2015

Exploring Essential Content of Defect Prediction and Effort Estimation through Data Reduction

Divya Ganesan

Exploring Essential Content of Defect Prediction and Effort Estimation through Data Reduction

Divya Ganesan

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
Elaine Eschen, Ph.D.
Vasudevan Jagannathan, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2015

Keywords: Data Mining, Software Defect Prediction, Software Effort Estimation, Machine Learning, Data Reduction, Clustering Techniques, Software Engineering

ABSTRACT

Exploring Essential content of Defect Prediction and Effort Estimation through Data Reduction

Divya Ganesan

Mining Software Repositories provides the opportunity to exploit/explore some of the behaviors, distinct patterns and features of software development processes, using which the stakeholders can generate models to perform estimations, predictions and make decisions on these projects.

When using data mining on project data in software engineering, it is important to generate models that are easy for business users to understand. The business users should be able to gain insight on how to improve the project using these models. Software engineering data are often too large to discern. To understand the intricacies of software analytics, one approach is to reduce software engineering data to its essential content, then reasoning about that reduced set.

This thesis explores methods (a) removing spurious and redundant columns then (b) clustering the data set and replacing each cluster by one exemplar per cluster then (c) making conclusions by extrapolating between the exemplars (via k=2 nearest neighbor between cluster centroids).

Numerous defect data sets were reduced to around 25 exemplars containing around 6 attributes. These tables of 25*6 values were then used for (a) effective and simple defect prediction as well as (b) simple presentation of that data. Also, in an investigation of numerous common clustering methods, we find that the details of the clustering method are less important than ensuring that those methods produce enough clusters (which, for defect data sets, seems to be around 25 clusters). For effort estimation data sets, conclusive results for ideal number of clusters could not be determined due to smaller size of the data sets.

Acknowledgments

I am grateful to a number of people who helped me through this research. First of all, I would like to thank Dr. Menzies for his guidance in the last two years. This thesis would not have been completed without his help and support. I would like to thank Dr. Eschen for encouraging me throughout my graduate program.

I take this opportunity to thank all my colleagues in Modeling Intelligence Lab (MILL lab) who have shared their knowledge and technical expertise. In particular, Vasil Papakroni for taking the time to explain his research work and answering my numerous questions about it.

A special thanks to my husband, Vinoth Pugazenthi, for encouraging me to do my graduate studies and motivating me to work hard and never give up. I would like to thank my parents who had been supporting me from both near and far throughout this research. I would like to thank my family for their constant care, love and encouragement during my time in graduate school.

Finally, I would like to acknowledge the support of Lane Department of Computer Science and Electric Engineering by providing me the necessary funding through Teaching Assistantship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

To efficiently sieve through to the essential section of data, it needs to be reduced into a subset that represents the very salient information. Within a data set, the most common redundancy to potentially exist is in the number of features. The features that are irrelevant do not contribute to accuracy of prediction and in addition might actually negatively impact it. Reducing the number of attributes is desirable because it reduces the overall complexity of the data, and a model generated using the selected attributes is easier to understand [15].

The other possible reduction opportunity is to cluster data to derive a condensed data set, find one exemplar that effectively represents a set of instances. The data set can be clustered to find groups of similar instances and replace them with the centroid of a cluster [2] [23].

The thesis attempts to explore the following research questions

- *RQ1: What are the different clustering techniques that can be employed for efficient instance reduction and what are their impacts on predictive ability? Do some clus-*

tering methods perform significantly better than others? Various clustering methods were applied to condense data sets effectively and the predictive ability of 2-Nearest Neighbour(2-NN) algorithm run over condensed data sets generated was evaluated.

- *RQ2: Will large data reduction result in information loss and predictive ability of learners trained on them?* When data reduction is performed, there is removal of irrelevant data, but some essential information could be lost as well. Hence, this thesis will assess the value of various clustering methods by applying data mining to the clustered data.

- *RQ3: What is the optimal number of clusters and how much data reduction is too much?* The size of the condensed data set should be optimal in striking a balance between simplifying the data to enable user engagement and minimizing information loss if any due to reduction of data. The predictive ability of the clustering techniques with different number of clusters and various parameter values were evaluated to analyze the impact of size of condensed data set on performance.

1.2 Context: Data Mining in Software Engineering

This thesis explores the above in the context of data mining for Software Engineering.

Software Engineering projects go through a process of planning, creating requirements, realizing it with software code, testing it to identify defects and errors and releasing it to be available for users.

A *defect* is a behavioral or functional non-compliance of the software to the specifications provided in the requirements. It is usually found during the testing phase, however it could be identified during any phase of the project. The defects need to be fixed or resolved so that the project can comply with the requirements.

The presence of defects in the software comes at a cost associated with it. While cost per defect used to be an overall metric, there could be different cost associated with the defects identified in different phases of the project. Defects found during requirements cost $250, during design costs $500, during coding and testing costs $1250 and after release costs $5000 [18]. The defects are unavoidable, but predicting the defects in advance will result in mitigation of risks, cost savings and better adherence to project plan.

The effort in the project is the time or money needed to develop the project. This would include all the phases and resources in the project. The effort estimation is predicting the amount of time or money needed to develop the project.

Effort estimation is key to the planning of the projects. A survey by Standish Group's 'Chaos Report' shows an average of 89% cost overrun [27] while other recent surveys report a 30 to 40 % [22] much less, yet high percentage of overrun. This cost can pose risks in the project and can be avoided with better estimation techniques.

It is imperative, in a project, to arrive at a better estimation, and predict the defects in the project. Being able to gain the defect prediction and effort estimation through effective mining of the project data using tools and techniques will give the project team, the capability to better estimate and execute the project.

The two main factors which need to be considered while presenting the results to the project team are (i) it should predict the defective models or estimate the effort with high accuracy (ii) it should provide insight to the factors affecting the prediction and estimation.

Lots of research, in effort estimation for example, concentrate on the evaluation of the estimation methods, but have very less focus on associating these estimation models to the needs of the users (about 61% of topics collected in a study over 300 papers from 76 journals) [3].

Many software engineering mining research is done on improving accuracy of defect pre-

dictions using complex learners (boot strapping, assorted analogy methods such as nearest neighbor, non-linear model such as decision trees) without considering the level of understanding that the project team needs. This is not an efficient solution to the players involved in the project. It has shown to affect the adaptability of the learners and prediction models in industrial practice, which continues to follow standard regression-based algorithm [17].

The effort estimation and defect prediction techniques and methodologies should not be a black box to the project team. The techniques should provide some specific details on the results and possibly offer information about the factors which directly impact them. This would help them to use the techniques as a leverage to take an action on the results obtained and tune the project code to minimize the defects and estimate it better [4].

Large amount of data available from the project can be used to derive these values. But, this poses the cumbersome task of mining through the entire data set to identify and generate meaningful information. The study done on software defect prediction in the NASA aerospace domain, Turkish whitegoods control software, and other open source software shows that the performance of data mining plateaus early in the data set, inferring that the regular patterns in the software projects convey that one can generate effective defect predictors with small subset of the information.

The research on the effort estimation using the COCOMO features on COCO81 data also shows similar patterns. The studies conducted on unrelated domains prove that the regularities are not just quirks within one domain and are rather evident regularities of software engineering projects [28].

In summary, in several software engineering domains, it is useful to *divide* the data before reasoning about it; hence, thesis studies different methods for clustering.

1.3 Statement of Thesis

The research work of the thesis can be summarized by the following statement:

> *The software engineering data can be summarized to generate concise models using feature and instance reduction. In case of defect data the choice of clustering method employed for instance reduction becomes less significant when sufficient number of clusters are generated*

1.4 Contribution of the Thesis

The contributions of the thesis can be summarized as follows:

- Data reduction using feature selection and instance selection was explored. Experimental results show that data reduction can be done with minimal loss of information.
- The efficiency of various clustering techniques for instance selection was evaluated. The results of the study shows the some standard clustering methods applied in the study perform better than standard learners and PEEKING2, a data reduction approach.
- The optimal value for number of instances in the reduced data set was investigated.

1.5 Structure of the Thesis

The rest of the thesis is organized as follows:

- Chapter 2 describes the related work done in mining software repositories for defect and effort data. This provides information on recent research conducted in data reduction for Software Engineering.

- Chapter 3 provides a brief introduction to feature selection and explores the theory behind various types of clustering techniques used in the study. This chapter provides information on data sets and describes the approach followed in the experiments to perform attribute and instance reduction. This chapter also explains the measures and rationale behind the choices made in comparing the different techniques.
- Chapter 4 presents the results of experiments in detail.
- Chapter 5 discusses the threats to the validity of the results of the study.
- Finally, Chapter 6 summarizes the observations of the experiments and items identified for further research.

Chapter 2

Related Work

2.1 Feature Selection

Feature selection is a widely studied area in the field of Data Mining. Feature selection is used as a part of data pre-processing before clustering or classification algorithms are applied on the data. Guyon and Elisseeff [15] defines the objectives of feature selection and discusses feature construction, feature ranking, multivariate feature selection, efficient search methods, and feature validity assessment methods. Liu and Yu [20] surveyed existing feature section and propose an integrated approach to feature selection. The paper also provides guidelines for selecting appropriate feature selection algorithm.

The importance of feature selection to remove irrelevant attributes in field of Software Engineering is explored in the following papers. Chen et al. [5] applied wrapper based feature selection techniques on software cost or effort estimation data and observed that reduced data could improve the estimation. Rodriguez et al. [25] evaluated three filters and wrappers by applying them to five software engineering data. They observed that reduced data sets with fewer features had predictability comparable to the original data set. They concluded that wrapper methods performed better than filter however, computational cost

for wrapper methods were much higher.

More recent study by Gao et al. [14] applied seven feature ranking methods along with three feature subset selection methods to software defect data found that performance of most of the feature selection methods to be similar. Based on the experimental results, they recommend the feature the following feature selection techniques: Information gain, Gain Ratio and Kolmogorov-Smirnov statistic. They also observed performance of defect prediction models improved or remained unchanged even when over 85% of the features were eliminated.

These related the work focus on evaluating different feature selection techniques to improve the performance of the learner, this thesis work focuses on applying feature selection together with instance reduction to effectively condense the data.

2.2 Instance Reduction

Most of the signals from software engineering data comes from a small section of data. Some data sets have extraneous instances or noise in them. The presence of redundant and confusing instances affect the performance of the model generated by learner. Instance selection can help mitigate the issue. Turhan et al. [29] explore instance sampling for software defect prediction. In the paper, the authors performed microsampling wherein N defective instances and non-defective instance were randomly selected from total instances. The results showed that for most data sets predictive ability of models did not improve by increasing the size of N beyond 25.

2.3 Data Reduction Methods

IDEA introduced in [2] performs data reduction by the following steps, dimensionality reduction using FastMap [10], hierarchical clustering and then feature selection using entropy. The IDEA was tested on multiple effort data set and its performance was compared with 10 methods and 9 pre-processors. The results indicated that there was no significant loss of essential information

PEEKING2, an improved version of IDEA, used optional feature selection by INFOGAIN and Nearest Neighbor method to extrapolate between centroids. PEEKING2 was tested on defect and effort data sets [23]. The performance of PEEKING2 was found close to standard learners. However, for some of the defect data sets, only PEEKING2 applied on all features performed well.

This thesis explores alternative for clustering technique used in PEEKING2, five widely used clustering techniques are evaluated for instance reduction. The experiments also attempts to find the optimal number of instances in the reduced data set. The results of the study shows the some standard clustering methods applied in the study perform better than PEEKING2 and standard learners.

Chapter 3

Methods

3.1 Feature Selection

Feature Selection is process used to reduce the number of features or attributes by selecting most promising/ relevant features which will improve the predictive accuracy of the learner. The presence of irrelevant features have been shown to deteriorate performances of decision trees (C4.5) and instance based learners. While some learners like Naive Bayes are robust to irrelevant features, redundant features can still impact the performance.

In addition to improving the accuracy of a learner, feature selection can also help to understand the data. It identifies features which influence the target variable the most. The two common types of feature selection methods are filters and wrappers. In filter methods, feature selection is done as a part of pre-processing step without considering the impact of the feature selection on the actual performance of the learner. The filters, in general, rank a feature or a subset of features based on the relevance to the class variable. The metric or measure of the relevance influences the selection of features. However, the selection of features is independent of the learning algorithm used.

In wrappers, subset of features are formed and the predictive ability of the learner when trained on the feature subset is evaluated. Wrappers explores different feature subsets and evaluate by training the actual learner on the corresponding features. The feature subset is then scored based on the accuracy of the learned model. Wrappers are more computationally intensive than filters, because the actual learner is run for each feature subset. For large data sets, convergence of a wrapper is not guaranteed when exhaustive search is used to evaluate all possible subsets of features. Consequently, wrappers use greedy search algorithms to converge to locally optimal solutions.

3.1.1 Information Gain (InfoGain)

A simple filter method is used to rank features based on the relevance of features to the class variable. Information Gain is used as the relevance measure. The idea of Information Gain is better understood with the notion of Decision Trees and Entropy.

Decision Tree: The decision tree is a simple tree with the root node representing a question, each branch acting as choices that lead to other nodes representing an attribute within the training set. These lead to the leaf nodes with the decision or outcome for condition or question posed in the root node.

Entropy: The entropy is defined as the disorder in predictability of the nodes and the branches in the decision tree. In a given set S, the entropy is

$$\text{Entropy (S)} = -\text{Positive(S)} \log_2 \text{Positive(S)} - \text{Negative(S)} \log_2 \text{Negative(S)}$$

where Positive(S) is the proportion of the positive values in the set and Negative(S) is the proportion of the negative values in the set. So, if there are all positive values and no negative values in Set S, then Positive(S) is 1, Negative(S) is 0, then the entropy would be 0. If there are equal number of positive and negative values, then Positive(S) is 0.5, Negative(S) is 0.5

and Entropy is 1. If the Positive(S) is 0.25 and Negative(S) is 0.75, then the Entropy is 0.811. This means that the more impurities, the Entropy leads towards 1. So, the entropy should be reduced for making optimal decisions at each node.

Information Gain: The Information Gain is defined as the Expected reduction of the above defined Entropy of the set, in relation with the given attribute in the node where the decision is being made.

$$\text{Gain (S, A)} = Entropy\ (S) - \sum_{v=1\ to\ n} \frac{|Sv|}{|S|} Entropy\ (S)$$

These values give the uncertainty of each attribute. The information gain can be used to rank the attributes and build a new decision tree. This tree will show the attributes with the highest information gain. After all features are ranked, a specified percentage of features with the highest Information Gain are selected.

3.2 Clustering Methods

Clustering is an unsupervised machine learning technique where data instances are grouped into several subsets based on the similarity between the instances. Hence, it is important to define the similarity or the distance measure used to compare two instances. Clustering techniques vary in both the induction principle and distance/similarity measure used. Many different classification of clustering techniques has been suggested based on the two criteria [16] [9]. The common classifications of clustering methods are:

- **Hierarchical clustering.** The clusters are formed by recursively partitioning the instances by using a top-down or bottom-up approach. In Agglomerative Hierarchical

clustering, each instance is initially considered as a cluster, clusters are merged in successive iteration based on similarity measures. In divisive hierarchical clustering, all instances are initially considered it to be in one cluster, which is divided in in successive iteration. The hierarchical clustering method produce a dendrogram which shows the nested grouping of instances and with value of similarity measure for each group. The clusters are obtained by cutting off the dendrogram based threshold value for similarity measure.

- **Partitioning clustering.** The methods form a set of clusters initially, and then refines them by moving instances between clusters. In general, Partitioning methods require the user to select the total number of clusters to be formed. Partitioning clustering algorithm use greed search algorithm to optimize the clusters. Partition based clustering methods typically use distance measures and form clusters which are spherical in shape.

- **Density-based clustering.** The density based clustering is based on the assumption that clusters are dense region in the dataspace surrounded by noise (low density region) [16]. Density based methods can form methods of arbitrary shape.

In this study we chose five clustering methods based on different underlying principles.

3.2.1 K-Means

K-Means uses a simple iterative method to identify a predetermined number (k) of centroids on a data set and clustering the data points to those centroids. The algorithm was originally proposed by Lloyd (1957) as a method in signal processing, later published by Forgy (1965). In addition, to coining the name "k-means", it was used by MacQueen (1967). A more refined, efficient and generalized versions were proposed later by Hartigan and Wong (1975). This is widely used in the field of data mining today [32].

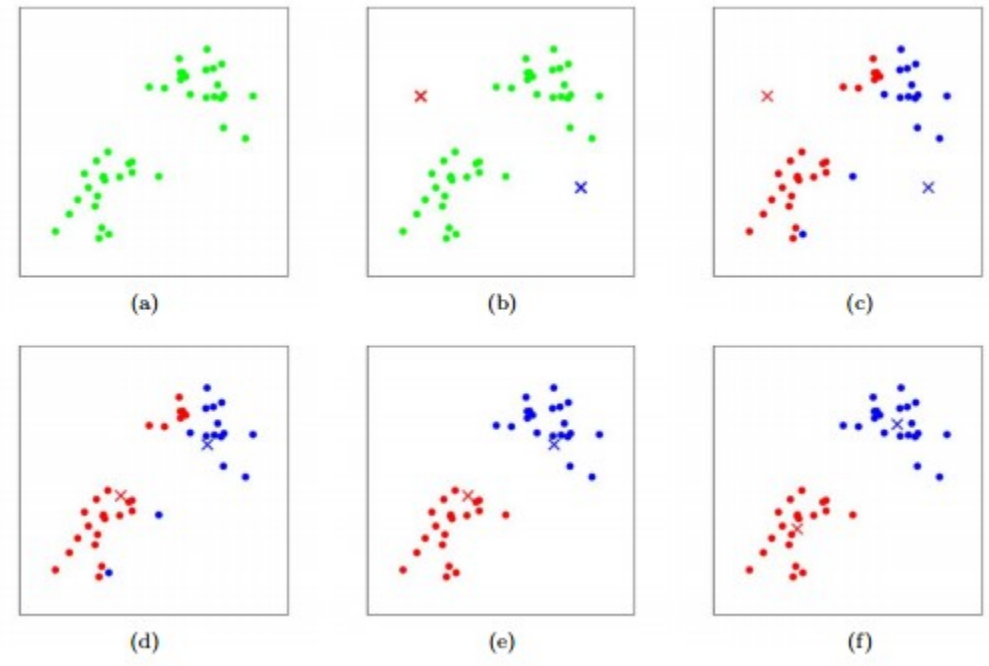

Figure 3.1: *K-Means clustering taken from [24]*

The steps that are involved in the k-means algorithm are:

1. Initialize a predetermined number (k) for number of clusters. This needs to be specified by the user.
2. Select k data points in the given data set as the centroids.
3. Assign all the other data points to the nearest centroid.
4. Once the clusters are formed, calculate the centroid of each cluster, creating a new set of k centroids.
5. Iteratively repeat steps 3 to 5 until the centroids do not move and the data set is converged into k clusters.

The above figure shows how k means is performed on a data set with a value of k set to 2 in the clusters. a) This represents the initial data set. b) The red and blue data points are the

two random centroids. c) The other data points are assigned to the centroids as either blue or red depending on each data points distance from the centroids, forming two clusters. d) The new centroids are identified by calculating the mean of the two clusters formed. e) The data points are reassigned by calculating the distance between them and the new centroids. f) This shows the converged data set with the two clusters after some iterations.

The initial number of clusters has to be set by the users. The initial k centroids points can be selected by mere random selection of k data points. The nearest centroid is calculated using the Euclidean distance, hence the cost function

$$\sum_{i=1}^{N} \left(\begin{array}{c} argmin \\ j \end{array} ||x_i - \ c_j|| \right)^2_2$$

will climb towards zero, so that the convergence will be happen within the subsequent iterations forming a final set of clusters within limited iterations.

For a data set of size N and number of centroids k, the algorithm will need to compare k X N times within each iteration. This defines the complexity of each iteration and the total number of iterations depends on the size of N.

The k-means algorithm is very simple in its nature and is not complex to implement. It is guaranteed to provide the clusters within limited iterations and has proven to be very useful in clustering. This is one of the reasons why it is widely used in data mining.

The algorithm, like any other has its disadvantages. The main limitation is that the cluster set depends on the number of clusters, k, provided by the user without predetermined knowledge about the data and its characteristics within the data set. The clustering also depends on the initial data points, randomly chosen as the centroids, causing good or bad results on the same data set. The data points are assigned to the centroid merely on the closeness of the data point. So, if the original data set is not formed by spherical or convex shaped clusters, this algorithm might not provide good results. This could be resolved by

using a different method of measuring the distance.

One of the disadvantages being the initial choice of k, the number of clusters, the desirable value of which, if not known in advance could cause bad results. This can be addressed by choosing increasing number for k and choosing the better result.

The mean method to identify the centroid might not be the suitable statistical measure in some cases, like when there are very small clusters. Combining small clusters to a larger cluster could fix this issue.

3.2.2 Mini-batch K-Means

Mini-batch K-Means (MB K-Means) is a variant of K-Means which addresses the issue of scalability of K-Means [26]. Instead of processing all input records at once, mini-batch K-Means samples B random input records at each iteration. This helps mini-batch k-means to converge much faster than K-Means for larger data sets. This method has a faster processing time, while producing results very close to those of K-Means.

Mini-batches are subsets of the training set, which are randomly chosen in every iteration. This significantly reduces the computation involved in the convergence into the initial number of clusters.

The steps that are involved in the Mini-batch k-means algorithm are:

1. Initialize a predetermined number (k) for number of clusters. This needs to be specified by the user.
2. Select k data points in the given data set as the centroids.
3. Select b samples randomly from the data set to form a Mini-batch. Assign them to the nearest centroid.

4. Once the clusters are formed, calculate the centroid of each cluster, creating a new set of k centroids. The centroids are calculated by getting the streaming average of all the samples and the previous samples that were assigned to that centroid

5. Iteratively repeat steps 3 to 5 until the centroids do not move and the data set is converged into k clusters.

This sampling instead of assigning the entire data set to the centroids decreases the rate at which the centroids change. With less amount of calculation each iteration and reducing overall number of iteration by faster convergence, this algorithm improves the performance. But, the results are slightly worse than the original K-Means. Even with the difference in the results, it is only slightly worse, proving to be useful in some cases.

3.2.3 DBSCAN

DBSCAN stands for Density Based Spatial Clustering of Applications with Noise. This is developed to address the below data set requirements

(i) There is limited domain knowledge of the data set to provide the input parameters

(ii) To identify arbitrary shaped clusters

(iii) Efficient performance on large data sets

These requirements are satisfied by DBSCAN which relies on density based clusters based on the assumption that the data set contains clusters which are high density areas and noise with low density area [8]. The method was proposed by Martin Ester, Hans-Peter Kriegel, Jorg Sander, Xiaowei Xu in 1996. It is one of the most cited algorithm and has been very robust in its performance and application on different data sets over years.

In DBSCAN, there are two necessary inputs:

- textbfMinPts - A predetermined number of points. This is generally set little low and there are methods to determine this without knowing the domain of the data set.
- textbfEps(ϵ) - This is the neighborhood where there are minimum MinPts points.

DBSCAN begins by scanning an arbitrary point p, retrieving all points, which are density reachable from point p.

- If p is a real core point, many points will be reachable from that and will form a cluster.
- If p is not a core point, it would not form the cluster. If p is a border point, DBSCAN moves to the next one, labeling that as noise. However, it could still be picked up as part of another cluster, if it is reachable by that cluster later in the program.
- If MinPts are higher, a recursive call will be necessary.

The running time of this algorithm is O(n log n). DBSCAN identifies all the clusters and noise points too. It does not need the pre-determined number of clusters, which is the case in k-means. It is also not very sensitive to the quality of the data set. The issue with this algorithm is that there could be points that are density reachable by more than one core point. This might assign the point to an incorrect cluster. If the input data set has high differences in density and low separation of the data points, DBSCAN does not perform well in properly clustering the input set.

3.2.4 Expectation Maximization

Expectation Maximization(E-Max) is a Gaussian mixture model algorithm. The algorithm initially forms random components. It then calculates a probability that any given point is generated by each component of the model. The algorithm then changes its parameter value to maximize the likelihood of the data given those assignments [7].

Expectation Maximization is an iterative process of estimating the Maximum Likelihood of the data set, that contains hidden information. Given two clusters, when the parameters like means and standard deviations for each cluster and the probabilities are unknown, the EM algorithm utilizes the k-means algorithm iteratively. The initial parameters are just guesses which are used to calculate the probability of each cluster. With the values obtained from this, the parameters which were initially guessed are re-estimated and the iteration is continued. It basically involves the following steps:

1. Calculate the expected values, called the Expectation step. The expectation z of the hidden variable is
$$E[z_{ij}] = \frac{p\left(x = x_i \mid = _j\right)}{\int_{n=1}^{k} p\left(x = x_i \mid = _n\right)}$$
2. Calculate the distribution of parameters, called the Maximization step, to increase the chances of the distribution with the available data.
$$\mu \leftarrow \frac{\int_{i=1}^{m} E[z_{ij}]\ x_i}{\int_{i=1}^{m} E[z_{ij}]}$$

Expectation Maximization algorithm increases the cluster probabilities by iteratively re-estimating the parameters after guessing the initial parameters.

While the iterations with different values have to be performed, it is still a very fast algorithm; it just relies on the speed of the K-Means algorithm performed within the iterations. With various values, it maximizes the likelihood of the unbiased clusters, without pushing the means towards zero.

Since the values are different, instead of converging, the results in different iterations could diverge it to irregular output.

3.2.5 Ward

Ward's method is a criterion in hierarchical clustering approach where the merge criteria for two clusters is an objective function, per the usersâĂŹ needs. This was suggested by Joe H. Ward in 1963 as a grouping approach to optimize and objective function. Each point in dataspace is initially considered to be in a cluster of it own, and the clusters are merged in successive iteration such that intra-cluster distance is kept to the minimum [30]

If there are n subsets, two subsets are merged, giving n-1 sets. This similarity is determined by specific characteristics. So, the n subsets are reduced to n-1 mutually exclusive subsets by checking the union of possible k(k-1)/2 pairs that can be formed while accepting the union with which an optimal value of the objective function is associated. This process is repeated till the subsets are in one group.

Since Ward's method uses agglomerative hierarchical clustering approach, the algorithm itself begins at the leaf, going through the branches and then the trunk. It is a bottom up approach starting at its own cluster. The criterion used by Ward is called the Ward's minimum variance criterion.

The implementation of the hierarchical clustering using the Ward's method has the following steps:

Let p denote the smaller of the two numbers used to identify the subset in the n original subsets.

Let q denote the larger of the two numbers used to identify the subset in the n original subsets.

1. Set k (number of groups) to n (number of elements)
2. Set the best value $Z[p_{k-1},\ q_{k-1},\ k\text{-}1]$ to an initial worse value.

3. Set i to the smallest active number.

4. Set j to the first active number which is more than i.

5. Compute best value *Z[i, j, k-1]* associated with the union of sets i and j.

6. If *Z[i, j, k-1]* is better than the previous initial *Z[*p_{k-1}*,* q_{k-1}*, k-1]*, continue, otherwise go to step 8.

7. Replace initial *Z[*p_{k-1}*,* q_{k-1}*, k-1]* with new *Z[i, j, k-1]*.

8. If j is not equal to the last active number, continue, otherwise go to step 10

9. Set j to the next active number, go to step 5.

10. If i is not equal to the last active number, continue, otherwise go to step 12.

11. Set i to the next active number, go to step 4.

12. The best union is identified by p_{k-1}, q_{k-1}along with its value *Z[*p_{k-1}*,* q_{k-1}*, k-1]*.

13. Identify the new union by the p_{k-1}. Set q_{k-1}as inactive.

14. If k (number of groups under consideration) equals 2, stop the program, otherwise continue.

15. Set k to k – *1*. Go to step 2.

Ward's method shows that the distance between the clusters A and B is the amount by which the sum of squares would increase while merging

$$\Delta(A,B) = \sum_{i \in AUB} ||\overrightarrow{x}_i - \overrightarrow{m}_{AUB}||^2 - \sum_{i \in A} ||\overrightarrow{x}_i - \overrightarrow{m}_A|| - \sum_{i \in B} ||\overrightarrow{x}_i - \overrightarrow{m}_B||$$

where m_j is the center of cluster j and n_j is the number of points. Δ is the cost to combine A and B.

The sum of the squares begin at zero growing slowly as the clusters are merged. It follows a greedy approach to form the clusters.

In hierarchical clustering, to calculate the distance, any valid distance measure can be used. So, Ward's method for using the objective function permits the user's choice of function that reflects the purpose of clustering. Just like DBSCAN, this method does not need preceding knowledge of the number of clusters required.

In some cases, Ward's method could be sensitive to noise in the data and will not perform well with clusters of varied sizes. It also has a very slow running time of $O(n^2 \log n)$.

3.3 PEEKING2

PEEKING2 is an algorithm which condenses the data set to create a summary of the data, making it easy for business users to review. It accomplishes this by using feature reduction and row reduction techniques on the raw data set. PEEKING2 was developed by Papakroni [23] by extending IDEA, originally introduced by Menzies and Borges [3].

IDEA uses feature projection, clustering and feature selection on the project data set to reduce the data. It then visualizes this reduced data using rule reduction. This visualization shows and recommends how the effort can be reduced on the projects. PEEKING2 improves upon IDEA to accomplish the data reduction and support both defect prediction and effort estimation. The steps involved in PEEKING2 are

- **Feature Selection using INFOGAIN** To remove irrelevant features, PEEKING2 uses Information Gain, which is a technique that splits the training instances using the value of the feature. This assigns the ranks on the attributes with respect to the class variable. The selection method is to simply choose the ones with highest information gain.

- **Projection via FASTMAP** The FASTMAP is an algorithm for projecting the feature in an n-dimensional space in the direction where there is highest variability. This is achieved in linear time.

- **Grid Clustering** The Grid clustering uses recursion to large clusters into smaller chunks and associate them with each higher level cluster creating a tree.

- **Plotting** PEEKING2 calculates the centroid of each of these clusters formed in the Grid Clustering step. The centroids are plotted to show defective and non-defective modules and the purity of each module. This creates a visual that is very easy for business users to comprehend quickly and make decisions using these.

The algorithm was applied to a data set of size 800+ rows and 20+ columns, resulting in 10 to 30 rows and 6 columns, which is 2.5% of the original rows and 25% of the original number of columns. Even with so much reduction, it was proven to be have the same performance or better performance than NaÃŕve Bayes and Random Forest in defect prediction and Linear Regression and M5P in effort estimation.

The algorithm is extremely fast with output running times as low as milliseconds with a reasonable computational capabilities (3 GHz and 4 GB RAM) on 10 data sets, with the running time only increasing linearly with larger data sets.

The algorithm has a very good performance in comparison to other learners. It is an improvement from IDEA, in that it provides defect prediction as well, uses k Nearest Neighbor (kNN), uses intrinsic dimensionality to make decisions, uses local learning to build predictive models.

There has been some experiments which show PEEKING2 under performs with some direct relationship to the quality of the features (purity of the clusters). This necessitates some analysis of the data to determine whether it is recommended to use PEEKING2 to make decisions. The performance and effectiveness of PEEKING2 can also be affected by the

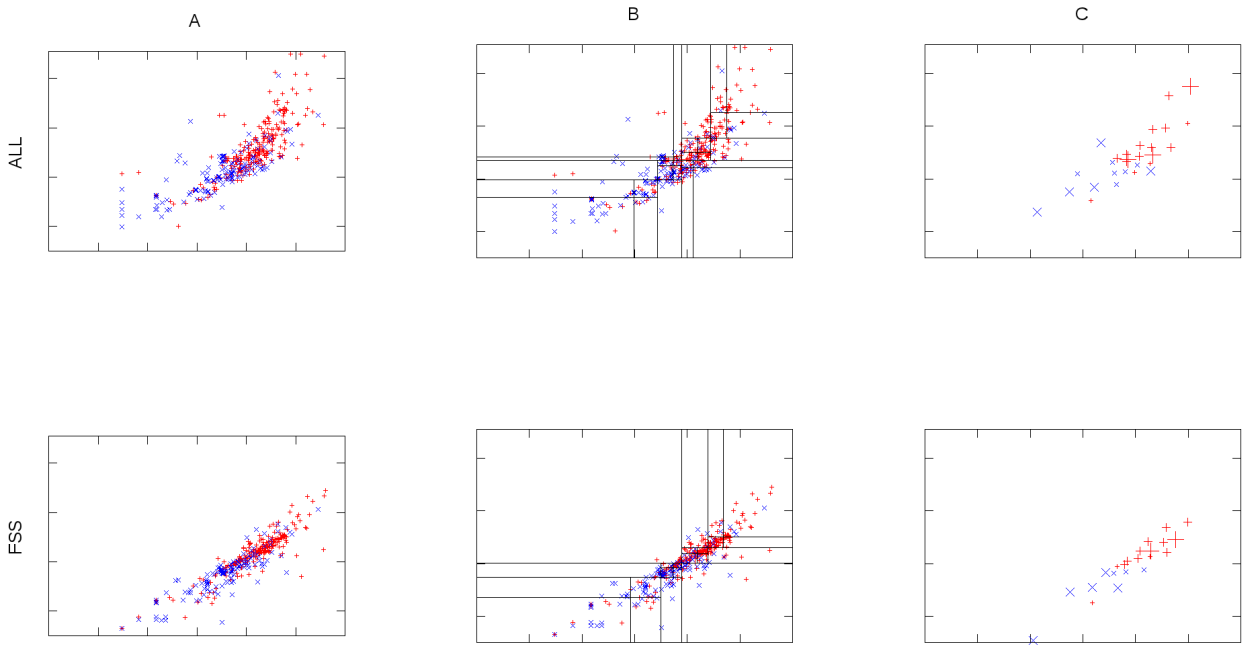

Figure 3.2: *Example of FastMap projection, Grid-Clustering, and centroids estimation applied on POI-3.0 data set that contains 442 examples.* ***Top Row****: Data transformations applied on the entire set of attributes. Each point represents a 20-dimensional instance.* ***Bottom Row****: Data transformations applied on 25% of the attributes with the highest information gain). Each point represents a 5-dimensional instance.* ***Column A (left-hand-side)****: Raw data projected into the first 2 dimensions found by FastMap. Blue points denote non-defective modules. Red points denote defective modules.* ***Column B (middle)****: The regions of data found in the leaves of a recursive grid-clustering. Each final cluster contains no more than* $2 * \sqrt{442} \approx 40$ *instances.****Column C (right-hand-side)****: after grid-clustering, each cluster is represented by its centroid. Blue points denote clusters with a majority of non-defective modules. Red points denote clusters with a majority of defective module. The size of each point is in proportion to the "purity" of the respective cluster. Note that the hundreds of original data points are have been now condensed to a few dozen centroids.*

choice of learners used to compare PEEKING2 with. Where PEEKING2 might work better in most cases on most data sets, it might not perform in comparison with some learners on some data sets.

3.4 Data Sets

The data sets used in the study were taken from PROMISE repository, a widely used source for software engineering mining data sets. When multiple versions of a data set were available in the repository, the latest version of the data set was used for the study.

The defect data sets contains information of open source JAVA projects. Table 3.1 lists the total number of instances, number of defective instances and percentage of defective instances in each data set. Each instance in the data set provides information about a class in the project.

Table 3.1: *Description of Defect Data Sets*

Data set	**# Instances**	**# Defects**	**% Defects**
ant-1.7	745	166	22
ivy-1.1	111	63	57
jedit-4.1	312	79	25
log4j-1.1	109	37	34
lucene-2.4	340	203	60
poi-3.0	442	281	64
synapse-1.2	256	86	34
velocity-1.6	229	78	34
xalan-2.6	885	411	46
xerces-1.4	588	437	74

There are 20 independent attributes and a boolean dependent attribute in each instance. A short description of the features is provided in Table 3.2. The dependent variable indicates whether any defect was found in a module post-release of the project. The data sets contain attributes based on Chidamber and Kemerer(CK) metrics designed to capture static features

of programs developed with Object Oriented methodology [6].

Table 3.2: *Description of attributes in the defect data sets*

amc	average method complexity	e.g. number of JAVA byte codes
avg_cc	average McCabe	average McCabe's cyclomatic complexity seen in class
ca	afferent couplings	how many other classes use the specific class.
cam	cohesion amongst classes	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.
cbm	coupling between methods	total number of new/redefined methods to which all the inherited methods are coupled
cbo	coupling between objects	increased when the methods of one class access services of another.
ce	efferent couplings	how many other classes is used by the specific class.
dam	data access	ratio of the number of private (protected) attributes to the total number of attributes
dit	depth of inheritance tree	
ic	inheritance coupling	number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an instance variable.
locm3	another lack of cohesion measure	if m, a are the number of $methods, attributes$ in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $lcom3 = ((\frac{1}{a}\sum_j^a \mu(a_j)) - m)/(1-m)$.
loc	lines of code	
max_cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
mfa	functional abstraction	number of methods inherited by a class plus number of methods accessible by member methods of the class
moa	aggregation	count of the number of data declarations (class fields) whose types are user defined classes
noc	number of children	
npm	number of public methods	
rfc	response for a class	number of methods invoked in response to a message to the object.
wmc	weighted methods per class	
defect	defect	Boolean: where defects found in post-release bug-tracking systems

The effort estimation data sets contain information about software projects and the ac-

tual effort spent to develop them. Table 3.3 lists the effort data sets used in the study. The COCOMO, NASA and data sets derived from them contains metrics defined by the COCOMO effort estimation model developed by Bohems [1]. Table 3.4 provides the description of various factors affecting the effort required to develop a software. These factors are labeled as effort multipliers in the COCOMO model. Actual development effort is linearly dependent on the multipliers. In addition to the effort multipliers listed in the table, Line of code (LOC) affects the effort exponentially.

Table 3.3: *Description of attributes in the effort data sets*

			Effort			
Data set	**# Features**	**# Instances**	**Min**	**Median**	**Mean**	**Max**
china	16	499	26	1829	3921	54620
cocomo81	18	63	6	98	683	11400
cocomo81e	17	28	9	354	1153	11400
cocomo81o	17	24	6	46	60	240
cocomo81s	17	11	6	156	850	6400
miyazaki	8	48	6	38	87	1586
nasa93	21	93	8	252	624	8211
nasa93c1	20	12	24	66	140	360
nasa93c2	20	37	8	82	223	1350
nasa93c5	20	40	72	571	1011	8211

The other data sets, china and miziyaki are not based on COCOMO model. Data set china is based on Function Point Analysis metrics, while miziyaki provides information on COBOL software development.

The dependent feature of the data set, effort is a continuous variable denoting actual effort measured in man-months with an exception of china. The measure of effort used in china is man-hours. Man-hours is defined as a unit of one hour's work by a person. A man-month is consists of 160 man-hours.

Table 3.4: *COCOMO effort multipliers. Taken from [21]*

upper: increase these to decrease effort	acap: analysts capability
	pcap: programmers capability
	aexp: application experience
	modp: modern programming practices
	tool: use of software tools
	vexp: virtual machine experience
	lexp: language experience
middle	sced: schedule constraint
lower: decrease these to increase effort	data: data base size
	turn: turnaround time
	virt: machine volatility
	stor: main memory constraint
	time: time constraint for cpu
	rely: required software reliability
	cplx: process complexity

3.5 Experimental Design

The experimental design of the study attempts to identify most suitable clustering technique to be used for instance reduction step and also determine ideal number of clusters or instances to be retained in the reduced data set.

The performance of clustering techniques were assessed by employing stratified M × K cross-validation. Herein K-fold cross-validation is repeated M times. In this study, 5 × 5 cross-validation was used. This mitigates the impact of the order of instances that is shown to have on the performance of some techniques [12].

In each run, a training set and test set are generated. The training set contains approximately fourth-fifth of the instances present in the data set. The following steps are executed

- **Feature Reduction**

  Feature Reduction is achieved by feature selection using InfoGain. Features are ranked based on Information Gain value and top 25% of features are retained. For applying InfoGain the both independent and dependent features needs to be discrete. Hence, any continuous valued feature is discretized. In case of effort estimation, the class

variable is discretized by creating two classes, instance with effort above median value of effort across the data set are placed grouped together. Independent attributes are discretized using Fayyad-Irani method. The algorithm recursively partitions the continuous attribute values in such a way that maximizes the Information Gain of the attributes [11].

- **Instance Reduction**

  After number of columns are reduced, the data is clustered using the clustering algorithms described in 3.2. The goal is to find one exemplar per cluster, this is done by finding the centroid of the instances present in a cluster. In case of continuous or numerical attribute, mean of the attribute value is calculated while mode is calculated for discrete or nominal attributes.

  For defect data, the class attribute defect value is a boolean attribute that denotes whether the majority of instances in the cluster are defective or not. Additionally, a numerical attribute defect rate is calculated to represent the fraction of defective instances in the cluster. For effort data, the class attribute is the average of actual development effort of all instances in the cluster.

  This effectively reduces the number of instances in the reduced data set to the number of clusters form by the clustering techniques.

- **Prediction**

  K-Nearest Neighbor is used to predict the class of an instance based on the class value of the k nearest centroids. The K-NN computes a weighted average of defect value of the centroids. If this value is greater or equal to a threshold, the instance is classified as defective.

  Instances in the test data were classified by extrapolating values of the condensed data set by applying K-NN (k=2) learner with a threshold of 0.5. In case of defect data, test instance is either classified as defective or non-defective. In addition to classifying an instance, the K-NN can also compute probability of defectiveness of an instance by

calculating the weighted average of the defect rate of the neighboring centroids. This can be useful to assess the confidence of the defect prediction. For effort data, the development effort for a given test instance was estimated.

Selection of parameter values

The selection of parameter values influences the performance of the clustering technique.The clustering techniques used in the study (with an exception of DBSCAN) require the number of clusters to be provided as an parameter. Hence, it is important to determine appropriate value for number of clusters to generated.

Experiments were conducted using the same methodology outlined in this section to determine optimal values for the parameters of the clustering techniques. The number of clusters (k $\bar{5}$, 10, 25, 50) were evaluated for K means, Expectation maximization, Mini Batch K-Means and Ward.

For DBSCAN, the following ϵ values: 0.01, 0.05, 0.1, 0.2 were chosen. The parameter values were then selected based on the median f-measure and rank obtained in experimental runs.

Implementation

The implementation of the experiments was done using Python. The clustering methods and learners were used from SciKit-Learn or Weka. SciKit-Learn is a python package which provides implementation of various machine learning algorithms. Weka is a java based data mining tool kit. Weka offers command line invocation of its methods which can be used to run clustering algorithms from a python program. Standard implementations of clustering techniques and learners were used to avoid potential errors in implementation and also ensure that the study can be repeated. PEEKING2 used in the study was implemented by Vasil Papakroni for his thesis work [23]. The data structure for storing the input data and supporting library functions were also reused from the work.

3.6 Statistical Methods

The definitions of the measures employed to compare the performance of the clustering methods can be found in Table 3.6. The definitions are based on the confusion matrix show in Table 3.5.

Table 3.5: *Confusion Matrix*

		Actual	
		Non-Defective	***Defective***
Predicted	***Non-Defective***	TN	FN
	Defective	FP	TP

Table 3.6: *Performance Metrics*

Measure	**Formula**
Recall (pd)	TP/(TP+FP)
Precision (prec)	TP/(TP+FN)
F-Measure	(2*pd*prec)/(pd+prec)
Accuracy	(TP+TN)/(TP+TN+FP+FN)

- **Recall (pd)** represents the probability of detection of a defective module.
- **Precision (prec)** is the probability that a module identified as defective is actually defective.

Ideally, a good defect predictor should have both high recall and high precision. However, an attempt at improving one of these measures might result in less than ideal value in the other measure. Hence, in our study we use f-measure, which is calculated from both pd and prec as the primary metric to compare the performance of clustering techniques.

For effort estimation, absolute residual error and magnitude of relative error is calculated. The definition of the measures are as follows:

- *Absolute Residual error* $(AR) =\mid actual_i - predicted_i \mid$
- *Magnitude of Relative Error* $(MRE) = \frac{|actual_i - predicted_i|}{actual_i}$

where $actual_i$ is the actual effort of a given test instance and $predicted_i$ is the estimated effort.

The Absolute Residual error measures the error in estimation in actual effort units, while Magnitude of Relative Error measures the estimation error in proportion to the actual effort of the project.

The clustering techniques are then ranked based on f-measure for defect data on values obtained over the 25 cross-validation runs(5 repeats $\times$ 5 folds).

Win-Tie-Lose based on Wilcoxon statistical method was used to compare the populations of f-measure values of the clustering algorithms as shown below.

Algorithm 1: Algorithm for Ranking learners based on F-Measure

```
if WILCOXON(Pi, Pj) > 95 or equals(f_i,f_j)
then
    tie_i = tie_i + 1;
    tie_j = tie_j + 1;
else
    if greater(f_i,f_j) then
      win_i= win_i + 1
      loss_j = loss_j + 1
    else
      win_i = win_j+ 1
      loss_i = loss_i + 1
    end if
end if
```

If two populations P_i, P_j (with median f-measures f_i and f_j respectively) are not statistically different according to Wilcoxon test at 95% confidence or if their median f-measures are equal, P_i and P_j are said to be tied. If not, the population with greater value for median f-measure is said to have won. The clusterers are then ranked based on the number of losses (lower values are ranked the better). The rank of a clustering method enables us to evaluate whether its observed advantage is statistically significant or not.

For effort estimation the ranking is done based on Absolute Error and Magnitude of Relative Error instead of f-measure in the above pseudo code.

Chapter 4

Results

The results of the empirical study are discussed in this chapter.

4.1 Defect Prediction

Number of Clusters

When evaluating the choice of the number of clusters, the predictive ability of the reduced data sets with 2-NN algorithm in 5×5 experimental run were compared individually for each clustering algorithm. The ranks were calculated based on median f-measure using the ranking algorithm described in Section 3.For each clustering algorithm and number of clusters, Table 4.1 lists the number of times it was ranked in the given position. The number number of clusters is diplayed in rows and ranks in columns. In case of DBSCAN, the different ϵ values are listed.

The table 4.1 shows that for all but one clustering method the number of clusters, N = 25 was consistently ranked higher than the other values of N. It is seen that increasing

Table 4.1: *Comparing ranks of clustering techniques with different number of clusters*

K-Means	**# Clusters**	**Rank1**	**Rank2**	**Rank3**	**Rank4**
	5	4	1	4	1
	10	5	3	2	0
	25	7	2	1	0
	50	7	2	0	1
Mini Batch K-Means	**# Clusters**	**Rank1**	**Rank2**	**Rank3**	**Rank4**
	5	1	4	2	3
	10	2	5	3	0
	25	4	4	2	0
	50	6	1	2	1
EM	**# Clusters**	**Rank1**	**Rank2**	**Rank3**	**Rank4**
	5	1	2	3	4
	10	6	1	2	1
	25	7	2	1	0
	50	4	3	2	1
Ward	**# Clusters**	**Rank1**	**Rank2**	**Rank3**	**Rank4**
	5	3	3	1	3
	10	4	4	2	0
	25	7	0	2	1
	50	3	5	2	0
DBSCAN	**Epsilon**	**Rank1**	**Rank2**	**Rank3**	**Rank4**
	0.01	2	3	3	2
	0.05	2	3	3	2
	0.1	4	4	2	0
	0.2	5	2	1	2

value of number of clusters beyond 25 often resulted in decrease in the performance of model trained over the reduced data set.

Consequently, for comparing different clustering techniques, the number of clusters to be generated was set to 25 based on the above results. The ϵ value for DBSCAN was chosen to be 0.1. This choice was based on the total number of times a parameter value was ranked first or second.

Data Reduction

High degree of data reduction was achieved by application of InfoGain along with clustering techniques.Table 4.2 shows data reduction achieved on the training data set when the number of clusters equals 25 for Expectation Maximization, K-Means, Mini Batch K-Means and Ward

clustering techniques. Table 4.3 and 4.4 lists the data reduction on the training data set using PEEKING2 and DBSCAN respectively(These models do not have a fixed number of clusters). The tables display the number of instances in the training data set (# Instance and in reduced data set (# Clusters), percentage decrease in instances, followed by number of cells in training data (# Cells), percentage decrease in cells after data reduction is applied. The size of the training data set is four-fifth of the actual data set.

Table 4.2: *Defect data reduction k=25 number of clusters and InfoGain(25%))*

Data set	**# Instances**	**% Reduction**	**# cells**	**% Reduction**
ant-1.7	596	96	11920	99
ivy-1.1	88	72	1760	91
jedit-4.1	249	90	4980	97
log4j-1.1	88	72	1760	91
lucene-2.4	272	91	5440	97
poi-3.0	354	93	7080	98
synapse-1.2	205	88	4100	96
velocity-1.6	183	86	3660	96
xalan-2.6	708	96	14160	99
xerces-1.4	471	95	9420	98
	Median	90	***Median***	97

Table 4.3: *Defect data Reduction using PEEKING2*

Data set	**# Instances**	**# Centroids**	**% Reduction**	**# cells**	**% Reduction**
ant-1.7	596	28	95	11920	99
ivy-1.1	88	19	78	1760	95
jedit-4.1	249	16	94	4980	98
log4j-1.1	88	19	78	1760	95
lucene-2.4	272	19	93	5440	98
poi-3.0	354	21	94	7080	99
synapse-1.2	205	16	92	4100	98
velocity-1.6	183	16	91	3660	98
xalan-2.6	708	28	96	14160	99
xerces-1.4	471	21	96	9420	99
		Median	93	***Median***	98

The data reduction is quantified by total instances and by total information (instances

Table 4.4: *Defect data reduction using DBSCAN(ϵ = 0.1)*

Data set	# Instances	# Centroids	% Reduction	# cells	% Reduction
ant-1.7	596	28	95	11920	99
ivy-1.1	88	19	78	1760	95
jedit-4.1	249	16	94	4980	98
log4j-1.1	88	19	78	1760	95
lucene-2.4	272	19	93	5440	98
poi-3.0	354	21	94	7080	99
synapse-1.2	205	16	92	4100	98
velocity-1.6	183	16	91	3660	98
xalan-2.6	708	28	96	14160	99
xerces-1.4	471	21	96	9420	99
		Median	93	***Median***	98

× attributes). For DBSCAN and PEEKING2, the median instance reduction was 93% and median total information reduction was 98%. For other clusters, the median instance reduction was 90% and median total information reduction was 97%. It is essential to ensure there is no significant information loss due to the process.

Performance of Defect Prediction Models

The performance of the reduced data sets with 2-NN algorithm in 5×5 cross validation runs is shown in Table 4.5. For each data set, the results are sorted in decreasing order of f-measure. The performance of Random Forest trained on the training data set is used as the baseline. No instance or feature selection was applied for baseline result. While comparing the performance, we use the term "almost equal" or "close" to denote that the median f-measure is less than 2% or 5% different respectively.

- For 8 out of 10 data sets, the predictive ability of 2-NN learner on at least two of the reduced data sets are better than or almost equal to the predictive ability of RF. For the other two data sets, synapse-1.2 and xalan-2.6, RF was clearly the winner.

Table 4.5: *Comparison of Defect Predictive ability of models based on different algorithms*

Data	*Learner*	*Instances*	*Prec*	*PD*	*F*	*Rank*	*Data*	*Learner*	*Instances*	*Prec*	*PD*	*F*	*Rank*
ant-1.7	K-Means	25	0.65	0.44	0.53	1	***poi-3.0***	RF	354	0.83	0.86	0.84	1
	MB K-Means	25	0.62	0.41	0.51	2		Ward	25	0.86	0.82	0.84	1
	PEEKING2	28	0.67	0.41	0.51	3		MB K-Means	25	0.86	0.82	0.83	1
	E-Max	25	0.63	0.42	0.5	2		K-Means	25	0.85	0.82	0.82	4
	Ward	25	0.64	0.42	0.5	2		E-Max	25	0.84	0.82	0.82	5
	RF	596	0.56	0.44	0.48	5		DBSCAN	17	0.83	0.82	0.81	6
	DBSCAN	15	0.64	0.36	0.48	6		PEEKING2	21	0.84	0.82	0.81	6
ivy-1.1	PEEKING2	19	0.69	0.69	0.72	1	***synapse-1.2***	RF	205	0.62	0.56	0.59	1
	MB K-Means	25	0.71	0.69	0.72	1		MB K-Means	25	0.6	0.47	0.54	2
	E-Max	25	0.73	0.69	0.71	1		K-Means	25	0.62	0.47	0.53	3
	K-Means	25	0.73	0.75	0.71	2		PEEKING2	16	0.64	0.41	0.52	4
	Ward	25	0.73	0.69	0.71	1		E-Max	25	0.59	0.44	0.52	3
	RF	88	0.69	0.69	0.69	1		Ward	25	0.6	0.41	0.5	5
	DBSCAN	6	0.71	0.5	0.6	7		DBSCAN	10	0.62	0.29	0.42	7
jedit-4.1	E-Max	25	0.64	0.5	0.58	1	***velocity-1.6***	DBSCAN	9	0.53	0.5	0.56	1
	MB K-Means	25	0.67	0.5	0.57	2		RF	183	0.55	0.5	0.54	2
	Ward	25	0.67	0.5	0.56	2		Ward	25	0.67	0.5	0.53	1
	RF	249	0.62	0.47	0.55	4		K-Means	25	0.58	0.44	0.52	2
	K-Means	25	0.64	0.5	0.55	4		MB K-Means	25	0.64	0.44	0.5	2
	DBSCAN	7	0.67	0.47	0.52	5		E-Max	25	0.6	0.44	0.48	4
	PEEKING2	16	0.67	0.44	0.5	5		PEEKING2	16	0.58	0.38	0.48	7
log4j-1.1	E-Max	25	0.75	0.62	0.67	1	***xalan-2.6***	RF	708	0.76	0.7	0.72	1
	K-Means	25	0.75	0.57	0.67	1		DBSCAN	32	0.8	0.57	0.68	2
	MB K-Means	25	0.8	0.57	0.67	1		MB K-Means	25	0.8	0.59	0.67	2
	Ward	25	0.8	0.57	0.67	1		E-Max	25	0.81	0.57	0.67	2
	RF	88	0.71	0.62	0.63	5		K-Means	25	0.82	0.57	0.66	4
	PEEKING2	19	0.75	0.43	0.6	6		Ward	25	0.81	0.56	0.66	4
	DBSCAN	5	0	0	0	7		PEEKING2	28	0.77	0.57	0.66	7
lucene-2.4	DBSCAN	8	0.7	0.8	0.74	1	***xerces-1.4***	RF	471	0.94	0.97	0.95	1
	K-Means	25	0.7	0.78	0.74	1		E-Max	25	0.92	0.98	0.95	2
	MB K-Means	25	0.71	0.76	0.73	3		PEEKING2	21	0.91	0.98	0.95	3
	RF	272	0.71	0.73	0.73	4		Ward	25	0.92	0.98	0.95	3
	E-Max	25	0.7	0.76	0.73	4		MB K-Means	25	0.91	0.98	0.94	4
	Ward	25	0.7	0.78	0.73	4		K-Means	23	0.92	0.98	0.94	6
	PEEKING2	19	0.7	0.76	0.73	6		DBSCAN	18	0.91	0.97	0.94	7

- For 5 out of 10 data sets, the predictive ability of 2-NN learner on all reduced data sets are close to each other, i.e. the difference in F measure between the highest and lowest was less than 5%.

- For 8 out of 10 data sets, performance of the clustering algorithm with fixed number of clusters (n = 25), are close to each other. Large variations in performance are seen with DBSCAN and PEEKING2 where the number of clusters formed cannot be specified directly and is based on other parameters. In particular, DBSCAN performed very poorly in log4j-1.1.

The above results indicate that the performance of models trained on reduced data sets is comparable to standard learners for most data sets.

The data reduction technique can be used to summarize data and aid in data visualization. Table 4.6 shows the condensed data set of Ant-1.7 obtained by applying InfoGain(25%) and K-Means(k=25). The training data set consists of 596 instances and 20 features. In the table size displays the number of instances that the were grouped together in the cluster. The values for the top 25% of features (cluster centroid) is listed along with the defect classification and defect rate of each cluster.

Table 4.6: *Ant-1.7 data set condensed using InfoGain(25%) and K-Means(k=25)*

cluster	size	loc	rfc	cam	wmc	lcom	defect	defect rate
1	45	723	89	0	29	132	1	0.76
2	10	50	13	1	5	8	0	0.20
3	24	447	63	0	26	133	0	0.42
4	51	180	27	0	8	4	0	0.14
5	15	672	67	0	19	115	0	0.47
6	28	258	41	0	21	128	0	0.29
7	14	263	37	1	5	4	0	0.43
8	26	15	5	1	2	1	0	0.08
9	19	719	90	0	26	133	1	0.74
10	40	222	36	0	14	65	0	0.18
11	40	184	27	0	6	5	0	0.15
12	18	597	80	0	20	49	1	0.61
13	15	551	66	0	11	18	1	0.60
14	21	96	16	1	8	15	0	0.00
15	42	310	42	0	12	14	0	0.26
16	4	673	12	0	5	1	0	0.00
17	52	35	8	1	4	2	0	0.00
18	38	128	22	0	10	28	0	0.18
19	21	395	51	0	16	61	1	0.57
20	16	116	21	1	3	3	0	0.19
21	69	90	16	0	5	4	0	0.09
22	8	16	0	0	0	0	0	0.00
23	38	29	8	1	4	4	0	0.03
24	32	110	18	1	3	2	0	0.03
25	59	8	3	1	1	0	0	0.03

4.2 Effort Estimation

Number of Clusters

The size of the data sets used in Effort Estimation is much smaller than the Defect Set used. Hence, the evaluation of number of clusters, N = 5, 10, 25 ,50 was not feasible. Additionally, given the large variability in the size, the attempt to find a single value for number of clusters yielded in poor performance of the learned model. Consequently, the following approach was used, if the total instances in training data set was less than or equal to 30, the number of clusters was set to half of total instances. In all other cases, 25 clusters were generated.

Data Reduction

Table 4.7 shows data reduction achieved by E-Max and K-Means along with feature selection using InfoGain(25%).Table 4.8 shows data reduction by Peeking2 along with feature selection using InfoGain(25%).

Table 4.7: *Effort data reduction when k=25 for K-Means and E-Max*

Data set	# Instances	# Centroids	% Reduction	# cells	% Reduction
china	399	25	94	6384	98
cocomo81	50	25	50	900	86
cocomo81e	19	9	53	342	87
cocomo81o	19	9	53	342	87
cocomo81s	9	4	56	162	88
miyazaki	38	25	34	304	84
nasa93	74	25	66	1554	92
nasa93c1	10	5	50	210	88
nasa93c2	30	15	50	630	88
nasa93c5	31	25	19	651	81
		Median	51	***Median***	87

The tables list the total instances in training data set, total instances in reduced data set(# Centroids), percentage reduction in number of instances, Cells in the training data set

Table 4.8: *Effort data reduction using PEEKING2*

Data set	# Instances	# Centroids	% Reduction	# cells	% Reduction
china	399	19	95	6384	99
cocomo81	50	16	68	900	91
cocomo81e	19	10	47	342	85
cocomo81o	19	9	53	342	87
cocomo81s	9	6	33	162	81
miyazaki	38	15	61	304	90
nasa93	74	17	77	1554	95
nasa93c1	10	6	40	210	86
nasa93c2	30	10	67	630	92
nasa93c5	31	9	71	651	93
		Median	64	***Median***	91

(Number of Instances × Numner of features) and the percentage reduction in total number of cells of the data set.

The median instance reduction is 51% for K-Means and E-Max and 64% for PEEKING2. The median information reduction is 87% for K-Means and E-Max and 91% for PEEKING2. The median value for instance reduction for effort data much less when compared to instance reduction of defect data.

Performance of Effort Estimation Models

The tables 4.9 and 4.10 display the performance of effort estimation models generated by 2-NN trained on reduced data. The performance of M5P and LR are used as baseline. For each learner, Actual Residual Error (AR) and Magnitude of Relative Error (MRE) are calculated from 25 experimental runs. For both measures, the table lists the median value for the measure, the Inter Quartile Range (IQR), Spread and Rank obtained when compared to other learners.

Table 4.9 lists the data sets were models based on reduced data set performs better or comparable to M5P and LR. Table 4.10 lists data sets where M5P performed better than

Table 4.9: *Results of Effort Estimation- Part 1*

			AR				*MRE*			
Data	***Learner***	***# Instances***	***Median***	***IQR***	***Spread***	***Rank***	***Median***	***IQR***	***Spread***	***Rank***
cocomo81	K-Means	25	92.17	265.51	10717.36	1	0.71	1.73	27.86	1
	E-Max	20	101.63	364.54	10835.75	2	0.79	2.78	45.83	2
	PEEKING2	16	177.63	639.45	10900.76	3	0.92	3.89	161.61	3
	M5P	50	180.31	447.70	10038.28	3	1.49	4.79	92.87	4
	DBScan	2	417.43	374.32	11135.59	5	2.67	11.75	124.45	5
	LR	50	546.53	786.36	8734.68	6	4.26	15.78	310.50	6
cocomo81e	K-Means	9	283.26	558.42	10480.05	1	0.73	1.84	58.74	1
	E-Max	9	307.24	693.95	10561.66	1	0.81	2.42	66.06	1
	M5P	19	330.26	806.73	10968.17	3	0.85	2.97	49.58	3
	PEEKING2	10	352.72	571.88	23707.60	3	0.86	2.09	61.26	3
	DBScan	1	865.75	725.83	10661.83	5	1.74	7.77	168.34	5
	LR	19	1588.05	4198.74	15321.74	6	3.08	12.84	495.35	6
cocomo81o	E-Max	9	22.63	30.67	178.83	1	0.47	0.86	18.68	1
	PEEKING2	9	23.61	32.12	160.56	1	0.48	0.95	7.44	2
	DBScan	1	23.73	33.33	192.84	2	0.50	2.01	9.80	3
	K-Means	9	25.73	37.55	193.08	4	0.50	0.79	18.53	3
	M5P	19	34.22	35.18	239.26	5	0.54	1.16	10.28	5
	LR	19	38.67	38.67	180.09	6	0.89	1.42	17.91	5
cocomo81s	PEEKING2	6	182.93	877.68	6381.10	1	1.00	4.94	137.31	1
	E-Max	4	365.66	876.09	6381.54	2	1.49	8.84	236.84	2
	LR	9	410.29	2274.60	6114.05	3	3.42	25.17	173.67	3
	K-Means	4	869.81	1277.30	6505.92	4	3.90	11.75	121.95	3
	M5P	9	944.91	648.82	6059.96	4	4.68	26.31	126.34	5
	DBScan	1	1112.36	1516.47	6241.84	6	5.37	68.11	195.65	5
miyazaki	K-Means	25	14.11	22.01	1448.45	1	0.43	0.61	8.33	1
	PEEKING2	15	14.40	23.63	1461.68	2	0.43	0.61	10.87	1
	E-Max	18	14.78	27.37	1415.87	3	0.45	0.61	8.33	3
	LR	38	19.79	24.99	1561.02	3	0.51	0.58	7.72	4
	DBScan	2	27.12	42.35	1692.27	5	0.60	1.18	17.13	5
	M5P	38	28.47	47.31	1539.36	5	0.67	1.06	9.56	5

the other models. However, it can be seen that the performance of 2-NN trained on reduced data sets is much better than LR.

The performance of DBSCAN was much poorer compared to other models. This can be attributed to the very few clusters generated by the technique. However, no appropriate value for parameters were found during the experimental study.

The IQR shows the consistency of the estimation made by the models in the experimental runs. It can be seen that the IQR of the clustering methods are comparable to M5P.

Table 4.10: *Results of Effort Estimation- Part 2*

			AR				***MRE***			
Data	***Learner***	***Instances***	***Median***	***IQR***	***Spread***	***Rank***	***Median***	***IQR***	***Spread***	***Rank***
nasa93	M5P	74	122.73	303.77	6595.05	1	0.49	0.80	43.64	1
	K-Means	25	140.79	455.21	6935.43	2	0.57	1.21	50.94	2
	DBScan	4	171.21	350.93	7834.44	3	0.68	0.57	41.87	2
	PEEKING2	17	202.79	607.04	7094.54	4	0.72	1.70	157.34	4
	E-Max	18	225.36	532.60	7265.87	4	0.72	1.89	108.11	5
	LR	74	501.71	326.56	7686.16	6	1.48	8.02	81.04	6
nasa93c1	M5P	10	35.80	163.57	302.16	1	0.33	0.30	0.74	1
	E-Max	3	37.48	72.09	140.52	2	0.47	2.01	7.24	2
	PEEKING2	6	79.72	140.05	317.37	3	0.54	0.60	7.49	2
	K-Means	5	91.96	140.31	245.62	4	0.64	0.93	6.13	3
	LR	10	96.17	108.36	260.46	5	1.15	1.08	5.67	5
	DBScan	1	104.60	83.72	203.77	6	1.17	1.57	5.83	6
nasa93c2	M5P	30	31.61	117.93	1195.57	1	0.37	0.99	52.93	1
	K-Means	15	52.75	160.48	1199.90	2	0.61	0.73	36.14	2
	PEEKING2	10	57.92	196.95	1287.23	3	0.68	0.94	25.74	3
	DBScan	3	86.86	235.85	1236.27	4	0.70	1.22	24.81	4
	E-Max	11	87.69	151.75	1213.81	5	0.70	1.71	36.41	4
	LR	30	172.91	93.04	1152.94	6	1.93	3.77	29.24	6
nasa93c5	M5P	31	342.00	726.83	6433.71	1	0.69	1.24	15.70	1
	K-Means	25	396.07	807.29	6457.50	1	0.69	1.06	11.52	2
	PEEKING2	9	402.32	603.40	7114.48	1	0.70	1.26	13.96	2
	DBScan	2	543.66	592.18	7617.42	4	0.78	1.35	13.59	4
	E-Max	11	620.37	912.69	6813.94	5	0.81	2.12	13.81	5
	LR	31	703.86	525.04	7558.40	5	0.93	2.40	15.33	6
china	M5P	399	188.90	457.09	22528.31	1	0.12	0.20	23.24	1
	K-Means	25	492.84	1282.75	50316.80	2	0.30	0.50	18.84	2
	E-Max	25	512.88	1191.86	50523.86	3	0.31	0.46	20.18	3
	DBScan	17	627.29	1530.02	50672.55	4	0.42	0.48	21.78	4
	PEEKING2	19	849.83	2417.19	49385.47	5	0.45	0.78	31.55	5
	LR	399	853.76	1624.88	45988.81	6	0.49	0.83	40.35	6

Chapter 5

Threats to Validity

Empirical studies are susceptible treats to validity. It is essential to discuss the potential treats to validity of the conclusions the experimental study. This chapter discusses concerns of validity: construct validity, internal validity and external validity.

5.1 Construct Validity

Construct validity refers to whether the study, models and measures the intended metrics relevant to it. It is necessary to construct the experiment to study the parameters pertinent to the research questions.

This study attempts to effectively summarize software engineering data by discarding superfluous information present in it. Hence, information loss due to data reduction is an important metric to be studied. However, this information loss cannot be quantified hence the loss of information due to data reduction is gauged indirectly by the predictive ability of the 2-NN algorithm used over the condensed data set when compared with a standard learner trained with the original data set. The rationale being that any loss of information

in reduced data set would lead to poor predictive ability compared to the standard methods.

Consequently, the choice of the learner used as baseline influences the outcome of the experiments. Random Forest was chosen for defect data because it has been known to be successfully applied in software engineering data mining [13] [19]. However, there are other learners that are being widely used as well.

The other choices that might have influenced the results are the metrics used to measure the performance, statistical method employed to compare the predictive ability, the clustering algorithms used in the study, the number of clusters or other parameter values required by the clustering algorithms.

5.2 Internal Validity

Internal validity refers to risk due to presence of confounding variables that would explain the observed results. It is essential to identify extraneous variables the might influence the conclusion of the study.

Software Engineering research is more susceptible to selection bias [31]. The data used for study controls the outcomes of the result. However, researchers using data available in the mining repositories do not have control have over the data collection process. To avoid data quality issues, the data sets used in the study were taken from the PROMISE repository and are widely used in software research projects.

5.3 External Validity

External validity refers to ability to generalize the results of an empirical study to other subjects in the domain. Software development process varies greatly based on the development

paradigm, programming language etc.

The defect data sets used are based on open-source Java projects. The development process of open source projects are remarkably different than proprietary projects. Hence, the conclusion of the study may not hold for other kinds of software. Therefore, this empirical study should to be repeated with other data sets based on different development models and programming languages to be able to generalize the results.

Chapter 6

Conclusion

This chapter presents the conclusions drawn from the results of the empirical study and addresses the research questions raised. In summary, the study explores the impact of data reduction in Software Engineering data and evaluates the performance of various clustering techniques that can be used for instance reduction.

- *RQ1: What are the different clustering techniques that can be employed for efficient instance reduction and what are their impacts on predictive ability? Do some clustering methods perform significantly better than others?*

  Some of the common clustering methods used in data mining were included in the study. While selecting clustering techniques methods based on different categories like hierarchical clustering, spectral clustering, density based clustering etc were included. The performances of Expectation Maximization, K-Means, Mini-Batch K-Means and Ward were close to each other in most of the defect data sets. However, high variation in the performances of PEEKING2 and DBSCAN were observed. One explanation might be that these two clustering techniques on an average generate less number of clusters compared to the other clusters. Based on the results it can be concluded that when sufficient number of clusters are generated, there does not seem to be one

clustering technique that is clearly better than the rest.

For effort estimation, K-Means, Expectation Maximization, DBSCAN were evaluated along with PEEKING2. The performance of PEEKING2 was much better in effort estimation when compared with its performance in defect prediction. However, performance of DBSCAN was worse than other models used.

- *RQ2: Will large data reduction result in information loss and predictive ability of learners trained on them?*

  The goal of the study is reduce large defect data down to a concise form that can then be used to generate simple models. The concise data should also enable software engineering practitioners to analyze it based on expert knowledge. Given these goals, it is difficult to directly measure information loss. However, the performance of 2-NN algorithm over the condensed data set is comparable to the standard learner on most (8 out of 10) of the defect data sets or even better in some of the data sets.

  For effort estimation, in 5 out of 10 effort data sets, the models based on reduced data out performed or matched the accuracy of M5P and LR. In the rest, models built on condensed data sets fared better than LR. This indicates that essential information contained in the data set is not compromised by data reduction.

- *RQ3: What is the optimal number of clusters and how much data reduction is too much?*

  The number of clusters should be compact enough for viewing the data manually but large enough to retain essential information without information loss. Based on the experimental results, we found 25 to be the optimal value for the clustering techniques for defect data sets used in the study. This represents the median instance reduction of 90% and median information (instances×attributes) reduction of 97%. The original number of instances in the training data sets varied between 88 and 708.

  The size of the data sets used in Effort Estimation is much smaller than the defect data used. The original number of instances in the training data sets varied between

9 and 399. Given the very small size of some data sets, the attempts to find a single value for number of clusters yielded in poor performance of the learned model. The data reduction in smaller data sets is much less than what can be achieved with larger data sets.

The goal of the study is not to challenge the current state of the art learners but to evaluate and suggest techniques for effective data reduction without much loss in terms of predictive ability. The results of the experimental study concurs with the objectives of the study. While the results of the study are not conclusive, they are promising to warrant further research using the data reduction technique.

References

[1] N. Bettenburg, M. Nagappan, and A.E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 60 –69, June 2012.

[2] Raymond Borges and Tim Menzies. Learning to change projects. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, PROMISE '12, pages 11–18, New York, NY, USA, 2012. ACM.

[3] Raymond Borges and Tim Menzies. Learning to change projects. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, PROMISE '12, pages 11–18, New York, NY, USA, 2012. ACM.

[4] Raymond P. L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 987–996, Piscataway, NJ, USA, 2012. IEEE Press.

[5] Zhihao Chen, Barry Boehm, Tim Menzies, and Daniel Port. Finding the right data for software cost modeling. *IEEE Software*, 22(6):38–46, Nov 2005. Copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) Nov/Dec 2005; Document feature - graphs; tables; references; equations; Last updated - 2013-05-23; CODEN - IESOEG.

[6] N.I. Churcher, M.J. Shepperd, S. Chidamber, and C.F. Kemerer. Comments on 'a metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, March 1995.

[7] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of The Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[8] Martin Ester, Hans peter Kriegel, JÃűrg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[9] Vladimir Estivill-Castro. Why so many clustering algorithms: A position paper. *SIGKDD Explor. Newsl.*, 4(1):65–75, June 2002.

[10] Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 163–174, New York, NY, USA, 1995. ACM.

[11] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pages 1022–1029, 1993.

[12] Douglas H. Fisher, Ling Xu, and Nazih Zard. Ordering effects in clustering. In *Proceedings of the Ninth International Workshop on Machine Learning*, ML '92, pages 162–168, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[13] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM.

[14] Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.

[15] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003.

[16] Jiawei Han, Micheline Kamber, and Anthony K. H. Tung. Spatial clustering methods in data mining: A survey. In Harvey J. Miller and Jiawei Han, editors, *Geographic Data Mining and Knowledge Discovery, Research Monographs in GIS*. Taylor and Francis, 2001.

[17] Jairus Hihn, Karen Lum, Tim Menzies, Dan Baker, and Omid Jalali. 2cee, a twenty first century effort estimation methodology, 2008.

[18] Capers Jones. Software quality metrics: Three harmful metrics and two helpful metrics. Technical report, Namcook Analytics LLC, Narragansett, RI 02882, June 2012.

[19] E. Kocaguneli, T. Menzies, and J.W. Keung. On the value of ensemble effort estimation. *Software Engineering, IEEE Transactions on*, 38(6):1403–1416, Nov 2012.

[20] Huan Liu and Lei Yu. Toward integrating feature selection algorithms for classification and clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 17(4):491–502, April 2005.

[21] Tim Menzies, Zhihao Chen, Jairus Hihn, and Karen Lum. Selecting Best Practices for Effort Estimation. *IEEE Transactions on Software Engineering*, 32:883–895, 2006.

[22] K. Molokken and M. Jorgensen. A review of software surveys on software effort estimation. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 223–230, Sept 2003.

[23] Vasil Papakroni. Data carving: Identifying and removing irrelevancies in the data. Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia Unviersity, 2013.

[24] Chris Piech. K means, 2015.

[25] D. Rodriguez, R. Ruiz, J. Cuadrado-Gallego, J. Aguilar-Ruiz, and M. Garre. Attribute selection in software engineering datasets for detecting fault modules. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 418–423, Aug 2007.

[26] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1177–1178, New York, NY, USA, 2010. ACM.

[27] The Standish Group. `http://www.cs.nmt.edu/~cs328/reading/Standish.pdf` timestamp = 2009-08-29T18:12:13.000+0200, title = Chaos Report, year = 1995.

[28] Burak Turhan, Ayse Bener, and Tim Menzies. Regularities in learning defect predictors. In M. Ali Babar, Matias Vierimaa, and Markku Oivo, editors, *Product-Focused Software Process Improvement*, volume 6156 of *Lecture Notes in Computer Science*, pages 116–130. Springer Berlin Heidelberg, 2010.

[29] Burak Turhan, Ayse Bener, and Tim Menzies. Regularities in learning defect predictors. In *Proceedings of the 11th international conference on Product-Focused Software Process Improvement*, PROFES'10, pages 116–130, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[31] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 411–414, New York, NY, USA, 2010. ACM.

[32] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Bing Ng, Angus a nd Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, December 2007.

Appendix A

Code Listing

The Appendix provides source code implemented to conduct this emprical study. It has to be noted code provide is not complete and only major modules have been included in this section.

A.1 Experiment.py

This python file contains the implementation of functions necessary to conduct the experimental method describe in Section 3.5. The datasets and learners to be used are specified in the function *Experiment*. For each dataset, learners are run 25 times using different set of training data and test data. The training and testing data are derived from original dataset by stratified sampling. In any given run, all learners are trained and test the same sets of data. Performance is calculated and logged for each run of a learner for a given dataset. The performances of all the learners are compared and ranked for each dataset.

Experiments.py

```
def experiment():
        #This is the main function used to start the experimental runs
```

```
        srcDir = "data_arff/" #Path of Input datasets
        logDir = "logs/"      #Path for Output files and logs
        folds = 5             #Initializing number of folds
        repeats = 5           #Initializing number of repeats

        dataSets=['ant-1.7.arff','ivy-1.1.arff', 'jedit-4.1.arff', 'log4j-1.1.arff',
            'lucene-2.4.arff', 'poi-3.0.arff', 'synapse-1.2.arff', 'velocity-1.6.arff',
            'xalan-2.6.arff', 'xerces-1.4.arff']
        #The datasets used are specified
        learners = ['K-Means(25)','PEEKING2','DBSCAN(0.1)','Ward(25)','E-Max(25)','MB_
            K-Means(25)','RF']
        #The learners and clustering techniques to be used are specified
        classVal = "1"
        #For classification problem classVal contains the target class for which the
            performances values are printed. Defective data is indicated by 1
        strD = getCurrentDate()
        generalLogFile = open(logDir + "log_" + strD + "_1.txt", "w")
        generalLogFile2 = open(logDir + "log_" + strD + "_2.txt", "w")

        #The 5X5 cross-validation is run for each dataset specified

        for dataSet in dataSets:
                ind = dataSet.rfind(".")
                dataSet = dataSet[0:ind]
                CompareLearnersC(srcDir + dataSet + ".arff", learners, noiseLevels, strD,
                    generalLogFile, generalLogFile2, classVal, repeats, folds)

        generalLogFile.close()
        generalLogFile2.close()



def CompareLearners(dataSetName, learners, strD, generalLogFile, generalLogFile2,
    classValue, repeats, folds):
        #This function tests all the specified learners on the given data set.
        #dataSetName - the name of the data set
        #learners - a list containing the definition of all the learners to be tested
        #strD - date and time when the experiment has started to execute (serves as id for
            the current experiment)
        #generalLogFile, generalLogFile2 - the main files of the experiment (the script
            writes a summary of the experiemtal results for all the data sets)
```

```
        #classVal - in case of classification problem this variable contains the target
            class for which the performances values are printed
        #repeats - number of repeats for the cross-validation experiment
        #folds - number of folds for the cross-validation experiment

        (procFileName, logFileName) = DefineFileNames(dataSetName,strD)
        logFile = open("logsDefect/" + logFileName, 'w')
        dataSet = ReadDataSet(dataSetName)
        #The dataset is read from the input filfolder and stored in object dataSet
        performances = InitializePerformances(learners, noiseLevels)
        #The object will contain the performance results collected on each experimental run
        for i in range(repeats):
                crossValDataSets = dataSet.split(folds)
                #The order of instances is randomized and the dataset is split by the
                    number of folds
                InitializePerformanceForRepeatStep(performances, i)
                for j in range(folds):
                        print str(i) + "␣" + str(j)
                        (trainDs,testDs) = GetTrainAndTestSets(crossValDataSets, fold=j)
                        RunLearners(learners, trainDs, testDs, performances, run=i, fold=j)
                        #Each Learner is trained and tested and the corresponding results
                            are stored in performances object
                        PrintPerformancesForExperimentalRun(logFile, performances, trainDs,
                            testDs, classValue, run=i, fold=j)
                        #The performances results for the current experimental run is
                            written in the specific log file

        winTieLoss = ComparePerformances(dataSet.isClassNumeric, performances)
        #The Win,Tie,Loss values are calculated for performance measures by applying
            Wilcoxon statistical tests
        PrintComparisons(dataSet.isClassNumeric, logFile, winTieLoss, classValue)
        #The Win,Tie,Loss values are written in the specific log file for the dataset.
        PrintSummary(dataSet.isClassNumeric, generalLogFile, generalLogFile2, procFileName,
            winTieLoss)
        #A summary of results is written on the general log file(s)
        logFile.close()

def ReadDataSet(dataSetName):
        #Reads the instances from .arff files and loads the values into dataset
        #dataSetName - Name of input file

        dataSet = readArff(dataSetName)
        #The arff file is read and instances are loaded in the dataSet
```

```
        dataSet.computeClassStatistics()
        #class statistics are computed for performance evaluation
        return dataSet

def GetTrainAndTestSets(crossValDataSets, fold):
        #The function forms train and test dataset from stratified splits of the original
            data

        crossValDataSets1 = crossValDataSets[:]
        testDs = crossValDataSets1[fold]
        crossValDataSets1.remove(testDs)
        trainDs = mergeDataSets(crossValDataSets1)
        return (trainDs, testDs)

def RunLearners(learners, trainDs, testDs, performances, run, fold):
        #The function executes experimental runs for each learner
        for learner in learners:
                learnerName = GetLearnerName(learner, noiseLevel, noiseLevels)
                RunLearner(learnerName, testDs, performances, run, fold)       #train in
                    noisy data

def RunLearner(learner, trainDs, testDs, performances, repeat, fold):
        #The function is used to run the experiment  for a learner

        (learnerName, params) = ParseLearner(learner)
        #Input String consisting of Learner name and params is parsed.

        #Parameter values are set
        stopingCriteria = 'C' #Specifies the criteria for stopping iteration for PEEKING2
        fssPercentage = 0.25  #Specifies the percentage of features which are to selected
            by InfoGain
        theta = 0.5           #Specifies the threshold for defect classification
        k = 2                 #Specifies the number of k value to used by K-NN algorthim
        normalize = 1         #Specifies whether the features have to be normalized (1 =
            normalize)
        numK=0                #Default value for number of clusters (In case of DBSCAN,
            represents epsilon value)

        predictions = None
        dataSet = None
        runtime = None

        if learnerName == "RF":
```

```
                (predictions, dataSet, trainTime, testTime, allRunTime) =
                    RandomForest(trainDs, testDs)
        elif learnerName == "PEEKING2":
                (predictions, dataSet, trainTime, testTime, allRunTime) = PEEKING2(trainDs,
                    testDs, stopingCriteria, k, fssPercentage, theta, normalize)
        elif learnerName == "LR":
                (predictions, dataSet, trainTime, testTime, allRunTime) =
                    LinearRegression(trainDs, testDs)
        elif learnerName == "M5P":
                (predictions, dataSet, trainTime, testTime, allRunTime) = M5P(trainDs,
                    testDs)
        else:
                (predictions, dataSet, trainTime, testTime, allRunTime) =
                    Clustering(learnerName, trainDs, testDs, k, fssPercentage, theta,
                    normalize, numK=float(params[0]))


        EstimatePerformanceForExperimentalRun(performances, predictions, dataSet,
            trainTime, testTime, allRunTime, repeat, fold, learner)
        #The Performance measures are calculated for the experimental run


def ParseLearner(command):
        #The method parses the given command string finding the learner and the
            corresponding parameters

        command = command.strip()
        learnerName = ""
        parameters = []
        ind1 = command.find("(")
        ind2 = command.find(")")
        if ind1 == -1 or ind2 == -1:
                learnerName = command
        else:
                learnerName = command[0:ind1]
                parameters = command[ind1+1:ind2].split(',')
                for i in range(len(parameters)): parameters[i] = parameters[i].strip()
        return (learnerName,parameters)


def Clustering(learner, trainDs, testDs, k, fssPercentage, theta, normalize,numK):
        #The method is used to train and test the models using reduced data
```

```
        t1 = datetime.now()
        condensed = Clustering_Train(learner, trainDs, stopingCriteria, fssPercentage,
            normalize,numK)
        t2 = datetime.now()
        predictions = Clustering_Test(testDs, condensed, k, theta, normalize)
        t3 = datetime.now()
        trainTime=t2-t1
        testTime=t3-t2
        allRunTime=t3-t1
        return (predictions, condensed, trainTime, testTime, allRunTime)

def Clustering_Train(learner, trainDs1, fssPercentage, normalize,numK):
        #The method is used to reduce data.

        fssTrainDs = FeatureSelectionViaInfoGain(fssPercentage, trainDs1)
        #Feature Selection is performed using InfoGain (25%)
        normTrainDs = NormalizeTrainDataSet(normalize, fssTrainDs)
        #The training dataset is then normalized
        numInstances = normTrainDs.NrInstances()
        #The number of instances in the dataset
        numAttributes = normTrainDs.NrAttributes()-1
        #The number of dependent attributes in the dataset

        #The dataset is transformed to an arrary format which can be input to the various
            clustering methods
        clusterData = np.ndarray(shape=(numInstances,numAttributes), dtype=float)
        for i in range(numInstances):
                for j in range(numAttributes):
                        clusterData[i][j] =
                            float(normTrainDs.instances[i].attributeValues[j])

        if learner == "DBScan":
                labels = DBScanClustering(clusterData,numK)
                numK = -1
                for i in range(len(labels)):
                        labels[i]= labels[i]+1
                        if(labels[i]>numK):
                                numK = labels[i]
                numK = int(numK) + 1

        elif learner == "K-Means":
                labels = KmeansClustering(clusterData,numK)
        elif learner == "E-Max":
```

```
            labels = EMClustering(clusterData,numK)
        elif learner == "MB_K-Means":
            labels = miniBatchKmeansClustering(clusterData,numK)
        elif learner == "Ward"
            labels = WardClustering(clusterData,numK)
        else:
            raise Exception("RunLearner - unknown learner: " + str(learner) + "!")

        clusters = FormClusters(normTrainDs, labels,int(numK))
        #The clusters are formed to based on the labels obtained
        condensed = GetClusterCentroids(clusters)
        #The dataset is condensed by reducing each cluster to one instance
        return condensed

def Clustering_Test(testDs1, condensed, k, theta, normalize):
        #The function predicts class value for instances in the test data
        fssTestDs = ApplyFeatureSelectionOnTestData(testDs1, condensed.attributes)
        #Only features selected by InfoGain in training data is retained
        testDs = NormalizeTestDataSet(normalize, fssTestDs, condensed.attributes)

        #Predictions are calculated for each instance
        predictions = []
        for instance in testDs.instances:
            predictions.append((instance.classVal,
                condensed.estimateClassFromNearestNeighbors_Exponential(instance, k,
                theta)))
        return predictions


def FeatureSelectionViaInfoGain(fssPercentage, trainDs):
        #The function runs InfoGain and selects the percentage of instance given as
            parameter
        dataSet = DiscretizeNumericClassByMedian(trainDs)
        #The numeric class value is discretized into two classes by using median value
        if fssPercentage != None:
            discTrainDs = dataSet.FayyadIraniDiscretizer("class")
            #Fayyad-Irani algorithm is used to discretize numerical dependant features
            selectedFeatures = discTrainDs.featureReductionViaInfoGain("class",
                fssPercentage)
            #Ranks features based on InfoGain and returns the selected features
            return trainDs.applyFSS(selectedFeatures)
            #The number of features in training dataset is reduced
        else:
```

```
                return trainDs

def ApplyFeatureSelectionOnTestData(testDs, attributes):
        #The function reduces the test dataset by retaining only features selected by
            InfoGain applied on training data
        if len(attributes) != len(testDs.attributes):
                selectedFeatures = getColumnIndexForAttributes(attributes)
                return testDs.applyFSS(selectedFeatures)
        else:
                return testDs

def NormalizeTrainDataSet(normalize, trainDs):
        #The function noramlizes the values of the features in train dataset

        if normalize == 0:
                return trainDs
        elif normalize == 1:
                return trainDs.NormalizeDataSet(outlierRemoval=True)  #Removes outliers
                    after normalization of data
        elif normalize == 2:
                return trainDs.NormalizeDataSet(outlierRemoval=False)
        else:
                raise Exception("Unknown normalize parameter: " + str(normalize))

def NormalizeTestDataSet(normalize, testDs, attributes):
        #The function noramlizes the values of the features in test dataset

        if normalize == 0:
                return testDs
        elif normalize == 1:
                return testDs.ApplyNormalization(attributes, outlierRemoval=True) #Removes
                    outliers after normalization of data
        elif normalize == 2:
                return testDs.ApplyNormalization(attributes, outlierRemoval=False)
        else:
                raise Exception("Unknown normalize parameter: " + str(normalize))


def DBScanClustering(clusterData, eps1):
        #Runs DBSCAN implementation in SciKit-Learn (sklearn.cluster.DBSCAN)

        db = DBSCAN().fit(clusterData, eps=eps1, min_samples=3)
        labels = db.labels_
```

```
        return labels


def KmeansClustering(clusterData,numK):
        #Runs K-Means implementation in Scipy (scipy.cluster.vq)

        centriods,variance = vq.kmeans(clusterData,numK)
        labels,distance = vq.vq(clusterData,centriods)
        return labels


def miniBatchKmeansClustering(clusterData,numK):
        #Runs Mini Batch K-Means implementation in SciKit-Learn
            (sklearn.cluster.MiniBatchKMeans)

        miniBatchKMeans = MiniBatchKMeans(numK).fit(clusterData)
        labels = miniBatchKMeans.labels_
        return labels


def WardClustering(clusterData,numK):
        #Runs Ward implementation in SciKit-Learn (sklearn.cluster.Ward)

        wd = Ward(n_clusters=numK).fit(clusterData)
        labels = wd.labels_
        return labels

def EMClustering(clusterData,numK)
        #Runs Mini Expectation Maximization implementation in SciKit-Learn
            (sklearn.mixture.GMM)

        em = GMM(n_components=numK).fit(clusterData)
        labels = em.predict(clusterData)
        return labels

def RandomForest(trainDs1, testDs1, discretize=True):
        #Runs Random Forest Learner from WEKA toolkit

        t = datetime.now()
        trainDs = trainDs1
        testDs = testDs1
        if discretize == True:
                trainDs = trainDs1.FayyadIraniDiscretizer("class")
```

```
            testDs = testDs1.discretizeDataSet(trainDs.attributes)
        learner = "weka.classifiers.trees.RandomForest"
        parameters = "-I 10 -K 0 -S 1"
        predictions = WekaLearner(trainDs, testDs, learner, parameters)
        overallRunTime = datetime.now() - t
        return (predictions, trainDs, None, None, overallRunTime)

def LinearRegression(trainDs, testDs):
        #Runs Linear Regression Learner from WEKA toolkit

        t = datetime.now()
        learner = "weka.classifiers.functions.LinearRegression"
        parameters = " -S 0 -R 1.0E-8 -i"
        trainDs.computeClassStatistics()
        predictions = WekaLearner(trainDs, testDs, learner, parameters)
        overallRunTime = datetime.now() - t
        return (predictions, trainDs, None, None, overallRunTime)

def M5P(trainDs, testDs):
        #Runs M5P Learner from WEKA toolkit

        t = datetime.now()
        learner = "weka.classifiers.trees.M5P"
        parameters = " -M 4.0 -i"
        trainDs.computeClassStatistics()
        predictions = WekaLearner(trainDs, testDs, learner, parameters)
        overallRunTime = datetime.now() - t
        return (predictions, trainDs, None, None, overallRunTime)

def WekaLearner(trainDs, testDs, learner, parameters):
        #The function is used to call WEKA functions using command line options
        #Weka functions require train and test data in .arff format

        jar = "/usr/share/java/weka.jar "      #Path of weka jar
        weka = "java -Xmx2048M -cp " + jar
        exportDataSetToArff(trainDs, "train.arff") #Creates a .arff file from the training
            dataset
        exportDataSetToArff(testDs, "test.arff")   #Creates a .arff file from the test
            dataset

        cmdLinux = weka + learner + " -p 0 " + parameters + " -t train.arff -T test.arff> 
            results.txt"
        #The call string for Linux
```

```
cmd = "java " + learner + " -p 0 " + parameters + " -t train.arff -T test.arff > "
    results.txt"
#The call string to used in Windows environment

os.system(cmdLinux) #Execute the command


#The output of the call execution is formatted and saved in "predictions"
i = 0
predictions = []

f = open("results.txt", "r")
for line in f:
        i += 1
        if i > 5:
                tokens = split(line)
                if len(tokens) < 4: continue
                actual = tokens[1]
                predicted = tokens[2]
                if trainDs.isClassNumeric == True: #regression
                        predictions.append((float(actual), float(predicted)))
                else: #classification
                        ind1 = string.rfind(actual, ":")
                        ind2 = string.rfind(predicted, ":")
                        if ind1 >= 0 and ind1 < len(actual)-1:
                                actual = actual[ind1+1:]
                        if ind2 >= 0 and ind2 < len(predicted)-1:
                                predicted = predicted[ind1+1:]
                        predictions.append((actual, predicted))
f.close()
return predictions
```

A.2 Evaluation.py

This file contains code which is used to calculate the performance measure of each learner and also compare the performances of different learners.

Evaluation.py

```
def EstimatePerformanceForExperimentalRun(performances, predictions, dataSet, trainTime,
    testTime, overallRunTime, repeat, fold, learner):
        #The function is used to calculate performance measures for each run

        if trainTime != None: trainTime = float(trainTime.microseconds)/float(1000)
        if testTime != None: testTime = float(testTime.microseconds)/float(1000)
        if overallRunTime != None: overallRunTime =
            float(overallRunTime.microseconds)/float(1000)

        if dataSet.isClassNumeric:
                meanError = UpdateErrors(performances, learner, predictions)
                meanMRE = UpdateMREs(performances, learner, predictions)
                scoreRange = dataSet.maxScore - dataSet.minScore
                performances[learner][repeat][fold] = {"mean_error":meanError, "mmre":
                    meanMRE, "instances":len(dataSet.instances),
                "columns":len(dataSet.attributes), "scoreRange": scoreRange,
                    "trainRunTime": trainTime, "testRunTime": testTime, "overallRunTime":
                    overallRunTime}
        else:
                confusionMatrix =
                    EstimateConfusionMatrix(dataSet.attributes[len(dataSet.attributes)-1],
                    predictions)
                performance = EvaluateClassification(confusionMatrix)
                expectedEntropy = None
                if hasattr(dataSet, 'expectedEntropy'):
                        expectedEntropy = dataSet.expectedEntropy
                performances[learner][repeat][fold] = {"class_measures":performance,
                    "instances":len(dataSet.instances),
                "columns":len(dataSet.attributes), "entropy": expectedEntropy,
                    "trainRunTime": trainTime, "testRunTime": testTime, "overallRunTime":
                    overallRunTime}

def UpdateErrors(performances, learner, predictions):
        #Calculates Absolute Residual Error
        sumError = 0
        for pred in predictions:
                actual = float(pred[0])
                predicted = float(pred[1])
                error = math.fabs(actual - predicted)
                sumError += error
                performances[learner]["all_errors"].append(error)
        return float(sumError)/float(len(predictions))
```

```
def UpdateMREs(performances, learner, predictions):
        #Calculates Magnitude of Relative  Error
        sumMRE = 0
        for pred in predictions:
                actual = float(pred[0])
                predicted = float(pred[1])
                if actual == 0: actual = 0.00000000001
                mre = math.fabs(actual - predicted)/actual
                sumMRE += mre
                performances[learner]["all_MRE"].append(mre)
        return float(sumMRE)/float(len(predictions))


def EstimateConfusionMatrix(classAttribute, predictions):
        #The funtion calculates confusion matrix for classification problem
        confusionMatrix = {}
        for classValue in classAttribute.attributeValues:
                a = 0 # true negatives
                b = 0 # false negatives
                c = 0 # false positives
                d = 0 # true positives
                for prediction in predictions:
                        if prediction[1] != classValue and prediction[0] != classValue:
                                a += 1
                        elif prediction[1] != classValue and prediction[0] == classValue:
                                b += 1
                        elif prediction[1] == classValue and prediction[0] != classValue:
                                c += 1
                        elif prediction[1] == classValue and prediction[0] == classValue:
                                d += 1
                confusionMatrix[classValue] = [a, b, c, d]
        return confusionMatrix

def EvaluateClassification(confusionMatrix):
        #The function calculates precision, recall, accuracy and F-Measure for
            classification problem
        performance = {}
        for classVal in confusionMatrix:
                pd = None
                pf = None
                prec = None
                acc = None
                fMeasure = None
```

```
                a = confusionMatrix[classVal][0]
                b = confusionMatrix[classVal][1]
                c = confusionMatrix[classVal][2]
                d = confusionMatrix[classVal][3]

                if (b+d) != 0: pd = float(d)/float(b+d)
                else: pd = float(0)
                if (a+c) != 0: pf = float(c)/float(a+c)
                else: pf = float(0)
                if (d+c) != 0: prec = float(d)/float(d+c)
                else: prec = float(0)
                if (a+b+c+d) != 0: acc = float(a+d)/float(a+b+c+d)
                else: acc = float(0)
                if prec != None and pd != None and (prec + pd) != 0: fMeasure =
                    2*(float(prec*pd)/float(prec+pd))
                else: fMeasure = float(0)
                #bal = 1 - ( ( ((0-pf)**2 + (1-pd)**2)**0.5 ) / (2**0.5) )
                if pd != None and pf != None and (pd + (1-pf)) != 0: g = float(2 * pd *
                    (1-pf))/float(pd + (1-pf))
                else: g = float(0)

                performance[classVal] = [pd, pf, prec, fMeasure, acc, g]
        return performance


def ComparePerformances(isClassNumeric, performances):
        #The function is used to compare performances of different learners for a dataset
        if isClassNumeric == True:
                return ComparePerformances_Regression(performances)
        else:
                return ComparePerformances_Classification(performances)

def ComparePerformances_Regression(performances):
        #The function is used to compare performances of different learners for a dataset
            when class is nurmeric
        for learner in performances:
                comparison_Instances = GetMedians(performances, "instances")
                comparison_Columns = GetMedians(performances, "columns")
                comparison_error = ComparePerformanceMeasure(performances, "all_errors",
                    None, None, 0.5, "lesser")
                comparison_mmre = ComparePerformanceMeasure(performances, "all_MRE", None,
                    None, 0.5, "lesser")
```

```
            #comparison_avgDist = GetMedians(performances, "avgDist")
            comparison_range = GetMedians(performances, "scoreRange")
        return {"mean_error":comparison_error, "mmre": comparison_mmre,
            "instances":comparison_Instances, "columns":comparison_Columns, "score_range":
            comparison_range}

def ComparePerformances_Classification(performances):
        #The function is used to compare performances of different learners for a dataset
            for two class problem

        for learner in performances:
            comparison_PD = ComparePerformanceMeasure(performances, "class_measures",
                "1", 0, 0.5, "greater")
            comparison_PF = ComparePerformanceMeasure(performances, "class_measures",
                "1", 1, 0.5, "lesser")
            comparison_Prec = ComparePerformanceMeasure(performances, "class_measures",
                "1", 2, 0.5, "greater")
            comparison_F = ComparePerformanceMeasure(performances, "class_measures",
                "1", 3, 0.5, "greater")
            comparison_Acc = ComparePerformanceMeasure(performances, "class_measures",
                "1", 4, 0.5, "greater")
            comparison_G = ComparePerformanceMeasure(performances, "class_measures",
                "1", 5, 0.5, "greater")
            comparison_Instances = GetMedians(performances, "instances")
            comparison_Columns = GetMedians(performances, "columns")
            comparison_entropy = ComparePerformanceMeasure(performances, "entropy",
                None, None, 0.5, "greater")
        return {"pd":comparison_PD, "pf":comparison_PF, "prec":comparison_Prec,
            "f":comparison_F, "acc":comparison_Acc,
                "g":comparison_G, "instances":comparison_Instances,
                    "columns":comparison_Columns, "entropy":comparison_entropy}

def ComparePerformanceMeasure(performances, key, classVal, measureIndex, alpha, criteria):
        #The function is used to rank perfomances of different learners for a given metric
            using wilcoxon statistical method
        #wilcoxon implementation is used from SciPy (scipy.stats.wilcoxon)
        learnerIndexes = []
        populations = []
        medians = []
        mins = []
        maxs = []
        q1s = []
        q3s = []
```

```
spreads = []
for learner in performances:
        population = GetMeasuresPopulation(performances[learner], classVal, key,
            measureIndex)
        #population = allPopulations[learner]
        if population[0] == None: continue
        learnerIndexes.append(learner)
        populations.append(population)
        quantiles = mquantiles(population, [0, 0.25, 0.5, 0.75, 1])
        mins.append(quantiles[0])
        q1s.append(quantiles[1])
        medians.append(quantiles[2])
        q3s.append(quantiles[3])
        maxs.append(quantiles[4])
        spreads.append(quantiles[3] - quantiles[1])


#initialize wintTieLoss matrix
winTieLoss = []
for i in range(len(populations)):
        winTieLoss1 = []
        for j in range(len(populations)):
                winTieLoss1.append(None)
        winTieLoss.append(winTieLoss1)

for i in range(len(populations)):
        for j in range(i+1, len(populations)):
                median_i = medians[i]
                median_j = medians[j]
                z_statistic, p_value = wilcoxon(populations[i], populations[j])
                if p_value > alpha or median_i == median_j:
                #the population are statistically the same
                        winTieLoss[i][j] = 0
                        winTieLoss[j][i] = 0
                else:
                        if criteria == "greater":
                                if median_i > median_j:
                                        winTieLoss[i][j] = 1
                                        winTieLoss[j][i] = -1
                                elif median_i < median_j:
                                        winTieLoss[i][j] = -1
                                        winTieLoss[j][i] = 1
                                else:
```

```
                                winTieLoss[i][j] = 0
                                winTieLoss[j][i] = 0
                        elif criteria == "lesser":
                                if median_i > median_j:
                                        winTieLoss[i][j] = -1
                                        winTieLoss[j][i] = 1
                                elif median_i < median_j:
                                        winTieLoss[i][j] = 1
                                        winTieLoss[j][i] = -1
                                else:
                                        winTieLoss[i][j] = 0
                                        winTieLoss[j][i] = 0
                        else:
                                raise Exception("ComparePerformanceMeasure - 
                                    Unknown criteria: " + criteria)

tie = {}
win = {}
loss = {}
medians1 = {}
mins1 = {}
maxs1 = {}
q1s1 = {}
q3s1 = {}
spreads1 = {}

for i in range(len(learnerIndexes)):
        learner = learnerIndexes[i]
        medians1[learner] = medians[i]
        mins1[learner] = mins[i]
        maxs1[learner] = maxs[i]
        q1s1[learner] = q1s[i]
        q3s1[learner] = q3s[i]
        spreads1[learner] = spreads[i]
        tie[learner] = 0
        win[learner] = 0
        loss[learner] = 0
        for j in range(len(learnerIndexes)):
                if i != j:
                        ret = winTieLoss[i][j]
                        if ret == 1: win[learner] += 1
                        elif ret == -1: loss[learner] += 1
                        else: tie[learner] += 1
```

```
        return [medians1, win, tie, loss, mins1, q1s1, q3s1, maxs1, spreads1]

def GetMeasuresPopulation(performance, classVal, key, measureIndex):
        population = []
        if key.startswith("all") == True:
                return performance[key]
        for i in performance:
                if isinstance(i, int) == False: continue
                for j in performance[i]:
                        if measureIndex != None:
                                population.append(performance[i][j][key][classVal][measureIndex])
                        else:
                                population.append(performance[i][j][key])
        return population

def GetMedians(performances, key):
        #The function calculates the median for a measure
        medians = {}
        for learner in performances:
                population = GetMeasuresPopulation(performances[learner], None, key, None)
                medians[learner] = np.median(population)
        return [medians]

def PrintWinTieLoss(logFile, comparison_results, measureName, classVal, criteria):
        #The function prints the Win Tie Loss Statistic for metric give in param
            measureName.
        medians = comparison_results[0]
        win = comparison_results[1]
        tie = comparison_results[2]
        loss = comparison_results[3]

        #sort by median
        if criteria == "greater":
                learners = sorted([(0-value,key) for (key,value) in medians.items()])
        elif criteria == "lesser":
                learners = sorted([(value,key) for (key,value) in medians.items()])
        else:
                raise Exception("PrintWinTieLoss - Unknown criteria: " + criteria)

        logFile.write("\n")
        logFile.write(measureName + " results" + "\n")
        if classVal == None:
```

```
        logFile.write("Learner,".ljust(30) + "Median, " + "Win, " + "Tie, " +
            "Loss" + "\n")
else:
        logFile.write("Learner,".ljust(30) + "class, " + "Median, " + "Win, " + "
            Tie, " + "Loss" + "\n")
for pair in learners:
        learner = pair[1]

        line = (learner + ", ").ljust(30)
        if classVal != None:
                line += classVal.rjust(5) + ", "
        line += ('%.2f' % medians[learner]).rjust(6) + ", "
        line += str(win[learner]).rjust(4) + ", "
        line += str(tie[learner]).rjust(4) + ", "
        line += str(loss[learner]).rjust(4)
        logFile.write(line + "\n")
```

A.3 Library.py

The data structures used to store the input data sets, instances and attributes were originally implemented by Vasil Papakroni as part of PEEKING2 [23]. The data structures enabled storing input data sets, clusters, condensed data sets and other intermediate results.