2013

Data Carving: Identifying and Removing Irrelevancies in the Data

Vasil Papakroni
West Virginia University

Data Carving: Identifying and Removing Irrelevancies in the Data

Vasil Papakroni

Thesis submitted to the
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

Timothy Menzies, Ph.D., Chair
Bojan Cukic, Ph.D.
Katerina Goseva-Popstojanova, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2013

Keywords: Feature Subset Selection, Instance Selection, Clustering, Classification, Regression

Abstract

Data Carving: Identifying and Removing Irrelevancies in the Data

Vasil Papakroni

Data mining has been successfully applied in the recent decades to automatically discover valuable hidden patterns from vast data. However, the current data mining approaches almost exclude business users from the process of knowledge discovery. Most of the learning algorithms operate as "black-boxes", which give predictions based on some input, without giving any hint on how they make their decisions. Moreover, their output is often difficult to interpret or understand by business users.

This thesis explores how to facilitate the users' engagement in the process of data analysis. We present PEEKING2, which combines a set of data reduction and data transformation techniques to create succinct summaries from raw data. In this way, users can "peek" at small representative data to reason over them. After removing uninformative features, PEEKING2 clusters the data combining FASTMAP projection and grid-clustering. A condensed summary of the data is then formed from the centroids of the resulted clusters. Finally, PEEKING2 extrapolates between centroids to predict the class of new instances.

PEEKING2 has been tested on Software Engineering data for software defect prediction and development effort estimation. Specifically, we have applied PEEKING2 on 10 defect data sets and 10 effort data sets from the PROMISE repository. PEEKING2 could reduce large data of 800+ rows and 20+ columns to just 10-30 rows and less than 6 columns.

To assess its predictive ability, we have compared PEEKING2 to more elaborate learners. Regarding defect prediction, PEEKING2 performed almost as well or better than Naive Bayes and Random Forest in most of the data sets. Similarly, when applied on effort estimation data, PEEKING2 outperformed or performed the same as Linear Regression and M5P in the majority of cases.

These results shows that *it is possible to "peek" at the data without losing significant information.* Consequently, we recommend PEEKING2 as a data summarization tool to assist managers and software engineers in their analysis of the project data.

Acknowledgments

First, I would like to thank my advisor Dr. Tim Menzies for his guidance in the wonderful research field of data mining. Next, I would like to thank Dr. Susan Partington for the financial support of my research studies via the USDA/National Institute of Food and Agriculture grant #2011-68001-30049.

Special thanks to Dr. Bojan Cukic for his valuable support throughout the two years of my graduate studies at WVU. I would also like to thank Dr. Katerina Goseva-Popstojanova and Dr. Arun Ross, whose course teachings provided me the foundations of my research.

I would like thank Fayola Peters for her contributions to my research and all my colleagues at the Modeling Intelligence (MILL) lab for their friendship and support.

I would like to acknowledge my beloved wife Erlina who first encouraged me to pursue my graduate studies at WVU. Next, I would like to give my thanks and love to my parents Aleksandra and Harallamb. I would also like to thank my sister Jorida and her fiance Tomi, whose support was crucial to my success. Finally, I give my thanks to all my family members and friends who have supported me throughout these years of my Master's studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advances of computer technology, such as the increase of computational power and storage capacity, we are overwhelmed by data. Data mining has been successfully applied in the recent decades to automatically discover valuable information from raw data. However, inferring information from increasingly larger and more complex data is becoming even more challenging.

To deal with this problem, the research community has been focused on designing "cleverer" learning algorithms. Surprisingly, we are often discovering that complex and supposedly clever learners do not produce significant more accurate results than simpler learners [10]. For example, a systematic literature review from Hall et al. on software defect prediction found that simpler learners such as Naive Bayes and Logistic Regression were superior to more elaborate learners such as Support Vectors Machine [19]. Similarly, a comparative study from Dejaeger et al. on effort estimation shows that simple linear regression performs better than more complex algorithms [9].

Furthermore, the current data mining practices almost exclude business users from the process of knowledge discovery. Most of the learners operate as "black-boxes" that give predictions based on some input, without giving any hint on how they reach their conclusions. Moreover, the results of the learners are often difficult to interpret or understand by business users. The transparency of a predictive model is a key factor to establish the confidence of business users in that model [8].

In addition, in domains such as Software Engineering (SE), it is very difficult to produce global predictive models that hold across the domain. This phenomenon, called *conclusion instability* is primarily caused by the large variability of data within Software Engineering. Software Engineering encompasses data from different development platforms, project sizes, and programmer capabilities. Several studies [31] [27] [26] [4], suggest local learning (i.e. building local models on regions of similar data) as a solution to conclusion instability in SE.

This thesis presents an algorithm, which simplifies raw training data by applying a combination of feature and row reductions. The algorithm, called PEEKING2, creates a condensed summary of the data that can be further reviewed and analyzed by business users. Optionally, an instance-based classifier is applied to infer predictions from the condensed data.

The major design goal of PEEKING2 is to engage users in the process of data analysis. The algorithm enables users to review and analyze raw data, by reducing large data sets to just a handful of rows and columns. Reasoning over the reduced data is very simple. Users can utilize their domain knowledge to infer important information from the condensed data. Simple spreadsheets tools could also assist in analyzing the data.

Furthermore, PEEKING2 is designed to support local learning. The algorithm does not produce a single general model that holds across the instance space. Instead, it identifies local regions of similar data and subsequently estimates their centroids. Predictions are then inferred extrapolating between centroids, which can be considered as "local models".

Specifically, PEEKING2 applies the following three steps to create a representative summary of the data:

- *Feature Selection via Information Gain.*

  The algorithm initially prunes uninformative features, by removing a specific percentage of features with the lowest values of information gain.

- *Projection via FASTMAP.*

  Next, the algorithm projects the instance space on the two directions of the greatest variability using the FASTMAP heuristic, which is a linear approximation of PCA.

- *Grid-Clustering.*

  Finally, the algorithm applies a simple grid clustering, which recursively partitions the projected space by the median value of each projected dimension (stopping when the number of instances in a partition drops below a specific value). The algorithm subsequently replaces each final partition with its corresponding centroid. A final condensed data set is formed from all these centroids.

The data reduction approach implemented by PEEKING2 is based on the argument that complex data can often be summarized into much simpler representations. As pointed out by Levina and Bickel in [25], high-dimensional data are in fact low-dimensional data embedded in a higher number of dimensions. *Intrinsic dimensionality* is used to evaluate the number of dimensions needed to represent a given data set. We use intrinsic dimensionality as a measure to indicate whether a set of data can be further simplified. When mining large and seemingly complex data, it is more important to find better representations of the data, rather than focusing our effort on designing and tuning complex learning methods.

PEEKING2 has been tested on Software Engineering data from the PROMISE repository. Specifically, the algorithm has been applied on 10 data sets for software defect prediction and 10 data sets for development effort estimation. When applied on these data sets, 800+ rows and 20+ columns were reduced to just 10-30 rows and less than 6 columns.

To investigate whether such large data reductions have caused significant information loss, we have compared PEEKING2 to more elaborate learners. For defect prediction, PEEKING2 was compared to Naive Bayes and Random Forest, while for effort estimation, it was compared to Linear Regression and M5P. In most of the cases (for both defect prediction and effort estimation),

PEEKING2 performed the same or even outperformed the other learners in terms of prediction accuracy.

These results suggest that we *do not* loose significant information from "peeking" at the data. Thus, business users can "rely" on the condensed summaries produced by PEEKING2 to infer comparatively accurate information. This thesis illustrates that "peeking" at SE data is not only harmless, but in fact can assist managers and software engineers in a variety of tasks such as targeting the testing process, refactor software modules, estimating and reducing the development effort.

1.1 Statement of Thesis

The research work of this thesis is summarized in the following statement:

> *This thesis shows that most of the signal in the Software Engineering data comes from small portions of the data. Consequently, using a combination of data reduction and projection techniques, we can effectively reduce the raw project data to simplify the data analysis and engage business users in the process of knowledge discovery.*

1.2 Contributions of the Thesis

The main contributions of this thesis are summarized as follows:

- Described the implementation of *PEEKING2* algorithm as a tool to simplify large and complex data to small data summaries.
- Investigated the combination of *feature selection* and *instance selection* as a method of data reduction. Although, there has been extensive research on both feature and instance selection separately, the combination of the techniques has not been thoroughly explored.

- Used *intrinsic dimensionality* as a measure to evaluate the simplicity of data.
- Demonstrated that "peeking" at the data is not harmful. Raw data can be largely reduced without losing significant information.
- Successfully applied PEEKING2 on SE data for both software defect prediction and development effort estimation.
- Shown how PEEKING2 can engage business users in the analysis of raw data to infer useful knowledge from them.

1.3 Paper

The following paper is based on the research work presented on this thesis:

V. Papakroni, T. Menzies, F.Peters, S. Partington, "Peeking at Data Considered Harmful?" to be submitted at: ASE 2013, the 28th IEEE/ACM International Conference on Automated Software Engineering, Nov 11th-15th, 2013.

1.4 Structure of the thesis

The rest of this thesis is organized as follows:

- *Chapter 2* initially gives a description of the first version of PEEKING algorithm presented by Borges and Menzies in [6] and how our approach is different. It subsequently surveys the previous work related to local learning, instance selection, and feature selection.
- *Chapter 3* gives a theoretical justification of the data reductions applied by PEEKING2. This chapter presents intrinsic-dimensionality as a measure to decide when to apply PEEKING2 on the data.

- *Chapter 4* describes the process of data summarization applied by PEEKING2.
- *Chapter 5* presents the inference methods implemented by the learner.
- *Chapter 6* reports the experiments conducted to assess the performance of PEEKING2.
- *Chapter 7* analyses the conditions under which the performance of PEEKING2 is not optimal.
- *Chapter 8* discusses the threats to validity of this thesis.
- Finally, *Chapter 9* summarizes the conclusions of this thesis and future work.

Chapter 2

Related Work

This chapter initially describes *PEEKING1*, which is the first version of PEEKING algorithm presented by Borges and Menzies in [6]. PEEKING1 combines several reduction and inference operators and it was tested on software project data for effort estimation. In this chapter, we also explain how our implementation of PEEKING differs from the original approach. Next, this chapter surveys the previous work in the research topics related to the implementation of PEEKING2. Specifically, we provide a survey of *local learning*, *instance selection*, and *feature selection*.

2.1 PEEKING1

The algorithm presented in this thesis is an adjustment of *PEEKING1* [1] presented by Borges and Menzies in [6]. PEEKING1 was designed to visualize software project data in order to assist project managers in their decision making process. PEEKING1 reduces the project data applying three data-reduction operators (feature projection, clustering, and feature selection). Next, it visualizes the reduced data by applying a visualization operator (rule reduction), which provides managers a set of recommendations on how to reduce the effort of a given project.

[1] PEEKING1 was originally called IDEA, which is short for Iterative Dichotomization on Every Attribute.

PEEKING1 has been tested on 20 effort estimation data sets from the PROMISE repository. The algorithm could reduce the data sets to few instances and features. In order to evaluate the quality of the reduced data, the Nearest Neighbor classifier approached on the reduced data was compared with 90 other learners (the combinations of 10 learners and 9 pre-processors) applied on the full data. The experimental results show that PEEKING1 performed almost as well as the other learners in terms of prediction accuracy. Consequently, the authors concluded that there is little information loss caused by the data reductions.

This thesis presents PEEKING2, which is an extension of PEEKING1 introduced by Borges and Menzies. We have adjusted PEEKING2 to support both software defect prediction and effort estimation. The following list gives the details of how our implementation differs from PEEKING1:

- *Support for both classification and regression problems.*

  PEEKING1 was originally implemented to handle regression problems, i.e. to predict continuous class values. In PEEKING2 we have added support for classification problems, such as software defect prediction.

- *Modification in the inference method.*

  PEEKING2 uses the k Nearest Neighbor learner to extrapolate between multiple centroids, instead of making conclusion based on only one centroid.

- *Intrinsic Dimensionality.*

  We present intrinsic dimensionality as a theoretical justification for the data reductions implemented by the algorithm. In fact, intrinsic dimensionality is used to decide whether to apply PEEKING2 on a given data set. The algorithm is applied only on the data with low intrinsic dimensionality.

2.2 Local Learning

Producing global models that hold across a particular domain is often very challenging due to the large variability of data. For example, the data collected in Software Engineering is tremendously large and diverse, encompassing different development paradigms, programming languages, project sizes, levels of developers expertise etc. Extensive research work has been conducted to build predictive models in SE. However, these models do often contradict each other [31]. The lack of consistent global models is referred to as *conclusion instability*.

Menzies and Shepperd in [31] survey the major sources of conclusion instability in SE. The authors recommend *local learning* as a solution to conclusion instability. They argue that "if a conclusion does not hold across all the data sets, then analysts might consider to find ... subsets of the data where different conclusions hold." [31] Local learning can be described as building predictive models on local regions of similar data. Clustering methods are typically used to identify such local regions.

Local learning in Software Engineering has been extensively examined by Menzies et al. in [27] and [26]. The authors present an approach that combines clustering with contrast-set rule learning. They compare global models with local models derived from clusters.

The instance space is initially projected in the directions of the greatest variability via the FASTMAP heuristic, which is a linear approximation of Participial Component Analysis. Next, the clustering algorithm, which is called WHERE, recursively partitions the instance space by the median values of the dimensions found by FASTMAP. Finally, WHERE merges neighboring partitions, until the density of the resulted cluster drops below a specific value. Figure 2.1 illustrates the mentioned steps followed by WHERE to cluster the NASA93 effort estimation data.

A *contrast-set learner* called WHICH is used to train local and global models from the data. Contrast-set learners generate rules from comparing the current context of data with target contexts, which are subsets of data with better class distributions (lower development efforts for effort

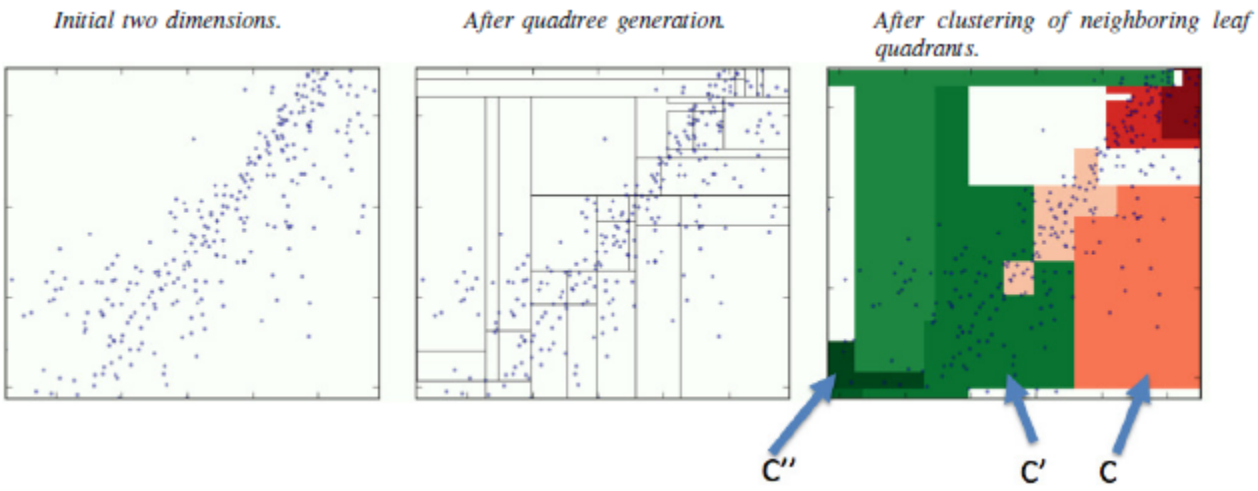

Figure 2.1: Clustering of NASA93 data set via WHERE. Taken from [27]. Each point represents a 24-dimensonal instance. The color of clusters is related to their corresponding median effort (red for clusters with the highest effort and green for clusters with the lowest effort).

estimation and lower number of defects for defect prediction). WHICH generates rules of the following format:

$$if\ R_x\ then\ (change = \varepsilon_1/\varepsilon_0 * support)$$

where R_x is a specific treatment of attributes, i.e. a combination of attribute values; ε_0 is the median class value in all the data (current context); ε_1 is the median class value in the data selected by the treatment (target context); *support* is the percentage of instances selected by the treatment; *change* represents the score assigned to the treatment. The score reflects how much is the median class value (effort or number of defects) reduced when the treatment R_x is applied.

WHICH searches the features space for attribute treatments with better class distributions. Initially, the algorithm creates a treatment for each attribute value. Then, it creates new treatments by combining pairs of existing treatments giving more priority to the ones with better scores. Finally, WHICH returns the rule with the best score after no improvement is observed.

Local rules are generated using the following strategy. For each cluster, WHICH is applied on the neighboring cluster with the best class distribution. The local rules generated on its neighbor

are then tested on the cluster. Let's take for example cluster C shown in Figure 2.1. C has a very large median intra-cluster effort and its neighboring cluster with the lowest median effort is C'. In this case, local rules are trained from C' and tested on C. *Contrast-set learning and the concept of learning from nearby clusters with desired properties are taken into consideration when designing PEEKING2 presented in this thesis.*

The experimental results described by Menzies at al. in [27] and [26] show of an advantage of local rules to global ones. Local rules could fit better in the data and produced better class distributions. [27] and [26] demonstrates that local models developed on subsets of similar data are superior to global models developed on large heterogonous data.

Similarly, Bettenburg et al. present a locality study on learning statistical regression models from effort and defect data [4]. The authors compare three learning strategies. First, they apply simple linear regression on the entire data. Second, the data is clustered via MCLUST and linear regression is applied on each cluster. Finally, the authors use MARS (Multivariate Adaption Regression Splines), which builds global regression models, taking into consideration local properties of the data. The experiments show that the approach combining global and local learning produced the most fit and accurate models. This study confirms the importance of locality on building predictive models in Software Engineering.

2.3 Instance Selection

Related work show that most of the signal in the training data is contained in a small subset of the data. Real-world data sets are "teemed" with irrelevant and redundant instances that could confuse learners. Removing such irrelevant and redundant parts of the data improves the accuracy of the learned models. This section reviews related studies on instance selection and pruning in SE.

Turhan et al. in [35] show that we need a small data sample to learn comparatively effective models for defect prediction. The authors analyze the effect of the sampling size on the accuracy

of defect prediction models learned by Naive Bayes. Experiments conducted on NASA project data sets (with size varying from 60 to 1,100) show that the learning performance does not further improve when the size of the training data sample is increased over 50 (once the data has been re-sampled such that the rate of defective vs. non-defective is 1:1 in the training data). Turhan et al.'s findings demonstrate that *some parts of project data are superfluous and that learners can achieve the same accuracy with less data*.

Other studies comment on the value of instance pruning. Kocaguneli et al. in [22] present an instance-based effort estimation learner called TEAK (short for Test Essential Assumption Knowledge). The authors argue that when designing a learner, it is very important to identify the key assumptions on which the algorithm is based and make sure that these assumptions are held on all the data. The parts of the training data that violate the learner's assumptions can confuse the learner and negatively impact the accuracy of the learned model. Consequently, the authors propose to remove these violations from the data prior to the learning process.

Instance-based effort estimation is based on the assumption that similar projects have similar efforts. Thus, we would expect that the variance of development effort in a cluster of similar projects to be relatively small. TEAK clusters the data via Greedy Agglomerative Clustering (GAC) that produces a tree of nested clusters called dendogram. Next, it traverses the tree to identify the cases when the class variance of cluster subtrees is greater than that of their parent. TEAK subsequently removes these clusters and learns from the rest of the data. Experimental results show that learning from clusters with lower class variance produced more accurate predictions than learning from the overall data.

Peters et al. in [32] presents an algorithm, called MORPH on privatizing project data without losing the ability to infer accurate predictive models from them. MORPH mutates original instances moving them within their respective class boundaries. In addition, an instance pruning operator called CLIFF is applied to remove irrelevant instances (i.e. outliers) from the data. CLIFF initially ranks each instance by the number of other instances that point to it as the nearest neigh-

bor. Subsequently, CLIFF removes the lowest ranked instances. Experiments show that pruning irrelevant instances before privatizing the data helps to build better predictive models.

2.4 Feature Selection

Related studies show that the presence of irrelevant, noisy and redundant features has a negative impact on the predictive performance of many learners [37] [18]. The degree of this impact varies among learners. Some learners, such as Naive Bayes are robust to irrelevant features, although they may suffer from redundant ones [37]. Other learners, such as instance-based algorithms are significantly vulnerable to irrelevant and noisy features [37]. *Feature Subset Selection* (FSS) algorithms analyze available features to determine which of them are appropriate for learning. FSS can work for classification problems, for regression problems or both.

FSS algorithms evaluate individual features or combination of features [18]. In the first case, FSS initially ranks each feature and subsequently selects the top-k ranked ones. However, determining the right number of features to be selected is often difficult. In the second case, FSS searches among different feature combinations and scores them via an evaluation function. When searching is completed, the feature combination with the highest score is returned.

There exist two main categories of FSS algorithms: *filters* and *wrappers* [18]. Filters evaluate feature combinations based on specific characteristics of the training data. Feature subsets are typically evaluated on how relevant they are to the class variable. As Figure 2.2 shows, filters are independent of the actual learning scheme that is going to be used. In fact, they pre-process the training data, before the learning algorithm is executed. Some of the popular filters are: Correlation-Based Feature Selection (CFS), RELIEF, Consistency-Based Evaluation (CBS), INFO GAIN, and PCA.

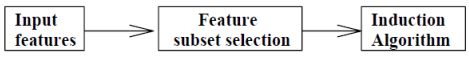

Figure 2.2: Filters. Taken from [24].

John and Kohavi in [21] and [24] give a new point of view on evaluating feature combinations. According to them, the relevance of features should be assessed in relation to a specific learning scheme. Consequently, they propose a FSS technique that ”wraps” the target learner inside the evaluation procedure. Figure 2.3 illustrates a typical wrapper.

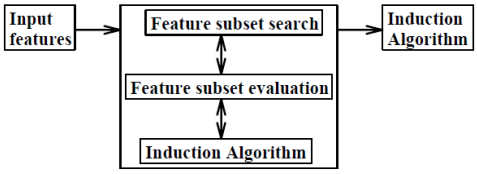

Figure 2.3: Wrappers. Taken from [24]

Wrappers use specific search heuristics to explore different feature subsets. A given feature subset is evaluated by applying the actual learning scheme on the corresponding features. The subset is then scored according the accuracy of the learned model. Wrappers are more computationally expensive than filters, because an actual learner is run for each feature subset that is evaluated. When applied on large data set, they do not even converge in a reasonably amount of time.

FSS algorithms must exhaustively search and evaluate all possible feature subsets to find optimal solutions. However, exhaustive search is practically not possible for large number of features. For this reason, FSS must rely on specific specific search methods that can converge to suboptimal solutions [14]. Some of the popular search methods used by FSS algorithms are: Sequential Forward Selection, Sequential Backward Selection, Best First, and Genetic algorithms.

Chapter 3

Intrinsic Dimensionality

As mentioned earlier, PEEKING2 reduces large data to few rows and columns. But how can we justify this large degree of data reduction? Levina and Bickel point out that complex data can often be synthesized into much simpler representations. They argue that highly-dimensional data "can be efficiently summarized in a space of much lower dimensions" [25]. For example, Figure 3.1 shows some data in a space of two dimensions. Although, the data appears to be two-dimensional, they can be better synthesized in only one dimension (the direction of maximum variability).

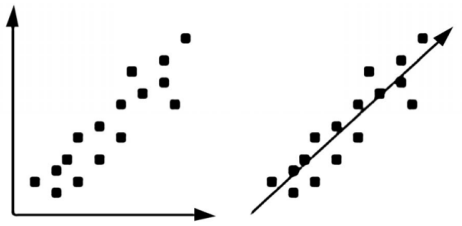

Figure 3.1: Example of feature synthesization.

The *intrinsic dimension* of a data set is the number of dimensions needed to represent the data [3]. Intrinsic dimensionality has been used in this study as a measure for "data simplicity". Low intrinsic dimensionality suggests that the data is superfluous and can be further simplified. Consequently, PEEKING2 is applied *only* on those data sets where the intrinsic dimension is considerable

smaller than the raw number of dimensions.

Levina and Bickel in [25], present their method on how to estimate intrinsic dimensionality. They make use of the *correlation dimension* or $C(r)$, which is a measure to indicate low dimensionality. $C(r)$ counts the pairs of instances with a distance lower than r and is then normalized by the total number of comparisons. We can evaluate the dimensionality of the data, comparing the differences in $C(r)$ when the distance r is gradually increased from r_1 to r_2.

The following example shows how the correlation coefficient can be used to evaluate intrinsic dimensionality. Let's suppose that some two-dimensional data are embedded in three dimensions. Next, we expand a "bubble" on all dimensions and continuously increase its radius by r. If the data were 3-dimensional, we would expect that r^3 more examples were included when the radius is increased by r. In fact, because the data is 2-dimensional, only r^2 more examples are included.

Figure 3.2 and Figure 3.3 shows the plots in logarithmic scale of $C(r)$ vs. r, for the defect and effort data sets used in this study. The intrinsic dimension of a data set is defined by the maximum slop of its corresponding plot [25]. In both figures, the data sets are listed in increasing order by their intrinsic dimensions. The left part shows the data sets with the lowest dimensionality, while the right part shows the rest.

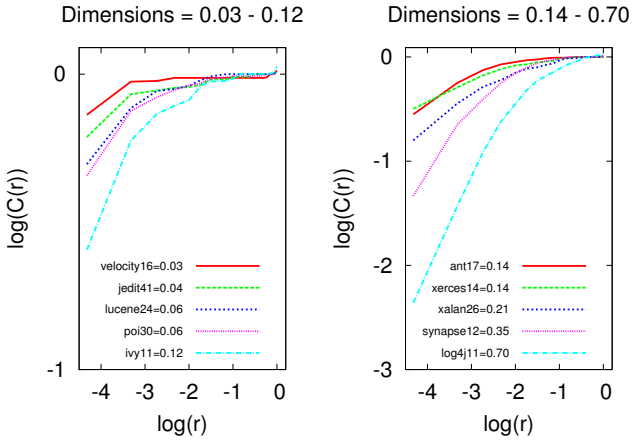

Figure 3.2: Intrinsic dimensionality for defect prediction data.

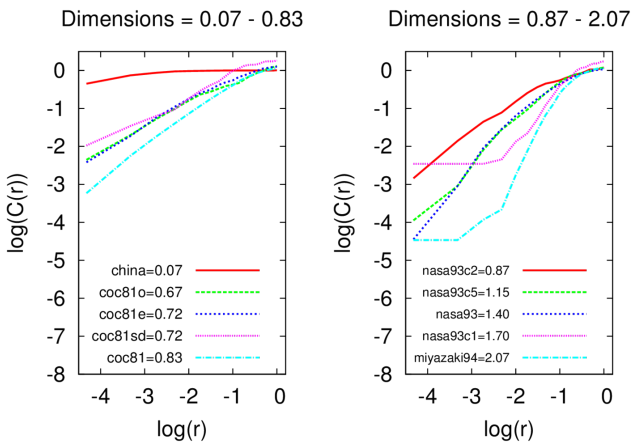

Figure 3.3: Intrinsic dimensionality for effort estimation data.

Note that the data sets considered in this thesis are characterized by a very low intrinsic dimensionality, which is far smaller than their raw numbers of dimensions. In fact, all defect data sets have intrinsic dimensions of smaller than 1, while their raw number of features is 20. Similarly, most of the effort data sets have an intrinsic dimensionality of smaller than 1, while their number of features mostly varies from 16 to 20.

The intrinsic dimensions reported in Figure 3.2 and Figure 3.3 suggest that our SE data sets are superfluous. Consequently, we think that data summarization tools such as PEEKING2 can further simplify these data into much condensed summaries.

Chapter 4

PEEKING2 - Data Summarization

This chapter describes the data summarization process implemented by PEEKING2 algorithm. As previously mentioned, PEEKING2 is designed to simplify superfluous data applying a series of data reduction and projection techniques. Specifically, the steps executed by PEEKING2 to summarize a given data set are the following:

- *Feature Selection via Information Gain.*

  The algorithm initially applies a simple feature subset selection method, which selects the attributes with the highest information gain in respect to the class.

- *Projection via FASTMAP.*

  Next, the algorithm projects the instance space on the directions of the greatest variability found by FASTMAP, which is a linear time approximation of PCA [33].

- *Grid-Clustering.*

  Finally, PEEKING2 applies a grid-clustering technique that recursively splits the instance space by the median values of the projected dimensions, until the number of instances in a cluster drops below a user-specified value.

A new condensed data set is then created containing the centroids of the final clusters. The projected features are *only* used during clustering. On the other hand, the final centroids are estimated using the original features before projection. We have followed this approach to make the condensed data more readable to business users.

Table 4.1 shows an example of a condensed data set produced by PEEKING2. Specifically, this figure shows the centroids that result from applying PEEKING2 on POI-3.0 data set. Note that the original 442 rows and 20 columns are reduced respectively to 21 rows and 5 columns.

Reasoning over such condensed data summaries as the one shown in Table 4.1 is very simple. In fact, business users can manually review the metrics data shown in the table. Optionally, PEEKING2 applies instance-based learning on the condensed data to make predictions or generate rules. These inference methods are explained in details in Chapter 5.

Note that the implementation of PEEKING2 is almost the same for software defect prediction that is a classification problem and effort estimation that is a regression problem. Nevertheless, we have pointed out some of the minor differences in the implementation of each case.

centroid	lcom3	ce	rfc	cbm	loc	defect rate
0	2	0.75	1.94	0	4.25	0.22
1	0.97	3.19	14.2	1.06	66.5	0.42
2	0.83	3.91	19.46	3.49	102	0.86
3	1.31	2	8	0.51	30	0.29
4	1.74	1.38	6.88	0.50	12.6	0
5	1.03	3.43	18.3	1.14	77.1	0.71
6	0.60	2.71	14.7	0.86	101	0.43
7	1.19	1	4.50	0	32.4	0.50
8	0.82	5.24	25	2.76	171	0.76
9	0.80	4.71	34	3.46	282	1
10	1.26	3.08	14.54	0.38	48.5	0.15
11	0.85	1.69	8.62	1	80.4	0.23
12	0.93	7.75	33.2	1.75	185	0.83
13	0.80	3.75	27.8	2.17	271	0.75
14	0.80	6.90	47.9	2.90	458	0.90
15	0.78	4.33	22.2	3.13	115	0.92
16	0.75	8.39	48.7	2.39	261	0.78
17	0.84	2.71	17.9	0.79	674	0.46
18	0.79	3.72	34.5	2.17	662	0.83
19	0.79	16.6	82	1.67	508	0.78
20	0.82	5.33	54.9	2.22	722	1
21	0.82	20.5	122	4.13	1324	0.87

Table 4.1: PEEKING2's summary of POI-3.0. The original data set is reduced from 442 rows and 21 columns to 21 rows and 6 columns. For details on the feature names see Table 6.1

The rest of this chapter further explains the operators applied by PEEKING2.

4.1 Feature Selection via InfoGain

The choice of features is crucial to the success of any data mining project [10]. Inappropriate features may have a negative impact in the predictive performance of many learners [17] [37]. Instance-based learning (used by PEEKING2 as inference method) is particularly sensitive to irrelevant features. For example, the k Nearest Neighbor classifier classifies a test instance based on its k closest training examples. If all features have the same weight in estimating the distances between examples, then irrelevant and relevant features will contribute the same to the classification.

PEEKING2 handles irrelevant features applying a simple feature selection method, which ranks all features based on their relevance to the class variable. The relevance of features is assessed by *Information Gain*, which is widely used in many algorithms such as decision tree learners. The Information Gain of a given feature is defined as the *entropy reduction* caused by splitting the training instances by the values of that feature. The formal definition of entropy and Information Gain are given respectively by the following equations:

$$H(D) = -\sum_{c \in Class} f_c \, log f_c \tag{4.1}$$

$$InfoGain(D,A) = H(D) - \sum_{v \in A} \frac{\mid D_{A=v} \mid}{\mid D \mid} H(D_{A=v}) \tag{4.2}$$

where D is the set of training instances, $D_{A=v}$ is the set of training instances with attribute A equal to v, f_c is the frequency of training instances with class equal to c, $Class$ is the class attribute, and A is a given independent attribute.

In order to compute equations 4.1 and 4.2, all independent attributes and the class variable must have discrete values. Continuous class values (for effort estimation) are discretized into two bins:

one bin for values lower than the median and the other bin for the rest. Additionally, numerical independent attributes are discretized via Fayyad-Irani discretizer. This algorithm recursively partitions the attribute values in such a way that maximizes the Information Gain of the attributes [13].

After all features have been ranked, PEEKING2 selects a specific percentage of features with the *highest Information Gain*. This percentage is arbitrarily specified by users. In our implementation, PEEKING2 selects 25% of the attributes. Table 4.2 illustrates an example of applying feature selection on POI-3.0 data set. For each feature, the table reports the entropy induced by the feature and the corresponding Information Gain.

Our FSS method differs from the original approach applied by Borges and Menzies in PEEKING1 [6]. In that study, attributes were ranked based on their Information Gain in respect to the

Table 4.2: Example of Feature Selection via Information Gain applied on POI-3.0 data set. Attributes are sorted in decreasing order by Information Gain. The top 25% of the attributes are selected (the ones shown in bold). Each attribute is discretized via Fayyad-Irani algorithm to compute its entropy and Information Gain. For more information about the attributes and the data set see respectively Table 6.1 and Table 6.2. The entropy of the entire data set is ≈ 0.95.

Attributes	Entropy	Information Gain
lcom3	0.68	0.27
ce	0.70	0.25
rfc	0.70	0.24
cbm	0.72	0.23
loc	0.74	0.21
max_cc	0.74	0.21
wmc	0.75	0.20
cbo	0.75	0.19
amc	0.76	0.19
dam	0.76	0.18
lcom	0.77	0.18
mfa	0.77	0.18
npm	0.80	0.15
avg_cc	0.80	0.15
cam	0.81	0.13
dit	0.89	0.06
moa	0.91	0.04
ca	0.93	0.01
ic	0.95	0.00
noc	0.95	0.00

generated clusters. This means that the features that contributed more in dividing instances into different clusters were more probable to be selected. Our preliminary results show a disadvantage of using the Information Gain in respect clusters. In this case, the attributes have roughly the same Information Gain, making it more difficult to distinguish between attributes. On contrary, Information Gain on class shows a clear distinction between attributes. This can be explained by the fact that the number of classes (which is two) is smaller than the number of created clusters (which varies from 10 to 30).

4.2 Feature projection via FastMap

FastMap is a heuristic feature projection algorithm, which similarly to Principal Component Analysis (PCA) projects the instance space in the directions of the greatest variability [12]. The algorithm produces approximately the same results as PCA, but much faster. In fact, FastMap executes in linear time, in contrast to PCA that executes in quadratic time. In addition, several studies show its effective application in empirical Software Engineering [27] [26].

Figure 4.1, describes how FastMap projects some given data on two new dimensions. The

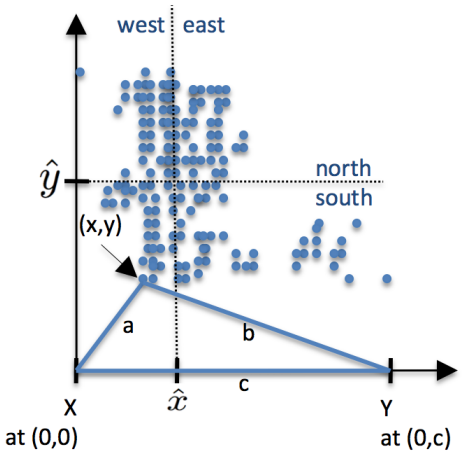

Figure 4.1: Projection of the instance space via FastMap. Taken from [27]

algorithm initially finds the direction of the greatest variability using a very simple approach:

- Pick randomly an instance Z.
- Find the furthest away instance X from Z.
- Find the furthest away instance Y from X.

As shown in Figure 4.1, an orthogonal dimension to $\overline{XY}$ can be found by declaring that the line $\overline{XY}$ is of length c and runs from point $(0,0)$ to $(0,c)$. Each instance now has a distance a to the origin (instance X) and distance b to the most remote point (instance Y). From the Pythagoras theorem and cosine rule, each instance is at the point $x = (a^2 + c^2 - b^2)/(2c)$ and $y = \sqrt{a^2 - x^2}$.

Figure 4.2-A illustrates the application of FASTMAP on POI-3.0 data set. The top picture shows the FASTMAP projection of the original 20-dimensional data. Whereas the bottom picture shows the projection of the 5-dimensional space defined by feature selection via Information Gain. Each point represents a specific software module. Defective modules are denoted by red, while non-defective modules are denoted by blue. As you may notice, the projected spaces do not differ very much from one another. In fact, defective and non-defective modules are approximately distributed in the same fashion in both spaces.

4.3 Grid-Clustering

After feature projection, PEEKING2 applies a grid-clustering algorithm to partition the instance space into a set of disjoint quadrants. Grid-clustering is a hierarchical clustering algorithm that recursively splits large clusters (starting from entire instance space) into smaller ones until the size of clusters (i.e. #instances) falls below a specific value [20]. The end-result of hierarchical-clustering algorithm is a tree of nested clusters called a dendrogram.

In our implementation, grid-clustering recursively divides each cluster by the median x and y dimensions found by FASTMAP. The recursion stops when the cluster size is smaller than a

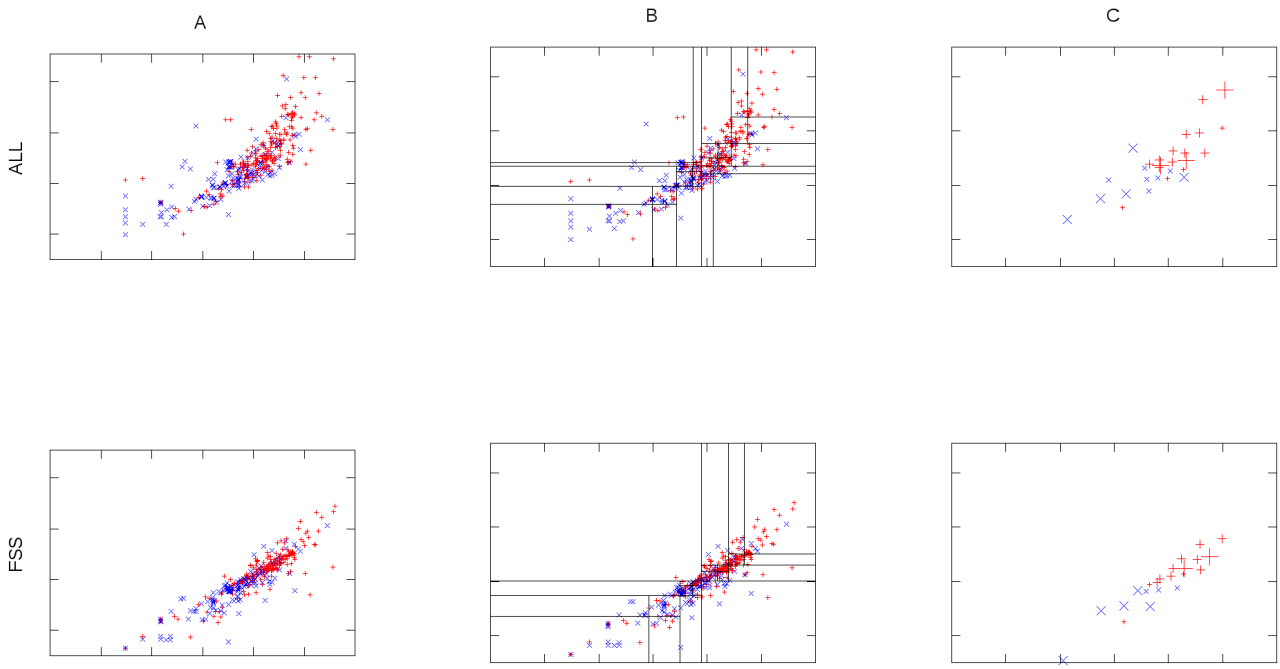

Figure 4.2: Example of FastMap projection, Grid-Clustering, and centroids estimation applied on POI-3.0 data set that contains 442 examples (for more on this data set see Table 6.1 and 6.2). **Top Row**: Data transformations applied on the entire set of attributes. Each point represents a 20-dimensonal instance. **Bottom Row**: Data transformations applied on 25% of the attributes with the highest Information Gain (refer to Table 4.2 to see which attributes are selected). Each point represents a 5-dimensonal 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 have been now condensed to a few dozen centroids.

specific value. Figure 4.2-B shows the partition of both projected spaces of POI-3.0 after the application of grid clustering.

The minimum allowed size of clusters enables us to control the level of data reduction applied on the data set. If the minimum cluster size is increased, then more data will be reduced and the final condensed data set will be smaller. When limited data are available, then PEEKING2 reduces less data in order to not loose much information. Thus, if the total number of instances (denoted with n) is greater than 100, then the minimum cluster size is set to $2 * \sqrt{n}$, otherwise it is set to $\sqrt{n}$.

Finally, PEEKING2 estimates the centroid of each leaf cluster from the resulted dendrogram. As mentioned earlier, centroids are estimated based on the raw attributes prior to FASTMAP projection. Each centroid is composed of the means of continuous attributes and the modes of discrete attributes estimated from all the instances contained in the corresponding cluster.

For defect prediction data, the class of centroids is changed from discrete ("defective" or "non-defective") to continuous. The new class (called $defect\_rate$) is defined as the rate of defective modules in each cluster. Thus, its values range from 0 to 1. In case of effort estimation data, the class of a centroid is simply defined by the mean development effort in the corresponding cluster.

Figure 4.2-C show the cluster centroids estimated for both projected spaces of POI-3.0 data. The point's color indicates whether the corresponding cluster has a majority of "defective" modules (red points) or a majority of "non-defective" modules (blue-points). In addition, the size of a point indicates the level of "purity" (entropy) in respect to the class of instances in that cluster. The smaller is the size of a point, the less "pure" is its corresponding cluster. Business user can quickly identify safe areas or faulty prone areas by the presence of "non-defective" and "defective" centroids and by their defect rates.

This chapter described the data summarization process implemented by PEEKING2. We illustrated how it is possible to condense software project data applying a combination of reduction techniques (feature selection, clustering, and centroid estimation). Next chapter describes the inference methods applied on the condensed data generated by PEEKING2.

Chapter 5

PEEKING2 - Inference

As the previous chapter demonstrated, PEEKING2 reduces large software data to a handful of centroids. In addition, PEEKING2 provides instance-based inference methods to assist users in analyzing the condensed data. Namely, the inference method implemented by PEEKING2 are:

- *k Nearest Neighbor Learner.*

  This method estimates the class of a test instance extrapolating between its k nearest centroids. In case of defect prediction, the method classifies a given software module as "defective" or "not-defective". In case of effort estimation, it estimates the development effort of a new software project.

- *Contrast-Set Rule Learner.*

  This method compares between centroids to generate rules on how to reduce the probability of defects in software modules or reduce the cost of development projects.

Instance-based learning is a simple and intuitive approach, which has been proved to be successful in many domains [37]. However, it is expensive in terms of memory and computational time. In addition, classifications may be confused by the presence of noise in the data [1]. The data reductions applied by PEEKING2 reduce the number of prototypes needed for instance-based

learning. In this way, PEEKING2 can significantly reduce its computational and memory needs. Furthermore, PEEKING2 handles the problem of local noise by averaging over all instances in each cluster.

The rest of this chapter describes in details the inference methods implemented by PEEKING2.

5.1 k Nearest Neighbor Learner

The first inference method is a modified version of the k Nearest Neighbor learner. In this implementation, the learner estimates the class of test examples based on their "similarity" to the cluster centroids. Given a test instance, the learner initially finds its k closest neighbors from the condensed data set. In order to do this, the learner estimates the Euclidean distance between the test instance and every centroid and then picks the k centroids with the smallest distance.

To optimize the process of distance estimation, the algorithm applies an approach called "*Euclidean partial sum of squares estimation*." According to this approach, the learner keeps track of the current k smallest sums of squares and stops computing the distance of a specific centroid, if the partial sum of squares exceeds the current k smallest values.

Next, the learner estimates the class of the test instance based on the *weighted average* of the class values of its k nearest centroids. The weights are specified in a way to give the nearest centroids more influence to the prediction. Specifically, the weight of each centroid is defined by the inverse of its distance to the test instance. Each weight is then normalized by the sum of all weights. Formally, the *average* of the class values is defined by the following equation:

$$avg\_class\_val = \sum_{i=1}^{k} \frac{1/d_i}{\sum_j 1/d_j} c_i \tag{5.1}$$

where d_i is the Euclidean distance of the i-th centroid from the test instance, and c_i is the class value of the i-th centroid.

Different values of k can be used to implement the k Nearest Neighbor learner. In this thesis, we

have analyzed only the k=2 Nearest Neighbor. We have made this choice, because our preliminary results have shown an advantage of this value in comparison to other values k.

Note that in case of defect prediction, the class of each centroid is defined by the rate of defective modules in the corresponding cluster. Given a software module, the kNN learner estimates the average defect rate of its k nearest centroids using equation 5.1. If the average defect rate is less than a user-specified threshold, the test instance will be classified as "non-defective", otherwise it will be classified as "defective". In our implementation, we have used a threshold of 0.5. Users can adjust this threshold according to their needs. For example, when testing safety-critical software, users may lower the threshold to consider even modules that are less probable to be defective.

On the other hand, in case of effort estimation, the class of the test instance is simply defined by the average development effort of its k nearest centroids.

5.2 Contrast-Set Rule Learner

The first inference method assesses the defectiveness of software modules or the development effort of projects. In addition, PEEKING2 can provide managers a set of recommendations on how to improve the quality of current modules or reduce the cost of their projects. These recommendations are generated comparing the current state (or context) of modules and projects with some desired or optimal states. To do this, PEEKING2 computes the differences between the nearest centroid of a given test example with its nearby centroids. Actual rules are then generated from these differences.

The approach of generating rules from comparing between different clusters (or contexts) of data is called *contrast-set rule learning* [27]. Our contrast-set method provides rules for reducing the development effort and the probability of defects. Borges and Menzies in [6] give detailed examples on how contrast-set learning can help reducing the cost of software development. This section is focused on the implementation of contrast-set learning for software code refactoring.

Code refactoring is recently becoming very popular in the software development industry. As defined in [16], software refactoring is changing "the internal structure of a software to make it easier to understand and cheaper to modify without changing its observable behavior". In other words, refactoring is the process of simplifying a software module without changing its external functionalities [2].

PEEKING2 can assist project managers and engineers to target the process of software refactoring by identifying the modules that need to be improved. Furthermore, PEEKING2 generates actual recommendations on how to refactor these modules. The method works as follows:

- *Identify the current context.* Given a software module, find its closest centroid, which we label as *"now"*.
- *Identify target context(s).* Find close centroid(s) with lower defect rate than the current context. These centroids are called *"envied"*.
- *Identify avoid context(s).* Find close centroid(s) with greater defect rate than the current context. These centroids are called *"feared"*.
- *Apply contrast-set operator.* Generate rules applying the contrast-set operator between the current context and the target/avoid contexts.

"what2do" and *"what2avoid"* rules are respectively generated by the following equations:

$$\begin{aligned} what2do &= envied - now \\ what2avoid &= feared - now \end{aligned} \tag{5.2}$$

Let us take an example that illustrates how these rules are generated. Suppose we want to improve the reliability of a software module from xalan-2.6 project. Figure 5.1 shows some of the centroids generated from xalan-2.6 data set. This figure displays the current context of the module

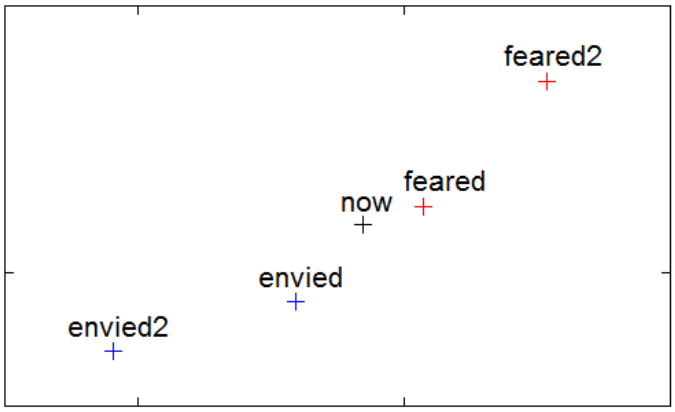

Figure 5.1: Example of the *current context* (“now”), *target contexts* (“envied”), and *avoid contexts* (“feared”) for a hypothetical new software module from xalan-2.6 project. The rules generated from these centroids are shown in Figure 5.2.

cluster/rule	rfc	lcom3	loc	cam	amc	defect_rate
a = now	1	6	1	4	1	0.39
b = envied	1	5	1	4	1	0.35
c = envied_2	1	9	1	5	1	0.18
d = feared	1	5	1	5	1	0.64
e = feared_2	2	4	1	4	1	0.76
what2do = b - a	0	-1	0	0	0	-0.04
what2do_2 = c - a	0	3	0	1	0	-0.21
what2avoid = d - a	0	-1	0	1	0	0.25
what2Avoid_2 = e - a	1	-2	0	0	0	0.37

Figure 5.2: The rules generated from the application of contrast-set operation on the centroids shown in Figure 5.1. Please notice that the attribute values **are not** raw values (except the class). They represent discrete values (ranging from 0 to 10) produced from 10 bins equal-width discretization of the attributes. For more information on the attributes see Table 6.1.

and some of the “envied” and “feared” contexts. PEEKING2 does not only offer an insight on the current location of the module in the instance space, but can also show *how to move* the module toward safer areas or divert it from dangerous ones.

Figure 5.2 demonstrates the application of the contrast set-operation between centroids. The first part of this figure shows the attribute values of the centroids, while the second part shows the resulted rules. The contrast-set operator applied between two centroids performs a simple subtraction between the attribute values of the centroids. The generated rule indicates how the

defect rate changes when the module's attributes are modified accordingly.

Please notice that the attribute values shown in Figure 5.2 **do not** represent raw values (except the class). In fact, they are discrete values (ranging from 0 to 10) produced from 10 bins equal-width discretization of the attributes. We have decided to use discretized values rather than raw values to make the generated rules more readable.

"What2do" and "what2avoid" rules provide hints on how to reduce the probability of defects in software modules. As you notice from Figure 5.2, managers can reflect on different alternatives provided by PEEKING2. They can choose to adopt a particular rule based on the degree of changes it requires to implement and the respective reduction of the defect rate.

For example, the first "what2do" rule recommends to slighty decrease *lcom* [1] of the module. However, the level of defect reduction is very small (only 0.04). On the other hand, the second "what2do" rule recommends more changes that result in a greater reduction of the defect rate (0.21). However, this rule requires a moderate increase of *lcom* and a slight increase of *cam* [2] metrics. Notice that "what2avoid" rules seems to agree with the second "what2do" rule on the direction of change of *lcom*, recommending to avoid decreasing this measure.

The rules produced by contrast-set learning are very small due to the closeness of centroids in the instance space. In addition, the contrast-set operator is very simple and can be applied manually via standard spreadsheet tools. *Therefore, by this stage of the process, we can eschew automatic algorithms and allow users to manually browse over the data*.

Nevertheless, business users should be careful on deciding to implement the changes recommended by PEEKING2. Although these rules show some correlation between specific attributes and the defect rate, *we are not claiming any causality*. PEEKING2 provides managers with alternative solutions to support them in their decision making. However, more rigorous experimental investigations must be conducted to confirm the validity of these recommendations.

[1] *lcom* is a software metric for the lack of cohesion.
[2] *cam* is a software metric of cohesion.

Chapter 6

Assessment

Chapter 4 presented the data summarization process implemented by PEEKING2, while Chapter 5 described the methods used to infer information from the resulted data summaries. In this chapter, we address the following concern:

How much should we "trust" the condensed data generated by PEEKING2?

As shown in the experimental results, PEEKING2 reduces large SE data to just a handful of centroids. Thus, we might expect that these reductions might actually throw away important data. To address this concern we have compared PEEKING2 with standard learners widely used in SE.

Furthermore, we have also evaluated the effect of feature selection on the predictive performance of the algorithm. For this reason, we have tested two versions of PEEKING2: PEEKING2(all), which uses the entire set of features and PEEKING2(infogain), which select 25% of features via Information Gain described in Section 4.1.

The rest of this chapter is organized as follows. Section 6.1 presents the defect and effort data sets used in our experiments. Section 6.2 describes the experiment that compares PEEKING2 with other standard learners in SE, while Section 6.3 presents the measures used to evaluate the learners. Section 6.4 and 6.5 respectively report the experimental results for the defect and effort data sets. Finally, Section 6.6 gives an empirical evaluation of the computational needs of PEEKING2.

6.1 Data Sets

PEEKING2 presented in this thesis has been applied on Software Engineering data from PROMISE repository. Specifically, we have tested PEEKING2 on 10 defect prediction data sets and 10 effort estimation data sets. All these data sets are publicly available at the Web site of the repository in [28].

Table 6.1: OO measures used in our defect data sets. The last line shows the dependent attribute (whether any defect is reported to a post-release bug-tracking system).

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	Binary class. Indicates whether any defect is found in a post-release bug-tracking system.

Table 6.2: Description of the Defect Data Sets. Note that each data set contains 20 features.

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

The defect prediction data sets contain metrics data collected from several software development projects. Specifically, the data sets provide for each software module the values of the 20 metrics described in Table 6.1. In addition, a binary class indicates whether the module has been found defective in any post-release testing [1].

Table 6.2 lists the defect prediction data sets selected in our experiments. If PROMISE contains several data sets for different releases of the same project, we have chosen *only* one data set. As a general rule, we have selected the data set corresponding to the last release. On the other hand, we have excluded the data sets in which Naive Bayes and Random Forest perform poorly[2]. We have excluded these data sets, because our preliminary results show that when applied on poor quality data, PEEKING2 deteriorates their quality. In fact, we do not present PEEKING2 as an algorithm to outperform the current state of the art learners. We want to show that PEEKING2 can considerably reduce the training data and achieve satisfying results that are comparable to other learners.

The effort estimation data sets contain details about several software development projects

[1] Note that the original defect prediction data sets contain a numeric class variable, which is defined as the number of defects detected for a module in any post-release testing.

[2] Both Naive Bayes and Random Forest achieve an F-measure lower than 0.5 for each class.

Table 6.3: COCOMO 81 effort multipliers. Taken from [29]

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

Table 6.4: Description of the Effort Data Sets.

			Actual Development Effort			
Dataset	**#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

and their actual costs. Table 6.4 shows the list of all effort data sets used in our experiments. Most of the data sets (cocomo81, nasa93 and their derived data sets) contain metrics defined by the COCOMO effort estimation model presented by Bohems in [5]. According to this model, the development effort of a project is exponential on LOC and linear on the 15 effort multipliers shown in Table 6.3. The other data sets (myazaki and china) collect other types of metrics. In most of the data sets, the actual development effort is measured in *man-months* (except china that measures the effort in man-hours). One man-month is equivalent of 160 working hours per person. More

information about the data sets and their attributes is available online in the PROMISE repository's Web site [28].

6.2 Description of the experiments

The experiments conducted in this study are designed to evaluate the ability of PEEKING2 to predict from the reduced data. We want to know whether accurate information can be actually inferred from the condensed summaries generated by PEEKING2. For this reason, we have compared PEEKING2 with other standard learners used in Software Engineering. The performance of PEEKING2 is evaluated via the *k=2 Nearest Neighbor* learner described in Section 5.1. Note that the kNN classifier is approached on the condensed data produced by PEEKING2, while the other learners are applied on the overall data.

For software defect prediction, PEEKING2 is compared with *Naive Bayes* (NB) and *Random Forest* (RF). We have chosen these learners for the following reasons. Related work shows that Naive Bayes is a widely used learner for defect prediction [19] [30]. To further improve its performance, NB is applied on data discretized via Fayyad-Irani algorithm [11]. Furthermore, we wanted to compare PEEKING2 with a sophisticated ensemble learner, such as Random Forest. Ensemble learners combine different predictive models to provide more accurate predictions. Ensemble learning has been successfully applied in many domains, including Software Engineering [23].

On effort estimation data, PEEKING2 is compared with *Linear Regression* and *M5P* regression tree learner. Several researchers suggest linear regression models to estimate the development cost of software projects [9]. In addition, we have included in our experiment M5P, which builds linear regression models on the leaves of a generated tree [34] [36]. Kocaguneli et al. in [23] recommend multimethods such as regression tree learners for effort estimation.

This study also evaluates the effect of feature selection on the performance of PEEKING2. For this reason, we have tested two versions of the algorithm. The first version does not apply any

feature selection on the data. On the other hand, the second version applies feature selection via Info. Gain described in Section 4.1. The percentage of selected features is 25%.

6.3 Evaluation Measures

All the learners assessed in this experiment are tested via a stratified *m=5 by k=5 cross-validation* scheme. The k-fold cross-validation is repeatedly executed m times. On each repeat, the order of instances is randomized to average the effect that the order of instances might have on the performance of some learners [15]. The total number of experimental runs is $5 * 5 = 25$. On each run, the learners are trained and tested on the same data sets.

Table 6.5: Confusion Matrix.

		Actual	
		non-defective	defective
Predicted	non-defective	TN	FN
	defective	FP	TP

The following gives the list of measures used to assess the accuracy of defect predictions inferred by Naive Bayes, Random Forest and PEEKING2. All these measures are based in the definition of the confusion matrix given in Table 6.5:

- $precision = \frac{TP}{TP+FP}$
- $recall(PD) = \frac{TP}{TP+FN}$
- $F = \frac{2*PD*prec}{PD+prec}$

A good defect predictor must precisely detect most of the defective modules (i.e. achieve high recall and precision). Assessing the classification performance using only one these measures can

be often misleading. For example, a defect predictor might classify as "defective" a large portion of modules that includes many "non-defective" ones (i.e. achieve high recall and low precision). Similarly, it might produce precise predictions, while missing a large portion of "defective" modules (i.e. achieve high precision and low recall). On the other hand, the F-measure takes into consideration both precision and recall by computing the harmonic mean of their values.

In case of effort estimation, the following measures are used to evaluate the effort estimates given by PEEKING2, Linear Regression, and M5P regression learner:

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

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

The Absolute Residual error measures the estimation error in actual effort units (most typically in man-months). On the other hand, Magnitude of Relative Error is commonly used to evaluate software effort estimation [7]. MRE measures the estimation error in proportion to the actual effort of the project. Consequently, the same absolute error is going to be penalized more for a project with a low effort than for a highly cost project.

Furthermore, learners are ranked according to the populations of measures observed in a 5-by-5 cross-validation experiment. Figure 6.1 shows the pseudo-code used to rank learners according

```
Compute Rank(learners, measure):
    order learners by median value of measure
    previous = null
    rank = 1
    for each learner in learners:
         if learner.IsStatisticallyDifferent(previous):
              rank = rank + 1
         end_if
         learner.rank = rank
    end_for
```

Figure 6.1: Ranking learners.

to a specific measure (like precision, PD, F, AR, and MRE). The learners are initially sorted by their median value of the measure. Each learner is then compared to the next one. A *Wilcoxon statistical test of 0.95 confidence* is then run to determine if their populations of measures are statistically different. If this is the case, the rank value of the current learner is increased by one. [3] Consequently, the rank of a learner enables us to evaluate whether its observed advantage in terms of a specific measure is statistically significant or not.

6.4 Results for defect prediction

Tables 6.6 and 6.7 report the amount of data reduction applied by PEEKING2 on the defect data sets. The first table compares the original number of rows in the training data with the median number of rows generated by PEEKING2 on a 5-by-5 cross-validation experiment. Similarly, the second table compares the number of cells in the training data with the median number of cells in the condensed data sets generated by PEEKING2.

Regarding row pruning, there is no significant difference in the degree of data reduction applied by both versions of PEEKING2. In terms of median results, both versions reduced the original number of rows by 93%. However, PEEKING2(infogain) discards a larger amount of data due to its feature selection operator. In fact, PEEKING2(infogain) has discarded at least 97% of the data for each data set. On the other hand, the median data reduction of PEEKING2(all) is 93%.

As to the classification performance, Table 6.8 reports the median precision, recall, and F-measure observed on 25 experimental runs (5 repeats x 5 folds) for each learner and data set [4]. In addition, this table reports the F-measure ranks computed by the pseudo-code shown in Figure 6.1. The results are grouped according to the performance of PEEKING2 in comparison to the other learners:

[3] Please note that the lower is the rank the better is the learner's performance.
[4] The classification measures are in respect to the "defect" class.

Table 6.6: Row pruning with or without InfoGain for defect data sets.

		no INFOGAIN		with INFOGAIN	
Data	**Examples**	**Centroids**	**%Reduction**	**Centroids**	**%Reduction**
ant-1.7	596	34	94	33	94
ivy-1.1	89	13	85	10	89
jedit-4.1	250	22	91	22	91
log4j-1.1	87	10	89	10	89
lucene-2.4	272	22	92	16	94
poi-3.0	354	22	94	25	93
synapse-1.2	205	22	89	16	92
velocity-1.6	183	10	95	19	90
xalan-2.6	708	37	95	34	95
xerces-1.4	470	25	95	25	95
			median = 93		median = 93

Table 6.7: Total data pruning (rows x columns) for defect data sets.

		no INFOGAIN		with INFOGAIN	
Data	**Cells**	**Size**	**%Reduction**	**Size**	**%Reduction**
ant-1.7	11,920	680	94	165	99
ivy-1.1	1,780	260	85	50	97
jedit-4.1	5,000	440	91	110	98
log4j-1.1	1,740	200	89	50	97
lucene-2.4	5,440	440	92	80	99
poi-3.0	7,080	440	94	125	98
synapse-1.2	4,100	440	89	80	98
velocity-1.6	3,660	200	95	95	97
xalan-2.6	14,160	740	95	170	99
xerces-1.4	9,400	500	95	125	99
			median = 93		median = 98

- *Group1* (4 data sets):

  Both versions of PEEKING2 perform as well or better as Naive Bayes and Random Forest.

- *Group2* (1 data sets):

  Only PEEKING2(infogain) performs almost as well as Naive Bayes and Random Forest.

- *Group3* (2 data sets):

  Only PEEKING2(all) performs close to Naive Bayes and Random Forest;

Table 6.8: Experimental Results for defect prediction.

Group 1 - Both versions of PEEKING2 perform well

ivy-1.1	*Learner*	*Prec*	*PD*	*F*	*Rank*
	PEEKING2(all)	0.75	0.83	0.78	1
	PEEKING2(infogain)	0.73	0.75	0.74	2
	NB	0.73	0.75	0.71	2
	RF	0.69	0.75	0.69	2
lucene-2.4	*Learner*	*Prec*	*PD*	*F*	*Rank*
	RF	0.73	0.76	0.74	1
	PEEKING2(all)	0.65	0.83	0.73	2
	NB	0.75	0.68	0.71	3
	PEEKING2(infogain)	0.66	0.75	0.70	4
poi-3.0	*Learner*	*Prec*	*PD*	*F*	*Rank*
	RF	0.84	0.86	0.85	1
	NB	0.87	0.82	0.84	1
	PEEKING2(all)	0.81	0.84	0.83	2
	PEEKING2(infogain)	0.82	0.82	0.83	2
ant-1.7	*Learner*	*Prec*	*PD*	*F*	*Rank*
	NB	0.47	0.65	0.55	1
	PEEKING2(all)	0.67	0.42	0.52	2
	PEEKING2(infogain)	0.69	0.42	0.51	2
	RF	0.61	0.45	0.50	3

Group 2 - Only PEEKING2 with infogain performs well

xerces-1.4	*Learner*	*Prec*	*PD*	*F*	*Rank*
	RF	0.94	0.95	0.95	1
	PEEKING2(infogain)	0.90	0.98	0.93	2
	PEEKING2(all)	0.82	0.93	0.87	3
	NB	0.94	0.80	0.86	4

Group 3 - Only PEEKING2 on all features performs well

xalan-2.6	*Learner*	*Prec*	*PD*	*F*	*Rank*
	NB	0.74	0.72	0.73	1
	RF	0.73	0.66	0.70	2
	PEEKING2(all)	0.71	0.62	0.68	3
	PEEKING2(infogain)	0.74	0.59	0.65	4
log4j-1.1	*Learner*	*Prec*	*PD*	*F*	*Rank*
	NB	0.67	0.62	0.67	1
	PEEKING2(all)	0.75	0.57	0.62	2
	RF	0.67	0.57	0.62	2
	PEEKING2(infogain)	0.71	0.50	0.56	3

Group 4 - Both versions of PEEKING2 do not perform well

synapse-1.2	*Learner*	*Prec*	*PD*	*F*	*Rank*
	NB	0.60	0.59	0.61	1
	RF	0.64	0.53	0.58	1
	PEEKING2(all)	0.56	0.50	0.55	2
	PEEKING2(infogain)	0.65	0.41	0.52	3
jedit-4.1	*Learner*	*Prec*	*PD*	*F*	*Rank*
	NB	0.61	0.62	0.63	1
	PEEKING2(all)	0.71	0.38	0.50	2
	RF	0.59	0.44	0.50	2
	PEEKING2(infogain)	0.67	0.31	0.38	3
velocity-1.6	*Learner*	*Prec*	*PD*	*F*	*Rank*
	NB	0.56	0.60	0.59	1
	RF	0.58	0.50	0.55	2
	PEEKING2(infogain)	0.54	0.33	0.40	3
	PEEKING2(all)	0.50	0.27	0.35	4

- *Group4* (3 data sets):

  Both versions of PEEKING2 perform worst.

We define that an algorithm performs as "well" as another, if its median F-measure is at most 0.05 lower than that of the other learner.

In the majority case, (see Group 1,2,3) at least one version of PEEKING2 performs as well as Naive Bayes and Random Forest. In one data set (ivy-1.1), the algorithm is clearly superior to the other learners. PEEKING2(all) performs well in 6 out of 10 data sets, while PEEKING2(infogain) performs very near the best learners in half of the data sets.

However, the experimental results show that occasionally PEEKING2 is not the optimal learner for certain data sets (e.g. Group4 results). But even then, in some cases (synapse-1.2 and jedit-4.1)

PEEKING2 performs almost as well or better than Random Forest that is considered a state-of-the art learner.

The experiment shows little difference in the classification performance of PEEKING2(all) and PEEKING2(infogain). In fact, feature selection via Information Gain shows no significant impact in the performance of the algorithm. The major challenge of our feature selection approach is specifying the right percentage of attributes to be selected. If very few attributes are selected, then important features may be discarded. Nevertheless, the FSS method improves visualization for the inferred rules described in Section 5.2, because the generated rules will contain fewer features.

6.5 Results for effort estimation.

Table 6.9 shows the level of row pruning applied by both versions of PEEKING2 on the effort estimation data sets. Similarly, Table 6.10 shows the amount of data reduction in terms of the number of cells (#rows * #columns). Both tables demonstrate that the amount of reduced data from the effort data sets is smaller to that of the defect data. Each effort data set (except china) contains less than 100 instances. For this reason, PEEKING2 applies a smaller level of data reduction to prevent large information loss.

Table 6.9: Row pruning with or without InfoGain for effort data sets.

		no INFOGAIN		with INFOGAIN	
Data	Examples	Centroids	% Reductions	Centroids	% Reductions
china	399	28	93	22	94
cocomo81	50	16	68	15	70
cocomo81e	19	11	42	10	47
cocomo81o	19	9	53	9	53
cocomo81s	9	7	22	6	33
miyazaki	38	16	58	15	61
nasa93	74	16	78	16	78
nasa93c1	10	5	50	5	50
nasa93c2	30	15	50	15	50
nasa93c5	31	12	61	9	71
			median = 55		median = 57

Table 6.10: Total data pruning (rows x columns) for effort data sets.

		no INFOGAIN		with INFOGAIN	
Data	Cells	Size	% Reductions	Size	% Reductions
china	6384	448	93	110	98
cocomo81	900	288	68	90	90
cocomo81e	342	198	42	60	82
cocomo81o	342	162	53	54	84
cocomo81s	162	126	22	36	78
miyazaki	304	128	58	45	85
nasa93	1554	336	78	96	94
nasa93c1	210	105	50	30	86
nasa93c2	630	315	50	90	86
nasa93c5	651	252	61	54	92
			median = 55		median = 86

Comparing the two version of PEEKING2, we notice that more data are reduced when feature selection is applied. In terms of median results, PEEKING2(infogain) reduced the number of original rows by 57%, while PEEKING2(all) by 55%. The amount of data reduction applied by the first version is considerably larger when measured in cells. Specifically, the median reduction applied by PEEKING2(infogain) is 86% of the overall training data.

Regarding the accuracy of effort estimations, Tables 6.11 and 6.12 give the estimation results of both versions of PEEKING2, Linear Regression, and M5P regression tree learner. The tables report the median, the inter-quartile range ($q3-q1$), and the spread ($max-min$) of the absolute residuals (AR) and the magnitude of relative errors (MRE) collected on a 5-by-5 cross-val. experiment. They also report the learners' ranks that are computed using the pseudo-code shown in Figure 6.1.

The experiment's results are grouped according to the performance of both versions of PEEKING2 compared to M5P and LR:

- *Group1* (5 data sets):

  PEEKING2 (both versions) is clearly superior to Linear Regression and M5P.

- *Group2* (2 data sets):

  PEEKING2(infogain) performs as well as M5P. Both versions perform better than LR.

- *Group3* (2 data sets):

  Both versions of PEEKING2 perform worse than M5P, but better than LR.

- *Group4* (1 data set):

  Both versions of PEEKING2 perform worst.

The results reported for effort estimation are very interesting. In this case, PEEKING2 is not only harmless, but over-performed both standard learners in half of the data sets. PEEKING2(infogain) performed as well or better than M5P regression tree learner in 7 out 10 data sets. Moreover, both versions outperformed Linear Regression in almost all the data sets. These results suggest that predicting on local data clusters is more advantageous than building a predictive model that hold across the space.

The only data set in which PEEKING2 performed poorly is china. Note that the intrinsic

Table 6.11: Experimental results for effort estimation - Part I

Group 1 - Both versions of PEEKING2 are superior to M5P and LR

Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
cocomo81	PEEKING2(infogain)	110.96	348.49	9869.54	0.78	2.25	16.22	1
	PEEKING2(all)	140.49	341.05	9493.92	0.79	2.49	36.26	1
	M5P	221.50	429.14	9362.12	1.45	4.16	110.82	3
	LR	601.90	761.96	9442.78	4.80	15.78	397.99	4
Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
cocomo81e	PEEKING2(infogain)	199.37	621.23	9464.67	0.62	0.57	13.31	1
	PEEKING2(all)	204.95	540.15	10756.14	0.61	0.76	30.54	1
	M5P	358.12	620.38	22595.06	1.07	3.04	92.73	3
	LR	1816.06	3301.47	13728.29	3.70	14.00	389.69	4
Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
nasa93c1	PEEKING2(all)	23.07	38.50	241.96	0.29	0.34	1.90	1
	PEEKING2(infogain)	25.19	60.94	186.26	0.32	0.37	1.38	1
	M5P	33.77	64.50	140.85	0.35	0.23	0.71	3
	LR	98.90	72.41	256.37	1.08	1.18	5.57	4
Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
cocomo81s	PEEKING2(infogain)	59.21	800.43	5909.04	0.80	0.92	10.01	1
	PEEKING2(all)	80.54	1271.75	5858.92	0.84	1.49	7.78	2
	LR	976.54	1054.83	8660.94	4.65	93.58	272.36	3
	M5P	1177.36	1723.74	6132.53	5.07	20.33	105.53	4
Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
miyazaki	PEEKING2(all)	20.70	29.42	1506.59	0.47	0.74	7.59	1
	PEEKING2(infogain)	18.07	33.33	1502.69	0.48	0.55	9.18	2
	LR	21.58	30.03	1106.55	0.56	0.63	8.55	2
	M5P	26.35	39.50	1315.90	0.66	1.07	16.10	4

Table 6.12: Experimental results for effort estimation - Part II

Group 2 - PEEKING2 with infogain performs as well as M5P, but better than LR

Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
nasa93c5	M5P	357.57	815.02	6655.73	0.65	1.26	10.14	1
	PEEKING2(infogain)	397.37	709.41	7455.13	0.69	1.18	20.66	1
	PEEKING2(all)	522.49	815.89	7572.51	0.84	1.71	13.76	3
	LR	678.39	505.81	7428.81	0.91	2.28	14.79	4
Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
nasa93c2	M5P	37.36	149.24	1033.92	0.41	1.15	51.91	1
	PEEKING2(infogain)	29.35	122.47	1199.23	0.42	0.66	5.93	2
	PEEKING2(all)	25.81	186.93	1198.22	0.48	0.83	7.20	3
	LR	175.32	108.53	1218.18	1.75	4.30	30.10	4

Group 3 - Both versions of PEEKING2 perform worse than M5P, but better than LR

Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
nasa93	M5P	127.11	306.92	6061.96	0.56	1.17	107.74	1
	PEEKING2(all)	162.92	465.57	7357.72	0.65	1.11	40.29	2
	PEEKING2(infogain)	195.41	516.85	7266.31	0.72	1.25	23.47	2
	LR	499.24	334.14	7738.81	1.45	7.93	84.39	4
Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
cocomo81o	M5P	22.73	34.53	180.65	0.53	1.29	6.86	1
	PEEKING2(all)	30.12	33.19	180.29	0.66	1.22	6.78	2
	PEEKING2(infogain)	31.34	35.43	190.80	0.67	1.05	7.73	3
	LR	43.04	53.34	397.07	0.89	1.44	12.97	4

Group 4 - Both versions of PEEKING2 perform worse than M5P and LR

Data	*Learner*	*Median AR*	*IQR AR*	*Spread AR*	*Median MRE*	*IQR MRE*	*Spread MRE*	*Rank*
china	M5P	199.07	447.51	40279.19	0.12	0.21	22.73	1
	LR	834.44	1677.51	52613.01	0.50	0.81	49.05	2
	PEEKING2(infogain)	1184.17	2116.81	43335.85	0.59	1.21	36.52	3
	PEEKING2(all)	1218.55	2138.81	43728.70	0.61	1.31	44.91	4

dimension of this data set is extremely low (0.07). We attribute these poor results of PEEKING2 on the lack of structure on this data set.

PEEKING2 also performs very well in terms of the error expansion. In most of the cases, the inter-quartile range and spread of ARs and MREs observed for PEEKING2 are almost the same as those observed for the other learners. In some data sets (see cocomo81, cocomo81e, cocomo81s and nasa93c2) the IQR and spread of the MRE values obtained from PEEKING2 are clearly the smallest. This means that PEEKING2 does not only produce more accurate predictions in terms of median results, but it also gives more stable predictions.

Analyzing the experimental results of both defect prediction and effort estimation, we can

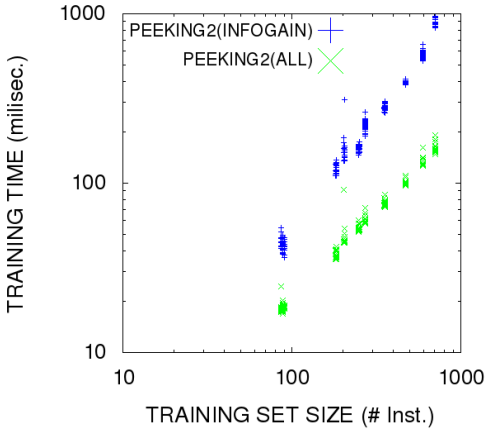

Figure 6.2: Training run-times in miliseconds of PEEKING2(infogain) implemented in PYTHON 2.7. The run-times are estimated on a 5-by-5 cross-validation experiment that is run on a standard 3GHz quad core machine with 4Gb of RAM (Windows7).

conclude that in general *there is little information loss caused by data reduction*. Therefore, we recommend PEEKING2 as an effective approach to simplify the analysis of project data. However, users should be cautious when using this approach, because occasionally these reductions may deteriorate the quality of the training data. Such cases are further analyzed in the next section.

6.6 Computation needs of PEEKING2

This section describes a simple empirical evaluation of the computational needs of PEEKING2. Figure 6.2 reports the training run-times in milliseconds of PEEKING2(all) and PEEKING2(infogain) in relation to the size of the training data for the 10 defect data sets. The plot is shown in logarithmic scale. The run-times are estimated on a 5-by-5 cross-validation experiment that is run on a standard 3GHz quad core machine with 4GB of RAM (Windows7 operating system). The algorithm is implemented in PYTHON 2.7.

The training run-times grow linearly as the size of the training data is increased. However, the training process is very fast even for the larger data sets that contain about 700 instances. In fact,

training is completed in less than one 1 second for all data sets. PEEKING(infogain) runs slower than PEEKING2(all), due to the feature selection method applied by the first. We attribute these fast run-times to FASTMAP that projects the instance space in linear time.

Chapter 7

When not to "peek" at the data?

Table 6.8 from the previous section shows that in some cases, the classification performance of PEEKING2 is worse than that the other learners. We think that this under-performance is directly related to the quality of the instance space, which results from the data transformations applied by the algorithm. *In fact, there seems to be a relation between the "purity" of clusters and the prediction accuracy of PEEKING2.*

PEEKING2 can give accurate predictions, if the instance space is composed of local regions containing instances of roughly the same class. On the other hand, important information is lost, when averaging the instances of "impure" clusters. Let us take an example of a cluster with a defect rate of 0.45. In this case, nearby test instances will be likely labeled as "non-defective", although the respective cluster contains a considerable proportion of defective modules.

To test our hypothesis, we have analyzed the relation between the expected entropy of clusters generated from the training data with the respective F-measure of PEEKING2. Figure 7.1 shows the plots of expected entropy vs. F-measure for PEEKING2(infogain) from a 5-by-5 cross-validation experiment. This figure confirms the correlation of clusters' entropy to the predictive ability of PEEKING2. In fact, the smaller is the entropy of clusters the more accurate are the learner's predictions.

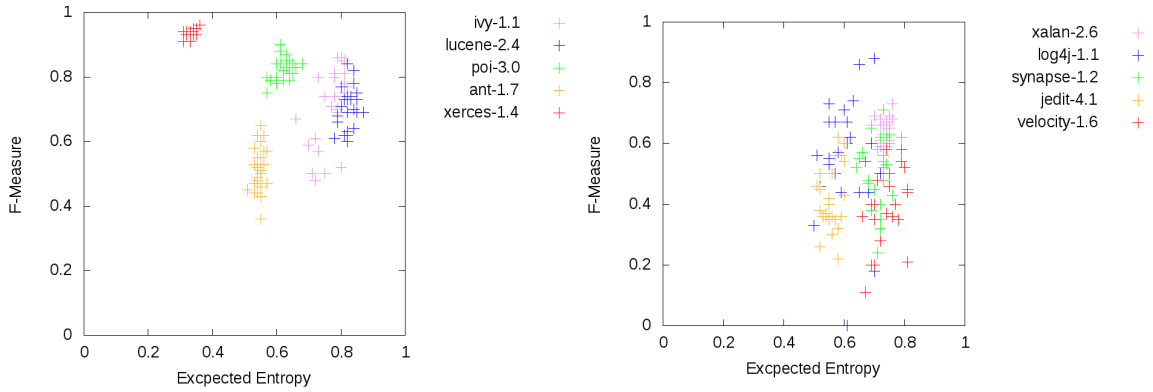

Figure 7.1: Plot of Expected Entropy vs. F-measure of PEEKING2(infogain). The left part shows the data sets in which the learner performs well, while the right part shows the rest. The F-measures are in respect to the "defect" class.

Therefore, we recommend a preliminary analysis of the data before using PEEKING2 as a tool for decision making. The entropy of clusters is directly related to the quality of data representation created by FASTMAP. In some cases, the synthesized space does not clearly distinguish between classes. Consequently, the entropy of generated classes is going to be very high, because instances of different classes are mixed all over the space. When encountering these conditions, business users should rely back on the more sophisticated defect predictors.

Chapter 8

Threats to Validity

Threats to validity are inevitable for empirical studies such as the one presented in this thesis. However, it is important to identify such threats and control them as much as possible. This chapter reports the threats of validity for the presented study. They are grouped into four categories: construct, internal, conclusion, and external validity.

8.1 Construct Validity

Construct validity is related whether the designed experiments are actually evaluating what we are supposed to evaluate. The primary goal of our experiments has been assessing the information loss incurred by the data summarization process of PEEKING2. To do this, we have compared the performance of PEEKING2 with other widely used learners for defect prediction and effort estimation.

We must note that the comparison between learners *does not quantify the amount of information loss* caused by PEEKING2. However, the difference in the performance metrics gives a roughly idea of the impact of the data reductions on the information contained in the data. If PEEKING2 was throwing away important details from the data, the performance of kNN on the reduced data

would have been very poor. On contrary, our results show that PEEKING2 could make relatively accurate predictions in comparison to the other learners applied on all data.

The choice of learners to be compared with PEEKING2 is also very important to the construct validity. A bad choice of learners could lead us to the wrong conclusions about the effectiveness of PEEKING2 and the quality of the condensed data. What makes this choice difficult is the vast variety of available learners. However, we have tried to choose some of the learners that have been successfully applied in Software Engineering for defect prediction and effort estimation.

8.2 Internal Validity

Internal validity is concerned with any factor that might influence the conclusions of the study, but cannot be controlled or assessed by the authors. Data mining studies particularly suffer from internal validity problems. In most of the cases, researchers do not have control or adequate information about the collection of the data. For this reason, business users should carefully examine the results generated from the training data (such as the rules generated by PEEKING2 for software refactoring in Section 5.2) before utilizing them.

8.3 Conclusion Validity

This section is concerned with the statistical validity of our conclusions. As previously mentioned, Wilcoxon statistical tests have been used to compare the populations of measures collected by 5-by-5 cross-validation experiments. Consequently, learners are not executed on distinct training data on each experimental run. For this reason the collected observations are not independent. This fact may have an impact on the validity of the results generated from the statistical test. We have applied cross-validation experiments due to limited amount of available training data.

8.4 External Validity

External validity is concerned with the generalization ability of the conclusions of the study. As previously mentioned Software Engineering is a large field that includes different software paradigms, programming languages, and project types. This large variability within SE limits the generalization of the conclusions derived by empirical studies such as this one.

For example the defect prediction data sets used in this study contain Object-Oriented metrics from different JAVA projects. Consequently, we cannot be sure whether the results observed on these data hold for different programming languages and software development paradigms.

Chapter 9

Conclusion and Future Work

In conclusion, this thesis presented PEEKING2 as a tool for data summarization that helps managers and engineers in analyzing the software project data. We have designed and implemented PEEKING2 with the primary focus of engaging business users in the process of knowledge discovery. In fact, users can now examine large amounts of project data condensed in just a few rows and columns. Furthermore, we illustrated how PEEKING2 can assist business users on a large variety of tasks such as predicting defective modules, estimating the development cost of projects, and refactoring software modules.

Our experiments demonstrated that little information is lost from the data reductions applied by PEEKING2. In fact, PEEKING2 could infer from the reduced data as accurate predictions as the other more sophisticated learners applied on overall data. In case of defect prediction, PEEKING2 performed as well as Naive Bayes and Random Forest in most of the cases. In many cases, PEEKING2 could outperform state-of-the-art learners such as Random Forests. When applied for effort estimation, PEEKING2 was clearly superior to Linear Regression and M5P that are widely used for development effort estimation. This thesis confirms again that most of the signal in the Software Engineering data is contained in a small subset of the data.

Nevertheless, in some data sets, the performance of PEEKING2 was not optimal. Our analysis

indicates that this under-performance is related to the quality of the instance space that results from projection. Thus, we recommend a preliminary analysis of the data, before deciding to apply PEEKING2.

The following is a list of future works that we want to further explore:

- *Improve the grid-clustering algorithm.*

  Having found a strong correlation between F-measure and expected entropy of clusters, we think that clusters should be formed in order to minimize the entropy within them. Consequently, a better approach would be to recursively partition the space in such a way that the expected entropy of clusters is minimal.

- *Handle outliers in FASTMAP.*

  Outliers may have a negative effect on the quality of the space synthesizes by FASTMAP. They can confuse the algorithm when trying to find the two most distant points in the instance space. The line drawn between these two instances is in fact the FASTMAP approximation of the direction of the greatest variability. Thus, it is important to find a way to identify and remove these outliers.

- *Apply PEEKING2 on other data domains.*

  This thesis presented the application of PEEKING2 on Software Engineering data. It is interesting to investigate how PEEKING would perform on other data domains. We are currently applying PEEKING2 in a public health study conducted by West Virginia University to assess the influence of local food outlets on local obesity. Using a combination of projection and clustering techniques, PEEKING2 could identify a small sample of representative outlets that can be used to evaluate the rest of outlets. In this way, we can reduce the cost of future revisits focusing the data collection effort on these representative outlets.

Bibliography

[1] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Mach. Learn.*, 6(1):37–66, January 1991.

[2] Kent Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1999.

[3] Robert S Bennett. Representation and analysis of signals part xxi. the intrinsic dimensionality of signal collections. Technical report, DTIC Document, 1965.

[4] 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.

[5] Barry W Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, (1):4–21, 1984.

[6] 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.

[7] Lionel C Briand, Khaled El Emam, Dagmar Surmann, Isabella Wieczorek, and Katrina D Maxwell. An assessment and comparison of common software cost estimation modeling techniques. In *Proceedings of the 21st international conference on Software engineering*, pages 313–322. ACM, 1999.

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

[9] Karel Dejaeger, Wouter Verbeke, David Martens, and Bart Baesens. Data mining techniques for software effort estimation: A comparative study. *IEEE Transactions on Software Engineering*, 38(2):375–397, 2012.

[10] Pedro Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, October 2012.

[11] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *Proc. 12th International Conference on Machine Learning*, pages 194–202. Morgan Kaufmann, 1995.

[12] Christos Faloutsos and King-Ip Lin. Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. *SIGMOD Rec.*, 24(2):163–174, May 1995.

[13] U M Fayyad and I H Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[14] FJ Ferri, P Pudil, Mohamad Hatef, and J Kittler. Comparative study of techniques for large-scale feature selection. *Machine Intelligence and Pattern Recognition*, 16:403–403, 1994.

[15] Douglas Fisher, Ling Xu, and Nazih Zard. Ordering effects in clustering. In *Proceedings of the ninth international workshop on Machine learning*, pages 163–168. Morgan Kaufmann Publishers Inc., 1992.

[16] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, August 1999.

[17] M.A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions On Knowledge And Data Engineering*, 15(6):1437– 1447, 2003.

[18] Mark A. Hall and Geoffrey Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans. on Knowl. and Data Eng.*, 15(6):1437–1447, November 2003.

[19] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, November 2012.

[20] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.

[21] George H John, Ron Kohavi, and Karl Pfleger. Irrelevant features and the subset selection problem. In *Proceedings of the eleventh international conference on machine learning*, volume 129, pages 121–129. San Francisco, 1994.

[22] Ekrem Kocaguneli, Tim Menzies, Ayse Bener, and Jacky W Keung. Exploiting the essential assumptions of analogy-based effort estimation. *Software Engineering, IEEE Transactions on*, 38(2):425–438, 2012.

[23] Ekrem Kocaguneli, Tim Menzies, and Jacky Keung. On the value of ensemble effort estimation. *IEEE Trans. Softw. Eng.*, 38(6):1403–1416, November 2012.

[24] Ron Kohavi and George H John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1):273–324, 1997.

[25] E. Levina and P. Bickel. Maximum likelihood estimation of instrinsic dimension. In *Advances in NIPS*, volume 17, 2005.

[26] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local vs. global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2012.

[27] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 343–351, Washington, DC, USA, 2011. IEEE Computer Society.

[28] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, June 2012.

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

[30] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, January 2007.

[31] Tim Menzies and Martin Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Softw. Engg.*, 17(1-2):1–17, February 2012.

[32] F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing privacy and utility in cross-company defect prediction. *Software Engineering, IEEE Transactions on*, PP(99):1, 2013.

[33] John C Platt. Fastmap, metricmap, and landmark mds are all nyström algorithms. In *Proc. 10th Int. Workshop on Artificial Intelligence and Statistics*, pages 261–268, 2005.

[34] Ross J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992. World Scientific.

[35] 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.

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

[37] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011.