

UNDERSTANDING HOW DEVELOPERS WORK ON CHANGE TASKS USING
INTERACTION HISTORY AND EYE GAZE DATA

By

Ahraz Husain

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

December 2015

UNDERSTANDING HOW DEVELOPERS WORK ON CHANGE TASKS USING
INTERACTION HISTORY AND EYE GAZE DATA

Ahraz Husain

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

Ahraz Husain, Student Date

Approvals:

Bonita Sharif, Thesis Advisor Date

Yong Zhang, Committee Member Date

Feng Yu, Committee Member Date

Sal Sanders, Associate Dean of Graduate Studies Date

Abstract

Developers spend a majority of their efforts searching and navigating code with the retention and management of context being a considerable challenge to their productivity. We aim to explore the contextual patterns followed by software developers while working on change tasks such as bug fixes. So far, only a few studies have been undertaken towards their investigation and the development of methods to make software development more efficient. Recently, eye tracking has been used extensively to observe system usability and advertisement placements in applications and on the web, but not much research has been done on context management using this technology in software engineering and how developers work.

In this thesis, we analyze an existing dataset of eye tracking and interaction history that were collected simultaneously in a previous study. We look into exploring navigational patterns of developers while they solve tasks. Our goal is to use this dataset to determine if we can perform prediction and recommendations solely based on eye gaze patterns. In order to do this, we conduct three experiments on Microsoft Azure on developer expertise recommendation and class recommendation for developers using only eye tracking data. Our results are quite promising. We find that eye tracking data can be used to predict expertise of developers with 85% accuracy. It is further able to recommend classes with good performance (a normalized discounted cumulative gain, NDCG ranging between 0.85 and 0.88). These findings are discussed with a view to

designing systems that can adapt to the individual user in real time and make intelligent adaptive suggestions while developers work.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Bonita Sharif, whose expertise, understanding, and patience, added considerably to my graduate experience. I appreciate her vast knowledge and skills in many areas and her guidance, which has been a key motivational factor towards completion of my research. Further, I would like to thank the esteemed members of my committee, Dr. Yong Zhang and Dr. Feng Yu for the assistance they provided at all levels of the research project and Dr. Alina Lazar for her expert guidance in the field of Data Science.

A very special thanks goes out to my family without whose motivation and encouragement I would not have considered a graduate career. They are among the many people who truly made a difference in my life.

In conclusion, I recognize that this research would not have been possible without the financial assistance from the Department of Computer Science and the STEM College at YSU. I express my gratitude to them for supporting me during my graduate studies.

TABLE OF CONTENTS

LIST OF FIGURES	VIII
LIST OF TABLES	IX
CHAPTER 1 INTRODUCTION.....	1
1.1 Motivation.....	1
1.2 Contributions.....	3
1.1 Research Questions.....	3
1.2 Organization.....	5
CHAPTER 2 BACKGROUND AND RELATED WORK.....	6
2.1 Eye Tracking Basics and Terminology.....	6
2.2 Program Comprehension	7
2.3 Tracing Developers Interaction.....	8
2.4 Eye tracking Studies	10
2.5 Discussion.....	12
CHAPTER 3 EXPERIMENTS	14
3.1 Dataset Used	14
3.2 Subject System and Tasks.....	15
3.3 Participants.....	16
3.4 Results from Previous Work.....	18
3.5 Data Snapshot	19

3.6	Experiments Overview.....	20
3.7	Data Pre-Processing.....	21
3.7.1	Data Congregation	21
3.7.2	Data Migration	22
3.8	Machine Learning Model Selection.....	23
3.8.1	Expertise Recommender Experiment	23
3.8.2	Class Recommender Experiments	24
3.9	Feature Engineering.....	25
3.10	Experiment Instrumentation.....	30
CHAPTER 4 RESULTS AND ANALYSES		37
4.1	RQ1: Can we predict the expertise of a developer based on gaze data and interactions with the IDE for a specific task?	37
4.2	RQ2: Can we use the class-based eye gaze and interaction of a developer to intelligently suggest classes of interest?	39
4.3	RQ3: Can we also use pupil dilation/contraction information alongside class-based gaze interaction to intelligently suggest classes of interest?.....	41
4.4	Threats to Validity	43
4.5	Discussion of Results.....	45
CHAPTER 5 CONCLUSIONS AND FUTURE WORK.....		47

LIST OF FIGURES

Figure 1: Subjects' Relative Experience Levels between Amateurs and Professionals ..	17
Figure 2: Sample iTrace XML Snippet.....	20
Figure 3: Sample Mylyn XML Snippet	20
Figure 4: A Subset of the Cleaned-up Master Dataset.....	23
Figure 5: Classifiers Generated by ML Models.....	24
Figure 6: Simple K-Means++ Clustering Result on the iTrace Dataset	25
Figure 7: Sample of "Rating_quantized" for Exp II	27
Figure 8: The "Rating_quantized" Feature Statistics for Exp III	28
Figure 9: "Rating_quantized" distribution bar plot for Exp III	28
Figure 10: Mylyn Dataset used towards the Recommender System in Exp V	29
Figure 11: Exp I: Data Cleanup and Feature Type Updating for Exp I.....	33
Figure 12: Training, Scoring and Evaluation in Exp I.....	34
Figure 13: Training, Scoring and Evaluation in Exp II and III.....	35
Figure 14: Preprocessing Data for Exp III.....	36
Figure 15: Model for Expertise Prediction for TaskID 4 using iTrace logs	45
Figure 16: Model for Expertise Prediction for TaskID 4 using Mylyn logs.....	46

LIST OF TABLES

Table 1: Study Tasks conducted by Kevic et al. in [1].	16
Table 2: Overview of Experiments Conducted	21
Table 3: Feature Importance Scores for Exp I	26
Table 4: The Initial “Rating” vs. “Rating_quantized” Feature Statistics for Exp II	26
Table 5: Feature Importance Scores for Expertise Prediction in Exp IV	29
Table 6: Subset used for Expertise Prediction	33
Table 7: Discovered and Used Parameters for Exp I	37
Table 8: Model Evaluation for Exp I	38
Table 9: Exp I: Model Cross Validation	38
Table 10: Discovered and Used Parameters for Exp II	39
Table 11: Recommender Evaluation for Exp II	40
Table 12: Recommender Pool Sample for Task 2 in Exp II	40
Table 13: Recommender Evaluation for Exp III	42
Table 14: Recommender Pool Sample for Task 2 in Exp III	42
Table 15: Recommender Pool Exp II vs. III for User 2 and Task 2.	42

CHAPTER 1

INTRODUCTION

Software is inherently quite complex. In this work, we aim to understand and evaluate the contextual patterns followed by software developers towards bug fixing and to use them towards the exploration of technologies and methods to make this process more efficient.

1.1 Motivation

Software developers spend a majority of their time working on change tasks, such as bug fixes or feature additions. They use complex Integrated Development Environments (IDE) such as MS Visual Studio or Eclipse making them more productive. The challenges faced by all developers often include searching and navigating to relevant blocks of code, retention and management of context upon change of tasks and across a long timeframe and wastage of time and effort on non-productive contemplation among other things. However, only a few studies have been undertaken towards their investigation and the development of methods to make software development more efficient. This is mainly due to the significant effort required towards acquiring the time of professional software developers to participate, the lack of any detailed, automatic and efficient method of capturing, transcribing and coding the long sessions of these developers' working environments and the fact that most context management systems are strictly event and/or interaction based.

In recent years, eye tracking has been used extensively to observe system usability and advertisement placements in applications and on the web, not much research has been done on context management using this technology. In previous work, Kevic et al. conducted a study [1] with 22 software developers both from industry and academia. They recorded the eye movements of these developers along with the interaction history data simultaneously. Interaction history data consists of mouse clicks and keyboard events including searches made by the developer. Eye tracking data consists of where the developer is looking and for how long. The elements captured by eye tracking were source code elements such as method signatures, method calls, variable declarations, and variable usage among others. The eye tracking data was gathered using iTrace [2] and the interaction history data was collected using a well-known interaction history manager, Mylyn [3]. In this work, we use this first-of-a-kind dataset for predictions and recommendations.

This dataset is unique in that it is the first eye tracking study that is conducted on a large open-source code base that supports scrolling and viewing multiple files without losing context of what the developer is looking at. This was made possible by the Eclipse plugin iTrace [2] developed at the Software Engineering Research and Empirical Studies Lab at YSU, that interfaces with the Java Abstract Syntax Tree within Eclipse to automatically map the eye gaze to a source code element such as a variable name or a method call.

1.2 Contributions

We explore how eye gaze features such as time spent looking at source code elements and the pupil diameter when the user is looking at these elements. These features can be used to record probabilistic impressionable context that may transform into actions. We further try to place how our findings can be used in the evolution of development tools towards making the developers more efficient while working on change tasks.

We use machine-learning algorithms to predict expertise of a developer based on his/her eye tracking sessions. In addition, we also use machine learning algorithms to recommend relevant classes to a developer based on his/her eye gaze during the session. We find that predicting the expertise of a developer based on where the eye is focusing is feasible and works well 85% of the time. This research seeks to lay the groundwork for future research based on knowledge discovery and prediction algorithms.

1.1 Research Questions

Of the range of questions that arise, we shall focus our research to seek answers towards the following questions. All these questions relate to the context of fixing bugs.

- RQ1: Can we predict the expertise of a developer based on gaze data and interactions with the IDE for a specific task?
- RQ2: Can we use the class-based eye gaze and interaction of a developer to intelligently suggest classes of interest?

- RQ3: Can we also use pupil dilation/contraction information alongside class-based gaze interaction to intelligently suggest classes of interest?

The first research question, RQ1 deals with prediction while RQ2 and RQ3 deal with recommendations. In particular, we are interested in predicting expertise of developers solely based on their eye gaze activity. The expertise prediction is useful to determine if an eye gaze pattern belongs to an expert or a novice. Such a prediction is useful during job interviews where the interviewer could possibly be eye tracking the candidate to determine if they are looking at relevant parts of the code. But more importantly, it could be used to suggest different recommendations to experts compared to novices in an online setting while they are programming in the IDE. This transitions into RQ2 and RQ3 that deal with recommending classes of interest to developers based on how they have been looking at classes so far. Hence, the result of RQ1 could be used as input to RQ2 and/or RQ3. However, in this thesis, we deal with them separately.

The main difference between RQ2 and RQ3 is that in RQ3 we use an additional eye gaze feature, i.e., pupil dilation. The pupil diameter is a measure that the eye tracker reports and is listed in millimeters for both the right eye and the left eye. The reason we chose pupil deviation based on pupil dilation and pupil contractions as a possible feature was because research in the past has shown that systematically chosen stimuli significantly affected the subjects' physiological reactions and subjective experiences. This consequently affects the pupil dilation/contraction and so it is possible to use pupil size variation as a computer input signal [4][5].

1.2 Organization

The thesis is organized as follows. The next chapter gives a brief introduction to eye tracking and related work. Chapter 3 presents the details of the experimental setup and an overview of all experiments conducted. Chapter 4 discusses observations and results and finally, Chapter 5 concludes the thesis and presents future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents an overview of existing work in program comprehension and eye tracking. We first start with giving some terminology on eye tracking.

2.1 Eye Tracking Basics and Terminology

Eye trackers have been around since the 1970's. Due to the affordability and smaller form factor, they are becoming popular in different areas of research. They capture the eye movements of the user on a screen/area based on how infrared light reflects back after it hits the retina. After an initial setup and calibration, the eye tracking system is able to calculate the point of interest with very good precision. The system usually consists of hardware and software. The output is a time series with several parameters such as validity codes for both eyes, the (x, y) coordinate of where the gaze was focused, and pupil dilation among others. All the parameters are discussed in Section 3.7. The granularity of modern eye trackers usually lies in the millisecond range. For the study that collected the data used in this thesis, the Tobii X60 eyetracker [6] was used that does not require the developer to wear any gear. Tobii X60 has an on-screen accuracy of 0.5 degrees and spits out 60 eye gaze locations per second. To accommodate for this and still have line-level accuracy of the eye gaze data, the font size should be set larger (usually to 20 points) for source code within Eclipse. We ran several tests to validate the accuracy of the collected data.

2.2 Program Comprehension

Current research into software comprehension models, suggests that programmers attempt to understand code using the somewhat clichéd taxonomy of ‘bottom-up comprehension’, ‘top-down comprehension’, and various combinations of these two processes [7]. Bottom-up comprehension models, propose that as source code is read, abstract concepts are formed by chunking together low-level information [8] [9]. In other words, reading the code and then mentally chunking or grouping these lines of code into higher-level abstractions builds understanding from the bottom up. The bottom-up model of software comprehension primarily addresses situations where the programmer is unfamiliar with the domain. Several ‘top-down’ models of software comprehension have been proposed to address the alternative situation, where the programmer has had some previous domain exposure. Essentially, these top-down models of comprehension suggest that the programmer utilizes knowledge about the domain to build a set of expectations that are mapped on to the source code [7][10].

Over time, several cognitive models of program comprehension have been suggested and researched all of which attempt to explain how a software developer goes about the process of understanding and navigating code. However, research has suggested that there is no one ‘all encompassing’ cognitive model that can explain the behavior of all the developers [11] and that it is more likely that they will swap between various models as per the need [12]. Further, eye tracking can provide us significant insight into capturing traceability links between different artifacts [13] and assistance towards software development [14] and computing education [15] among others.

2.3 Tracing Developers Interaction

For decades, researches have unceasingly aimed at understanding the way developers apprehend code. Initial researches aimed to do this via interviews and questionnaires after the participants had interacted with the code [7][18]. With advancements in processing and storage capabilities and reducing costs [15], Altmann analyzed a ten minute interval of an expert programmer performing a task and used computational simulation to study the near-term memory [19]. However, the cost of hand-coding developers' actions is very high, which have in turn led to only a limited number of studies providing detailed insights on the developers' behaviors.

With the availability of more advanced technology, newer ways of automatically recording such data have been brought about. Two of the most popular ones being: User Interaction Monitoring [20][21] and Biometric Sensing [22][23]. Based on the captured monitoring logs based on the interaction with code elements (also known as Areas of interest or AOIs') that the developer interacted with, we can use this knowledge for defect prediction [24] and towards determining a developers' cognitive abilities in real time [25] among other uses. Additional information such as navigation patterns [26][27][28] within an IDE further helps in defect prediction [24]. Even the Eclipse team themselves undertook a major data collection project called the Usage Data Collector [29] that, at its peak, collected data from thousands of developers using Eclipse. Overall, the automatic monitoring of user interactions was able to significantly reduce the cost for certain empirical studies [30]. Along with this, a popular context management tool called Mylyn [3] was developed. Mylyn is a plugin for the Eclipse IDE, which further

strengthened this approach towards research. However, these studies are limited to the granularity and detail of the monitoring approach. In case of user interaction monitoring, the granularity is predominately at the method or class level and detailed information such as the time a developer spends reading a code element or when the developer is not looking at the screen, is missing and this not only makes it more difficult to fully understand the developers' traces in detail but also effects the precision of such studies.

In recent times, the interaction of developers with IDE's have been a good source of learning and has led to the development of Mylyn. Mylyn is an open source execution of the Task-Focused Interface; it's additionally an application lifecycle management (ALM) structure for Eclipse [3]. It provides an interface to developers and can help proficiently with a wide range of errands, (for example, bugs, issue reports or new elements). Mylyn can incorporate with archives, for example, Bugzilla, Trac, Redmine, Mantis, JIRA, Unfuddle and GitHub. It focuses on improving productivity by reducing searching, scrolling, and navigation. By making task context explicit, Mylyn is also meant to facilitate multitasking, planning, reusing past efforts, and sharing expertise. The log files generated by Mylyn have been used to explore the interaction patterns and methods between developers and IDEs.

Apart from Mylyn, a more conventional method to capture interactions is to obtain a video screen capture of tasks performed by developers. This is a long and tedious process since, the videos then have to be manually studied frame by frame and flags need to be noted down. This makes the process manual and hence, is prone largely to human errors and inconsistencies. People have in the recent past, attempted to develop and use

video scraping tools to obtain the interactions but they are heavily limited by the platform in question and hence, cannot be used on a diversity of environments.

Another straightforward but tedious approach was followed Ko et al. [16]. In this study, the authors screen captured ten developers' desktops while they worked on five distinct tasks on a small program and then hand-coded and analyzed each of the 70 minute sessions. Similarly, in a study on developers performing more realistic change tasks, Fritz et al. [17] used a similar technique and manually transcribed and coded the screen-captured videos of all participants.

2.4 Eye tracking Studies

Alongside to the IDE instrumentation efforts, researchers in the software development domain have also started to take advantage of the maturing of biometric sensors. Most of this research focuses on eye-tracking [22], while only few studies have been conducted so far that also use other signals, such as an fMRI to identify brain activation patterns for small comprehension tasks. Sharif et al. conducted an eye tracking study to determine the effectiveness and efficiency of identifier styles such as camel case and underscore [31]. A combination of eye-tracking, EDA, and EEG sensors to measure aspects such as task difficulty, developers' emotions and progress, or interruptibility [32][33] have also been done but cannot be completely generalized. Eye tracking ensures not only automatic capturing where a developer is looking (eye gaze) but also has helped researchers gain deeper insights into developers' code comprehension. Walters et al.

conducted an eye tracking study to automatically capture traceability links between bug reports and source code [13].

Crosby et al. carried out one of the first studies in program comprehension using eye tracking. It was discovered that experts and novices vary in the way they looked at English and Pascal versions of an algorithm [34]. Ever since, various researchers have used visual monitoring methods to examine the impact of developers' eye gaze on their comprehension of different kinds of representations and visualizations including but not limited to UML diagrams [35][32], graphs [25], 3D visualizations [36], design pattern layouts [37], programming languages [38], and identifier styles [31][39]. Some researchers have also discovered that developers usually read the entire source code first to get an overview though this study was limited to small code snippets [40]. Other researches focused on the examination of the different strategies novice and expert developers employ towards program comprehension and debugging [41][42], as well as the developers time consuming areas when reading a method in order to help devise a better method summarization technique [43]. In addition, researchers have also attempted to explore the potential of eye tracking towards the detection of software traceability links [13][44][45]. Finally, Conati et al. in their research [25] attempted to provide a detailed analysis of different eye gaze feature sets, as well as over-time accuracies and inferred that these predictions are significantly better than a baseline classifier even during the early stages of visualization usage and therefore can adapt to the individual user in real time. Their work was not in the field of software engineering.

2.5 Discussion

In the previous section, a short description of relevant studies using eye tracking in software engineering is given. This is not an exhaustive list. Since 2006, there has been a surge in the number of software engineering papers using eye tracking. To date, there are about 35 papers in the area. This trend is expected to continue.

Most of the aforementioned studies lay a good basis for further research but are either limited to very small toy applications or single-page code tasks that do not involve scrolling. Furthermore, the link between the eye gaze (e.g. a developer looking at pixel at x, y coordinate on the screen) to the elements in source code (i.e., method class within a method) had to be done manually which raises human error, accuracy and precision concerns. The study conducted by Kevic et al. [1] changed that. It made use of an eye tracking plugin namely, iTrace that was capable of recording eye gaze on longer documents and not limited to a single screen of text.

Through our research, we aim to explore how change tasks are handled by developers and how we can assist/streamline this process for them in real time. We aim to develop efficient real-time recommender systems such as used by Netflix to recommend/customize their users' dashboard [46][47][48]. This should save developers significant time wasted in scanning and finding areas of interest [49][50].

To maximize productivity and to ensure the quality of collected data, we used our in-house developed plugin for Eclipse called iTrace and bundled it with the data collected by Mylyn. This helped us collect good quality data, while reducing the need of manual mapping. This further helped us overcome the single page code task limitation of

previous studies, allowing for change tasks on a realistic-sized code base with developers being able to naturally scroll and switch editor windows.

We see the experiments conducted in this thesis as a first step towards showing that using eye tracking data from a bug fixing session can be used to predict and recommend relevant source code items to developers. The next section introduces the experiments.

CHAPTER 3

EXPERIMENTS

This chapter presents the details of the various experiments we conducted on Microsoft Azure that use machine learning algorithms.

3.1 Dataset Used

Good quality data is the biggest need of data science [51]. We now explain how the dataset was collected by Kevic et al. [1]. In order to ensure we had high quality reliable data, we used data collected across 22 participants. So, as to get the best accuracy possible, calibration were done before every study. The participants in Kevic's study [1] were asked to fill out a background questionnaire on their previous experiences with programming. The tasks recorded had the goal to fix the specified bugs based on user generated bug reports. All participants were seated in front of a 24- inch LCD monitor. The total time spent by each participant was one hour spread across three tasks (or bugs to fix). Further, the participants were requested to type their answer (i.e. the class(es)/method(s)/attribute(s) where they might fix/find the bug) in a text file. The source code in question belonged to JabRef [52] - an open source bibliography reference manager the details of which are discussed in Section 3.2. Each participant was able to make any necessary edits to this code and run it. They were also able to switch back and forth between the Eclipse IDE and the JabRef application. The eye tracking data was only collected when Eclipse was in focus.

3.2 Subject System and Tasks

JabRef (<http://jabref.sourceforge.net/>) was the subject system used in the study conducted in [1]. JabRef is a graphical application for managing bibliographic databases that uses the standard LaTeX bibliographic format BibTeX, and can also import and export many other formats. JabRef is an open source, Java based system available on SourceForge [52] and consists of approximately 38 KLOC spread across 311 files. The version of JabRef was 1.8.1, release date 09/16/2005. To have realistic change tasks, the tasks were directly taken from the bug descriptions submitted to JabRef on SourceForge. Information about each task is provided in Table 1. These are from [1] and are listed here for easy reference.

All of these change tasks represent actual JabRef tasks that were reported by someone on SourceForge and that were eventually fixed in a later JabRef release. The tasks were randomly selected from a list of closed bug reports with varied difficulty as determined by the scope of the solution implemented in the repository. Three change tasks were performed by all participants. This is a reasonable number of tasks without causing fatigue in the one hour of the study. A time limit of 20 minutes was placed for each task so that participants would work on all three tasks during the one-hour study. To familiarize participants with the process, each participant was also given a sample task before starting with the three main tasks for which we did not analyze the tracked data. The task order of the three tasks was randomly chosen for each participant.

Table 1: Study Tasks conducted by Kevic et al. in [1].

ID	Bug ID	Date Submitted	Title	Scope of Solution in Repository
2	1436014	02/21/2006	No comma added to separate keywords	Multiple classes: EntryEditor, GroupDialog, FieldContentSelector, JabRefFrame
3	1594123	11/10/2006	Failure to import big numbers	Single method: BibtexParser.parseFieldContent
4	1489454	05/16/2006	Acrobat Launch fails on Win98	Single method: Util.openExternalViewer

3.3 Participants

We describe the participants of the study conducted in [1]. Kevic et al. gathered two sets of participants: twelve professional developers working at ABB Inc. that spend most of their time developing and debugging production software, and ten undergraduate and graduate computer science students. Participants were recruited through personal contacts and via a recruiting email. All participants were compensated with a gift card for their participation. All professional developers reported having more than five years of programming experience. Seven of the twelve reported having more than five years of experience programming in Java, while the other five reported having about one year of Java programming experience. Nine of the twelve professional participants also rated their bug fixing skills as above average or excellent. With respect to IDE usage, four of the twelve stated that they mainly use Visual Studio for work purposes and that they were not familiar with the Eclipse IDE, and one participant commented on mainly being a vim and command line user. Two of the professional developers were female and ten were male.

Among the ten student participants, one participant had more than five years of programming experience, five students had between three and five years programming experience, and four of them had less than two years programming experience. Three of the students had between three and five years of Java programming experience, and seven students had less than two years. Three of the ten students rated their bug fixing skills as above average, and seven rated them as average. All but one student stated that they were familiar with the Eclipse IDE. There was one female and nine male among the student population.

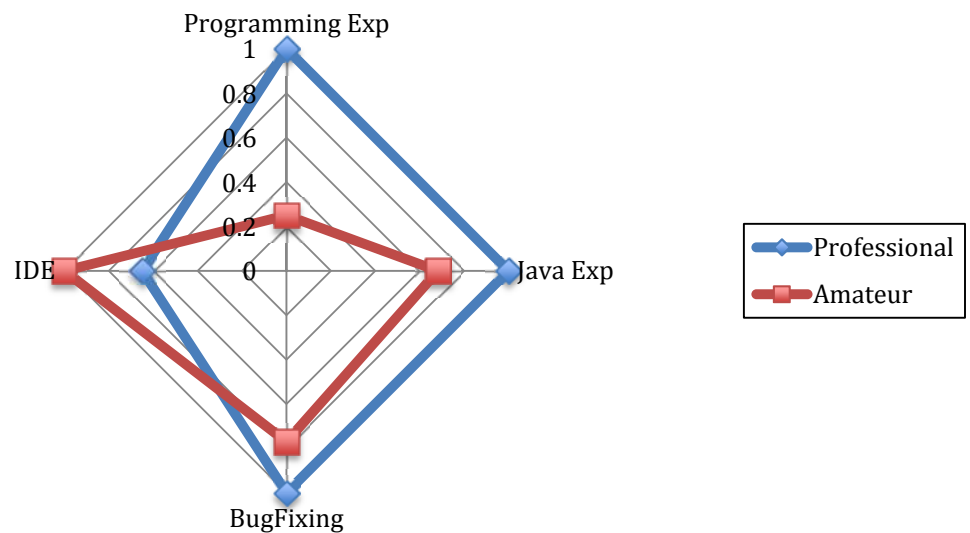


Figure 1: Subjects' Relative Experience Levels between Amateurs and Professionals

As stated in [1], data across a total of 66 change task investigations were collected and for each of these investigations. In order to maintain highest quality in terms of data selection, Kevic et al. excluded 11 change task investigations whose collected data was

inconsistent and therefore, ended up with 55 investigations (18 subjects investigating task 2, 16 subjects investigating task 3, and 21 subjects investigating task 4). With respect to all participants and tasks, Kevic et al. use a total of 688 method investigation instances; collected by iTrace and Mylyn simultaneously. We use the same dataset from Kevic et al. [1] for the experiments in this thesis.

3.4 Results from Previous Work

The dataset used in this thesis was generated by the study conducted in [1]. Towards investigating the developers' detailed behavior while performing a change task, that study [1] showed that gaze data contains substantially more data, as well as more fine-grained data than other similar sources. The study further added that gaze data is in fact different and captures more aspects about each interaction event as compared to just interaction-based data. The analysis also showed that developers working on a realistic change tasks only looked at a very few lines within a method rather than read the whole method as was often found in studies on single method tasks. Also, when it comes to switches between methods, the eye traces reveal that developers only rarely follow call graph links and mostly only switch to the elements in proximity of the method within the class.

These detailed findings provide insights and opportunities for future developer support. For instance, the findings demonstrate that method summarization techniques could be improved by applying some program slicing first and focusing on the lines in the method that are relevant to the current task rather than summarizing all lines in the

whole method. In addition, the findings suggested that a fisheye view of code zooming in on methods in close proximity and blurring out others might have potential to focus developers' attention on the relevant parts and possibly speed up code comprehension. To avoid this, iTrace automatically links eye gazes to source code entities in the IDE and overcomes limitations of previous studies by supporting developers in their usual scrolling and switching behavior within the IDE. This approach indeed opened up new opportunities for our experiments, which have been extended, based on this previous research.

In this thesis, we use the dataset from our earlier work in [1] for prediction and recommendations. In particular, we are trying to predict developer expertise based on gaze data. For recommendations, we are trying to recommend classes to developers based on gaze data. None of these types of experiments on eye gaze data have been conducted before and hence the results derived from this thesis provide some food for thought to improve future prediction and recommendations based on gaze data alone.

3.5 Data Snapshot

In this section, we show a small snippet of the fields contained in the iTrace data and the Mylyn data for a task. The data for iTrace was collected in XML and JSON formats. The data for Mylyn was in XML. The XML files for a single entry for iTrace and Mylyn can be seen in Figure 2 and Figure 3 respectively. The relevant features are discussed in detail in Section 3.9.

```

<?xml version="1.0" encoding="utf-8"?>
<itrace-records>
<environment>
<screen-size width="1920" height="1080"/>
<line-height>0</line-height>
<font-height>0</font-height>
</environment>
<gazes>
<response file="2.txt" type="text" x="429" y="228" left-validation="1.0" right-
validation="1.0" left-pupil-diameter="2.31939697265625" fixation="false" right-
pupil-diameter="2.1541290283203125" tracker-time="1406896742480" system-
time="1406898293351" nano-time="3493902747115" line_base_x="425"
line="7" col="0" hows="DECLARE;DECLARE" types="METHOD;TYPE"
fullyQualifiedNames="net.sf.iabref.label.KeyWord.KeyWord();net.sf.iabref.label.K
evWor" line_base_y="219"/>
</gazes>
</itrace-records>

```

Figure 2: Sample iTrace XML Snippet

```

<?xml version="1.0" encoding="UTF-8"?><InteractionHistory
Id="local-2" Version="1"><InteractionEvent Delta="null"
EndDate="2014-08-01 08:56:50.790 EDT" Interest="8.2"
Kind="propagation"
Navigation="org.eclipse.mylyn.core.model.edges.containment"
OriginId="org.eclipse.mylyn.core.model.interest.propagation"
StartDate="2014-08-01 08:54:32.316 EDT" StructureHandle=""
StructureKind="java" NumEvents="10" CreationCount="1"/>
</InteractionHistory>

```

Figure 3: Sample Mylyn XML Snippet

3.6 Experiments Overview

Moving towards our research goal, we design five experiments, the details of which can be seen in Table 2. The first three experiments were only based on eye

tracking data. The last two were based on interaction history data generated from Mylyn. Even though we wanted to focus on eye tracking data, we also ran experiments on Mylyn data which was collected simultaneously with the eye tracking data during the study conducted in [1]. Since we had this data, we wanted to be able to see if we can determine similar findings from the interaction history data as well.

Table 2: Overview of Experiments Conducted

Experiment ID	Experiment Name	Description	Data Source	RQ
Exp I	Expertise Recommender	Expertise Prediction	iTrace	1
Exp II	Simple Class Recommender	Class Recommender System based on gaze interaction events	iTrace	2
Exp III	Class Recommender with Pupil Deviation	Class Recommender System based on gaze interaction events and respective pupil dilation/contraction.	iTrace	3
Exp IV	Expertise Recommender	Expertise Prediction	Mylyn	1
Exp V	Class Recommender	Class Recommender System based only on interaction events	Mylyn	2

3.7 Data Pre-Processing

Since the dataset we had was spread across several XML log files which used different formats based on their derivative tools i.e. iTrace or Mylyn, we had to first make the data consistent. The following subsections explain the steps we took to pre-process the data.

3.7.1 Data Congregation

In this step, we used a Python script to parse all the XML files and extract data into a single csv file. The resultant file was 6704696 rows and 24 columns strong for the

iTrace dataset and 3789 rows and 17 columns strong for Mylyn logs. The aim of this was to have all data in one place for time efficient manipulation or access and it did help us in terms of time and space complexities by saving on time and space while designing the experiments.

3.7.2 Data Migration

In this step we first uploaded all the data to the Azure Machine Learning Studio. Here, we used the inbuilt modules to transform the dataset into a consistent format by splitting the FullyQualifiedNames into classes/ methods in question. The fields available in master dataset included File, Type, x, y, left-validation, right-validation, left-pupil-diameter, Fixation, right-pupil-diameter, tracker-time, system-time, nano-time, line_base_x, Line, Col, SCE-Count, hows, types, fullyQualifiedNames, line_base_y, Difficulty, Expertise, PID, TaskID, Class and Method. We use combinations and extractions of these fields towards our experiments, which are discussed below. It is important to note that up till this step, we obtain a generic master file with the aforementioned fields. Up ahead, we narrow down these features and even create new features as per the need of the experiments. A subset example can be seen in Figure 4.

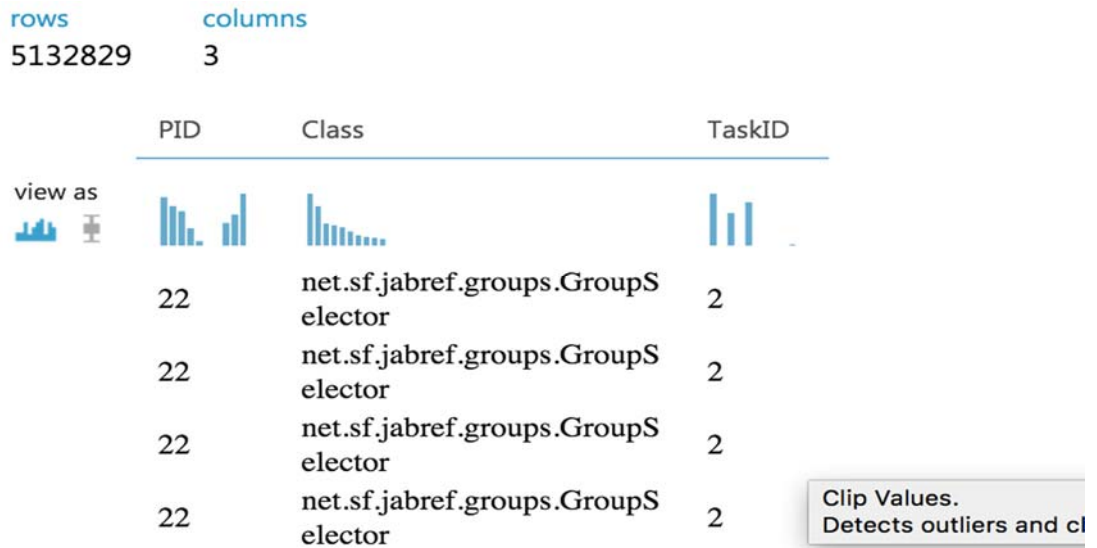


Figure 4: A Subset of the Cleaned-up Master Dataset

The data collected by Mylyn did not need much preprocessing since Mylyn collects data aggregated to a session so this does not leave many null values.

3.8 Machine Learning Model Selection

We designed and executed several experiments to gain better insight into our dataset and to explore optimal models for them. Towards our conclusive experiments, we used the supervised learning approach and put to use Classification and Recommender systems towards our exploratory research.

3.8.1 Expertise Recommender Experiment

Towards Exp I, we opted to go for Two-Class boosted Decision Trees based on its high accuracy and relative moderate training time as compared to other models. As a somewhat generic rule, boosted decision trees yield are strong classifiers [53] specially when features have a low degree of entropy (i.e. they are related), which in our case they

are since they are strongly bound to a same event. In addition, our training and testing set was exponentially larger than the features used and hence, protected our model from over fitting. The way a good model should fit the data can be seen in Figure 5. We can see that a good classifier divides the data using a simple function, which varies with values but can divide the objects into clear labels. An over fitted model cannot adapt to newer data i.e. is not generalized and hence, will fail in the longer run. Similarly, a very simple classifier may not get the desired accuracy/ precision.

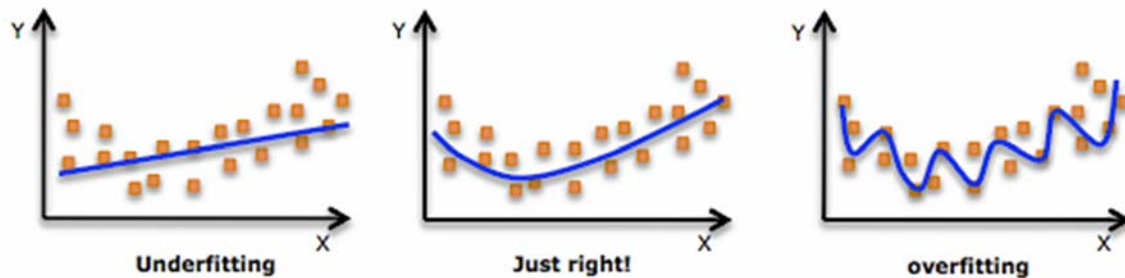


Figure 5: Classifiers Generated by ML Models

3.8.2 Class Recommender Experiments

Towards Exp II and III, we opted to go for the Matchbox Recommender [54]. The objective of a recommendation system is to recommend one or more "items" to "users" of the system, which in our case are "classes" being served to "developers". The Matchbox recommender is hybrid recommender, which combines collaborative filtering with a content-based approach. This means that in case of a new system user, predictions are initialized based on items and their general popularity and improved or personalized as more information about the user is gathered. This successfully takes care of the "cold-

start” problem. This is done via a smooth transition from content-based recommendations to recommendations based on collaborative filtering. Even if user or item features are not available, Matchbox will still work in its collaborative filtering mode, which made it an ideal choice in our case.

3.9 Feature Engineering

The goal of feature engineering is to find an “optimal” subset of relevant features towards enhancing the overall accuracy and consistency while minimizing the size of the training dataset [55]. Of all the steps in feature engineering, feature selection is one of the major problems in areas of machine learning and data mining [56]. Further, a slight possibility of classification emerges due to clear emergence of almost perpendicular axes when clustered into two distinct clusters using K-Means++ Clustering on our cleaned features. A sample of various clusters we generated can be seen in Figure 6 that follows.

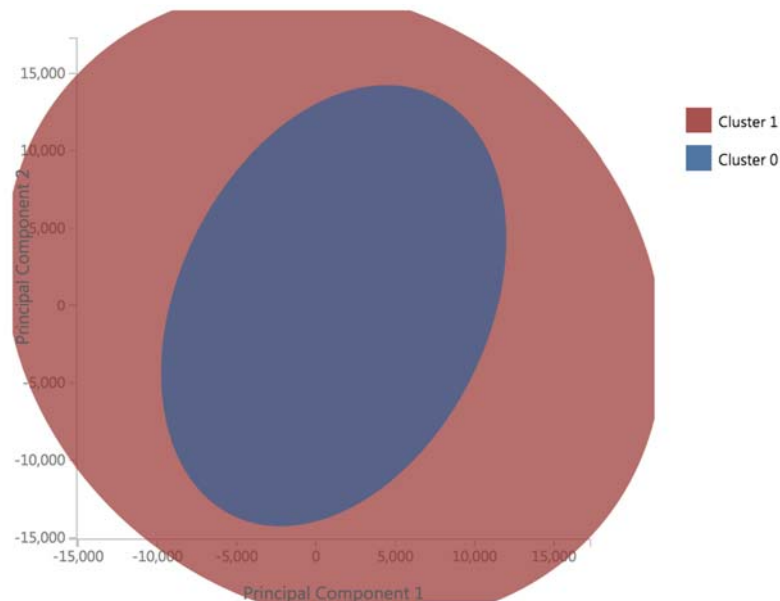


Figure 6: Simple K-Means++ Clustering Result on the iTrace Dataset

In order to choose optimal features, we extracted the feature importance using the ‘Permutation Feature Importance’ module in the Azure ML Studio. This module computes the permutation feature importance scores of feature variables given a trained model and a test dataset. We ran this on several experiments and based on our prior choice of models, discovered that towards Exp I, the “Class” field held the most importance in general followed by “Line” and “Method” as seen in Table 3. Other features had negligible weights, so we could safely neglect them.

Table 3: Feature Importance Scores for Exp I

TaskID	Class	Line	Method
2	0.280722	0.189896	0.022244
3	0.107471	0.196431	0.018746
4	0.263595	0.155731	0.033794

Towards Exp II and III, we created a new feature called “Rating” since we needed only one feature to weight the “users” to their recommended or navigated “class”. For Exp II, we used the class-based event count for each user. We used it to create a “Rating_quantized” feature that referred to the relative time a user spent interacting with a class and the elements associated with it. We clipped peak-outliers based on 98% quartiles and then binned based on PQuartiles into 10 equal bins to get a normally distributed rating. The details and samples can be seen in Table 4 and Figure 7.

Table 4: The Initial “Rating” vs. “Rating quantized” Feature Statistics for Exp II

Type	Mean	Median	Min	Max	S.D.	Unique Values
Actual	586.0654	83	1	4890	1211.3128	499
Binned	5.4741	5	1	10	2.9084	10





PID	Class	Rating	Rating_quantized
			
1	.res	2	1
1	java.awt.Component	339	8
1	java.awt.Container	1428	9
1	java.awt.GridBagLayout	1260	9
1	java.awt.Window	77	5
1	java.lang.Character	4	2
1	java.lang.String	327	8
1	java.util.HashSet	2	1

Figure 7: Sample of “Rating_quantized” for Exp II

Finally, for Exp III, since we wanted to also make use of Pupil deviations, the “Rating” feature was a combination of quantized and binned class-based event counts and relative pupil diameters for each event. As an additional step, we extracted the average pupil diameter for each user per task and then the relative pupil deviation for each interaction event. Proceeding further as before, we clipped peak-outliers based on 98% quartiles and then binned based on PQuartiles into 10 equal bins to get a normally distributed rating for pupil dilation/contraction as well. Towards merging both the features i.e. class-based event count and relative pupil deviation, we simply added both the binned/quantized values. This new feature was renamed as “Rating”- and the values of “Rating_quantized” were binned into 10 bins. The statistics of “Rating_quantized” can

be seen in Figure 8 and the distribution in Figure 9. Finally, we use three features for Exp II and III – PID, Class and Rating to develop to our recommender system with.

▲ Statistics

Mean	10.9758
Median	11
Min	2
Max	20
Standard Deviation	4.1089
Unique Values	19
Missing Values	0
Feature Type	Numeric Feature

Figure 8: The “Rating_quantized” Feature Statistics for Exp III

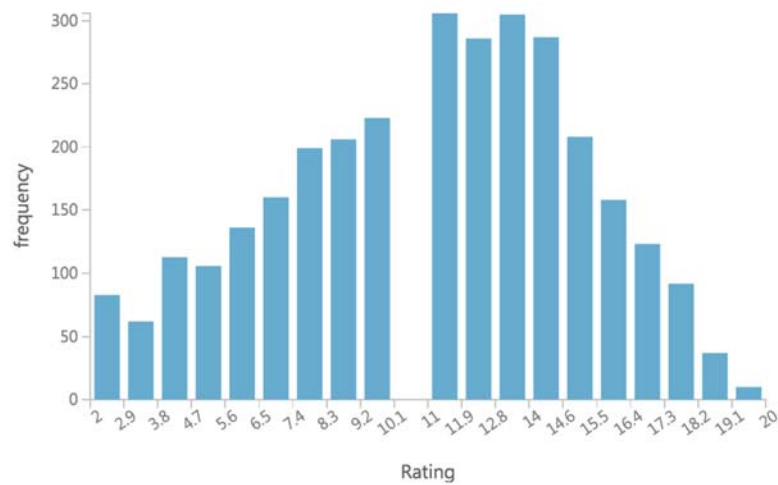


Figure 9: “Rating_quantized” distribution bar plot for Exp III

Similarly, for the Mylyn log dataset i.e. for Exp V, we binned and quantized the “Interest” rating to obtain the “Ranking”. This single manipulation after cleaning duplicate values gave us a dataset that can be seen in

Figure 10.

Rating	StructureHandle	PID
1	=jabref- 1.8.1/srcVjava<net.sf.jab ref{JabRef.java[JabRef~ma in~\QString;	2
1	=jabref- 1.8.1/srcVjava<net.sf.jab ref{JabRef.java[JabRef~ma in~\QString;	2
6	=jabref- 1.8.1/srcVjava<net.sf.jab ref{JabRef.java[JabRef =jabref-	2

Figure 10: Mylyn Dataset used towards the Recommender System in Exp V

Towards Expertise prediction i.e. Exp IV, we had the Rating, Kind, Navigation, OriginId, and StructureHandle features to classify into the Expertise label. However, upon the evaluation, we found the Feature importance scores to be non-consistent to carry out any further experimentation since the trained model may be severely over/under fitted based on these features. These scores can be seen in Table 5. This possibly was due to log aggregation by Mylyn, which led to ambiguous feature values for all logical labels, thus rendering machine learning useless.

Table 5: Feature Importance Scores for Expertise Prediction in Exp IV

TaskID	Rating	Kind	Navigation	OriginID	StructureHandle
2	0	0	0	0	0
3	-0.02907	-0.011628	0.017442	0.023256	0
4	0.094118	-0.011765	0	-0.035294	0

3.10 Experiment Instrumentation

Microsoft Azure ML gave us a scalable and powerful set of tools known as modules to design our experiments with. The various modules we used in our experiments are:

1. Clean Missing data: This is the step which allows us to remove rows with missing critical values or to replace null values with a default value (could be a constant or mathematical operation)
2. Remove Duplicate Rows: The name tells us all. We used this to preserve the first unique row and then to eliminate all duplicates. This was done in cases where all unique rows had to be given equal weightage for the ML models to come up with unbiased functions.
3. Project Columns: This is the step where we can select the columns to be selected from the entire dataset. We used this to select relevant columns so as to obtain logical relationships.
4. Metadata Editor: This module is used to make values of a feature categorical or to modify the data type of all values in a column.
5. Indicator Values: This module converted all the values under a feature into indicator-features and assigned them Boolean values. The output of this module added columns to the input dataset to achieve this.
6. Split: This step allows us to select rows based on different parameters such as cell value, column name among others. We can split our input dataset into two parts

based on a ratio of the first dataset to the input dataset (We used between 0.5 and 0.7). Another options include randomization of the outputs -which we used to be able to train our models with a variety of samples. A seed value helps in retrieving the randomly generated dataset again, if needed.

7. Train Model: This is the step where the selected model is used on the Split dataset to train it. This step is moderately time consuming.
8. Sweep Parameters: This module uses the training algorithm and comes up with the most optimal parameter values for the training and testing dataset. This though, takes up a lot of time but is useful since we can optimize the predictions based on accuracy or precision apart from others.
9. Cross Validate Model: Cross-validates parameter estimates for classification or regression models by partitioning the data. We use this on the data to be classified based on our model selection.
10. Permutation Feature Importance: Computes the permutation feature importance scores of feature variables given a trained model and a test dataset.
11. Score Model: We use the trained model on the non-trained dataset from the prior split. This is a time consuming process based on the dataset size. A column is added to our dataset for cross checking and evaluation purposes.
12. Evaluate Model: This is the final step where we evaluate the results of a classification or regression model with standard metrics. For most of our experiment, we obtained and used the following parameters:
 - a. Accuracy: It measures the goodness of a classification model as the

proportion of true results to total cases.

- b. Precision: It is the proportion of true results over all positive results.
- c. Recall: The fraction of all correct results returned by the model.
- d. F-score: The computed weighted average of precision and recall between 0 and 1, where the ideal F-score value is 1.
- e. AUC: This is the measured area under the curve plotted with true positives on the y-axis and false positives on the x-axis. This metric is useful because it provides a single number that lets us compare models of different types.
- f. Average log loss: It is a single score used to express the penalty for wrong results. It is calculated as the difference between two probability distributions – the true one, and the one in the model.
- g. Training log loss: It is a single score that represents the advantage of the classifier over a random prediction.
- h. NDCG (Normalized discounted cumulative gain): This measures the performance of a recommendation system based on the graded relevance of the recommended entities. It varies between 0.0 and 1.0, with the latter representing the ideal ranking of the entities. This metric is commonly used in information retrieval and to evaluate the performance of recommenders such as web search engines.

As previously discussed, we used Two-Class Boosted Decision Tree for Exp I i.e. towards Expertise Prediction. The details of the dataset used for this experiment can be

seen in Table 6. We split our experiment into three parts as per the three tasks we have. This allowed for more accurate classifications since the values of each feature varies significantly across tasks. The experiment flow can be seen in Figure 11 and Figure 12.

Table 6: Subset used for Expertise Prediction

Column	Value Type	Variable Type	Unique Values
Line	Numeric Value	Feature	975
Class	Categorical Feature	Feature	1224
Method	Categorical Feature	Feature	733
Expertise	Categorical Feature	Label	2

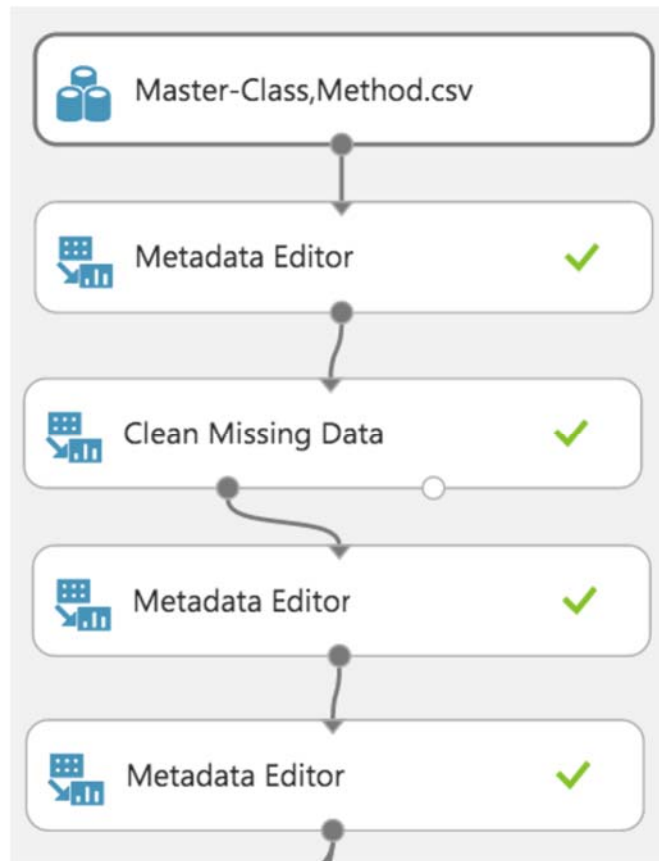


Figure 11: Exp I: Data Cleanup and Feature Type Updating for Exp I

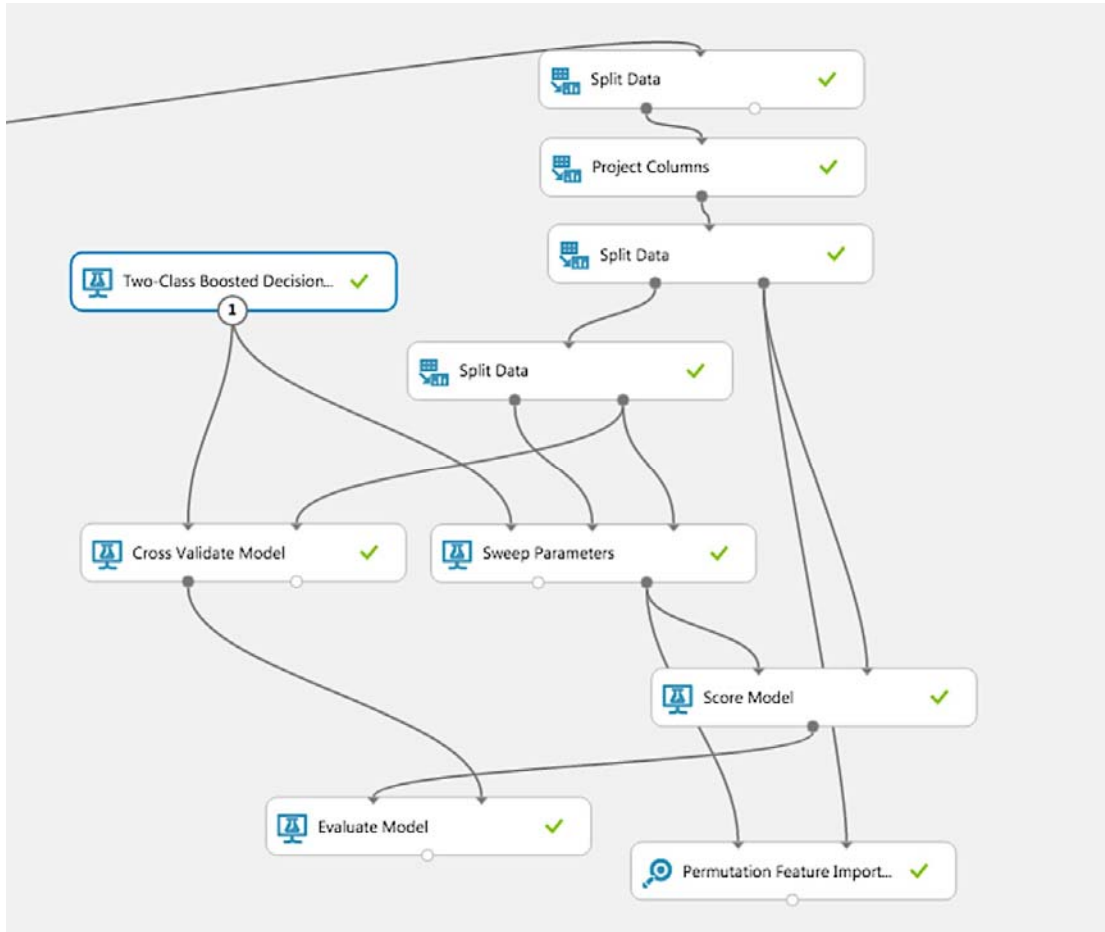


Figure 12: Training, Scoring and Evaluation in Exp I

Towards Exp II and III i.e. for designing the recommender systems, we used a Bayesian Recommender using the Matchbox algorithm. The details of the dataset used for this experiment can be represented in Figure 7. We yet again, split our experiment into three parts as per the three tasks we have. The experiment flow can be seen in Figure 13 and Figure 14.

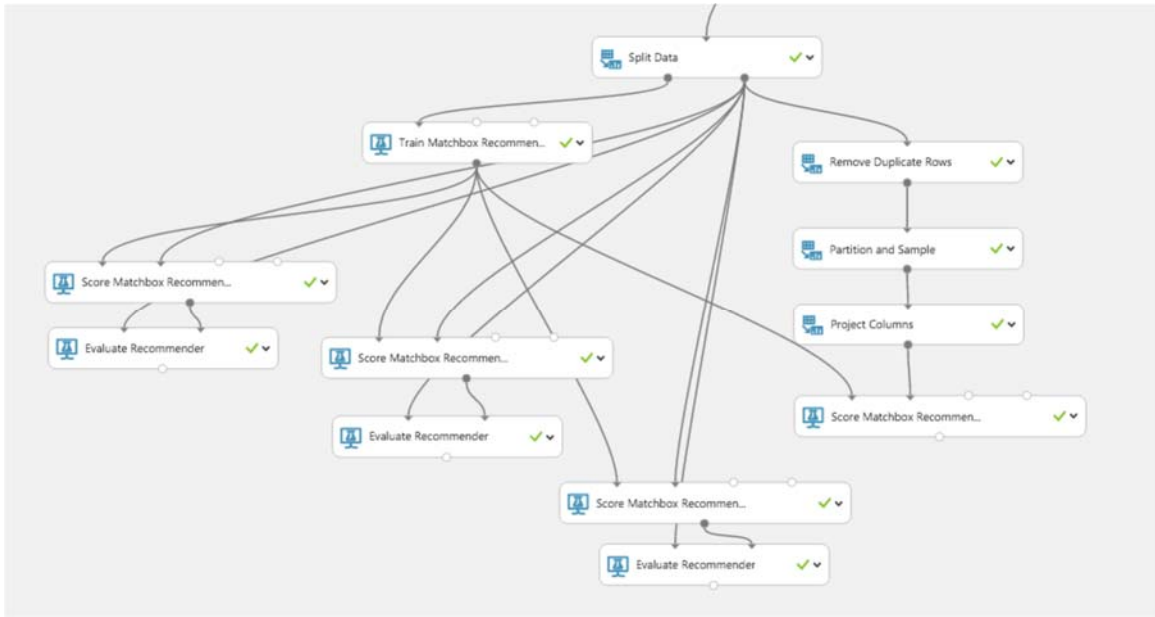


Figure 13: Training, Scoring and Evaluation in Exp II and III.



Figure 14: Preprocessing Data for Exp III

CHAPTER 4

RESULTS AND ANALYSES

This chapter presents the performance of our trained models on the scoring dataset based on the Evaluate Model module parameter as discussed in Section 3.5.

4.1 RQ1: Can we predict the expertise of a developer based on gaze data and interactions with the IDE for a specific task?

After several tweaks of our process flows, the optimal parameters discovered and used for the various modules can be seen in Table 7 and the results of scoring can be seen in Table 8.

Table 7: Discovered and Used Parameters for Exp I

S.No.	Parameter	Final Value
1	Training Split	0.7 (Random)
2	Sweep Parameters – Split	0.6 (Random)
3	Sweep Parameters – Runs	5
4	Model - Max Leaves/ Tree	36
5	Model - Min Samples/ Node	7
6	Model – Learning Rate	0.333
7	Model – Number of Trees	182

Parameter 1 is the training/testing split i.e. 70% to train and 30% to score and evaluate. These rows are chosen randomly. Parameters 2 and 3 are those discovered by the Sweep Parameters module. Finally, Parameters 4 to 7 describe the design of our classification model.

Table 8: Model Evaluation for Exp I

TaskID	Accuracy	Precision	Recall	F1 Score	AUC (Per 0.5 Threshold)
2	0.905	0.941	0.906	0.923	0.971
3	0.794	0.787	0.828	0.807	0.883
4	0.853	0.875	0.860	0.867	0.937
Mean	0.851	0.868	0.865	0.866	0.930

In this experiment, since the AUC is way above 0.5, we can safely assume that the system did not make random guesses. Further, based on the cross validation of our selected model we can see that the Standard Deviation (Table 9) is very small for the various parameters and hence, confirms the reliability of both the variability of our training dataset and the selection of our model.

Table 9: Exp I: Model Cross Validation

	Examples /Fold	Accuracy	Precision	Recall	F-Score	AUC	Average Log Loss	Training Log Loss
Mean	57126.7	0.89278	0.939846	0.88516	0.911681	0.960275	0.244246	63.073989
S.D.	< 1	0.002	0.001373	0.003103	0.001803	0.00091	0.002556	0.388006

The terms accuracy and precision are discussed in detail in Section 3.10 under the Evaluate Model module.

Based on a mean accuracy of 85% and after having obtained a satisfactory result with Cross Validation, we can conclude that eye gaze data can be used to classify developers across different tasks based on Expertise. This was possible due to significant differences in gaze patterns between the two classes of Experts, which allowed the

Machine Learning model to define clear rules for assigning developers to either class. Further, a narrow grey area helped us obtain a high amount of accuracy.

4.2 RQ2: Can we use the class-based eye gaze and interaction of a developer to intelligently suggest classes of interest?

Similar to Exp I, the optimal parameters discovered and used for the various modules towards Exp II can be seen in Table 10, the model evaluations in Table 11 and the results of scoring in Table 12.

Table 10: Discovered and Used Parameters for Exp II

S.No.	Parameter	Final Value
1	Training Split	0.75 (Recommender Split)
2	Model – Number of Traits	20
3	Model – Algorithm Iterations	10
4	Model – Training Batches	4
5	Recommender – Max pool	5
6	Recommender – Min pool	2
7	Model – Number of Trees	182

Same as before, Parameter 1 is the training/testing split i.e. 75% to train and 25% to score and evaluate. These rows are chosen as per the input requirements of the Matchbox Recommender. Parameters 2 to 4 are those discovered by the Sweep Parameters module and Parameters 5 and 6 are those set up by us. Min pool defines the minimum classes to be recommended and Max pool, the maximum. Finally, Parameter 7 describes the design of our recommender model.

Table 11: Recommender Evaluation for Exp II

TaskID	Exp II: NDCG
2	0.899
3	0.905
4	0.843
Mean	0.882

Table 12: Recommender Pool Sample for Task 2 in Exp II

User	Item 1	Item 2	Item 3	Item 4	Item 5
29	net.sf.jabref.groups. .GroupsPrefsTab	javax.swing. AbstractButton	net.sf.jabref. Util	net.sf. jabref. Globals	net.sf.jabref. BibtexEntry
26	javax.swing. AbstractButton	java.util.HashMap	net.sf.jabref. Util	net.sf.jabre f.Globals	net.sf.jabref.JabRef Preferences
4	net.sf.jabref. groups. GroupsPrefsTab	javax.swing. AbstractButton	net.sf.jabref. Globals	net.sf. jabref.Util	java.awt. Component
9	java.util.Hashtable	net.sf.jabref.label. KeyWord. keyWordTable	-	-	-
2	java.util.HashMap	net.sf.jabref. BibtexEntry	net.sf.jabref. Util	net.sf.jabre f.Globals	net.sf.jabref.JabRef Preferences

As discussed in Section 3.10 under the Evaluate Model module, NDCG stands for Normalized discounted cumulative gain (NDCG). It measures the performance of a recommendation system based on the graded relevance of the recommended entities. It varies from 0.0 to 1.0, with 1.0 representing the ideal ranking of the entities. In this experiment the mean being 0.882, our recommender can be considered a success and performs quite well. Further, the Recommendation Pool has recommendations ranging between 2 and 5 classes since; they were the parameters we chose in the previous section.

Most users are recommended 5 classes but user 9 is not - possibly due to the fact that no other classes had a higher enough predicted ranking to be considered of relevance. However, had we increased the min pool size, more recommendations would have been made. Therefore, based on the mean NDCG of 0.882, we can say that we designed a successful recommender system using class-based eye gaze information of a developer.

4.3 RQ3: Can we also use pupil dilation/contraction information alongside class-based gaze interaction to intelligently suggest classes of interest?

Similar to Exp II, the optimal parameters discovered and used for the various modules towards Exp III are the same as Table 10; the model evaluations can be seen in Table 13 and the results of scoring in Table 14.

Similar to RQ2, in this experiment the mean being 0.836, our recommender can be considered a success though it falls behind the earlier one. Further, all users are recommended 5 classes due to the fact that all classes had significant predicted ranking to be considered of relevance. One interesting thing to note is that the recommendations pools of the two experiments though not entirely different, allocate different probabilities based ranking to the same item. An example of recommendation pool comparison for TaskID 2 and UserID 2 can be seen in Table 15.

Table 13: Recommender Evaluation for Exp III

TaskID	Exp III: NDCG
2	0.787
3	0.890
4	0.830
Mean	0.836

Table 14: Recommender Pool Sample for Task 2 in Exp III

User	Item 1	Item 2	Item 3	Item 4	Item 5
23	net.sf.jabref.JabRefPreferences	net.sf.jabref.Util	Buffer	net.sf.jabref.Util.openExternalViewer.cmdArray	javax.swing.JOptionPane
1	net.sf.jabref.JabRefPreferences	net.sf.jabref.BasePanel	java.util.Set	net.sf.jabref.ContentSelectorDialog2	net.sf.jabref.ContentSelectorDialog2.fieldPan
28	net.sf.jabref.EntryComparator	java.awt.Container	net.sf.jabref.FieldContentSelector	net.sf.jabref.FieldContentSelector.actionPerformed.chosen	net.sf.jabref.EntryComparator.sortField
2	net.sf.jabref.JabRefPreferences	javax.swing.text.JTextComponent	java.util.HashMap	net.sf.jabref.groups.KeywordGroup.m_pattern	java.io.PrintStream
6	net.sf.jabref.JabRefPreferences	java.util.HashMap	javax.swing.text.JTextComponent	net.sf.jabref.groups.KeywordGroup.m_pattern	net.sf.jabref.groups.KeywordGroup.addOldContent

Table 15: Recommender Pool Exp II vs. III for User 2 and Task 2.

Exp	Item 1	Item 2	Item 3	Item 4	Item 5
II	java.util.HashMap	net.sf.jabref.BibtexEntry	net.sf.jabref.Util	net.sf.jabref.Globals	net.sf.jabref.JabRefPreferences
III	net.sf.jabref.JabRefPreferences	javax.swing.text.JTextComponent	java.util.HashMap	net.sf.jabref.groups.KeywordGroup.m_pattern	java.io.PrintStream

On the basis of the mean NDCG of 0.836, we can say that we have designed successful recommender system using pupil dilation/contraction information alongside class-based gaze interactions in RQ2. Though, the performance of this system falls a little short of our previous recommender system, it still performs at a satisfactory level.

4.4 Threats to Validity

The quality of data is the most critical requirement of a good machine learning system. In our case, one threat to validity could be the short time period each participant had for working on a change task since, we were limited by the time availability of the participants and had to restrict the study to one hour with 20 minutes given for each task. This means that while the data does not represent the completion of a task but does provide insight into represent the navigation of users. Another threat to validity is the choice of JabRef as the subject system. JabRef is written in a single programming language and its code complexity and quality might influence the study. For instance, code with low quality and/or high complexity might result in developers spending more time to read and understand it, and thus longer eye gaze times and consequently the higher class “Ranking”. We tried to mitigate this risk by choosing a generally available system that is an actively used and maintained open source application and that was also used in other studies and by developing a second recommender system making use of pupil deviation. Constricted pupils in general refer to active interaction - usually known as accommodative response and vice versa. Further studies, however, are needed to examine other factors, such as code quality, to generalize the results.

As stated in Kevic et al.'s work [1], JabRef had to be run through the command prompt using ANT and not directly in Eclipse. This meant that participants were not able to use breakpoints and the debugger within Eclipse and might have influenced the results. In addition, iTrace collects eye gazes only within Eclipse editors. This means that we do not record eye gaze when the developer is using the command prompt or running JabRef. However, since we were interested in the navigation between the code elements within the IDE, this does not pose any limitations to our analysis. If the user opens the "Find in File" or "Search Window" within Eclipse, or a tooltip pops up when hovering over an element in the code, the eye gaze is not recorded as this overlaps a new window on top of the underlying code editor window and iTrace did not support gazes on search windows at the time of the study.

To minimize the time in which eye gazes could not be recorded, we requested the participants to close these windows so gaze recording can continue. Finally, most professional developers were mainly Visual Studio users for their work; we conducted our study in Eclipse. However, all professional developers stated that they did not have problems using Eclipse during the study.

For a lab-based study, our machine learning system performs to satisfactory levels but this may vary in real time and under various circumstances. The beauty of implementing this system as a cloud service would be that we could train the system to adopt if any patterns are found in the deviations of our classifications/ recommendations to the real world usage, which is part of our future work.

4.5 Discussion of Results

Based on the above findings, we can confidently say that these experiments lay a basis for further research in recommender systems and classification of expertise. This was made possible by the granularity of data collected by iTrace. We failed to obtain satisfactory results from the Mylyn logs towards designing similar systems. For sake of comparison, we did go ahead and design Exp IV, the results of which were over fitted as can be seen in Figure 16. As can be seen in comparison to Figure 15, the Mylyn log based model is way over fitted.

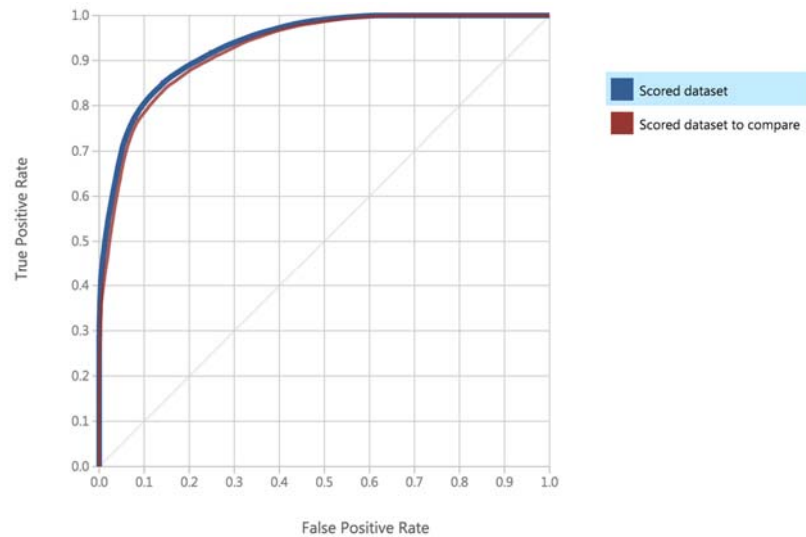


Figure 15: Model for Expertise Prediction for TaskID 4 using iTrace logs

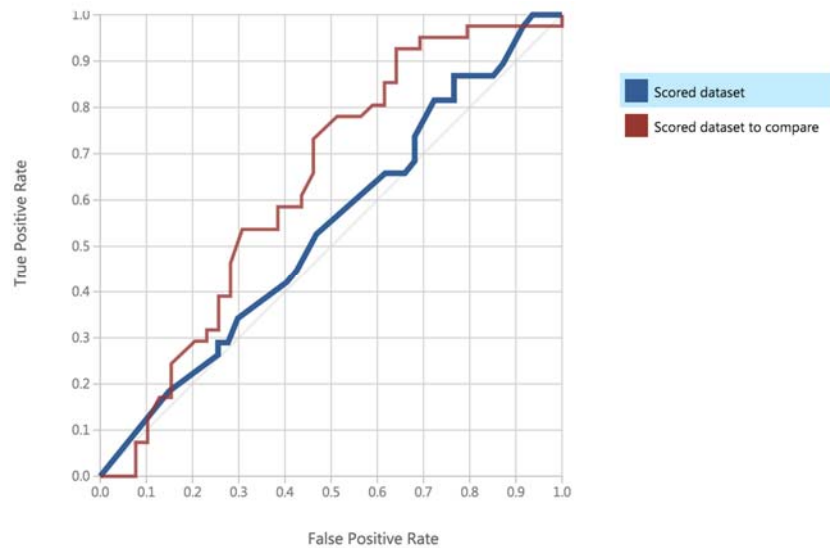


Figure 16: Model for Expertise Prediction for TaskID 4 using Mylyn logs

So far, based on our findings, we can say that tracing developers' eyes during their work on change tasks does in fact offers a variety of new insights and opportunities to support developers at work. Especially, the richness and granularity of the data provided by iTrace provides to the potential for new and improved tool support, which was not possible earlier. This support may include but is not limited to navigational recommendations and expertise classification among others. We can in fact put these two approaches together and train out recommender model based on the real time feed of Professionals. This may have the ability to enhance the performance of amateurs by helping them navigate code.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

To explore the possibilities of categorizing developers as amateurs or professionals based on their eye tracking patterns, we developed a machine learning system using the two-class Boosted Decision Tree algorithm with satisfactory results. We further strengthen our previous study's premise that eye gaze based interaction data has more possibilities in terms of applications and research. The future work may include adding and modifying features from the iTrace dataset to scale this system to work for a more generalized set of data and in real time while maintaining or improving the accuracy while the system keeps learning online. This system also needs to be tested under wide array of circumstances to observe the performance and learning mechanisms in real time. Several other types of classifications such as determining the task difficulty or interest levels based contextual processing via pupil dilation data [4] are also possible as future work.

Towards the recommender systems, we came up with two systems using Bayesian recommenders and Matchbox Algorithms. One of them only made use of the time spent interacting with the various elements of a class as the "Rating" whereas the other also made use of the pupil contractions. Though their recommendation pools varied, they did perform well in terms of the NDCG metrics. In addition to the NDCG metrics, we also

plan to report more standard measures such as the sum of errors to have our experiment easily compared to others in the future.

Other possible future modifications include more parameters to the “Rating” while enhancing the recommendations. Also, the evaluation of the evolution of the recommender pool in real time scenarios can be studied to enhance the aforementioned “Rating”. Sentimental analysis on bug fixing reports combined with the iTrace logs can be used towards a newer formula for “Rating”. This can be further extended to analyze reports or comments in real time to alter the recommendation pools and can provide assistance in remote working environments.

Finally, the granularity of eye tracking data can give us a different insight into the contextual process of a developer and when coupled with other biometric logs on a time series can provide us an unparalleled amount of understanding of the developers’ thought and development process.

References

- [1] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, “Tracing Software Developers’ Eyes and Interactions for Change Tasks,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2015, pp. 202–213.
- [2] T. R. Shaffer, J. L. Wise, B. M. Walters, S. C. Müller, M. Falcone, and B. Sharif, “iTrace: Enabling Eye Tracking on Software Artifacts Within the IDE to Support Software Engineering Tasks,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2015, pp. 954–957.
- [3] Mik Kersten, “eclipse.org/mylyn/.” .
- [4] T. Partala and V. Surakka, “Pupil size variation as an indication of affective processing,” *Int. J. Hum.-Comput. Stud.*, vol. 59, no. 1–2, pp. 185–198, Jul. 2003.
- [5] J. Hyönä, J. Tommola, and A.-M. Alaja, “Pupil Dilation as a Measure of Processing Load in Simultaneous Interpretation and Other Language Tasks,” *Q. J. Exp. Psychol. Sect. A*, vol. 48, no. 3, pp. 598–612, Aug. 1995.
- [6] www.tobii.com/ .
- [7] R. Brooks, “Towards a Theory of the Comprehension of Computer Programs,” *Int. J. Man-Mach. Stud.*, vol. 18, pp. 543–554, 1983.
- [8] N. Pennington, “Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs,” *Cognit. Psychol.*, no. 19, pp. 295–341, 1987.

- [9] Shneiderman B. and R. Mayer, “Syntactic Semantic Interactions in Programmer Behavior: A Model and Experimental Results,” *Intl J Comp Info Sci.*, vol. 18, pp. 219–238, 1979.
- [10] Teresa M. Shaft and I. Vessey, “The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension,” *Inf. Syst. Res.*, vol. 6, no. 3, pp. 286–299, 1995.
- [11] Letovsky S. and E. Soloway, “Delocalized Plans and Program Comprehension,” *IEEE Softw.*, pp. 41–49, 1986.
- [12] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. Paterson, C. Schulte, B. Sharif, and S. Tamm, “Eye Movements in Code Reading: Relaxing the Linear Order,” in *International Conference on Program Comprehension*, 2015, pp. 255–265.
- [13] B. Walters, T. Shaffer, B. Sharif, and H. Kagdi, “Capturing software traceability links from developers’ eye gazes,” in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 201–204.
- [14] Bonita Sharif and T. Shaffer, “The Use of Eye Tracking in Software Development,” presented at the Augmented Cognition for Daily Living - HCI International (HCII 2015), Los Angeles, CA, USA, 2015.
- [15] A. Begel, M. Hansen, R. Bednarik, P. Orlov, P. Ihantola, G. Shchekotova, and M. Antropova with T. Busjahn, C. Schulte, and B. Sharif, Simon, “Eye Tracking in Computing Education,” in *Proceedings of the Tenth Annual Conference on International Computing Education Research*, New York, NY, USA, 2014, pp. 3–10.

- [16] A. J. Ko, B. A. Myers, M. J. Coblentz, and H. H. Aung, “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks,” *Softw. Eng. IEEE Trans. On*, vol. 32, no. 12, pp. 971–987, Dec. 2006.
- [17] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, “Using Psychophysiological Measures to Assess Task Difficulty in Software Development,” in *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 402–413.
- [18] Soloway E. and K. Ehrlich, “Plans in Programming: Definition, Demonstration and Development,” *Empir. Stud. Program. Knowl. IEEE Trans. Softw. Eng.*, pp. 595–609, 1984.
- [19] E. M. Altmann, “Near-term memory in programming: a simulation-based analysis,” *Int. J. Hum. Comput. Stud.*, vol. 54, no. 2, pp. 189–210, 2001.
- [20] M. Kersten and G. C. Murphy, “Mylar: A Degree-of-interest Model for IDEs,” in *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, New York, NY, USA, 2005, pp. 159–168.
- [21] M. Kersten and G. C. Murphy, “Using task context to improve programmer productivity,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 1–11.
- [22] K. Rayner, “Eye movements in reading and information processing: 20 years of research.,” *Psychol. Bull.*, vol. 124, no. 3, p. 372, 1998.

- [23] M. Just and P. Carpenter, “A Theory of Reading: From Eye Fixations to Comprehension,” *Psychol. Rev.*, vol. 87, pp. 329–354, 1980.
- [24] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, “Micro Interaction Metrics for Defect Prediction,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, Szeged, Hungary, 2011, pp. 311–321.
- [25] B. Steichen, C. Conati, and G. Carenini, “Inferring Visualization Task Properties, User Performance, and User Cognitive Abilities from Eye Gaze Data,” *ACM Trans Interact Intell Syst*, vol. 4, no. 2, pp. 11:1–11:29, Jul. 2014.
- [26] C. Parnin and C. Gorg, “Building Usage Contexts During Program Comprehension,” in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 13–22.
- [27] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy, “Modeling programmer navigation: A head-to-head empirical evaluation of predictive models,” in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, 2011, pp. 109–116.
- [28] D. Čubranić and G. C. Murphy, “Hipikat: Recommending Pertinent Software Development Artifacts,” in *Proceedings of the 25th International Conference on Software Engineering*, Washington, DC, USA, 2003, pp. 408–418.
- [29] “Usage Data Collector.” [Online]. Available: <https://eclipse.org/epp/usagedata/>. [Accessed: 21-Sep-2015].

- [30] C. Bird, T. Menzies, and T. Zimmermann, *The Art and Science of Analyzing Software Data: Analysis Patterns*. Elsevier, 2015.
- [31] B. Sharif and J. I. Maletic, “An Eye tracking Study on camelCase and Under_score Identifier Styles,” in *18th IEEE International Conference on Program Comprehension (ICPC’10)*, pp. 196–205.
- [32] S. Yusuf, H. Kagdi, and J. I. Maletic, “Assessing the Comprehension of UML Class Diagrams via Eye Tracking,” in *Program Comprehension, 2007. ICPC ’07. 15th IEEE International Conference on*, 2007, pp. 113–122.
- [33] S. Müller and T. Fritz, “Stuck and Frustrated or In Flow and Happy: Sensing Developers’ Emotions and Progress.”
- [34] M. E. Crosby and J. Stelovsky, “How Do We Read Algorithms? A Case Study,” *Computer*, vol. 23, no. 1, pp. 24–35, Jan. 1990.
- [35] B. de Smet, L. Lempereur, Z. Sharafi, Y.-G. Guéhéneuc, G. Antoniol, and N. Habra, “Taupe: Visualizing and Analysing Eye-tracking Data,” *Sci. Comput. Program. J. SCP*, vol. 87, 2011.
- [36] B. Sharif, G. Jetty, J. Aponte, and E. Parra, “An Empirical Study Assessing the Effect of SeeIT 3D on Comprehension,” in *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*, pp. 1–10.
- [37] B. Sharif and J. I. Maletic, “An Eye tracking Study on the Effects of Layout in Understanding the Role of Design Patterns,” in *26th IEEE International Conference on Software Maintenance (ICSM’10)*, pp. 1–10.

- [38] R. Turner, M. Falcone, B. Sharif, and A. Lazar, “An eye-tracking study assessing the comprehension of C++ and Python source code,” in *Proc. of the Symposium on Eye Tracking Research & Applications*, Safety Harbor, Florida, 2014, pp. 231–234.
- [39] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, “The Impact of Identifier Style on Effort and Comprehension,” *Empir. Softw. Eng. J. Invit. Submiss.*, vol. 18, no. 2, pp. 219–276, 2013.
- [40] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto, “Analyzing individual performance of source code review using reviewers’ eye movement,” in *Proc. of the Symposium on Eye Tracking Research & Applications*, San Diego, California, 2006, pp. 133–140.
- [41] R. Bednarik, “Expertise-dependent Visual Attention Strategies Develop over Time During Debugging with Multiple Code Representations,” *Int. J. Hum.-Comput. Stud.*, vol. 70, no. 2, pp. 143–155, Feb. 2012.
- [42] R. Bednarik and M. Tukiainen, “An eye-tracking methodology for characterizing program comprehension processes,” in *Proc. of the Symposium on Eye Tracking Research & Applications*, 2006, pp. 125–132.
- [43] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D’Mello, “Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers,” in *Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 390–401.

- [44] B. Walters, M. Falcone, A. Shibble, and B. Sharif, “Towards an Eye-tracking Enabled IDE for Software Traceability Tasks,” in *7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pp. 51–54.
- [45] B. Sharif and H. Kagdi, “On the Use of Eye Tracking in Software Traceability,” in *6th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE’11)*, pp. 67–70.
- [46] Y. Koren, “The BellKor Solution to the Netflix Grand Prize,” Aug-2009.
- [47] Andreas Toscher, Michael Jahrer, and Robert M. Bell, “The BigChaos Solution to the Netflix Grand Prize.”
- [48] Martin Piotte and Martin Chabbert, “The Pragmatic Theory solution to the Netflix Grand Prize.”
- [49] B. Sharif, M. Falcone, and J. I. Maletic, “An Eye-tracking Study on the Role of Scan Time in Finding Source Code Defects,” in *Symposium on Eye Tracking Research and Applications (ETRA)*, pp. 381–384.
- [50] C. Aschwanden and M. Crosby, “Code Scanning Patterns in Program Comprehension,” in *Proceedings of the 39th Hawaii International Conference on System Sciences*, 2006.
- [51] G. E. A. P. A. Batista and M. C. Monard, “An analysis of four missing data treatment methods for supervised learning,” *Appl. Artif. Intell.*, vol. 17, no. 5–6, pp. 519–533, May 2003.
- [52] “jabref.sourceforge.net/.”

- [53] K. M. Svore and C. J. C. Burges, “Large-scale Learning to Rank using Boosted Decision Trees,” in *Scaling Up Machine Learning: Parallel and Distributed Approaches*, Cambridge University Press, 2011.
- [54] D. Stern, R. Herbrich, and T. Graepel, “Matchbox: Large Scale Bayesian Recommendations,” in *Proceedings of the 18th International World Wide Web Conference*, 2009.
- [55] M. Dash and H. Liu, “Consistency-based search in feature selection,” *Artif. Intell.*, vol. 151, no. 1–2, pp. 155–176, Dec. 2003.
- [56] Y. Qian, J. Liang, W. Pedrycz, and C. Dang, “Positive approximation: An accelerator for attribute reduction in rough set theory,” *Artif. Intell.*, vol. 174, no. 9–10, pp. 597–618, Jun. 2010.

December 15, 2015

Dr. Bonita Sharif, Principal Investigator
Mr. Ahraz Husain, Co-investigator
Department of Computer Science & Information Systems
UNIVERSITY

RE: HSRC Protocol Number: 065-2016
Title: Understanding How Developers Work on Change Tasks Using Interaction
History and Eye Gaze Data

Dear Dr. Sharif and Mr. Husain:

The Institutional Review Board has reviewed the abovementioned protocol and determined that it is exempt from full committee review based on a DHHS Category 5 exemption.

Any changes in your research activity should be promptly reported to the Institutional Review Board and may not be initiated without IRB approval except where necessary to eliminate hazard to human subjects. Any unanticipated problems involving risks to subjects should also be promptly reported to the IRB.

The IRB would like to extend its best wishes to you in the conduct of this study.

Sincerely,

Mr. Michael A. Hripko
Associate Vice President for Research
Authorized Institutional Official

MAH:cc

c: Dr. Kriss Schueller, Chair
Department of Computer Science & Information Systems

