2015

The Effects of Parameter Tuning on Machine Learning Performance in a Software Defect Prediction Context

Benjamin N. Province

Follow this and additional works at: <https://researchrepository.wvu.edu/etd>

Recommended Citation

Province, Benjamin N., "The Effects of Parameter Tuning on Machine Learning Performance in a Software Defect Prediction Context" (2015). *Graduate Theses, Dissertations, and Problem Reports*. 6457.
<https://researchrepository.wvu.edu/etd/6457>

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

The Effects of Parameter Tuning on Machine Learning Performance in a Software Defect Prediction Context

Benjamin N. Province

Thesis submitted
to the Statler College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements for the degree of

Master of Science in
Computer Science
with concentration in
Software and Knowledge Engineering

Tim Menzies, Ph.D., Chair
Katerina Goseva-Popstojanova, Ph.D.
Thirimachos Bourlai, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2015

Keywords: Data Mining, Software Defect Prediction, Parameter Tuning,
Machine Learning, Cross-Version Learning, Software Engineering

Copyright 2015 Benjamin Province

ABSTRACT

The Effects of Parameter Tuning on Machine Learning Performance in a Software Defect Prediction Context

Benjamin N. Province

Most machine learning techniques rely on a set of user-defined parameters. Changes in the values of these parameters can greatly affect the prediction performance of the learner. These parameters are typically either set to default values or tuned for best performance on a particular type of data. In this thesis, the parameter-space of four machine learners is explored in order to determine the efficacy of parameter tuning within the context of software defect prediction.

A distinction is made between the typical within-version learning scheme and forward learning, in which learners are trained on defect data from one software version and used to predict defects in the following version. The efficacy of selecting parameters based on within-version tuning and applying those parameters to forward learning is tested. This is done by means of a cross-validated parameter-space grid search with each tuning's performance being compared to the performance of the default tuning given the same data.

For the Bernouli naive Bayes classifier and the random forest classifier, it is found that parameter tuning within-version is a viable strategy for increasing forward learning performance. For the logistic regression classifier, it is found that tuning can be effective within a single version, but parameters learned in this manner do not necessarily perform well in the forward learning case. For the multinomial Bayes classifier, no substantial evidence for the efficacy of parameter tuning is found.

Acknowledgments

First, I would like to thank my advisor, Dr. Tim Menzies, his colleague Dr. Susan Partington, the Lane Department of Computer Science and Electrical Engineering, and the Program of Human Nutrition and Foods for employing me as a GRA for the first year and a half of my graduate studies and thus providing me with an important source of much-needed funding.

I would also like to thank the West Virginia Space Grant Consortium, the Jet Propulsion Laboratory at the California Institute of Technology, and my JPL mentor Justin Lin for making possible the summer internship which inspired me to pursue graduate studies in the field of Computer Science as a bridge into the field of intelligent robotics. I would be remiss not to also thank Dimitris Vassiliadis, the Colorado Space Grant Consortium, and the WVU department of Physics and Astronomy for their support of RockSat, a program in which I participated as an undergraduate which put me on the path towards JPL and eventually graduate school.

I would like to thank my wife, Randy, who helps me to stay focused, and keep my nose to the grindstone. Without her support, this thesis would have almost certainly taken another semester to complete.

I would like to thank my parents and grandmother for encouraging my academic success through childhood and for supplementing my undergraduate scholarships so that I could avoid the burden of student loans.

I would like to thank my friend Andrew Duncan for his friendship and for sharing some of his technical expertise. Any time I face a software or electronics challenge that has me stumped, Andrew is the first person I ask for advice.

Last, but not least, I would like to thank all those who have contributed to Python, SciPy, NumPy, SciKit-Learn, SPyDER, Matplotlib, Ubuntu, LinuxMint, MacPorts, L^AT_EX, and all the other free software I use. Without free and open-source software, this thesis could not have happened, and the world would truly be a much less wonderful place.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Questions	3
1.3	Statement of Thesis	5
1.4	Structure of Thesis	6
2	Background	7
2.1	Simple Methods Work Well for Defect Prediction	7
2.2	Parameter Tuning of Machine Learners	8
2.3	Parameter Tuning in Evolutionary Algorithms	9
3	Methods	10
3.1	Machine Learning Techniques	10
3.1.1	Bayesian Classification	11
3.1.2	Random Forest Classifier	14
3.1.3	Logistic Regression	14
3.2	Evaluating Machine Learning Performance	16
3.2.1	Confusion Matrices	17

3.2.2	Measures of Classification Performance	18
3.3	Data	20
3.3.1	Data Sources	20
3.3.2	CKJM Metrics	21
3.3.3	Data Selection	23
3.4	Experimental Design	25
3.4.1	Parameter Grid Search	26
3.4.2	Cross-Validation Setup	30
3.4.3	Current-Version Vs. Forward-Version Evaluation	31
3.4.4	Machine Learning Trials	32
3.5	Statistical Methods	33
3.5.1	Testing Significance with the Wilcoxon Signed Rank Test	33
3.5.2	Limiting False Discovery for Multiple Hypothesis Testing with the Benjamini-Hochberg Procedure	36
3.5.3	Comparing Current and Forward Learning Performance	39
3.5.4	Splitting Helpful Hairs with Effect Size	40
4	Results	42
4.1	Multinomial Naive Bayes: a Negative Outcome	42
4.2	Logistic Regression: a Mixed Outcome	45
4.3	Random Forest: a Positive Outcome	49
4.4	Bernoulli Naive Bayes: a Very Positive Outcome	53
5	Threats to Validity	56
5.1	Construct Validity	56

5.2	Internal Validity	57
5.3	Conclusion Validity	58
5.4	External Validity	59
6	Conclusions and Suggestions for Implimentation	60
6.1	Current-Version Tuning	61
6.2	Forward Tuning	62
6.3	Pre-Trials: Does Tuning Work for My Learner?	63
6.4	Tuned Learner Pooling	64
6.5	Summary of Conclusions: Research Questions Revisited	65
A	Multinomial Naive Bayes Forward Results By Project	71
B	Logistic Regression Forward Results By Project	73
C	Random Forest Forward Results By Project	75
D	Bernoulli Naive Bayes Forward Results By Project	77
E	Multinomial Naive Bayes Forward Results By Dataset	79
F	Logistic Regression Forward Results By Dataset	82
G	Random Forest Forward Results By Dataset	85
H	Bernoulli Naive Bayes Forward Results By Dataset	88
I	Multinomial Bayes Current Version Results By Dataset	91

J	Logistic Regression Current Version Results By Dataset	92
K	Random Forest Current Version Results By Dataset	93
L	Bernoulli Bayes Current Version Results By Dataset	94

List of Figures

3.1	Confusion matrix examples and labels	18
3.2	g contours as a function of pD , pF	20
3.3	Confusion Matrix for m tests of significance	37
4.1	Multinomial Naive Bayes Mean g Box Plots	44
4.2	Histogram of Logistic Regression Current-Version Successful Tunings	46
4.3	Logistic Regression Mean g Box Plots	48
4.4	Histogram of Random Forest Current-Version Successful Tunings	50
4.5	Random Forest Mean g Box Plots	52
4.6	Histogram of Bernoulli Bayes Current-Version Successful Tunings	53
4.7	Bernoulli Naive Bayes Mean g Box Plots	55

List of Tables

3.1	CKJM Extended Metrics [17]	22
3.2	CKJM datasets of the PROMISE repository	24
3.3	Sci-kit Learn Multinomial Naive Bayes Parameters	27
3.4	Sci-kit Learn Bernoulli Naive Bayes Parameters	28
3.5	Sci-kit Learn Logistic Regression Parameters	28
3.6	Sci-kit Learn Random Forest Parameters	29
3.7	Version-Forward Learning Outcomes	40
3.8	Cohen '88 Effect Sizes	41
3.9	Version-Forward Learning Outcomes by Effect Size	41
4.1	Version-Forward Learning Outcomes: Multinomial Naive Bayes Total Counts	43
4.2	Version-Forward Learning Outcomes: Logistic Regression Total Counts	47
4.3	Version-Forward Learning Outcomes: Logistic Regression Xalan counts	47
4.4	Version-Forward Learning Outcomes: Random Forest Total Counts	50
4.5	Version-Forward Learning Outcomes: Random Forest Total Counts	51
4.6	Version-Forward Learning Outcomes: Bernoulli Bayes Total Counts	54
4.7	Version-Forward Learning Outcomes: Bernoulli Bayes Total Counts	54

Chapter 1

Introduction

1.1 Motivation

The ultimate end-goal of the work presented in this thesis is the betterment of the quality of future software. This is a difficult goal to approach as no single solution the software quality problem exists. As Fred Brooks famously stated “We see no silver bullet. There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement...” [7] Therefore, the only way to make progress towards the end-goal of higher quality software is through incremental improvement to one of a variety of issues with current development practices.

This thesis focuses on one of the major problems plaguing modern software: the prevalence of bugs. Although testing and validation are part of most software development cycles, a shocking number of bugs still make their way into production software to be discovered by the end-users. The simplistic solution the bug problem is to increase the testing effort during development. If bugs are caught by developers before the software is released, they can be fixed before the software is released to end-users. There are several approaches by which bugs may be caught including extensive unit testing, code walkthroughs, and more

exhaustive testing of combinations of use cases and options.

The problem with this simplistic approach is that testing consumes resources. Any increase in software testing and validation effort will also lead to an increase in overall development costs and overall development time. Therefore, the trade-off between greater validation effort and lower development costs is a business decision. Software that has many bugs will be unappealing, but software that comes at great cost may be equally unappealing. For this reason, the holy grail of software validation is the development of processes by which more bugs can be caught without an increase in effort. The real question is how to find bugs by looking *smarter* rather than *harder*.

A data-mining approach is often used to work towards increasing software quality. In this approach, various quantifiable statistics or metrics are extracted from the source code of the software or from the development process. These statistics and metrics are usually collected for several subsets of a software development project along with the number of bugs reported for each subset. In this way, hypotheses about the relation between the software metrics and the likelihood of a bug in the software can be constructed and evaluated. After a correlation has been established, developers can close the loop by either altering their development practices to avoid “bad” characteristics or by applying more testing scrutiny to subsets of the software which exhibit “bad” characteristics.

Data-driven software defect prediction generally relies on machine learning techniques, most of which have several parameters which can be adjusted to vary the performance of the learning algorithm. Most machine learners have a default set of parameters which are chosen by the learner’s creators to reflect the “best” settings for general performance. The default parameters, however, are not necessarily the top-performing set of parameters in all situations. The practice of choosing parameters that lead to increased performance within a particular domain or when applied to a particular type of data is known as parameter tuning.

The intent of this thesis is to explore the practice of parameter tuning, determine it's effectiveness in a Software Engineering defect prediction context, and offer insight it's effective application. By furthering the understanding of effective parameter tuning, the practice of software defect prediction can be improved. Improved defect prediction can lead to more informed software testing, which can contribute to an increase in the overall quality of software.

1.2 Research Questions

1) Can within-version parameter tuning lead to increased classification performance on software defect data when compared to default parameters?

Each machine learner in Scikit-Learn has a default value for each of its parameters. The set of default parameter values have been chosen by the authors of Scikit-Learn or the original developers of the algorithms as a best guess for the appropriate values. It is possible and even probable that for a given combination of training set and test set, some other parameter tuning will produce better results than the default tuning. The real question is weather some parameter tunings preform *significantly* better under repeated re-sampling of a pool of training and testing data than the default tuning. To explore this question, a cross-validation study must be preformed and a test of statistical significance must be applied to the performance results of each non-default tuning.

2) Is parameter tuning effective when the parameters are tuned and the classifier is trained on data from one version of a software project but tested on the next version?

The main goal of machine learning for defect prediction is that lessons learned from previously developed software may be applied to software that is currently in development. Given this goal, it is curious that most defect prediction studies are conducted by sampling both the training and testing data from the same dataset. This typically comes in the form

of a cross-validation study where training and testing data are repeatedly resampled from data reflecting a single version of a single software project. This within-version approach to tuning evaluation doesn't fit the business use scenario in which tuning cannot be conducted on as-of-yet untested versions as class-labels have not yet been established. For example, it may be that the tunings that yield the best performance when training and testing on version 1.1 of a software project do not perform as well when the learner is trained on data from version 1.1 and tested on version 1.2.

3) Is the extent to which a parameter tuning out-performs the default tuning on data from one software version indicative of it's likelihood of success in future versions?

This question extends RQ2 by considering effect size. Assuming that tunings that do well in one software version tend to yield helpful results when they applied to a learner that is tested on a newer version, does the magnitude of their success matter? It may be that any tuning which performs better than the defaults when tested on one version is likely to do so on the next version as well. It may also be that there is some sort of effect size threshold below which tunings should be ignored. It is important to consider "setting the bar" of how much better than the defaults a particular tuning must to in one version to be trusted a subsequent version.

4) Is there a set of parameter tunings that will increase learning performance ubiquitously in the domain of software defect prediction?

This question has the potential to undo RQs 1 through 3 if it can be answered in the affirmative. Another way of thinking about it is "If one tuning is consistently doing better than defaults, why don't we just change the defaults?" If the tuning experiments keep finding a particular tuning which out-performs the defaults, then perhaps the defaults were poorly chosen, or were not set to a value which reflects the best performance within the domain of software defect prediction. If this is the case, then developers should be made aware of the "new defaults" and told that implementing parameter tuning in practice is a moot point.

1.3 Statement of Thesis

The results of the research presented in this thesis can be summarized as follows:

In the context of software defect prediction, parameter tuning is an effective means of boosting prediction performance of certain machine learning algorithms. For some of these learners, this holds true not only when they are trained and tested on data from the same software version, but also when they are tested on future versions not used in training.

This statement is conservatively worded to reflect the fact that parameter tuning does not produce positive results under *all* circumstances. Based on the evidence in this thesis, is the position of the author that on the whole, parameter tuning is a worthwhile endeavor. As parameter tuning is not ubiquitously effective for all learners and all data, trials should be conducted to judge its effectiveness before applying it in new situations.

1.4 Structure of Thesis

The remainder of this thesis is organized as follows:

Chapter 2: Background provides some background information on machine learning for software defect prediction and parameter tuning of machine learners

Chapter 3: Methods discusses the machine learning techniques which are used in this thesis and the means of testing and measuring their predictive performance. It also describes the data sources used and the method of data selection. Chapter 3 also contains a description of the design of the experimental trials and finally, descriptions of the statistical methods employed for comparing results.

Chapter 4: Results presents the results of this study starting with the learner that produces the weakest evidence for the efficacy of parameter tuning and progressing to the learner which produces the strongest evidence for the efficacy of parameter tuning.

Chapter 5: Threats to Validity presents some observations which may pose a threat to the validity of the results of Chapter 4 and the generalizability of the conclusions in Chapter 6.

Chapter 6: Conclusions and Suggestions for Implementation summarizes the findings of this thesis and provides some suggestions for the practical implementation of parameter tuning techniques within the context of software defect prediction.

Appendices list tabular results on a per-project and per-dataset basis.

Chapter 2

Background

2.1 Simple Methods Work Well for Defect Prediction

While training a classifier on previous examples, there may be overfitting problems for complex learning schemes. It is a repeated result in Software Engineering defect prediction that simplistic methods seem to work about as well as more complicated methods. This is why in their 2012 review of fault prediction literature, Hall et al. concluded that “models which perform well tend to use simple, easy to use modeling techniques like Naive Bayes or Logistic Regression. More complex modeling techniques, such as Support Vector Machines, tend to be used by models which perform relatively less well.” [12] In addition, Menzies et al. have shown that there is a ceiling effect of sorts when dealing with SE defect prediction. [21] This means that machine learners tend to get about as good of results when training on a small subset of the instances in the training set.

2.2 Parameter Tuning of Machine Learners

The tuning of parameters of machine learning algorithms is not a new concept. In literature, the general process of parameter tuning is sometimes referred to as “hyperparameter optimization”. The term “hyperparameter” is used in the context of statistical model fitting to refer to a parameter governing shape of the assumed prior distribution of data. This distinguishes it from parameters which are internal to the model which are scaled to achieve the best fit. In this thesis, some, but not all parameters explored fit this strict definition of “hyperparameter”, so the more general term “parameter tuning” is used instead of “hyperparameter optimization”

The most commonly discussed methods of parameter tuning are grid search and random search. Other more sophisticated methods have been proposed which intelligently search the parameter space, choosing new sets of test parameters based on the performance of previous trials. These methods have a few drawbacks when compared to random search and grid search. They generally assume some continuity of performance throughout the parameter space, which may not be true. [5] These iterative search methods are also often difficult to parallelize.

Grid search is a simpler method of parameter optimization that is resistant to discontinuities and nonlinearities provided that the grid is sufficiently fine. It is easily parallelizable and simple to understand, but is also generally regarded as the most computationally expensive method of parameter tuning. Interestingly, Bergstra and Bengio have shown that given the same computational time, a random search is likely to find a better-performing parameter tuning than a grid search with the same bounds. [4] This makes random search an attractive method of parameter tuning for practice, but for reasons of repeatability this thesis will focus on grid search for parameter optimization.

2.3 Parameter Tuning in Evolutionary Algorithms

Aside from classification algorithms, parameter tuning is also used for other complex algorithms including regression, computer vision, and evolutionary algorithms. The literature on tuning of evolutionary algorithms is relatively vast, possibly due to the large computation costs associated with random or grid search of their parameter spaces. Some of these evolutionary algorithm results are of particular interest here due to their use in a closely related domain: search-based Software Engineering.

Search-based Software Engineering (SBSE) is a different type of machine learning. Where defect prediction data mining is concerned with creating predictive models of software defects, SBSE is concerned with finding optimal solutions to a given model. In other words, a search of the model’s input space is conducted in order to find input-values corresponding to a desired output. SBSE often relies on a genetic algorithms approach to search due to the complexity of the models.

In 2011, Arcuri and Fraser explored the effects of parameter tuning of genetic algorithms in a SBSE context. They found that “Different parameter settings cause very large variance in the performance.” and that “Default parameter settings perform relatively well, but are far from optimal on individual problem instances.”. [1] In this thesis, these conclusions are found to be generally extensible to the software defect prediction context with a few exceptions.

In a paper that is not software engineering specific, Smit and Eiben compared three parameter tuning strategies for evolutionary algorithms and found that they all produced parameter values that yielded performance superior to that of manual human-in-the-loop tuning. [33] They also found in a survey of tuning literature that the “tunability” of parameters may vary from algorithm to algorithm and dataset to dataset but that “The efforts [of tuning] are moderate and the gains in performance can be very significant.” [11]

Chapter 3

Methods

3.1 Machine Learning Techniques

Various machine learning techniques have been designed to solve the problem of classification. In order to apply a classification technique, the features which have been extracted from the software data must be mapped into the domain required for the learner's inputs. This may require pre-processing that converts numeric values into categorical values such as high/med/low or the assignment of categorical values to integer representations depending on the cardinality of the data and the requirements of the learning scheme.

This thesis will explore the effects of parameter tuning on three types of machine learners: Bayesian classifiers, logistic regression, and the random forest classifier. It is important to understand the basics of each of these machine learning techniques before we can explore their parameter spaces. While each of these techniques has been discussed at length and with great depth in machine learning literature, this section merely aims to provide an overview of each of these methods.

3.1.1 Bayesian Classification

Bayesian classification is one of the oldest, simplest, and most widely used machine learning techniques ever developed. Several variants of Bayesian classification schemes exist, but the premise of all Bayesian classifiers is the same: the application of Bayes' Rule. When expressed in terms of a single datum of evidence E , the Bayesian probability of a single class outcome C is given as

$$p(C|E) = \frac{p(C) \cdot p(E|C)}{p(E)} \quad (3.1)$$

where $p(C|E)$ is the probability of particular classification given the evidence, $p(C)$ is the prior probability of the classification, $p(E|C)$ is the likelihood of observation E within training examples of class C , and $p(E)$ is the prior probability of the evidence. [37]

In a Bayesian classifier, the goal is not to determine the *actual probability* of a single class, but rather to choose the *most likely* class from a set of possible classifications. In this case, let $\hat{i}$ be the most likely class from the set $\{C_0, C_1, \dots, C_n\}$ given evidence E . Under these conditions, the most likely class given the evidence can be expressed as

$$\hat{i} = \arg \max_{i \in \{0,1,\dots,n\}} \frac{p(C_i) \cdot p(E|C_i)}{p(E)} \quad (3.2)$$

It should be noted that this expression assumes a single datum of evidence. In order to be useful as a classifier, equation 3.2 must be modified to support a set of evidence such that $E = \{E_0, E_1, \dots, E_k\}$. This presents two challenges. Firstly, a specific combination of evidence may not exist in training data. This means that $p(E)$ would become zero. In order to prevent this, $p(E)$ can be removed from the equation because it is a constant across for all C_i evaluated within the argmax. This simplification yields:

$$\hat{i} = \arg \max_{i \in \{0,1,\dots,n\}} p(C_i) \cdot p(E_0, \dots, E_k|C_i) \quad (3.3)$$

The second problem with the expansion from a single evidence datum to a set of evidence

is the expansion of $p(E_0, \dots, E_k | C_i)$. Under the chain rule for conditional probabilities:

$$p(E_0, \dots, E_k | C_i) = p(E_0 | C_i) \cdot p(E_1 | E_0, C_i) \cdot p(E_2 | E_1, E_0, C_i) \cdot \dots \cdot p(E_k | E_{k-1}, \dots, E_0, C_i) \quad (3.4)$$

This is where the “naivety” of a naive Bayes classifier comes in to play. In order to simplify equation 3.4, an assumption of independence between the evidence variables is made. If each element of the evidence is assumed to be independent from each other element (which may or may not hold true) then the conditions for each probability in equation 3.4 can be reduced to C_i such that:

$$p(E_0, \dots, E_k | C_i) = p(E_0 | C_i) \cdot p(E_1 | C_i) \cdot p(E_2 | C_i) \cdot \dots \cdot p(E_k | C_i) = \prod_{j=0}^k p(E_j | C_i) \quad (3.5)$$

Plugging equation 3.5 into equation 3.3 yields the final simplified equation for the class predication of a naive Bayes classifier using a discrete set of evidence. [15]

$$\hat{i} = \arg \max_{i \in \{0,1,\dots,n\}} p(C_i) \prod_{j=0}^k p(E_j | C_i) \quad (3.6)$$

Gaussian Bayes Classifier

One way to modify the Bayesian classifier for continuous numeric features is to assume that the distribution of values for each feature is Gaussian. In this way, the probability of each feature value for a given class can be expressed as:

$$P(E_j | C_i) = \frac{1}{\sqrt{2\pi\sigma_C^2}} \cdot \exp - \frac{(E_j - \mu_C)^2}{2\sigma_C^2} \quad (3.7)$$

where μ_C and σ_C are the mean value and standard deviation of E_j seen in training instances of class C_i . [35]

Multinomial Bayes Classifier

The Multinomial Bayes classifier addresses the problem of missing values through smoothing. It adds a small value, α , to the count of each feature when computing the probability such that all combinations of features and classes have a non-zero probability. This process is called Laplace smoothing when $\alpha = 1$ and Lidstone smoothing when $\alpha < 1$. The smoothed feature values are given as:

$$\hat{E}_{Cj} = \frac{E_{Cj} + \alpha}{E_C + \alpha n} \quad (3.8)$$

where E_{Cj} is the sum of all values of feature j given class C seen in training, E_C is the sum of E_{Cj} for all classes, and n is the number of features. Multinomial Bayes classifier is primarily used in text mining where the features are word counts, which may be sparse. [30]

Bernoulli Bayes Classifier

The Bernoulli Bayes classifier assumes the special case that all features are binary-valued. For numeric features, this requires a binarization pre-processing step. [35] For Bernoulli Bayes, the probability of each feature value for a given class can be expressed as:

$$p(E_j|C_i) = \begin{cases} E_j : & p(j|C_i) \\ \neg E_j : & 1 - p(j|C_i) \end{cases} \quad (3.9)$$

This provides a case for explicitly handling missing features, rather than ignoring them, though the smoothing technique described for Multinomial Bayes can also be applied.

When using numerically valued data is being used with a Bernoulli Bayes classifier, it must be pre-processed to meet the binary value assumption. This is typically done by normalizing each dataset attribute and then applying a binarization threshold. For this thesis, any data which is being passed to a Bernoulli Bayes classifier is normalized by shifting and linearly scaling such that the minimum and maximum values of each attribute are 0 and

1 respectively. A binarization threshold between 0 and 1 is then used as a parameter of the learner.

3.1.2 Random Forest Classifier

The random forest classifier is an ensemble method in which a “forest” of several decision tree classifiers is created. At each node of each decision tree, the best feature on which to split the tree is chosen from a random subset of available features. These best feature on which to split the decision tree is typically selected such that a split would either minimize the node’s Gini impurity or maximize the node’s information gain relative to all other features available for the split. Each decision tree continues to be split until a stopping criteria such as maximum depth or minimum training instances is reached, or the class of all training data for a given node is homogenous. [6]

In addition to the random subsamples of the feature space at each node of each decision tree, some random forest implementations also reduce overfitting by bootstrap sampling the training data to be used on each tree. In this way, if there are n instances of training data, each tree will be trained on n instances randomly sampled with replacement from the original training data.

After a random forest has been trained, new examples can be classified by traversing each of the individual decision trees until a leaf is reached. The majority class of the leaf reached in each decision tree is that tree’s “vote” and the class with the most votes among all trees in the forest is the classifier’s prediction.

3.1.3 Logistic Regression

Regression-based algorithms have the advantage of natively handling continuously defined numeric inputs. When using a regression for classification, some allowances must be made

due to the discrete nature of the defendant variable. One clean way to handle this is by using the logistic function as the basis for regression. The logistic function's range is asymptotically bounded between 0 and 1, which makes it useful for modeling probabilities. The logistic function is given as:

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}} \quad (3.10)$$

The probability of a boolean classification such as the presance of a software bug being “true”, $p(C|E)$, can be modeled as a logistic function given that t is a linear combination of evidence and weights such that:

$$p(C|E) = \sigma(t) \quad \text{where} \quad t = \beta_0 + \beta_1 E_1 + \beta_2 E_2 + \dots + \beta_k E_k \quad (3.11)$$

In this way, the classifier can be trained by finding the values of $\beta_0, \beta_1, \dots$ which best fit the probability of C in training data. [29] In order to do this, we must consider the inverse of logistic function: the logit function. Applying equation 3.11 to the logit function yields:

$$g(E) = \ln \frac{p(C|E)}{1 - p(C|E)} = \beta_0 + \beta_1 E_1 + \beta_2 E_2 + \dots + \beta_k E_k \quad (3.12)$$

The β coefficients from equation 3.12 can be estimated from the training data using an maximum likelihood estimator and an iterative solving technique such as Newton's method. [23] This can lead to overfitting issues which can be mitigated by using regularization. This is typically done using L1 (lasso) or L2 (ridge) regularization. In both cases an extra term reflecting the complexity of the fit is added to the loss equation for the least-squares fit. In L1 regularization, the extra loss term is the sum of the absolute values of all β coefficients, while in L2 regularization, the extra loss term is the geometric sum of all β coefficients, which is equivalent to the magnitude of the β vector. [24] This loss term is scaled through multiplication by a parameter of the learner called “regularization strength”.

It is also possible to solve the dual linear program of the loss minimization function for lo-

gistic regression. This transforms the minimization problem into a transposed maximization problem which results in the same optimal coefficients. Solving the dual problem is more computationally efficient when the evidence set is sparse, or where there are more evidence features than there are instances in the training set. [36]

3.2 Evaluating Machine Learning Performance

The general process of evaluating the performance of a machine learner involves two datasets: a training set and a test set. The classifier uses the training set as a pool of examples from which to draw inference. This means that each datum within the training set must have a class label which is known a-priori. Once a learner instance has received several examples in the form of a training set, it is considered a *trained* classifier.

After a classifier is trained, its performance can be tested by attempting to classify examples from the test set. The test set is passed to the learner *without* class labels, and the learner returns the class which to which each example is most likely to belong based on the training examples and the principles governing the classifier's behavior. In a real-world classification situation, the class labels of the test set may not yet be known; it is the classifier's job to determine the label. When evaluating the performance of a classifier, however, the actual class labels of the test set must be known so that the classifier's predicted labels can be compared to the ground-truth labels.

In order to compare the performance of multiple machine learners, meaningful numeric measures of performance must be chosen. In this case, a performance measure is any way of assigning a number to measure how well a classifier's predicted classes for the test set match up with the ground-truth classes. This step is important because some measures, such as accuracy, may not tell the whole story of performance depending on the nature of class distributions within the test set.

Another consideration for establishing the performance of a given learner on a given dataset is how to partition the set into training and testing sets. When performing a single trial with one train/test partition, the learner’s performance may be sensitive to the chosen partition. A better way to handle this is the LOO (leave one out) method. In LOO, a separate learning trial is conducted for each individual datum in which the selected datum is used as a test set and the rest of the data constitutes the training set. LOO can be impractically computationally expensive, so a subsampling method called stratified k-fold cross-validation with repeats is employed here instead. Cross-validation will be explained below in section 3.4.2.

3.2.1 Confusion Matrices

Before discussing classification performance measures, it is useful to discuss the concept of a confusion matrix. Confusion matrices show the number of items of each class that a classifier predicts to be each other class. Each row in a confusion matrix represents a true class while each column represents a predicted class. (See Figure 3.1 for examples) In the case of 100% accurate prediction, the confusion matrix should be a diagonal matrix as all predictions match the ground-truth. The number of rows and columns in a confusion matrix are equal to the number of possible classes represented, which may be any whole number of two or greater.

For the special case of a two-class problem, in which the only possible outcomes can be represented as positive and negative (or true and false), the confusion matrix consists of only four squares which can be labeled TP (True Positive), FP (False Positive), TN (True Negative), and FN (False Negative). In this category system, the terms “positive” and “negative” refer to the predicted classification and the terms “true” and “false” indicate whether or not the predicted classification was correct. For a visual representation of these labels, see Figure 3.1c.

Figure 3.1: *Confusion matrix examples and labels. Rows represent ground-truth and columns represent predicted classes. Shaded squares represent correct predictions. TP, TN, FP, and FN represent True Positive, True Negative, False Positive, and False Negative respectively.*

	A	B	C
A	10	1	2
B	2	9	3
C	1	1	11

(a) Three-Class CM Example

	T	F
T	11	4
F	0	15

(b) Two-Class CM Example

	T	F
T	TP	FN
F	FP	TN

(c) Two-Class CM Labels

Software bug detection can be represented as a two-class problem in which the presence of a bug is considered “positive” and the absence of a bug is considered “negative”. Although more classes could be used to represent different types of bugs or bugs which were reported or treated in different ways, it is useful to represent the presence of a bug as a two-class problem. Restricting the number of classes to two increases performance by reducing the number of categories between which a classifier must differentiate and also allows the use of performance measures such as “probability of detection” which have little meaning when there are more than two classes.

3.2.2 Measures of Classification Performance

In order to evaluate the efficacy of machine learners, some measurements need to be applied to the confusion matrix yielded from the testing of a trained classifier. For most machine learning circles, the typical performance evaluation metrics are precision and recall. Within the domain of Software Engineering defect prediction, the majority of the recent publications favor pD (probability of detection) and pF (probability of false-alarm). This is largely due to scaling issues with precision that arise when the testing data has positive/negative ratios near 0 or 100%. [20]

It should be noted that for a two-class problem, pD is actually equivalent to recall. pD

can be expressed as the number of true positives divided by the total number of positive cases, while pF can be expressed as the number of false positives divided by the total number of negative cases.

$$pD = \text{recall} = \frac{\# \text{ of defects correctly predicted}}{\# \text{ of actual defects}} = \frac{TP}{TP + FN} \quad (3.13)$$

$$pF = \frac{\# \text{ of false defect predictions}}{\# \text{ of non-defective items}} = \frac{FP}{FP + TN} \quad (3.14)$$

This thesis will focus primarily on pD and pF as the chief metrics for comparing machine learning results because they are business-intuitive for defect prediction and because they are in line with recent publications. The goal is to have the highest possible pD and the lowest possible pF , but there is often a trade-off between the two based on the parameters chosen for a given classification scheme.

In order to effectively rank results it is useful to have a single metric which reflects overall performance, but neither pD or pF is sufficient on their own. As an example, a dummy classifier which always predicts “true” is going to have a pD of 100% but will also have a pF of 100%. Conversely, a dummy classifier which always predicts “false” will have a pF of 0, which is excellent, but will also score 0 on pD , making it useless. Consequently, we can define a “no information” line on a pD , pF plot where $pD=pF$. A dummy classifier which guesses positive $n\%$ of the time and negative $100 - n\%$ of the time is expected to have a performance along this line. For a visual representation of the line of no information, see figure 3.2.

In order to reflect overall performance, a synthetic metric will be used. A typical synthetic metric for combining precision and recall is the balance F-score which is the harmonic mean of precision and recall. In this case, as we are focusing on pD and pF , a similar metric, g , will be used where g defined as the harmonic mean of pD and $1-pF$.

Figure 3.2: g contours as a function of pD , pF . The solid line represents the line of no information, the expected performance from random guess.

$$g = \frac{2}{\frac{1}{pD} + \frac{1}{1-pF}} = \frac{2(pD)(1-pF)}{1+pD-pF} \quad (3.15)$$

A contour plot of the values of g based on pD and pF can be seen in figure 3.2.

3.3 Data

3.3.1 Data Sources

Some companies keep records of completed software projects which may be used to learn lessons for future development. These software companies have little incentive to release their development data to the general public. Most Software Engineering researchers rely on public data from various sources including the PROMISE repository [22], a engineering project data repository supported by North Carolina State University.

The PROMISE repository includes many Software Engineering defect and effort prediction datasets from previous software projects. The software project data hosted on PROMISE has features which are measured with a variety of different types of software metrics. Many of these software projects have data available on PROMISE from multiple versions.

In the interest of openness and reproducibility of results, only data publicly available on the PROMISE dataset will be used in this thesis. In addition, to this restriction, the data used in this thesis will also be restricted to software projects for which three or more versions (numbered releases) are available. This will allow for the evaluation of cross-version learning. Another necessary restriction is that the software metrics used be homogenous to ensure fair comparison. To this end, only software projects using the CKJM object-oriented metrics will be considered as choosing CKJM metrics allows for the largest plurality of datasets to be included when considering the first two restrictions.

3.3.2 CKJM Metrics

CKJM (Chidamber and Kemerer Java Metrics) is a tool for tabulating the object-oriented software metrics devised by Chidamber and Kemerer [8] for given Java classes. An updated version of CKJM, called CKJM Extended, also tabulates several other object-oriented software metrics on a per-class basis. [17] CKJM Extended includes Henderson-Sellers' LCOM3 metric [13], Martin's afferent and efferent couplings [18], Bansiya Davis' QMOOD metrics [2], Tang, Kao, and Chen's coupling and complexity metrics [34], and McCabe's cyclomatic complexity measure [19] as well as the original Chidamber and Kemerer metrics and lines of code (LOC). For descriptions of all CKJM Extended metrics, see Table 3.1.

Several open source Java projects were analyzed by Jureczko and Spinellis using CKJM Extended, and a dataset was constructed from each version of each project. [17] These datasets consisted of the CKJM Extended metrics as well as a class variable indicating

Table 3.1: CKJM Extended Metrics [17]

Name	Description	Source
Weighted methods per class (WMC)	The value of the WMC is equal to the number of methods in the class (assuming unity weights for all methods).	C&K [8]
Depth of Inheritance Tree (DIT)	The DIT metric provides for each class a measure of the inheritance levels from the object hierarchy top.	C&K [8]
Number of Children (NOC)	The NOC metric simply measures the number of immediate descendants of the class.	C&K [8]
Coupling between object classes (CBO)	The CBO metric represents the number of classes coupled to a given class (efferent couplings and afferent couplings). This couplings can occur through method calls, field accesses, inheritance, method arguments, return types, and exceptions.	C&K [8]
Response for a Class (RFC)	The RFC metric measures the number of different methods that can be executed when an object of that class receives a message. Ideally, we would want to find for each method of the class, the methods that class will call, and repeat this for each called method, calculating what is called the transitive closure of the method call graph. This process can however be both expensive and quite inaccurate. Ckjm calculates a rough approximation to the response set by simply inspecting method calls within the class method bodies. The value of RFC is the sum of number of methods called within the class method bodies and the number of class methods. This simplification was also used in the Chidamber and Kemerer's description of the metric.	C&K [8]
Lack of cohesion in methods (LCOM)	The LCOM metric counts the sets of methods in a class that are not related through the sharing of some of the class fields. The original definition of this metric (which is the one used in ckjm) considers all pairs of class methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods do not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that do not share a field access the number of method pairs that do.	C&K [8]
Lack of cohesion in methods (LCOM3)	$\left(\frac{\frac{1}{a} \sum_{j=1}^{10} \mu(A_j)}{1-m} \right) - m$ <div style="display: flex; justify-content: space-between; font-size: small;"> <div> m - number of procedures (methods) in class a - number of variables (attributes) in class $\mu(A)$ - number of methods that access a variable </div> <div>Henderson-Sellers [13]</div> </div>	
Afferent couplings (Ca)	The Ca metric represents the number of classes that depend upon the measured class.	Martin [18]
Efferent couplings (Ce)	The Ca metric represents the number of classes that the measured class is depended upon.	Martin [18]
Number of Public Methods (NPM)	The NPM metric simply counts all the methods in a class that are declared as public. The metric is known also as Class Interface Size (CIS)	QMOOD [2]
Data Access Metric (DAM)	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.	QMOOD [2]
Measure of Aggregation (MOA)	This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of class fields whose types are user defined classes.	QMOOD [2]
Measure of Functional Abstraction (MFA)	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class. The constructors and the java.lang.Object (as parent) are ignored.	QMOOD [2]
Cohesion Among Methods of Class (CAM)	This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.	QMOOD [2]
Inheritance Coupling (IC)	This metric provides the number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of its inherited methods functionally dependent on the new or redefined methods in the class.	Tang [34]
Coupling Between Methods (CBM)	The metric measures the total number of new/redefined methods to which all the inherited methods are coupled. There is a coupling when at least one of the given in the IC metric definition conditions is held.	Tang [34]
Average Method Complexity (AMC)	This metric measures the average method size for each class. Size of a method is equal to the number of Java binary codes in the method.	Tang [34]
McCabe's cyclomatic complexity (CC)	CC is equal to number of different paths in a method plus one. Max(CC) - the greatest value of CC among methods of the investigated class Avg(CC) - the arithmetic mean of the CC value in the investigated class	McCabe [19]
Lines of Code (LOC)	The LOC metric based on Java binary code. It is the sum of number of fields, number of methods and number of instructions in every method of the investigated class.	

whether a bug was present in each Java class and if that bug was fixed or not. It should be noted that “class variable” refers to the classification of a bug being present, and may be potentially confused with a Java class. In this case, each Java class in each version of each project has 19 metric features and one class variable indicating the presence or lack thereof of a bug. The datasets that Jureczko and Spinellis constructed in this manner were contributed to the PROMISE repository. Several of these data are used in this thesis.

3.3.3 Data Selection

In summary, the criteria for the selection of data which can help answer the research questions are as follows:

1. The data sets must have a ground-truth class which indicates the presence or absence of a bug in each software module.
2. All datasets selected must have homogenous feature types. In other words, the same software metrics must be present in all datasets.
3. For each software project selected, there must be available datasets reflecting the development of multiple versions.
4. All selected data must be available for study and replication

In order to meet requirement 4, data is selected from the PROMISE repository, which is publicly available at <http://openscience.us>. [22] In order to meet requirement 1, only defect prediction data will be considered. In order to meet requirement 2 while allowing the largest quantity of projects meeting requirement 3, only the CKJM-featured defect data will be considered for selection. A list of these datasets can be seen in Table 3.2

Table 3.2: *CKJM datasets of the PROMISE repository*

Project	Versions	Selected
Ant	1.3, 1.4, 1.5, 1.6, 1.7	Yes
Arc	1	No
Berek	1	No
Camel	1.0, 1.2, 1.4, 1.6	Yes
CKJM	1	No
e-Learning	1	No
Forrest	0.6, 0.7, 0.8	No
Intercafe	1	No
Ivy	1.1, 1.4, 2.0	Yes
Jedit	3.2, 4.0, 4.1, 4.2, 4.3	Yes
Kalkulator	1	No
Log4j	1.0, 1.1, 1.2	Yes
Lucene	2.0, 2.2, 2.4	Yes
Nieruchomosci	1	No
Pbeans	1, 2	No
Pdftranslator	1	No
Poi	1.5, 2.0, 2.5, 3.0	No
Prop	1, 2, 3, 4, 5, 6	No
Redaktor	1	No
Serapion	1	No
Synapse	1.0, 1.1, 1.2	Yes
Systemdata	1	No
SzybkaFucha	1	No
Termoproject	1	No
Tomcat	1	No
Velocity	1.4, 1.5, 1.6	Yes
workflow	1	No
wspomaganiepi	1	No
Xalan	2.4, 2.5, 2.6, 2.7	Yes
Xerces	1.0, 1.2, 1.3, 1.4	Yes
Zuzel	1	No

3.4 Experimental Design

In order to experiment with parameter tuning, a set of Python scripts were developed for evaluating the performance of four different classifiers: the Multinomial Bayes, the Bernoulli Bayes, logistic regression, and random forests. In order to reduce the possibility of errors in the implementation of these learners and to facilitate reproducibility, SciKit-Learn’s Python implementation of these learners was used. [28]

The *Dataset* class used in this project to manage the data and dataset attributes was re-used from the work of Vasil Papakroni’s masters work in the development of a machine learning algorithm called PEEKING2. [27]

A Python package called “BenTune” was created for the purpose of this thesis which includes, among others, a *Learner* class and a subclass for each of the selected learners which. The *Learner* classes act as front-ends for the SciKit-Learn learners, accepting training and testing data in the form Paporkoni’s *Dataset* objects and parameter dictionaries in a consistent format. The *Learner* objects then initialize the SciKit-Learn learners in the appropriate ways for each individual learner with parameters reflecting the *Learner*’s parameter values and with training data coerced from the *Dataset* object into the proper format for the learner.

After initializing and training their Scikit-Learn learner, the *Learner* objects then evaluate its performance by coercing each row of the testing *Dataset* object into a format appropriate for the learner, and soliciting a prediction of that row’s class. These predictions along with the ground-truth value for each row are used to initialize an object of the *Results* class, which is also implemented as part of BenTune.

The *Results* class contains methods for computing confusion matrices and performance metrics based on the pairs of ground-truth and predicted class values. *Results* also contains methods for combining with other results to form aggregations. In this way, a *Results* tree can be formed. Non-leaf nodes in such a tree can be statistically compared to other nodes

of the same height using their children samples.

The general experimental design of this thesis is that the classes described above are manipulated in such a way as to conduct a grid-search of the parameter space of each selected learner. Each parameter tuning evaluated is done so using a 5-fold, 5-repeat, class-stratified cross-validation. The results of these cross-validation trials are each individually compared to the default tuning using a test of statistical significance. This is done in two different evaluation schemes, current-version and forward evaluation. Afterwards, the results are analyzed. Each of these individual steps is described in greater detail in the following sections.

3.4.1 Parameter Grid Search

For the sake of relative completeness, a grid search method is used for finding parameter tunings which out-perform the default parameters. In a grid search, a list of possible values for each parameter of each learner is constructed, then all combinations of each listed parameter value are exhaustively tested. For discretely-valued parameters with a finite number of valid values, this is a simple matter of listing all valid parameter values. For parameters in a continuous regime, the definition of a list of values to be explored is somewhat arbitrary.

When selecting discrete parameter values from continuous ranges, it is tempting to define a minimum sensible value and a maximum sensible value and explore 100 values between the minimum and maximum spaced at regular intervals. However, as a matter of pragmatism, one must remember that the total computation time for a grid-search of all parameter permutations is an exponential problem. If a learner has three continuously-valued parameters and each is tested at 100 discrete values, the total number cross-validated trials required to test all parameter permutations is $100^3 = 1 \text{ million}$ trials. More practically, if the same three parameters are discretized into ten values each, a much more reasonable $10^3 = 1000$ trials are required.

Table 3.3: *Sci-kit Learn Multinomial Naive Bayes Parameters*

Parameter	Description	Values		
		Default	Tested	#
alpha	Smoothing coefficient: $\alpha = 1$ for Laplace smoothing, $0 < \alpha < 1$ for Lidstone smoothing, $\alpha = 0$ for no smoothing	1.0	0, 0.1, ..., 3.9, 4.0	41
fit_prior	Determines whether or not class prior probabilities are included in classification. If false, class prior probabilities are assumed to be equal.	True	True, False	2
Total Permutations:				82

In order to keep the parameter space small enough to be practically explored in a grid search, some decisions must be made about which values to explore for each continuously-valued parameter. These choices are the product of a rational line of reasoning, but can not, by their nature, be defended against all alternate sets of values which might have been explored instead. For instance, if $\{2, 4, 6, 8\}$ is chosen as a rational set of values to be explored for a given parameter, it can not be argued to be any better or worse of a choice than $\{2.1, 4.1, 6.1, 8.1\}$ or $\{1.9, 3.9, 5.9, 7.9\}$ would have been.

For each learner which was tested, the parameter values used and their default values are listed in Tables 3.3 through 3.6. For the most part, default values from Sci-Kit learn’s documentation were used as defaults for this project. In the case of two Random Forest parameters, the defaults were modified, but not in ways which would greatly effect performance. For more details on this substitution, see Table 3.6.

For Logistic Regression, when use of one parameter value precludes certain values of other parameters, the original SciKit-Learn values were modified. This was done in two instances. In the first one, Logistic regression originally had a *penalty* parameter which could be set to ‘l1’ or ‘l2’ and a *dual* parameter which could be set to True or False, but could only be True if *penalty* was set to ‘l2’. To simplify this, the intermediate LogisticRegression class which calls the SciKit-Learn logistic regression learner accepts only the *penalty* parameter, but with values of ‘l1’, ‘l2’, or ‘l2-dual’ and sets SciKit-Learn’s *penalty* and *dual* parameters

Table 3.4: *Sci-kit Learn Bernoulli Naive Bayes Parameters*

Parameter	Description	Values		
		Default	Tested	#
alpha	Smoothing coefficient: $\alpha = 1$ for Laplace smoothing, $0 < \alpha < 1$ for Lidstone smoothing, $\alpha = 0$ for no smoothing	1.0	0, 0.1, ..., 3.9, 4.0	41
fit_prior	Determines whether or not class prior probabilities are included in classification. If false, class prior probabilities are assumed to be equal.	True	True, False	2
binarize	Threshold for binarizing input values. (Input values are normalized between 0 1 in preprocessing)	0.50	0, 0.05, ..., 0.95, 1.00	21
Total Permutations:				1722

Table 3.5: *Sci-kit Learn Logistic Regression Parameters*

Parameter	Description	Values		
		Default	Tested	#
penalty	Specifies the norm used in regularization: L1 or L2. 'l2_dual' also specifies use of the dual formulation of the regression model whereas 'l1', 'l2' use primal formulation.	l2	'l1', 'l2', 'l2_dual'	3
C	Inverse regularization strength	1.0	0.1, 0.2, ..., 2.9, 3.0	30
intercept_scaling	a constant-value synthetic feature added to the data. This scales β_0 from Equation 3.12. If None, removes the constant from the decision function, effectively setting $\beta_0 = 0$.	1.0	None, 0.1, 0.2, ..., 2.9, 3.0	31
Total Permutations:				2790

Table 3.6: *Sci-kit Learn Random Forest Parameters*

Parameter	Description	Values		
		Default	Tested	#
n_estimators	the number of 'trees' in the forest	10	10, 20, 30, 50, 90	5
criterion	specifies the function used to test the fitness of a split. 'gini' represents Gini impurity, and 'entropy' represents information gain.	'gini'	'gini', 'entropy'	2
max_features	the number of random features to be considered when finding a split. Note: Scikit-Learn's default is 'auto' which determines a value based on $\sqrt{n_features}$. As there are 19 features of the CKJM metrics, and $\text{int}(\sqrt{19}) = 4$, 4 has been substituted for 'auto' as the default.	4	2, 4, 8, 16	4
max_depth	maximum tree depth Note: Scikit-Learn's default is None. In order to prevent exceedingly large trees when min_samples_split and min_samples_leaf are small, 75 was substituted as default.	75	5, 15, 30, 50, 75	5
min_samples_split	the minimum number of instances that must be in a node in order for that node to be eligible for a split	2	1, 2, 4, 6	4
min_samples_leaf	the minimum number of instances that must be in each leaf	1	1, 2, 4, 8, 16	5
Total Permutations:				4000

accordingly. For a description of the new *penalty* parameter, see Table 3.5.

The other Logistic Regression parameter which was altered is *intercept_scaling*. SciKit-Learn’s Logistic Regression has a parameter called *fit_intercept* which can be set to True or False. If *fit_intercept* is True, then a floating point value must be provided for *intercept_scaling*. In the intermediate LogisticRegression class which calls the SciKit-Learn logistic regression learner, the *fit_intercept* parameter was eliminated, and *intercept_scaling* was made to allow a floating point value or None. If *intercept_scaling* is set to None, then SciKit-Learn’s Logistic Regression was called with *fit_intercept*=False. For a description of the new *intercept_scaling* parameter, see Table 3.5.

3.4.2 Cross-Validation Setup

Before we can begin to explore an experimental setup which can answer the research questions at hand, it is important to consider the validation strategy to be used. In this case, because multiple experimental treatments are being given to the same data, the results may be skewed due to sampling error if the data is re-sampled for cross-validation at the beginning of each phase of the experiments. Therefore, it makes the most sense to divide the data sets into cross-validation folds and repeats and save the subsets to disk. Then, rather than loading and re-sampling the data for each treatment of each experiment, each fold and repeat of the cross validations can be loaded from their individual files and the sampling will remain constant across all treatments. In this way, the results can be compared using a paired significance test.

For this thesis, stratified cross-validation is used. In stratified cross-validation, the distribution of classes is preserved while subsampling. In a two-class case, this distribution of classes is equivalent to the positive to negative ratio. Stratified cross-validation is implemented by sampling from the set of positive examples independently from the set of negative examples.

In this case, a 5x5 cross-validation study has been chosen as this size yields a set of evaluation scores for each treatment and version of 25 while maintaining a train/test split of 80%/20%. This means that each dataset is divided randomly into five sub-samples or “folds” and this sup-sampling process is repeated five times. The pseudocode for the generation of cross-validation subsets is given in Algorithm 1.

Algorithm 1: Split Data For Cross-Validation

Data: There are P projects, each with $V(p)$ versions;

Helping Functions:

getDataset(p, v) returns the dataset of version index v of project index p ;

stratified_random_sample(D, n) splits D into n equal-width bins stratified by class;

export_dataset($D, filename$) saves D to disk;

```

for  $p \in [0 : P]$  do
  for  $v \in [0 : V(p)]$  do
     $current\_version \leftarrow getDataset(p, v)$ ;
    for  $repeat \in [0 : 5]$  do
       $F[0 : 5] \leftarrow stratified\_random\_sample(current\_version, 5)$  ;
      for  $fold \in [0 : 5]$  do
         $export\_dataset(F[fold], p\_v\_repeat\_fold.arff)$ ;

```

3.4.3 Current-Version Vs. Forward-Version Evaluation

Research question one is stated as “*Does parameter tuning lead to increased classification performance for defect prediction data when compared to default parameters?*” It can be answered by comparing the results yielded by testing learners of various parameter permutations to the results yielded by testing the same learners on the same data but with default parameters.

Research question two is stated as “*Is there a difference in the effects of parameter tuning when a trained classifier is tested within a single software version vs. the next version of the same software?*” This can be tested by looking at the parameter tunings that preform

relatively well when trained and tested within a single version of a software and comparing their performance when trained on one version and tested on the next. To answer this, each parameter tuning must be compared to default tunings under two different scenarios: a *current* evaluation and *forward* evaluation.

In the *current* evaluation scenario, a learner is trained and tested on data from the same software version. This is not to say that the training and testing data is the same. Using a five-fold cross-validation, a learner is trained on four of the folds while being tested on the other. This is repeated using all folds and all repeats of the cross-validation scheme as the test set and all non-testing folds from the same repeat as training sets.

Similarly in the *forward* evaluation scenario, the cross-validation scheme described above is used, but with one difference: the testing data comes from the next software version after the version used for training. Four of five folds from the “current” version are still used to train the classifier, but the corresponding fold and repeat from the *next* version is substituted as the testing set.

3.4.4 Machine Learning Trials

Due to the large amount of CPU time required to preform the hundreds of thousands of cross-validated machine learning trials, the procedure for preforming the trials is implemented separately from the procedure for analyzing and comparing the relative performance of the trials. In order to do this, each project, version, learning scheme, and parameter permutation is assigned a unique index. The process which preforms the learning trials saves the confusion matrix resulting from the testing of each cross-validation fold and repeat to a CSV file with a name reflecting the project, version, learner, and parameter indexes. Several of these machine learning trial processes are run in parallel on a multi-core computer, and the confusion matrices are analyzed and compared post-hoc by another script.

It should be noted that the trials required for the *current* evaluation scenario described in the previous section in reference to RQ2 are also sufficient to answer RQ1. Therefore, a combined trials script can be used to generate the performance data necessary to answer both questions. The pseudocode for this procedure for the RQ1 and RQ2 trials is given in Algorithm 2.

In the actual Python implementation of this pseudocode, the output file is a CSV with 50 rows and four columns. Each pair of two rows represents confusion matrices from a different cross-validation fold and repeat. The first two columns represent confusion matrices from the *current* evaluation scenario and the third and fourth columns represent confusion matrices from the *forward* evaluation scenario. It should also be noted that in the actual implementation, the processes corresponding to each combination of project, version, learner, and tuning are parallelized for speed. In other words, everything inside of the fourth for-loop of Algorithm 2 may be executed asynchronously by any one of a handful of worker processes.

309,528 of these trials, each with 25 *current* cross-validation sub-trials and 223,548 with 25 *forward* cross-validation sub-trials (13,326,900 total sub-trials) were preformed in 2 worker threads on a 3.6 GHz Intel Core2 Duo machine running Linux Mint 17 x64 in 2 days, 12 hours, and 48 minutes.

3.5 Statistical Methods

3.5.1 Testing Significance with the Wilcoxon Signed Rank Test

After the machine learning trials were conducted, the performances of each parameter tuning were compared to the performance of the default tuning. The first step in doing this is to label the performance of each tuning on each dataset into one of three categories: significantly better than default, significantly worse than default, and not significantly different from

Algorithm 2: Learning Trials for RQ 1 and RQ 2

Data:

Each project has multiple versions associated with it;
Each version has 25 files for 5x5 cross-validation folds and repeats;
Each learner has multiple parameter “tunings” associated with it;

Helping Functions:

load_dataset(version, repeat, fold) loads and returns a dataset file;
join_datasets(...) joins all datasets passed to it and returns the superset;
learner.apply_parameters(P) returns a copy of learner with parameters P;
learner.train(D) trains learner on dataset *D*;
learner.test(D) tests learner on *D* and returns the confusion matrix;
file(p, v, l, t) returns an open file with name unique to *p, v, l, t*;
execute_async(fn, args) calls function *fn* with arguments *args* in a new process;

```
def worker_kernel(project, version, learner, tuning):  
    for repeat ∈ [0 : 5] do  
        for fold ∈ [0 : 5] do  
            cur_version[fold] ← load_dataset(version, repeat, fold);  
            for fold ∈ [0 : 5] do  
                training_set ← join_datasets(cur_version where index ≠ fold);  
                cur_test ← cur_version[fold];  
                learner ← learner.apply_parameters(tuning);  
                learner.train(training_set);  
                output[repeat][fold][0] ← learner.test(cur_test);  
                if version+1 exists then  
                    nxt_test ← load_dataset(version + 1, repeat, fold);  
                    output[repeat][fold][1] ← learner.test(fwd_test);  
                output[repeat][fold][1] ← null  
            file(project, version, learner, tuning) ← output;
```

```
work_queue = [ ];  
for project ∈ Projects do  
    for version ∈ project do  
        for learner ∈ Learners do  
            for tuning ∈ learner do //including default tuning  
                work_queue.append((project, version, learner, tuning))  
            while work_queue ≠ [ ] do  
                if CPU is free then  
                    execute_async(worker_kernel, work_queue.pop());
```

default. To do this, the set of g metrics (see Section 3.2.2) for each of the 25 cross-validation sub-trials of each tuning is compared to the set of g metrics for the default tuning. This is done on a per-dataset basis. A test of statistical significance is applied using the g metrics of the 25 cross-validation sub-trials for a given dataset as samples.

In choosing a test of significance, it is important to consider two things. Firstly, the performance data may or may not be normally distributed, so a non-parametric test is most appropriate. Secondly, because each tuning is tested on the same cross-validation folds and repeats as the default tuning, the performance data can be considered paired samples. The only difference between the populations is which tuning “treatment” was applied. Due to the paired nature of this data, a paired test can be applied for greater statistical power. Given these two considerations, the Wilcoxon signed rank test was chosen for determining whether or not a given parameter tuning performed significantly different from the default tuning on the same data. The specific implementation of the Wilcoxon test used comes from the `scipy.stats` package [16] with the zero-difference behavior set to “zsplit”. For more information, see the Scipy Wilcoxon documentation at <http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html>

The Wilcoxon test has three main assumptions. Firstly the data must be paired samples. As discussed above, this assumption is met because the data and learners remain the same, with the only difference being the parameter tuning used. Secondly, the scale of the values tested must be ordinal or greater. This assumption is met as the g values are numeric. Finally, each pair must be independent from the other pairs. This assumption is satisfied because the g values are a result of the performance when testing on each fold and repeat of the cross-validation partitions. Each of these folds is chosen from stratified random subsampling of the whole dataset and thus constitutes an independent sample.

The `scipy.stats.Wilcoxon` function returns the largest rank sum from the Wilcoxon test as well as a p-value which is based on the assumption of normality of the expected distribution of Wilcoxon signed rank sums. This assumption is rooted in the asymptotic normality of

the Wilcoxon rank sum under the null hypothesis and states that the distribution of rank sums may be assumed to be normal for large sample sizes. [31] The Scipy documentation recommends that sample sizes be greater than 20 in order for this assumption to be valid and for the p-values to be trusted. In this case, the sample size for the comparison of cross-validation trials is 25.

3.5.2 Limiting False Discovery for Multiple Hypothesis Testing with the Benjamini-Hochberg Procedure

Typically, when a test of significance is applied, an α value is set prior to testing. The test is considered to indicate significance if the p-value associated with the probability of the truth of the null hypothesis is less than the chosen α . In this way, the odds of incorrectly rejecting the null hypothesis in a given test are at most α .

When many alternative hypotheses are tested against a null hypothesis in this manner, it is expected that for *some* of the tests, the null hypothesis will be incorrectly rejected in favor of the alternative hypothesis. If the confidence that the null hypothesis is not falsely rejected for each test is $1 - \alpha$, then the overall confidence that there are no false discoveries in *all* of the tests can be expressed as $(1 - \alpha)^m$ where m is the total number of tests conducted. For $\alpha = 0.05$, the overall confidence of one test is 95%, the overall confidence of five tests is 77%, and the overall confidence of fifty tests is 8%.

Consider then our comparison of 4000 random forest parameter tunings to the default tuning. At $\alpha = 0.05$, our overall confidence in no false discoveries for *all* of these test results is about 7.8×10^{-90} . To put that in perspective, imagine that the rules of the Daily 3 lottery were re-written so that instead of drawing three balls out of a cage, three *atoms* were drawn out of 1,000 gallon tank of water. Under these rules, the odds of winning the lottery would still be more than twice as good as the chance of no false discoveries in all 4,000 tests.

The most conservative and simplest strategy for dealing with this problem of multiple comparisons is the Bonferroni correction. This states that instead of rejecting the null hypothesis when the p-value is less than α , the null hypothesis should only be rejected in cases where the p-value is less than α/m where m is the total number of tests performed. [25] Under the Bonferroni correction, if 4,000 tests are performed with a desired α of 0.05, the threshold for significance is 0.0000125, which is much too conservative for the effects observed in this thesis.

Clearly, attempting to say with any level of confidence that no false discoveries were made due to the multiple comparisons problem in a study of this size is a fool's errand. In some studies, the presence of *any* false discoveries can have dire consequences, but in this study a few false discoveries are tolerable and even to be expected as long as their effects on the conclusions can be quantified and controlled. In order to proceed to a more practical means of limiting false discoveries, consider the confusion matrix for m tests of significance below.

Null-Hyp	Rejected	Not Rejected	Total
False	TP	FN	S
True	FP	TN	$m - S$
Total	R	$m - R$	m

Figure 3.3: *Confusion Matrix for m tests of significance*

In the context of Figure 3.3, assume m tests are performed at an individual level of significance of α . $E(x)$ will be used to represent the expected value of x and $P(x)$ will be used to represent the probability of x . The per-comparison error rate (PCER) is defined as $E\left(\frac{FP}{m}\right)$, the family-wise error rate (FWER) is defined as $P(FP > 0)$, and the false discovery rate (FDR) is defined as $E\left(\frac{FP}{R}\right)$.

Bonferroni correction attempts to hold $FP = 0$ by adjusting α of the individual tests such that the FWER is equal to a prescribed value, typically 0.05 or less. This typically results in a decrease in TP and an increase in FN when compared to less-conservative methods of handling the multiple comparison problem.

Many methods of FWER controls have been proposed which are less conservative than the Bonferroni correction. Several of these were attempted in the course of the present research, but they were all found to be too conservative to preserve some of the weaker signals observed given the number of tests performed. Most notably, Hochberg’s 1988 step-up procedure [14] was found to have sufficient power to detect the effects of parameter tuning on the Bernoulli Bayes learner, but was too conservative to yield positive results for parameter tuning on any of the other learners.

If some false positives can be tolerated, then the FWER can be abandoned entirely. If it is sufficient to claim that at least one TP exists, then a very simplistic approach involving PCER can be taken. If $\frac{R}{m} > \text{PCER}$, then at least one TP is expected to exist. If only one test is conducted then by definition, $\text{PCER} = \alpha$. For multiple tests, $\text{PCER} \leq \alpha$. [3] Therefore, by syllogism, at least one TP is expected whenever $\frac{R}{m} > \alpha$. This approach is effective if the objective is only to demonstrate the likely existence of at least one TP and $\frac{R}{m}$ is much greater than α , but does not make any statement about *how many* TP results exist. It may also be somewhat dubious when $\frac{R}{m}$ is close to α .

A more robust approach is to prescribe the FDR to an acceptable level. In this way, the expected number of true positive can be stated as $E(\text{TP}) = R(1 - \text{FDR})$. Benjamini and Hochberg provide a method of controlling FDR. [3] Their procedure involves sorting all alternative hypotheses, $H_1, H_2, \dots, H_m$, by their p-values $P_1, P_1, \dots, P_m$ from least to greatest. They then reject the null hypotheses in favor of $H_1, H_2, \dots, H_k$ where:

$$k \text{ is the largest } i \text{ for which } P_i \leq \frac{i}{m}q^* \quad (3.16)$$

and q^* is the target FDR. Benjamini and Hochberg prove that $\text{FDR} \leq q^*$ without assuming independence of the test statistics $P_1, P_1, \dots, P_m$. This procedure was ultimately used to determine the significance of the difference in performance between each alternative tuning and the default tuning. The FDR was limited using $q^* = 0.05$.

3.5.3 Comparing Current and Forward Learning Performance

As discussed in Section 1.2, an increase in current-version performance may be an interesting academic result, but doesn't necessarily mean that parameter tuning is helpful in a real-world scenario. In order to make business sense, parameters which preformed well when evaluated on current versions must also preform well when extrapolating from the current versions to future versions. For this, we must analyze the the "forward" step results in an attempt to answer RQ 2.

In the same way that a tuning can be labeled as "better", "worse", or "not better or worse" than the default for current-version performance, the same labels can be applied to that turning's forward performance. As with the current-version significance tests, the Wilcoxon test is used to compare a tuning's g metric when trained on the current version and tested on the next version to the default tuning's g metric using the current version training data and future version testing data. Significance or nonsignificance is then determined using the Benjamini-Hochberg procedure.

By labeling each parameter with respect to both it's current version performance and it's forward performance, it can be counted as one of nine scenarios as shown in Table 3.7. The scenario that represents the best solutions to the tuning problem is the combination of superior performance in the current-version learning and superior performance in forward learning. This scenario is the bottom right corner of Table 3.7.

All other scenarios in the bottom row represent cases in which a tuning that would appear to preform well when tested within the current version does not perform as well when tested in the forward version. In a a real-world situation, where the forward performance cannot be directly evaluated and must be inferred from the current version performance, the three scenarios in the bottom row are indistinguishable from one another. If results from all of these tunings were to be amalgamated into one set of predictions, some of the individual tunings may help or harm the value of that set of predictions as compared to the predictions

Table 3.7: *Version-Forward Learning Outcomes*

	forward<default	forward=default	forward>default
current<default	non-useful tunings		missed tunings
current=default			missed tunings
current>default	harmful tunings	neutral tunings	helpful tunings

which would be produced by the default tuning. For this reason, the situations represented in the “current>default” row can be considered helpful, harmful, or neutral depending on their forward performance.

The other interesting scenarios in Table 3.7 are the ones in which forward performance was good, but current version performance is less than or equivalent to the default. These represent “missed” tunings which would achieve positive outcomes in the forward scenario, but these outcomes would not have been predicated by their current version performance. Missed tunings would not necessarily harm the accuracy of an amalgamated tuned forward result, but if identified from their current version performance, they could contribute to it.

3.5.4 Splitting Helpful Hairs with Effect Size

For the case where a tuning’s performance is determined to be significantly better than that of the default tuning, one might wonder if the size of that effect has any bearing on forward performance. In other words, are forward predictions any better from tunings that performed *vastly* better than defaults in current evaluation compared to tunings that performed *marginally* better in current evaluation?

Answering this question requires a measure of effect size. For the Wilcoxon test, effect size can be approximated to the same scale as Pearson’s correlation coefficient, r by dividing the Z-value by the square root of the total number of observations. For a paired test, the total number of observation is twice the number of pairs. [26] In this case, cross-validations with a sample size of 25 are being compared, so the total number of observations is 50. The

Table 3.8: *Cohen '88 Effect Sizes*

Effect Size	"Small"	"Medium"	"Large"
r range	0.1-0.3	0.3-0.5	0.5+

Z-values in this case are tabulated using the inverse survival function `scipy.stats.norm.isf( $\alpha$ )` which returns a Z-score based on an assumption of normality and an alpha value.

$$r = \frac{Z}{\sqrt{N}} = \frac{Z}{\sqrt{25 \times 2}} = \frac{Z}{\sqrt{50}} \quad (3.17)$$

After the Z-score have been calculated from the α value with the normality assumption and the correlation coefficient, r , has been approximated from the Z-score and the sample size, some meaningful interpretations of r must be applied. Here, we will use the “small”, “medium”, and “large” effect sizes and their respective thresholds as suggested by Cohen '88 [9] which are listed in Table 3.8. It should be noted that the lower limit for a “small” effect is not applicable in this context as the effect size is only considered for items which are determined to be statistically significant. At $\alpha = 0.05$, the corresponding effect size is $r = 0.23$, but due to the application of the Benjamini-Hochberg procedure, an r as low as 0.23 is rather unlikely.

By applying these labels, the bottom row of Table 3.7 can be split into three rows, shown in Table 3.9. Note that this table *only* contains results which performed significantly superior to the default parameters in current evaluation. If the effect size of current version performance matters in predicting forward performance, we would expect the “Large Effect” row to have the largest percentage of “helpful” results and the smallest percentage of “harmful” results of any of the rows.

Table 3.9: *Version-Forward Learning Outcomes by Effect Size*

	forward<default	forward=default	forward>default
Small Effect	harmful	neutral	helpful
Medium Effect	harmful	neutral	helpful
Large Effect	harmful	neutral	helpful

Chapter 4

Results

The results for each machine learning scheme will be presented in this chapter starting with the learner which produces the least evidence for the efficacy of parameter tuning and progressing in order of strength of evidence to the learner which produced the most positive result.

4.1 Multinomial Naive Bayes: a Negative Outcome

By classifying each tuning of Multinomial NB which was explored into “worse than”, “better than”, or “not better or worse than” the default tuning and counting the results, we can see that very few of Multinomial NB’s parameter tunings ever performed significantly better than default on the datasets tested. 82 parameter tunings were tested in the current version evaluation case on 36 datasets yielding $82 * 36 = 2952$ individual current parameter-specific results. Of these 2952 results, only 16 (0.5%) were determined to be significantly better than default. Strong evidence that Multinomial NB benefits from parameter tuning was not found with the datasets selected and the set of parameters tested in this study.

Table 4.1: *Version-Forward Learning Outcomes: Multinomial Naive Bayes Total Counts*

	forward<default	forward=default	forward>default
current<default	2 (2%)	41 (48%)	41 (48%)
current=default	41 (1%)	1925 (93%)	82 (4%)
current>default	0 (0%)	0 (0%)	0 (0%)

If we explore the current-version Multinomial NB results on a per-dataset basis, we find that the 16 positive cases came entirely from two datasets: 13 from *xalan-2.7* and 3 from *xerces-2.4*. These correspond to 16% and 4% of the tunings tested on these sets.

In the forward learning case, parameter tuning of Multinomial NB fared slightly better. Out of the 2132 trials forward performed, 123 of them (6%) were found to be significantly better than default. At first glance, this result seems somewhat positive, but closer examination reveals that none of the parameter tunings which succeeded in the forward case were predicated by success in the current version case. This is demonstrated by the summed results in Table 4.1. The green cell of the table with a value of 0 represents the number of tunings which succeeded in both the current and forward cases.

An examination of the box plots in Figure 4.1 reveals that for most datasets, there was an extremely narrow range of performance across all the parameter tunings tried in both current and forward evaluation. It appears from these results that parameter selection has little effect on the performance of this implementation of the multinomial naive Bayes classifier.

(a) Multinomial Naive Bayes Current-Version Mean g

(b) Multinomial Naive Bayes Forward Mean g

Figure 4.1: Multinomial Naive Bayes Mean g Box Plots. The g value for each tuning is averaged across all 25 cross-validation sub-trials. The default tuning's mean performance is indicated by a green diamond.

4.2 Logistic Regression: a Mixed Outcome

For logistic regression, 2790 individual parameter tunings were tested on each of the 36 datasets. Of these 100,440 total current-version trials, 1.3% were found to be significantly better than default. Only 9 of the 36 datasets had at least one tuning that was found to perform better than default. For 15 of the 36 datasets, there were no tunings that were found to perform better *or* worse than the default. It appears current version tuning can effect logistic regression’s performance on some but not all of the datasets tested.

One might reasonably be inclined to ask “Does the existence of tunings which out-perform the defaults indicate that parameter tuning works, or just that the defaults are no good?” as in RQ4. In other words, “Should the default just be changed to the top-performing tuning?” This can be answered by examining a histogram of successful parameter tunings. Figure 4.2 shows the number of datasets for which each tuning is considered to be significantly better than default. It should be noted that even the best performing tuning is only considered to be better than default on 2 of the 36 datasets. Recall that the benefits from tuning were found in a total of 9 datasets. This implies that the parameters which work best for one dataset do not necessarily work best for all datasets. In other words, selecting a “better” default parameter tuning will not lead to a ubiquitous increase in performance.

In the forward learning case, the logistic regression results do not present particularly positive evidence for the efficacy of parameter tuning. While about 5.8% of the 72,540 forward trials were determined to represent performance superior to default tunings, they do not seem to be strongly correlated with superior performance in the current evaluation regime, at least not in aggregate. Table 4.2 demonstrates this. Note that if a given tuning performed significantly better than the default tuning in current evaluation, it only had a 1.7% chance of performing better than defaults in forward evaluation. Conversely, the overwhelming majority (99.6%) of tunings which fared better than default in forward evaluation were not significantly better than default in current evaluation.

Figure 4.2: *Histogram of Logistic Regression Current-Version Successful Parameter Tunings. The vertical axis indicates the number of datasets for which a given parameter tuning is determined to be significantly better than the default tuning.*

When the logistic regression results are examined on a per-project basis, it can be seen that one software project, *Xalan*, accounts for all 14 of the cases shown in the bottom right cell of Table 4.2. This cell represents situations in which superior forward performance was predicated by superior current-version performance. Eleven of these cases came from forward learning from *xalan-2.5* to *xalan-2.6* and three came from forward learning from *xalan-2.6* to *xalan-2.7*.

In the *Xalan*-only aggregates of Table 4.3, signs of a weak correlation can be seen. The majority of the bottom-row cases, where performance is better than default tunings in current evaluation, still result in mediocre performance in forward evaluation, but this time 46% of these bottom row cases result in superior forward performance. While a 46% overlap between superior current-version performance and superior forward performance in one project out

Table 4.2: *Version-Forward Learning Outcomes: Logistic Regression Total Counts*

	forward<default	forward=default	forward>default
current<default	7144(65%)	3065(27%)	755(6%)
current=default	3704(6%)	53558(88%)	3481(5%)
current>default	0(0%)	819(98%)	14(1%)

of 10 is hardly an earth-shattering success, it at least provides a hint that for some datasets, parameter tuning of a logistic regression classifier may be worthwhile. This hint of success on a subset of the projects studied should not overshadow the aggregate results of Table 4.2 which show that on the whole, logistic regression tuning may not necessarily be worthwhile.

Table 4.3: *Version-Forward Learning Outcomes: Logistic Regression Xalan counts*

	forward<default	forward=default	forward>default
current<default	1465(75%)	400(20%)	69(3%)
current=default	1728(26%)	2855(44%)	1823(28%)
current>default	0(0%)	16(53%)	14(46%)

Some sense of why tuning was so ineffective can be gained by examining the boxplots of Figure 4.3. Here we can see that for most datasets, the default parameters (green diamonds) perform quite well relative to all other tunings tested. In a few cases, this is not true, but in most places where there is much room for improvement in the forward learning case such as *camel-1.0*, *ivy-1.4*, *velocity-1.4*, and *xerces-1.3* the default tunings performed about as well in current version evaluation as the best tunings tested. One notable exception to this is *xerces-1.2* where the boxplots would seem to indicate that there was much room for improvement in both current and forward evaluation but under the Benjamini-Hochberg procedure, none of the forward learning trials were found to be significantly different from the default.

(a) Logistic Regression Current-Version Mean g

(b) Logistic Regression Forward Mean g

Figure 4.3: *Logistic Regression Mean g Box Plots.* The g value for each tuning is averaged across all 25 cross-validation sub-trials. The default tuning's mean performance is indicated by a green diamond.

4.3 Random Forest: a Positive Outcome

For the Random Forest learner in the current-version evaluation scheme, there were 16 of the 36 datasets for which none of the 4000 parameter tunings tested were determined to be significantly better than default. This leaves 20 of the 36 datasets for which we can confidently say that some parameters out-performed the default parameters. These current-version successful tunings constitute 12% of all current-version trials and 22% of all trials on the 20 datasets where any tuning was found to be significantly better than default.

As with the logistic regression results, one might ask “Does the existence of tunings which out-perform the defaults indicate that parameter tuning works, or just that the defaults are no good? Should the default just be changed to the top-performing tuning?” The answer lies in Figure 4.4 which shows the number of datasets for which each tuning is considered to be significantly better than default. It should be noted that even the best performing tuning is considered to be better than default on 14 of the 36 datasets. This means that there were six datasets for which *some* tuning performed significantly better than default, but the overall best tuning did not. Furthermore, almost all of the 4000 tunings tested performed better than the default tuning on *at least one* dataset. These two facts imply that the parameters which work best for one dataset do not necessarily work best for all datasets and that a change of defaults is unlikely to lead to a ubiquitous increase in performance.

When the forward-learning case is considered, the Random Forest results present an even stronger case for the efficacy of parameter tuning. By examining the matrix of current and forward performance categories defined in Table 3.7, there is a clear correlation between the success of tunings in the current learning scenario and their likelihood of success in forward learning when trained on the same dataset. Table 4.4 shows that a tuning which does not preform significantly better or worse than default on a set’s current-version evaluation only had a 5% chance of performing significantly better than default in forward evaluation. This chance rose to 47% if the tuning preformed significantly better than default on evaluations

Figure 4.4: *Histogram of Random Forest Current-Version Successful Parameter Tunings. The vertical axis indicates the number of datasets for which a given parameter tuning is determined to be significantly better than the default tuning.*

within the current version. This is even more impressive when coupled with the fact that for these tunings, the chance of performing significantly worse than default in forward evaluation is less than 1%.

Table 4.4: *Version-Forward Learning Outcomes: Random Forest Total Counts*

	forward<default	forward=default	forward>default
current<default	14363(62%)	8474(37%)	0(0%)
current=default	7688(11%)	57460(83%)	3593(5%)
current>default	32(0%)	6536(52%)	5874(47%)

This pattern becomes even more clear when the the “current>default” case in the bottom row of Table4.4 is split by effect size. All rows in Table 4.5 represent cases for which tunings preform significantly better than defaults. Each row represents a different effect size for the difference between the performance of a given tuning and the performance of

the default tuning. Clearly, for all of these cases, performances significantly inferior to defaults in the forward case are unlikely. It is also apparent that larger effect sizes lead to an increased likelihood of significantly superior performance in the forward case. In this case, if a parameter could be shown to significantly outperform the default parameters with a large effect size in evaluation within the current version, it was 95% likely to outperform the defaults in forward evaluation.

Table 4.5: *Version-Forward Learning Outcomes: Random Forest Total Counts*

	forward<default	forward=default	forward>default
Small Effect	4	849	428(33%)
Medium Effect	28	5630	4337(43%)
Large Effect	0	57	1109(95%)

When the aggregate results presented in Table 4.4 are broken down by software project, it becomes apparent that this effect is entirely due to the *ant*, *camel*, *xalan*, and *xerces* projects. On *ivy*, *lucene*, and *velocity* no tunings were found to be significantly superior to default in either current or forward evaluation. On *jedit*, *log4j*, and *Synapse* some successful tunings were found in current evaluation, but less than 1% of tunings were found to be successful in forward evaluation.

This project-level success breakdown is also reflected in some ways by the boxplots in Figure 4.5. In *ant*, *camel*, *xalan*, and *xerces*, we generally see wide margins for improvement in both current and forward performance. In *ivy*, *jedit*, *log4j*, *lucene*, *synapse*, and *velocity* we generally see much narrower margins for improvement in forward evaluation.

In regards to the effect size trend seen in table 4.5, the trend of large effect sizes producing more superior forward results than medium effect sizes held true in the project-level aggregations for all applicable individual projects. There were two cases where small effect sizes seemed to produce better results than medium or large effect sizes, but this may be due to small sample sizes (15 and 5) of the “small effect” category .

(a) Random Forest Current-Version Mean g

(b) Random Forest Forward Mean g

Figure 4.5: Random Forest Mean g Box Plots. The g value for each tuning is averaged across all 25 cross-validation sub-trials. The default tuning's mean performance is indicated by a green diamond.

4.4 Bernoulli Naive Bayes: a Very Positive Outcome

In current-version evaluation, at least 4.6% of the parameter tunings tried on *every dataset* were found to be significantly better than the default tunings. Examination of the parameter success histogram in Figure 4.6 reveals that the top-performing parameter was successful on 33 of the 36 datasets. Compared to the other learners studied in this thesis, Bernoulli Bayes has the highest rate of overall current version tuning success and also the highest rate of success of the tested parameters as evidenced by the histogram in Figure 4.6. This seems to imply that parameter tuning is highly effective on this learner, but that the parameters are only somewhat dataset-dependent. Picking a better set of default values would go a long way towards improving performance here.

Figure 4.6: *Histogram of Bernoulli Bayes Current-Version Successful Parameter Tunings. The vertical axis indicates the number of datasets for which a given parameter tuning is determined to be significantly better than the default tuning.*

The forward learning aggregate result matrix for the Bernoulli Bayes classifier also shows the most positive result of this study. As can be seen in the bottom row of Table 4.6, a tuning that performed significantly better than default in current evaluation had an 85% chance of performing better than default in forward evaluation and only a 2% chance of performing worse! Furthermore, a clear correlation is evidenced by examination of the cells on the diagonal of Table 4.6. In each row, the element which is part of the diagonal has the row’s largest plurality of cases. When broken down on a per-dataset basis, it was found to be true in 24 of 26 cases that the majority of tunings which performed better than default in current evaluation also did so in forward evaluation. The exceptions were *velocity-1.4* with 45% and *jedit-4.2* with 6% of good current-version tunings being helpful.

Table 4.6: *Version-Forward Learning Outcomes: Bernoulli Bayes Total Counts*

	forward<default	forward=default	forward>default
current<default	9402(77%)	2049(16%)	717(5%)
current=default	2687(21%)	5588(45%)	4119(33%)
current>default	474(2%)	2413(11%)	17323(85%)

The success of forward parameter tuning of Bernoulli Bayes is further accentuated by the effect-size breakdown in Table 4.7. Tunings with larger effect sizes in current evaluation were more numerous and more likely to be successful in forward evaluation.

Table 4.7: *Version-Forward Learning Outcomes: Bernoulli Bayes Total Counts*

	forward<default	forward=default	forward>default
Small Effect	16	136	381(71%)
Medium Effect	73	1017	4213(79%)
Large Effect	385	1260	12729(88%)

The boxplots of individual dataset performance in Figure 4.7 show general low performance of the default parameters which is consistent with the histogram in Figure 4.6. It also shows a wide range of performance across the parameter space on most datasets which would imply that performance is highly dependent on choosing the right parameter values. Unfortunately, these boxplots do not convey any clear reason as to why *jedit-4.2* and *velocity-1.4* performed more poorly than other sets in tuning for forward learning.

(a) Bernoulli Naive Bayes Current-Version Mean g

(b) Bernoulli Naive Bayes Forward Mean g

Figure 4.7: *Bernoulli Naive Bayes Mean g Box Plots.* The g value for each tuning is averaged across all 25 cross-validation sub-trials. The default tuning's mean performance is indicated by a green diamond.

Chapter 5

Threats to Validity

5.1 Construct Validity

An assessment of construct validity entails answering the following question: “Does the study accurately represent and evaluate the constructs (aka variables) under investigation?” [10] In this study the main construct under investigation is the relationship between the performance of default tunings and the performance of many non-default tunings. The performance measures of the non-default tunings are somewhat protected by the law of large numbers. If for some reason the performance of a few non-default tunings somehow become skewed, it will likely not have a significant effect on the overall outcome due the aggregation of results from many tunings. The default tunings however form the yardstick against which everything else is measured. If the default tuning scores anomalously low on a particular dataset, all other tunings will seem to perform very well in comparison. Conversely, an anomalously high-scoring default tuning will result in a negative skew to the perceived performance of all other tunings on that dataset.

Some of the factors which could contribute to such a skew are better covered in the following section as internal validity issues. The main concern for the construct validity

of this study is the use of the performance measure g . The rationale for the selection of g as the chief measure of performance is covered in Section 3.2.2, but g is not the only valuable performance measure. g is a harmonic mean of pD and $1-pF$, but depending on the preferences of a user, a high pD may be more valuable than a low pF or vice versa. If a different synthetic metric were used for scoring, the relative performances of different tunings could change substantially. It sounds plausible to assume that if a learner could be tuned for better g scores, it could also be tuned for better scores on other metrics, but that assumption is not tested in this thesis.

5.2 Internal Validity

An assessment of internal validity involves an analysis of “whether a cause-and-effect relationship between variables can be determined.” [10] For the positive cases, we must ask if tuning *actually* led to better-than-default performance. As mentioned in the previous section, a default which scores anomalously high or low compared to its expected performance would seriously affect the results.

One such threat is that of a quirk of the combination of learner, tuning, and dataset which yields results not reflective of general performance. By subsampling the original dataset, such highly data-dependent quirks are smoothed and the results of a cross-validated trial should in principle reflect the expected performance of a particular tuning on a particular dataset.

Another threat is the possibility of random variations in performance due to the inner workings of the machine learners. For the Bayesian learners and logistic regression, this concern is a non-issue as these algorithms are deterministic. Repeated runs with the same parameters and data will yield identical results. Random forest is not deterministic in that the inner workings of the algorithm rely on heavily on random choices. This randomness could *technically* be removed by using the same random seed at for each trial run, but

this would be moot as one could just as easily argue that setting the random seed to any particular value for all trials introduces bias and that the results may not hold for a different seed value. The effects of this random variation can be controlled through repetition. Due to CPU time constraints, no additional repetition of trials of the random forest learner was conducted, but some degree of repetition is implicit in the cross-validation trials. Recall that comparisons are not being made on the basis of individual trials, but from applying the paired Wilcoxon test to 25 pairs of independent cross-validation sub-trials.

For the negative cases, in which parameter tuning was not found to be effective, the same arguments against validity still apply, but additionally one can ask “Were enough parameter tunings tried?” As discussed in Section 3.4.1, there is no good argument as to the merits of testing parameter values of $\{2, 4, 6, 8\}$ rather than $\{2.1, 4.1, 6.1, 8.1\}$. For numerically valued parameters, there are infinite values which could potentially be tested, but a finite number were selected for testing in this study. It is possible that the most ideal parameters “slipped through the cracks” in our grid search and were missed. This problem is unavoidable in this type of study and can only be mitigated by testing as many discrete values as practical given time restrictions. For lists of the specific parameter values selected for evaluation, see Tables 3.3 through 3.6.

5.3 Conclusion Validity

An assessment of conclusion validity involves evaluation of “the accuracy and strength of the data analyses and statistical conclusions that are drawn from a study.” [10] The main threat to the statistical validity of the conclusions in this study is the strength of the cross-validation scheme used. Leave-one-out validation is the gold standard in machine learning, but is so computationally intensive that it is rarely used in large studies. Cross-validation provides a close proxy, but the strength of conclusions which can be drawn from a cross-validated study are directly linked to the number of individual sub-trials within the cross-

validation (sub-trials=folds×repeats). In this case, a 5x5 cross-validation was used with 25 individual sub-trials. This is greater than the minimum sample size of 20 which is generally recommended for use of the Wilcoxon test.

In other studies, the use of cross-validation is often linked with validity threats stemming from the re-sampling of data between runs of each learning scheme. The experimental design of this study specifically circumvents this concern by splitting the data for cross-validation prior to the learning trials and then using the same splits for each trial. Another concern raised in other studies is the skew to the positive/negative ratio that can occur due to the random nature of cross-validation. This concern was circumvented by splitting the data in a class-stratified manner.

5.4 External Validity

An assessment of external validity involves “making a systematic evaluation of the accuracy and strength of the ability to generalize the results beyond the current study.” [10] This study focuses on a narrow slice of software defect prediction. For data, we only consider open-source Java projects which have been analyzed using the CKJM extended metrics suite and incorporated into the PROMISE repository. For learners, we only considered one package’s implementation of four common machine learning schemes. While we cannot make any explicit claim to the generalizability of results beyond these particular learners and datasets, we have no reason *not* to expect similar results if this study were to be replicated in other domains.

Chapter 6

Conclusions and Suggestions for Implimentation

The overall conclusion to this line of research is that parameter tuning can make a difference in the practical performance of some machine learners when data mining in a software defect prediction context. Parameter tuning doesn't have a positive effect for all machine learning schemes so in practice, trials should be conducted with the learner of choice before a "tuned" result is given credence over the result yielded by the learner with default parameters. Furthermore, in cases where tuning works *in general*, it doesn't necessarily mean that each individual "tuned" result is better than the result yielded by default parameters, so some means of aggregation is suggested to pool the results from the best tunings rather than relying on a single tuning. These caveats and suggestions for practical implementation of parameter tuning are discussed in the following sections.

6.1 Current-Version Tuning

It is clear from the results that parameter tuning within the current version can produce results that are superior to those yielded by the default parameters. This effect is not ubiquitous and is highly dependent on the learning scheme and the dataset. While this effect was not observed for the multinomial Bayes classifier, tunings that preformed significantly better than the default tunings were found for logistic regression in 25% of datasets, for random forest in 55% of datasets, and for Bernoulli Bayes in 100% of datasets.

This means that for some learners, the classification performance is highly dependent on the parameters which used. This could explain conflicting results in literature where the same learning scheme can have a wide range of reported performance values on the same data. In particular, differences in parameter tuning practices may help account for Shepperd, Bowes, and Hall’s curious and embarrassing meta-analysis result: that the most significant factor affecting machine learning performance in software defect prediction literature is researcher affiliation. [32]

In the least negative interpretation, parameter tuning practices may differ from place to place based on shared wisdom. Some groups of researchers may only use defaults tunings, while others may apply parameter tuning techniques to all learners and report only the best-case results. In a more damning interpretation, researchers could be reporting the best-case tuning results from their favorite learning schemes and the default tuning results from competing learners.

Regardless of the level or cause of researcher bias present in current literature, this within-version result demonstrates that one must take any reported machine learner performance measure with a grain of salt unless the parameters used and the method of their selection is clearly reported. One suggestion for handling this issue in future publications is to chose the parameter tuning for each learner which produces the best overall result among all datasets in the study. Reporting the results from this tuning on each dataset represents a form of

course-grained parameter tuning which should be easily replicable, especially if the actual parameter values used are published.

6.2 Forward Tuning

Whereas the current-version tuning results are mostly applicable to academic discussion of the merits of machine learners, the forward tuning problem is rooted in practical applications of machine learning on data which has not yet been assigned a class. In real life, machine learning is only useful as long as you can make predictions about data which has not yet been scored. In the case of software defect prediction, that means predicting bugs in modules that have not yet been fully tested. Similarly, parameter tuning is only valuable if it can be used to improve results without prior knowledge of the class values of the data for which predictions are being made.

For two of the four learners studied, random forest and Bernoulli Bayes, it is clear that forward evaluation performance can be improved through parameter tuning on the most recent version for which class labels are available. This finding is significant to practical applications because it means that a parameter tuning trials may be conducted on a recent software version and the best tunings from those trials can be used to improve the classification results for a software version that is currently in development.

The effect size tables (Tables 4.5, 4.7) also indicate that the parameter tunings which perform better than defaults by a large margin in within-version evaluation stand a greater chance of superior performance in forward evaluation. In other words, tunings which do exceptionally well in trials on the latest classified version are much more likely to succeed in forward learning than those that are only marginally better than default.

6.3 Pre-Trials: Does Tuning Work for My Learner?

While the conclusion up until this point has taken a positive tone towards the efficacy of parameter tuning, it is also important to remember that parameter tuning wasn't found to be effective in *all* cases. For real-world applications of parameter tuning, it is important to test the efficacy of tuning the target learner before trusting tuned parameters over the default parameters. Put simply, parameter tuning doesn't always work, so it's best to run some trials first.

When conducting these pre-trials, it is best to use data that is a close proxy to the data which will be evaluated. For example, a developer who has data from versions 1 through 5 of a software project and is in the process of developing version 6 may want to conduct current and forward tuning trials similar to the ones in this study on versions 1-2, 2-3, 3-4, and 4-5 as an indication of the effect of tuning on forward evaluation. If tunings which perform better than default within the lower version of these trials tend to perform better than default when applied in the forward learning scenario, then it is likely that parameter tuning can produce better-than-default results when applied to versions 5-6. If this is the case, the developer should conduct within-version tuning trials on version 5 and then use the best performing parameters to instantiate a pool of learners trained on version 5 which can make predictions about version 6.

In the case of new software projects where data from several versions of that project's history is not available, some insight may be gleaned by applying intended metrics suite to several open-source projects and using them as proxy. It is important to realize however that the actual tunings which perform well on this other data may not correlate with the tuning that perform well on the target project, so at least one historical version should be available for best results. In any case, it is also important to understand that efficacy in pre-trials *does not guarantee* efficacy in practice. These pre-trials can only serve as an *indication* as to whether or not tuning is likely to be advantageous.

6.4 Tuned Learner Pooling

Even in the best case, not every tuning produces positive results. For the learners with positive forward tuning outcomes (random forest, Bernoulli Bayes) the *majority* of tunings which performed significantly better in current evaluation performed *at least as well* as the default tunings, but *some* performed significantly worse. For random forest, the largest plurality (52%) of the tunings which performed well in current evaluation did not perform significantly different from defaults in forward evaluation. Since the practical application of software defect prediction ultimately boils down to a set of yes-or-no predictions about software modules, how does one utilize the advantage of parameter tuning to produce a set of predictions that are more accurate than those yielded by default parameters?

The answer lies in the pooling of predictions from multiple learners with multiple tunings. This generalized form of this process is called *ensemble learning* and is a topic of current research in machine learning literature. Ensemble learning generally involves taking predictions from several classifiers, assigning each predictor a confidence level, and producing a set of consensus predictions based on the predictions of each classifier and its associated confidence values.

In its most simplistic form, ensemble learning could be applied here by deciding the consensus predictions by simple majority of predictions generated by the application of each better-than-default tuning. This would be equivalent to an ensemble learner in which each constituent learner uses a different parameter tuning and has the same confidence level. A more clever way of doing this may be to apply some more advanced ensemble learning technique with each learner's confidence value determined by its performance characteristics in current version trials. Selection of an ensemble method for intelligently pooling results from selected tunings is an interesting topic that will be left future research.

6.5 Summary of Conclusions:

Research Questions Revisited

1) Can within-version parameter tuning lead to increased classification performance on software defect data when compared to default parameters?

In some cases, yes, for logistic regression, random forest, and Bernoulli Bayes we can answer this question in the affirmative. It should be noted however, that this effect may be dataset-dependent. Also note that we found no evidence supporting this with the Multinomial Bayes learner.

2) Is parameter tuning effective when the parameters are tuned and the classifier is trained on data from one version of a software project but tested on the next version?

In some cases, yes. Of the three learners for which RQ1 can be answered in the affirmative, two of them, random forest and Bernoulli Bayes yielded evidence of a correlation between the success of tunings within a software version and their success in forward prediction of the next dataset. It should be noted that good evidence of this correlation was not found for most datasets with the logistic regression classifier.

3) Is the extent to which a parameter tuning out-performs the default tuning on data from one software version indicative of it's likelihood of success in future versions?

Yes. In both cases for which there was any indication that forward parameter tuning could be successful, (random forest and Bernoulli Bayes) there was also indication that tunings which out-performed the default in within-version evaluation with a large effect size were more likely to also excel in forward evaluation than those with a smaller effect size.

4) Is there a set of parameter tunings that will increase learning performance ubiquitously in the domain of software defect prediction?

No, not ubiquitously, at least not that could be found in this study. For one of the four learners studied, performance could have been improved on the majority of the datasets by “changing the defaults”, but not on all datasets. For the other learners, no single tuning performed better than defaults on more than 40% of the datasets.

References

- [1] Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *Search Based Software Engineering*, pages 33–47. Springer, 2011.
- [2] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4–17, Jan 2002.
- [3] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.
- [4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.
- [5] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.
- [6] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [7] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.
- [8] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

- [9] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences (2nd Edition)*. Routledge, 2 edition, July 1988.
- [10] Stephen F Davis and William Buskist. *21st Century Psychology: a Reference Handbook*, volume 1. Sage, 2008.
- [11] Agoston E Eiben and Selmar K Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.
- [12] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, Nov 2012.
- [13] Brian Henderson-Sellers. *Object-oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [14] Yosef Hochberg. A sharper bonferroni procedure for multiple tests of significance. *Biometrika*, 75(4):800–802, 1988.
- [15] Liangxiao Jiang, H. Zhang, and Zhihua Cai. A novel bayes model: Hidden naive bayes. *Knowledge and Data Engineering, IEEE Transactions on*, 21(10):1361–1371, Oct 2009.
- [16] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2015-03-12].
- [17] Marian Jureczko and Diomidis Spinellis. Using object-oriented design metrics to predict software defects. In *Models and Methodology of System Dependability. Proceedings of RELCOMEX 2010: Fifth International Conference on Dependability of Computer Systems DepCoS*, Monographs of System Dependability, pages 69–81, Wrocław, Poland, 2010. Oficyna Wydawnicza Politechniki Wrocławskiej.
- [18] Robert Martin. Oo design quality metrics. *An analysis of dependencies*, 1994.
- [19] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

- [20] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". *Software Engineering, IEEE Transactions on*, 33(9):637–640, Sept 2007.
- [21] Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 47–54, New York, NY, USA, 2008. ACM.
- [22] Jelber Sayyad Tim Menzies Bora Caglayan Zhimin He Ekrem Kocaguneli Joe Krall Tim Menzies Fayola Peters Burak Turhan Carter Pape Mike Chapman, Justin DiStefano and Corbin Steele. *tera-promise*, 2014.
- [23] Thomas P. Minka. A comparison of numerical optimizers for logistic regression. Technical report, 2003.
- [24] Andrew Y Ng. Feature selection, l_1 vs. l_2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.
- [25] G.R. Norman and D.L. Streiner. *Biostatistics: The Bare Essentials*. Pmph USA Ltd Series. B.C. Decker, 2007.
- [26] J. Pallant. *SPSS Survival Manual: A Step by Step Guide to Data Analysis Using SPSS for Windows (Version 15)*. McGraw-Hill Companies, Incorporated, 2007.
- [27] Vasil Papakroni. Data carving: Identifying and removing irrelevancies in the data. Master's thesis, West Virginia University, 2013.
- [28] 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.

- [29] Peter Peduzzi, John Concato, Elizabeth Kemper, Theodore R Holford, and Alvan R Feinstein. A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology*, 49(12):1373–1379, 1996.
- [30] Judita Preiss. Probabilistic word sense disambiguation. *Computer Speech Language*, 18(3):319 – 337, 2004. Word Sense Disambiguation.
- [31] Denise Rey and Markus Neuhuser. Wilcoxon-signed-rank test. In Miodrag Lovric, editor, *International Encyclopedia of Statistical Science*, pages 1658–1659. Springer Berlin Heidelberg, 2014.
- [32] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *Software Engineering, IEEE Transactions on*, 40(6):603–616, June 2014.
- [33] Selmar K Smit and Agoston E Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *Evolutionary Computation, 2009. CEC’09. IEEE Congress on*, pages 399–406. IEEE, 2009.
- [34] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. An empirical study on object-oriented metrics. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 242–249, 1999.
- [35] Vangelis Metsis Telecommunications and Vangelis Metsis. Spam filtering with naive bayes – which naive bayes? In *Third Conference on Email and Anti-Spam (CEAS*, 2006.
- [36] Hsiang-Fu Yu, Fang-Lan Huang, and Chih-Jen Lin. Dual coordinate descent methods for logistic regression and maximum entropy models. *Machine Learning*, 85(1-2):41–75, 2011.
- [37] Harry Zhang. The optimality of naive bayes. *AA*, 1(2):3, 2004.

Appendix A

Multinomial Naive Bayes Forward Results By Project

SciKit-Learn_Multinomial_NB ant Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	41 (12%)	283 (86%)	4 (1%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB camel Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	41 (100%)
cur = def	0 (0%)	205 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB ivy Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	164 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB jedit Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	328 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB log4j Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	41 (100%)	0 (0%)
cur = def	0 (0%)	86 (69%)	37 (30%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB lucene Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	164 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB synapse Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	2 (100%)	0 (0%)	0 (0%)
cur = def	0 (0%)	162 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB velocity Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	164 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB xalan Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	246 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB xerces Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	123 (75%)	41 (25%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

Appendix B

Logistic Regression Forward Results By Project

SciKit-Learn_Logistic_Regression ant Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	11160 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression camel Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	7 (1%)	404 (98%)	0 (0%)
cur = def	24 (0%)	7935 (99%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression ivy Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	869 (90%)	95 (9%)	0 (0%)
cur = def	0 (0%)	3686 (79%)	930 (20%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression jedit Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	2156 (67%)	1054 (32%)	0 (0%)
cur = def	222 (2%)	7428 (94%)	173 (2%)
cur > def (all)	0 (0%)	127 (100%)	0 (0%)
cur > def (S)	0 (0%)	37 (100%)	0 (0%)
cur > def (M)	0 (0%)	90 (100%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression log4j Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	627 (100%)	0 (0%)
cur = def	0 (0%)	4953 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression lucene Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1623 (86%)	243 (13%)	0 (0%)
cur = def	28 (0%)	3073 (90%)	276 (8%)
cur > def (all)	0 (0%)	337 (100%)	0 (0%)
cur > def (S)	0 (0%)	61 (100%)	0 (0%)
cur > def (M)	0 (0%)	271 (100%)	0 (0%)
cur > def (L)	0 (0%)	5 (100%)	0 (0%)

SciKit-Learn_Logistic_Regression synapse Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	991 (97%)	29 (2%)	0 (0%)
cur = def	27 (0%)	4523 (99%)	10 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression velocity Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	31 (3%)	213 (22%)	686 (73%)
cur = def	1612 (34%)	2762 (59%)	269 (5%)
cur > def (all)	0 (0%)	7 (100%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	7 (100%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression xalan Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1467 (75%)	400 (20%)	69 (3%)
cur = def	1729 (26%)	2852 (44%)	1823 (28%)
cur > def (all)	0 (0%)	16 (53%)	14 (46%)
cur > def (S)	0 (0%)	0 (0%)	3 (100%)
cur > def (M)	0 (0%)	16 (59%)	11 (40%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression xerces Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	62 (1%)	5186 (98%)	0 (0%)
cur > def (all)	0 (0%)	332 (100%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	332 (100%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

Appendix C

Random Forest Forward Results By Project

SciKit-Learn_Random_Forest ant Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1792 (46%)	2056 (53%)	0 (0%)
cur = def	178 (2%)	7748 (91%)	539 (6%)
cur > def (all)	0 (0%)	249 (6%)	3438 (93%)
cur > def (S)	0 (0%)	77 (17%)	355 (82%)
cur > def (M)	0 (0%)	172 (7%)	2202 (92%)
cur > def (L)	0 (0%)	0 (0%)	881 (100%)

SciKit-Learn_Random_Forest camel Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	3670 (96%)	128 (3%)	0 (0%)
cur = def	2918 (41%)	3662 (52%)	381 (5%)
cur > def (all)	32 (2%)	625 (50%)	584 (47%)
cur > def (S)	4 (2%)	122 (67%)	55 (30%)
cur > def (M)	28 (2%)	499 (51%)	447 (45%)
cur > def (L)	0 (0%)	4 (4%)	82 (95%)

SciKit-Learn_Random_Forest ivy Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	8000 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest jedit Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	54 (0%)	12459 (97%)	215 (1%)
cur > def (all)	0 (0%)	3272 (100%)	0 (0%)
cur > def (S)	0 (0%)	426 (100%)	0 (0%)
cur > def (M)	0 (0%)	2812 (100%)	0 (0%)
cur > def (L)	0 (0%)	34 (100%)	0 (0%)

SciKit-Learn_Random_Forest log4j Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	40 (100%)	0 (0%)
cur = def	20 (0%)	6368 (99%)	0 (0%)
cur > def (all)	0 (0%)	1572 (100%)	0 (0%)
cur > def (S)	0 (0%)	222 (100%)	0 (0%)
cur > def (M)	0 (0%)	1349 (100%)	0 (0%)
cur > def (L)	0 (0%)	1 (100%)	0 (0%)

SciKit-Learn_Random_Forest lucene Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	2836 (100%)	0 (0%)
cur = def	0 (0%)	5164 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest synapse Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1645 (75%)	540 (24%)	0 (0%)
cur = def	312 (5%)	5464 (94%)	0 (0%)
cur > def (all)	0 (0%)	39 (100%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	39 (100%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest velocity Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	2738 (58%)	1979 (41%)	0 (0%)
cur = def	2247 (68%)	1036 (31%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest xalan Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	2087 (92%)	161 (7%)	0 (0%)
cur = def	996 (13%)	4229 (57%)	2117 (28%)
cur > def (all)	0 (0%)	685 (28%)	1725 (71%)
cur > def (S)	0 (0%)	0 (0%)	15 (100%)
cur > def (M)	0 (0%)	667 (29%)	1564 (70%)
cur > def (L)	0 (0%)	18 (10%)	146 (89%)

SciKit-Learn_Random_Forest xerces Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	2431 (76%)	734 (23%)	0 (0%)
cur = def	943 (20%)	3330 (72%)	341 (7%)
cur > def (all)	0 (0%)	94 (42%)	127 (57%)
cur > def (S)	0 (0%)	2 (40%)	3 (60%)
cur > def (M)	0 (0%)	92 (42%)	124 (57%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

Appendix D

Bernoulli Naive Bayes Forward Results By Project

SciKit-Learn_Bernoulli_NB ant Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	2012 (88%)	219 (9%)	48 (2%)
cur = def	107 (9%)	663 (60%)	324 (29%)
cur > def (all)	0 (0%)	203 (5%)	3312 (94%)
cur > def (S)	0 (0%)	45 (48%)	47 (51%)
cur > def (M)	0 (0%)	112 (12%)	807 (87%)
cur > def (L)	0 (0%)	46 (1%)	2458 (98%)

SciKit-Learn_Bernoulli_NB camel Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	775 (99%)	4 (0%)	0 (0%)
cur = def	640 (47%)	396 (29%)	313 (23%)
cur > def (all)	0 (0%)	11 (0%)	3027 (99%)
cur > def (S)	0 (0%)	0 (0%)	7 (100%)
cur > def (M)	0 (0%)	10 (1%)	576 (98%)
cur > def (L)	0 (0%)	1 (0%)	2444 (99%)

SciKit-Learn_Bernoulli_NB ivy Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	411 (96%)	16 (3%)	0 (0%)
cur = def	775 (43%)	584 (32%)	425 (23%)
cur > def (all)	0 (0%)	277 (22%)	956 (77%)
cur > def (S)	0 (0%)	18 (32%)	38 (67%)
cur > def (M)	0 (0%)	246 (31%)	527 (68%)
cur > def (L)	0 (0%)	13 (3%)	391 (96%)

SciKit-Learn_Bernoulli_NB jedit Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1419 (73%)	493 (25%)	8 (0%)
cur = def	120 (9%)	749 (59%)	389 (30%)
cur > def (all)	0 (0%)	1170 (31%)	2540 (68%)
cur > def (S)	0 (0%)	27 (29%)	64 (70%)
cur > def (M)	0 (0%)	427 (51%)	407 (48%)
cur > def (L)	0 (0%)	716 (25%)	2069 (74%)

SciKit-Learn_Bernoulli_NB log4j Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1206 (88%)	150 (11%)	0 (0%)
cur = def	155 (19%)	374 (46%)	278 (34%)
cur > def (all)	0 (0%)	7 (0%)	1274 (99%)
cur > def (S)	0 (0%)	1 (1%)	57 (98%)
cur > def (M)	0 (0%)	6 (2%)	239 (97%)
cur > def (L)	0 (0%)	0 (0%)	978 (100%)

SciKit-Learn_Bernoulli_NB lucene Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	282 (41%)	336 (49%)	56 (8%)
cur = def	461 (18%)	1264 (50%)	784 (31%)
cur > def (all)	0 (0%)	21 (8%)	240 (91%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	21 (8%)	240 (91%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Bernoulli_NB synapse Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1210 (80%)	202 (13%)	91 (6%)
cur = def	41 (7%)	214 (36%)	329 (56%)
cur > def (all)	0 (0%)	179 (13%)	1178 (86%)
cur > def (S)	0 (0%)	24 (33%)	48 (66%)
cur > def (M)	0 (0%)	84 (16%)	429 (83%)
cur > def (L)	0 (0%)	71 (9%)	701 (90%)

SciKit-Learn_Bernoulli_NB velocity Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	650 (41%)	477 (30%)	444 (28%)
cur = def	134 (8%)	1010 (67%)	359 (23%)
cur > def (all)	0 (0%)	105 (28%)	265 (71%)
cur > def (S)	0 (0%)	17 (53%)	15 (46%)
cur > def (M)	0 (0%)	86 (33%)	170 (66%)
cur > def (L)	0 (0%)	2 (2%)	80 (97%)

SciKit-Learn_Bernoulli_NB xalan Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	723 (80%)	103 (11%)	69 (7%)
cur = def	27 (2%)	234 (24%)	683 (72%)
cur > def (all)	416 (12%)	292 (8%)	2619 (78%)
cur > def (S)	0 (0%)	4 (12%)	29 (87%)
cur > def (M)	40 (5%)	2 (0%)	675 (94%)
cur > def (L)	376 (14%)	286 (11%)	1915 (74%)

SciKit-Learn_Bernoulli_NB xerces Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	714 (93%)	49 (6%)	1 (0%)
cur = def	227 (40%)	100 (17%)	235 (41%)
cur > def (all)	58 (2%)	148 (6%)	1912 (90%)
cur > def (S)	16 (17%)	0 (0%)	76 (82%)
cur > def (M)	33 (16%)	23 (11%)	143 (71%)
cur > def (L)	9 (0%)	125 (6%)	1693 (92%)

Appendix E

Multinomial Naive Bayes Forward Results By Dataset

SciKit-Learn_Multinomial_NB ant-1.3 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	82 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB ant-1.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	41 (50%)	37 (45%)	4 (4%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB ant-1.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	82 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB ant-1.6 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	82 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB camel-1.0 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	41 (100%)
cur = def	0 (0%)	41 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB camel-1.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	82 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	82 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB lucene-2.0 Benjamini :			
	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	82 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Multinomial_NB velocity-1.5 Benjamini :			
	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	82 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	41 (50%)	41 (50%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

Appendix F

Logistic Regression Forward Results By Dataset

SciKit-Learn_Logistic_Regression ant-1.3 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	2790 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression ant-1.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	2790 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression ant-1.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	2790 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression ant-1.6 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	2790 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression camel-1.0 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	2790 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression camel-1.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	404 (100%)	0 (0%)
cur = def	0 (0%)	2386 (100%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression jedit-4.0 Benjamini :			
	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	999 (100%)	0 (0%)
cur = def	0 (0%)	1664 (100%)	0 (0%)
cur > def (all)	0 (0%)	127 (100%)	0 (0%)
cur > def (S)	0 (0%)	37 (100%)	0 (0%)
cur > def (M)	0 (0%)	90 (100%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression lucene-2.0 Benjamini :			
	fwd < def	fwd = def	fwd > def
cur < def	902 (99%)	3 (0%)	0 (0%)
cur = def	28 (1%)	1581 (83%)	276 (14%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression lucene-2.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	721 (75%)	240 (24%)	0 (0%)
cur = def	0 (0%)	1492 (100%)	0 (0%)
cur > def (all)	0 (0%)	337 (100%)	0 (0%)
cur > def (S)	0 (0%)	61 (100%)	0 (0%)
cur > def (M)	0 (0%)	271 (100%)	0 (0%)
cur > def (L)	0 (0%)	5 (100%)	0 (0%)

SciKit-Learn_Logistic_Regression synapse-1.0 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	976 (98%)	17 (1%)	0 (0%)
cur = def	0 (0%)	1787 (99%)	10 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression synapse-1.1 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	15 (55%)	12 (44%)	0 (0%)
cur = def	27 (0%)	2736 (99%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression velocity-1.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	213 (23%)	686 (76%)
cur = def	29 (1%)	1855 (98%)	0 (0%)
cur > def (all)	0 (0%)	7 (100%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	7 (100%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression velocity-1.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	31 (100%)	0 (0%)	0 (0%)
cur = def	1583 (57%)	907 (32%)	269 (9%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression xalan-2.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	1109 (39%)	984 (35%)	697 (24%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression xalan-2.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	895 (88%)	42 (4%)	69 (6%)
cur = def	603 (34%)	834 (47%)	333 (18%)
cur > def (all)	0 (0%)	3 (21%)	11 (78%)
cur > def (S)	0 (0%)	0 (0%)	3 (100%)
cur > def (M)	0 (0%)	3 (27%)	8 (72%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression xalan-2.6 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	572 (61%)	358 (38%)	0 (0%)
cur = def	17 (0%)	1034 (56%)	793 (43%)
cur > def (all)	0 (0%)	13 (81%)	3 (18%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	13 (81%)	3 (18%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression xerces-1.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	2458 (100%)	0 (0%)
cur > def (all)	0 (0%)	332 (100%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	332 (100%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Logistic_Regression xerces-1.3 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	62 (2%)	2728 (97%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

Appendix G

Random Forest Forward Results By Dataset

SciKit-Learn_Random_Forest ant-1.3 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	300 (100%)	0 (0%)	0 (0%)
cur = def	160 (4%)	3540 (95%)	0 (0%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest ant-1.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	865 (30%)	1934 (69%)	0 (0%)
cur = def	0 (0%)	1157 (97%)	32 (2%)
cur > def (all)	0 (0%)	12 (100%)	0 (0%)
cur > def (S)	0 (0%)	12 (100%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest ant-1.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	627 (83%)	122 (16%)	0 (0%)
cur = def	18 (0%)	2899 (92%)	234 (7%)
cur > def (all)	0 (0%)	60 (60%)	40 (40%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	60 (60%)	40 (40%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest ant-1.6 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	0 (0%)	152 (35%)	273 (64%)
cur > def (all)	0 (0%)	177 (4%)	3398 (95%)
cur > def (S)	0 (0%)	65 (15%)	355 (84%)
cur > def (M)	0 (0%)	112 (4%)	2162 (95%)
cur > def (L)	0 (0%)	0 (0%)	881 (100%)

SciKit-Learn_Random_Forest camel-1.0 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	0 (0%)	0 (0%)	0 (0%)
cur = def	2006 (50%)	1930 (48%)	64 (1%)
cur > def (all)	0 (0%)	0 (0%)	0 (0%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	0 (0%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Random_Forest camel-1.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	1808 (93%)	128 (6%)	0 (0%)
cur = def	20 (1%)	1054 (76%)	297 (21%)
cur > def (all)	0 (0%)	174 (25%)	519 (74%)
cur > def (S)	0 (0%)	42 (43%)	55 (56%)
cur > def (M)	0 (0%)	131 (25%)	382 (74%)
cur > def (L)	0 (0%)	1 (1%)	82 (98%)

SciKit-Learn_Random_Forest camel-1.4 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	1862 (100%)	0 (0%)	0 (0%)	
cur = def	892 (56%)	678 (42%)	20 (1%)	
cur > def (all)	32 (5%)	451 (82%)	65 (11%)	
cur > def (S)	4 (4%)	80 (95%)	0 (0%)	
cur > def (M)	28 (6%)	368 (79%)	65 (14%)	
cur > def (L)	0 (0%)	3 (100%)	0 (0%)	

SciKit-Learn_Random_Forest ivy-1.1 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	4000 (100%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest ivy-1.4 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	4000 (100%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest jedit-3.2 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	980 (100%)	0 (0%)	
cur > def (all)	0 (0%)	3020 (100%)	0 (0%)	
cur > def (S)	0 (0%)	426 (100%)	0 (0%)	
cur > def (M)	0 (0%)	2572 (100%)	0 (0%)	
cur > def (L)	0 (0%)	22 (100%)	0 (0%)	

SciKit-Learn_Random_Forest jedit-4.0 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	3760 (100%)	0 (0%)	
cur > def (all)	0 (0%)	240 (100%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	240 (100%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest jedit-4.1 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	4000 (100%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest jedit-4.2 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	54 (1%)	3719 (93%)	215 (5%)	
cur > def (all)	0 (0%)	12 (100%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	12 (100%)	0 (0%)	

SciKit-Learn_Random_Forest log4j-1.0 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	40 (100%)	0 (0%)	
cur = def	20 (0%)	2368 (99%)	0 (0%)	
cur > def (all)	0 (0%)	1572 (100%)	0 (0%)	
cur > def (S)	0 (0%)	222 (100%)	0 (0%)	
cur > def (M)	0 (0%)	1349 (100%)	0 (0%)	
cur > def (L)	0 (0%)	1 (100%)	0 (0%)	

SciKit-Learn_Random_Forest log4j-1.1 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	4000 (100%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest lucene-2.0 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	4000 (100%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest lucene-2.2 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	2836 (100%)	0 (0%)	
cur = def	0 (0%)	1164 (100%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest synapse-1.0 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	1645 (97%)	44 (2%)	0 (0%)	
cur = def	312 (13%)	1999 (86%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest synapse-1.1 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	496 (100%)	0 (0%)	
cur = def	0 (0%)	3465 (100%)	0 (0%)	
cur > def (all)	0 (0%)	39 (100%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	39 (100%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest velocity-1.4 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	1677 (45%)	1979 (54%)	0 (0%)	
cur = def	0 (0%)	344 (100%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest velocity-1.5 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	1061 (100%)	0 (0%)	0 (0%)	
cur = def	2247 (76%)	692 (23%)	0 (0%)	
cur > def (all)	0 (0%)	0 (0%)	0 (0%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	0 (0%)	0 (0%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest xalan-2.4 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	2065 (93%)	153 (6%)	0 (0%)	
cur = def	384 (21%)	1236 (70%)	141 (8%)	
cur > def (all)	0 (0%)	2 (9%)	19 (90%)	
cur > def (S)	0 (0%)	0 (0%)	15 (100%)	
cur > def (M)	0 (0%)	2 (33%)	4 (66%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest xalan-2.5 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	0 (0%)	0 (0%)	0 (0%)	
cur = def	0 (0%)	1264 (45%)	1523 (54%)	
cur > def (all)	0 (0%)	388 (31%)	825 (68%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	371 (32%)	776 (67%)	
cur > def (L)	0 (0%)	17 (25%)	49 (74%)	

SciKit-Learn_Random_Forest xalan-2.6 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	22 (73%)	8 (26%)	0 (0%)	
cur = def	612 (21%)	1729 (61%)	453 (16%)	
cur > def (all)	0 (0%)	295 (25%)	881 (74%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	294 (27%)	784 (72%)	
cur > def (L)	0 (0%)	1 (1%)	97 (98%)	

SciKit-Learn_Random_Forest xerces-1.2 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	1891 (72%)	730 (27%)	0 (0%)	
cur = def	0 (0%)	1320 (98%)	14 (1%)	
cur > def (all)	0 (0%)	30 (66%)	15 (33%)	
cur > def (S)	0 (0%)	2 (40%)	3 (60%)	
cur > def (M)	0 (0%)	28 (70%)	12 (30%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

SciKit-Learn_Random_Forest xerces-1.3 Benjamini :				
	fwd < def	fwd = def	fwd > def	
cur < def	540 (99%)	4 (0%)	0 (0%)	
cur = def	943 (28%)	2010 (61%)	327 (9%)	
cur > def (all)	0 (0%)	64 (36%)	112 (63%)	
cur > def (S)	0 (0%)	0 (0%)	0 (0%)	
cur > def (M)	0 (0%)	64 (36%)	112 (63%)	
cur > def (L)	0 (0%)	0 (0%)	0 (0%)	

Appendix H

Bernoulli Naive Bayes Forward Results By Dataset

SciKit-Learn_Bernoulli_NB ant-1.3 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	551 (83%)	108 (16%)	0 (0%)
cur = def	4 (1%)	281 (79%)	68 (19%)
cur > def (all)	0 (0%)	44 (6%)	666 (93%)
cur > def (S)	0 (0%)	17 (39%)	26 (60%)
cur > def (M)	0 (0%)	27 (13%)	175 (86%)
cur > def (L)	0 (0%)	0 (0%)	465 (100%)

SciKit-Learn_Bernoulli_NB ant-1.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	386 (90%)	39 (9%)	0 (0%)
cur = def	66 (17%)	266 (71%)	41 (10%)
cur > def (all)	0 (0%)	159 (17%)	765 (82%)
cur > def (S)	0 (0%)	28 (96%)	1 (3%)
cur > def (M)	0 (0%)	85 (56%)	66 (43%)
cur > def (L)	0 (0%)	46 (6%)	698 (93%)

SciKit-Learn_Bernoulli_NB ant-1.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	597 (83%)	70 (9%)	48 (6%)
cur = def	26 (8%)	75 (23%)	213 (67%)
cur > def (all)	0 (0%)	0 (0%)	693 (100%)
cur > def (S)	0 (0%)	0 (0%)	15 (100%)
cur > def (M)	0 (0%)	0 (0%)	459 (100%)
cur > def (L)	0 (0%)	0 (0%)	219 (100%)

SciKit-Learn_Bernoulli_NB ant-1.6 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	478 (99%)	2 (0%)	0 (0%)
cur = def	11 (20%)	41 (75%)	2 (3%)
cur > def (all)	0 (0%)	0 (0%)	1188 (100%)
cur > def (S)	0 (0%)	0 (0%)	5 (100%)
cur > def (M)	0 (0%)	0 (0%)	107 (100%)
cur > def (L)	0 (0%)	0 (0%)	1076 (100%)

SciKit-Learn_Bernoulli_NB camel-1.0 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	24 (100%)	0 (0%)	0 (0%)
cur = def	540 (54%)	175 (17%)	270 (27%)
cur > def (all)	0 (0%)	8 (1%)	705 (98%)
cur > def (S)	0 (0%)	0 (0%)	6 (100%)
cur > def (M)	0 (0%)	8 (1%)	468 (98%)
cur > def (L)	0 (0%)	0 (0%)	231 (100%)

SciKit-Learn_Bernoulli_NB camel-1.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	431 (100%)	0 (0%)	0 (0%)
cur = def	2 (2%)	83 (97%)	0 (0%)
cur > def (all)	0 (0%)	3 (0%)	1203 (99%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	2 (16%)	10 (83%)
cur > def (L)	0 (0%)	1 (0%)	1193 (99%)

```

SciKit-Learn_Bernoulli_NB camel-1.4 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 320 (98%) | 4 (1%)   | 0 (0%)   |
+-----+-----+-----+-----+
| cur = def | 98 (35%)  | 138 (49%) | 43 (15%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 0 (0%) | 1119 (100%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 0 (0%) | 1 (100%) |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 0 (0%) | 98 (100%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 0 (0%) | 1020 (100%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB ivy-1.1 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 411 (96%) | 16 (3%)  | 0 (0%)  |
+-----+-----+-----+-----+
| cur = def | 348 (37%) | 242 (26%) | 334 (36%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 104 (28%) | 267 (71%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 11 (27%) | 29 (72%) |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 91 (37%) | 152 (62%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 2 (2%)  | 86 (97%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB ivy-1.4 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 0 (0%)   | 0 (0%)   | 0 (0%)   |
+-----+-----+-----+-----+
| cur = def | 427 (49%) | 342 (39%) | 91 (10%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 173 (20%) | 689 (79%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 7 (43%)  | 9 (56%)  |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 155 (29%) | 375 (70%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 11 (3%)  | 305 (96%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB jedit-3.2 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 418 (91%) | 31 (6%)  | 8 (1%)  |
+-----+-----+-----+-----+
| cur = def | 0 (0%)   | 369 (53%) | 324 (46%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 122 (21%) | 450 (78%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 4 (6%)   | 59 (93%) |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 117 (38%) | 185 (61%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 1 (0%)   | 206 (99%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB jedit-4.0 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 480 (97%) | 11 (2%)  | 0 (0%)  |
+-----+-----+-----+-----+
| cur = def | 16 (13%)  | 99 (86%) | 0 (0%)  |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 65 (5%)  | 1051 (94%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 0 (0%)  | 0 (0%)  |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 10 (8%)  | 105 (91%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 55 (5%)  | 946 (94%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB jedit-4.1 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 354 (100%) | 0 (0%)  | 0 (0%)  |
+-----+-----+-----+-----+
| cur = def | 104 (35%)  | 124 (42%) | 65 (22%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 94 (8%)  | 981 (91%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 7 (58%)  | 5 (41%)  |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 61 (34%) | 117 (65%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 26 (2%)  | 859 (97%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB jedit-4.2 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 167 (27%) | 451 (72%) | 0 (0%)  |
+-----+-----+-----+-----+
| cur = def | 0 (0%)   | 157 (100%) | 0 (0%)  |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 889 (93%) | 58 (6%)  |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 16 (100%) | 0 (0%)  |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 239 (100%) | 0 (0%)  |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 634 (91%) | 58 (8%)  |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB log4j-1.0 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 674 (83%) | 132 (16%) | 0 (0%)  |
+-----+-----+-----+-----+
| cur = def | 10 (3%)   | 181 (60%) | 109 (36%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 0 (0%)   | 616 (100%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 0 (0%)   | 18 (100%) |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 0 (0%)   | 129 (100%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 0 (0%)   | 469 (100%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB log4j-1.1 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 532 (96%) | 18 (3%)  | 0 (0%)  |
+-----+-----+-----+-----+
| cur = def | 145 (28%) | 193 (38%) | 169 (33%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 7 (1%)   | 658 (98%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 1 (2%)   | 39 (97%)  |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 6 (5%)   | 110 (94%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 0 (0%)   | 509 (100%) |
+-----+-----+-----+-----+

```

```

SciKit-Learn_Bernoulli_NB lucene-2.0 Benjamini :
+-----+-----+-----+-----+
|         | fwd < def | fwd = def | fwd > def |
+-----+-----+-----+-----+
| cur < def | 120 (25%) | 334 (71%) | 16 (3%)  |
+-----+-----+-----+-----+
| cur = def | 0 (0%)   | 904 (84%) | 167 (15%) |
+-----+-----+-----+-----+
| cur > def (all) | 0 (0%) | 21 (11%) | 160 (88%) |
+-----+-----+-----+-----+
| cur > def (S) | 0 (0%) | 0 (0%)  | 0 (0%)  |
+-----+-----+-----+-----+
| cur > def (M) | 0 (0%) | 21 (11%) | 160 (88%) |
+-----+-----+-----+-----+
| cur > def (L) | 0 (0%) | 0 (0%)  | 0 (0%)  |
+-----+-----+-----+-----+

```

SciKit-Learn_Bernoulli_NB lucene-2.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	162 (79%)	2 (0%)	40 (19%)
cur = def	461 (32%)	360 (25%)	617 (42%)
cur > def (all)	0 (0%)	0 (0%)	80 (100%)
cur > def (S)	0 (0%)	0 (0%)	0 (0%)
cur > def (M)	0 (0%)	0 (0%)	80 (100%)
cur > def (L)	0 (0%)	0 (0%)	0 (0%)

SciKit-Learn_Bernoulli_NB synapse-1.0 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	757 (74%)	165 (16%)	91 (8%)
cur = def	0 (0%)	86 (21%)	320 (78%)
cur > def (all)	0 (0%)	15 (4%)	288 (95%)
cur > def (S)	0 (0%)	8 (15%)	45 (84%)
cur > def (M)	0 (0%)	7 (3%)	194 (96%)
cur > def (L)	0 (0%)	0 (0%)	49 (100%)

SciKit-Learn_Bernoulli_NB synapse-1.1 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	453 (92%)	37 (7%)	0 (0%)
cur = def	41 (23%)	128 (71%)	9 (5%)
cur > def (all)	0 (0%)	164 (15%)	890 (84%)
cur > def (S)	0 (0%)	16 (84%)	3 (15%)
cur > def (M)	0 (0%)	77 (24%)	235 (75%)
cur > def (L)	0 (0%)	71 (9%)	652 (90%)

SciKit-Learn_Bernoulli_NB velocity-1.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	122 (12%)	428 (43%)	444 (44%)
cur = def	0 (0%)	363 (67%)	173 (32%)
cur > def (all)	0 (0%)	105 (54%)	87 (45%)
cur > def (S)	0 (0%)	17 (58%)	12 (41%)
cur > def (M)	0 (0%)	86 (53%)	75 (46%)
cur > def (L)	0 (0%)	2 (100%)	0 (0%)

SciKit-Learn_Bernoulli_NB velocity-1.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	528 (91%)	49 (8%)	0 (0%)
cur = def	134 (13%)	647 (66%)	186 (19%)
cur > def (all)	0 (0%)	0 (0%)	178 (100%)
cur > def (S)	0 (0%)	0 (0%)	3 (100%)
cur > def (M)	0 (0%)	0 (0%)	95 (100%)
cur > def (L)	0 (0%)	0 (0%)	80 (100%)

SciKit-Learn_Bernoulli_NB xalan-2.4 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	499 (82%)	38 (6%)	69 (11%)
cur = def	9 (14%)	51 (79%)	4 (6%)
cur > def (all)	4 (0%)	4 (0%)	1044 (99%)
cur > def (S)	0 (0%)	4 (23%)	13 (76%)
cur > def (M)	0 (0%)	0 (0%)	29 (100%)
cur > def (L)	4 (0%)	0 (0%)	1002 (99%)

SciKit-Learn_Bernoulli_NB xalan-2.5 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	112 (63%)	65 (36%)	0 (0%)
cur = def	0 (0%)	132 (16%)	664 (83%)
cur > def (all)	0 (0%)	0 (0%)	749 (100%)
cur > def (S)	0 (0%)	0 (0%)	14 (100%)
cur > def (M)	0 (0%)	0 (0%)	646 (100%)
cur > def (L)	0 (0%)	0 (0%)	89 (100%)

SciKit-Learn_Bernoulli_NB xalan-2.6 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	112 (100%)	0 (0%)	0 (0%)
cur = def	18 (21%)	51 (60%)	15 (17%)
cur > def (all)	412 (26%)	288 (18%)	826 (54%)
cur > def (S)	0 (0%)	0 (0%)	2 (100%)
cur > def (M)	40 (95%)	2 (4%)	0 (0%)
cur > def (L)	372 (25%)	286 (19%)	824 (55%)

SciKit-Learn_Bernoulli_NB xerces-1.2 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	225 (85%)	37 (14%)	0 (0%)
cur = def	227 (47%)	87 (18%)	159 (33%)
cur > def (all)	58 (5%)	148 (14%)	781 (79%)
cur > def (S)	16 (19%)	0 (0%)	66 (80%)
cur > def (M)	33 (35%)	23 (25%)	36 (39%)
cur > def (L)	9 (1%)	125 (15%)	679 (83%)

SciKit-Learn_Bernoulli_NB xerces-1.3 Benjamini :

	fwd < def	fwd = def	fwd > def
cur < def	489 (97%)	12 (2%)	1 (0%)
cur = def	0 (0%)	13 (14%)	76 (85%)
cur > def (all)	0 (0%)	0 (0%)	1131 (100%)
cur > def (S)	0 (0%)	0 (0%)	10 (100%)
cur > def (M)	0 (0%)	0 (0%)	107 (100%)
cur > def (L)	0 (0%)	0 (0%)	1014 (100%)

Appendix I

Multinomial Bayes Current Version Results By Dataset

Dataet Name	Cur<Def	Cur=Def	Cur>Def (all)	Cur>Def (S)	Cur>Def (M)	Cur>Def (L)
ant-1.3	0	82	0	0	0	0
ant-1.4	0	82	0	0	0	0
ant-1.5	0	82	0	0	0	0
ant-1.6	0	82	0	0	0	0
ant-1.7	0	82	0	0	0	0
camel-1.0	41	41	0	0	0	0
camel-1.2	0	82	0	0	0	0
camel-1.4	0	82	0	0	0	0
camel-1.6	0	82	0	0	0	0
ivy-1.1	0	82	0	0	0	0
ivy-1.4	0	82	0	0	0	0
ivy-2.0	0	82	0	0	0	0
jedit-3.2	0	82	0	0	0	0
jedit-4.0	0	82	0	0	0	0
jedit-4.1	0	82	0	0	0	0
jedit-4.2	0	82	0	0	0	0
jedit-4.3	40	42	0	0	0	0
log4j-1.0	41	41	0	0	0	0
log4j-1.1	0	82	0	0	0	0
log4j-1.2	2	80	0	0	0	0
lucene-2.0	0	82	0	0	0	0
lucene-2.2	0	82	0	0	0	0
lucene-2.4	0	82	0	0	0	0
synapse-1.0	2	80	0	0	0	0
synapse-1.1	0	82	0	0	0	0
synapse-1.2	0	82	0	0	0	0
velocity-1.4	0	82	0	0	0	0
velocity-1.5	0	82	0	0	0	0
velocity-1.6	0	82	0	0	0	0
xalan-2.4	0	82	0	0	0	0
xalan-2.5	0	82	0	0	0	0
xalan-2.6	0	82	0	0	0	0
xalan-2.7	22	47	13	0	3	10
xerces-1.2	0	82	0	0	0	0
xerces-1.3	0	82	0	0	0	0
xerces-1.4	60	19	3	3	0	0

Appendix J

Logistic Regression Current Version Results By Dataset

Dataet Name	Cur<Def	Cur=Def	Cur>Def (all)	Cur>Def (S)	Cur>Def (M)	Cur>Def (L)
ant-1.3	0	2790	0	0	0	0
ant-1.4	0	2790	0	0	0	0
ant-1.5	0	2790	0	0	0	0
ant-1.6	0	2790	0	0	0	0
ant-1.7	787	2003	0	0	0	0
camel-1.0	0	2790	0	0	0	0
camel-1.2	404	2386	0	0	0	0
camel-1.4	7	2783	0	0	0	0
camel-1.6	0	2790	0	0	0	0
ivy-1.1	964	1826	0	0	0	0
ivy-1.4	0	2790	0	0	0	0
ivy-2.0	0	2790	0	0	0	0
jedit-3.2	1255	1535	0	0	0	0
jedit-4.0	999	1664	127	37	90	0
jedit-4.1	956	1834	0	0	0	0
jedit-4.2	0	2790	0	0	0	0
jedit-4.3	0	2790	0	0	0	0
log4j-1.0	0	2790	0	0	0	0
log4j-1.1	627	2163	0	0	0	0
log4j-1.2	0	2790	0	0	0	0
lucene-2.0	905	1885	0	0	0	0
lucene-2.2	961	1492	337	61	271	5
lucene-2.4	1216	1184	390	108	282	0
synapse-1.0	993	1797	0	0	0	0
synapse-1.1	27	2763	0	0	0	0
synapse-1.2	945	1819	26	4	22	0
velocity-1.4	899	1884	7	0	7	0
velocity-1.5	31	2759	0	0	0	0
velocity-1.6	0	2790	0	0	0	0
xalan-2.4	0	2790	0	0	0	0
xalan-2.5	1006	1770	14	3	11	0
xalan-2.6	930	1844	16	0	16	0
xalan-2.7	841	1949	0	0	0	0
xerces-1.2	0	2458	332	0	332	0
xerces-1.3	0	2790	0	0	0	0
xerces-1.4	930	1799	61	1	60	0

Appendix K

Random Forest Current Version Results By Dataset

Dataet Name	Cur<Def	Cur=Def	Cur>Def (all)	Cur>Def (S)	Cur>Def (M)	Cur>Def (L)
ant-1.3	300	3700	0	0	0	0
ant-1.4	2799	1189	12	12	0	0
ant-1.5	749	3151	100	0	100	0
ant-1.6	0	425	3575	420	2274	881
ant-1.7	0	532	3468	248	2657	563
camel-1.0	0	4000	0	0	0	0
camel-1.2	1936	1371	693	97	513	83
camel-1.4	1862	1590	548	84	461	3
camel-1.6	2324	1299	377	73	304	0
ivy-1.1	0	4000	0	0	0	0
ivy-1.4	0	4000	0	0	0	0
ivy-2.0	264	3736	0	0	0	0
jedit-3.2	0	980	3020	426	2572	22
jedit-4.0	0	3760	240	0	240	0
jedit-4.1	0	4000	0	0	0	0
jedit-4.2	0	3988	12	0	0	12
jedit-4.3	0	4000	0	0	0	0
log4j-1.0	40	2388	1572	222	1349	1
log4j-1.1	0	4000	0	0	0	0
log4j-1.2	3042	958	0	0	0	0
lucene-2.0	0	4000	0	0	0	0
lucene-2.2	2836	1164	0	0	0	0
lucene-2.4	0	4000	0	0	0	0
synapse-1.0	1689	2311	0	0	0	0
synapse-1.1	496	3465	39	0	39	0
synapse-1.2	0	4000	0	0	0	0
velocity-1.4	3656	344	0	0	0	0
velocity-1.5	1061	2939	0	0	0	0
velocity-1.6	0	2698	1302	0	1302	0
xalan-2.4	2218	1761	21	15	6	0
xalan-2.5	0	2787	1213	0	1147	66
xalan-2.6	30	2794	1176	0	1078	98
xalan-2.7	1630	2345	25	17	8	0
xerces-1.2	2621	1334	45	5	40	0
xerces-1.3	544	3280	176	0	176	0
xerces-1.4	124	3824	52	0	52	0

Appendix L

Bernoulli Bayes Current Version Results By Dataset

Dataet Name	Cur<Def	Cur=Def	Cur>Def (all)	Cur>Def (S)	Cur>Def (M)	Cur>Def (L)
ant-1.3	659	353	710	43	202	465
ant-1.4	425	373	924	29	151	744
ant-1.5	715	314	693	15	459	219
ant-1.6	480	54	1188	5	107	1076
ant-1.7	447	75	1200	0	21	1179
camel-1.0	24	985	713	6	476	231
camel-1.2	431	85	1206	0	12	1194
camel-1.4	324	279	1119	1	98	1020
camel-1.6	460	90	1172	35	17	1120
ivy-1.1	427	924	371	40	243	88
ivy-1.4	0	860	862	16	530	316
ivy-2.0	538	295	889	58	86	745
jedit-3.2	457	693	572	63	302	207
jedit-4.0	491	115	1116	0	115	1001
jedit-4.1	354	293	1075	12	178	885
jedit-4.2	618	157	947	16	239	692
jedit-4.3	0	1300	422	0	422	0
log4j-1.0	806	300	616	18	129	469
log4j-1.1	550	507	665	40	116	509
log4j-1.2	175	1044	503	1	254	248
lucene-2.0	470	1071	181	0	181	0
lucene-2.2	204	1438	80	0	80	0
lucene-2.4	432	246	1044	12	422	610
synapse-1.0	1013	406	303	53	201	49
synapse-1.1	490	178	1054	19	312	723
synapse-1.2	486	348	888	35	448	405
velocity-1.4	994	536	192	29	161	2
velocity-1.5	577	967	178	3	95	80
velocity-1.6	402	101	1219	25	70	1124
xalan-2.4	606	64	1052	17	29	1006
xalan-2.5	177	796	749	14	646	89
xalan-2.6	112	84	1526	2	42	1482
xalan-2.7	659	363	700	17	190	493
xerces-1.2	262	473	987	82	92	813
xerces-1.3	502	89	1131	10	107	1014
xerces-1.4	1025	495	202	0	82	120