IMPROVING PERFORMANCE OF DEFECT PREDICTORS USING CONFIRMATION BIAS METRICS

by

Handan Gül Çalıklı

B.S., Mechanical Engineering, Boğaziçi University, 2000

M.S., Computer Engineering, Boğaziçi University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2011

IMPROVING PERFORMANCE OF DEFECT PREDICTORS USING CONFIRMATION BIAS METRICS

APPROVED BY:

Prof. Ayşe Bener .....
(Thesis Co-supervisor)

Prof. Oğuz Tosun .....
(Thesis Co-supervisor)

Prof. H. Levent Akın .....

Prof. Fikret Gürgen .....

Prof. Ali Tekcan .....

Burak Turhan, Ph.D. .....

DATE OF APPROVAL: 21.12.2011

Dedicated to my parents Hilal and Ömer

Yücel Çalıklı

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Prof. Ayşe Bener for giving me the opportunity to do research in an interdisciplinary field such as human aspects in software engineering. She believed in me and let me find my own way in the jungle of cognitive psychology, statistics, computer science and software engineering. I thank Prof. Oğuz Tosun for being my co-advisor; and Prof. Levent Akın, Prof. Ali Tekcan, Prof. Fikret Gürgen and Dr. Burak Turhan for being in my thesis committee. My gratitude goes to Turgay Aytaç for his intellectual guidance and for being a role model. I also thank Turgay Aytaç, Ayhan İnal from Logo Business Solutions and Turkcell A.Ş. for providing data for my empirical research. There is one person whom I believe when she says what I have done as a research work is satisfactory enough, Ayşe Tosun. Many thanks to her for our academic discussions as well as for her friendship, advices and support. My gratitude also goes to my other friends from SoftLab, Bora Çağlayan, Gülfem Işıklar and Aslı Uyar Özkaya. From the department, I'd like to thank Reyhan Aydoğan and Arzucan Özgür for their support, Itır Karaç for her sincere friendship as well as being our leader in sailing and holiday adventures, Özgür Kafalı for the great tennis talk and critiques, Nadin Kökciyan for being a great tennis partner, Suzan Bayhan for the chats at department's lounge, Didem Gözüpek and Gaye Genç Kara for the great time we had at the conference in Lefkoşa during summer 2009 and many other friends whose names I could not mention, but who made each day at the department meaningful. The last but not the least, many thanks to a person of good character, Deniz Gayde, whose friendship and support means a lot to me. I cannot forget my brother Yüceer Çalıklı, who helped me to preprocess data for my thesis and encouraged me whenever I lost my faith.

Finally, I dedicate this thesis to my parents Ömer Yücel and Hilal Çalıklı. I love you really much and feel myself very lucky for being your daughter, although I sometimes curse my genes of perfectionism I inherited from you both. Thank you for believing in me. I hope I can always make you proud.

ABSTRACT

IMPROVING PERFORMANCE OF DEFECT PREDICTORS USING CONFIRMATION BIAS METRICS

Software defect prediction models help managers to prioritize their testing effort. Algorithms, which are used to learn defect predictors, have reached a ceiling such that further improvements may only come from increasing information content of input data. The main goal of this research is to build defect predictors which are learnt from developers' levels of *confirmation bias*, which is defined as tendency of people to seek for evidence to verify hypotheses rather than seeking for evidence to refute them. Our main goal is to overcome the ceiling effect in defect prediction performance. As a first step, we define a methodology to measure/quantify confirmation bias and then we perform the following empirical analysis: i) investigation of factors which have the potential to affect confirmation bias levels of software developers and testers, ii) empirical analysis of how confirmation bias affects software developer and tester performance, iii) a benchmark analysis comparing performance of defect predictors which use combinations of static code, confirmation bias and churn metrics, iv) defining a methodology to build defect predictors which can learn using incomplete confirmation bias metric values as input. Our results on industrial data show that: i) no effect of experience in software development/testing has been observed on confirmation bias, whereas hypotheses testing and reasoning skills affect confirmation bias, ii) confirmation biases of developers lead to an increase in defects, while testers' confirmation bias causes an increase post-release defects, iii) using *only* confirmation bias metrics, we can build defect predictors with higher or comparable prediction performance when compared to defect predictors that are learnt by using *only* churn metrics or *only* static code metrics, iv) promising results can also be obtained by using incomplete confirmation bias metric values to learn defect predictors.

ÖZET

DOĞRULAMA SAPMASI METRİKLERİ İLE YAZILIMDA HATA TAHMİNİ PERFORMANSININ İYİLEŞTİRİLMESİ

Hata tahmini modelleri, yöneticilere bir yazılımda test edilmesi gereken kısımların önceliklendirilmesinde yardımcı olur. Yazılımda hata tahmini için kullanılan algoritmalar performans sınırılarına ulaşmıştır, öyleki daha ileri gelişmeler ancak giriş verilerinin içeriğini arttırmakla olabilir. Bu tezin amacı, yazılımcıların doğrulama sapması seviyelerini kullanarak performans limitini aşan hata tahmini modelleri oluşturmaktır. Doğrulama sapması metriklerini elde etmek için bir metodoloji tanımladıktan sonra sırası ile şu aşamaları gerçekleştirdik: i) yazılımcı ve testçilerin doğrulama sapması seviyelerini etkilemesi olası faktörlerin incelenmesi, ii) doğrulama sapmasının yazılımcı ve testçi performansını nasıl etkilediğine dair deneysel analizlerin yapılması, iii) statik kod ve doğrulama sapması metrikleri ile kaynak kod dosyalarında yapılan değişikliklerden elde edilen metriklerin değişik kombinasyonlarda kullanılmasıyla oluşturulan hata tahmini modellerinin karşılaştırmalı performans analizinin yapılması, iv) eksik doğrulama sapması metrik değerleri ile hata tahmini modellerini oluşturma metodolojisinin tanımlanması. Endüstriyel verileri kullanarak şu sonuçları elde ettik: i) yazılım geliştirme ve test etme deneyiminin doğrulama sapması üzerinde etkisine rastlanmamıştır, öte yandan hipotez test etme ve muhakeme becerilerinin doğrulama sapması üzerinde etkileri gözlemlenmiştir, ii) yazılımcıların doğrulama sapması, yazılımda hata miktarı artışına neden olurken, testçilerin doğrulama sapması sürüm sonrası hata miktarının artmasına neden olmaktadır, iii) sadece doğrulama sapması metrikleri ile elde edilen hata tahmin performansının sadece statik kod veya kaynak kod dosyalarında yapılan değişikliklere ilişkin metriklerle elde edilen hata tahmini performansından daha iyi ya da bu değerlere yakın olduğu gözlemlenmiştir, iv) Eksik doğrulama sapması metrikleri ile kayda değer hata tahmini performansı elde edilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

AI	ArtificialIntelligence
bal	Balance
CGBR	Call Graph Based Ranking
DBMS	Database Management System
E-R	Entity-Relationship
ERP	Enterprize Resource Planning
FN	False Negatives
FP	False Positives
GCD	Greatest Common Divisor
ISV	Independent Software Vendor
IT	Information Technology
MSE	Mean Square Error
$NPROD_{DEF}$	Number of Production Defects
SME	Small Medium Enterprise
PCA	Principal Component Analysis
pd	Probability of Detection
pf	Probability of False Alarms
PSP	Personal Software Process
TDD	Test Driven Development
TN	True Negatives
TP	True Positives
TSP	Team Software Process
3P	Product Process People (3 pillars of software development)

1. INTRODUCTION

Software testing is the most resource consuming phase of software development life-cycle. Approximately 50% of a project schedule is allocated to testing phase [3]. Defect predictors provide guidance to project managers to effectively allocate resources in testing phase by pointing out defect-prone parts of the software. However, defect prediction algorithms have reached a performance ceiling. Reported results in software defect prediction literature suggest that further progress in defect prediction performance can be achieved by increasing the content of input data that defect predictors learn rather than using different algorithms or increasing the size of input data [4–6]

Among the three pillars (3Ps) of software development, which are *product*, *process* and *people*, product and process have long been taken into account in software defect prediction. Yet, modeling *people* related aspects in software defect prediction still remains as a challenge, since it requires conducting more interdisciplinary research and practice to combine mathematics, statistics, sociology, and psychology. We are aware of the fact that we cannot cover all human aspects. On the other hand, cognitive biases, which are deviations of human mind from the laws of mathematics and logic, are very likely to have significant effects on software development. In this thesis, we focus on a particular cognitive bias type called *confirmation bias* which is defined as the tendency of people to confirm their hypotheses rather than refuting them. During all levels of software testing, including unit testing, the attempt should be to fail the code [7–9]. However, due to confirmation bias especially developers might perform unit tests to make their program work. This results in the propagation of more defects to testing phase and hence probably an increase in software defect density. Similarly, defects which are overlooked by testers, during testing phase of a specific release of a software are likely to propagate to the next releases of that software.

1.1. Software Defect Prediction Models and the Ceiling Effect

1.1.1. Software Defect Prediction Models

In software defect prediction, various machine learning algorithms have been employed by researchers. Munson and Khoshgoftaar [10] construct discriminant models by using static code metrics as independent data, where multicolinearity among static code metrics is eliminated by Principle Component Analysis. Bullard et. al.[11] propose a rule-based classification model for prediction of defects in a large legacy Telecomunication system. In [12], Classification and Regression Trees (CART) algorithm is used to identify fault-prone modules in embedded systems. Neural networks is another machine learning technique used by Khosgoftaar and Szabo [13] to learn defect predictors. Regression models have also been widely used [14–17]. The model consisting of an ensemble of classifiers proposed by Tosun *et al.* [18] combines three algorithms which are Naïve Bayes, Neural Networks and Voting Feature Intervals respectively. In his repeatable set of experiments, Menzies *et al.* [5] discovered that Naïve Bayes classifier with a log-filtering preprocessor on the numeric data, outperforms methods such as OneR and J4.8. Results obtained by Menzies *et al.* are in line with the results of the benchmark study by Lessmann *et al.* [4]. In this benchmark study, Lessmann *et al.* also found no significant difference between performance of Naïve Bayes and more complex machine learning algorithms.

In order to find out whether performance of defect predictors can be increased by sampling methods due to the unbalanced nature of the defect data, Menzies *et al.* [6] performed a series of experiments. As algorithm, they used Naïve Bayes since it was useful in their previous experiments [5] as well as J4.8 which was used in prior under-over sampling experiments [19, 20]. According to the results obtained, throwing away data (i.e. undersampling) does not degrade the performance of the learner. For J4.8 algorithm throwing away data improved median performance from around 40% to 70%, while under-sampling outperformed over-sampling for both J4.8 and Naïve Bayes. These results are consistent with those of Drummond *et al.* [19] and Kamei *et al.*[20].

1.1.2. Ceiling Effect in Software Defect Prediction Performance

Research in software defect prediction has shown that content of input data that defect predictors learn is more essential than the algorithm used and size of the input data. In software defect prediction, various machine learning algorithms have been employed by researchers, ranging from regression models [14–16], decision trees [21], neural networks [22] and Naïve Bayes [5, 23, 24] as well as application of cascading classifiers [18]. However, Menzies *et al.* [5] discovered that Naïve classifier with a log-filtering preprocessor on the numeric data, outperforms methods such as OneR and J48. In their benchmarking study [4], Lessmann *et al.* also concluded that there is no significant difference between performance of Naïve and more complex machine learning algorithms. In summary, using different algorithms did not result in any significant improvement in the performance of defect predictors. In order to find out whether size of input data affects performance, Menzies et. al. [6] applied their "micro-sampling" method which takes only a small portion of data to learn defect predictors using Naïve algorithm. The results they obtained showed that size of the input data did not affect defect prediction performance.

On the other hand, enhancement of input data content mostly resulted in improved defect prediction performance. Nagappan *et al.* [25] state that code churn metrics are good predictors of post-release defects. Jiang *et al.* [26] compared predictor performances that were learnt from design metrics, static code features and both for 13 NASA projects. Design metrics were extracted from requirements documents with a text miner. More accurate results were obtained by using both design and static code metrics rather than individual use. The results obtained were consistent with results of the similar experiments which were previously conducted by Zhao. *et al.* [27] for the analysis of a real time Telecomunication system. Zimmerman and Nagappan [28] developed a metric suite which defines dependency of binary files from a graph theoretic point of view. The authors used these metrics as input to linear and logistic regression models to predict post-release failures of Windows Server 2003. Zimmerman and Nagappan report 10% increase in defect prediction performance due to the inclusion of dependency graphs as input data. Following this research, Nagappan and Ball [25], combined dependency and churn metrics to predict post-release faults in binary files of Windows Server 2003. The

authors conclude that they can predict post-release failure using regression models at a statistically significant level. Tosun *et al.* [29] used network and churn metrics as well as static code metrics in order to build defect predictors for different defect categories. According to their results, churn metrics gave the best result to predict all types of defects. Moreover, Tosun *et al.* [24] obtained improved performance results by using churn metrics in addition to static code metrics as input data to learn defect predictors for a telecommunication infrastructure software. Turhan *et al.* in another study [30] reduced probability of false alarms by supplementing static code metrics by their Call Graph Based Ranking (CGBR).

1.2. Human Factors in Defect Prediction Models

Since software products are developed by *people*, information content of data input to defect prediction models needs to be enhanced by people metrics. Although it is not a trivial task to model *people*, as shown in Figure 1.1, the first step should be to list human aspects which are most likely to cause introduction of defects in the software product being developed. Every phase in a software development life cycle requires analytical problem solving skills. Unlike a person working in an assembly line in a factory, a software developer and tester requires more sophisticated problem solving skills. Moreover, using everyday life heuristics instead of laws of logic and mathematics may affect quality of the software product in an undesirable manner.

Cognitive biases are deviations of the mind from the laws of logic and mathematics. Cognitive biases are believed to be among the factors which lead to an increase in software defect density. Therefore, while building defect prediction models cognitive biases should be taken into account. In cognitive psychology literature, various cognitive bias types have been defined such as representativeness, availability, adjustment and anchoring, and confirmation bias. In this thesis, we focus on *confirmation bias* which is described as the tendency of people to verify a hypothesis rather than refuting that hypothesis. All levels of software testing, including unit testing, a systematic hypothesis testing procedure should be followed similar to the one followed by a scientist making experiments in his/her laboratory. In general scientific inferences are based on the principle of eliminating hypotheses while provisionally accepting the remaining ones. As a result

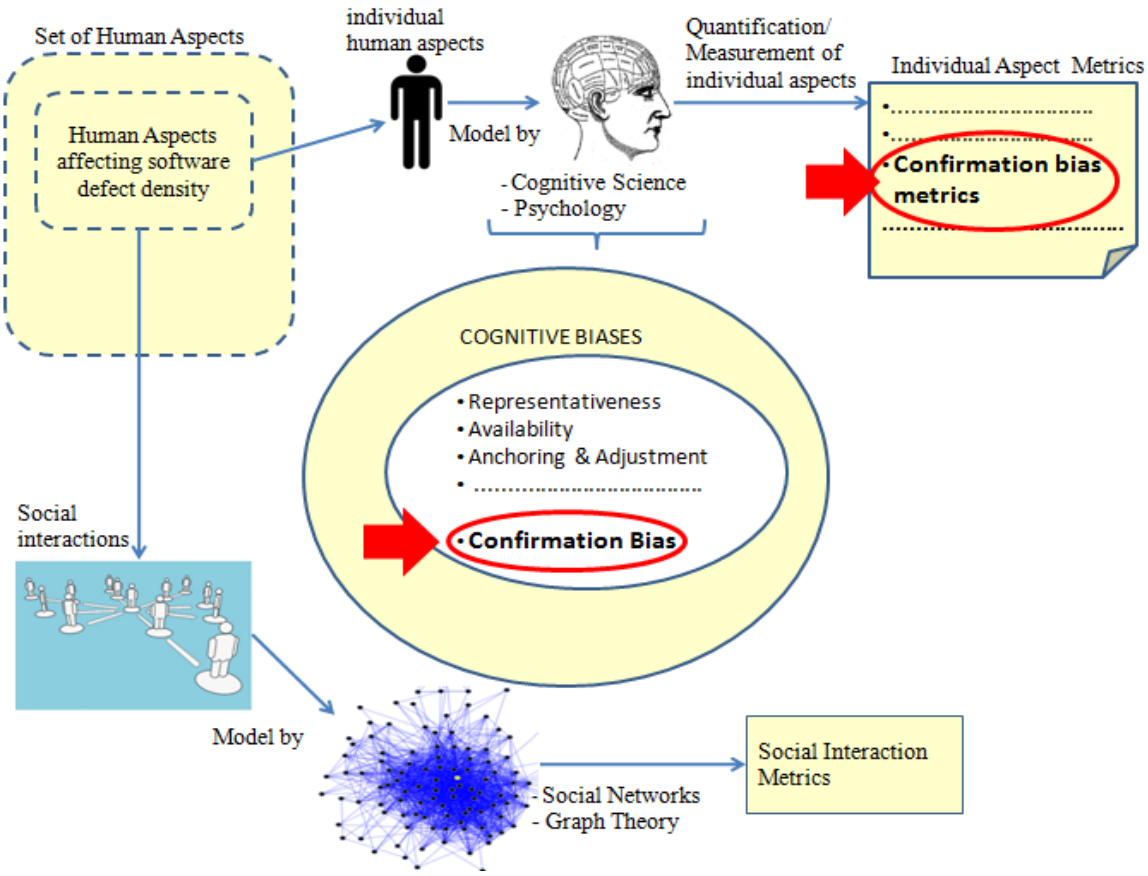


Figure 1.1. Human aspects affecting software defect density.

of an eliminative induction, it is possible to make adequate controls so that both positive and negative experimental results give information about the possible determinants of a phenomenon. Therefore, similar to a scientist a software developer/tester should try to find test scenarios which aim to fail the code being tested instead of making that code execute without any errors. However, determination of such test scenarios is not trivial, since in most cases there are infinitely many scenarios. A hypothesis testing strategy needs to be followed during all levels of software testing so that defect density of the software product can be reduced increasing software quality significantly.

1.3. Literature Review

1.3.1. Effects of Cognitive Biases on Software Engineering

Researchers have recently started to use concepts from contemporary psychology and cognitive psychology to analyze *people* aspects within the context of software engi-

neering.

In contemporary psychology, the "Big Five Model" of personality consists of five broad dimensions of personality which are extraversion, agreeableness, conscientiousness, emotional stability/neuroticsm, openness to experience, respectively. In [31] Hannay *et al.* analyze the effect of *personality* on paired programming. In this study, a well established Big Five Model based personality test is performed to 196 IT consultants from 10 different companies on three different countries. As a result of the analysis of this personality test outcomes, the authors find no strong indications that personality affects pair programming performance. Another study which uses the Big Five Personality dimensions is by Acuna *et al.* [32], where they investigate how personality relates to job satisfaction and software quality. As practical implications of their research, Acuna *et al.* recommend software professionals to measure the extraversion personality trait of developers and to form development teams with average extraversion levels in order to improve software quality.

In the field of cognitive psychology, there are many important results which can be used to explain people aspects in software engineering. These results can be used to analyze the effects of *cognitive biases* on software engineering. To the best of our knowledge, Stacy and MacMillian are the two pioneers who recognized the potential effects of cognitive biases on software engineering. In [33], Stacy and MacMillian emphasize the fact that thought process of developers are a fundamental concern in software development. The authors discuss how cognitive biases might show up in software engineering activities by giving examples from several contexts. However, this work contains no empirical investigations. The authors put forward some ideas as possible explanations and as potential areas that require further research. Empirical evidence which supports existence of confirmation bias among software testers is provided by Teasley *et al.* in [7, 8]. In their work, Teasley et. al conduct laboratory experiments as well as observing software testers in their naturalistic environment. Another empirical study which provides empirical evidence about the existence of another cognitive bias anchoring and adjustment within the context of software development belongs to Parsons and Saunders. In [34], Parson and Saunders conduct two experiments, which investigate the existence of anchoring and adjustment in software artifact reuse. The first experiment they con-

duct examines the reuse of object classes in a programming task, whereas their second experiment investigates how anchoring and adjustment bias affects reuse of software design artifacts. In their second experiment, Saunders and Parsons ask the participants to develop an Entity-Relationship (E-R) model for an airplane application. Finally, Mair and Shepperd [35] discuss how cognitive biases of software engineers contaminate the results obtained by software effort predictors making them far from being objective. This is due the fact that input parameters of prediction models are estimated by software engineers whose performance is affected by cognitive biases such as over-optimism and over-confidence. The authors also state that experiments must be made on software professionals in realistic settings to investigate possible de-biasing strategies. Mair and Shepperd also emphasize the fact that such experiments should be conducted by an interdisciplinary team consisting of cognitive psychologists and computer scientists. This work by Mair and Sheppard is in the form of a preliminary research and it contains no empirical/experimental investigation. On the other hand, Jørgensen et. al empirically investigates some cognitive bias types within the scope of software development effort estimation. According to empirical findings of Jørgensen [36], increase in the effort spent on risk identification during software development effort estimations leads to an illusion of control which in turn leads to more over-optimism and over-confidence. Moreover, as a result of the cognitive bias type availability, risk scenarios which are more easily recalled are over-emphasized so that inaccurate effort estimations are made. Jørgensen also empirically investigates how anchoring and adjustment heuristic leads to inaccurate effort estimates [37]. Jørgensen indicates that reasonable results can be obtained only if the reference value for the estimates (i.e. the anchor) is the typical effort of tasks of same category or effort of the closest analogy.

In the literature, the above mentioned empirical studies or preliminary works have been made either to investigate or discuss the existence of some cognitive bias types within the context of software engineering. However, our work is the first one that uses cognitive bias type related information as input data to enhance performance of defect prediction models.

1.3.2. People Related Metrics in Software Defect Prediction

In the literature, various people-related metrics have been used to build defect predictors, yet these are not directly related to people's thought processes or other cognitive aspects.

Nagappan *et al.* [14] defined a metric suite to quantify the complexity of organizations consisting of many teams of software professionals working together. The authors built a model to predict failure proneness of Windows Vista. They compared the performance of this defect predictor with the performance of models that are learnt using code churn, code complexity, code coverage, pre-release bugs and dependencies respectively. In terms of precision and recall values, their model outperformed all these mentioned models.

Graves *et al.* [38] also used metrics regarding development organization that worked on a specific code and number of developers who made changes on that code, as well as churn metrics for prediction of defective modules. According to the results obtained by the authors, the number of developers who have changed a module did not improve defect prediction performance. Weyuker *et al.* [39] also found that number of developers is not a major influence to increase defect prediction performance.

On the other hand, Mockus *et al.* [40] found that developer experience is essential to predicting failures. In [41], Weyuker et.al. used developer information that distinguishes developers who are new to a working file or who share responsibility of that file with other developers, since it is more likely that changes made by such developers would result in faults. However, Weyuker *et al.* detected no significant contribution of this kind of developer information to defect prediction performance. Following this research, the authors later analyzed the effectiveness of individual developer performance on defect prediction performance and no evidence of a significant improvement in defect prediction performance was found either [42].

Social interaction between developers who have collaborated to the same file during same period of time was modeled as social networks to be used in defect prediction

by Meneely et. al [43]. The model constructed for an industrial product from Nortel was able to explain 60% of the variance of failures during the testing phase. Pinzger *et al.* [44] formed a contribution network by combining modules with developers who contribute to those modules and defined centrality measures to quantify the number of developers making contribution to a specific module. Empirical analysis of the data from Windows Vista project showed that centrality metrics can predict software failures up to a significant extend. Bird *et al.* [45] formed a network which is a combination of module dependency and contribution networks to predict fault prone modules. As a result, they were able to predict fault prone binary files with greater accuracy than prior methods which use dependency networks [28] or contribution networks [44] in isolation.

1.4. Contributions

Below, we briefly explain the contributions of this thesis. Detailed explanation about the contributions shall be presented in Chapter 6.

- *Measurement/Quantification of Confirmation Bias*: Since our research is based on empirical analysis, we had to find a methodology to quantify/measure confirmation bias levels of software developers and testers. For this reason, we firstly made the definition of *confirmation bias* within the scope of software development and testing. Taking this definition as our reference point, we prepared written and interactive tests after having made an extensive survey in the cognitive psychology literature. An iterative procedure was followed to form the confirmation bias metric set. Initial metric set was defined during the literature survey and preparation of the confirmation bias tests. Metric set is updated while conducting tests and evaluating their outcomes during earlier stages of this research. Metric values are evaluated for each subject based on the answers he/she gives to these to the questions in the tests.

  Definition of a methodology to quantify/measure confirmation bias was crucial for us to proceed in our research. Moreover, this metric set can also be used by other researchers to make further research regarding confirmation bias in software engineering context. The methodology we formed to quantify confirmation bias (i.e. to form a confirmation bias metric set) can also adapted to quantify/measure other

cognitive bias types such as availability, representativeness, adjustment and anchoring, and many others. In this way, it might be possible to investigate the effects of various cognitive biases on software development life-cycle and to find de-biasing strategies to eliminate their negative effects on software quality.

- *Empirical Analysis of Factors Affecting Confirmation Bias Levels of Software Developers*: The goal of this analysis was to gain more insight about confirmation bias within the scope of software testing and development. Moreover, identifying the factors affecting confirmation bias is a step toward to find strategies for software developers to challenge their biases (i.e. de-biasing strategies). Similar analysis can be also be replicated for software testers and analysts in order to develop de-biasing strategies for each role in software development life-cycle.
- *Empirical Analysis of the Effects of Confirmation Bias on Software Developer and Tester Performances* During unit testing software developers are likely to overlook defects due to confirmation bias, which leads to propagation of more defects to the testing phase of software development. As a result, increase in software defects is inevitable. We empirically investigated the validity of this statement by building a regression model where independent variables are the confirmation bias levels of developers and the response variable is the defect density. This analysis also served as a prerequisite step to get an idea about the suitability of confirmation bias metrics to be used to build defect predictors.

  We also analyzed the effect of confirmation bas on software tester performance. Although unlike developers, the code tested by testers is not the one implemented by them, still effects of confirmation bias can be observed. As well as the tendency to make the code fail, it is essential to follow a proper testing strategy in order to find as much defects as possible. Defects overlooked by testers may propagate to the next release of the software or those defects can be detected after the release of the software. In order to empirically investigate this phenomenon, we perform a correlation analysis between the total number of post release defects detected in the files that were tested by a tester T and he confirmation bias level /i.e. values of the confirmation bias metrics for that tester T) of that tester T.
- *Benchmark Analysis to Assess Power of Defect Predictors Built Using Confirmation Bias Metrics*: In this benchmark study, our goal was to asses the power of software defect predictors which are learned using *only* confirmation bias metrics.

We used three data sets in this comparative analysis. The first and second data sets ($DataSet1$ and $DataSet2$) were collected from a large scale telecommunication company; whereas the third data set ($DataSet3$) belongs to a project group that consists of developers of the largest ISV (Independent Software Vendor) in Turkey. For each data set, we built seven different software defect prediction models using Naïve Bayes algorithm. Each of these defect predictors uses one of the seven combinations of three different metric sets (i.e. For each data set, we formed $N = 2^n - 1 = 7$ defect prediction models where $n = 3$ is the total number of metric types). The three different metric types are static code, churn and confirmation bias metrics respectively. Static code metrics represent the product pillar of software development, while churn metrics are a subset of process metrics. Finally, confirmation bias metrics are categorized under people metrics.

The results of these analysis show that defect predictors learnt using *only* confirmation bias metrics have prediction performances which are comparable with the prediction performances of defect predictors which are learnt using *only* either static code metrics or churn metrics. On the other hand, confirmation bias is a single *aspect* of *people* and can only give a limited explanation about the effect of people aspects on software defects. The empirical results we obtained in spite of this fact indicate the importance of including people metrics as input to defect predictors in order to overcome the performance ceiling.

- *Benchmark Analysis to Assess Power of Defect Predictors Built Using Confirmation Bias Metrics with Missing Data*: Compared to product and process metrics, it takes more effort to collect *people* metrics. For instance, static code and churn metrics can be automatically retrieved, while confirmation bias metric collection requires conducting tests to developers/testers and assessing the outcome of these tests. Both of these activities may take considerable time and human resource depending size of the project team. Moreover, depending on the lifetime of a software product being developed, some developers might no longer work in the company so that confirmation bias metric values of such developers cannot be obtained. If developers who used to be part of the development team were also top committers and a high portion of the files created and/or updated by them are still a part of the software product being developed, then considerable amount of confirmation bias metric values shall be missing. In such as case, we may prefer to take into account

files/modules/packages created and/or updated by developers whose confirmation bias metrics are known. However, in this case it is highly probable that our defect predictor model shall cover only a small portion of the software.

Therefore, a method is required to deal with missing data due to departure of developers so that defect prediction models built using confirmation bias metrics can cover a high portion of the software being developed. Such a method can also be used by software development companies which seek for an alternative confirmation bias metric extraction process especially for large development groups. In order to overcome the missing data problem, we employ Roweis' [46] Expectation-Maximization (EM) Algorithm for Principal Component Analysis (PCA). The algorithm naturally accommodates missing information.

The output of Roweis' EM Algorithm is the imputed form of the incomplete confirmation bias metrics which we used to learn defect predictors. In order to assess performance of these defect predictors we made a benchmark analysis. For each data set we used in our benchmark study, we formed $2^{N_d} - 2$ missing data configurations, where N_d is the number of developers in the project group. Imputation is performed as an outcome of Roweis' EM Algorithm for PCA, so that imputed forms of missing data configurations for each data set are obtained. For each data set, defect prediction experiments are replicated for the imputed forms of missing data configurations and defect prediction performance results are compared with the results obtained for original data sets (i.e. data sets with no missing data). The performance results of defect predictors which are built using imputed form of incomplete confirmation bias metrics were comparable with those of defect predictors built using complete confirmation bias metric values.

The results we obtained are promising such that Roweis' EM Algorithm for PCA can be used to perform confirmation bias based defect prediction for software products developed by large groups. Confirmation bias tests can be conducted on a subgroup and missing confirmation bias metric values can be imputed before using them to learn defect predictors. In addition, as we mentioned earlier missing confirmation bias metric values due to departure of developers from the project group can be handled using the method we propose.

1.5. Thesis Outline

This thesis is organized as follows: Chapter 2 gives necessary background on cognitive biases in general, confirmation bias as well as explaining the role of confirmation bias in software development and testing. Research questions are also presented in this chapter. In Chapter 3, we explain the methodology we used to measure/quantify confirmation bias to be used in our empirical analysis. Chapter 3 also mentions statistical methods we used to interpret the results of the experiments we conducted to investigate the factors affecting confirmation bias as well as the effect of confirmation bias on software developers and testers. Defect prediction and missing data imputation methods are explained in Chapter 3. Details about the overall data set used in the experiments are presented in Chapter 4. Design and results of the experimented conducted in this study as explained in Chapter 5. Chapter 5 also gives information about which parts of the overall data set is used in which experiment together with the reason for such a data subset selection. In Chapter 6, we conclude our research, summarize our results giving answers to the research questions we stated previously and finally we point out possible future directions.

2. PROBLEM STATEMENT: BACKGROUND AND RESEARCH QUESTIONS

2.1. Cognitive Biases in General

Cognitive biases are defined as the deviation of human mind from the laws of logic and accuracy [33]. The notion of cognitive biases was first introduced by Tversky and Kahneman [47]. Other than confirmation bias, among cognitive biases that are most likely to affect software development and testing are representativeness, availability, and adjustment and anchoring. In the following subsections, we will briefly explain the cognitive biases representativeness, availability, adjustment and anchoring, as well as giving examples on how these biases may affect software development and testing. Detailed information about confirmation bias, which is the cognitive bias type we focus in this thesis, is given in the next section.

2.1.1. Representativeness

While making predictions and judgments under uncertainty, people do not appear to follow the statistical theory of prediction. As Kahneman and Tversky states in [47], people rely on heuristics which sometimes yield reasonable judgments and which sometimes lead to severe and systematic errors. Although outcomes representative of a population of instances are more likely to belong to that population, this is not always true. One should also take into account the base rate (i.e. prior probability).

Representativeness can also affect testing phase. Since exhaustive testing is not possible, we have to plan tests such that test data that does not produce an error are eliminated. We can identify test data to be eliminated based on their similarity to the data that did not produce any errors in previous tests. However, assessment of similarity is not trivial. Assume that a developer has to decide whether a set of test data y should be eliminated or not. Such a decision depends on the similarity of y to X where X belongs to the population of test data that did not produce any errors. According to Bayes' Theorem, we can represent such similarity by the notation $P(X|y)$ and formulate

as follows:

$$P(X|y) = \frac{P(y|X)P(X)}{P(y)} \tag{2.1}$$

Due to representativeness, people usually do not take base rates (prior probabilities) $P(X)/P(y)$ into account and hence similarity and likelihood values coincide. In other words, the degree to which y is representative of population X (i.e. $P(X|y)$) is calculated without taking base rates into account.

2.1.2. Availability

Similar to representativeness, availability is also a heuristic to judge probability or frequency of instances. Using availability heuristic, people estimate the probability of an instance according to its associative distance. Lifelong experience has told us that instances of large classes are recalled much more easily compared to instances of less frequent classes. However, availability is also affected by factors which are unrelated to actual frequency. For instance, humans judge the frequency of instances by the ease that they come to mind. Availability might reveal itself during the developer testing phase of the software. In addition to formal tests, good programmers also use some heuristic techniques to expose errors in their code. One heuristic technique is "error guessing", which means creating test cases based upon guesses about where the program might have errors. Such guesses might be based on past experience of the developer about the type and location of the errors he has made most frequently. In this case, he can misidentify the most frequently occurring error types and locations based on the ease that they come to mind. For instance, areas of the code where the developer spent more time might be much more easily remembered by the developer. Moreover, as a result, most frequently occurring errors might be overlooked.

2.1.3. Adjustment and Anchoring

When people are given problems whose solutions exist, their estimation of the problem varies depending on different starting points. In other words, their estimates are biased towards the initially given values. We call this phenomenon anchoring. The

study mentioned in [47] illustrates this effect.

Two groups of high school students were asked to estimate a numerical expression within 5 seconds. The numerical expression given to the first group was 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1. On the other hand, the following expression was given to the second group: 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8. The median estimate of the descending sequence was 2250, while the median estimate for the ascending sequence was 512 and the correct answer is $40,320$. In this experiment, after a few steps of computation, people estimate the product by extrapolation. Anchoring heuristic helps to simplify a complex problem without conscious effort. However, people often fail to make adequate modifications to an initial solution, which result in incorrect outcome. This phenomenon is known as adjustment bias.

Parsons and Saunders [34] analyze the effects of adjustment and anchoring in software artifact reuse. Due to the anchoring and adjustment bias, software developers unconsciously adhere to the reuse artifact. Therefore, the errors existing in the reuse artifacts propagate or necessary functionality additions indicated in the requirements specification document are omitted. Moreover, extraneous functionalities that do not exist in the requirements specification remain in the reuse artifacts.

2.1.4. Wason's Rule Discovery Task

In this experiment, Wason asked his subjects to discover a simple rule about triples of numbers [1]. Initially, subjects are given a record sheet on which the triple "2, 4, 6" is written. The experimental procedure can be explained as follows: The subjects are told that "2 4 6" conforms to this rule.

In order to discover the rule, they are asked to write down triples together with the reasons of their choice on the record sheet which is shown in Figure 2.1 After each instance, the tester tells whether the instance conforms to the rule or not. The subject can announce the rule only when he/she is highly confident. If the subject cannot discover the rule, he/she can continue giving instances together with reasons for his/her choice. This procedure continues iteratively until either the subject discovers the rule or he/she

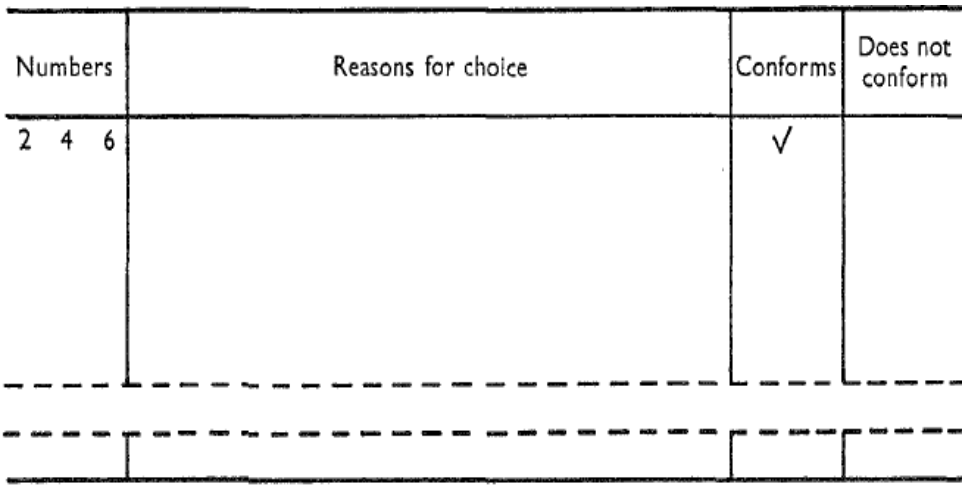


Figure 2.1. Record sheet used by Wason on his Rule Discovery Task [1].

wishes to give up. If the subject cannot discover the rule in 45 minutes, the experimenter aborts the test. The flowchart explaining the above mentioned test protocol is given in Figure 2.2.

Wason designed this experiment in a way such that subjects mostly showed a tendency to focus on a set of triples that is contained inside the set of all triples conforming to the correct rule. Due to this fact, discovery of the true rule was possible only by refuting hypotheses that come to mind. Figure 2.3 shows the relationship between a hypothesis that is likely to come to participant's mind once he/she sees the triple "2 4 6", the rule to be discovered and the set of all rules in the universe about triples of numbers.

While designing his experiment Rule Discovery Task, Wason's inspiration was Popper's criticism approach:

> A universal statement cannot be verified purely logically by a limited number of observations but it can be falsified

As Poletiek also indicates, Popper's influence on Wason is recognizable in the analogy he makes between a scientific laboratory experiment and his Rule Discovery Task:

> The task simulates a miniature scientific problem where the variables are unknown and in which evidence has to be systematically adduced to refute or support hypotheses. Generating an instance corresponds to doing an experiment, knowledge that the instance conforms or does not conform, corresponds to its result and an

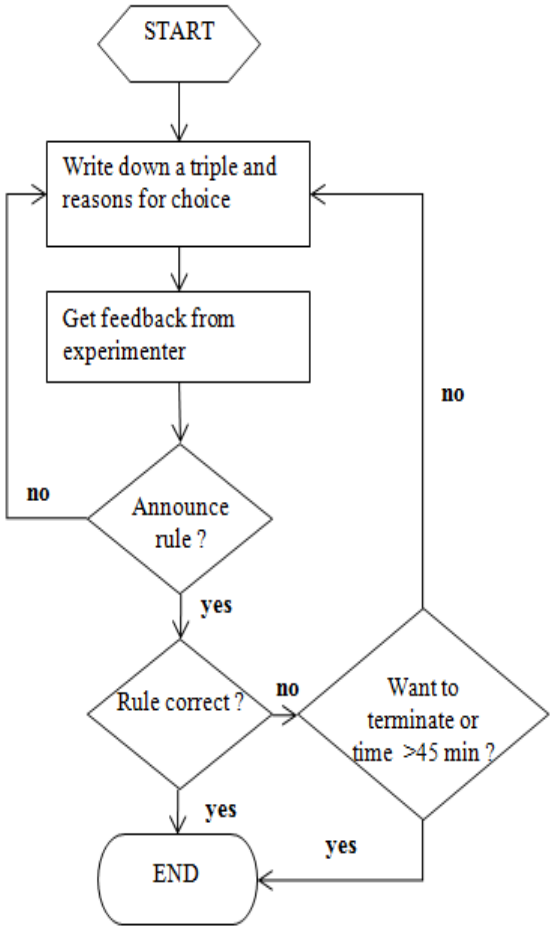


Figure 2.2. Protocol followed during Wason's Rule Discovery Task.

incorrect announcement corresponds to an inference from uncontrolled data.

As we shall see in the following subsections, one must also follows a hypothesis testing strategy has to succeed in Wason's Rule Discovery Task. Once, the subject sees the triple "2 4 6" a set of hypotheses come to her/his mind. An ideal hypothesis testing strategy is to start by giving examples which does not refute all hypotheses the subject has in his/her mind at once. The examples of triples that refute more hypotheses should be given as the test proceeds. The hypotheses should be eliminated,modified and created in a strategic manner, so that subject can come up with a single hypothesis at the end. Once the subject is sure about what the rule to be discovered is, he/she can give additional triple instances to verify his/her guess.

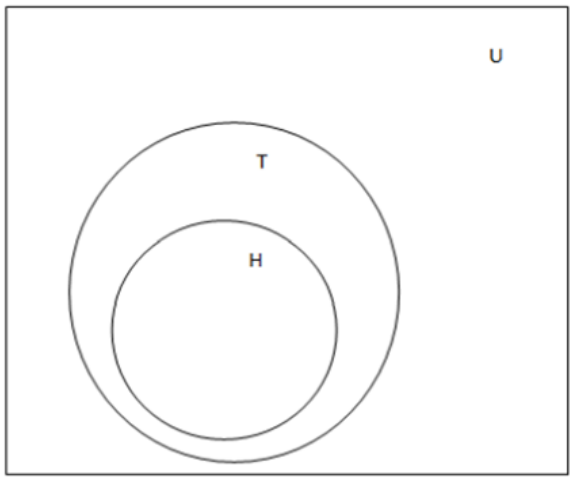


Figure 2.3. Relation between alternative hypotheses and correct rule in Wason's Rule Discovery Task (T: set of triples conforming to the correct rule, H: set of triples conforming to the hypothesis, U: set of all possible triples).

2.1.4.1. Wason's Elimination/Enumeration Index. Eliminative/enumerative index was constructed by Wason [1] in order to determine the proportion of the total number of instances that are incompatible with reasons given to those that are compatible.

According to Wason, it is desirable that eliminative/enumerative index is greater that one. The higher the eliminative/enumerative index, the more tendency the subject as for refuting his own hypotheses. However, as we shall see in the following subsections "eliminative" testing strategy is not alone adequate to succeed in Wason's Rule Discovery Task.

2.1.4.2. Negative Instance Frequency. In [1], for each subject Wason evaluates the ratio of the number of instances which does not obey the rule to be discovered (i.e. negative instances) to the total number of instances given by that subject. Wason also observes a highly significant correlation between eliminative/enumerative index and negative instance index (Kendall's Tau being 0.72, where $p < 0.00003$).

2.1.4.3. Test Severity. Poletiek [2] mentions severity of the tests. The term "test" used by Poletiek corresponds to a single instance given by the subject, during Wason's Rule Discovery Task , during which a subject gives several instances to discover the rule.

Popper, who was Wason's source of inspiration in the design of his famous Rule Discovery Task, designated the severity of a test (i.e. instance) as follows:

$$S(t, H, b) = \frac{P(t|H, b)}{P(t|b)} \tag{2.2}$$

According to the above equation, the severity of a test t is interpreted as the supporting evidence of the hypothesis H given the background knowledge b. A test is more severe when the chance of the supporting observation occurring under the assumption of the hypothesis H exceeds the chance of its occurring without the assumption of the H (i.e. with the assumption of the background knowledge b only). The higher this ratio is (exceeds 1), the higher the severity of the test is. In other words, when the severity of a test is high, more alternative hypotheses are eliminated.

In line with Popper's idea, it was expected that rule discovery performance would increase as the subject tested more severely (i.e. gave instances that would eliminate more alternative hypotheses). In order to investigate the validity of this expectation, Poletiek performed an experiment, where she estimated the severity of the tests performed by participants. In this experiment, different from the original Wason's Selection Task participants were given "2-7-6" as an initial example and the rule to be discovered was "even-uneven-even number". Poletiek knew that calculation test severities (i.e. severity of given instances) was not a trivial task, since the set of "all possible hypotheses" is infinite as shown in Equation 2.2. On the other hand, in psychological terms this set is not infinite due to the fact that people cannot easily easily more than one hypothesis at a time in their mind. Moreover, this set is fuzzy in people's mind and in order to perform a more or less severe hypothesis testing, it is not necessary to explicitly generate all the hypotheses. Hence, in order to make test severity calculations in her experiment, Poletiek took all plausible reasons given by all the participants as the set of all alternative hypothesis S_H (i.e. background knowledge). The illustration of positive and negative instances by Poletiek is given in Figure 2.4. In this Figure, H is the set of triples conforming to the hypothesis of the participant. A_1, A_2 and A_3 are the three sets of triples conforming to the sets of alternative hypotheses A_1, A_2 and A_3 respectively. U is the set of all possible triples. In Figure 2.4, Poletiek uses the notation x^+ to represent the set of severe positive triples, the notation x^- to represent the set of severe negative

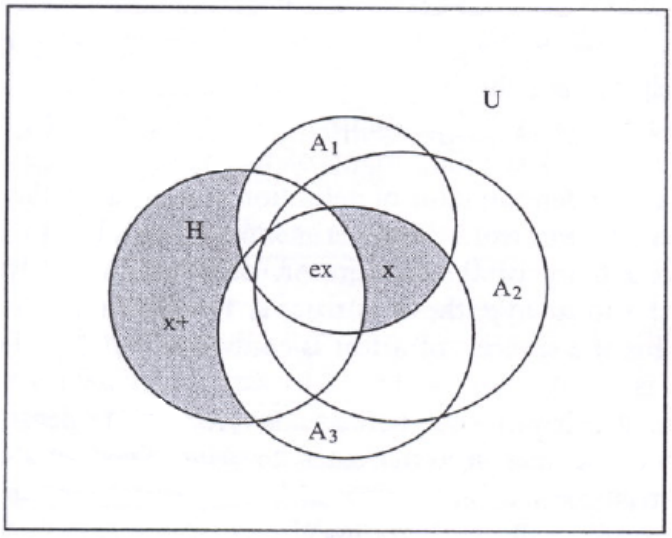


Figure 2.4. A severe positive and a severe negative test in Wasons rule selection task)[2].

triples. Finally, she uses *ex* to represent the example triple that's been given by the participant during the experiment.

Using the set of alternative hypotheses, she calculated severity of each instance given by each participant as follows:

- If the instance I is a positive instance (i.e. conforms to the correct rule), then severity of the instance I is the number of hypothesis in the set S_H to which I does not conform.
- If the instance I is a negative instance (i.e. does not conform to the correct rule), then severity of the instance I is the number of hypothesis in the set S_H to which I conforms.

When she performed Mann Whitney U Test to find difference between test severity values of participant who discovered the correct rule and those of who could not, she did not observe any significant difference. Therefore, she concluded that a falsifying strategy and high severity values is not enough to discover the correct rule. Hence, according to Poletiek [2, 48] (and also McDonald [49]) a reasonable strategy to discover the correct rule is start with giving a low level "severe" instance and progressively exclude more alternative hypothesis rather than maximizing the severity per given instance.

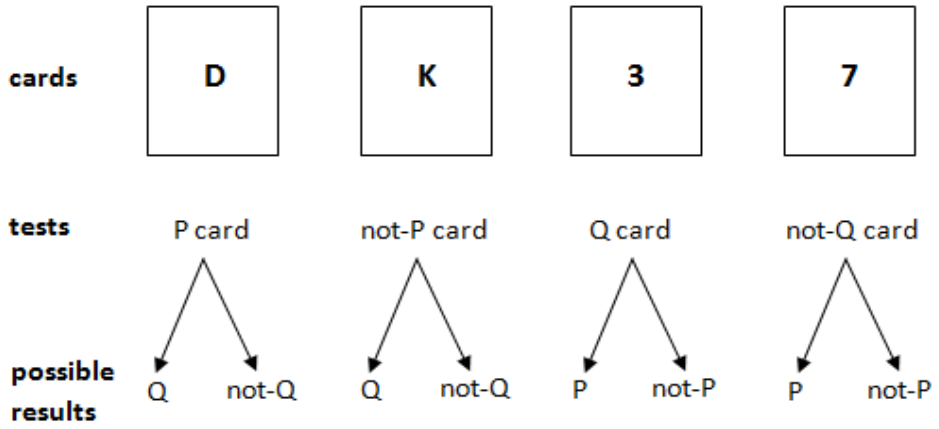


Figure 2.5. The logical structure of Wason's selection task [2].

2.1.5. Wason's Selection Task

In the original task, the subject is given four cards which are placed on a table showing respectively D, K, 3, 7. Each card has a letter (either D or K) on one side and a number (either 3 or 7) on its other side. Given the rule (hypothesis): "Every card that has a D on one side has a 3 on the other side", the subject is asked which card(s) must be turned over to find out whether the rule is true or false.

Figure 2.5 explains the logical structure of Wason's Selection Task and how it simulates hypothesis testing behavior. Given a hypothesis (rule), we make tests/experiments. Based on the results of our tests/experiments, we arrive to a conclusion about the validity of that hypothesis. In the setting of "Wason's Selection Task", we find "Card selection" corresponds to making some experiments/tests each of which has two possible results (i.e. For each card there are two possibilities about what the letter/integer might be on its invisible side).

The hypothesis can be translated into the logical implication of the form "If P, then Q" ($P \Rightarrow Q$), whereas each test is the selection of one of the cards (P, not-P, Q, not-Q). Wason interprets selection of the cards D and 3 (i.e. P and Q) as a choice of a *verifier*, whereas the subject is defined to be a *falsifier* if he/she also selects the cards D and 7 (i.e. P and not-Q). Wason's selection task measures the capability of the subject to use

two rules of logic, as well as his/her tendency to refute the given statement. The first rule is modus ponens:

$$\frac{P, P \Rightarrow Q}{Q} \tag{2.3}$$

Given a statement S of the form $P \Rightarrow Q$ (i.e."If P, then Q"), according to modus ponens if statement S is true and P is true (i.e. there is a card with letter D on its visible side), then Q must also be true. This implies that we must turn over the card which has D on its visible side. In order to prove or disprove statement S, we must also try to falsify it, which is possible by modus tollens (In the equation below ¬P stands for not-P, while ¬Q stands for not-Q).

$$\frac{\neg Q, \neg Q \Rightarrow \neg P}{\neg P} \tag{2.4}$$

The statement $P \Rightarrow Q$ is equivalent to $\neg Q \Rightarrow \neg P$. According to modus tollens, we must turn over the card which does *not* have a 3 on its visible side (i.e. card which has a 7 on its visible side) to see if it has a K on its other side (i.e. statement S is true) or it has a D on its other side (i.e. statement S is false).

2.1.5.1. Matching Bias. Matching bias may lead subjects to select cards on the basis of a simple judgment of relevance. In other words, the selection of the cards D and 3 can also result due to matching of the letter D and number 3 in the stated hypothesis. Determining whether the card selection is made by matching or employing logical rules requires use of rules with negated components as shown in Table 2.1. Evans and Lynch [50] used the negated version of the selection task (i.e. if P, then not-Q) as well as the original task (i.e. if P, then Q). In this experimental study, the subjects chose P and Q cards, instead of P and not-Q cards. Evans and Lynch interpreted subjects' behavior as either being falsifying or matching. However, if a subject, who has chosen P and Q cards in the standard version, also selects P and Q cards in the negated version, such behavior can be explained only by *matching bias*. Otherwise, subject's *verifying* behavior accompanied by *falsifying* behavior would not make sense. Table 2.2 shows, all four negated components that should be used to predict matching bias. Reich and Ruth

Table 2.1. Prediction of selection of cards according to different response tendencies.

Tendency	**P ⇒ Q**	**P ⇒ ¬Q**	**¬P ⇒ Q**	**¬P ⇒ ¬Q**
Matcher	P, Q	P, Q	P, Q	P, Q
Verifier	P, Q	P, ¬Q	¬P, Q	¬P, ¬Q
Falsifier	P, ¬Q	P, Q	¬P, ¬Q	¬P, Q

Table 2.2. All four negated versions of Wason's Selection Task statements.

Rule	**Logical Statement**
If there is a **D** on one side, then there is a **3** on the other side.	$P \Rightarrow Q$
If there is a **D** on one side, then there is **not** a **3** on the other side.	$P \Rightarrow \neg Q$
If there is **not** a **D** on one side, then there is a **3** on the other side.	$\neg P \Rightarrow Q$
If there is **not** a **D** on one side, then there is **not** a **3** on the other side.	$\neg P \Rightarrow \neg Q$

[51] used all negated versions of Wason's Selection Task as well as the original task itself, in order to determine response tendencies. The method of Reich and Ruth is explained in Table 2.3.

This method of determining response tendencies is advantageous, as it does not confound strategies that might have contributed to a particular selection. However, it neglects a large proportion of data provided by the subjects. On the other hand, it gives a general view about the subjects' responses and it is the only classification strategy we came across in the existing psychology literature. For these reasons, we used the method of Reich and Ruth and we labeled subjects, whom we could not classify, as *None*.

Table 2.3. Response tendencies according to Reich and Ruth's Categorization.

Rule	**Card**	**Tendency**
If there is a **D** on one side, then there is a **3** on the other side.	7	Falsifier
If there is a **D** on one side, then there is *not* a **3** on the other side.	7	Verifier
If there is **not** a **D** on one side, then there is a **3** on the other side.	D	Matcher
If there is **not** a **D** on one side, then there is a **3** on the other side.	7	Falsifier
If there is **not** a **D** on one side, then there is **not** a **3** on the other side.	D	Matcher
If there is **not** a **D** on one side, then there is **not** a **3** on the other side.	7	Verifier

2.1.5.2. Realistic Content Replications of Wason's Selection Task. Wason and Shapiro [52] performed the first thematic content replication of Wason's selection task. In this study, the rule with thematic content was: "Every time I go to Manchester, I travel by train". On the exhibited faces of two of the four cards, destination names "Manchester" and "Leeds" were written; whereas on the remaining two one could see the the modes of transportation, namely "car" and "train". The results showed that more subjects succeeded in this thematic variant compared to the original Wason's selection task. Griggs and Cox [53] used the hypothesis "If a person drinks beer, he must be older than 19." This hypothesis was in the form of a rule that one must obey. *Deontic* nature of this hypothesis facilitated more correct results.

In the literature, there are various studies which aim to explain the facilitating effect of thematic content in Wason's selection task [53–56]. Among these, is the *memory cueing*, which has been elaborated by Griggs and Cox [53]. Griggs and Cox explain *memory cueing* as the production of cues by memory to solve the selection task correctly while pure logical reasoning is bypassed. However, various studies showed that *memory cuing* cannot be the sufficient reason for the facilitation effect. In [54], the experiment subjects were placed in the situation of a store manager and they were given four receipts to check whether the following rule is violated or not:"Receipts in excess of 30 dollars must have the signature of the department manager on the back". Compared to the abstract selection task, there was a facilitation effect which obviously cannot be explained by memory cueing as none of the test subject has ever had an experience as a store manager. In order to explain such results, Cheng and Holyoak [57] proposed the *pragmatic reasoning effect*. Poletiek [2] makes a comparison of reasoning patterns with memory cueing and logical reasoning as follows:

> A *reasoning pattern* is more abstract than a specific knowledge resident in memory and more specific and concrete than rules of logical reasoning.

Cheng and Holyoak [57] indicate *permission pattern* as the fundamental pattern of selection tasks with facilitating contents. The beer drinking problem of Griggs and Cox [53] can an example where *permission pattern* is employed for the correct answer. Cosmides [55] proposed *social contract theory* as an explanation to the facilitating effect in Wason's selection task with thematic content. Cosmides state that conditional hypotheses

comply with the following basic rule: "If you profit (in a social contract), you must pay the costs". According to Cosmides, *cheating detection mechanism* is triggered during the solution of selection tasks with thematic content. On the other hand, Manktelow and Over [56] indicate that facilitation effect in every variant of Wason's selection task with thematic content cannot be explained by social contract theory. When propositions are conditional obligation such as "If you clean up blood, you must wear rubber gloves", then facilitation occurs but without invoking social contracts. In this case violation is related with high costs.

Table 2.4. Examples of different contents used in the selection task.

Name	**Original Study**	**Example Rule**
Postal rule	Johnson-Laird *et al.* (1972) [58]	If the letter is sealed, then it has a 50 lire stamp on it.
Food and drinks	Manktelow and Evans (1979) [59]	If I eat haddock, then I drink gin.
Store manager problem	D'Anrade in (1983) [60]	If a purchase exceeds 30 dollars , then the receipt must be signed by the departmental manager.
Drinking age rule	Griggs and Cox (1982) [53]	If a person is drinking beer, then that person must be over 19 years of age.
Clothing age rule	Cox and Griggs (1982) [61]	If a person is wearing blue, then that person must be over 19 years of age.
Abstract permission rule	Cheng and Holyoak (1985) [57]	If one is to take action "A", then one must first satisfy precondition "P".
Standard social contract	Cosmides (1989) [55]	If a man eats cassava root, then he must have a tattoo on his face.
Deontic conditionals	Manktelow and Over (1990) [56]	(Mother to son): "If you tidy your room, then you may go out to play."
Precautionary rule	Manktelow and Over (1990) [56]	If you clean up blood, you must wear rubber gloves.
Towns and transport	Wason and Shapiro (1971) [52]	Every time I go to Manchester, I travel by train.

2.2. Confirmation Bias in Software Development and Testing

During all levels of testing the goal must be to make the code fail, otherwise defects shall be overlooked leading to an increase in software defect density. Software developers must also consider test scenarios that have the potential to fail their code during unit testing, so that less defects propagate to the testing phase. As the number of defects which propagate to the testing phase increases, it becomes very likely that more defects are overlooked before the software is released due to the limited time reserved for testing. Therefore, the number of post-release defects increases which in turn becomes a risk for producing high quality software. However, as mentioned previously it is not enough for a developer to have a tendency to make the code fail. The number of test scenarios which can be used to test the code can be infinitely many. Moreover, time for unit testing before the code freeze date (i.e. the time when development of a software release ends and testing phase for that release starts) is limited. The unit testing performed by developers is analogous to Wason's Rule Discovery Task, because firstly developer is supposed to make his/her *own* code fail; whereas participants need to refute the alternative hypotheses in their *own* mind during Rule Discovery Task. Looking at the overall format of Wason's Rule Discovery Task, we can say that the procedure followed by the participant resembles black box testing.

Different from Wason's Rule Discovery Task, in Selection Task, participant is given a statement he/she must try to refute to prove its validity regarding four cards presented to him/her, each with only one side visible. The statement does not belong to the participant himself/herself. From this point of view, Wason's Selection Task is analogous to testing software after code freeze date when testers test codes developed by developers (i.e. testers test someone else's code). Unlike unit testing performed by developers, tester needs to get an idea about how the software piece implemented by someone else works. Therefore, it is more appropriate to start with test scenarios that make the code work. Progressively test scenarios which are more likely to make the code fail may be tried. Such a strategic testing approach is analogous to the hypothesis testing strategy that helps to discover the correct rule in Wason's Rule Discovery Task.

On the other hand, "eliminative" approach that was suggested by Wason is required

during all types of testing. As stated by Popper, we cannot verify a universal statement by limited number of observations, yet falsify it. In our case, this corresponds to testing codes in an attempt to see them function properly using limited number of test cases. Time is limited to complete all the testing in order to release the software. Moreover, there is usually infinitely manu test scenarios. For this reason, it is crucial to find test scenarios which are more likely to fail the code. Moreover, we must cover the test scenario space in a way such that we do not always pick scenarios which leads to frequently detecting the same defect. In other words we must avoid "enumerative" testing. Only covering the space of test scenarios in a strategic manner may help us to detect as much defect as possible.

Finally, using logical inference rules modus ponens and modus tollens may be useful during software testing, as it helps to select the correct cards during Wason's Selection Task. We can explain our claim by the question below which is an adaptation of Wason's Selection Task to software engineering domain by Stacy and McMillian [33]. In this question, we are given the following hypothesis: *If an instance's class is Controller, then it has been initialized.*. The remaining of the original question was as follows:

Which of the following(s) need(s) investigation ?

i. *An instance of Controller that may or may not be initialized.*

ii. *An instance of a class other than Controller that may or may not be initialized.*

iii. *An initialized instance whose class is unknown.*

iv. *An uninitialized instance whose class is unknown.*

The correct answer of the question is "*i* and *iv*".Therefore, knowledge of modus ponens and modus tollens may help us to detect defects especially during white box testing.

2.3. Research Questions

In this section, we state our research questions. In the rest of this thesis, we will perform several experiments in order to obtain empirical evidence to find the answers of these questions.

2.3.1. What are the factors affecting confirmation bias levels in software development domain?

In order to answer this question, we make a list of factors which have the potential to affect confirmation bias levels of software developers. We focus on two main factors which are experience, and reasoning and hypothesis testing skills respectively. We also analyze the effect of company size, and the effect of roles in the company (i.e. developer, tester, analyst, etc.). We perform statistical tests to find out whether there is a significant correlation between each of these factors with confirmation bias levels of software developers, testers and analysts.

The goal of this analysis is to gain insight about confirmation bias metrics. Moreover, knowing which factors affect confirmation bias may help us to find methods to circumvent the negative effects of confirmation bias. In this way some de-biasing strategies may be developed.

2.3.2. How does confirmation bias affect software developers and testers?

We use total number of post release defects which originate from part of software tested by each tester T as the performance measure of that tester. We perform a correlation analysis to find out if there is a correlation. Moreover, using confirmation bias metrics of testers as independent variables and performance measure od testers as response variable, we perform a regression analysis. A similar regression analysis is performed for software developers.

Although there are some hypotheses about the effect of confirmation bias on software tester and developer performance, we needed some empirical results.We needed to make this analysis to have some empirical evidence about the potential of confirmation bias metrics to be used to learn defect predictors so that we can overcome performance ceiling of defect predictors.

2.3.3. How can we build defect predictors with comparable prediction performance by using only confirmation bias metrics as input data?

In order to answer this research question, we built defect predictors using all combinations of static code, churn and confirmation bias metrics. The results of our experiments showed that comparable prediction performance results can be obtained using *only* confirmation bias metrics to build defect predictors. This result also has a very important implication regarding the use of people metrics to overcome performance ceiling of defect predictors. Although confirmation bias is a very small aspect of people, obtaining such promising prediction performance results implies that researchers must focus on people aspects to build defect predictors.

2.3.4. How can we build defect predictors with comparable prediction performance using incomplete confirmation bias metric data as input?

Unlike static code and churn metrics, which are automatically collected, it requires more effort to collect confirmation bias metrics. Moreover, as the lifetime of the software being developed increases some developers might leave the software company so that it would be impossible to collect their confirmation metrics. Eliminating entries with missing confirmation bias metrics would result in building defect predictors which can make prediction only for a specific portion of the software product. This portion might even be very low depending on the number of no longer existing developers who once created/updated most of the files of the software product which is still being developed. Hence, we searched for an alternative solution where we can impute the missing data and use the imputed data consisting of only confirmation bias metrics to learn defect predictors. The results we obtained were promising which would help us to deal with missing confirmation bias metric values. In addition, the employed methodology can be used to make defect prediction in large development groups spending as much resources as possible to collect confirmation bias metrics. After having collected confirmation bias metrics from a certain amount of developers, the confirmation bias metrics related to rest of the developers in the project group can be treated as missing data.

3. METHODOLOGY

This chapter explains in detail the methodology used to measure/quantify confirmation bias levels of software developers/testers, the statistical methods used to analyze the effects of some factors on confirmation bias as well as the effect of confirmation bias on software developer and tester performance. In addition, this chapter contains information about Naive Bayes Algorithm used to build defect predictors and Roweis' EM Algorithm for PCA to impute missing confirmation bias metrics.

3.1. Measurement/Quantification of Confirmation Bias

The overall methodology to define confirmation bias metrics and extract metric values is shown in Figure 3.1. Based on our extensive survey about confirmation bias, we formed an initial confirmation bias metric set. We kept on modifying our confirmation bias metrics set, while we prepared our written and interactive tests. Moreover, the content of the written tests were also determined based on the some confirmation bias metrics we decided to include into our metric pool. Hence, we can say that there was feedback from the outcomes of confirmation bias metric set formation phase to test preparation phase as well as the other way round (i.e. feedback from test preparation phase to metric phase formation phase). During conducting our interactive test, our observations about participants' hypothesis testing behavior to find the correct rule led us to define metrics to monitor hypothesis testing strategies. In addition, while we were evaluating both written and interactive test outcomes, we were able to update our metric pool. The final content of the metric set, we used in our experiments is given in Table 3.2 and 3.3.

3.1.1. Preparation of Confirmation Bias Tests

Interactive test is Wason's Rule Discovery Task itself, whereas written test is based on Wason's Selection Task. The details about both written and interactive tests are given in the following subsections.

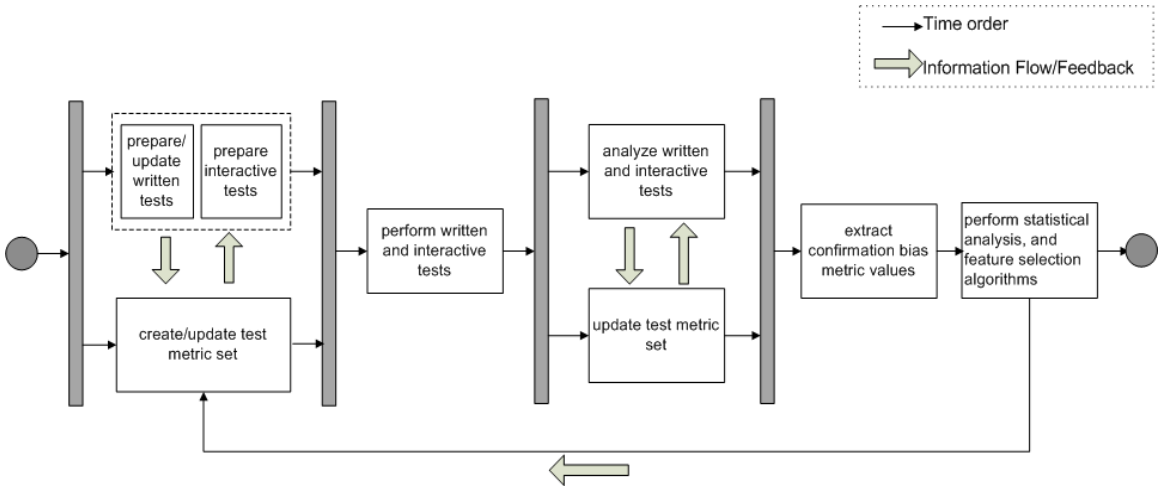


Figure 3.1. Methodology used to measure/quantify confirmation bias metrics.

3.1.1.1. Written Tests. There are four different types of questions in the written test which are abstract, abstract-thematic and thematic questions as well as questions with software development/testing theme. Written test consists of two parts. In the first part, abstract and thematic questions are presented in an order such that an abstract question is followed by a thematic question and vice versa. In addition to abstract and thematic questions,the first part also contains an abstract-thematic question. The second part also consists of thematic questions, however these questions have software development/testing theme.

Abstract questions require pure logical reasoning to be answered correctly. In our test, there are eight abstract questions. One of the questions is the original question in Wason's Selection Task [1]. In addition to Wason's original question, there are 3 questions where the statement that is to be either proved or disproved is of the form "*If P, the Q*" ($P \Rightarrow Q$). Moreover, in order to identify the tendency of the participant according to Reich and Ruth's categorization (i.e. whether the participant is *falsifier*, *verifier*, *matcher* or *none*) all three negated versions of the original question in Wason's Selection Task are also included in the written test.

- *If P, then not Q:* $P \Rightarrow \neg Q$
- *If not P, then Q:* $\neg P \Rightarrow Q$
- *If not P, then not Q:* $\neg P \Rightarrow \neg Q$

As mentioned previously, there is only one abstract-thematic question and it is in the first part of the written test. Abstract-thematic questions seem to have theme based on daily life, however they can be answered correctly only based on logical reasoning.

Unlike abstract and abstract-thematic questions, it is possible to answer thematic questions without pure logical reasoning. The first part of the written test contains seven thematic questions. These questions aim to stimulate various mechanisms/phenomena that people can employ to answer thematic questions rather than pure logical reasoning. Among these mechanisms/phenomena are daily life experiences, memory cueing, precautionary effect, cheating detection mechanism from Cosmides' social contract theory. While preparing thematic questions, we consulted related literature (refer to Table 3.1). One of the questions is the famous "Drinking Age Problem" by Griggs and Cox [53]. There is also one "Precautionary Rule" very similar to the one by Manktelow and Evans in [59]. Two different versions of a single question based on Cosmides' original "Social Contract" rule [55] are included in the thematic test. In both of these versions, the participants are asked to prove/disprove the following statement:

"If an employee gets a pension, that employee must have worked at least ten years.". In the first version, the participant is told that he/she must assume he/she is an *employee*; whereas in the second version we try to make the participant think from *employer*'s point of view. If a participant answers these two questions based on cheating detection mechanism according to Cosmides' Social Contract Theory, then he/she cannot give the same answer to them. On the other hand, if the participant answers these two questions using pure abstract reasoning he/she gives the same answer to both questions which is the correct one. One can give the correct answer to "Drinking Age Problem" by Griggs and Cox by employing the cheating detection mechanism rather than pure logical reasoning. There is one more question which also stimulates the cheating detection mechanism and this makes four such questions in total.

Finally, we also included an adaptation of the "Towns and Transport" rule by Wason and Shapiro [52]. In the written tests which we conducted to Turkish participants, the original statement to be proved/disproved "Every time I go to Manchester, I travel by train" is replaced by the statement "Every time I go to Ankara, I travel by train" in

Table 3.1. Examples of different contents used in the selection task.

Question Type	**# of Questions**
Written Test: Part I	
Abstract	7
Abstract+Thematic	1
Thematic	7
Written Test: Part II	
Software Development/Testing	8
TOTAL	23

order to facilitate the familiarity of the participants based on their daily life experiences and memory cueing.

The questions in the second part of the written test were also thematic questions where pure logical reasoning can be bypassed by experience in software development and testing. Yet, the answers to these questions can be correct. While preparing this part of the written test, one of the questions is taken from [33]. This question was mentioned in Section 2.2 of this thesis, whereas the rest of the questions were prepared by us.

3.1.2. Definition of Confirmation Bias Metrics

In Table 3.2 and Table 3.3, the final set of confirmation bias metrics are shown. These metrics can be categorized under the following titles.

3.1.2.1. Performance Metrics. Performance metrics are concerned with the final results of the tests. In interactive tests, final results consist of whether subject discovers the correct rule or aborts the test (IND_{ABORT}), total number of rule announcements made by the subject (N_A) and total time it takes for the subject to complete the test with or without success (T_I). Performance metrics for written tests consist of test scores as shown in Table 3.2 and time (in minutes) it takes for the subject to complete the tests (T_{SW}, T_{Th+ABS}).

Table 3.2. List of confirmation bias metrics (Performance Metrics).

Metric	**Explanation**	**Test Type**	**Data Type**
S_{Th}	Score in thematic questions	Written	Categorical
S_{ABS}	Score in abstract questions	Written	Categorical
S_{SW}	Score in questions with software theme	Written	Categorical
T_{Th+ABS}	Time it takes to solve general test (in minutes)	Written	Continuous
T_I	Interactive test duration (in minutes)	Interactive	Continuous
N_A	Total number of rule announcements	Interactive	Categorical
IND_{ABORT}	Total number of rule announcements	Interactive	Categorical

3.1.2.2. Metrics to Monitor Hypothesis Testing. These metrics are only extracted for interactive test to monitor procedure followed by the subject and to find out whether subject follows a sensible hypothesis testing strategy or not. Behaviors such as rule or reason repetition are indicators of the fact that the subject is stuck with a single hypothesis. Within the context of software development and unit testing, this is analogous to testing the code for similar scenarios and similar input/output combinations. Immediate rule announcement may also be an indicator of the lack of hypothesis testing strategies. A developer who makes immediate announcements during the interactive test, is most likely not to take into account the results of his/her previous unit test scenarios before deciding on the scope of his/her next test scenario. Eliminative/enumerative index ($Ind_{el/en}$) was introduced by Wason to evaluate the results of his rule discovery task in order to determine the proportion of the total number of instances that are incompatible with reasons to those that are compatible. It is desirable that eliminative/ enumerative index is greater that one, and the higher it is, the higher the tendency of the subject to refute his/her own hypotheses. The original elimination/enumeration index and its variations defined by us are listed in Table 3.3. Frequency of negative instances was also defined by Wason to interpret the results of this rule discovery experiment in addition to the elimination/enumeration index [1]. He had found high correlation between elimination/enumeration index values and negative instance frequencies. However, these two are not entirely the same. Therefore the metric list in Table 3.3 also includes Wason's negative instance frequency as well as the variations of both eliminative/enmerative index and negative instance frequency which have been defined by us.

Table 3.3. List of confirmation bias metrics (Metrics to Monitor Hypothesis Testing Procedure).

Rule/Reason Repetition			
Metric	**Explanation**	**Test Type**	**Data Type**
$avgF^{R}{}_{R}$	Average rule repetition frequency	Interactive	Continuous
$avgF^{Rs}{}_{R}$	Average reason repetition frequency	Interactive	Continuous
$inst_{P+C}$	Positive and compatible instances given after incorrect rule announcements	Interactive	Continuous
$inst_{P+IC}$	Positive and incompatible instances given after incorrect rule announcements	Interactive	Continuous
$inst_{N+C}$	Negative and compatible instances given after incorrect rule announcements	Interactive	Continuous
$inst_{N+IC}$	Negative and incompatible instances given after incorrect rule announcements	Interactive	Continuous
Immediate Rule Announcements			
Metric	**Explanation**	**Test Type**	**Data Type**
$avgF^{IR}$	Average immediate rule announcement frequency	Interactive	Continuous
$avgL^{IR}$	Average length of immediate rule announcement	Interactive	Continuous
Eliminative/Enumerative Behavior			
Metric	**Explanation**	**Test Type**	**Data Type**
$Ind_{el/en}$	Wason's Eliminative/enumerative index (average, minimum, maximum are also available)	Interactive	Continuous
Frequency of Negative Instances			
Metric	**Explanation**	**Test Type**	**Data Type**
F_{neg}	Frequency of negative instances (average, minimum, maximum are also available)	Interactive	Continuous
Subject's Method to Test Hypotheses			
R/T	Rules announced per unit time	Interactive	Continuous
Rs/T	Reasons announced per unit time	Interactive	Continuous
UR/T	Unique rules announced per unit time	Interactive	Continuous
URs/T	Unique reasons announced per unit time	Interactive	Continuous

3.1.3. Conducting Confirmation Bias Tests

In order to collect confirmation bias metrics in a controlled manner, we conducted confirmation bias tests under a predefined standard procedure. The following information is valid for both written and interactive tests:

- The environment where both tests were conducted was isolated from noise having adequate lighting.
- Both Turkish and English versions of the tests were prepared. Participants took Turkish version of the tests if their native language is Turkish. Participants from Canada took English versions of the tests.
- Participants were informed about the fact that this test shall not be used in performance evaluations and their identity shall be kept anonymous. The goal was not to exert pressure on participants which could affect participants' performance.
- Moreover, participants were told that there was no time constraint to complete the written test in order not exert time pressure.
- After the completion of both tests, participants were warned not to inform other software developers and testers in their company about the content of the tests.

Below we explain the standard procedures followed for both written and interactive tests.

3.1.3.1. Written Tests. In Wason's studies related to his Selection Task, participants were allowed to inspect real packs of cards, before the experimenter secretly selected four cards from the pack and placed then on a table so that only a single side of each card is visible. However, most recent studies in this field rely on the description of the cards, and pictorial representations of cards' facing sides either on pencil and paper or on a computer screen. These procedural differences have made very little differences in the obtained results [62]. Therefore, we used pen and pencil approach to apply the test which consists of variations of Wason's Selection Task as well as its original form. Written test consists of two booklets. The first booklet (i.e. *General Test*) included abstract, thematic-abstract and thematic questions, while the second booklet (i.e. *Software Test*) consisted of thematic questions with software development/testing theme. The following

standard procedure was followed while conducting written tests:

- Each group of participants, corresponding to each data set listed in Chapter 4, took written tests altogether in a meeting/seminar room.
- Before starting the tests, the participants were told to fill in the form where personal information such as gender, age, education and experience in software development and/or testing were asked.
- Firstly, *General Test* was given to participants so that they started the test simultaneously.
- In order to measure time:
  - **i.** We recorded each participant's completion time of the *General Test*, once he/she completed and submitted the test.
  - **ii.** Starting time for the *General Test* was same for all the participants.
  - **iii.** After a participant completed and submitted *General Test*, he/she was given *Software Test*. Starting time for this test was also recorded, once the subject started.
  - **iv.** Similar to *General Test*, completion time of *Software Test* was also recorded once the participant submitted the test he/she completed.

3.1.3.2. Interactive Tests.

- Each participant took interactive test separately and for each participant there was one experimenter to conduct the test.
- Participants were asked whether they give permission to record their voices during the interactive test. The goal of sound recording was to catch every detail about the way a participant thinks to discover the correct rule. However, voice recording was made only if the participant gave us the permission.
- Before the test, the original text in Wason's Rule discovery Task, was read to the participant:

  > You will be given three numbers which conform to a simple rule that I have in mind. This rule is concerned with a relation between any three numbers and not with their absolute magnitude, i.e. it is not a rule like all numbers above (or below) 50, etc.
  >
  > Your aim is to discover this rule by writing down sets of three numbers, together with reasons for your choice of them After you have written down

> each set, I shall tell you whether the numbers conform to the rule or not, and you can make a note of this outcome on the record sheet provided. There is no time limit but you should try to discover this rule by citing minimum set of numbers.
>
> Remember that your aim is not simply to find numbers which conform to the rule, but to discover the rule itself. when you feel highly confident that you have discovered it and *not before,* you are to write it down and tell me what it is. Have you any questions?

3.2. Analysis of Factors Affecting Confirmation Bias

In this section we explain methods and techniques we employed during the analysis of factors affecting confirmation bias.

3.2.1. Test Severity Calculation

In order to calculate Test Severity (i.e. severity of each instance) for each participant, we used the method proposed by Poletiek [48]. The details about this method were given in Section 2.1.4.3, where we explained Popper's definition about the test severity. In Wason's Selection Task, test severity corresponds to the severity of a given instance. Poletiek's method was a way to evaluate severity of each instance given by a participant during the experiment she conducted. Her experiment was similar to Wason's Rule discovery Task. However the initial instance given to the participant was "2-7-6" and she wanted the participant to the discover the rule "even-uneven-even numbers". We replicated her method for our interactive test which is based on Wason's Selection Task. We formed a set of plausible alternative rules which are listed in Table 3.4, 3.5, 3.6 and 3.7. This list is a collection of alternative hypotheses which were gathered during interactive tests we conducted to active/inactive software developers/testers from various companies/institutions in two different countries. Details about data sets can be seen in Chapter 4.

Table 3.4. Set of plausible alternative hypothesis to be used in test severity calculations: Part I.

	Notation	**Explanation**
1	$a + b = c$	Sum of the 1^{st} and 2^{nd} number is equal to the 3^{rd} number
2	$a < b < c$ and $a + b = c$	Numbers are in ascending order and sum of the 1^{st} and 2^{nd} number is equal to the 3^{rd} number
3	$a < b < c$ and $b - a = c - b$	Numbers are in ascending order with constant increments
4	$c/(b - a) = k$ and $k \in Z$	3^{rd} number is divisible by the difference between the 2^{nd} and 1^{st} number
5	$b - a = c - b = 2$	Numbers are in ascending order with constant increments of 2
6	$a < b < c$ and $a + b = c$ and $b = 2 * a$	Numbers are in ascending order and 2^{nd} number is twice the 1^{st} number and 3^{rd} number is three times the 1^{st} number
7	$c > a$ and $c > b$	3^{rd} number is greater than 1^{st} and 2^{nd} numbers
8	$c \geq a$ and $c \geq b$	3^{rd} number is greater than or equal to 1^{st} and 2^{nd} numbers
9	$a = 2 * n$ and $b = 2 * (n + 1)$ and $c = 2 * (n + 2)$ and $n \in Z^+$	Even positive integers are in ascending order with constant increments of 2
10	$a + c = 2 * b$	Sum of the 1^{st} and 3^{rd} numbers is equal to twice the 2^{nd} number
11	$(a + b + c)/3 = k$ and $k \in Z$	Sum of three numbers is divisible by 3
12	$a < b < c$ and $(a + b + c)/3 = k$ and $a, b, c, k \in Z$	Integers in ascending order such that sum of three numbers is divisible by 3
13	$b = a + x$ and $c = b + y$, $y \geq x$, $x, y \in Z^+$	Ascending numbers with integer increments and difference between 3^{rd} and 2^{nd} number is greater than or equal to the difference between 2^{nd} and 1^{st} number

Table 3.5. Set of plausible alternative hypothesis to be used in test severity calculations: Part II.

	Notation	**Explanation**
14	$a, b, c \in Z$ and $b > a$	Three integers such that 2^{nd} integer is greater than 1^{st} integer
15	(a, b, c) and $c > a$	Three numbers such that 3^{rd} number is greater than the 1^{st}
16	$c - a = k$ and $kmod2 = 2$	Difference between the 3^{rd} and 1^{st} number is even
17	$GCD(a, b, c) = 2$	Greatest Common Divisor of three numbers is 2
18	$(a + 1, b + 1, c + 1)$ are all prime	1 greater than each integer is a prime number
19	(a, b, c) are all real numbers	Three real numbers
20	$a < b < c$ and $a, b, c \in R^+$	Positive real numbers in ascending order
21	$a < b < c$ and $a, b, c \in Z^+$	Positive integers in ascending order
22	$(a + b + c)mod2 = 0$ and $a, b, c \in Z$	Three integers whose sum is divisible by 2
23	$(a + b + c)mod2$ and $a, b, c \in Z^+$	Three positive integers whose sum is divisible by 2
24	$a, b, c \in Z$ and $a \neq b \neq c$	Three unique integers
25	$a, b, c \in Z$	Three integers
26	$a * b - a = c$	3^{rd} number is obtained when 1^{st} is subtracted from product of the 1^{st} and 2^{nd} number
27	$a * b - a = c$ and $b = a * a$	3^{rd} number is obtained when 1^{st} is subtracted from product of the 1^{st} and 2^{nd} number and square of 1^{st} number is 2^{nd} number
28	$a * b - a = c$ and $b = a * a$	3^{rd} number is product of the 1^{st} number with the average of the 1^{st} and 2^{nd} number
29	$a = 2 * n$ and $b = 2 * (n + 1)$ and $c = 2 * (n + 2)$ and $n \in Z$	Even integers are in ascending order with constant increments of 2

Table 3.6. Set of plausible alternative hypothesis to be used in test severity calculations: Part III.

	Notation	**Explanation**
30	$a + b \geq c$	Sum of the 1^{st} and 2^{nd} number is greater than or equal to the 3^{rd} number
31	$a, b, c \in R$ and $b > a$	Three real numbers such that 2^{nd} number is greater than the 1^{st} number
32	$a, b, c \in Z$ and $a < b < c$ and $(c/2) + 1 = b$	Three integers in ascending order such that 1 greater than half of the 3^{rd} number is equal to the 2^{nd} number
33	$a, b, c \in Z$ and $a < b < c$ and $(b = a + 2$	Three integers in ascending order such that difference between the 2^{nd} and 1^{st} number is equal to 2
34	$a, b, c \in Z$ and $a < b < c$ and $b^2 = 2 * (a + c)$	Three integers in ascending order such that twice the sum of 1^{st} and 3^{rd} number is equal to the square of the 2^{nd} number
35	$a, b, c \in Z$ and $b^2 = 2 * (a + c)$	Three integers such that twice the sum of 1^{st} and 3^{rd} number is equal to the square of the 2^{nd} number
36	$a, b, c \in Z$ and $a < b < c$ and $(a * c)/3 = b$	Integers in ascending order such that product of 1^{st} and 3^{rd} number is equal to three times the 2^{nd} number
37	either $a < b < c$ or $a > b > c$ and $a, b, c \in Z^+$ and $\|b - a\| = \|c - b\| = 2$ and a, b, c are even	Consecutive positive even integers either in ascending or in descending order.
38	$a < b < c$ and $a, b, c \in Z^+$ and $\|b - a\| = \|c - b\| = 2$ and a, b, c are even	Consecutive positive even integers either in ascending order.
39	either $a < b < c$ or $a > b > c$ and $a, b, c \in Z$ and $\|b - a\| = \|c - b\| = 2$ order and a, b, c are even	Consecutive even integers either in ascending or in descending

Table 3.7. Set of plausible alternative hypothesis to be used in test severity calculations: Part IV.

	Notation	**Explanation**
40	$a < b < c$ and $a, b, c \in Z$ and $\|b - a\| = \|c - b\| = 2$ and a, b, c are even	Consecutive even integers either in ascending order.
41	either $a < b < c$ or $a > b > c$ and $a, b, c \in Z$ and $\|b - a\| = \|c - b\| = 2$	Integers either in ascending or in descending with increments /decrements of 2.
42	$a < b < c$ and $a, b, c \in Z$ and $GCD(a, b, c) > 1$	Integers in ascending order such that their Greatest Common Factor (GCF) is greater than 1.
43	$a, b, c \in Z$ and $GCD(a, b, c) > 1$	Integers with Greatest Common Factor greater than 1.
44	$a < b < c$ and $(b - a) = (c - b) = 2 * m$ and $a, b, c \in Z$ and $m \in Z^+$	Integers ascending with even integer valued increments.

3.2.2. Formation of Test Severity Graphs Using Vincent Curves

Vincent curves [63] represent performance of subjects towards a criterion, which is not defined by fixed number of trials. In his Selection Task, Wason used Vincent curves to compare mean number of instances incompatible with reasons up to the first rule announcement for two groups. One of the two groups was the group of participants who announce the correct rule at their first trial, whereas the other group consisted of participants who could not find the correct rule at their first trial.

We use Vincent Curves to be able to compare hypothesis testing strategies of different groups of participants. During interactive tests, total number of instances given before discovery of the correct rule varies from one subject to another. Hence, Vincent curves can be used to visualize the change in test severity of a group of subjects until the correct rule is discovered. Although, there are variants of Vincent curves, we use the original method proposed by Vincent to obtain a composite curve. This method can be explained as follows:

- Total number of instances given by each subject in the group is divided into N equal fractions.
- Within each fraction, we calculate the average of test severities of the instances that fall into that fraction. This calculation is done for each subject in the group.
- For N equal fractions, $N + 1$ data points are obtained per subject. The average of the i^{th} data point of all subjects gives the i^{th} data point for the group of subjects, where $i = 1, 2, , N + 1$.

We have selected total number of fractions N to be equal to the minimum number of instances given within the group before discovery of the correct rule. For number of instances which are not divisible by N, we used Vincent's original procedure. For instance, the division of 22 instances given by each subject among 5 fractions would be 5, 5, 4, 4, 4. In other words, 2 additional instances are distributed one by one, starting from the first fraction.

3.3. Defect Prediction

3.3.1. Naive Bayes Algorithm

Naive Bayes is based on Bayes Rule [64]. Using Bayes Rule, the probability that an observation $\vec{x}$, which is a d-dimensional vector where $d > 1$ belongs to class C_i, $P(C_i|\vec{x})$ can be formulated as follows:

$$P(C_i|\vec{x}) = \frac{P(\vec{x}|C_i)P(C_i)}{P(\vec{x})} \tag{3.1}$$

In Equation 3.1, $P(C_i)$ is called the *prior probability* of class C_i. $P(\vec{x}|C_i)$ is called the *class likelihood* that observation $\vec{x}$ comes from the distribution which generates class C_i. $P(\vec{x})$ is called the *evidence* and it is the marginal probability that observation $\vec{x}$ is seen. For all classes C_i, $P(\vec{x})$ is identical:

$$P(C_i|\vec{x}) = \frac{P(\vec{x}|C_i)P(C_i)}{\sum_{k=1}^{K} P(\vec{x}|C_k)P(C_k)} \tag{3.2}$$

Since evidence $P(\vec{x})$ is identical for all classes C_i, we can define the *discriminant function* as follows:

$$g_i(\vec{x}) = P(\vec{x}|C_i)P(C_i) \tag{3.3}$$

Discriminant function $g_i(\vec{x})$ can also be defined by taking logarithm of the product $P(\vec{x}|C_i)P(C_i)$.

$$g_i(\vec{x}) = log(P(\vec{x}|C_i)) + log(P(C_i)) \tag{3.4}$$

In a classification problem, for each observation $\vec{x}$ in the sample we calculate the posterior probability $P(C_i|\vec{x})$ for all classes $C_1...C_n$. We assign each observation $\vec{x}$ to the

class for which maximum posterior probability is obtained. Finding maximum posterior probability is equivalent to obtaining the maximum value for the discriminant function $g_i(\vec{x})$. In order to calculate value of discriminant function $g_i(\vec{x})$ for each observation $\vec{x}$, we need to be able to calculate prior probability $P(C_i)$ and the likelihood term $P(\vec{x}|C_i)$. Prior probability of each class $P(C_i)$ can be obtained from the sample by counting. On the other hand, we need to choose a suitable distribution for class likelihood $P(\vec{x}|C_i)$. In Naïve Bayes, it is assumed that $P(\vec{x}|C_i)$ are Gaussian:

$$P(\vec{x}|C_i) = \frac{1}{(2\pi)^{d/2}(\Sigma_i)^{1/2}} exp[-\frac{1}{2}(\vec{x} - \vec{\mu_i})^T(\Sigma_i)^{-1}(\vec{x} - \vec{\mu_i})] \tag{3.5}$$

As a result of inserting $log(P(\vec{x}|C_i))$ in equation 3.3 by the Gaussian distribution formula in equation 3.4, discriminant function $g_i(\vec{x})$ becomes:

$$g_i(\vec{x}) = -\frac{d}{2}log2\pi - \frac{1}{2}log|\Sigma_i| - \frac{1}{2}(\vec{x} - \vec{\mu_i})^T\vec{\Sigma_i}^{-1}(\vec{x} - \vec{\mu_i}) + logP(C_i) \tag{3.6}$$

We can eliminate the term $-\frac{d}{2}log2\pi$ since it is constant in discriminant functions of all classes C_i, $i = 1, ..., n$. According to the assumptions of Naïve Bayes, each class has a common covariance matrix Σ where off-diagonal entries are equal to 0. Hence, we can reformulate discriminant function $g_i(\vec{x})$ as follows:

$$g_i(\vec{x}) = -\frac{1}{2}\sum_{j=1}^{d}\frac{(x_j - m_{ij})^2}{s_j^2} + logP(C_i) \tag{3.7}$$

In the above equation, x_j is the jth dimension of vector $\vec{x}$, m_{ij} is the jth dimension of the mean vector for class C_i and s_j is the jth component of the standard deviation vector which is common for all classes.

We use Naïve Bayes to build defect prediction models for Experiment III. In our defect prediction problem domain, each vector $\vec{x}$ corresponds to attributes of a single source code file. These attributes can be static code metrics, churn metrics of the file

as well as the confirmation bias metrics of the group of developers who created/updated that file. We have two classes which are "defected" class C_1 and "non-defected" class C_2. Defective files belong to class C_1, whereas defect-free files belong to class C_2.

3.3.2. Performance Measurement

In order to evaluate the performance of the defect predictors built by using different metric set combinations, we used the well-known performance measures which are probability of detection, false-alarm rate and balance respectively [5].

- *Probability of Detection (pd)*: Pd measures how good a predictors is in finding defective modules, where modules can be files, methods or packages depending on the granularity level. In the ideal case, we expect a predictor to catch all defective modules (i.e. $pd = 1$).
- *Probability of False Alarms (pf)*: Pf measures false alarm rates, when predictor classifies defect-free modules as defective. In the ideal case, we expect a predictor to classify none of the defect-free modules as defective (i.e. $pf = 0$).
- *Balance (bal)*: In practice, the ideal case where a defect predictor has high probability of detecting defective modules (i.e. high pd) and low probability of false alarm (i.e. $pf = 0$) is very rare. Therefore, we try to balance between pd and pf values. The notion of balance is formulized to be the Euclidean distance from the sweet spot ($pd = 1$ and $pf = 0$) normalized by the maximum possible distance to this spot. It is desirable that predictor performance is close to the sweet spot as much as possible (i.e. high balance values are desirable).

$$bal = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{3.8}$$

Pd and pf values are calculated using *Confusion Matrix* that is given in Table 3.8. In the confusion matrix, TP is the number of correctly classified defective modules, FP is the number of non defective modules that are classified to be defective, FN is the number of defective modules that are classified to be non-defective and finally TN is the number of correctly classified non-defective modules. Formulations for pd and pf in terms of confusion matrix values is given below:

Table 3.8. Confusion matrix *TP:True Positives, FN:False Negatives, FP:False Positives, TN:True Negatives.*

	Predicted	
Actual Case	Defected	Not-defected
Defected	TP	FN
Not-defected	FP	TN

3.4. Missing Data Problem in Defect Prediction

In the literature, there are various methods to impute missing data. Imputations are means or draws from a predictive distribution of missing values. There are mainly two types of imputation methods, which are single imputation and multiple imputation respectively.

3.4.1. Single Imputation Methods

Single imputations can be used to impute one single value for each missing item. Below, we list some single imputation methods:

- *Mean imputation:* Assume that we have a partially complete data set Y, where $y_{i,j}$ stands for a missing value of the jth attribute for the ith item. Then, we can replace each missing value $y_{i,j}$ by the stratified mean can be formulated as follows:

$$\bar{y}_{st} \equiv \bar{y}_w = \frac{1}{n}\sum_{i=1}^{n} w_i y_i \tag{3.9}$$

In Equation 3.9, n is the total number of samples and w_i is the sampling weight attached to item i.

$$w_i = \frac{n\pi_i^{-1}}{\sum_{k=1}^{n}\pi_k^{-1}} \tag{3.10}$$

and π_i is the selection probability of item i which implies that item i represents π_i^{-1}

units in the sample population. Hence, in Equation 3.10 is scaled to the sum of the sample size $n = \sum_{k=1}^{n} \pi_k^{-1}$. However, y_w can only be calculated using complete data in the sample, since we have no idea about the missing data. As a result Equation 3.10 becomes:

$$w_i = \frac{r(\pi_i \phi_i)^{-1}}{\sum_{k=1}^{r}(\pi_k \phi_k)^{-1}} \tag{3.11}$$

In the above equation, r is the total number of complete data, ϕ_i is the probability that the selected item i is *not* missing. Therefore $Pr(selection\ and\ not\ missing) = Pr(selection)*Pr(not\ missing\ |selection) = \pi_i \phi_i$ and $\phi_i = r/n$.

- *Regression imputation:* Regression imputation can be applied to missing data patterns where Y_1,Y_2,...,Y_{k-1} are fully observed and Y_k is observed for the first r observations and missing for the last $n-r$ observations. Using Y_1,Y_2,...,Y_{k-1} as independent variables and the first r rows of Y_k as the response variable, we can construct a regression equation.

$$\hat{y_{ik}} = \tilde{\beta}_{k0.12...k-1} + \sum_{j=1}^{k-1} \tilde{\beta}_{kj.12...k-1} y_{ij} \tag{3.12}$$

Missing values can be imputed by the results obtained from the resulting regression model.

- *Stochastic regression imputation:* Different from *regression imputation* where a conditional mean is imputed, *stochastic regression* imputation is used to impute a conditional draw.

$$\hat{y_{ik}} = \tilde{\beta}_{k0.12...k-1} + \sum_{j=1}^{k-1} \tilde{\beta}_{kj.12...k-1} y_{ij} + z_{i,k} \tag{3.13}$$

Un the above equation $z_{i,k}$ is a random normal deviate with mean 0 and residual variance obtained from the regression of Y_k on $Y_1, Y_2, ..., Y_{k-1}$. In this way, distortions due to imputing mean of the predictive distributions are improved.

- *Hot deck imputation:* Hot deck imputation involves substituting individual values which are drawn from items similar to the item whose missing value(s) are to be imputed. In the literature there are various hot deck imputation methods.

- *Substitution:* Missing values of items are replaced by those of the items which are not included in the sample. For instance assume that during our analysis in this thesis, confirmation bias metrics of a developer are missing since that developer no longer works at the company. Hence, we substitute confirmation bias metrics of a developer who is not included in our analysis. Such a developer might be excluded form the analysis since he/she is a new comer to the project group so that she does not have any commit information in version management system regarding the project developed by the group. During substitution, there must always be a resistance to treat the imputed data as complete, because substituted values may differ significantly from the missing values.
- *Cold deck imputation:* Cold deck imputation replaces a missing value of an item by a constant value from an external source. For instance missing values in a survey result may be imputed by using previous results of the same survey that was conducted to the same person. However, similar to substitution method, one must be cautious while using this method. In our case, since there are no past results of confirmation bias tests, cold deck imputation is not a suitable method for us.

3.4.2. Multiple Imputation Methods

Using multiple imputation method, one can impute more than one value for each missing item which makes it possible to assess imputation uncertainty. For each replacement of the missing values, an imputed data set is obtained. Each data set obtained is analyzed using standard complete data methods. Multiple imputation reflects sampling variability under one imputation model or uncertainty about the correct model used for imputation.

3.4.3. Expectation Maximization Methods for Imputation

Expectation-Maximization algorithm formalizes a relatively ad hoc idea for handling missing data [65]. It consists of iteratively replacing missing values by estimated values which is followed by re-estimating the parameters. E-step (Expectation step) finds the conditional expectation of the missing data given the observed data and current estimated parameters and then substitutes these expectations for the missing data. On the

other hand, M-step (Maximization step) performs maximum likelihood estimation of the parameters θ just as if there were no missing data. Unlike ad-hoc imputation methods, EM does not necessarily substitute the missing values themselves. Instead of the missing values, functions of the missing values appearing in the log likelihood of the complete data $\ell(\theta|Y) = lnL(\theta|Y)$ are estimated.

3.4.4. Roweis' EM Algorithm for PCA

In this thesis, we use Roweis' Expectation-Maximization (EM) Algorithm for Principal Component Analysis (PCA) [46], in order to deal with the missing data problem. Our goal is not to perform PCA factorization. However, the reason of our choice is the ability of this algorithm to permit the computation of eigenvalues and eigenvectors while working with many data with high dimensions in the presence of *missing*. Roweis' method, had also inspired Bell, Koren and Volinsky [66] to develop a matrix factorization algorithm to improve the accuracy of large recommender systems such as Netflix Cinematch [67, 68].

The goal of PCA is to find a mapping from the data Y in the original d-dimensional space to a new k-dimensional space where $k < d$ such that there is minimum loss of information [64].

$$X = w^T Y \tag{3.14}$$

The above equation can be reformulated as $Y = CX$, where $C = (w^T)^{-1}$, which in turn can be reformulated as $X = C^{-1}Y = C^{-1}IY$, where $I = (C^T)^{-1}C^T$ is the identity matrix. Therefore, the following formulation holds for X:

$$X = (C^T C)^{-1} C^T Y \tag{3.15}$$

Equation 3.15 forms the E-step of Roweis' EM Algorithm. In the above equation, Y is a pxn matrix of the original data, X is a kxn matrix which shall be the output of PCA, C is a kxp matrix that spans the space of the first k principal components of Y. Similarly,

M-step of the algorithm can be formulated as in Equation 3.16

$$C = YX^T(XX^T)^{-1} \tag{3.16}$$

During the e-step of the EM algorithm, for any data entry y in the data matrix Y with some of its coordinates missing, a unique pair x^* and y^* can be calculated such that $||Cx^* - y^*||$ is minimized. In other words, missing values can be imputed by solving the least squares problem for $||Cx^* - y^*|| = 0$. Pseudo code of Roweis' EM Algorithm is given in Figure 3.2.

ImputeMissingData(Y)
% Initially set matrix C randomly
for all *iteration* in $[1:1:MAX(iter)]$ **do**
  *%**E-Step of EM Algorithm***
  for all y in Y **do**
    for all i in [1:1:size(y,1)] **do**
      *% **Missing Data Imputation is done within E-step***
      if $isMissing(y(i)) == TRUE$ **then**
        % Find Least Squares Solution to Cx = y by QR decomposition
        if $isSparseMatrix(C) == TRUE$ **then**
          $[Q, R] = qrDecompose(C)$
        else
          $R = upperTriangle(qrDecompose(C))$
        end if
        $x = R^{-1}(R^T)^{-1}C^T y$; $r = y - Cx$
        $e = R^{-1}(R^T)^{-1}C^T r$; $x = x + e$
      end if
      if $isMissing(y(i)) == FALSE$ **then**
        $x = (C^T C)^{-1}C^T y$
      end if
      $X(i,:) = x$; % Assign x to i^{th} row of matrix X.
    end for
  end for
  *% **M-Step of EM Algorithm***
  $C = YX^T(XX^T)^{-1}$
end for$Y_{imputed} = CX$
return$(Y_{imputed})$

Figure 3.2. Pseudocode for the implementation of Roweis' EM Algorithm

4. DATA SET

4.1. Participants of Confirmation Bias Tests

In this thesis, our goal was to perform our experiments in real software development settings. Hence, we conducted both written and interactive tests to software development professionals in order to collect confirmation bias metrics. In Table 4.1, except for Group 8 confirmation bias metrics were collected from participants working in large scale software development companies or SMEs (Small Medium Enterprizes). Out of 8 groups, 4 groups consisted of software engineers working in large scale companies. One of these companies is located in Canada, while the remaining 3 companies are located in Turkey. Unlike other participant groups, Group 8 consists of computer engineering graduate students at Boğaziçi University. Among these computer engineering graduate students, 14 participants have average software development experience of 2.51 years, while six of them are still active and they are developing embedded software for RoboCup, which is an international robotics competition founded in 1993.

In addition to developers testers are included in the groups in large scale companies, while groups from which confirmation bias metrics were collected in SMEs consist of only developers. On the other hand, only Group 6 contains analysts and architects as well as developers and testers. Except for Group 8 members, participants have only bachelor degrees in computer science, mathematics or other engineering fields. In the large scale software development company where Group 6 members work, analysts are responsible from the preparation of both requirement analysis documents and test scenarios for the testing phase. Gender distribution and average age values are given in Table 4.2. The highest percentage of females is in Group 6, where 85.71% of analysts are female. The development methodology employed by the company is scrum which is an agile development approach. All SMEs to which Groups 3, 4, and 5 belong also employ agile software development methodology, while the large Canadian company develops Data Management System (DBMS) software using Test Driven Development (TDD) methodology. Different than from all these groups, within Group 1 there are 3 separate project groups where each project group employs a different software methodology. Details about project groups

within Group 1 are given in Table 4.3. Project Groups 1 and 2 were temporary pilot project groups developing a pilot software in order to decide whether TDD and TSP/PSP methodologies would be beneficial during the production of company's software products. Moreover, some group members were also working in other projects at the same time.

Table 4.1. Details about groups from which confirmation bias metrics were collected.

Group #	**Developers**	**Testers**	**Analysts**	**Architects**	**Total**	**Domain**	**Company Size**	**Country**
*Group*1	22	14	0	0	36	telecommunication	large scale	Turkey
*Group*2	24	10	0	0	34	DBMS development	large scale	Canada
*Group*3	12	0	0	0	12	finance	SME	Turkey
*Group*4	8	0	0	0	8	CRM	SME	Turkey
*Group*5	6	0	0	0	6	mobile solutions	SME	Turkey
*Group*6	18	0	35	10	63	finance	large scale	Turkey
*Group*7	6	0	0	0	6	ERP	large scale	Turkey
*Group*8[1]	14[2]	0	0	0	29	various[3]	None[3]	Turkey

[1] Members of Group9 are Computer Engineering graduate students at Boğaziçi University.

[2] 14 of Computer Engineering Graduate Students have at least 2 years of development experience at large scale companies and SMEs.

[3] 8 of Computer Engineering Graduate Students have at least 2 years of development experience at large scale companies and SMEs specialized in telecommunication/CRM domain. 6 of them are developing embedded software for AI/robotics research.

Table 4.2. Gender distribution and average age values of Groups 1-8.

Group #	**Female**	**Male**	**Age (years)**
*Group*1	10	26	29.06
*Group*2	8	26	34.21
*Group*3	1	11	27.17
*Group*4	1	7	26.93
*Group*5	0	6	24.00
*Group*6	30	33	31.72
*Group*7	30	33	30.83
*Group*8	7	21	27.96

Table 4.3. Details about project groups within Group 1.

Project Group #	**Developer**	**Tester**	**Total**	**Development Methodology**
*ProjectGroup*1	12	14	26	Waterfall
*ProjectGroup*2	4	0	4	TDD
*ProjectGroup*3	8	0	8	TSP/PSP

4.2. Collection of Static Code Metrics

Although confirmation bias related information and metrics are adequate for Experiments I and II, for Experiment III static code metrics of the files created and/or updated by developer groups are required. Hence, we used Prest tool [69] to extract static code metrics at file granularity level. The list of the static code metrics Prest can extract from source code files is given in Table 4.4. The reason why file granularity level is preferred for static code metrics extraction rather than method, class or package granularity levels can be explained as follows: Firstly, for defect prediction in Experiments II and III we were able to obtain defect information at file granularity level. Secondly, using commit logs it was only possible to match each source code file and group of developers who created and/or updated that file. In other words, mapping of developer(s) to methods was not possible. Although file level granularity is coarser than method level granularity, it is still a fine granularity for experiments compared to package level granularity.

Table 4.4. List of static code metrics used in Experiment III.

Attribute	**Description**
McCabe Metrics	
Cyclomatic Complexity $v(G)$	number of linearly independent paths
Cyclomatic Density $vd(G)$	the ratio of the files's cyclomatic complexity to its length
Decision Density $dd(G)$	condition/decision
Essential Complexity $ev(G)$	the degree to which a file contains unstructured constructs
Essential Density $ed(G)$	$(ev(G) - 1)/(v(G) - 1)$
Maintenance Severity	$ev(G)/v(G)$
Lines of Code Metrics	
Unique Operands Count	n_1
Unique Operators Count	n_2
Total Operands Count	N_1
Total Operators Count	N_2
Lines of Code (LOC)	source lines of code
Branch Count	number of branches
Conditional Count	number of conditionals
Decision Count	number of decision points
Halstead Metrics	
Level (L)	$(2/n_1) * (n_2/N_2)$
Difficulty (D)	$1/L$
Length (N)	$N_1 + N_2$
Volume (V)	$N * log(n)$
Programming Effort (E)	$D * V$
Programming Time (T)	$E/18$

Table 4.5. List of churn metrics used in Experiment III.

Attribute	**Description**
commits	number of commits made for a file
committers	number of developers who committed a file
commitsLast	number of commits made for a file since last release
committersLast	number of developers who committed a file since last release
rmlLast	number of removed lines from a file since last release
alLast	number of added lines to a file since last release
rml	number of removed lines from a file
al	number of added lines to a file
topDevPercent	percentage of top developers who committed a file

4.3. Collection of Commit Log Data

We use commit log data to refer to the commit logs obtained from version management systems. We needed commit log data for two reasons. First reason was to be able to match each source code file with group of developers who created and/or updated that file. This step was crucial for conducting Experiment II, since analyzing the effects of confirmation bias metrics on software developers requires the knowledge about which developer groups created and/or updated which file. In this way, it was possible to calculate percentage of defected files for each developer group so that a regression model could be constructed using confirmation bias metrics as independent variables and defected file percentage as the response variable. Based on the obtained results, one of the goals of Experiment II was to investigate the effect of confirmation bias on software developers. The second reason about why information in commit logs was required is the fact that to conduct Experiments III and IV we had to match confirmation bias metrics with source code files. Hence, we needed to be able to map each file to a group of developers. Moreover, we also needed to evaluate churn metrics for Experiment II and this is possible only using information contained in commit logs. List of the churn metrics extracted from commit log data is given in Table 4.5.

5. EXPERIMENTS AND RESULTS

5.1. Experiment I: Analysis of Factors Affecting Confirmation Bias

In this experiment, we aim to gain insight about main factors which affect confirmation bias in software development context. Details about the data set, analysis techniques used as well as our findings are given in the following sections.

5.1.1. Data

In the first part of this experiment, we used data which belong to Group 1, Group 2, Group 7 and Group 8. However, in order to eliminate confounding factors and analyze a single factor during each analysis, we use only data corresponding to a specific group of members of Groups 1,2,7 and 8. For this reason, in the rest of this section we designate the subgroups obtained after the elimination of confounding factors as $Group1^*$, $Group2^*$, $Group7^*$ and $Group8^*$ respectively. Potential confounding factors and how we handled them can be explained as follows:

- *Role of the Software Professional*: Each subgroup consists of *only* developers except for $Group8^*$. However, each member of $Group8^*$ has software development experience of at least 2 years, while 6 members are still developing research oriented embedded robotics software.
- *Age*: Age distribution within each resulting subgroup is similar and maximum age value is 32.
- *Gender*: Ratio of female developers to male developers is similar for all subgroups.
- *Experience*: Members in each subgroup has at least 2 years of experience in software development industry. Moreover, average years of experience within each subgroup is identical.
- *Development Language*: Developers in subgroups $Group1^*$ and $Group7^*$ use Java as development language, whereas the language used by developers in $Group2^*$ is C^{++}. However, both languages are object oriented. On the other hand, members of Group 8^* are experienced in both Java and C^{++} programming languages.

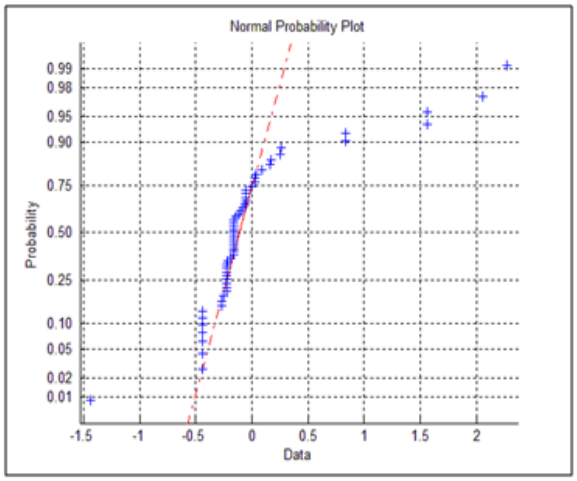


Figure 5.1. Normal probability plot for the residuals of the confirmation bias metric F_R^{Rs}.

- *Development Domain*: Members of both $Group1^*$ and $Group7^*$ develop customer services software package for GSM operator clients and bank customers respectively. Members of $Group8^*$ have industrial software development experience within similar domains. On the other hand, $Group2^*$ members are developing database management system (DBMS).

5.1.2. Design

We perform pairwise comparison of confirmation bias metric values for these four subgroups using χ^2 test for independence. We did not prefer to use ANOVA or t-test, since residuals for confirmation metric values are not normally distributed [70]. Figure 5.1 and Figure 5.2 show normal probability plot and histogram of residuals for the confirmation bias metrics F_R^{Rs}. As it can be seen from Figure 5.1 and Figure 5.2, residuals for the values of the confirmation bias metric F_R^{Rs} are not normally distributed.

Another alternative for pairwise metric value comparisons would be the non-parametric method Mann Whitney U test. However, there are too many identical values (i.e. ties) in confirmation bias metrics with *continuous* values, which would lead to unreliable results [71]. Moreover, as it can be seen in Table 3.2 and 3.3 some metrics take *discrete* or *categorical* values. Table 5.1 shows how we divided the metric $avgInd_{el/en}$ into categories

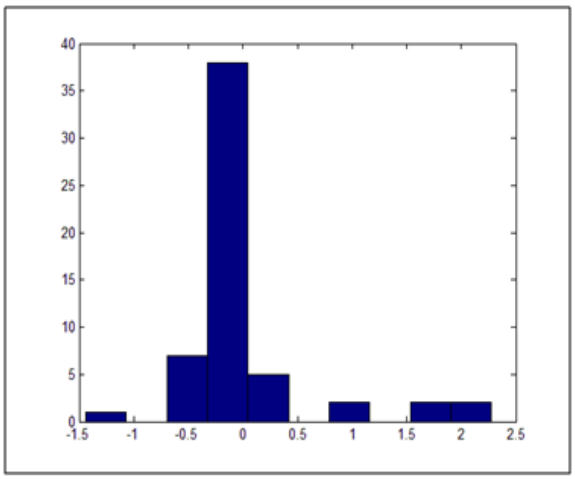


Figure 5.2. Histogram for the residuals of the confirmation bias metric F_R^{Rs}.

Table 5.1. Dividing continuous values of $avgInd_{el/en}$ into categories for χ^2 test.

	$x \in [0,1)$	$x \geq 1$
Group 1^*	9	10
Group 2^*	6	14
Group 7^*	11	7
Group 8^*	6	9

for χ^2 test.

In addition pairwise statistical comparisons of confirmation bias metrics values for Groups 1^*, 2^*, 7^* and 8^*, we compare overall hypothesis testing strategies of these subgroups during interactive test in a pairwise manner. Moreover, as an outcome of the written test we compare distribution of *Falsifiers*, *Verifiers* and *Matchers* using Reich and Ruth's categorization method [51].

5.1.3. Results and Discussions

When we look at χ^2 test results in Table 5.2, 5.5 and 5.6 we can conclude that Group 2^* members find the correct rule during interactive test with significantly less number of trials. This implies that unless they are really sure, Group 2^* members avoid making rule announcements. Moreover, especially compared to Group 7^* members, Group 2^* members exhibit a more eliminative behavior during interactive tests as well as employing an ideal

hypotheses testing strategy. As shown in Figure 5.9 according to the overall hypothesis testing strategy profile of Group 2* members consist of starting with less severe instances and progressively increasing severity of the instances given. There is a decrease in the severity of the instance given at the last bin of trials, however during interactive test once the subject is sure about the rule to be discovered, she/he can can decrease the severity of the instances given just before the rule announcement. Moreover, as it can be seen in Figure 5.4 and 5.10 there are more *Falsifiers* and less *Verifiers* and *Matchers* among Group 2* members compared to members of the Groups 1* and 7*. *Country* seems to be a factor which can explain why Group 2* members outperform members of the Groups 1* and 7*, since company Group 2* members work is in Canada, while Group 1* and 7* members work at software development companies located in Turkey. However, Group 8* members are also in Turkey as well as having worked for companies in Turkey and Group 8* members also outperform both Group 1* and 7*. As it can be seen in Figures 5.7 and 5.13, overall hypothesis testing strategy followed by members of Group 8* is much closer to the ideal hypothesis testing strategy defined by Poletiek [2]. In addition, majority of Group 8* members are Falsifiers as Figures 5.8 and 5.14.

Under the light of these obtained results, we state that hypotheses testing and logical reasoning skills are two factors that affect confirmation bias levels of developers. 50 % of the developers in Group 2* hold PhD degrees from Computer Science, Mathematics or related fields. On the other hand, members of Group 8* are Computer Engineering PhD students. We do not claim that PhD is required to eliminate confirmation bias. However, since hypothesis testing and logical skills are crucial in scientific research, this makes it obligatory for graduate students to obtain these skills fulfill the requirements of a PhD study. On the other hand, one does not need to have a PhD degree to acquire these skills. Training provided to software professionals can help them to eliminate negative effects of confirmation bias.

5.2. Experiment II: Analysis of the Effects of Confirmation Bias on Software Developers and Testers

This experiment consists of two parts. In the first part, effect of confirmation bias on software developer performance is analyzed, while in the second part of the experiment

Table 5.2. Statistical comparison of confirmation bias metric values for two developer subgroups which belong to Group 1 and Group 2 respectively.

Confirmation Bias Metrics	**Result**	**Critical Value**	**Explanation**
$avgF_R^R$	$\chi^2(1, n = 38) = 0.1101$	2.71 ($p < 0.1$)	No significant difference
$avgF^{IR}$			Expected frequency in at least one cell is less than 5
$avgL^{IR}$			Expected frequency in at least one cell is less than 5
$minInd_{el/en}$	$\chi^2(1, n = 39) = 1.2927$	2.71 ($p < 0.1$)	No significant difference
$maxInd_{el/en}$	$\chi^2(1, n = 38) = 1.2166$	2.71 ($p < 0.1$)	No significant difference
$avgInd_{el/en}$	$\chi^2(1, n = 31) = 1.2418$	2.71 ($p < 0.1$)	No significant difference
$Ind_{el/en}$	$\chi^2(1, n = 33) = 0.2026$	2.71 ($p < 0.1$)	No significant difference
N_A	$\chi^2(1, n = 38) = 4.496$	2.71 ($p < 0.1$)	$Group2^*$ outperforms $Group1^*$
IND_{ABORT}			Expected frequency in at least one cell is less than 5
S_{Th}	$\chi^2(1, n = 38) = 3.3135$	2.71 ($p < 0.1$)	$Group2^*$ outperforms $Group1^*$
S_{ABS}	$\chi^2(1, n = 38) = 0.0653$	2.71 ($p < 0.1$)	No significant difference
$S_{precautional}$	$\chi^2(1, n = 38) = 0.9360$	2.71 ($p < 0.1$)	No significant difference
$S_{cheatingDetection}$	$\chi^2(1, n = 38) = 0.7415$	2.71 ($p < 0.1$)	No significant difference
S_{Th+ABS}	$\chi^2(1, n = 38) = 0.0144$	2.71 ($p < 0.1$)	No significant difference

Table 5.3. Statistical comparison of confirmation bias metric values for two developer subgroups which belong to Group 1 and Group 7 respectively.

Confirmation Bias Metrics	**Result**	**Critical Value**	**Explanation**
$avgF_R^R$	$\chi^2(1, n = 38) = 1.6889$	2.71 ($p < 0.1$)	No significant difference
$avgF^{IR}$	$\chi^2(1, n = 38) = 0.3466$	2.71 ($p < 0.1$)	No significant difference
$avgL^{IR}$	$\chi^2(1, n = 38) = 0.0499$	2.71 ($p < 0.1$)	No significant difference
$minInd_{el/en}$	$\chi^2(1, n = 38) = 0.9070$	2.71 ($p < 0.1$)	No significant difference
$maxInd_{el/en}$	$\chi^2(1, n = 38) = 1.3329$	2.71 ($p < 0.1$)	No significant difference
$avgInd_{el/en}$	$\chi^2(1, n = 38) = 0.7029$	2.71 ($p < 0.1$)	No significant difference
$finalInd_{el/en}$	$\chi^2(1, n = 31) = 2.2452$	2.71 ($p < 0.1$)	No significant difference
$Ind_{el/en}$	$\chi^2(1, n = 31) = 0.0266$	2.71 ($p < 0.1$)	No significant difference
N_A	$\chi^2(1, n = 38) = 9.5742$	7.88 ($p < 0.005$)	$Group7^*$ outperforms $Group1^*$
IND_{ABORT}	$\chi^2(1, n = 38) = 0.2179$	2.71 ($p < 0.1$)	No significant difference
S_{Th}	$\chi^2(1, n = 38) = 5.2157$	2.71 ($p < 0.1$)	$Group7^*$ outperforms $Group1^*$
S_{ABS}	$\chi^2(1, n = 38) = 0.0334$	2.71 ($p < 0.1$)	No significant difference
$S_{precautional}$			Expected frequency in at least one cell is less than 5
$S_{cheatingDetection}$			Expected frequency in at least one cell is less than 5
S_{Th+ABS}			Expected frequency in at least one cell is less than 5

Table 5.4. Statistical comparison of confirmation bias metric values for two developer subgroups which belong to Group 1 and Group 8 respectively.

Confirmation Bias Metrics	**Result**	**Critical Value**	**Explanation**
$avgF_R^R$	$\chi^2(1, n = 38) = 0.1101$	2.71 ($p < 0.1$)	No significant difference
$avgF^{IR}$			Expected frequency in at least one cell is less than 5
$avgL^{IR}$			Expected frequency in at least one cell is less than 5
$minInd_{el/en}$	$\chi^2(1, n = 39) = 0.9247$	2.71 ($p < 0.1$)	No significant difference
$maxInd_{el/en}$	$\chi^2(1, n = 38) = 0.1983$	2.71 ($p < 0.1$)	No significant difference
$avgInd_{el/en}$	$\chi^2(1, n = 38) = 0.846$	2.71 ($p < 0.1$)	No significant difference
$finalInd_{el/en}$	$\chi^2(1, n = 31) = 0.0154$	2.71 ($p < 0.1$)	No significant difference
$Ind_{el/en}$	$\chi^2(1, n = 33) = 0.5833$	2.71 ($p < 0.1$)	No significant difference
N_A	$\chi^2(1, n = 38) = 2.5909$	2.71 ($p < 0.1$)	No significant difference
IND_{ABORT}			Expected frequency in at least one cell is less than 5
S_{Th}	$\chi^2(1, n = 38) = 6.6165$	2.71 ($p < 0.1$)	$Group8^*$ outperforms $Group1^*$
S_{ABS}	$\chi^2(1, n = 38) = 1.9539$	2.71 ($p < 0.1$)	No significant difference
$S_{precautional}$			Expected frequency in at least one cell is less than 5
$S_{cheatingDetection}$			Expected frequency in at least one cell is less than 5
S_{Th+ABS}			Expected frequency in at least one cell is less than 5

Table 5.5. Statistical comparison of confirmation bias metric values for two developer subgroups which belong to Group 2 and Group 7 respectively.

Confirmation Bias Metrics	**Result**	**Critical Value**	**Explanation**
$avgF_R^R$	$\chi^2(1, n = 38) = 2.6316$	2.71 ($p < 0.1$)	No significant difference
$avgF^{IR}$			Expected frequency in at least one cell is less than 5
$avgL^{IR}$			Expected frequency in at least one cell is less than 5
$minInd_{el/en}$	$\chi^2(1, n = 38) = 1.7989$	2.71 ($p < 0.1$)	No significant difference
$maxInd_{el/en}$	$\chi^2(1, n = 38) = 5.7965$	2.71 ($p < 0.1$)	$Group2^*$ outperforms $Group7^*$
$avgInd_{el/en}$	$\chi^2(1, n = 38) = 3.7089$	2.71 ($p < 0.1$)	$Group2^*$ outperforms $Group7^*$
$finalInd_{el/en}$	$\chi^2(1, n = 31) = 6.6523$	2.71 ($p < 0.1$)	$Group2^*$ outperforms $Group7^*$
$Ind_{el/en}$	$\chi^2(1, n = 31) = 0.2026$	2.71 ($p < 0.1$)	No significant difference
N_A	$\chi^2(1, n = 39) = 1.9006$	2.71 ($p < 0.1$)	No significant difference
IND_{ABORT}			Expected frequency in at least one cell is less than 5
S_{Th}	$\chi^2(1, n = 38) = 0.2680$	2.71 ($p < 0.1$)	No significant difference
S_{ABS}	$\chi^2(1, n = 38) = 0.1997$	2.71 ($p < 0.1$)	No significant difference
$S_{precautional}$	$\chi^2(1, n = 38) = 0.0653$	2.71 ($p < 0.1$)	No significant difference
$S_{everytime}$	$\chi^2(1, n = 38) = 0.2319$	2.71 ($p < 0.1$)	No significant difference
$S_{cheatingDetection}$	$\chi^2(1, n = 38) = 5.7184$	2.71 ($p < 0.1$)	$Group2^*$ outperforms $Group7^*$
S_{Th+ABS}			Expected frequency in at least one cell is less than 5

Table 5.6. Statistical comparison of confirmation bias metric values for two developer subgroups which belong to Group 2 and Group 8 respectively.

Confirmation Bias Metrics	**Result**	**Critical Value**	**Explanation**
$avgF_R^R$	$\chi^2(1, n = 38) = 1.8044$	2.71 ($p < 0.1$)	No significant difference
$avgF^{IR}$			Expected frequency in at least one cell is less than 5
$avgL^{IR}$			Expected frequency in at least one cell is less than 5
$minInd_{el/en}$	$\chi^2(1, n = 39) = 1.3398$	2.71 ($p < 0.1$)	No significant difference
$maxInd_{el/en}$	$\chi^2(1, n = 38) = 2.9167$	2.71 ($p < 0.1$)	Group 2* outperforms Group 8*
$avgInd_{el/en}$	$\chi^2(1, n = 38) = 0.3804$	2.71 ($p < 0.1$)	No significant difference
$finalInd_{el/en}$	$\chi^2(1, n = 31) = 0.8949$	2.71 ($p < 0.1$)	No significant difference
$Ind_{el/en}$	$\chi^2(1, n = 31) = 0.1379$	2.71 ($p < 0.1$)	No significant difference
N_A	$\chi^2(1, n = 38) = 0.1536$	2.71 ($p < 0.1$)	No significant difference
IND_{ABORT}			Expected frequency in at least one cell is less than 5
S_{Th}	$\chi^2(1, n = 38) = 6.6165$	2.71 ($p < 0.1$)	Group 8* outperforms Group 2*
S_{ABS}			Expected frequency in at least one cell is less than 5
$S_{precautional}$			Expected frequency in at least one cell is less than 5
$S_{everytime}$			Expected frequency in at least one cell is less than 5
$S_{cheatingDetection}$			Expected frequency in at least one cell is less than 5
S_{Th+ABS}			Expected frequency in at least one cell is less than 5

Table 5.7. Statistical comparison of confirmation bias metric values for two developer subgroups which belong to Group 7 and Group 8 respectively.

Confirmation Bias Metrics	**Result**	**Critical Value**	**Explanation**
$avgF_R^R$	$\chi^2(1, n = 38) = 0.0354$	2.71 ($p < 0.1$)	No significant difference
$avgF^{IR}$			Expected frequency in at least one cell is less than 5
$avgL^{IR}$			Expected frequency in at least one cell is less than 5
$minInd_{el/en}$	$\chi^2(1, n = 38) = 1.3398$	2.71 ($p < 0.1$)	No significant difference
$maxInd_{el/en}$	$\chi^2(1, n = 38) = 0.0134$	2.71 ($p < 0.1$)	No significant difference
$avgInd_{el/en}$	$\chi^2(1, n = 38) = 1.4599$	2.71 ($p < 0.1$)	No significant difference
$finalInd_{el/en}$	$\chi^2(1, n = 31) = 2.3467$	2.71 ($p < 0.1$)	No significant difference
$Ind_{el/en}$	$\chi^2(1, n = 31) = 2.7000$	2.71 ($p < 0.1$)	No significant difference
N_A	$\chi^2(1, n = 38) = 1.9006$	2.71 ($p < 0.1$)	No significant difference
IND_{ABORT}			Expected frequency in at least one cell is less than 5
S_{Th}	$\chi^2(1, n = 38) = 0.2481$	2.71 ($p < 0.1$)	No significant difference
S_{ABS}	$\chi^2(1, n = 38) = 2.5574$	2.71 ($p < 0.1$)	No significant difference
$S_{precautional}$			Expected frequency in at least one cell is less than 5
$S_{everytime}$	$\chi^2(1, n = 38) = 5.1219$	2.71 ($p < 0.1$)	$Group8^*$ outperforms $Group7^*$
$S_{cheatingDetection}$			Expected frequency in at least one cell is less than 5
S_{Th+ABS}			Expected frequency in at least one cell is less than 5

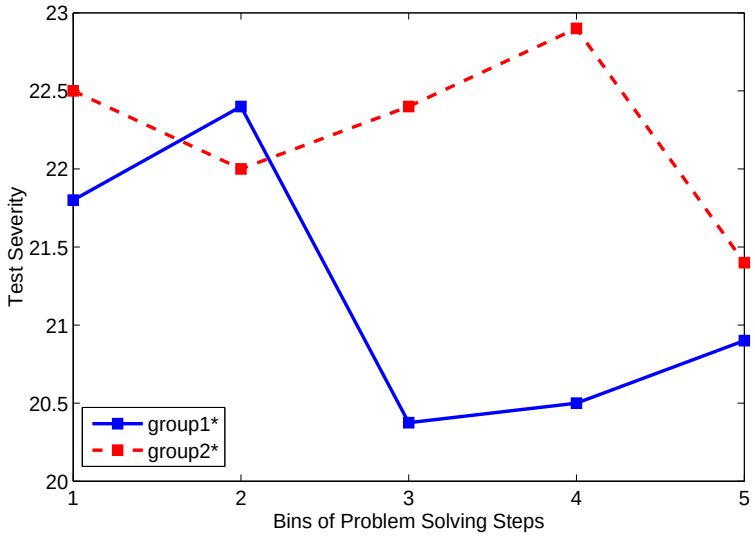


Figure 5.3. Comparison of interactive test hypothesis testing strategies of $Group1^*$ and $Group2^*$.

we analyze how confirmation bias affects performance of software testers.

5.2.1. Data

In this experiment, we analyze members of *Project Group 1* which is a subgroup consisting of developers and testers within *Group 1*. Details about *Group 1* and its subgroup *Project Group 1* are given in Table 4.1 and Table 4.3 respectively. In *Project Group 1*, there are 12 developers and 14 testers who are responsible from the development of a customer services package.

5.2.1.1. Data Used to Analyze Confirmation Bias Effects on Software Developers. We used data corresponding to the developers of *Project Group 1*, since all three of required data which consist of *confirmation bias metrics*, *file commit information* and *list of defected files* for each release were available for this group of developers. In order to obtain information about past source code commitment activities of each developer, log file of version management system data were examined. These file contains commitment history of all Java source codes since the beginning of the customer services software package development project. The project was launched in 2001, and the churn data also contains names

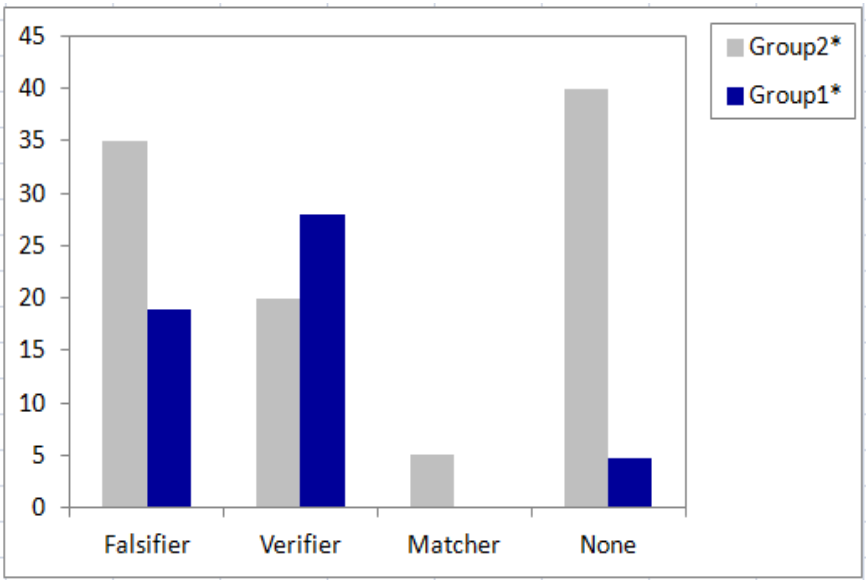


Figure 5.4. Distribution of *Falsifiers*, *Verifiers*, *Matchers* and *None* within $Group1^*$ and $Group2^*$.

of the developers who no longer work in the project. Therefore, files which were committed by any of these past developers are not taken into consideration in our analysis. Moreover, in the present project group out of the 12 developers, commit activities of only 6 developers could be observed in file commitment history. The rest of the development team was new to the project due to sudden change in the organizational structure at the time confirmation bias tests were conducted.

We obtained defect information after the analysis of the testing and release dates of the software package as well as the list of defected files detected during each testing phase. Every two weeks, a new release of the software is delivered and hence testing phase of one release and the development phase of the next release overlap. In this study, we analyzed 10 releases of the software that were developed and tested between the last week of May 2009 and second week of November 2009. For each release, we categorized each file to be defected or not based on the results of the testing phase for that release. For defects detected within a file during testing phase of each release, developers who created and/or updated that file before that date of testing phase were held responsible. Therefore, we were able to map each file to a group of developers who created and/or updated that file.

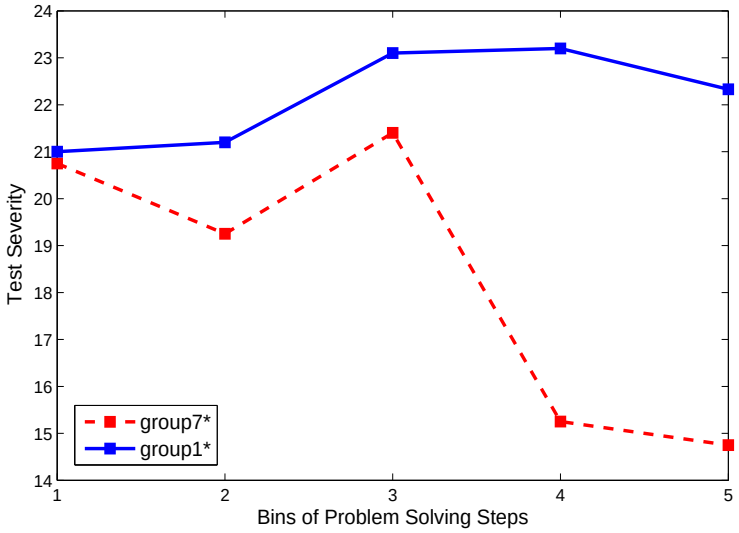


Figure 5.5. Comparison of interactive test hypothesis testing strategies of $Group1^*$ and $Group7^*$.

Finally, as a performance metric, we defined *defect density* for each developer group as the ratio of the total number of defected files created/updated by that group to the total number of files that group created/updated. Defining a metric for assessing individual performance of each developer was not possible. This is due to the fact that during testing phase it is highly probable that some defects might have been overlooked as a result of which defects may propagate from earlier releases of the software to its latest release. Therefore, any defect which is detected during testing phase might be caused during development phase of any earlier release.

5.2.1.2. Data Used to Analyze Confirmation Bias Effects on Software Testers. We used testers of *Project Group 1* in this part of the experiment, since in addition to confirmation bias metrics we were able to access information which can be used to assess software tester performance. In order to determine a reliable tester performance metric, we made use of company's tester competence reports. In these reports for each tester, number of post-release defects detected on parts of the software product which were tested by that tester is given. Hence we can set number of production defects ($NPROD_{DEF}$) as a tester performance metric.

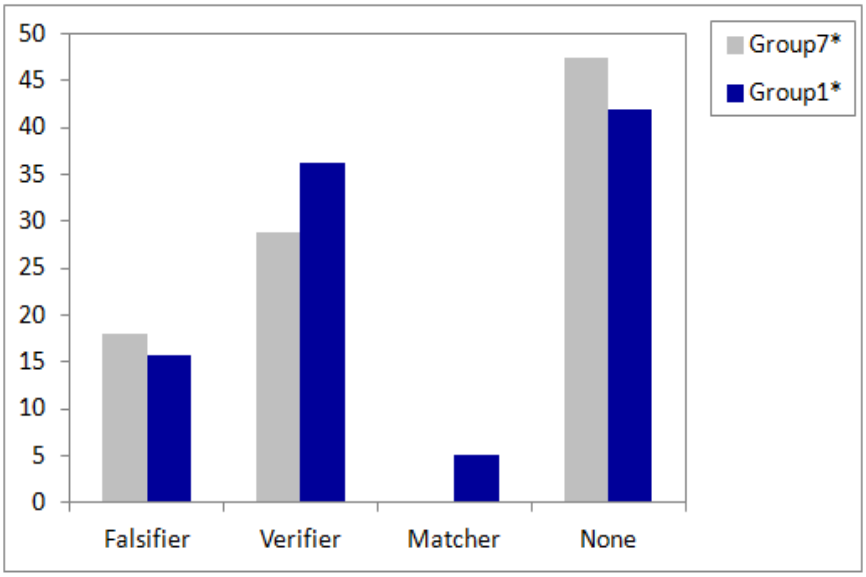


Figure 5.6. Distribution of *Falsifiers*, *Verifiers*, *Matchers* and *None* within *Group*1* and *Group*7*.

5.2.2. Design

5.2.2.1. Design for Analysis of Confirmation Bias Effects on Software Developers. In order to visualize the effect of confirmation bias on software defect density, we constructed a linear regression model with confirmation bias metrics as the predictor (independent) variables and *defect density* as the response variable. In this experiment, we used only confirmation bias metrics with continuous values. In order to evaluate confirmation bias metrics of developer groups, we took the average value of developer group members'confirmation bias metrics. In the case when a developer group consists of a single developer, this reduces to the confirmation bias metrics of that developer.

We decided to construct a linear regression model instead of a quadratic or an interaction regression model in order not to compromise the estimation of the regression coefficients, calculation of the confidence intervals and significance test results. In order to make this decision we had to construct all three models to see if the resulting residuals are normally distributed. Since linear dependency exists among the predictor variables leading to matrix singularity problem during the regression coefficient calculation, we performed principle component analysis (PCA). The scree plot of the percent variability explained by each principal component is shown in Figure 5.15. The variance is explained

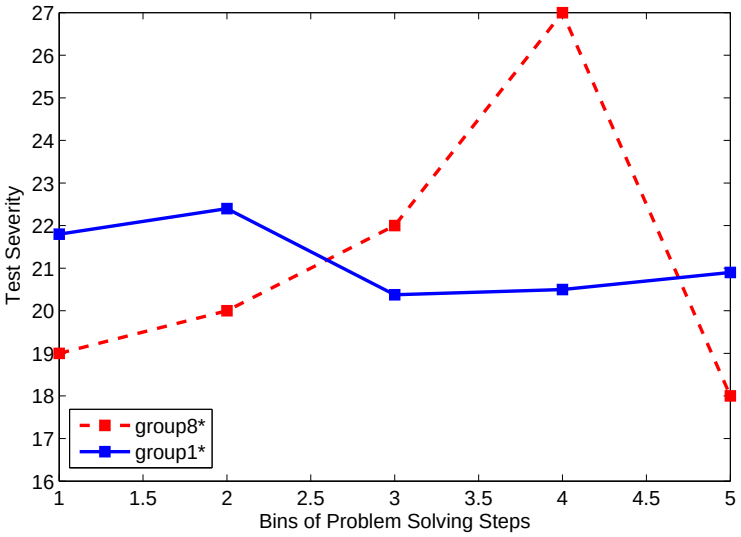


Figure 5.7. Comparison of interactive test hypothesis testing strategies of $Group1^*$ and $Group8^*$.

by the first five columns of the resulting matrix.

After having constructed linear, quadratic and interaction regression models, and plotted normal probability plot of residuals, we were able to observe that residuals only for the quadratic and interaction models were not normally distributed compared to the residuals of the linear model as it is shown in Figure 5.16. As a result, in order to visualize the effect of confirmation bias on software defect density, we performed linear regression modeling.

5.2.2.2. Design for Analysis of Confirmation Bias Effects on Software Testers. For the group of tester we calculated the average value for the tester performance metric $NPROD_{DEF}$. This allowed us to classify testers into two groups as being testers having $NPROD_{DEF}$ value above and below average respectively. Having categorized testers in two groups, performed Reich and Ruth's categorization to find the distribution of *Falsifiers*, *Verifiers* and *Matchers* within each group as well as testers who cannot be classified at all. Moreover, in order to visualize overall hypothesis testing strategy employed by each group, test severity graphs are drawn.

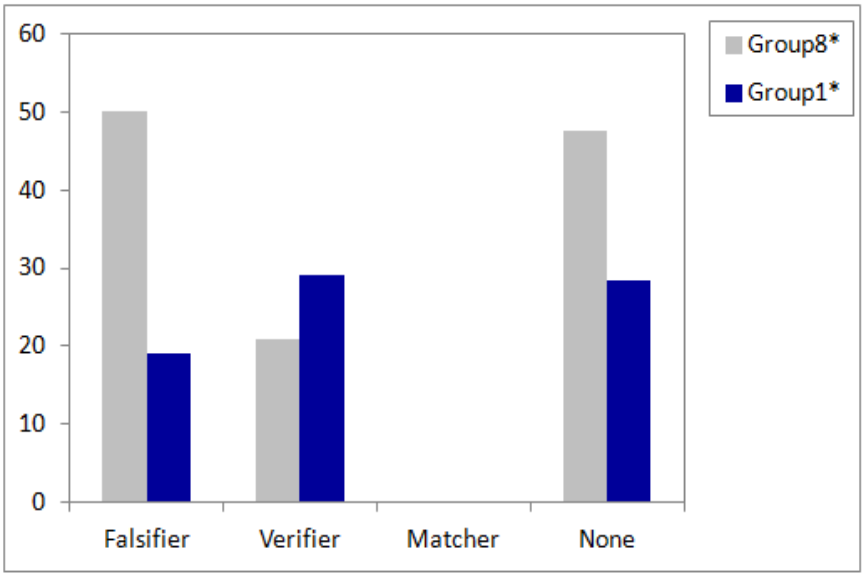


Figure 5.8. Distribution of *Falsifiers*, *Verifiers*, *Matchers* and *None* within $Group1^*$ and $Group8^*$.

5.2.3. Results and Discussions

5.2.3.1. Results for Analysis of Confirmation Bias Effects on Software Developers. We performed test for significance of regression to determine whether a linear relationship exists between the defect rate and a subset of confirmation bias metrics, given the following hypotheses: H_0: $\beta_1 = \beta_2 = ... = \beta_6$ and H_1: $\beta_1 \neq 0$, for at least one j. The F-statistics obtained with $\alpha = 0.05$ significance level is $F_0 = 17.0983 > F_{\alpha,k,n-k-1} = F_{0.05,6,117}$. This results in the rejection of the null hypothesis H_0 which implies that at least one of the independent variables contributes significantly to the model. As we performed PCA before regression modeling, each independent variable is a linear combination of confirmation bias metric values. Hence, confirmation bias metric values contribute significantly to the resulting linear regression model. We also performed significance tests on the individual regression coefficients. The null hypothesis is $H_0 : \beta_j = 0$ and alternative hypothesis is $H_0 : \beta_j \neq 0$. Except for significance tests of the third and fifth regression coefficients, null hypothesis is rejected. p values for each regression coefficient are given in Table 5.8. Finally, the amount of reduction in the variability of defect rate obtained by using the linear combinations of confirmation bias metrics is less than 50%. This amount is given by $R^2 = 0.4477$. Adjusted R^2 statistic is equal to 0.4243. As values of ordinary and adjusted R^2 are very close to each other, we can say that non-significant terms are

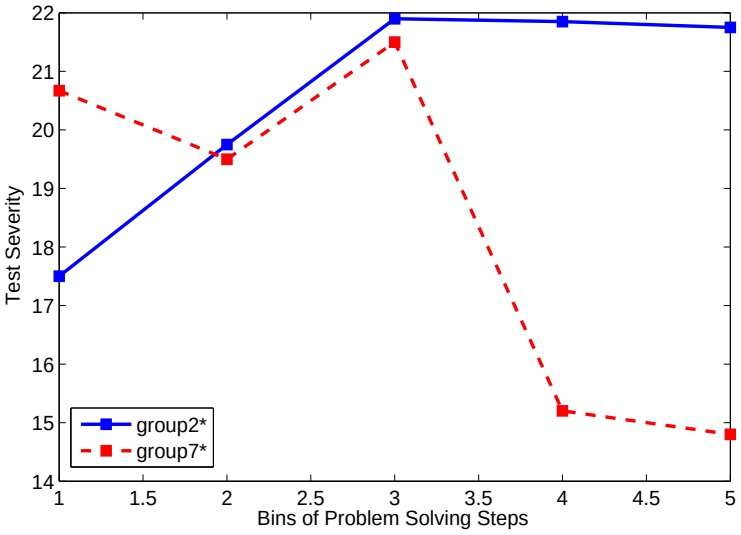


Figure 5.9. Comparison of interactive test hypothesis testing strategies of $Group2^*$ and $Group7^*$.

not added to the model. The value of R^2 for prediction is 0.3264, which means that we could only expect the constructed model to explain only about 32% of the variability in predicting new observations.

Results show that although confirmation bias metrics are not direct indicators of defect ratio of software developer groups, they affect software defect density. Moreover, in our case study during the prediction of the defect rate, confirmation bias metrics explain 32% of the variability the constructed linear regression model which is a significant amount in social sciences. If we take into account the fact that defect rate is affected by processes, and many human aspects other than confirmation bias, the results obtained are reasonable. Among human aspects affecting software defect, we can name cognitive biases such as representativeness, availability, adjustment and anchoring in addition to the social interactions. Therefore the capability of confirmation bias metrics to explain 32% of the variability in defect rate is a promising result.

5.2.3.2. Results for Analysis of Confirmation Bias Effects on Software Testers. Group of testers with $NPROD_{DEF}$ value below average (i.e. testers who were responsible from testing files which ended up having less post-release defects) exhibited a more strategic

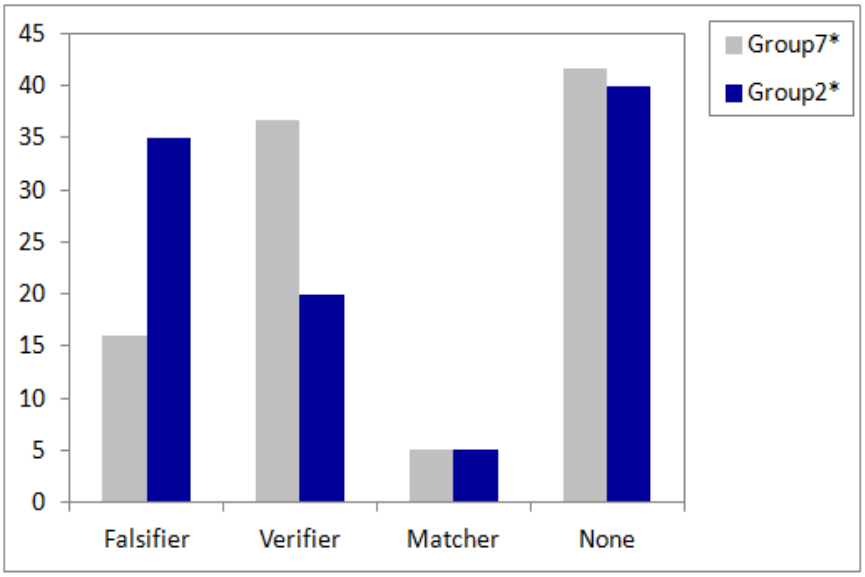


Figure 5.10. Distribution of *Falsifiers*, *Verifiers*, *Matchers* and *None* within $Group2^*$ and $Group7^*$.

approach during interactive tests. As it can be seen in Figure 5.17, this group of testers starts with a low level severe test and they progressively exclude more alternatives. As mentioned previously in Chapter 2, starting with highly severe instances during interactive test leads to the elimination of almost all hypotheses in subject's mind. Moreover, fluctuations observed in the Vincent curve of testers with $NPROD_{DEF}$ value above average are far away from being indicators of an ideal hypothesis testing strategy. Moreover, among tester with $NPROD_{DEF}$ value above average, there are no *Falsifiers*, but only *Verifiers* and *Matchers*.

Table 5.8. The values regression coefficients, their confidence intervals and significance test results.

Coefficient	**Coefficient Value**	**Confidence Interval**	**p-value**
β_1	6.5669	6.0569-7.0688	1.0791E-12
β_2	0.2696	0.0507-0.4896	0.0162
β_3	-0.1472	-0.4809-1.1866	0.3843
β_4	1.4814	1.0971-1.8657	6.543E-12
β_5	0.6248	0.0496-1.2000	0.0335
β_6	-1.2697	-1.9005- -0.6309	1.167E-4

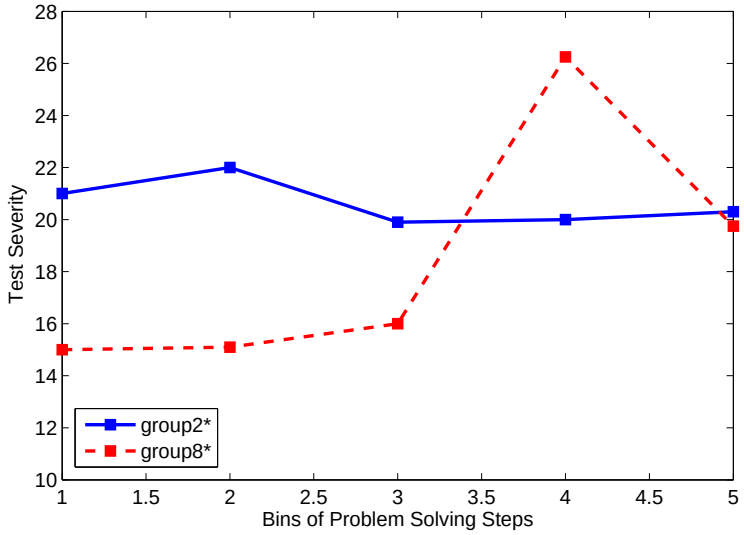


Figure 5.11. Comparison of interactive test hypothesis testing strategies of $Group2^*$ and $Group8^*$.

These results imply that lack of the tendency and reasoning skills to test hypothesis leads to overlooking defects during testing phase of the software which in turn leads to an increase in the number of post-release defects.

5.3. Experiment III: Using Confirmation Bias Metrics to Learn Defect Predictors

This experiment is in the form of a benchmark study, which aims to asses the power of defect prediction models that are built using *only* confirmation bias metrics as information content. In this experiment, we use two data sets which we collected from developers of two software development companies which are specialized in different domains. For each data set, we form defect prediction models using all combinations of static code metrics, confirmation bias metrics and churn metrics. This results in the formation of $2^3 - 1 = 7$ defect prediction models for each data set.

5.3.1. Data

In this study, we used two data sets which belong to Group 1 and Group 7 respectively. As shown in Table 4.1, Group 1 consists of developers and testers of a large scale

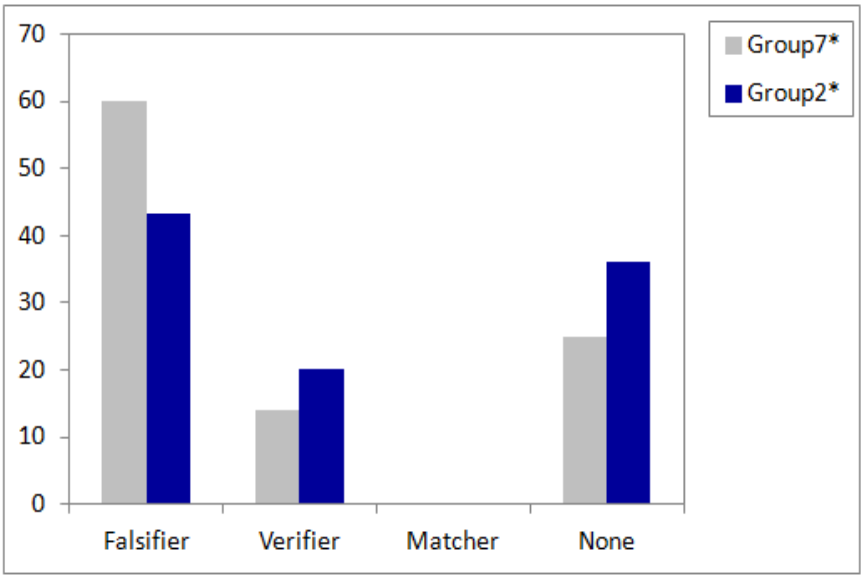


Figure 5.12. Distribution of *Falsifiers*, *Verifiers*, *Matchers* and *None* within *Group*2* and *Group*8*.

telecommunication company in Turkey, while Group 3 is a project group consisting of 6 developers in Turkey's largest Independent Software Vendor (ISV). There are 3 project groups in Group 1 and details about these project groups are given in Table 4.3. We used data set which belongs to Project Group 1, due to the following reason: while Project Group 2 and Project Group 3 are pilot project groups formed to assess the suitability of the software development methodologies TDD and TSP/PSP, Project Group 1 has been developing the customer services package since September 2001. Therefore, Project Group 3 is much more representative of a project group developing a software product in software development industry. Moreover, information about files that are found to be defective have been collected only for this project group.

As mentioned in the previous section, Project Group 3 consists of 12 developers and 14 testers. In our analysis, we were unable to match testers to the files since no information regarding which file is tested by which tester was not available. Hence, our experiments focus on developers. The project was launched in 2001, and the churn data also contains names of the developers who no longer work in the project. We were able to perform our confirmation bias tests to 3 software professionals who used to part of this development team. Therefore, files which were committed by any of these past developers, except for those of these 3 ex-developers, are not taken into consideration in our analysis.

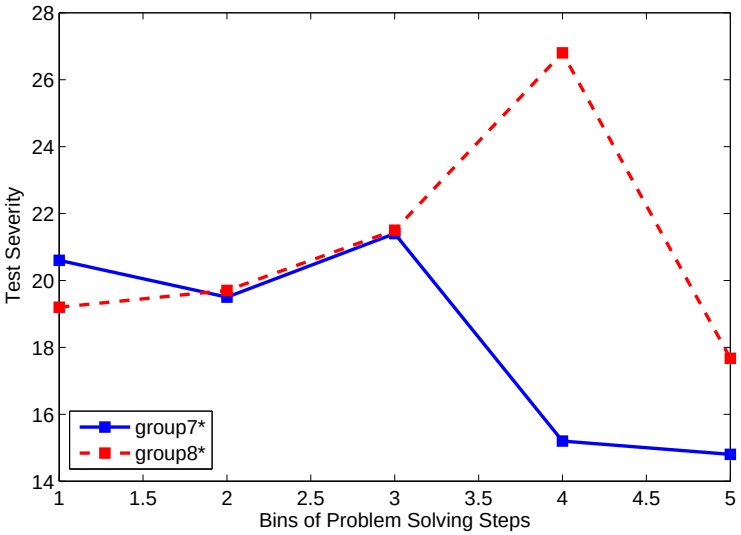


Figure 5.13. Comparison of interactive test hypothesis testing strategies of $Group7^*$ and $Group8^*$.

Moreover, in the present project group out of the 12 developers, commit information of only 6 developers could be found. The rest of the development team was new to the project due to sudden change in the organizational structure at the time confirmation bias tests were conducted. Therefore, regarding Project Group 3, we have confirmation bias metrics of 9 software professionals. Finally, traditional waterfall methodology is used for software development delivering a new release approximately every two weeks. In this study, we cover four versions of the software product released during June and July 2009 and each version consists of 545 files on average.

The second data set belongs to a project group that consists of 6 developers of the largest ISV (Independent Software Vendor) in Turkey. The software developed by the project group is an enterprize resource planning (ERP) software. The snapshot of the software that was retrieved from the version management system belongs to period of March 2011 and it consists of 3199 files. On the other hand, log file retrieved from the version management system for the second data set covers file commit activities starting from the beginning of July 2007 till the end of February 2011.

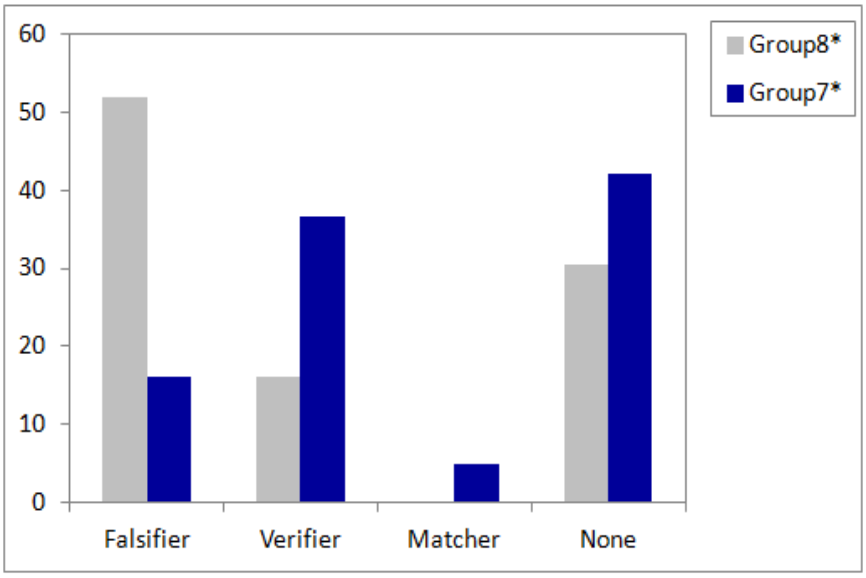


Figure 5.14. Distribution of *Falsifiers*, *Verifiers*, *Matchers* and *None* within *Group*7* and *Group*8*.

5.3.2. Design

For each file in each version, the developers who created and/or modified that file before the code freeze date are considered to be responsible from any defects found in that file. This is due to the fact that some previously introduced defects can be overlooked during testing phase of earlier versions resulting in the propagation of defects. Therefore, for each file in each version we examined, using the code freeze dates (i.e. dates when development phase for that release is over and testing phase starts) in the release calendar together with file commit information retrieved from the version control system, we obtained a group of developers responsible from that file.

In order to calculate confirmation bias metrics corresponding to each file, we consolidated confirmation bias metrics from individual developers to developer groups. For this purpose, we applied three different operators (i.e. min, max and average) for each confirmation bias metric of developers who contributed to the same source file and results are assigned to be the source file's corresponding feature. Assuming that, A_{di} represents the i^{th} confirmation bias metric value of d^{th} developer, $d \in G_j$ means that developer d is among the group of developers who created and/or modified jth source file, and finally, S_{ij}^{op} represents the i^{th} feature value of jth source file when operator op is applied, where

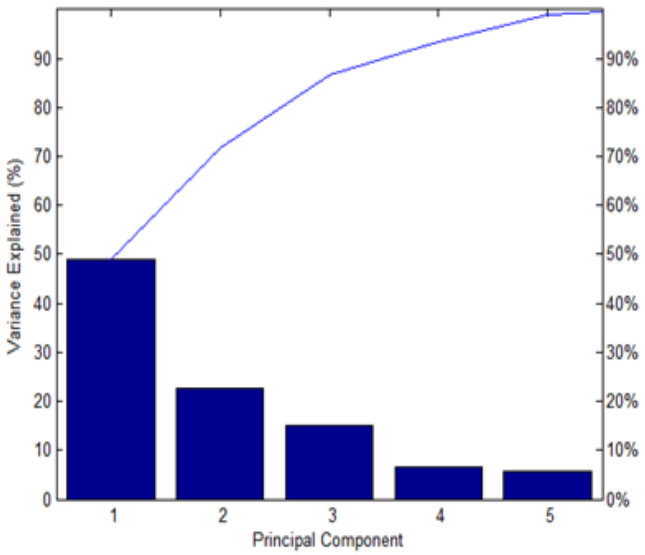


Figure 5.15. Percentage of variance explained by each of the principle components of the resulting matrix of predictor variables.

op can be one of the three operators, namely *min*, *max* or *avg*. Under these assumptions *min*, *max* and *avg* operators can be formalized as follows:

$$S_{ji}^{max} = max(A_{mi}|\forall m \in G_j) \tag{5.1}$$

$$S_{ji}^{min} = min(A_{mi}|\forall m \in G_j) \tag{5.2}$$

$$S_{ji}^{avg} = \frac{\sum_m (A_{mi}|m \in G_j)}{\sum_m (1|m \in G_j)} \tag{5.3}$$

In this experiment, confirmation bias metrics which takes continuous values are used as a result of the nature of the operators used to consolidate confirmation bias metrics of individual developers to those of developer groups. Using only continuous metrics was adequate to arrive a conclusion as it shall be explained in the section where we discuss results of this experiment.

5.3.2.1. Defect Matching. We have been doing research using data provided by the large scale telecommunication company for a couple of years which resulted in the improvement of their software development life cycle and software quality in various projects. Hence, the company has already established an infrastructure to list source files where bugs are detected during testing phase for each release of the software product.

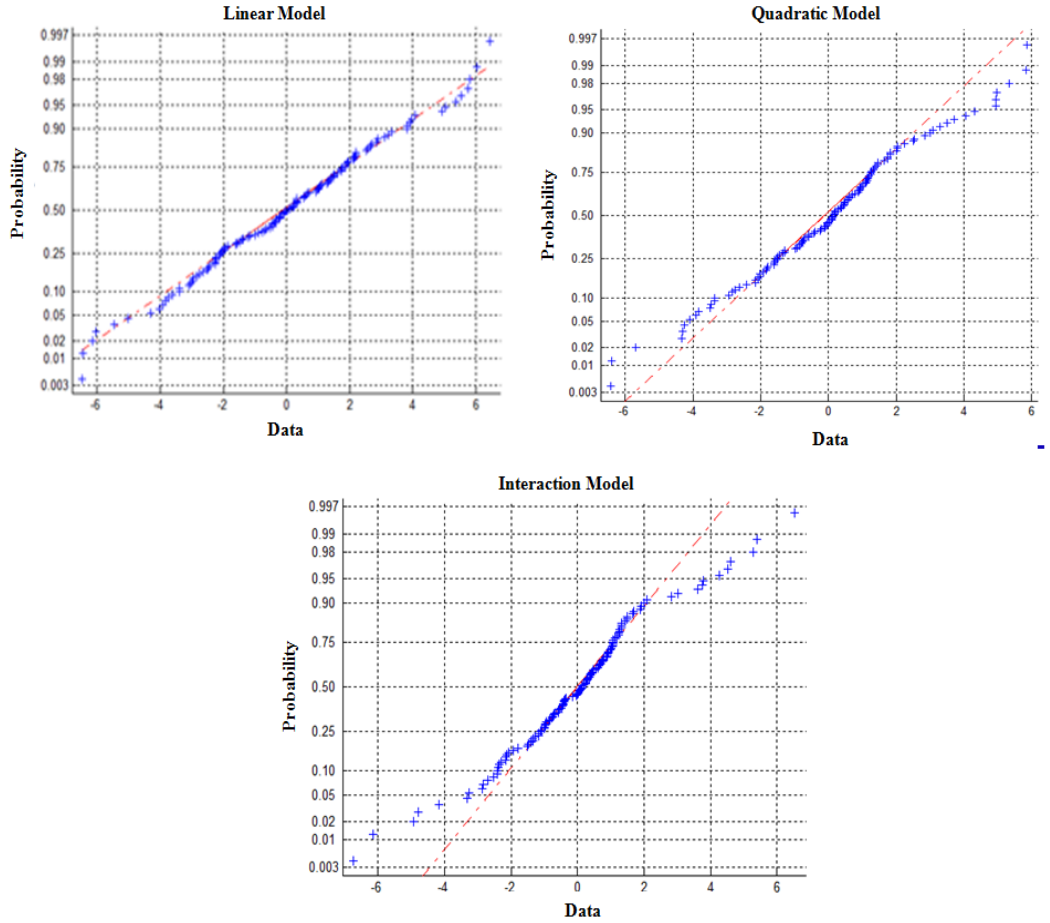


Figure 5.16. Normal probability plots of residuals for linear, quadratic and interaction regression models.

On the other hand, since this was the first time we collaborated with the ISV company, defect matching required more effort for the second data set. We had to learn about the work flow followed by the ISV during their software development life cycle. Figure 5.19 shows this work flow followed within the ISV to fix bugs which are detected during the testing phase.

The company uses an issue management system. Each issue is stored in this system with a unique issue code and it can be a new feature to be added to the software being developed, a regular project item or a defect that needs to be fixed. We managed to match issue items that were labeled as defect with source code files. According to the company's software development policy, developers must write the corresponding unique issue code as a comment before they commit file(s) to the version control system. There-

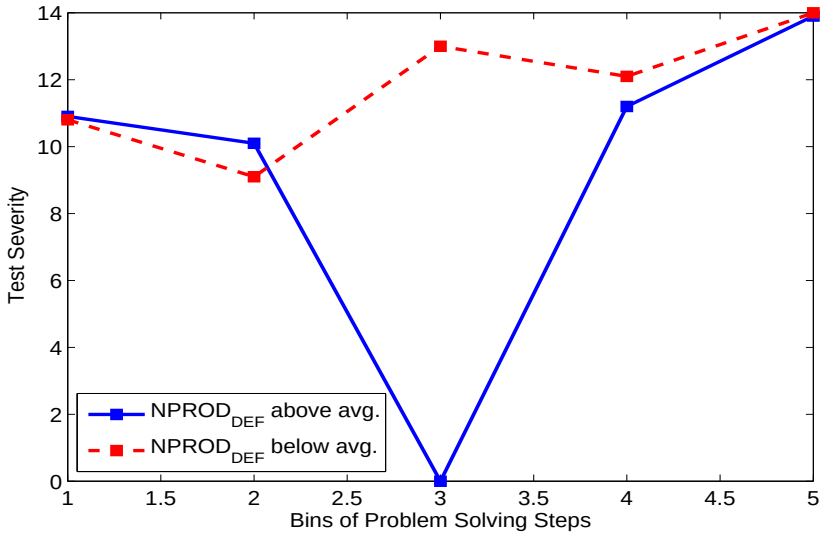


Figure 5.17. Vincent curves for test severity of testers with $NPROD_{DEF}$ values above and below average respectively.

fore, it was possible to match the file committed to the version management system with the corresponding issue item in the issue management system. Figure 5.20 shows the methodology we followed to extract the list of defective files. The company provided us with the issue list extracted from their issue management system. We formed a "final issue list" by taking into account only issue entries where request type is "defect" and issue status is different than "canceled". States of issues having request type "defect" are labeled according to the work flow given in Figure 5.19. According to this work flow, once a defect is detected during testing phase, requirement analysis and design may be revisited depending on the type of the defect. If analysts are busy, request regarding the defect is taken to the "Analysis Queue". However, requirement analysis and design phase need not to be revisited for all defect types. Hence, developers might directly be informed about the defect to be fixed. Defects of this type enter the "Coding Queue" only if developers are busy to fulfill tasks with higher priority in their to do lists. Once the defect (i.e. bug) is fixed by developers, testers are requested to test the piece of software related to the fixed defect. Once testers are informed, to ensure whether the defect was fixed by developers at the previous stage of the work flow related item may wait in the "Testing Queue" or may directly be taken to the testing phase depending on both the availability of testers and severity of the fixed defect. After the software with fixed defects is tested, in order to be sure about whether problems shall still arise or

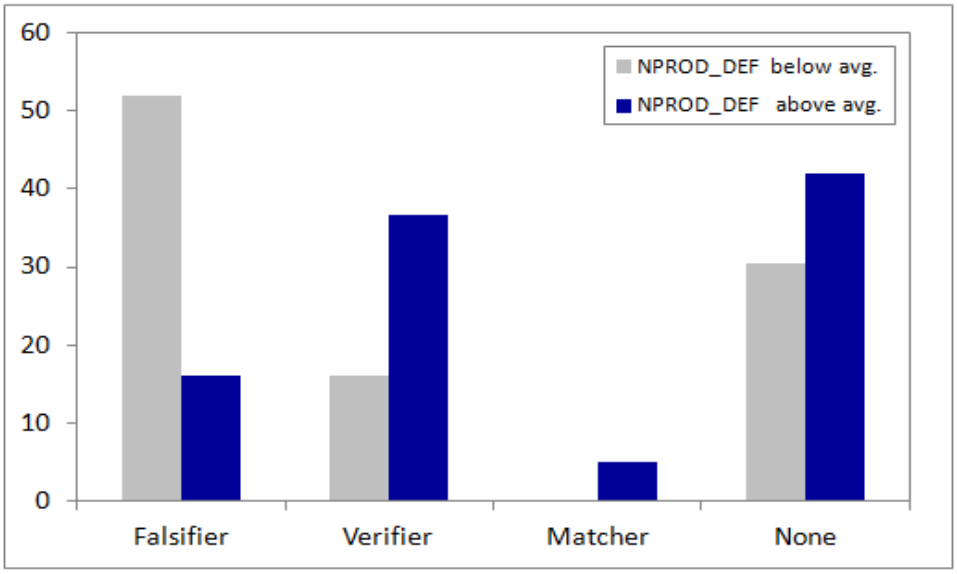


Figure 5.18. Distribution of falsifiers, verifiers, and matchers among testers who report bugs above and below average amount, according to Reich and Ruth's method.

not software is tested in the environment of the customer. This is the last stage in the work flow to fix defects. However, there might also be defects whose existence can not be verified in customers' environment settings although they may have been detected as a "defect" during testing phase. Items corresponding to this case are labeled with status "canceled" in the issue management system. Such detect detections are mostly due to the factors related to the testing environment of the tester and thus they do not affect the customer. We mined the commit log file obtained from the version control system to get a commit history file where format of each commit log entry is in the form as shown in Figure 5.20. Finally, for each issue in our final issue list we found names of source files in commit history file and marked those files as defective.

5.3.2.2. Formation of Train/Test Sets. The first data set is obtained by merging files in four versions of the software being developed. Before the merging process, defective files are labeled based on the list of defective files for each version. A developer is assigned as a member to the group of developers who committed a given file F, only if (s)he committed file F before the *code freeze date* of the version file F belongs to. *Code freeze date* is obtained from the release calendar provided to us by the telecommunication company. During the merging process, file entries with identical file names are assumed

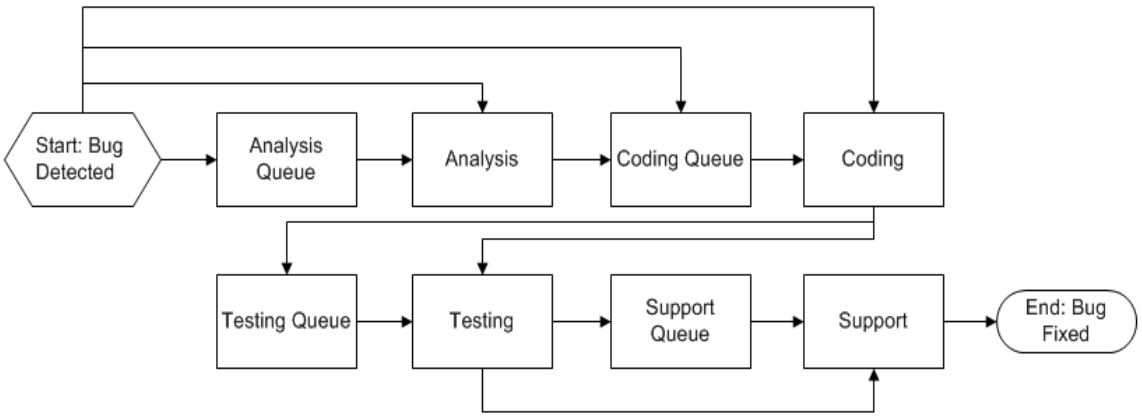


Figure 5.19. Work flow followed in the ISV company to fix bugs detected during testing phase.

to be different files if and only if corresponding static code metrics are different (i.e. that file has been modified). Otherwise, such a file is included to the list only once.

Since second data set consists of a single version of the software product, no merging process was necessary. The methodology followed to label defective files has already been explained in the previous section. In order to find group of developers who created/modified each file, procedure explained in the previous paragraph is used.

Within each data set, a file entry is excluded if there was at least one developer with unknown confirmation bias metrics among the committers of that file. Such a case is possible only if a file had been modified by at least one developer who no longer works at the company, so that it is not possible to conduct confirmation bias tests to these people.

5.3.2.3. Construction of Prediction Model. In this study, we used Naïve Bayes algorithm, since it combines signals coming from different attributes[5]. In software defect prediction studies, it is also empirically proven that the performance of Naïve Bayes is amongst the top algorithms[4]. Both data sets are imbalanced (defective file percentage in first data set is 11.1% and 8.4% in second data set). Therefore, we use under-sampling method that is the most suitable sampling method for our data sets [6]. 10-fold cross

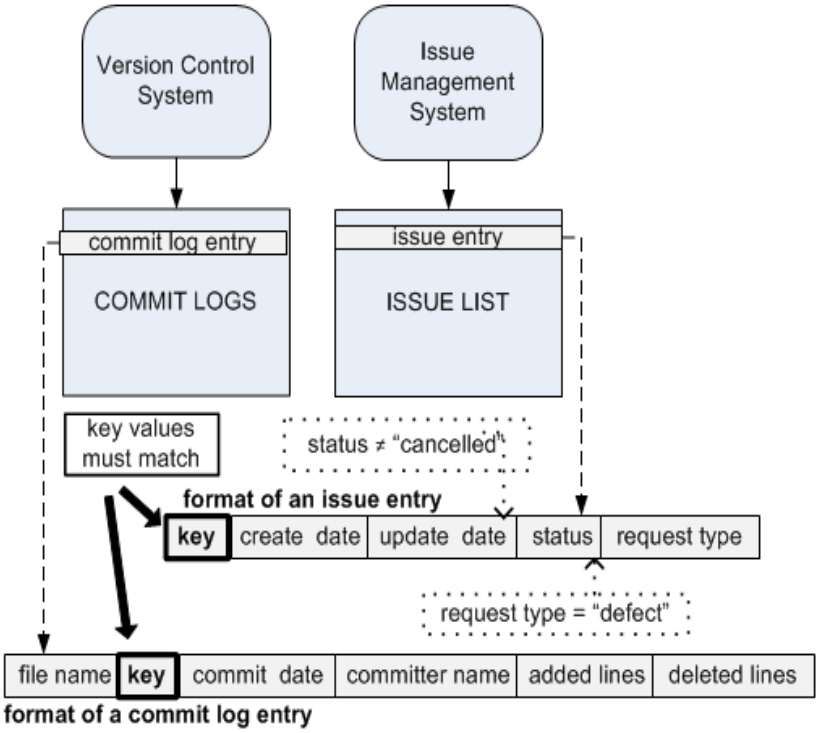


Figure 5.20. : Defect matching procedure of a file to prepare list of defected files for the second data set.

validation is used to avoid sampling bias in our experiments. In order to overcome ordering effects we shuffled data 10 times and 10-fold cross validation is used for each ordering configuration of input data. As a result, during each experiment Naive Bayes algorithm is executed 10x10=100 times, for each data set.

5.3.3. Results and Discussions

Two experiments are performed and each experiment is replicated for both data sets. In the first experiment, no data preprocessing method is used, while in the second experiment log-filtering is used to preprocess data. Log-filtering was used by Menzies *et al.* in [5] to improve predictor performance. All numeric values are replaced with those obtained by taking natural logarithms of these numeric values.

5.3.3.1. Using Confirmation Bias Metrics as Single Metric Set to Learn Defect Predictors.

If we take into account the results obtained for both data sets, the following can be con-

Table 5.9. Abbreviations used for metric type combinations in Tables.

Abbreviation	**Metric Type Combination**
SC	Static Code Metrics
CB	Confirmation Bias Metrics
Ch	Churn Metrics
SC+CB	Static Code and Confirmation Bias Metrics
SC+Ch	Static Code and Churn Metrics
CB+Ch	Confirmation Bias and Churn Metrics
SC+CB+Ch	Static Code, Confirmation Bias and Churn Metrics

Table 5.10. Defect prediction results for first data set.

Metrics				**Performance**		
Abbreviation	**Static Code**	**Confirmation Bias**	**Churn**	**pd**	**pf**	**balance**
SC	+	-	-	0.58	0.34	0.61
CB	-	+	-	0.63	0.32	0.64
Ch	-	-	+	0.61	0.39	0.58
SC+CB	+	+	-	0.64	0.29	0.67
SC+Ch	+	-	+	0.61	0.33	0.63
CB+Ch	-	+	+	0.66	0.31	0.67
SC+CB+Ch	+	+	+	0.68	0.28	0.69

Table 5.11. Defect prediction results for first data set with log filtering.

Metrics				**Performance**		
Abbreviation	**Static Code**	**Confirmation Bias**	**Churn**	**pd**	**pf**	**balance**
SC	+	-	-	0.53	0.34	0.54
CB	-	+	-	0.68	0.35	0.66
Ch	-	-	+	0.57	0.33	0.59
SC+CB	+	+	-	0.71	0.34	0.67
SC+Ch	+	-	+	0.66	0.31	0.65
CB+Ch	-	+	+	0.70	0.34	0.67
SC+CB+Ch	+	+	+	0.71	0.34	0.67

Table 5.12. Defect prediction results for second data set.

Metrics				**Performance**		
Abbreviation	**Static Code**	**Confirmation Bias**	**Churn**	**pd**	**pf**	**balance**
SC	+	-	-	0.62	0.12	0.69
CB	-	+	-	0.64	0.27	0.62
Ch	-	-	+	0.52	0.10	0.61
SC+CB	+	+	-	0.62	0.25	0.62
SC+Ch	+	-	+	0.58	0.12	0.65
CB+Ch	-	+	+	0.45	0.21	0.52
SC+CB+Ch	+	+	+	0.50	0.17	0.57

Table 5.13. Defect prediction results for second data set with log filtering.

Metrics				**Performance**		
Abbreviation	**Static Code**	**Confirmation Bias**	**Churn**	**pd**	**pf**	**balance**
SC	+	-	-	0.23	0.16	0.39
CB	-	+	-	0.65	0.26	0.62
Ch	-	-	+	0.70	0.16	0.68
SC+CB	+	+	-	0.58	0.13	0.63
SC+Ch	+	-	+	0.59	0.16	0.62
CB+Ch	-	+	+	0.57	0.20	0.59
SC+CB+Ch	+	+	+	0.52	0.19	0.56

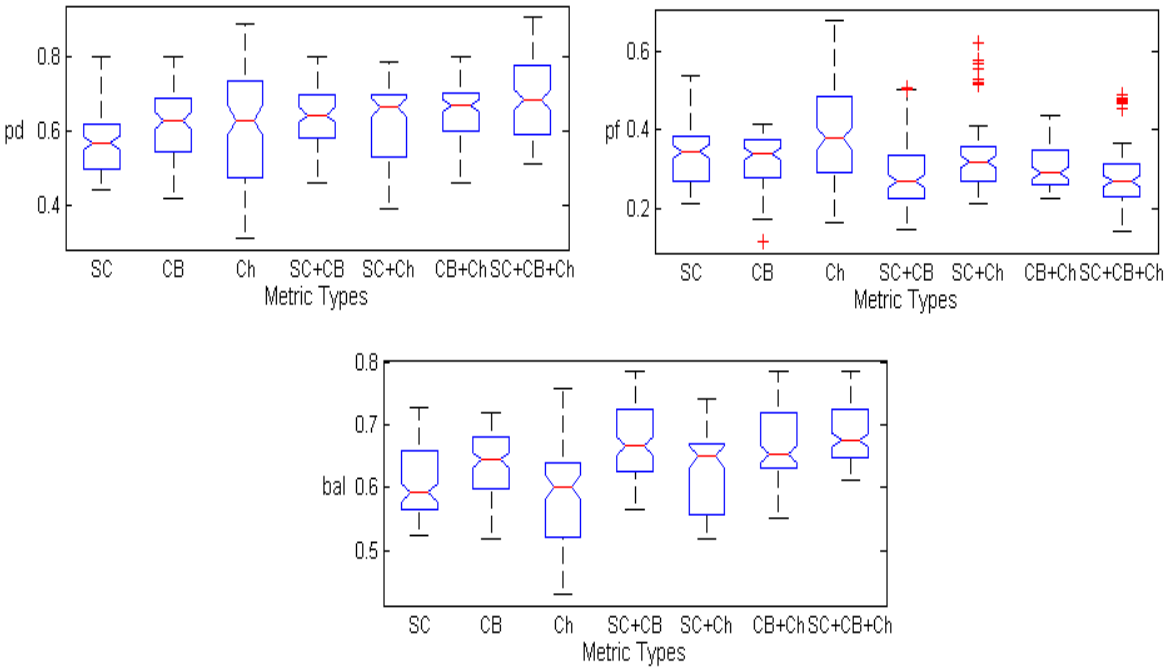


Figure 5.21. Experiment III boxplots for first dataset.

cluded regarding using confirmation as single metric set to learn defect predictors:

- Defect predictors learnt by using only confirmation bias metrics as input data have at least as high pd and balance values as the defect predictors learnt by using only static code metrics or only churn metrics as input data. Moreover, higher pd values are obtained for the first data set for both experiments.
- In the first experiment for the first data set, using only confirmation bias metrics instead of only static code metrics and only churn metrics leads to lower false alarm rates.
- However, in all these experiments for both data sets, learning defect predictors using only confirmation bias metrics never leads to an average false alarm rate lower than 0.26.
- Despite this fact, balance value is never lower than those of defect predictors which are learnt using either only static code metrics or only churn metrics.

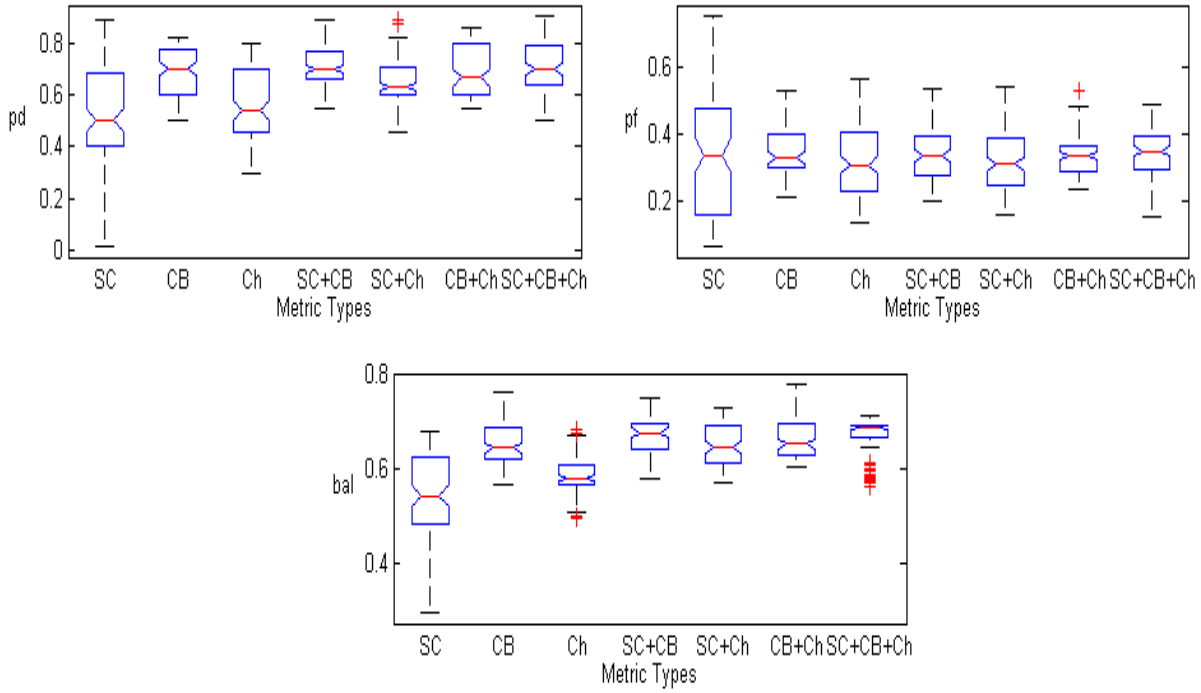


Figure 5.22. Experiment III boxplots for first dataset, where log-filtering is used to preprocess data.

5.4. Experiment IV: Learning Defect Predictors Using Incomplete Confirmation Bias Metric Set

Unlike static code and churn metrics which can be retrieved automatically, collecting confirmation bias metrics requires more effort and time. Therefore, defect prediction models built using only static code or churn metrics may be preferred rather than defect predictors which are built using only confirmation bias metrics, especially when comparable prediction performances are obtained using both models. In such cases, it might be a feasible choice to collect confirmation bias metrics of a certain percentage of developers and to use an imputation technique to obtain imputed confirmation bias metrics that can be used to learn defect predictors. Such an alternative can be considered as a solution only if comparable performance results are obtained using imputed form of confirmation bias metrics as input to defect prediction models.

Moreover, unlike static code and churn metrics, usage of confirmation bias met-

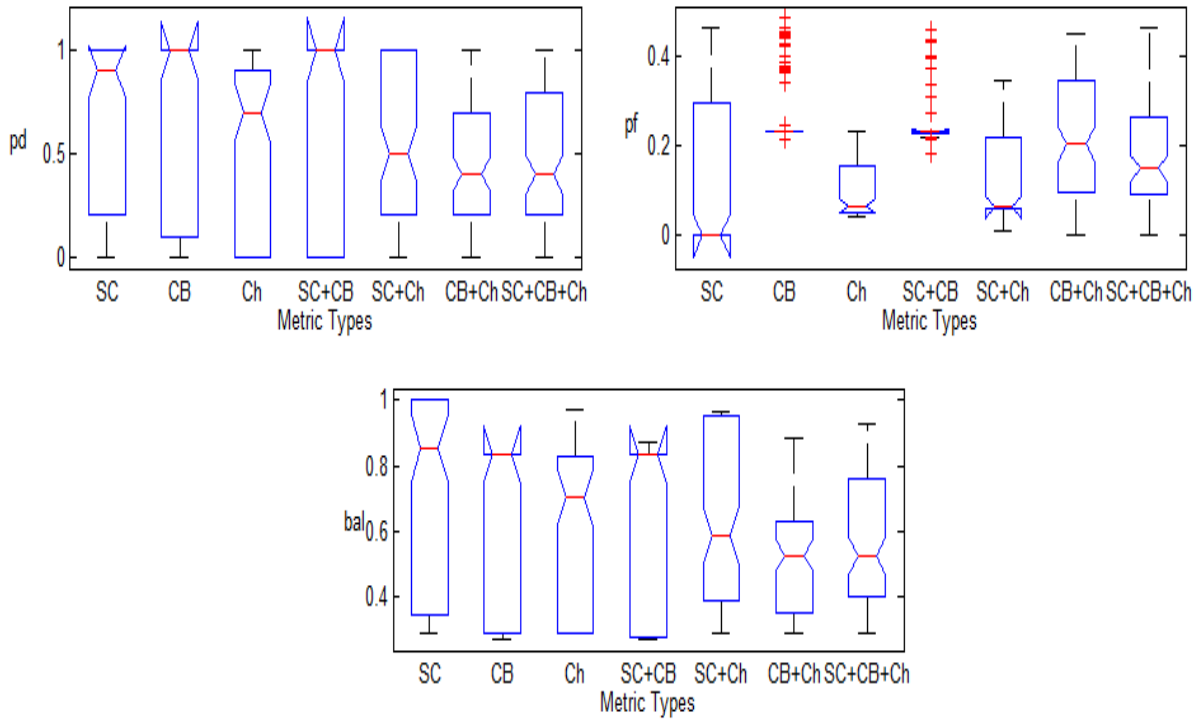


Figure 5.23. Experiment III boxplots for second dataset.

rics to learn defect predictors may confine defect prediction to a limited portion of the software product. For instance, in the case of defect prediction at file granularity level, each file F within each release R should be mapped with a group of developers G_d who created and/or updated file F starting from its creation till code freeze date of release R. Learning a defect predictor using confirmation bias metrics such that resulting predictor covers each source file of the software, is possible only if confirmation bias metrics of all developers who are mapped to file F are known. Even if the existence of a single developer within developer group G_d with unknown results in the exclusion of file F. Therefore, in such cases it is very likely to come up with a defect prediction model which covers only a small percentage of source code files that make up the software product. Such a scenario is inevitable as lifetime of the software being developed increases, since some of the developers shall be replaced by new comers. As a result, it shall be impossible to collect confirmation bias metrics of development team's previous members. In order to learn confirmation bias metric based defect predictors which cover the complete software product and obtain comparable prediction performances, imputation techniques are required.

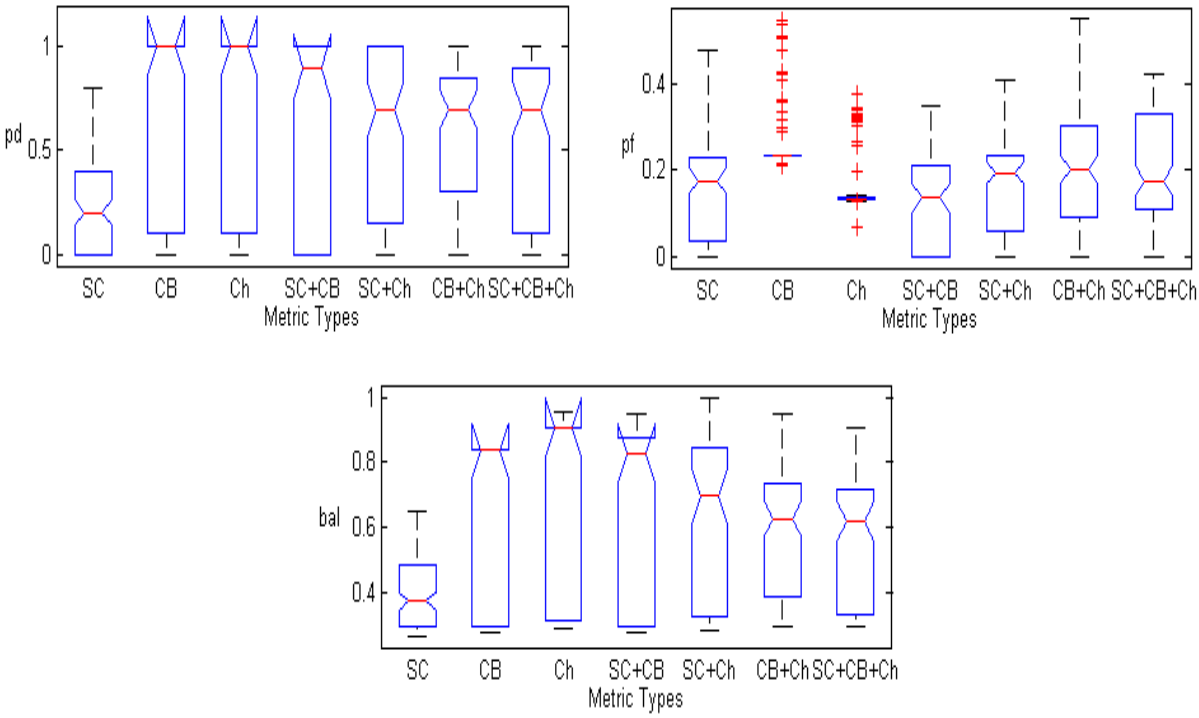


Figure 5.24. Experiment III boxplots for second dataset, where log-filtering is used to preprocess data.

In this section, we investigate whether it is possible to build defect predictors which are learnt using incomplete confirmation bias metric set so that comparable prediction performance results can still be obtained.

5.4.1. Data

We use the two data sets that were also used in Experiment III. Detailed information regarding these two data sets were given in the previous subsection. However, unlike Experiment III, defect predictors are built using only confirmation bias metrics.

5.4.2. Design

In this experiment, we use only standardized forms of confirmation bias metric values to learn defect predictors. In the original data sets, confirmation bias metrics of developers are known. As we have already stated, our goal in this experiment is

to compare the performance of defect predictors built using *complete* confirmation bias metrics with the performance of prediction models which are learnt using *incomplete* confirmation bias metric set. In order to make a comparison, we assume that confirmation bias metrics of a subgroup of developers is unknown, which implies that confirmation bias metrics of any file created and/or updated by any member of this subgroup of developers is missing. Under this assumption for each complete data set, it is possible to create 2^N-2 incomplete data sets, where N is the total number of developers whose confirmation bias metric values are known. 2^N is the total number of subsets of the set of developers with known confirmation bias metrics. When we exclude the empty set and the complete set of developers, we obtain $2^N - 2$ missing data configurations (i.e. data sets with incomplete data).

For each incomplete data set, we employ Roweis' Expectation Maximization Algorithm to impute the missing data. The pseudo code explaining details of Roweis' EM Algorithm is given in Figure 3.2. Output of Roweis' Algorithm is the imputed form of the incomplete data set. We build defect prediction models using each of these imputed confirmation bias metric sets as input. This results in the formation of $2^N - 2$ defect predictors. Prediction performance of each of these predictors is compared with the performance of the prediction model that is built using the complete confirmation bias metric set.

The pseudo code for the formation of all missing data configurations for both data sets is given in Figure 5.25, while the pseudo code in Figure 5.26 explains how each missing data configuration is imputed using Roweis' EM algorithm. In the pseudo code, in Figure 5.26 a call to function $ImputeMissingData()$ is made. As shown in Figure 3.2, function $ImputeMissingData()$ which takes a missing data configuration as input is the implementation of Roweis's EM Algorithm. Finally, Naïve Bayes algorithm is used to predict defects in each imputed data set. As a sampling method, under-sampling is used to deal with imbalanced data problem.

Percentage of missing data (i.e. missing %) for each incomplete data set, that is created using the algorithm 5.25, can be calculated as $N_{complete}/(N_{complete}+N_{missing})x100$. In this simple formulation, $N_{complete}$ is total number of complete entries in the data set,

```
MetricType = "Confirmation Bias Metrics"
Group[1] = "Project Group 1"; Group[2] = "Group 3";
for all i in [1:1:2] do
  DataSet[i] = DataSetOf(Group[i], MetricType);
  % list of source file names in the first column of DataSet[i]
  Files[i] = filesOf(DataSet[i])
  % find developer group members of i^th data set
  Developers[i] = DevelopersOf(DataSet[i])
  % find total numbers of members in the developer group of i^th data set
  N[i] = Size(Developers[i]);
  % form the list of subgroups of the developer group of i^th data set
  DeveloperSubsets[i] = DeveloperSubsetsOf(Developers[i])
  % for each missing data configuration
  for all j in [1:1:2^N[i] − 2] do
    for all developer in DeveloperSubsets[i, :] do
      for all k in [1:1:size(DataSet[i], 1)] do
        fileName = DataSet[i][k][1]
        if IsACommitterOf(developer, fileName) == TRUE then
          for all t in [2:1:size(DataSet[i], 1) − 1] do
            DataSet[i][k][t] = "missing"
          end for
        end if
      end for
    end for
  end for
  IncompleteDataSet[i][j]= DataSet[i]
end for
```

Figure 5.25. Pseudo code for generation of missing data configurations for data sets "Project Group 1" and "Group 3".

```
for all i in [1:1:2] do
  % for each missing data configuration
  for all j in [1:1:2^N[i] − 2] do
    ImputedDataSet[i][j] = ImputeMissingData(IncompleteDataSet[i][j])
  end for
end for
```

Figure 5.26. Pseudo code for imputation of each missing data configuration.

and $N_{missing}$ corresponds to total number of entries with missing confirmation bias metric values. Each entry in the data set corresponds to a source code file and it consists of the confirmation bias metric values of the group of developers who created and/or updated that file. Confirmation bias metric values of each developer is estimated from the outcomes of confirmation bias tests conducted by us. In order to calculate confirmation bias metric values of each developer group we use, the operators defined by the equations 5.3. Therefore, missing confirmation bias metrics of one or more developers from the group of developers, who created and/or updated a source code file, automatically implies that confirmation bias metrics for the developer group of that file are completely missing. In other words, the entry in the data set corresponding to that file is completely missing.

5.4.3. Results and Discussions

As a result of Experiment IV, $2^N - 2$ prediction performance values (i.e. *pd*, *pf* and *balance*) are obtained for each data set, where N is the total number of developers who develop the software product for which defect prediction models are build. Since we build defect predictors using imputed form of each missing data configuration of data sets, this implies that $2^9 - 2 = 510$ defect prediction models are learnt for the first data set, while for the second data set this corresponds to $2^6 - 2 = 62$ defect predictors. Missing confirmation bias metric values of developer(s) correspond to a specific missing data percentage (missing %). We categorized imputed forms of missing data configurations based on the resulting missing % in the overall data set. For this purpose formed 9

Table 5.14. Defect prediction performance results for complete form of first data set.

Metric Type	**pd**	**pf**	**balance**
Confirmation Bias	0.63	0.32	0.64
Static Code	0.58	0.34	0.61
Churn	0.61	0.39	0.58

Table 5.15. Defect prediction performance results for incomplete confirmation bias metric values of the first data set as input.

Missing %	**pd**	**pf**	**balance**
$(0, 10]$	0.8013	0.2834	0.7087
$(10, 20]$	0.7929	0.3029	0.6950
$(20, 30]$	0.7395	0.3118	0.6628
$(30, 40]$	0.7141	0.3419	0.6306
$(40, 50]$	0.7079	0.3804	0.6075
$(50, 60]$	0.6792	0.3424	0.6097
$(60, 70]$	0.6003	0.2136	0.6110
$(70, 80]$	0.6073	0.2856	0.5555
$(80, 90]$	0.6660	0.4263	0.4869

categories, such that a defect predictor built using a data set with missing % greater than $(i-1)*10\%$ and less than or equal to $i*10\%$ belongs to category C_i, where $1 \leq i \leq 9$. Within each category C_i, we calculated the average of prediction performance results (i.e. *pd*, pf, *balance*) of defect predictors which belong to that category. Table 5.16 and Table 5.18 show average *pd*, pf and *balance* values corresponding to each missing % category for the first and second data sets respectively. Mean Square Error (MSE) values for the imputed form of the first and second datasets are shown in Figure 5.15 and Figure 5.19, respectively. As it can be seen from Table 5.16 and 5.18, increase in missing % leads to a decrease in balance value.

The inverse relation between balance and missing percentage is also shown in the upper left graphs in Figures 5.27 and 5.28. Pearson correlation between balance and missing % is -0.9092 ($p = 0.9955E-171$) for the first data set, while this value is -0.8700 ($p = 1.5794E-018$). As it can be seen in Figures 5.27 and 5.28, correlation

Table 5.16. MSE values for imputed form of incomplete confirmation bias metric values for the first data set.

Missing %	**MSE**
$(0, 10]$	1.0020
$(10, 20]$	1.5601
$(20, 30]$	1.5596
$(30, 40]$	1.6507
$(40, 50]$	1.7448
$(50, 60]$	1.3638
$(60, 70]$	0.8622
$(70, 80]$	0.9169
$(80, 90]$	0.9801

Table 5.17. Defect prediction performance results for complete form of the second data set.

Metric Type	**pd**	**pf**	**balance**
Confirmation Bias	0.64	0.26	0.62
Static Code	0.58	0.34	0.61
Churn	0.62	0.12	0.69

Table 5.18. Defect prediction performance results for incomplete confirmation bias metric values of the second data set as input.

Missing %	**pd**	**pf**	**balance**
$(0, 10]$	0.8013	0.2834	0.7087
$(10, 20]$	0.7929	0.3029	0.6950
$(20, 30]$	0.7395	0.3118	0.6628
$(30, 40]$	0.7141	0.3419	0.6306
$(40, 50]$	0.7079	0.3804	0.6075
$(50, 60]$	0.6792	0.3424	0.6097
$(60, 70]$	0.6003	0.2136	0.6110
$(70, 80]$	0.6073	0.2856	0.5555
$(80, 90]$	0.6660	0.4263	0.4869

Table 5.19. MSE values for imputed form of incomplete confirmation bias metric values for second data set.

Missing %	**MSE**
$(0, 10]$	1.0020
$(10, 20]$	1.5601
$(20, 30]$	1.5596
$(30, 40]$	1.6507
$(40, 50]$	1.7448
$(50, 60]$	1.3638
$(60, 70]$	0.8622
$(70, 80]$	0.9169
$(80, 90]$	0.9801

Table 5.20. Pearson correlation results for missing % vs. pd, pf, balance and MSE.

	Data Set # 1		**Data Set # 2**	
Missing % vs.	ρ	**p**	ρ	**p**
balance	-0.9092	0.9955E-171	-0.8700	1.5794E-018
pd	-0.4120	1.0674E-19	-0.5500	9.3399e-006
pf	-0.2287	1.0596E-06	0.2090	0.1188
MSE	0.2227	2.33E-06	-0.6566	2.9167e-008

between missing % and *pd*, pf and MSE values are not as high as the correlation values obtained for the relation between missing % and balance values for both first and second data sets. Table 5.20 list Pearson correlation values for missing % versus balance, *pd*, pf and MSE values respectively, for both first and second data sets used in this experiment.

According to the results given in Table 5.14 and 5.16 for the first data set, defect predictors which are built using confirmation bias metrics data with missing % up to 30% results in comparable balance values with the balance value of the defect predictor built using complete input data. Although balance values decrease as missing % increases, even usage of input data with missing % within the range 80%-90% results in defect predictors having balance values the comparable with general performance values of rule-based prediction models [24].

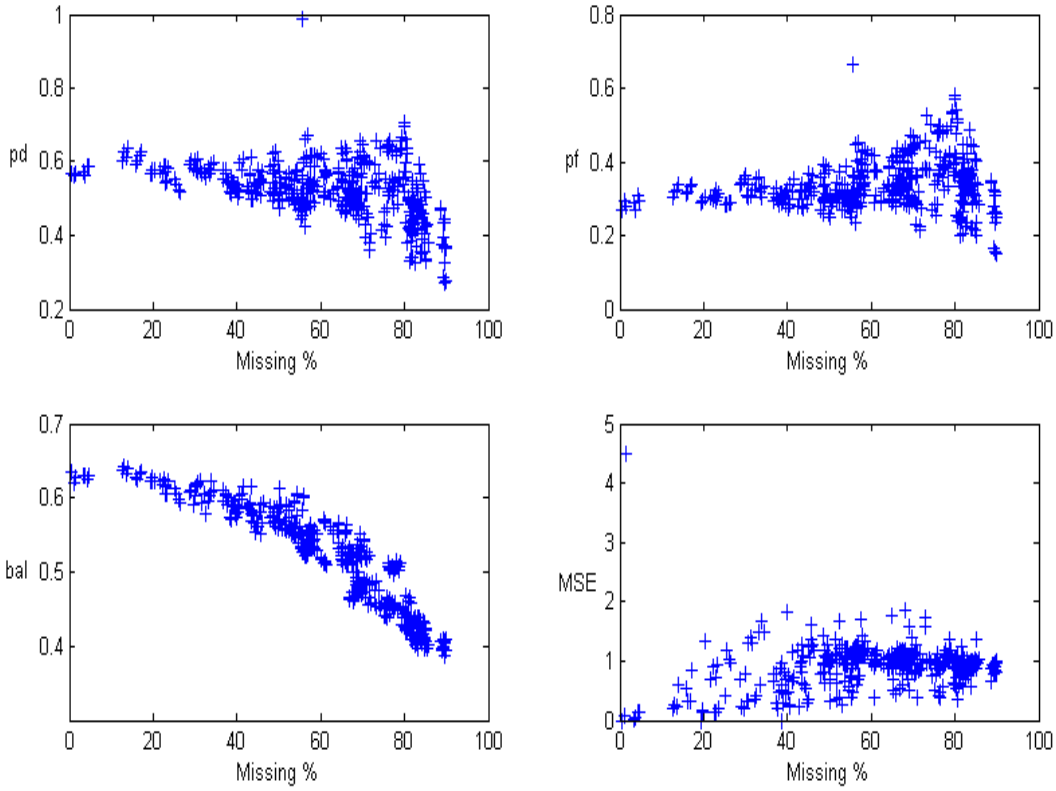


Figure 5.27. Missing data percentage versus balance, pd, pf and MSE for the first data set.

As it is shown in Table 5.17 and 5.18, results obtained for the second data set are in line with the results obtained for the first data set. Moreover, up to 40% missing data percentage, average performance results of the defect predictors learning from incomplete data outperforms prediction performance results of the defect predictor that uses complete confirmation bias metrics as input. Within the missing data percentage range of 40%-70%, still the performance results obtained are comparable with performance of the original prediction model. Finally, prediction performance within missing data percentage range of 70%-90% performance results are comparable with prediction performance of rule-base defect predictors.

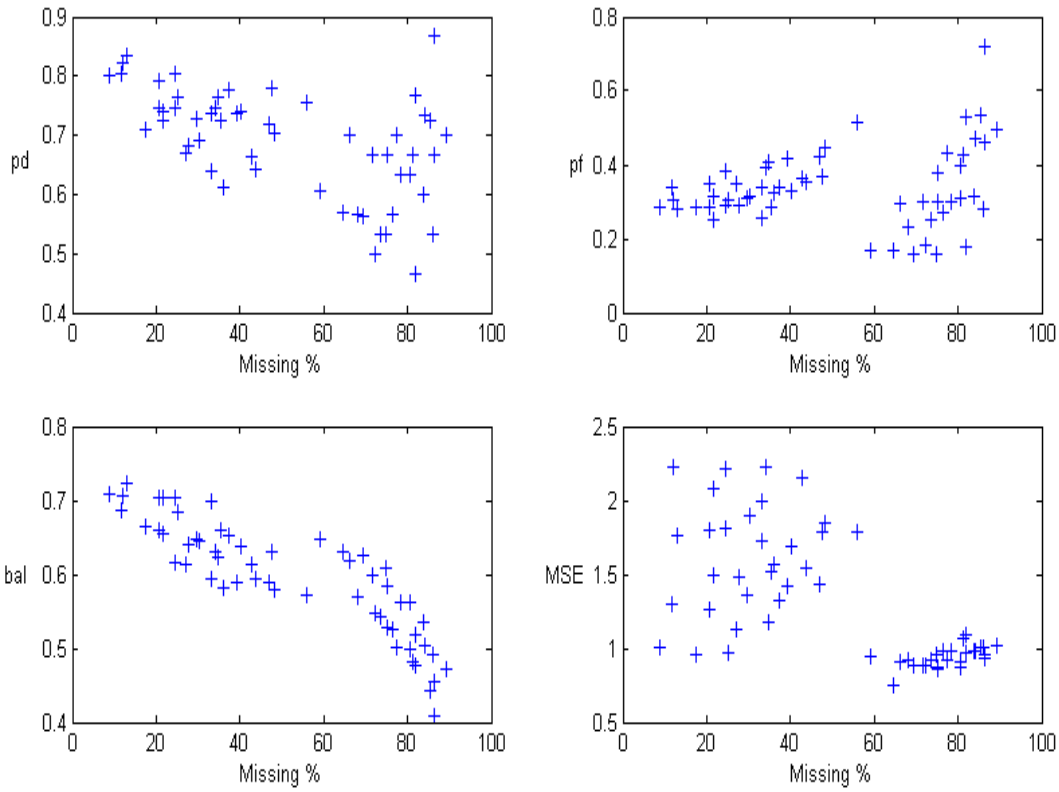


Figure 5.28. Missing data percentage versus balance, pd, pf and MSE for the first data set.

5.5. Threats to Validity

5.5.1. Threats to Validity for Definition and Extraction of Confirmation Bias Metrics

In order to avoid mono-method bias, which is one of the threats to construct validity, we used more than a single version of a confirmation bias measure. In other words, we defined a set of confirmation bias metrics. In order to form our confirmation bias metric set, we made an extensive survey in cognitive psychology literature covering significant studies which have been conducted since the first introduction of the term "confirmation bias" by Wason in 1960 [1]. Moreover, we made a definition of confirmation bias in relation to software development life cycle. Since our metric definition and extraction methodology is iterative, we were able to improve contents of our metric set through pilot study and datasets collected during our previous related research [72], [73], [74]. As

a result, we were able to demonstrate that multiple measures of key constructs we use behave as we theoretically expect them to.

Another threat to construct validity is interaction of different treatments. Before administration of confirmation bias test to groups of participants, we ensured that none of the participants were involved simultaneously in several other programs designed to have similar effects.

Evaluation apprehension is a social threat to construct validity. Many people are anxious about being evaluated. Moreover, some people are even phobic about testing and measurement situations. In order to avoid participants' poor performance due to their apprehension and not to exert psychological pressure on them, before solving both written question set and interactive question, participants were informed about the fact that the questions they are about to solve do not aim to measure IQ or any related capability. Participants were also told that results shall not be used in their company's performance evaluations and their identity shall be kept anonymous. Moreover, participants were told that there is no time constraint to complete the questions, although some of our metrics are requires measurement of time it takes to answer questions.

Another social threat to construct validity is the *expectancies of the researcher* [75]. There are many ways a researcher can bias the results of a study. Hence, the outcomes of both written question set and interactive question were evaluated by two researchers independently, one of the researchers not actively being involved in the study. She was given a tutorial about how to evaluate the confirmation bias metrics from the outcomes of the written question set and interactive question. However, in order not to induce a bias she was not told about what the desired answers to the questions are. The inter-rater reliability was found to be high, for evaluation of each confirmation bias metric. Average value for Cohen's kappa was 0.92. During administration of the confirmation bias test, explanations made to the participants before they started solving the questions did not include any clue about ideal responses. Moreover, while participants were solving interactive question an independent researcher attended the session in order to observe whether the researcher in charge affects participants' response or not through his/her gestures or facial expressions. Dialogues, which took place during solution of the

interactive question, were also recorded. These recordings were later examined to find out whether researcher in charge gives any clues to the participant about the expected result. Parts of the datasets, which were found to be affected by the expectancies of the researcher, were excluded from the empirical investigation.

In order to avoid internal threats to validity, for all project groups we selected test dates, when workload of developers are not intense. Within any of the groups, there was no event in between the confirmation bias tests that can affect subjects' performance. Members of the developer group corresponding to the first dataset took confirmation bias test, which consists of written question set and interactive question, within a week. Remaining developer groups took the confirmation bias test in a single day. As a result, within a project group for each member we managed to create similar conditions while administering confirmation bias test. If one group member were tested when work load and time pressure were intense, whereas another member of the same group were tested under much more suitable and relaxed conditions, then our methodology would not have been reliable. Another attempt to avoid internal validity was to administer confirmation bias test in environments, which are isolated from distraction factors such as noise.

5.5.2. Threats to Validity in General

We consider three major threats to the validity of our experiments:construct, internal, external. To avoid the construct validity threats during Experiments III and IV, we used in terms of measurement artifacts, we used three popular performance measures in software defect prediction research: probability of detection (pd), probability of false alarm rates (pf) and balance values (bal). In order to avoid internal validity threats, in Experiments III and IV, we used 10-fold cross validation to avoid sampling bias in our experiments. In order to overcome ordering effects we shuffled data 10 times and 10-fold cross validation is used for each ordering configuration of the input data. As a result, during each experiment Naïve Bayes algorithm is executed 100 times, for each data set. For statistical validity, we used Chi Square Test [71, 76–78] for Experiment I while analyzing the effects of some factors on confirmation bias and we used Mann-Whitney U test for performance comparison of the defect prediction models which we built for Experiments III and IV. As another attempt to avoid internal validity threats,

we performed written and interactive tests, which we use to collect confirmation bias metrics, within a week for the first group and within a single day for the second group, respectively. As a result, within a project group for each member we managed to create similar testing conditions. Otherwise, if one group member were tested when work load and time pressure were intense, whereas another member of the same group were tested under much more suitable and relaxed conditions, then our methodology would not have been reliable. For both project groups we selected test dates, when workload of developers are not intense. Within any of the two groups, there was no event in between the confirmation bias tests that can affect subjects' performance. Moreover, both testing environments were isolated from distraction factors such as noise. During Experiments I, in terms of internal validity our quasi-independent variables are experience, education, hypothesis testing and reasoning , and software development methodology.

In order to externally validate our results, we collected data from 7 different software development companies located in two different countries. Moreover, we tried to keep the variety of the domains in which these companies are specialized as high as possible. However, since Experiments II, III, and IV we require information regarding source file commit logs retrieved from version management systems, our analysis were confined with participant groups for which such data was available. Moreover, since most companies do not keep list of defected files after each testing phase of the software product, we had to use data sets from only two developer groups. Yet, up to a certain extend we managed to cover variety in software development domain since one data set belongs to a telecommunication company while the other belongs to an ISV specialized in developing ERP solutions.

6. CONCLUSIONS

6.1. Summary of Results

Results of Experiment I, showed that hypothesis testing and reasoning skills have a significant effect in circumventing negative effects of confirmation bias. Confirmation bias levels of individuals who have been trained in logical reasoning and mathematical proof techniques are significantly lower. In other words, given a statement such individuals show tendency to refute that statement rather than immediately accepting its correctness. A significant effect of experience in software development/testing has not been observed. This implies that training in organizations is focused on tasks rather than personal skills. Considering that the percentage of people with low confirmation bias is very low in the population [1, 62] an organization should find ways to improve basic logical reasoning and strategic hypothesis testing skills of their

In addition to the fact that both Groups 2^* and 8^* both outperformed in confirmation bias tests compared to members of the Groups 1^* and 7^*, Group 8^* members also outperforms members of $Group2^*$. Individuals, who are experienced but inactive in software development/testing, score better in confirmation bias tests than active experienced software developers/testers. This implies that companies should balance work schedule of testers similar to jet pilots and allow them periodically to take some time off the regular routine.

Results of Experiment II empirically showed the effect of confirmation bias on software developers and testers. The amount of reduction in the variability of defect rate obtained by using the linear combinations of confirmation bias metrics is less than 50%. This amount is given by $R^2 = 0.4477$. Adjusted R^2 statistic is equal to 0.4243. The value of R^2 for prediction is 0.3264, which means that we could only expect the constructed model to explain only about 32% of the variability in predicting new observations. Results show that although confirmation bias metrics are not direct indicators of defect ratio of software developer groups, they affect software defect density. Moreover, in our case study during the prediction of the defect rate, confirmation bias metrics explain 32%

of the variability the constructed linear regression model which is a significant amount in social sciences. If we take into account the fact that defect rate is affected by processes, and many human aspects other than confirmation bias, the results obtained are reasonable. Among human aspects affecting software defect, we can name cognitive biases such as representativeness, availability, adjustment and anchoring in addition to the social interactions. Therefore the capability of confirmation bias metrics to explain 32% of the variability in defect rate is a promising result. Besides software developers, results of Experiment II showed that number of post-release defects in source files is directly proportional to confirmation bias levels of testers who test those source files.

Defect prediction models we built during Experiment III using *only* confirmation bias metrics gave prediction performance results comparable to performance results of the defect predictors which we built using *only* either static code or churn bias metrics. Finally, in Experiments IV we tackled missing data problem. The results of this last experiment showed that defect predictors which use imputed confirmation bias metric data as input gives comparable prediction performance results when compared with the performance of defect predictors built using complete confirmation bias metric values as input. Static code metrics we used to learn defect prediction models include Lines of Code (LOC), Halstead [79] and McCabe [80] metrics. In other words, we have covered the set of all major metrics which can be extracted from source code regarding code complexity based on program flow and readability of the code. Similarly, the churn metric set, which we have employed in our analysis, contains extensive information about the changes in source code during the implementation phase. We extracted significant portion of information regarding code change history from version management systems.

On the other hand, confirmation bias metrics represent only a single aspect about people's thought processes. Despite this, according to our empirical findings using only confirmation bias metrics to learn defect predictors yields comparable performance results. Moreover, the phenomena of cognitive biases, which is only one dimension regarding people's thought processes, comprise other bias types such as representativeness, availability, adjustment and anchoring in addition to confirmation bias. In cognitive psychology, the causes of biases have been extensively investigated in various domains over the past three decades since the introduction of the concept of bias by Kahneman and

Tversky [47]. In addition to cognitive biases, concepts widely studied in cognitive science such as attention, memory, reasoning, motivation, social cognition are also among the cognitive aspects which require special attention. Hence, there is extensive amount of findings in the field of cognitive psychology which can be employed to form a metric suite covering developers' cognitive aspects which have significant effect on software defect density, hence on software quality.

We are aware of the fact that our research is empirical and as stated by Popper, we cannot verify a theory with limited number of empirical findings, yet falsify it [81]. Hence, we need to be careful while generalizing our experimental results in order not to be subject to confirmation bias we have been discussing in this paper. However, our empirical findings suggest that the effects of cognitive aspects and people's thought processes on software quality deserves to be investigated. Moreover, obtaining comparable performance results in software defect prediction by confirmation bias metrics implies that further investigation of people's thought processes may help us to overcome the ceiling effect in defect prediction performance.

6.2. Contributions

There are three ingredients of software projects, which are product, process and people. Performance increase of defect predictors depends on the enhancement of input data content. Although product and process metrics have been widely used in software defect prediction, there is not much work regarding people metrics. In spite of the fact that confirmation bias is a *single* aspect of people who consists of many other aspects as well, defect predictors built using only confirmation bias metrics give prediction performance results comparable with those of traditional defect prediction models which are built using either static code or churn metrics. Our findings very strongly support the claim that the ceiling effect in the performance of software developers can only be overcome by increasing the information content of prediction models using *people* metrics.

However, unlike static code and churn metrics it takes time and effort to collect people related metrics. In our research, we collected confirmation bias metrics by conducting written and interactive tests to software professionals. Especially interactive test

may result in spending significant amount of effort for confirmation bias metric collection. In this case, confirmation bias metrics seem to loose their attraction. However, in our research we proposed a methodology to build confirmation bias metrics based defect prediction models which can learn from incomplete confirmation bias metrics data. Using this methodology, it is sufficient to built defect prediction model for a software product by conducting confirmation bias tests only to 70% of the development team. Moreover, event when 20-30% of the development team is covered during the collection of confirmation bias metrics performance results obtained are better than the performance of rule based defect predictors [24].

In addition, as the lifetime of a software being developed increases number of developers who used to work in the project increases. Since it is impossible to conduct confirmation bias tests to previous members of the development team, defect predictors build using only confirmation bias metrics inevitably cover only a portion of the software product being developed. As the number of ex-top developers increases this portion also decreases. The methodology we proposed in Experiment IV also solves this problem by treating confirmation bias metrics of the groups containing at least one ex-developer as missing data.

6.3. Future Directions

Inclusion of people aspects which affect software defect density as input data to prediction models is essential to overcome the performance ceiling in defect prediction models. For this purpose, people aspects that have the potential to affect the introduction of defects into software must be identified. After the identification of those potential aspects, empirical studies need to be carried out to ensure the effect of these people aspects on software defect density. Moreover, a methodology needs to be introduced to measure/quantify these people aspects.

In this research, we concentrated on confirmation bias and identified a methodology to define confirmation bias metrics. As mentioned previously, collection of these metrics require more effort unlike static code and churn metric that can be collected by using available tool. As future work, the goal of this study to automate the process of conduct-

ing confirmation bias tests as well as the estimation of confirmation bias metric values from the test outcomes. This automation tool shall give us the opportunity to gather more data from software development companies specialized in various domains in different countries. We aim to increase the variety and quantity of our data set to replicate our experiments, since we are aware of the fact that as human beings we ourselves are also under the risk of being influenced by the confirmation bias. In addition, we also aim to extend our defect prediction models to cover other cognitive bias types such as availability, representativeness and adjustment and anchoring [47, 62, 82].This software will help us to improve our metric suite to cover other relevant cognitive aspects that are briefly mentioned above. Since our software has been designed to be a decision support tool, it shall also be able to analyze the metrics and make recommendations to software professionals.

The objective of this research in the long run is to help software development managers make specific resource allocation decisions by considering metrics related to people's thought processes. Such a metric scheme will help managers to determine the right person to test the defective parts of the software. As a result, guidance of metrics related to people's thought processes may decrease the uncertainty in Human Resource (HR) related decisions up to a significant extent.

APPENDIX A: Interactive Test (English Version)

There are English and Turkish versions of the interactive test. We present the English version of the interactive test in Figure A.1, A.2 and A.3.

Figure A.1 shows the first page of the interactive test, where participant is supposed to fill in his/her personal information. Requested personal information consists of age, gender, educational status as well as years of experience in development and testing. Second page of the interactive test is shown in Figure A.2. The first information given about the interactive test is that this test does not aim to measure IQ or performance of a subject. This is followed by the detailed information about the procedure followed during the interactive test. Third page is shown in Figure A.3. On this page, participant writes down triples of numbers and reason of choice for each triple. According to the feedback given by the examiner, third or fourth columns are marked. If a triple of numbers conforms to the rule which is supposed to be discovered by the participant, then the examiner marks the third column. Otherwise, examiner marks the fourth column to indicate that the corresponding triple of numbers does not conform to the rule which is supposed to be discovered. At the end of the third page, there writes the following: "If you want extra pages, you can request from the examiner." This implies that there is no restriction on the total number of triples given by the participant. However, as it is indicated on the second page of the interactive test, which is shown in Figure A.2, the participant should try to find the correct rule by giving as minimum triples of numbers as possible, in the ideal case.

PERSONAL INFORMATION

Age: .....

Sex: ☐ Female ☐ Male

Education Status:

Undergraduate Degree

Name of the University: .....

Major: .....

2. Major: .....

(If you have a double major degree)

M.S./M.A. Degree

Name of the University: .....

Major: .....

PhD Degree

Name of the University: .....

Major: .....

Do you have experience as a professional software developer? ☐ Yes ☐ No

If your answer to the above question is "Yes", please write down your years of experience:

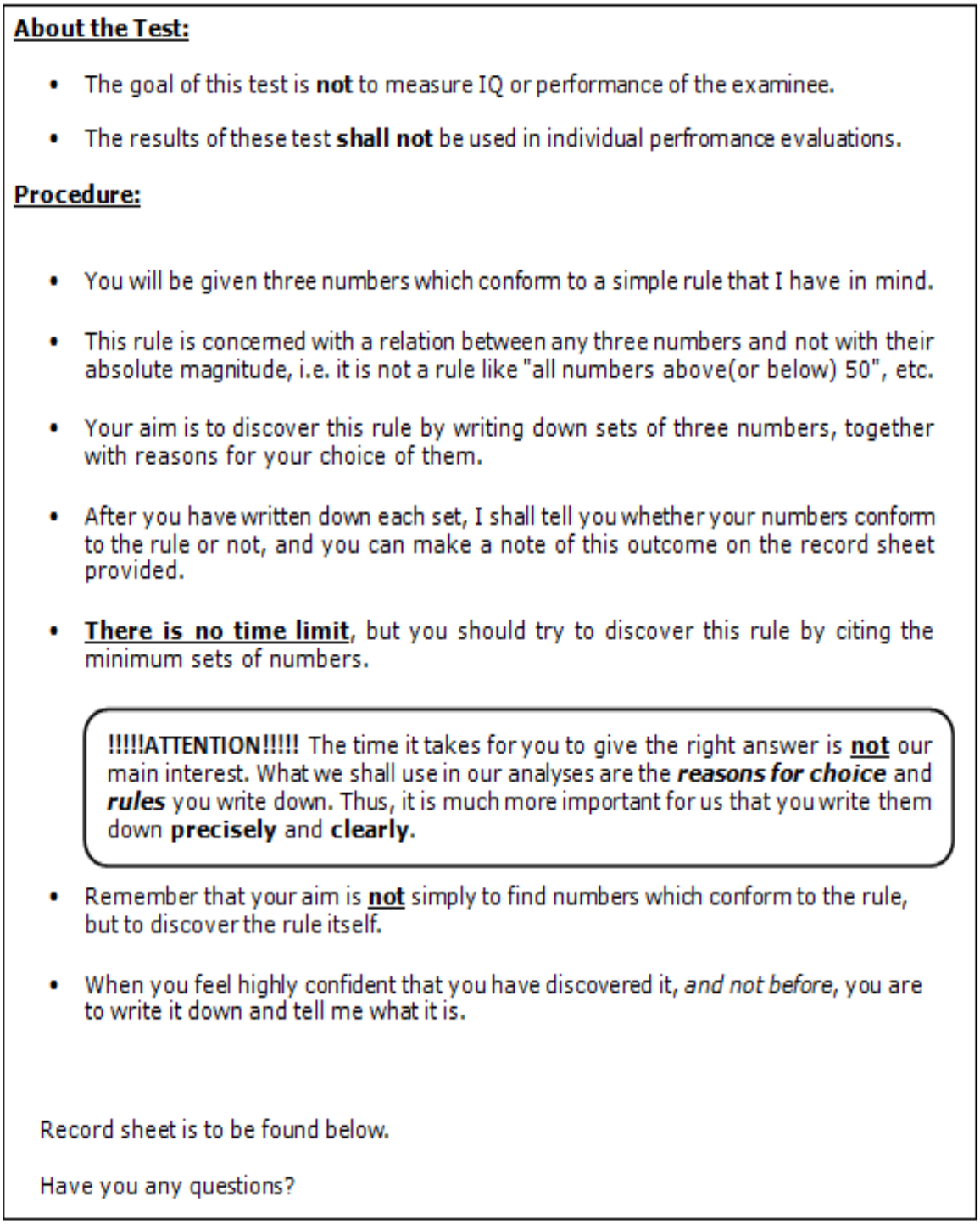
.....

Do you have experience as a professional software tester ? ☐ Yes ☐ No

If your answer to the above question is "Yes", please write down your years of experience:

.....

Figure A.1. Interactive test personal information page.



About the Test:

- The goal of this test is **not** to measure IQ or performance of the examinee.
- The results of these test **shall not** be used in individual perfromance evaluations.

Procedure:

- You will be given three numbers which conform to a simple rule that I have in mind.
- This rule is concerned with a relation between any three numbers and not with their absolute magnitude, i.e. it is not a rule like "all numbers above(or below) 50", etc.
- Your aim is to discover this rule by writing down sets of three numbers, together with reasons for your choice of them.
- After you have written down each set, I shall tell you whether your numbers conform to the rule or not, and you can make a note of this outcome on the record sheet provided.
- **There is no time limit**, but you should try to discover this rule by citing the minimum sets of numbers.

> **!!!!!ATTENTION!!!!!** The time it takes for you to give the right answer is **not** our main interest. What we shall use in our analyses are the ***reasons for choice*** and ***rules*** you write down. Thus, it is much more important for us that you write them down **precisely** and **clearly**.

- Remember that your aim is **not** simply to find numbers which conform to the rule, but to discover the rule itself.
- When you feel highly confident that you have discovered it, *and not before*, you are to write it down and tell me what it is.

Record sheet is to be found below.

Have you any questions?

Figure A.2. Interactive testing procedure information page.

Triples	Reasons for Choice	Conforms to the rule	Does not conform to the rule
2 4 6	----------		

IF YOU NEED EXTRA PAGES, YOU CAN REQUEST FROM THE EXAMINER

Figure A.3. Interactive test record sheet.

APPENDIX B: Written Test (English Version)

There are both English and Turkish versions of the written test. In Appendix B.1 and B.2, we present English versions of the General Written Test and written test with software development/testing theme, respectively.

Written test consists of two parts: The first part is the General Written Test, whereas the second part is the test with software development/testing theme. The questions which the general written test comprises are given in Figure B.2, B.3, B.4, B.5, B.6, B.7, B.8 and B.9. We present the questions of the written test with software development/testing theme in Figure B.10, B.11, B.12, B.13 and B.14.

B.1. General Written Test (English Version)

Information about the General Written Test is given in Figure B.1. Firstly, it is indicated that the test does not aim to measure IQ or performance of the examinee (participant). This statement is also indicated in the information page of the interactive test in Figure A.1. What must be taken into account by the participant regarding the General Written Test is also mentioned on the test's information page. Finally, information page contains the written test procedure. On the last page, the participant is reminded to request the written test with software development/testing from the examiner after (s)he has completed the General Written Test. The last page of the General Written Test is shown in Figure B.9.

B.2. Written Test with Software Development/Testing Theme (English Version)

Written test with software development/testing theme consists of eight questions. Similar to the questions in the General Written Test, more than one choice might be selected as the answer of each question. Test duration is recorded on the first page of the test.

About the Written Test

- The goal of this test is not to measure IQ or performance of the examinee
- The results of this test shall **not** be used in individual performance evaluations.
- Written test consists of two parts which are *General Test* and *Software Test* respectively.

The following must be taken into account about the Written Test:

- During the test, please turn your cell phones off and do not deal with anything else except for the test, so that test outcomes reflect exactly what is we intend to measure.
- Before starting the test, please do not forget to fill in the personal information page.
- After having completed the test, please do not inform anyone who did not take this test about the test's content.

Written Test Procedure

- In each question, there is an explanation about that question, and four choices. For each question, one or more choices may need to be selected to answer the question of interest correctly.
- In order to make a selection, it shall be adequate to cross inside the rectangular shapes where each rectangle represents a choice.
- Any extra selection in addition to the correctly selected ones leads to an **incorrect** answer.
- Statement to be proved or disproved in each question is valid for that question and hence irrelevant with other questions in the test.
- There is **no time constraint** for this test.
- After having finished the *general test*, please submit it to the examiner. The examiner shall give you the *software test*.

Thank you for your interest.

Boğaziçi University, Software Research Laboratory (SoftLab)

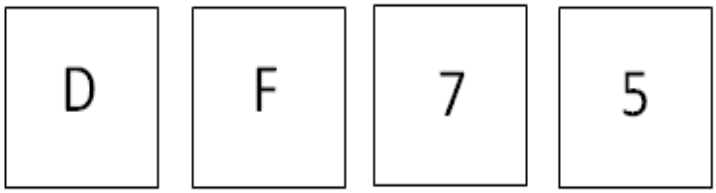
Figure B.1. General written test information page.

PART I: GENERAL WRITTEN TEST

QUESTIONS

1. You are shown a set of four cards each of which has a number on one side and a letter on the other side. The visible faces of the cards show D, F, 7 and 5. Which card(s) should you turn over in order to test the truth of the following rule:

"If a card has a D on one side, then it has a 7 on its other side."



2. Mary has the following opinion about the way John dresses:

"If John wears a blue shirt, then he wears gray trousers"

You want to find out whether Mary's claim is true or false regarding four days during which you observe the way John dresses. Each card below consists of observations about the way John gets dressed. On one side of each card there is information about the color of the shirt John wears, while on the other side there writes the color of the trousers he wears during that day.

Which card(s) should be turned over to find out whether Mary's claim is true or false with respect to the four days during which you observed the way John gets dressed.

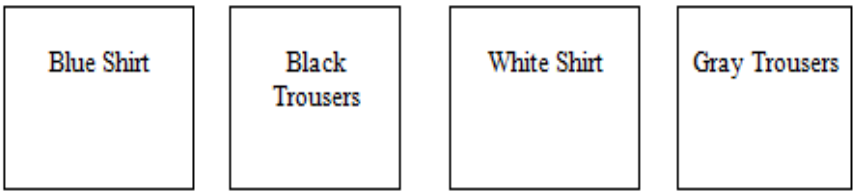


Figure B.2. General written test (first page).

3. In biology laboratories, experiment about viruses are frequently conducted. If viruses contact with researchers' skin, they may cause diseases. Paragon Company, is a pharmaceutical company which conducts experiments about viruses in some of its laboratories, while research and development of new drugs take place in other laboratories of the company. Paragon has the following security rule:

"If you are conducting experiments with viruses, you must wear rubber gloves."

You want to find out whether the above rule is violated or not. The cards below represent four employees of Paragon. On one side of each card, for each of these four employees it writes whether that employee works in laboratories or not, while on the other side of the card it writes if (s)he wears gloves or not.

Which card(s) should be turned over to find out whether any of these four employees violate the security rule stated above or not?

(s)he conducts experiments with viruses	(s)he does not wear gloves	(s)he wears gloves	(s)he does not conduct experiments with viruses

4. You are shown a set of four cards each of which has a number on one side and a letter on the other side. The visible faces of the cards show A, K, 7 and 4. Which card(s) should you turn over in order to test the truth of the following rule:

"If a card has a vowel on one side, then it has an even number on its other side."

A	K	7	4

Figure B.3. General written test (second page).

5. Assume that you are an employee of a company which has the following rule as company policy:

"If an employee gets a pension, that employee must have worked at least ten years."

Each of these cards shown below represents one of your the employees. On one side of each card, there is the total number of years an employee has been working in the company; whereas on the other side there is information about whether that employee gets a pension or not. Which of the following cards would you definitely need to turn over to see if the company policy stated about is violated or not?

worked 12 years	no pension	pension	worked 8 years

6. You are shown a set of four cards each of which has a number on one side and a letter on the other side. The visible faces of the cards show D, F, 7 and 5. Which card(s) should you turn over in order to test the truth of the following rule:

"If a card does not have a D on one side, then it has a 7 on its other side."

D	F	7	5

Figure B.4. General written test (third page).

7. Assume that you are a bouncer at a bar and you must enforce the following rule:

"If someone is drinking beer, then he/she is 21 years old or older."

Each of the four cards given below represents one customer in your bar. One side of each card shows the person's age and the other side shows what that person is drinking. Pick only the cards you definitely need to turn over to see if any of these people are breaking the law and need to be thrown out.

23 years old	Diet Coke	16 years old	Beer

8. You are shown a set of four cards each of which has a number on one side and a letter on the other side. The visible faces of the cards show K, E, 3 and 8. Which card(s) should you turn over in order to test the truth of the following rule:

"If a card does not have an odd number on one side, then it has a consonant on its other side."

K	E	3	8

Figure B.5. General written test (fourth page).

9. Teenagers who don't have their own cars usually end up borrowing their parents' cars. Assume that a father set the following rule for four of his teenager kids:

"If you borrow my car, then you have to fill up the tank with gas."

You are given four cards where each card represents one of these four teenage kids. Which of the following cards would you definitely need to turn over to see if any of these teenagers are breaking their father's rule?

borrowed car	did not borrow car	filled up tank with gas	did not fill up tank with gas

10. You are shown a set of four cards each of which has a number on one side and a letter on the other side. The visible faces of the cards show D, F, 7 and 5. Which card(s) should you turn over in order to test the truth of the following rule:

"If a card does not have a D on one side, then it does not have a 7 on its other side."

D	F	7	5

Figure B.6. General written test (fifth page).

11. Jane says that she prefers train whenever she wants to go to Manchester. Each card below represents four trips of Jane. On one side of each card, there writes the destination points of her trips, while on the other side of the card the transportation medium she used for each trip. In order to check whether Jane really prefers to go to Manchester by train, which cards should be turned over?

"Whenever Jane goes to Manchester, she travels by train."

train	plane	Manchester	Oxford

12. Jack planted a lovely garden with flowers of every color. However, he has not been able to enjoy it, because his flowers are getting harmed by fungi. To get rid of the fungi in a week, Jack's grandmother gave him the following advice:

"If you spray lacana tea on your flowers, you can get rid of the fungi that harm your flowers."

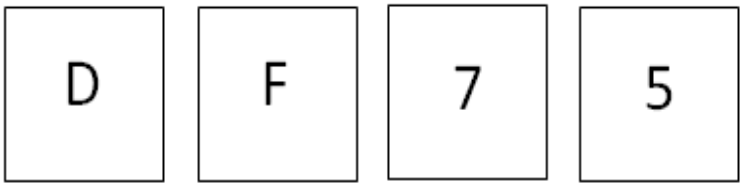
In order to see if that really works, Jack decided to make an experiment and convinced some of his neighbors, who also have complaints about fungi in their gardens, to spray their flowers with lacana tea. After a week he decided to check whether any of the results of this experiment violate his Grandmother's rule. Each of the cards below represents one of the four gardens near Ali's house. One side of the card shows whether lacana tea was sprayed on the flowers in a yard and the other side tells whether fungi still exist or not Which of the following cards would you definitely turn over to see if what happened in any of these gardens violated his Grandmother's rule?

sprayed with lacana tea	not sprayed with lacana tea	No fungi remained in the garden	Fungi still exists in the garden

Figure B.7. General written test (sixth page).

13. You are shown a set of four cards each of which has a number on one side and a letter on the other side. The visible faces of the cards show D, F, 7 and 5. Which card(s) should you turn over in order to test the truth of the following rule:

"If a card has a D on one side, then it does not have a 7 on its other side."



14. Assume that you are an employer and your company has the following rule as company policy:

"If an employee gets a pension, that employee must have worked at least ten years."

Each of these cards shown below represents one of your the employees. On one side of each card, there is the total number of years an employee has been working in the company; whereas on the other side there is information about whether that employee gets a pension or not. Which of the following cards would you definitely need to turn over to see if the company policy stated about is violated or not?

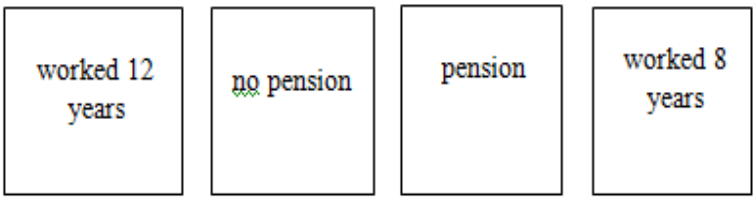
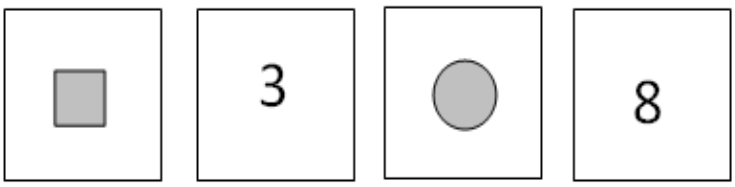


Figure B.8. General written test (seventh page).

15. You are shown a set of four cards each of which has a number on one side and a geometric shape on the other side. The visible faces of the cards show rectangle, 3, square and 8. Which card(s) should you turn over in order to test the truth of the following rule:

"If a card has an even number on one side, then it does not have a square on its other side."



IF YOU HAVE FINISHED **GENERAL WRITTEN TEST** PLEASE SUBMIT IT TO THE EXAMINER AND REQUEST FOR **WRITTEN TEST** WITH **SOFTWARE DEVELOPMENT/TESTING.**

Figure B.9. General written test (eighth page).

For each question you can choose one or more choices.

It is enough to mark these choices with a circle.

Please write down your starting and ending times of the test in the areas provided below.

Thank you.

Starting time: …….. **Ending time: ……..**

---

1. Suppose you want to make sure that a program avoids dereferencing (deleting) a null pointer by always checking before dereferencing. Someone tells you there are four sections of code that needs examination and the following have been determined about those sections:

a) Section A checks whether the pointer is null. The pointer may or may not be dereferenced there.

b) Section B does not check whether the pointer is null. The pointer may or may not be dereferenced there.

c) Section C dereferences the pointer. The pointer may or may not have been checked for nullity.

d) Section D does not dereference the pointer. The pointer may or may not have been checked for nullity.

Which of the sections mentioned above should you definitely check to avoid the situation mentioned above?

Figure B.10. Written Test with software development/testing theme (first page).

2. To evaluate the following hypothesis:

"If an instance's class is Controller, then it has been initialized."

Which of the following needs investigation?

a) An instance of Controller that may or may not be initialized.

b) An instance of a class other than Controller that may or may not be initialized.

c) An initialized instance whose class is unknown.

d) An uninitialized instance whose class is unknown.

3. Suppose you want to make sure that file F is opened successfully, before any attempt to access that file. Thus, you have the following hypothesis:

"If there is an attempt to access file F, then file F has been opened successfully."

You are previously told that there are four sections in your code which need to be investigated carefully and you know the following about those sections:

a) Section A: There is no check whether NULL pointer returns as a result of the attempt to open file F. There may be an access operation (e.g. read, write, etc.) to file F or not.

b) Section B: There is an attempt to read file F. There is also an attempt to open file F before the read operation, but there may or may not be a check to find whether NULL pointer returns as a result of the attempt to open file F.

c) Section C: There is no attempt to access file F. There may or may not be a check to find whether NULL pointer returns as a result of the attempt to open file F.

d) Section D: There is a check whether NULL pointer returns as a result of the attempt to open file F. There may or may not be an access operation (e.g. read, write, etc.) to file F.

Which of the sections mentioned above should you definitely check so that you can find out whether the stated hypothesis about file F is valid or not?

Figure B.11. Written Test with software development/testing theme (second page).

4. Suppose you have a local variable *my_var* of type float and you also have assignment operations in various parts of your program, where you assign the value of *my_var* to other variables. You want to make sure that any variable to which value of *my_var* is assigned is also of type float.

"If value of my_var is assigned to a variable, then type of that variable is float."

You are previously told that there are four sections in your code which need to be investigated carefully and you know the following about those sections:

a) Section A: Value of *my var* is assigned to variable *x*. Type of variable *x* needs to be checked.

b) Section B: Type of variable *z* is integer and it is not known whether *my var* is assigned to *z* or not.

c) Section C: Type of variable *v* is float and it is not known whether *my var* is assigned to *v* or not.

d) Section D: Variable *w* is not assigned to value of *my var* in any part of the code. Type of variable *w* needs to be checked.

Which sections need investigation?

5. Suppose you want to make sure that an array is initialized before accessing one of its elements. You have the following hypothesis:

"If there is an attempt to access an element of an array, then array is initialized before this operation."

You are previously told that there are four sections in your code which need to be investigated carefully:

a) Section A: There is an attempt to access an element of the array. Array may or may not be initialized.

b) Section B: There may be an attempt to access an element of the array. Array is not initialized.

c) Section C: Array may or may not be initialized. There is no attempt to access an element of the array.

d) Section D: Array is initialized. There may or may not be an attempt to access an element of the array.

Which sections need investigation?

Figure B.12. Written Test with software development/testing theme (third page).

6. Suppose you want to make sure that no division by zero is carried out within a function whose returning value may be used. You have the following hypothesis:

"If there is an attempt to use the returning value of the function f(), then no division by zero is carried out within the function."

You are previously told that there are four sections in your code which need to be investigated carefully:

a) Section A: There may or may not be an attempt to use the returning value of the function. No division by zero is carried out within the function.

b) Section B: There is no attempt to use the returning value of the function. Division by zero may or may not be carried out within the function.

c) Section C: There is an attempt to use the returning value of the function. Division by zero may or may not be carried out within the function.

d) Section D: There may be an attempt to use the returning value of the function. Division by zero is carried out.

Which sections need investigation?

7. Suppose you want to make sure that connection to a database is opened successfully before any attempt to access a table of the database. You have the following hypothesis:

"If there is an attempt to access a table of the database, then database connection has been established successfully."

You are previously told that there are four sections in your code which need to be investigated carefully:

a) Section A: There is an attempt to access a table. There may or may not be a check whether error occurs as a result of database connection establishment.

b) Section B: There is no check whether error occurs as a result of database connection establishment.

c) Section C: There is no attempt to access a table. There may or may not be a check whether error occurs as a result of the attempt to access a table.

d) Section D: There is a check whether error occurs as a result of database connection establishment. There may or may not be an access operation to a table.

Which sections need investigation?

Figure B.13. Written Test with software development/testing theme (fourth page).

8. Suppose you want to make sure that an instance of an object is created before accessing a non-static member from a static method. You have the following hypothesis:

"If there is an attempt to access a non-static member from a static method, then an object instance is created before this operation."

You are previously told that there are four sections in your code which need to be investigated carefully:

a) Section A: There is an attempt to access a non-static member from a static method. An object instance may or may not be created.

b) Section B: There is no attempt to access a non-static member from a static method. There may or may not be a creation of an object instance.

c) Section C: There may or may not be an attempt to access a non-static member from a static method. An object instance is created.

d) Section D: No object instance is created. There may be an access operation to a non-static member from a static method.

Which sections need investigation?

Figure B.14. Written Test with software development/testing theme (fifth page).

REFERENCES

1. Wason, P. C., "On the Failure to Eliminate Hypotheses in a Conceptual Task", *Quarterly Journal of Experimental Psychology*, Vol. 12, pp. 129–140, 1960.

2. Poletiek, F., *Hypothesis Testing Behavior (Essays in Cognitive Psychology)*, Psychology Press Ltd., East Sussex, 2001.

3. Harrold, M., "Testing: A Roadmap", *Proceedings of the Conference on the Future of Software Engineering*, pp. 61–72, ACM, New York, NY, 2000.

4. Lessmann, S., B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings", *IEEE Transactions on Software Engineering*, Vol. 34, No. 4, pp. 485–496, 2008.

5. Menzies, T. Z., C. J. Hihn, and K. Lum, "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Transactions on Software Engineering*, Vol. 33, No. 1, pp. 2–13, 2007.

6. Menzies, T., B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of Ceiling Effects in Defect Predictors", *Proceedings of the 3rd Workshop on Predictive Models in Software Engineering*, 2008.

7. Teasley, B., L. M. Leventhal, and D. S. Rohlman, "Positive Test Bias in Software Engineering Professionals: What is Right and Whats Wrong", C.R. Cook, J. S. and J. C. Spohrer (editors), *Empirical Studies of Programmers: Fifth Workshop*, 1993.

8. Teasley, B. F., L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman, "Why Software Testing is Sometimes Ineffective: Two Applied Studies of Positive Test Strategy", *Journal of Applied Psychology*, Vol. 79, pp. 142–155, 1994.

9. McConnell, S., *Code Complete*, Microsoft Press, Redmond, 2004.

10. Munson, J. C. and T. M. Khoshgoftaar, "Detection of Fault Prone Programs", *IEEE Transactions on Software Engineering*, Vol. 18, pp. 423–433, 1992.

11. Khoshgoftaar, T., A. Bullard, Lofton, and K. Gao, "An Application of a Rule-Based Model in Software Quality Classification", *Proceedings of the 6th International Conference on Machine Learning and Applications*, 2007.

12. Khoshgoftaar, T. and E. B. Allen, "Predicting Fault-Prone Software Modules in Embedded Systems with Classification Trees", *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*, 1999.

13. Khoshgoftaar, T. M. and R. M. Szabo, "Using Neural Networks to Predict Software Faults During Testing", *IEEE Transactions on Reliability*, Vol. 45, pp. 456–462, 1996.

14. Nagappan, N., "Toward a Software Testing and Reliability Early Warning Metric Suite", *Proceedings of the 26th International Conference on Software Engineering*, 2004.

15. Bell, O. T. J., R. M. and E. J. Weyuker, "Looking for Bugs in All the Right Places", *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, 2006.

16. Ostrand, T. J., Weyuker, and R. M. Bell, "Where the Bugs are", *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004.

17. Ostrand, T. J., J. Weyuker, Elaine, and R. M. Bell, "Automating Algorithms for the Identification of Fault-Prone Files", *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

18. Tosun, A., B. Turhan, and A. Bener, "Ensemble of Software Defect Predictors: A Case Study", *Proc. of 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM 2008)*, Kaiserslautern, Germany, October 2008.

19. Drummond, C. and R. C. Holte, "C4.5, Class Imbalance and Cost Sensitivity: Why Under-Sampling Beats Over-Sampling", *Proceedings of 2nd Workshop on Learning from Imbalanced Datasets*, 2003.

20. Kamei, Y., A. Monden, T. Matsumoto, and K. Matsumoto, "The Effects of Over and Under-Sampling on Fault Prone Module Detection", *Proceedings of the 1st International Symposium on Empirical Software engineering and Measurement*, 2007.

21. Liu, Y. and T. M. Khoshgoftaar, "Building Decision Tree Software Quality Classification Models Using Genetic Programming", *Proc. of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, IL, USA, July 2003.

22. Khoshgoftaar, T. M., J. Van Hulse, and A. Napolitano, "Supervised Neural Network Modeling: An Empirical Investigation into Learning from Imbalanced Data with Labeling Errors", *IEEE Transactions on Neural Networks*, Vol. 21, No. 5, pp. 813–830, 2010.

23. Turhan, B. and A. Bener, "Analysis of Naive Bayes' Assumptions on Software Fault Data: An Empirical Study", *Data and Knowledge Engineering*, Vol. 68, No. 2, pp. 278–290, 2009.

24. Tosun, A., B. Turhan, and A. Bener, "Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Turkish Telecommunication Industry", *Proc. of 5th International Conference on Predictor Models in Software Engineering (PROMISE 2009)*, ACM. New York, Vancouver, Canada, May 2009.

25. Nagappan, N. and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures", *Proceedings of the 1st Symposium on Empirical Software Engineering and Measurement*, 2007.

26. Jiang, Y., B. Cuki, T. Menzies, and N. Bartlow, "Comparing Design and Code Metrics for Software Quality Prediction", *Proceedings of the* 4^{th} *International Workshop on Predictor Models in Software Engineering*, 2008.

27. Zhao, M., C. Wohlin, N. Ohlsson, and M. Xie, "A Comparison between Software Design and Code Metrics for the Prediction of Software Fault Content", *Information and Software Technology*, Vol. 40, pp. 801–809, 1998.

28. Zimmerman, T. and N. Nagappan, "Predicting Subsystem Failures Using Dependency Graph Complexities", *Proceedings of the 18th IEEE International Symposium on Software Reliability*, 2007.

29. Misirli-Tosun, A., B. Caglayan, A. Mirasky, A. Bener, and N. Ruffolo, "Different Strokes for Different Folks: A Case Study on Software Metrics for Different Defect Categories", *Proceedings of the 2nd Workshop on Emerging Trends in Software Metrics*, 2011.

30. Turhan, B., G. Kocak, and A. Bener, "Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework", *Proc. of. 34th Intl. EUROMICRO Software*

Engineering and Advanced Applications Conference (EUROMICRO 2008), Parma, Italy, September 2008.

31. Hannay, J. E., E. Arisholm, H. Engvik, and D. I. K. Sjoberg, "Effects of Personality on Paired Programming", *IEEE Transactions on Software Engineering*, Vol. 36, No. 1, 2010.

32. Acuna, S. T., M. Gomez, and N. Juristo, "How Do Personality, Team Processes and Task Characteristics Relate to Job Satisfaction and Software Quality?", *Information and Software Technology*, Vol. 51, No. 3, pp. 627–639, March 2009.

33. Stacy, W. and J. MacMillan, "Cognitive Bias in Software Engineering", *Communication of the ACM*, Vol. 38, No. 6, pp. 57–63, 1995.

34. Parsons, J. and C. Saunders, "Cognitive Heuristics in Software Engineering: Applying and Extending Anchoring and Adjustment to Artifact Reuse", *IEEE Transactions on Software Engineering*, Vol. 30, No. 12, pp. 873–888, 2004.

35. Mair, C. and M. Shepperd, "Human Judgement and Software Metrics: Vision for the Future", *Proceeding of the 2nd international workshop on Emerging trends in software metrics (WETSoM '11)*, New York, NY, USA, 2011.

36. Jorgensen, M., "Identification of More Risks Can Lead to Increased Over-Optimism and Over-Confidence in Software Development Effort Estimates", *Journal of Information and Software Technology*, Vol. 52, pp. 506–516, 2010.

37. Jorgensen, M., "Estimation on Software Development Work Effort: Evidence on Expert Judgement and Formal Models", *International Journal of Forecasting*, Vol. 23, pp. 449–462, 2007.

38. Graves, T. L., A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History", *IEEE Transactions on Software Engineering*, Vol. 26, pp. 653–661, 2000.

39. Weyuker, O. T. J., Elaine J. and R. M. Bell, "Do too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models", *Empirical Software Engineering*, Vol. 13, No. 5, October 2008.

40. Mockus, A. and D. M. Weiss, "Predicting Risk of Software Changes", *Bell Labs Technical Journal*, pp. 169–180, 2000.

41. Weyuker, O. T. J., E.J. and R. M. Bell, "Using Developer Information as a Factor for Fault Prediction", *Proceedings of the 1st International Workshop on Predictor Models in Software Engineering*, 2007.

42. Ostrand, T. J., E. J. Weyuker, and R. M. Bell, "Programmer-Based Fault Prediction", *Proceedings of the 3rd Workshop on Predictor Models in Software Engineering*, 2010.

43. Meneely, A., L. Williams, W. Snipes, and J. Osborne, "Predicting Failures with Developer Networks and Social Network Analysis", *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

44. Pinzger, M., N. Nagappan, and B. Murphy, "Can Developer-Module Networks Predict Failures?", *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

45. Bird, C., N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting It All Together: Using Socio-Technical Networks to Predict Failures", *Proceedings of the 17th International Symposium on Software Reliability Engineering*, 2009.

46. Roweis, S., "EM Algorithms for PCA and SPCA", *in Advances in Neural Information Processing Systems*, pp. 626–632, MIT Press, 1998.

47. Kahneman, D., P. Slovic, and A. Tversky, *Judgment under Uncertainty: Heuristics and Biases*, Cambridge University Press, New York, New York, 1982.

48. Poletiek, F., "Paradoxes of Falsification", *Quarterly Journal of Experimental Psychology*, Vol. 49A, pp. 447–462, 1996.

49. McDonald, J., "Is Strong Inference Superior to Simple Inference?", *Syntheses*, Vol. 92, pp. 261–282, 1992.

50. Evans, J. S. B. T. and J. S. Lynch, "Matching Bias in the Selection Task", *British Journal of Psychology*, Vol. 64, No. 3, pp. 391–397, August 1973.

51. Reich, S. S. and P. Ruth, "Wason's Selection Task: Verification, Falsification and Matching", *British Journal of Psychology*, Vol. 73, pp. 395–405, 1982.

52. Wason, P. C. and D. Shapiro, "Natural and Contrieved Experience in a Reasoning Problem", *Quarterly Journal of Experimental Psychology*, Vol. 23, pp. 63–71, 1971.

53. Griggs, R. A. and J. R. Cox, "The Elusive Thematic Materials Effect in Wasons Selection Task", *British Journal of Psychology*, Vol. 73, pp. 407–420, 1982.

54. Rumelhart, D. E., *Schemata: The Building Blocks of Cognition, In "Theoretical Issues in Reading Comprehension"*, Lawrance Erlbaum Associates Inc., Hillsdale, NJ, 1980.

55. Cosmides, L., "The Logic of Social Exchange: Has Natural Selection Shaped How Humans Reason? Studies with Wasons Selection Task", *Cognition*, Vol. 31, pp. 187–276, 1989.

56. Manktelow, K. I. and D. E. Over, *Inference and Understanding: A Philosophical and Psychological Perspective*, London, 1990.

57. Cheng, P. W. and K. J. Holyoak, "Pragmatic Reasoning Schemas", *Cognitive Psychology*, Vol. 17, pp. 391–416, 1985.

58. Johnson-Laird, P. and J. M. Tridgell, "When Negation is Easier than Affirmation", *Quarterly Journal of Experimental Psychology*, Vol. 24, pp. 87–91, 1972.

59. Manktelow, K. I. and J. S. B. T. Evans, "Facilitation of Reasoning by Realism: Effect or Non-Effect?", *British Journal of Psychology*, Vol. 70, pp. 477–488, 1979.

60. Griggs, R. A., *The Role of Problem Content in the Selection Task and in the THOG Problem. In "Thinking and Reasoning: Psychological Approaches"*, London, 1983.

61. Cox, J. R. and R. A. Griggs, "The Effects of Experience on Performance in Wasons Selection Task", *Memory and Cognition*, Vol. 10, pp. 496–502, 1982.

62. Evans, J. S. B. T., S. E. Newstead, and R. M. J. Byrne, *Human Reasoning: The Psychology of Deduction*, Lawrence Erlbaum Associates Publishers, East Sussex, 1993.

63. Hilgard, E. R., "A Summary and Evaluation of Alternative Procedures for the Construction of Vincent Curves", *Psychology Bulletin*, Vol. 35, pp. 282–297, 1938.

64. Alpaydin, E., *Introduction to Machine Learning*, The MIT Press, Cambridge, 2004.

65. Little, R. J. A. and D. B. Rubin, *Statistical Analysis with Missing Data*, John Wiley & Sons Inc., New Jersey, 2002.

66. Bell, R., Y. Koren, and C. Volinsky, "Modeling Relationships at Multiple Scales to Improve Accuracy of Large Recommender Systems", *ACM Int. Conference on Knowledge Discovery and Data Mining (KDD'07)*, 2007.

67. Bell, R. M., J. Bennet, Y. Koren, and C. Volinsky, "The Million Dollar Programming Prize", *IEEE Spectrum*, pp. 28–33, May 2009.

68. Bell, R. M. and Y. Koren, "Lessons from the Netflix Prize Challenge", *SIGKDD Explorations*, Vol. 9, No. 2, pp. 75–79, 2007.

69. Kocaguneli, E., A. Tosun, A. Bener, B. Turhan, and B. Caglayan, "Prest: An intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool", *21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Boston, USA, July 2009.

70. Montgomery, D. C., *Design and Analysis of Experiments*, John Wiley and Sons, Inc., New Jersey, 2009.

71. Gravetter, F. J. and L. B. Wallnau, *Statistics for Behavioral Sciences*, Wadsworrth Cengage Learning, Belmont, California, 2009.

72. Calikli, G., A. Bener, and B. Arslan, "An Analysis of the Effects of Company Culture, Education and Experience on Confirmation Bias Levels of Software Developers and Testers", *Proceedings of 32nd International Conference on Software Engineering*, 2010.

73. Calikli, G., B. Arslan, and A. Bener, "Confirmation Bias in Software Development and Testing: An Analysis of the Effects of Company Size , Experience and Reasoning Skills", *Proceedings of the 22nd Annual Psychology of Programming Interest Group Workshop*, 2010.

74. Calikli, G. and A. Bener, "Empirical Analyses Factors Affecting Confirmation Bias and the Effects of Confirmation Bias on Software Developer/Tester Performance",

Proceedings of 5th International Workshop on Predictor Models in Software Engineering, 2010.

75. Cook, D., Thomas and D. T. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Setting*, Houghton Mifflin Company, Boston, Boston, 1979.

76. Cohen, J., *Statistical Power Analysis for the Behavioral Sciences*, Lawrance Erlbaum Associates, Publishers, New Jersey, 1988.

77. Cohen, J., P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*, Lawrance Erlbaum Associates Publishers, New Jersey, 2003.

78. Cohen, J., "A Power Primer", *Psychology Bulletin*, Vol. 112:1, pp. 155–159, 1992.

79. Halstead, M., *Elements of Software Science*, Elsevier, 1977.

80. McCabe, T., "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. 2, pp. 308–320, 1976.

81. Popper, K., *The Logic of Scientific Discovery*, Routledge Classics, London, 1959.

82. Johnson-Laird, P. and P. C. Wason (editors), *Thinking: Readings in Cognitive Science*, Cambridge University Press, London, 1977.