mp3gain and cross- validation. There is a drop in accuracy from the previous run of the models on just the Mp3gain datasets. The total mean error is quite high for most cases, the exception being the least median squares algorithm. This indicates that against data that has not been seen before, with a small sample size for training, the other models predicted quite poorly when compared to the zeroR model which just predicts the mean value for each. But, when combining the datasets and using the ten-fold cross-validation method in order to test on a more knowledgeable dataset, the error found in the models is back in line with where it would be expected. Note again that the error in predicting the correct rank is less important than the accuracy of the models in correctly identifying the top ten percent, twenty percent and so on.

The average percent correct for the predictions made by the models using the training data from Mp3gain and the test data from Mv is lower than with the cross-validation, because the training set used is based on a single software project, so the relationships between the attributes and the class are particular to that instance. Note that zeroR and Kstar are all zero in this case because they determined a single value and predicted that single value for each instance, leaving no correctly identified instances. Alternatively, the least median squared algorithm gets much better results in the twenty percent range and beyond. The IBk and the M5P algorithms fall behind the least median squared algorithm in their predictive ability, catching up only when determining the bottom percent ranges of eighty percent of variables or higher. Linear regression is particularly bad at dealing with this test case. It is interesting that linear regression does poorly in this case, but the least median squared algorithm which uses linear regression for predicting numeric classes does quite well compared to the other baseline models.

When the datasets are combined into the single dataset for validating through cross- validation, the models saw an increase in predicting ability because of the training process incorporating data points from numerous software projects which provides a better informed model. IBk returns to being the best model in this case, but least median squares is not far behind in accuracy. The most interested data is the in top ten to thirty percent prediction ranges for variables, because the user is most likely interested in upping the error handling on those areas. Overall it appears that least median squared is the best choice for predictions for relative importance of variables, despite performing worse than IBk in the validation dataset. The cross-validation is a good way to measure performance of a smaller sample size that we have in this research, but it is important to measure the capability of the models when dealing with potentially unknown data trends from software projects with different architectures. Figure 1 shows a representation of the prediction accuracy of the different models tested against the Mv dataset using the Mp3gain dataset for training. Figure 2 shows the same, but for the combined validation dataset using cross-validation.

The models that have lower correct predictions also have a greater variance in the predictions made by that model. For example, the least median squared algorithm had the best prediction rates and also the lowest variance in the actual and predicted ranks for each of the variables with respect to the relative importance of each variable. The models that provided a lower true positive prediction

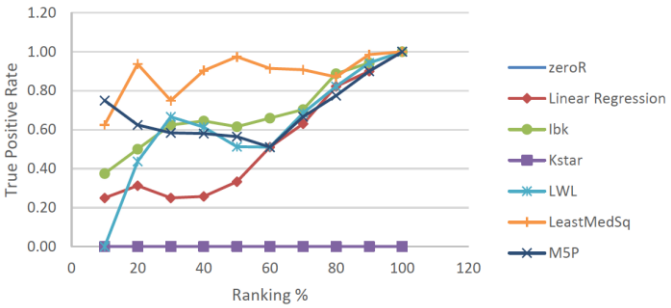

Figure 1. Training/Test Set Mv Predictions

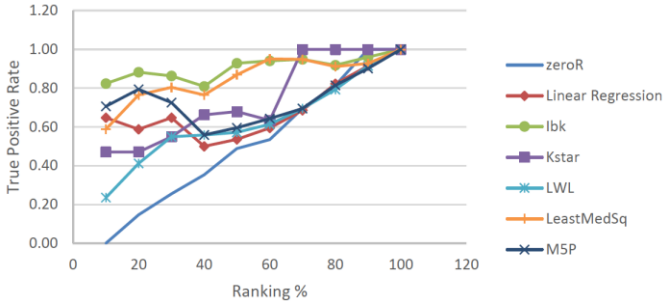

Figure 2. Combined Validation Set Predictions

rate also had a higher difference on average between the actual ranks and the predicted ranks of the variables. Again, zeroR and Kstar have the same values because they both pick threshold values and attempt to match the variables to those as opposed to calculating a unique prediction for each variable. In the top thirty percent of the variable rankings, least median squared, M5P, and Ibk had the best overall results, because the Kstar and zeroR model results are not usable for actual ranking.

The validation dataset results reinforce the trend shown in the previous data for the validation dataset. For the combined dataset, IBk gives the best results, then least median squares and M5P in a distant third. Similarly to the prediction metrics, the overall best model in this experiment is least median squared. When comparing the results of the model predictions against an unseen dataset and the combined dataset, least median squares performs better overall, with IBk outperforming on the combined set, but falling far behind for the dataset from the Mv program. Figure 3 shows the curve of the average difference in actual and predicted rank for the models. Figure 4 shows the same relationship but on the data from the validation dataset.

6. CONCLUSION

The results of the baseline testing on Mp3gain show that machine learning is an effective choice for increasing the efficiency of the framework by correctly predicting the importance of functions and variables. A combination of standard software metrics and the

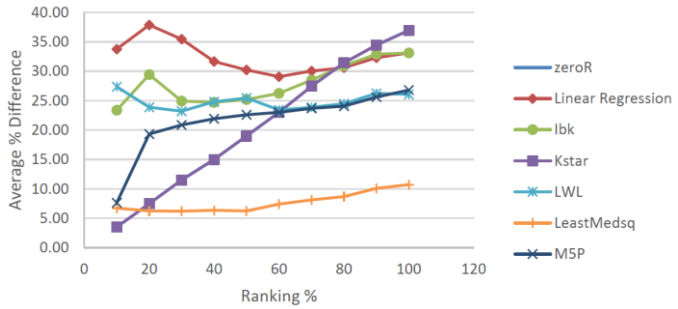

Figure 3. Train/Test Average Percent Difference

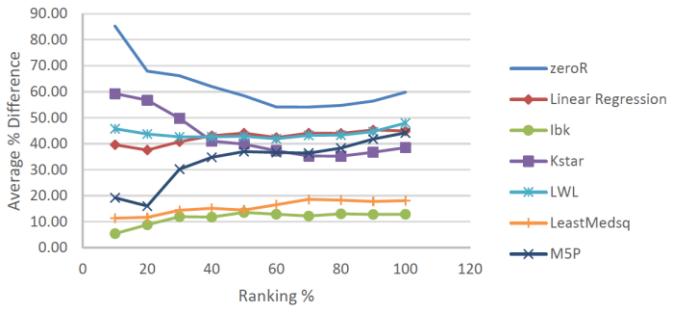

Figure 4. Combined Validation Set Average Percent Difference

relative metrics introduced in this work need to be used in order to make the best possible prediction. In addition to these metrics, some general aspects of the code that are used as features in the dataset are particularly important in the prediction process. For example, the type of variable has a major impact on the importance of the variable for data faults in the system. Pointer variables, including strings, are in general more indicative of higher risk variables in a system due to their more complex nature in how they are stored and utilized in memory. The number of writes occurring for a particular variable was another indicator of high risk variables. A greater focus on the real probabilities of the occurrence of a data fault and its execution was an effective way to measure the importance of variables in the system. The high impact of the propagation metric for predicting function priority indicates that the architecture of the system and data-flow has a high impact on the priority of a function in a system.

Of the various machine learning pre-processing methods, cost benefit analysis proved to be the most effective at increasing the true positive prediction rate. Though the overall percentage of correct predictions lowered in some cases, the cost-benefit analysis increased the total number of true positives. For efficient reliability verification we can afford to have some false positives, but false negatives could lead to letting a potentially high risk function go by untested. Discretization of the attributes in the datasets did not provide a consistent improvement to the prediction capabilities of the models. Using the Multilayer Perceptron and J48 models for function prediction saw true positive prediction rates of over eight-percent, as high as eighty-seven percent with Multilayer Perceptron. It should be noted that overall between both cross-validation and test data, J48, a tree algorithm, performed the best for the function model.

The best algorithms for variable prediction were Least Median Squared, IBk and M5P. In the case of variable prediction, Least Median Squared was far ahead of the others, with eighty to ninety percent correctly ranked variables in the top ten to thirty percent of rankings. Even when dealing with the fresh datasets and the accuracy drop, Least Median Squared showed an average accuracy of seventy-seven percent against the top thirty percent of unseen data points. Comparatively, the lazy and rule based algorithms had drastic drops when dealing with completely unseen data points. Given that it is unlikely that it will be possible to collect enough training data for these models to have a rule for every scenario, it is best to pick a model that gracefully handles unseen data points. Least Median Squared is the best overall model for dealing with ranking the variables according to relative importance. This is the case despite the cross-validation showing better results with IBk, because the lazy model is on average twenty percent worse than Least Median Squared on unseen data points.

7. FUTURE WORK

As work continues on the framework, we are focused on three key goals moving forward: converting the code to python, automating the process from giving the framework a software sample to getting the prediction listing and integrating more features into the prediction algorithm. The framework is currently being converted to Python for its simplicity and efficiency in rapidly prototyping software and its strength in string processing. Another advantage to utilizing python over the original C source code is the abundance of python packages that can be utilized to facilitate faster development. The automation of the project is key in releasing the software for other researchers to incorporate into their own work or just replicate the work that we are doing. With a fully automated framework, the number of open-source software samples that we can analyze and add to the training set will increase much more quickly. In order to facilitate this transition to an automated framework, we are utilizing the pycparser [23] package in python to create abstract syntax trees from the software samples. The desired features are then extracted from the tree and fed into the machine learning algorithm to make predictions. Lastly, to expand the feature list we are starting with the Clang Static Analyzer. [16]

Clang is a compiler front end for C-Type languages running on the LLVM (Low Level Virtual Machine) compiler infrastructure. Clang Static Analyzer is a tool running the Clang compiler designed to perform static code slicing on source files. The tool runs against C-type source files and uses an automated approach to generating reports of possible errors and warnings in the code. As stated previously, the current method of metric generation is done manually. Utilizing the error reports generated by Clang we can automate the process allowing more source to be tested and a greater number of relevant features to be extracted from the source to be used in the prediction algorithm.

We plan to utilize this tool for the error report generation. We can then take the resulting reports to generate our own metrics. Clang will allow us to increase our sample size by helping to automate the analysis of larger samples of source code. Utilizing the output will allow us to generate larger sample metrics and increase our overall feature list. The current goal is to have a first release of the code out in September that can be utilized to replicate the results presented in this paper, and subsequent papers, and can be integrated into other research projects.

This work is sponsored by the NSA CAE Cybersecurity Program under Grant# H98230-15-1-0279.

8. REFERENCES

[1] Pape, P., & Hamilton, D. (2016). Better Reliability Verification in Open-Source Software Using Efficient Test Cases. *CrossTalk*, 31.

[2] Wei, J., Rashid, L., Pattabiraman, K., Gopalakrishnan, S. (2011, June). Comparing the effects of intermittent and transient hardware faults on programs. In Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on (pp. 53-58). IEEE.

[3] Jin, C., Jin, S. W. (2014). Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms. Applied Soft Computing, 15, 113-120.

[4] Pati, J., Shukla, K. K. (2015, February). A Hybrid Technique for Software Reliability Prediction. In Proceedings of the 8th India Software Engineering Conference (pp. 139- 146). ACM.

[5] Wiper, M. P., Palacios, A. P., Marin, J. (2012). Bayesian software reliability prediction using software metrics information. Quality Technology and Quantitative Management, 9(1), 35-44.

[6] Hu, H., Jiang, C. H., Cai, K. Y., Wong, W. E., Mathur, A. P. (2013). Enhancing software reliability estimates using modified adaptive testing. Information and Software Technology, 55(2), 288-300.

[7] Hsu, C. J., Huang, C. Y. (2011). An adaptive reliability analysis using path testing for complex component-based software systems. Reliability, IEEE Transactions on, 60(1), 158-170.

[8] Zhou, Y., and Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. Software Engineering, IEEE Transactions on, 32(10), 771-789.

[9] Gyimothy, T., Ferenc, R., Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. Software Engineering, IEEE Transactions on, 31(10), 897-910.

[10] Suresh, Y., Kumar, L., and Rath, S. K. (2014). Statistical and Machine Learning Methods for Software Fault Prediction Using CK Metric Suite: A Comparative Analysis. International Scholarly Research Notices, 2014.

[11] Malhotra, R., and Jain, A. (2012). Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality. JIPS, 8(2), 241-262.

[12] Jiang, Y., Cuki, B., Menzies, T., and Bartlow, N. (2008, May). Comparing design and code metrics for software quality prediction. In Proceedings of the 4th international workshop on Predictor models in software engineering (pp. 11-18). ACM.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten (2009); The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.

[14] Scikit-learn, Machine Learning in Python. Retrieved April 1, 2016:http://scikit-learn.org/stable/

[15] Kenji Kira, Larry A. Rendell: A Practical Approach to Feature Selection. In: Ninth International Workshop on Machine Learning, 249-256, 1992.

[16] Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. Machine learning, 6(1), 37-66.

[17] John G. Cleary, Leonard E. Trigg: K*: An Instance-based Learner Using an Entropic Distance Measure. In: 12th International Conference on Machine Learning, 108-114, 1995.

[18] Eibe Frank, Mark Hall, Bernhard Pfahringer: Locally Weighted Naive Bayes. In: 19th Conference in Uncertainty in Artificial Intelligence, 249-256, 2003.

[19] Peter J. Rousseeuw, Annick M. Leroy (1987). Robust regression and outlier detection.

[20] Peter J. Rousseeuw, Least Median of Squares regression. Journal of the American Statisctical Association, December 1984, 79(388)

[21] Ross J. Quinlan: Learning with Continuous Classes. In: 5th Australian Joint Conference on Artificial Intelligence, Singapore, 343-348, 1992.

[22] Y. Wang, I. H. Witten: Induction of model trees for predicting continuous classes. In: Poster papers of the 9th European Conference on Machine Learning, 1997.

[23] Pycparser: Complete C99 parser in pure Python, Retrieved April 1, 2016: https://github.com/eliben/pycparser

[24] Clang Static Analyzer homepage. Retreived April 1, 2016: http://clang-analyzer.llvm.org/