Fall 12-2-2020

Representational Learning Approach for Predicting Developer Expertise Using Eye Movements

Sumeet Maan
University of Nebraska-Lincoln, sumeetmaan@huskers.unl.edu

REPRESENTATIONAL LEARNING APPROACH FOR PREDICTING DEVELOPER EXPERTISE USING EYE MOVEMENTS

by

Sumeet Maan

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Bonita Sharif

Lincoln, Nebraska

November, 2020

REPRESENTATIONAL LEARNING APPROACH FOR PREDICTING DEVELOPER EXPERTISE USING EYE MOVEMENTS

Sumeet Maan, M.S.

University of Nebraska, 2020

Adviser: Bonita Sharif

The thesis analyzes an existing eye-tracking dataset collected while software developers were solving bug fixing tasks in an open source system. The analysis is performed using a representational learning approach namely, Multi-layer Perceptron (MLP). The novel aspect of the analysis is the introduction of a new feature engineering method based on the eye-tracking data. This is then used to predict developer expertise on the data. The dataset used in this thesis is inherently more complex because it is collected in a very dynamic environment i.e., the Eclipse IDE using an eye tracking plugin, iTrace. Previous work in this area only worked on short code snippets that do not represent how developers usually program in a realistic setting.

A comparative analysis between representational learning and non-representational learning (Support Vector Machine, Naïve Bayes, Decision Tree, and Random Forest) is also presented. The results are obtained from an extensive set of experiments (with a 80/20 training and testing split) which show that the representational learning (MLP) works well on our dataset reporting an average higher accuracy of 30% more for all tasks. Furthermore, a state-of-the-art method for feature engineering is proposed to extract features from the eye-tracking data. The average accuracy on all the tasks is 93.4% with a recall of 78.8% and a F1 score of 81.6%. We discuss the implications of these results on the future of automated prediction of developer expertise.

ACKNOWLEDGMENTS

I have received plenty of support and guidance throughout the writing of this thesis. First, I would like to thank my advisor, Dr. Bonita Sharif, and my collaborator Dr. Mohammad Rashedul Hasan, whose expertise was crucial in articulating the research questions and methodology. Your insightful feedback pushed me to hone my skills and brought my work to a higher level. You guided my thoughout the program and provided constant guidance and direction which helped me successfully completed my thesis. I want to acknowledge my colleagues from the SERES lab who helped me whenever needed.

Besides, I would also like to thank Dr. Jitender Deogun, who was always there for moral support and had my back whenever I needed him. I am thankful to my family and friends for their wise counsel and sympathetic ear.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is said that eyes are windows to the soul. Eye movements of a person speak the state of their cognitive ability and skill level of the task being performed[25, 23, 28]. An eye tracker (a combination of hardware and software) is used to collect eye movement data while a participant works on some task. In this thesis, the participant is a software developer and the context and task is that of fixing bugs. The eye-tracking data gives fine granular details such as pupil diameter, gaze duration on a particular element, which can be an identifier such as variables, conditional statements, function or class declarations, and even the coordinates of the elements which are looked at. Analyzing such fine details helps the researchers comprehend the developers' behavior while solving the task [10, 23, 32]. For instance, Busjahn et al. found that novice developers read code more linearly as compared to the expert developers[10]. Kevic et al. analyzed the developer's detailed navigation behavior for realistic change tasks using eye-tracking data and identified a distinct pattern in the eye-movement of expert users [23].

Results attained from the previous studies propose that the pattern in the gaze data of developers' vary as a function of expertise level. The modern AI techniques can learn such functions automatically. Lee et al. built an ML model to predict developers' expertise (expert/novice) and the difficulty of the task(easy/difficult).

They used eye-tracking data and also data from electroencephalography (EEG) to feed into the ML models. They used Support Vector Machines(SVM) to classify the skill level and task difficulty. Although such a solution is significant towards automating the prediction-based analysis, it does not extend to real-life practical debugging tasks. The reason behind the prior statement is that the developers' tasks that they used were very simple and significantly easy. Figure 1.1, 1.2 shows a sample task used in [42]. It contains a single method with a very simple and small task to be performed. Also, the tasks' difficulty level were very different from each other, and all the tasks were stored in a single file. As a consequence, the model was capable of learning the pattern behind eye-tracking data and EEG data to predict developers' expertise/task difficulty. The way software is developed in the real-world is far from just viewing 10 lines of code. In reality, software developers need to work with thousands of lines of code spanning across several hundred files. The dataset [23] we use mimics this scenario as closely as possible with respect to how a developer would fix a bug in a realistic setting. This also makes it harder to reuse existing feature models. The challenge we faced was coming up with a unique feature engineering model that works well for realistic development scenarios. In order to do this, we needed to take advantage of semantic properties within the source code itself and map those to the gaze recorded on those regions.

In a realistic setting, a developer has to solve complex tasks. A realistic change task could impact several classes and methods which are spread across multiple files[23] (see an example in Figure 1.3). Hence, the ML model would need more distinctive features to learn the hidden data pattern. As far as we know, **no ML-based predictive model is developed, which predicts developers' expertise using eye-tracking data from realistic change tasks**.

```
1 public static void main(String[] args) {
2   String word = "Hello";
3   String result = new String();
4
5   for (int j = word.length() - 1; j >= 0; j--)
6     result = result + word.charAt(j);
7
8   System.out.println(result);
9 }
```

Figure 1.1: Simple comprehension task with expected output 'olleH' [42].

```
1 public static void main(String[] ) {
2   String word = "Hello';
3   String result = new String();
4
5   for (int j = word.length() - 1; j >= 0; j--)
6     result = result + word.charAt(j);
7
8   System.out.println{result);
9 }
```

Figure 1.2: Syntax task with errors in Line 1, 2, and 8 [42].

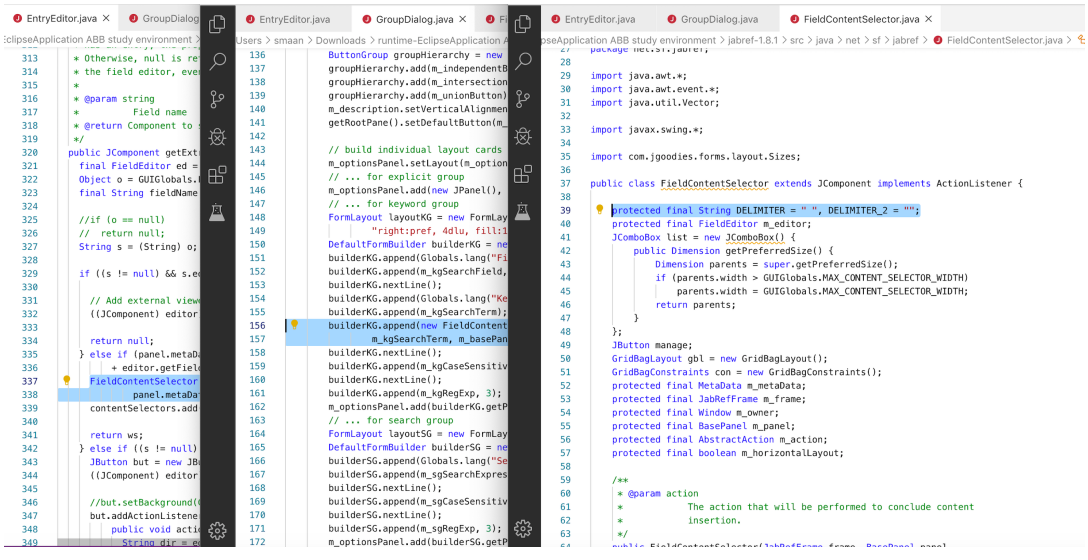

Figure 1.3: Complex task spanned across multiple files and methods.

1.1 Motivation and Challenges

Predicting developer expertise is an important problem in the software development industry. If automated tools could be developed that predict developer expertise based on the task, such a tool could be used to choose and recommend appropriate developers based on the task. It could also be used during interviewing to determine if a particular candidate is close to solving a task. This research aims to present an ML-based approach to predict the developer's expertise level based on eye-tracking data generated from realistic change tasks. These bug fixing tasks differ in the number of files per task, the number of classes, complexity, and number of code lines. This is a specifically challenging problem due to **two major components**. Primarily, it is non-trivial to create features from raw gaze data [8]. Although the data is temporal, it contains non-linear elements in many forms. The non-linearity arises from the reading pattern [8]. Another factor that contributes to non-linearity is that the developer switches between the compiler messages and source code. She might also look at the empty spaces while thinking, which could further create noise in the data. The amount of raw gaze data varies in accordance with the complexity of the bug fixing task. Therefore, we don't have a constant feature dimension for all the tasks.

The above mentioned challenge is addressed by creating a novel method for extracting features from eye tracking data. After the feature engineering process, the processed data is used to feed into the models. In our analyses, both representational learning and non-representational learning methods were used. The representation learning technique involves learning representations of data by either extracting features or transforming them, which makes the classification/prediction task easier [18]. A good representation is able to learn the hidden pattern behind the data by learning

the cause of variability that explains the structure of the distribution. It learns the representation of the data, and once it does that classification becomes trivial. For predicting the expert level based on simple tasks (e.g., one line bug fixes in just one method), non-representational techniques such as SVM have been shown to work effectively [26]. However, for complex tasks (e.g., bug fixes that span multiple methods and files), the representational learning models give the best performance because these models have in-built algorithms for extracting and learning features from the dataset which non-representational models do not.

The models developed in this thesis are trained from the data obtained from the study by Kevic et al. [23]. The data consists of raw gaze data of 22 participants. Although the model's scope is restricted by the size of the data set, **this is the first step towards predicting developers' expertise in software engineering for realistic bug fixing tasks using their eye movement data**.

1.2 Contributions

The main contributions of this thesis are follows:

1. A novel feature engineering method was developed based on eye movements from developers fixing realistic bugs.

2. A representational learning approach for predicting expertise of developers using eye-tracking data is developed.

3. A comparative analysis on the eye-tracking dataset was done between representational and non-representational learning.

1.3 Organization

The remainder of the thesis is organized as follows. First, relevant literature is reviewed in Chapter 2 followed by a description of the study for generating the eye-tracking dataset in Chapter 3. The feature engineering method and the ML framework are presented in Chapter 4. Then, an extensive experimental analysis of the framework is provided in Chapter 5. Various aspects of the approach are discussed in Chapter 6. Finally, the paper is concluded with a summary of the observations and discussion of future work in Chapter 7.

1.4 Publication Note

Parts of this thesis will be written up for publication to conferences and journals in the field of software engineering and machine learning.

Chapter 2

Related Work

We present related work in three major areas. In the first section, a selected overview of eye tracking studies in software engineering is presented. This is followed by software engineering studies that have worked with expertise prediction albeit with simple tasks. This is followed by related work about eye tracking and expertise prediction in other domains. Finally the concluding section of this chapter we provide a discussion about how our work differs from the related work done in the field.

2.1 Program Comprehension Studies Using Eye Tracking In Software Engineering

Since 2006, there has been a surge in the number of eye-tracking studies involving program comprehension in the software engineering domain [32]. While the programmer is solving a bug, an eye tracker is used to gauge her eye movements. It gives fine granular details about the fixation (where the developer has looked at) such as her pupil diameter, duration, which element she looked at in the code even if she scrolled the screen. These details provide an insight into his mental model while she was solving the bug.

Sharafi et al. [41] made a one-stop solution for people who want to conduct

eye-tracking studies. They presented when, how, and why eye trackers should be used to conduct studies in software engineering. They established that eye trackers provide rich and granular data that can be used to make useful findings in software engineering research, such as the mental model of the participants based on the mind-eye hypothesis. Collecting such data is not possible with the help of surveys or fMRI. Also, eye trackers are suitable for software engineering due to the fact that they can utilize visual attention artifacts.

Busjahn et al. [10] designed global and local measures based on gaze to characterize linearity while reading source code. Their results showed that experts read the source code more non-linearly as compared to novices. And novices read source code more non-linearly as compared to natural language. Their results reveal the reading patterns between experts and novices and that reading behavior does change after gaining expertise.

Naser et al. [29] made a comparative analysis between an existing data set of source code eye movement and natural language using the E-Z reader model [36] of eye movement control. The results indicated that source code made fine predictions of eye movements using the E-Z reader model. The results were further confirmed by doing a comparative analysis between model predictions and eye movement data by calculating correlation values for every metric. They also found that gaze duration is affected by token frequency in natural text and source code.

In [39], Saddler et al. conducted a study with 30 participants, both students, and professionals who read Stack Overflow posts. While reading the posts, the participants also answered API comprehension questions by summarizing API elements without showing them the source code. They found differences in gaze behavior between the participants familiar with Stack Overflow or API in the question versus those who were not familiar. They found that those familiar with Stack Overflow

spent lesser time per page fixating on code, paragraphs, and the overall page. Although there was not much difference found in accuracy between students and professionals, differences did exist in the gaze behavior that depended on the page content.

In [4], Abid et al. and her team conducted a study with 18 experts and novices where they had to read and summarize Java methods. They used varying sizes of methods in order to gauge the impact of the length of the method on the reading behavior of developers. They found that the signature of the method was not visited as much by both novices and experts. Also, both groups spent considerable gaze time and had more gaze visits while they read call terms. Also, both the groups revisited the control flow terms rather than focusing and memorizing them by reading them for a longer time. These results were different compared to Rodeghero's work (that Abid et al. replicated) that indicated that the signature of the method was the most looked at item. Rodeghero et al. [37] did, however, use small source code snippets, whereas Abid et al. used large open-source systems without restricting to small methods that fit on the screen. This shows that given a realistic setting, prior results do not always hold.

Barik et al. [6] conducted a study to find out if developers read the error messages in Eclipse IDE. The tasks were chosen from prior work done by the team. They picked frequently occurring costly error tasks from around 26 million builds from Google. They picked up ten error messages from all the categories. Since they did not have access to Google's code, they injected the bugs into Apache Commons Collections. The participants were asked to identify and resolve ten source code defects, which were presented in the IDE in the form of compiler messages. Each participant had 5 minutes to solve a task and was asked to provide a reasonable solution to the bug. After the analysis of the data collected, they found out that the developers do read error messages, but the difficulty in comprehending them is similar to that of reading

source code, which hinted that it was an intensive cognitive task. They analyzed the revisit count of error messages and found that due to difficulty in reading and comprehending error messages solving the task, all in all, can be difficult.

2.2 Expertise Prediction Using Eye Tracking on Software Tasks

We discuss related work done in the expertise prediction area for software tasks. As evidenced from this section, we were only able to find a few related works in this area.

In [23], Kevic et al. analyzed the eye-tracking data and found that it is more detailed and finer granular than the interaction data(such as mouse and keyboard clicks). Eye-tracking data is capable of providing perception about how developers read code(linear or non-linear way). While the authors did not find any significant differences between novices and experts, several metrics insinuated the underlying differences in the eye-tracking data of inexperienced and experienced developers [22]. To verify and analyze this further, the dataset from this study has been used as the basis of further analysis done in the thesis. Sophisticated ML techniques have been used to gain further insight into the dataset, which is not done previously in [23] and [22].

Some researchers implemented other machine learning models, such as the Naïve Bayes ML model. Fritz et al. [16] used a Naïve Bayes classifier and compared the results derived from various psycho-physiological sensors such as eye tracker, electroencephalography sensor(EEG), and electrodermal activity sensor(EDA). The authors figured out that an electroencephalography sensor(EEG) gives the optimal precision and an eye tracker gives the optimal recall. They also found that an electrodermal activity sensor(EDA) combined with an eye tracker is optimal to predict task difficulty

in real-time while the developers are coding.

Lee et al. [27], and her team designed and developed an experiment based on EEG (electroencephalogram) in 2016. They observed developers with the sensors' help and recorded data while the developers performed program comprehension tasks. The authors were able to clearly distinguish between experts and novices. The experts clearly had higher brainwave activation than novices. The results hinted that experts have incredible skills to solve program comprehension tasks efficiently. Later Lee et al. [26] conducted a study with 38 participants consisting of both experts and novices to investigate if an eye-tracker and EEG (electroencephalogram) can be used to predict task difficulty(easy/difficult) and user expertise(expert/novice). They used Support Vector Machines(SVM) and were able to predict task difficulty with 64.9% precision and 68.6% recall; and programmer expertise with 97.7% precision and 96.4% recall. Although they were able to make predictions but the tasks used were very simple and could not be generalized to the real world.

Bednarik et al. [8] used SVM to predict the user's performance and problem-solving cognition states of the user while they played an 8-tiles puzzle game. The data used was collected from a previous study done by the author. The gaze data were mapped to human cognition states by linking them to footnotes of the think-aloud protocol. The features were extracted from eye movement data and then fed into the model for prediction. The system predicted the performance of the user's problem-solving behavior with 79% accuracy.

Liu et al. [28] analyzed eye-tracking data, which was collected from users performing a collaborative task, and applied machine learning models to predict the skill level of the participants. The data had 64 first-year students(46 male and 18 female) and were randomly assigned into pairs. They were asked to read texts and build the concept map of the subject in the text. The results showed that they were able to

predict the expertise level of users with 96% accuracy by applying the Hidden Markov Model(HMM) with only one minute of eye gaze data into the experiment.

Prior work shows that novice programmers have difficulty in learning new concepts and keeping track of their progress. Beck et al. [7] tried to predict students' metacognition levels with the help of source code comments. The data set consisted of 98 student's lab assignments, which were in Python. They fed the data to a multinomial logistic regression classifier and achieved 88% accuracy. The results hinted that a real-time feedback system could be developed for introductory programming courses.

Lalle et al. [25] conducted a study with MetaTutor, an Intelligent Tutoring System(ITS) which provided hints, prompts to students for adaptive learning. However, there have been prior studies done which show that such systems have a negative impact on students, such as frustration, boredom, etc. They collected eye movement data while students interacted with MetaTutor and tried to predict students' achievement goals and emotional valence in students. Boosted Logistic Regression (BL) classifier was trained with eye movement data for real-time prediction of students' achievement goals and achieved 81% accuracy. SVM was used to predict emotional valence(positive or negative), and the classifier achieved 64% accuracy. The results suggested that students can learn from such systems if the system can rectify the negative episodes which impact student's achievement goals.

2.3 Expertise Prediction Using Eye Tracking in Other Domains

Researchers in other domains have also applied the modern deep learning techniques to eye-tracking data for classification of expertise. The authors in [11] used Convolu-

tional Neural Network(CNN) and applied it to eye-tracking data, which was gathered while dentists were viewing OPT. They were able to classify the expertise of dentists with 93% accuracy. The authors used the image patches and linked them to their respective fixation while the dentists viewed the OPT. They fed this as an input to the CNN model.

Ahmidi et al. performed classification based on skill level by creating 14 different Hidden Markov Models for seven surgical tasks. The authors targeted both expert and novice levels and used a repeated k-fold cross-validation method. Six novices and five experts performed 14 trials, which summed up to a total of 95 data points generated from expert surgeons and 139 tasks data points cumulated from novice surgeons. The authors cleaned the data by removing irregular procedures. They achieved an accuracy of 77.8% in surgical task prediction, and 82.5% in surgeon's skill level prediction [5].

The authors in [15] combined SVM and computational modeling techniques of machine learning and applied it to eye-tracking data to predict problem-solving behavior. In the study, the think-aloud method was used while the participants were solving an 8-tiles puzzle game. Such a method allows the researchers to understand the participants' mental models while they were solving the problem. Jerman et al. used machine learning on eye-tracking data to discover expertise and coordination in a collaborative Tetris game setting [21]. The authors found that the game players tune their behavior if they interact with an expert player. Machine learning was also used to develop a real-time feedback system for novice developers [7]. In [43], Steiche et al. used classification of the visual tasks to predict the properties of performance, user's visualization task, and individual cognitive abilities such as visual working memory, and perceptual speed, and verbal working memory.

2.4 Discussion

In summary, researchers have used various machine learning techniques on eye-tracking data to predict user expertise and gain insight into user behavior patterns and mental models based on certain tasks in software engineering. The only catch was that the tasks were quite simple [28, 7, 25]. In order to bridge this gap, the research presented in this thesis has used data generated from realistic bug fixing tasks taken from an open-source repository. These tasks were quite complex (see Figure 1.3). This fact makes the findings of the study generalizable to the real world where developers solve such tasks.

Chapter 3

Dataset

This research aims to predict developer's expertise by using AI techniques on eye-tracking data. This data is generated while the developer solves realistic bug fixing tasks. This research has used the data set generated from the study in [23]. The study has eye gaze data of 22 developers who tried to solve three bug fixing tasks labeled Task 2, Task 3, and Task 4 in one hour. Each task was given a total of 20 minutes. The details about the bug tasks taken verbatim from open source systems are mentioned in Figure 3.1. It can be seen in the Figure 3.1 that Task 2's scope is spanned across multiple classes. But Task 3 and Task 4 scope is in a single method. There was an additional task used as a tutorial to familiarize the participants to the system and environment before they began the study. That practice task is not used for training the model. Please refer to Kevic et al. [23] for more details on the study.

ID	Bug ID	Date Submitted	Title	Scope of Solution in Repository
T2	1436014	2/21/2006	No comma added to separate keywords	multiple classes: `EntryEditor`, `GroupDialog` `FieldContentSelector`, `JabRefFrame`
T3	1594123	11/10/2006	Failure to import big numbers	single method: `BibtexParser.parseFieldContent`
T4	1489454	5/16/2006	Acrobat Launch fails on Win98	single method: `Util.openExternalViewer`

Figure 3.1: JabRef Bug Details taken from [23]. The Task IDs are kept the same and referred to as Task 2, Task 3, and Task 4.

The study has 22 developers, out of which there were 12 students (labeled as

novices), and 10 were industry professionals (labeled as experts). The bugs were taken from JabRef. It is an open-source project available on SourceForge[1] related to reference management. It contains 38 KLOC approximately spread across 311 files. The study used version 1.8.1 of JabRef - release date was 9/16/2005. The authors in the study chose bugs that were already fixed in the system. That way, they ensured that they knew exactly how these bugs were fixed by the original developers. An older version (1.8.1) of the JabRef was used by the authors in the study so that the bugs could be reproduced.

As mentioned earlier, there was a total of three tasks that needed to be completed in an hour by the participants. Each task was 20 minutes long. While the participants were solving these tasks, their eye movements were recorded with a screen based eye tracker [2] and the iTrace community infrastructure [40, 19]. When the study was conducted, iTrace was only available as an Eclipse plugin used to capture the gaze data on source code elements. The gazes were captured in the presence of scrolling the page or switching between files. Since the study was conducted, iTrace has evolved into a community infrastructure supporting Eclipse, Visual Studio, Atom, and Chrome (see `www.i-trace.org` for more details). It provides very fine granular gaze details of the data. For instance, it gives information on which line the developer is looking at and the element she fixes her gaze upon.

Fixation is a widely used term in any eye-tracking research [14]. It is defined as the action of looking at something for some amount of time. It is the most sought after feature that researchers look for while analyzing any eye-tracking dataset. Fixation is that point in time when the user holds her gaze at a certain element while reading a stimulus (text or images). The user stops there to process information in the brain and that is what the researchers are interested in. The researchers are interested in the mental model of the user while she was solving the task. Practically, the fixation

is built from raw gaze points with the help of an event detection algorithm. There are many different versions of fixation filters available.

The data set generated from the study was in an XML format that had raw gaze information about the participants. To remove any invalid gazes and cluster gazes into a single fixation, these files were run through a fixation filter [34] available in iTrace. The fixation filter which forms clusters of raw gaze data with the help of line and column information in the source code files. If the line and column information is same for consequent rows, iTrace clusters it and forms a fixation row in the resultant fixation file. The filter also removes gazes below threshold value. The threshold value in the filter is 60ms and all the gazes below this value are removed. The reason behind removing such gazes is that any gaze lower than this value cannot contribute to any realistic cognitive process. After running the filter, the files generated are in CSV format. Each row in the CSV represents one fixation and the data is recorded in time as fixations occur during the task. The CSV file contains several columns related to various eye tracking data such as gaze validity and pupil diameter. The most important attributes of the CSV file are explained below.

1. *Coordinates x and y:* This metric tells the x and y coordinates of the gaze on the screen.

2. *Fixation Duration:* It is defined as the time period in which one maintains gaze on some element and is measured in milliseconds (ms).

3. *Line and Column number:* iTrace records the line and column number in the source code file at which the participant looked at during the fixation. The line and column is derived from the x and y coordinates on the screen.

4. *Pupil Diameter:* iTrace records both left and right pupil diameter. Pupil diam-

eter is a good metric to gauge if the participant was focused on the fixation or not. A dilated pupil means that the person is trying to focus on something and a relaxed pupil means that he is not as focused. The catch is that it varies from person to person. Hence, it becomes difficult to come up with a threshold value that defines focus on the fixation. We do not use this feature in our analysis at this time.

5. *Fully Qualified Names:* iTrace gives the fully qualified names of the elements looked at. For instance, there is a for loop that a person looked at and it is present in a class $->$ method $->$ for. iTrace will record the <`name of the class`> . <`name of the method`>`.for` in the specified format. For instance, let's consider `net.sf.jabref.Util.sortWordsAndRemoveDuplicates` as an example. Here, the words separated by dots represent hierarchy looked at in descending order. In this case `net` is the outermost package and `sortWordsAnd RemoveDuplicates` is the method looked at which is present in `Util` class which is in turn present in `jabref` package.

Out of all these features, line number and fixation duration were used to engineer and extract the features. Then the final input file was fed into the model to perform predictions on developer expertise.

Chapter 4

Methodology

The thesis aims to predict a developer's level of expertise for realistic bug fixing tasks. Thus, the research problem is formulated as a **binary classification** problem. The Machine Learning pipeline used to predict the expertise is shown in 4.1. In this pipeline, Data pre-processing and Feature Engineering tasks are done manually. Each of the steps on the pipeline are explained in detail in this chapter.

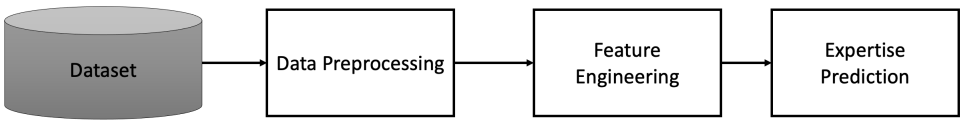

Figure 4.1: Machine Learning Pipeline for Predicting Developer's Expertise Level

4.1 Data Pre-processing

The data labels (target features) of the 22 participants were determined first. Then feature extraction is done.

Determining Data Labels: Data labeling was required to train the classifi-

cation models of the framework. Nonetheless, it is non-trivial to find an appropriate label for the debugging tasks. In [23], students were classified as novices, and professionals were classified as experts. However, just having a certain number of years of experience does not make one an expert in solving a task. An expert could be good at debugging one task but might not be able to solve some other task. It is not possible that a developer who claims himself to be an expert could solve all the tasks. The same hold for the novice developer. There have been instances where the novice developers have performed better and are able to solve the task better than expert developers. To verify this fact, a comparison was made with the developers' level of expertise to whether he could correctly solve the debugging task.

The study in [23] stored the metadata of the study results where the authors have reported the task correctness of the developers after the study was completed. In the study, the students were labeled as novices and industry professionals as experts. A developer's task correctness was stored as 0 and 1. 0 means that he was not able to solve the task and 1 means that he was able to solve the task correctly. There was a detailed investigation done that whether the experts were able to solve the tasks correctly and we found that not all the experts were able to do it. In fact there was a mismatch and some novices were able to solve the task correctly. Hence, task correctness was used as a label to determine the expertise level of developers which meant that if a participant is able to solve the task, he is an expert in it. Depending on the task's complexity, a developer's level of expertise(task correctness) could vary. Hence, as shown in Table 4.1, the number of labels vary across three tasks. The beacons and features are explained in the sections that follow.

Table 4.1: Information about the Three Bug Fixing Tasks in JabRef

Attribute	**Task 2**	**Task 3**	**Task 4**
No. of Experts	8	7	11
No. of Novices	14	14	10
No. of Files Impacted	Multiple	Single	Single
No. of Methods Impacted	Multiple	Single	Single
No. of Lines of Code	3321	771	1268
No. of Beacons	73	60	97
No. of Features	219	180	291

4.2 Feature Engineering

The major challenge faced during this research was defining features for the ML model. Previous work [23, 10] indicates that experts differ from novices based on where they focus their attention at, the variation of the fixation, how they navigate the code, and for how much time they looked at the code. This key observation was used during feature engineering.

The eye-tracking data was recorded in an increasing sequence of time while the developers solved the bug fixing tasks. Then, the fixation files were generated by running the data through the fixation filter available in iTrace to detect fixations. The iTrace filter also removes any invalid data. Invalid data could be present due to many reasons. For instance, while a developer solves a task, the iTrace records whether the fixation was looked at with both eyes or one eye in the raw gaze data. Data with one eye is still considered valid. However there might be cases where a gaze is not recorded correctly and this is classified as invalid data. These are marked in the raw gaze files and are discarded as part of the fixation filtering process.

Later, when we pass the raw gaze data through the fixation filter, it removes any invalid data which was looked at with just one eye. iTrace also removes fixations whose

duration is less than the threshold (60 ms). The fixation filter also makes clusters of data which are consecutively at the same gaze point or at the same element. It adds the fixation duration for each fixation after the clustering raw gazes together.

Even after the filtration is performed, the fixation files can have noisy data, which is not useful. Useful information from the fixation files was extracted by identifying the most looked at regions. We use the most looked at regions because these were considered to be most relevant to the task for a majority of the developers. This process was done using a sorting script. After identifying the regions, further, these regions were divided into logical segments of code. Logical segments can be defined as a block of code of related comments, declaration, and statements. We also refer to these logical segments as beacons [45]. After this process, fixation related values are calculated from these logical segments. The detailed steps of the feature engineering process are described below as shown in Figure 4.2.

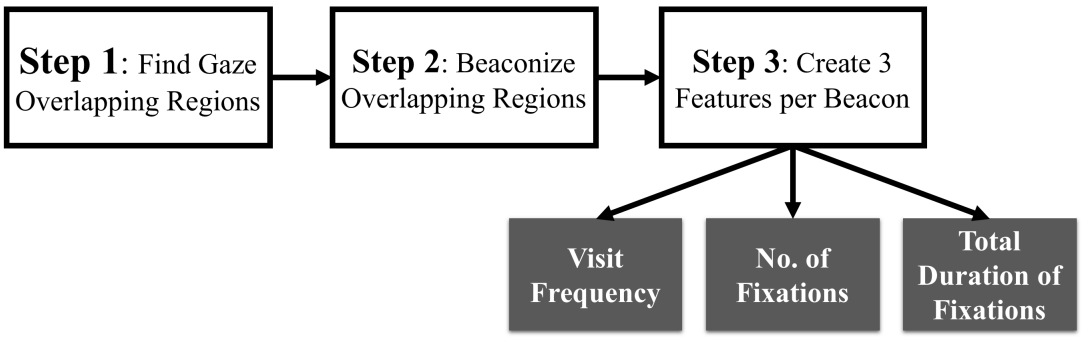

Figure 4.2: Feature Engineering Pipeline

Step 1: Find gaze overlapping regions in the code: The first step towards finding the most viewed regions in the code is to analyze the fixations files and mark all the regions in the actual code looked by all the participants. These regions can be identified with the help of line numbers in the fixation files. All the tasks had a different set of files. The next step was to identify the frequency of the regions. For

instance, there may be a region in the code that is looked at by 10 participants versus a region that is just looked at by one participant. A threshold limit was decided and agreed upon based on the complexity of the tasks. For some tasks, regions were selected on the threshold limit greater than two, and for some, it was greater or equal to 4. Once these regions were marked, then beaconization of the regions was done, which is explained in the next step.

```
58    /**
59     * Check whether the source is in the correct format for this importer.
60     */
61    public static boolean isRecognizedFormat(Reader inOrig)
62      throws IOException {
63      // Our strategy is to look for the "PY <year>" line.
64      BufferedReader in =
65        new BufferedReader(inOrig);
66
67      //Pattern pat1 = Pattern.compile("PY:  \\d{4}");
68      Pattern pat1 = Pattern.compile("@[a-zA-Z]*\\{");
69
70      String str;
71
72      while ((str = in.readLine()) != null) {
73
74        if (pat1.matcher(str).find())
75          return true;
76      }
77
78      return false;
79    }
```

Figure 4.3: Partial/Sample code in BibtexParser.java used to illustrate beaconization

Step 2: Beaconize the overlapping region in the code: Beaconization is performed manually after identifying the most looked at overlapping region in the code. The smallest piece of a logical segment in the code is called a beacon. Beacons render the most logical and granular information of the program. A beacon may contain a block of variable declarations, logic, comments, or method names. In [13], the author has shown that the beacons for expert developers are different from the novice developers. A more detailed description on beacons can be found in [45].

The method for beaconization is illustrated in Figure 4.3 using the sample code.

This code is from the JabRef project. Every beacon is associated with a group of line numbers. These line numbers need to be in sequence to qualify for a beacon. For instance, the lines from 58-60 have a multi-line comment and are considered as a beacon (first row in Table 4.2). Please refer to Table 4.1 for the number of beacons per task.

Table 4.2: Identifying Beacons from Figure 4.3

File	Line	Beacon ID	Rationale
BibtexParser.java	58-60	b1	Multi-line comment
BibtexParser.java	61-62	b2	Method declaration
BibtexParser.java	64-65	b3	Variable declaration
BibtexParser.java	67-68	b4	Variable declaration
BibtexParser.java	70	b5	Variable declaration
BibtexParser.java	72	b6	While loop
BibtexParser.java	74-76	b7	If statement
BibtexParser.java	78-79	b8	return statement

Step 3: Create features from beacons: The beacons are the logical section of code capable of consisting of the most distinguishing information about developers' level of expertise. Fixation is the fundamental attribute in every beacon, i.e., how long a person has maintained his gaze at a certain point. In the prior work, it is shown that the level of cognitive processing is indicated by the fixation duration [17]. For instance, long fixation duration indicates a deeper processing level, but shorter fixation duration could mean superficial information processing. It has also been shown that the order of changing fixations, frequency of the fixation, and the fixation duration of the beacons, are meaningful metrics to gauge the developer's behavior [23]. These foundational metrics, which are already established in the prior studies, are used to create three features per beacon: visit frequency, number of fixations, and the total duration of fixation.

Let us understand the method of creating the features as mentioned above with

the help of an example code in Figure 4.3 and use the fixation file in Table 4.3. The description below illustrates the extraction of features for beacon id 7.

Table 4.3: Fixations on lines in BibtexParser.java looked at in time.

Index	File	Line	Fixation Duration (ms)	Beacon ID
1	BibtexParser.java	58	63	b1
2	BibtexParser.java	61	120	b2
3	BibtexParser.java	62	90	b2
4	BibtexParser.java	74	85	b7
5	BibtexParser.java	75	65	b7
6	BibtexParser.java	78	60	b8
7	BibtexParser.java	74	100	b7
8	BibtexParser.java	74	70	b7

1. *Visit Frequency:* This metric tells us the number of times a participant looked at (visited) a beacon. The visit frequency is 2 for b7 in this example. The developer was on index 4 and 5, and then on index 7 and 8, he revisited the same beacon (see Table 4.3). The value of the visit frequency evaluates to 2. As you can witness that if the developer stays on the same beacon for n number of lines, then the value of frequency will be considered as 1 and not n. For instance, since the developer was successively on index 2 and 3, the value of visit frequency for beacon b2 is 1. b2 was never visited again.

2. *No. of Fixations:* It is defined as the total number of fixations any developer spends in a beacon. The total number of times the developer fixates on b7 is 4, which is at index 4, 5, 7, and 8.

3. *Total Duration of Fixations:* It is defined as the sum total of the fixation duration in a beacon. The sum of the duration of fixation at index 4, 5, 7, and 8 is $320ms$; hence it becomes the value for this feature metric.

The value for all the beacon features will be 0 if the participant does not visit that beacon. After performing all the calculations mentioned above, a feature vector is formed by collating all the beacons and its associated features sequentially. To sum up the feature engineering process, the raw gaze data is processed as a fixation file after passing it through iTrace filter. Then beacons are extracted out of it. After that, the three features are created from the beacons. Each bug fixing task has a different set of features. Refer to Table 4.1 for the number of beacons per task. In this way, a separate data matrix is created for every task. The columns are represented as features in the input matrix, and rows are represented by users/participants.

4.3 Model Selection

The experiments were performed with two types of learning models namely representational and non-representational. While picking the models, the initial choice was SVM. It is one of the most common model used and hence it was one of the most obvious picks. Looking at the high non-linearity in the dataset, Decision Trees, Random Forest, and ANN were also looked at . The features created in the feature engineering step are directly used by the models after scaling.

4.3.1 Non-representational Learning

First, it is shown that non-representational learning models don't produce optimal results on complex tasks. The following non-representational learning models are used: Naïve Bayes Classifier, SVM Linear Classifier, Kernelized SVM (Gaussian Radial Basis Function) [12], Decision Tree [35], and Random Forest (RF) [9].

4.3.2 Representational Learning

The Feed-Forward ANN model is used to perform optimal classification. More specifically, a Multi-Layer Perceptron (MLP) is used due to its inherent capability to learn representations using layers of hidden neurons [38, 31]. The following hyper-parameters of the MLP are tuned: number of hidden layers and neurons, activation function, solver, learning rate, and regularization. The hyper-parameters were tuned manually by plugging in values one by one to select the optimal model.

Chapter 5

Results and Analyses

The experiments are conducted to evaluate the performance of the ML models. At first, the dataset is preprocessed, features are manually engineered, and the data matrices are obtained for each task. Then, suitable learning ML techniques are used for training using the task-based data matrices.

5.1 Non-Representational Learning

5.1.1 Experiment Setup and Overview

The experiments were conducted with the following non-representational models: Support Vector Machine(Linear and Gaussian), Naïve Bayes(Gaussian), Decision Tree, and Random Forest. The models were chosen based on the fact that there was a combination of linear and non-linear data. All the models are non-linear except SVM-linear. The selection of the variety of models provided an insight that which tasks perform well on which models. Apart from it, the corresponding hyperparameters to the models were tuned by performing a grid search on the hyperparameters. F1 score was selected to pick the best model since it works well for imbalanced class distributions. It can be seen in Table 4.1 that the number of experts are way less than the number of novices. To balance this, class weight was also tuned during grid

search. Class weight becomes an important parameter when the data set is imbalanced. Also, it was required in this case, as the number of sample points are very low. The dataset was divided into 80-20 split of training and test set respectively. This was done randomly in every run while recording the average of performance metrics. The split was done by the method provided by SciKit Learn library. Since the dataset was small we had around 18 points in the training set and 4 points in the test set. The models were trained on the training set to provide the performance metric. After finding out the best model, the performance metrics were reported by running the model for 100 iterations for every trial. Finally, the grand total of the performance metrics was reported for 100 trials.

5.1.2 Performance Analysis

In this section, we present results of the non-representational learning approaches. Tables 5.1, 5.2 and 5.3 show results for accuracy, precision, recall, and F-measure for the SVM-RBF, SVM-Linear, Naïve Bayes-Gaussian, Decision Tree, and Random Forest.

5.1.2.1 Task 2

Based on the model's test accuracy in Table 5.1, the best model for Task 2 is Gaussian Naïve Bayes, with 63.3% accuracy. Although the test accuracy is the highest of all models, it is not as good. The other performance metrics are relatively low, which does not make it a suitable choice. Random Forest also reports the performance metric, which is comparable to Naïve Bayes.

5.1.2.2 Task 3

For Task 3, it can be seen in Table 5.2 that SVM-Linear has the highest performance metrics in all aspects. It has 75.6% test accuracy and 48.4% as its F1 score. The test accuracy is not bad, but the other metrics are quite low performing.

5.1.2.3 Task 4

For task 4, the best model based on test accuracy in Table 5.3 is Decision Trees with 50.9% accuracy. But, SVM-Gaussian has better recall and F1 scores than Decision Trees. Even with 50% accuracy, these models cannot be deemed acceptable to perform the classification.

Table 5.1: Performance Evaluation of Task 2 (in %)

Performance Task 2	**SVM-RBF**	**SVM-Linear**	**Naïve Bayes-Gaussian**	**Decision Tree**	**Random Forest**
Avg. Train Accuracy	91.0	91.1	83.1	72.1	79.6
Avg. Test Accuracy	61.7	61.9	63.3	58.2	61.1
Avg. Precision	0.0	33.6	38.2	17.7	36.6
Avg. Recall	0.0	24.9	34.3	19.7	32.9
Avg. F1 Score	0.0	25.7	32.8	17.1	31.1

Table 5.2: Performance Evaluation of Task 3 (in %)

Performance Task 3	**SVM-RBF**	**SVM-Linear**	**Naïve Bayes-Gaussian**	**Decision Tree**	**Random Forest**
Avg. Train Accuracy	100.0	100.0	89.0	100.0	93.0
Avg. Test Accuracy	64.1	75.6	61.0	64.9	67.6
Avg. Precision	0.0	55.7	31.0	36.3	30.0
Avg. Recall	0.0	48.0	27.9	33.1	23.9
Avg. F1 Score	0.0	48.4	26.4	31.2	24.4

Table 5.3: Performance Evaluation of Task 4 (in %)

Performance Task 4	**SVM-RBF**	**SVM-Linear**	**Naïve Bayes-Gaussian**	**Decision Tree**	**Random Forest**
Avg. Train Accuracy	90.6	87.8	77.2	85.6	68.3
Avg. Test Accuracy	47.1	31.4	35.0	50.9	43.3
Avg. Precision	44.7	36.3	30.5	47.5	41.4
Avg. Recall	70.8	44.4	26.6	41.8	39.6
Avg. F1 Score	52.0	34.8	25.4	40.1	34.8

It can be clearly seen that Task 3 has best performance metric in non- representational learning models. The reason that it performs the best of all the tasks can be attributed to the fact that task 3 had maximum number of participants who looked at the same region creating a well defined feature set. Also, the most looked at region was in the same file and in a single method with fewer lines of code.

But in the case of Task 2 and Task 4, the code was distributed in multiple files. The highest number of participants who viewed the most looked at region was 4 times lower than Task 3. Hence, the feature set was not as well defined in this case. The Task 2 and Task 4 were highly non-linear and complex due to the reasons mentioned above.

Hence, an alternative approach was found to get better performance metrics. It is possible that with more data points the results would have been different. After consulting with the eye-tracking literature in this field, it was found that most eye-tracking studies have between 9-25 participants so our study sample was quite representative of past participants. There is a good reason for this low number. First, the study needs to be done one at a time with each person. This takes a lot of time an effort however, the end results are a much more insightful dataset.

5.2 Representational Learning

5.2.1 Experiment Setup and Overview

The MLP models were trained via the Backpropagation algorithm and using adaptive moment estimation (Adam) optimization function [24]. Sigmoid is used as the activation function for the hidden layers and output layer. All experiments were performed using the Keras and Tensorflow 2.0 [3] frameworks. Table 5.4 shows the model architectures, hyperparameters and optimal values used in the experiments.

Table 5.4: Model Architecture & Hyperparameters Settings

Hyper-Parameters	**Values**
Hidden Layers	1
No. of Neurons	20
Optimizer	Adam
Activation Function (Hidden Layer)	Sigmoid
Activation Function (Output Layer)	Sigmoid
Epochs	50
Alpha	0.001

The results from binary classifier to predict the level of expertise of developers was inconsistent due to the following reasons. Primarily, there were only 22 data points, and the number of features was between 180 to 291, depending on the task. Looking at these metrics, it is not hard to estimate that due to data scarcity finding the pattern would be difficult for any model. The next problem is linked to the former problem, i.e., hyper parameter tuning is difficult because of fewer data points. The small-scale data also restricts the efficiency of the k-fold cross-validation process as separate validation sub-sets cannot be guaranteed.

To overcome this problem, the data set is randomly divided into 80-20% split

of train and test subsets. To find the optimal accuracy for the model, the validation set was extracted out of the training set with 90-10 split. The model is run for 100 trials, and inside each trial, there are 100 iterations. Each iteration runs the model and calculates its performance metrics using the test data. Test - train split is created randomly for each iteration. The performance metrics for all the experiments are recorded and saved for each iteration. After each trial (which comprises of 100 iterations), its average is calculated and recorded. And after all the trials, its grand average is calculated. Accuracy, precision, recall, and F1 score are used as the performance metrics for all the experiments.

Table 5.5: Performance Evaluation of MLP (in %)

Performance	**Task 2**	**Task 3**	**Task 4**
Avg. Train Accuracy	90.8	99.3	90.3
Avg. Test Accuracy	89.3	99.0	88.5
Avg. Precision	81.1	88.6	94.4
Avg. Recall	68.2	88.2	80.0
Avg. F1 Score	72.1	88.2	84.7

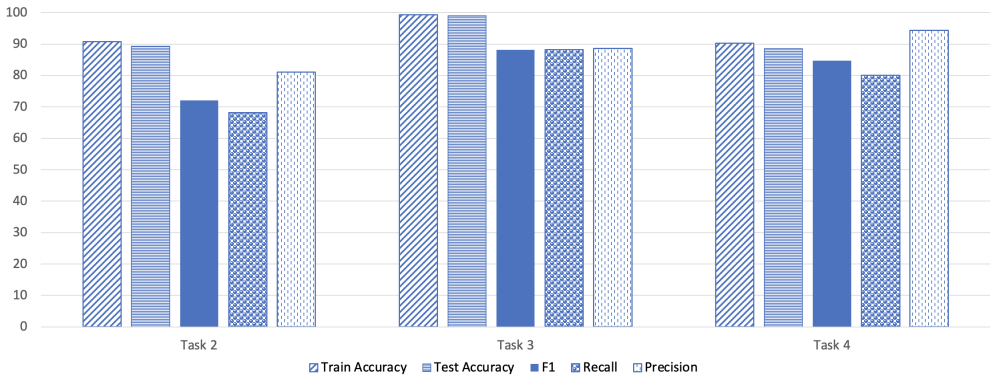

Figure 5.1: Average Precision, Recall & F1 Score of Three Tasks

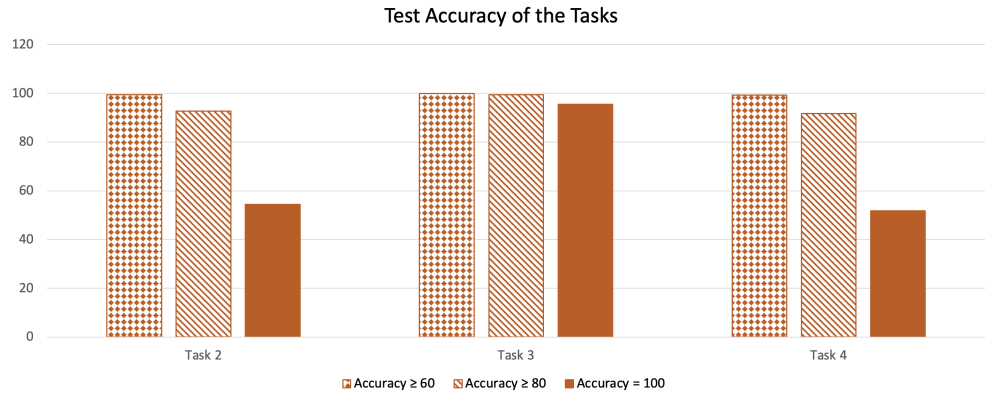

Figure 5.2: Number of Accuracy counts >60, >80 and = 100

5.2.2 Performance Analysis

The performance of ANN is quite high compared to the former non-representational models. A graph of all the performance metrics is presented in Figure 5.1. While calculating the average accuracies, number of times the accuracy was greater than 60, greater than 80 and equal to 100 was also recorded. This can be seen in Figure 5.2. It can be seen that number of times the accuracy was greater than 60 is almost the same for all the tasks. The number of times the accuracy was greater than 80 is higher for Task 3 and almost the same for Task 2 and Task 4. But the number of times the accuracy equal to 100 is highest for Task 3. For Task 2 and Task 4 it is the lowest of all the three. This shows that learning Task 3 was easier for the ANN model as compared to other tasks such that it was able to give 100% accuracy highest number of times.

Further the performance of MLP is shown in Table 5.5. A detailed analysis for each task is discussed below.

5.2.2.1 Task 2

The average test accuracy of Task 2 is 89.3% which is quite high compared to the highest accuracy of 63.3% in non-representational model. The other metrics of the model are also high. ANN's recall and F1 score is 68.2% and 72.1% respectively. This shows that ANN is very powerful and capable of learning the hidden patterns in the data.

5.2.2.2 Task 3

It can be seen that ANN works best on Task 3 with an exceptional test accuracy of 99% compared to Linear SVM's accuracy of 75.6%. It is the highest accuracy witnessed in all the three tasks. Its recall score and F1 score is both 88.2% and precision is 88.6%. Such high accuracy is due to the fact that it is stored in a single files and a single method. In addition to this, the overlapping regions are visited by the maximum number of participants. That's why it has a relatively smaller number of features. It's relative simplicity makes it possible for the single hidden-layer MLP to perform well.

5.2.2.3 Task 4

The average test accuracy is 90.3% with precision score of 94.4% compared to the best accuracy in the non-representational model of 50.9%.

Task 2 and Task 4 performs comparatively poorly than Task 3. The poor performance can be due to the high-dimensional feature space and the non-linear relationship of the features. For example, Task 2 and Task 4 have very high number of beacons compared to Task 3. Also, these tasks are spanned across multiple files and multiple methods. They also have very high number of lines of code. Due to these

facts, their performance metrics is less than Task 3.

Chapter 6

Observations and Discussion

The feature engineering presented in the thesis is a novel approach to extract the most discriminating features from eye tracking data. This feature set was used to predict developers' level of expertise based on gaze data for realistic bug fixing tasks. Furthermore, it can be seen that the representational learning models perform way better than the non-representational learning models. This is due to the fact that the former is capable of learning the functions with the help of extracting or transforming the features in the hidden layers.

Developing tools that leverage eye tracking helps understand the mental model of developers. Experts in one task cannot be considered a universal expert in solving all kinds of tasks. The ability to predict expertise in any task can help efficient allocation and utilization of human resources saving time to train new people by utilizing the existing ones. Such tools would also help companies to hire right kind of people to do a specific task. This is extremely helpful because organizations spend a lot of time and effort to find right people but due to lack of insight into strategic thinking while solving a task makes finding the right candidate difficult. Such tools can help predict expertise for a specific task at hand. Further recommendation systems can be developed and merged into IDEs which predict the expertise while a developer is solving a bug. In case the developer behaves like a novice at certain point, the system

can start recommending where the bug could be localized or prompt the developer to take a rest and start afresh. All in all, predicting expertise could be extremely beneficial where ever there is need to map expertise to any kind of software task.

While working on the thesis, three main challenges were identified and addressed to build an effective and resilient model. All these are linked to the same issue, i.e., data scarcity. The primary challenge is the lack of data. To address the inconsistency caused by this issue in the results, each model was run 10,000 times and then the average was calculated. The second challenge is the high non-linearity in the dataset. Many ML models work very effectively on non-linear data. But with just 22 data points and such high non-linearity it was very challenging for any ML model to make reliable predictions. The third challenge was the very high dimensions of the task. With just 22 data points, the final dataset's dimensions were between 180-291, which is extremely high. The root of all the above problems is data scarcity. Hence, the next step would be to collect more samples to further validate the research.

Applying the limited restriction on the dataset and the results obtained from them, an outline is presented for the future work and expansion of the research work done so far. It can be seen than Task 3 performs exceptionally well on all the models as compared to other tasks. Also Task 2 and Task 4 has similar performance metrics considering any model. This hints us to the fact that Task 3 can be classified as simple task and Task 2 and Task 4 can be classified as complex tasks. In Figure 3.1, it can be seen that Task 3 was in a single method, Task 2 was in multiple methods and files and Task 4 was in single method. Upon further investigation, it was found that Task 4 had very high number of lines of code which made it difficult to solve. Similarly, Task 2 was difficult to solve. Also, during the process of beaconization, it was found that Task 3 had most number of people who looked at the same region while for other tasks that number was 4 or more times lower. Due to these factors,

Task 3 had a very discriminative feature set but Task 2 and Task 4 did not had very well defined feature set. This made the learning in the model challenging for Task 2 and Task 4. The complexity level can be determined from the study in [23]. This arises the very need to explore other ANN architectures.

It cannot be ignored that the dimensions of the data set is very high especially for Task 2 and Task 4. To curb this, Principal Component Analysis(PCA)[20] was used but it did not perform well. More dimensionality reduction techniques need to be explored such as t-distributed stochastic neighbor embedding (t-SNE)[44],[33] and Uniform Manifold Approximation and Projection (UMAP)[30]. This is only possible if there are more data samples. Therefore, depending on the complexity and number of data samples, there are possibilities to extend the approach in different directions.

Chapter 7

Conclusions and Future Work

The thesis provides a comparative analysis between representational and non- representational learning techniques to predict the level of expertise of a developer based on their gaze data. The dataset consisted of developers eye gaze while they were solving realistic bug tasks from an open source Java application. There are two significant components in the pipeline: feature engineering and expertise prediction. A novel feature engineering method has been developed that extracts the distinguishing features from the raw gaze data. Engineered features are used to train ML models. Manually engineered features are fed into the model. The results are derived by performing many sets of experiments and show that the ML framework achieves good performance metrics (average accuracy is 93.4% with 78.8% recall and 81.6% F1 score) for all tasks.

The main hindrance to the design process is the scarcity of data. Based on the limited amount of data, an effective solution was designed to automatically learn various mapping functions for predicting the developer's expertise level. In the future, the plan is to automate the feature engineering process and extend the research to incorporate different types of task complexities and different types of tasks such as code summarization, code completion, and code refactoring along with bug fixing tasks. Also, a more extensive controlled experiment will need to be conducted to

generate more data on realistic change tasks. The framework will be extended using the large data set. Different dimensionality reduction techniques will also be used in the future for further comparison.

Since, it is already seen that the tasks can be divided according to complexities and types, in the future, this metric can be used to design a full fledged framework towards automating the developer expertise based on task type and task complexity. Data augmentation can also be performed to generate more data with similar characteristic from existing data.

Bibliography

[1] jabref.sourceforge.net/. accessed: 2020-11-27.

[2] www.tobii.com/. accessed: 2020-11-27.

[3] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, and J. I. Maletic. Developer reading behavior while summarizing java methods: Size and context matters, 2019.

[5] N. Ahmidi, G. D. Hager, L. Ishii, G. Fichtinger, G. L. Gallia, and M. Ishii. Surgical task and skill classification from eye tracking and tool motion in minimally invasive surgery. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 295–302. Springer, 2010.

[6] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin. Do developers read compiler error messages? In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 575–585, 2017.

[7] P. J. Beck, M. Jean Mohammadi-Aragh, C. Archibald, B. A. Jones, and A. Barton. Real-time metacognition feedback for introductory programming using ma-

chine learning. In *IEEE Frontiers in Education Conference (FIE)*, pages 1–5. IEEE Press, 2018.

[8] R. Bednarik, S. Eivazi, and H. Vrzakova. A computational approach for prediction of problem-solving behavior using support vector machines and eye-tracking data. In *Eye Gaze in Intelligent User Interfaces*, 2013.

[9] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5?32, Oct. 2001.

[10] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the linear order. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC 15, pages 255–265. IEEE Press, 2015.

[11] N. Castner, T. C. Kuebler, K. Scheiter, J. Richter, T. Eder, F. Huettig, C. Keutel, and E. Kasneci. Deep semantic gaze embedding and scanpath comparison for expertise classification during opt viewing. In *ACM Symposium on Eye Tracking Research and Applications*, ETRA '20 Full Papers, New York, NY, USA, 2020. Association for Computing Machinery.

[12] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[13] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. pages 58–73, 2002.

[14] A. Duchowski. *Eye Tracking Methodology.* 05 2017.

[15] S. Eivazi and R. Bednarik. Predicting problem-solving behavior and performance levels from visual attention data. In *2nd Workshop on Eye Gaze in Intelligent Human Machine Interaction*, 2011.

[16] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 402–413, New York, NY, USA, 2014. Association for Computing Machinery.

[17] A. Glockner and A.-K. Herbold. An eye-tracking study on information processing in risky decisions: Evidence for compensatory strategies based on automatic processes. *Journal of Behavioral Decision Making*, 24(1):71–98, 2011.

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[19] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif. itrace: Eye tracking infrastructure for development environments. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, ETRA '18, pages 105:1–105:3, New York, NY, USA, 2018. ACM.

[20] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, 1933.

[21] P. Jermann, M. Nüssli, and W. Li. Using dual eye-tracking to unveil coordination and expertise in collaborative tetris. In T. McEwan and L. McKinnon, editors, *Proceedings of the 2010 British Computer Society Conference on Human-Computer Interaction, BCS-HCI 2010, Dundee, United Kingdom, 6-10 September 2010*, pages 36–44. ACM, 2010.

[22] K. Kevic, B. Walters, T. Shaffer, B. Sharif, D. Shepherd, and T. Fritz. Eye gaze and interaction contexts for change tasks – observations and potential. *Journal of Systems and Software*, 128:252 – 266, 2017.

[23] 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*, pages 202–213. ACM, 2015.

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014. 3rd International Conference for Learning Representations, San Diego, 2015.

[25] S. Lallé, C. Conati, and R. Azevedo. Prediction of student achievement goals and emotion valence during interaction with pedagogical agents. In E. André, S. Koenig, M. Dastani, and G. Sukthankar, editors, *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 1222–1231. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM, 2018.

[26] S. Lee, D. Hooshyar, H. Ji, K. Nam, and H. Lim. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing*, pages 1–11, 1 2017.

[27] S. Lee, A. Matteson, D. Hooshyar, S. Kim, J. Jung, G. Nam, and H. Lim. Comparing programming language comprehension between novice and expert programmers using eeg analysis. In *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 350–355, 2016.

[28] Y. Liu, P. Hsueh, J. Lai, M. Sangin, M. Nussli, and P. Dillenbourg. Who is the expert? analyzing gaze data to predict expertise level in collaborative applications. In *Proceedings of the 2009 IEEE International Conference on Multimedia and Expo*, pages 898–901. IEEE Press, 2009.

[29] N. A. Madi, C. S. Peterson, B. Sharif, and J. Maletic. Can the e-z reader model predict eye movements over code? towards a model of eye movements over source code. In *ACM Symposium on Eye Tracking Research and Applications*, ETRA '20 Short Papers, New York, NY, USA, 2020. Association for Computing Machinery.

[30] L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020.

[31] T. Nagamine and N. Mesgarani. Understanding the representation and computation of multilayer perceptrons: A case study in speech recognition. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2564–2573. PMLR, 2017.

[32] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng. A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys (CSUR)*, 51(1):5, 2018.

[33] F. Oliveira, A. Machado, and A. O. Andrade. On the use of t-distributed stochastic neighbor embedding for data visualization and classification of individuals with parkinson's disease. *Computational and mathematical methods in medicine*, 2018:17, 2018.

[34] P. Olsson. Real-time and offline filters for eye tracking, 2007.

[35] J. Quinlan. Decision trees as probabilistic classifiers. In P. Langley, editor, *Proceedings of the Fourth International Workshop on MACHINE LEARNING*, pages 31 – 37. Morgan Kaufmann, 1987.

[36] E. Reichle, K. Rayner, and A. Pollatsek. The e-z reader model of eye-movement control in reading: comparisons to other models. *The Behavioral and brain sciences*, 26 4:445–76; discussion 477–526, 2003.

[37] 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*, ICSE 2014, page 390–401, New York, NY, USA, 2014. Association for Computing Machinery.

[38] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[39] J. A. Saddler, C. S. Peterson, S. Sama, S. Nagaraj, O. Baysal, L. Guerrouj, and B. Sharif. Studying developer reading behavior on stack overflow during api summarization tasks. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 195–205, 2020.

[40] 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*, ESEC/FSE 2015, page 954?957, New York, NY, USA, 2015. Association for Computing Machinery.

[41] Z. Sharafi, B. Sharif, Y.-G. Guéhéneuc, A. Begel, R. Bednarik, and M. Crosby. A practical guide on conducting eye tracking studies in software engineering. In *Empir Software Eng*, page 3128–3174, 2020.

[42] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional

magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 378?389, New York, NY, USA, 2014. Association for Computing Machinery.

[43] B. Steichen, C. Conati, and G. Carenini. Inferring visualization task properties, user performance, and user cognitive abilities from eye gaze data. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 4(2):11, 2014.

[44] L. van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[45] S. Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697 – 709, 1986.