Assessing Software Documents by Comprehension Effort

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Talal El Afchal

under the supervision of
Prof. Michele Lanza
co-supervised by
Prof. Gabriele Bavota, Dr. Andrea Mocci, Dr. Luca Ponzanelli

September 2017

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Talal El Afchal
Lugano, 1 September 2017

Your living is determined not so
much by what life brings to you as
by the attitude you bring to life;
not so much by what happens to
you as by the way your mind
looks at what happens

Gubran Khalil Gubran

Abstract

Recommender systems for software engineering have become increasingly popular in recent years. These systems combine several methodologies to provide suggestions that meet the developer's needs. Recommender systems collect data from online resources, such as blogs, forums, Q&A websites, and suggest documents or pieces of code that are most likely helpful to the developers. However, these systems are not taking into consideration their comprehension effort, which may vary depending on the document familiarity and readability.

In this thesis, we present our approach to calculating the comprehension effort, by creating a language model able to capture a document familiarity, that we combine with the document readability. Usually developers are more interested in documents which they are familiar with. By calculating the comprehension effort, a recommender system can complement the rank and suggest the most comprehensive and appropriate ones to the developer.

Acknowledgements

I would first like to thank my thesis supervisor Prof. Michele Lanza for the opportunity to develop this thesis idea, for providing clear, detailed guidelines, for steering me in the right direction whenever he thought I needed it.

I would also like to thank my thesis co-supervisors Prof. Gabriele Bavota, Dr. Luca Ponzanelli, Dr. Andrea Mocci, for their support and assistence, for the invested time and the extremely fast responses.

Finally, I must express my very profound gratitude to my parents and to my spouse Antonia, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of writing this thesis. This accomplishment would not have been possible without them.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Context

The complexity of software systems is increasing and new technologies are introduced constantly [11]. Software developers often have to work with new technologies they are not familiar with, and as increasingly more comes out, the amount of information that they need to know increases.

For example, Android[1] was introduced 10 years ago in 2007, and in 2008 the first version was released, and nowadays, there are 8 versions of Android. When Android started to become popular, developers had to learn this technology and stay updated with each new release. They had to search how activities work in Android, and how to use several APIs to implement different tasks assigned to them. However, the process of searching the right piece of information as a tutorial or a documentation is time-consuming and requires considerable effort. For example, if an Android developer needs to use a new Android API, she will search web artifacts such as forums, blogs, questions and answers (Q&A) websites, and API documentation [23].

The amount of resources is vast. A popular Q&A website as Stack Overflow contains millions of questions tagged as Android[2]. Github, one of the most popular version control systems where developers can store their projects, hosts more than 500 thousand Android repositories[3].

A tool helping developers to gather information among the available resources would ideally improve the search process, that would result in time saved for the developers. Similar tools to suggest items of interest already ex-

[1] `https://www.android.com`

[2] `https://stackoverflow.com/questions/tagged/android`

[3] `https://github.com/search?utf8=&q=android&type=`

ists outside the context of software engineering, for example Amazon[4], Ebay[5] and Netflix[6], employ recommender systems to suggest their products.

What is a recommender system? The proposed definition by the organizers of the ACM International Conference on Recommender Systems [7] is:

> *"Recommendation systems are software applications that aim to support users in their decision-making while interacting with large information spaces. They recommend items of interest to users based on preferences they have expressed, either explicitly or implicitly. The ever-expanding volume and increasing complexity of information [...] has therefore made such systems essential tools for users in a variety of information seeking [...] activities. Recommendation systems help overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance."*

In the context of software engineering, developers need other items useful to their scope as tutorials, code snippet and libraries. Recommender system targeting similar items are known as recommender systems for software engineering (RSSE). According to Robillard et al. [19] *"An RSSE is a software application that provides information items estimated to be valuable for a software engineering task in a given context"*.

There is a vast number of proposed RSSEs, some of them suggest relevant project artifacts or code examples [7] [27] [28]. Others focus on suggesting relevant code samples, documents and discussions from the web resources [17] [21] [25] [9]. But none of them has considered the required effort to comprehend the suggested documents.

Searching for documentation and tutorials is a crucial step in learning a new technology. Developers can find online resources as blogs, forums, Q&A websites, but the real challenge is to find the most suitable one for their needs. Singer et al. [24] reported in 1997 that the most frequent developer activity was code search, and Sadowski et al. [20] did a case study on how developers at Google search for code. They found that developers are generally seeking answers to questions about how to use an API, what code does, why something

[4] `https://www.amazon.com`
[5] `https://www.ebay.com`
[6] `https://www.netflix.com/ch-en/`
[7] `https://recsys.acm.org/recsys09`

is failing, or where the code is located. The interesting point in this study was the fact that most searches focus on code that is familiar, or somewhat familiar to the developers. Therefore, we believe that RSSEs have to take into consideration the familiarity of a document when they suggest it to the developer.

Understanding a document is a cognitive process, and it depends on human intelligence, yet if we are familiar with a subject, we will likely comprehend it with less effort. For example, a computer engineer might comprehend a document explaining how to implement a sorting algorithm, with much less effort compared to a document that explains a constitutional law, and the reason is not that the algorithm is simple, rather that a computer engineer is more familiar with sorting algorithms than law.

In this thesis, we try to evaluate similar situations. For example, given two documents that have approximately the same subject, how can we decide which one is easier to comprehend? To answer this question we need to introduce two concepts:

- **Familiarity**: how much are developers familiar with the document content?
- **Readability**: how difficult is it to read a given document?

Familiarity and readability metrics are important for us, since the comprehension effort can be derived from the document familiarity and readability. For example, if we have a tool that gives us a score to indicate which document requires less effort to be comprehended, where a higher score indicates a high effort. Given two documents that contain the same Android task but implemented in different versions(7.1 and 4.0). If the developer is more familair with Android 7.1, we expect that the first document must have a lower score since it requires less effort to be comprehended by developers who are more familiar with Android 7.1.

What if both documents have the same task, and both tasks are implemented with Android 7.1 ? Which one will have a lower score? In this case, the document readability has a big impact on the comprehension effort: The developer would likely prefer to read the document with the best readability.

1.2 Objective and Results

Our goal is to assess documents by their comprehension effort which can be leveraged by RSSE to improve their suggestions. To calculate the comprehen-

sion effort we need to compute the familiarity, and we need to evaluate our approach to understand if it effectively works. To evaluate our approach we ran two studies:

In the first study we evaluate our familiarity approach. We select a large set of Android documents, which represents the hypothetical developer knowledge, where a document can be a mix of code and natural language. Then we create a *Language Model* [6] and we train it with these documents (training set). In this way, we are able to simulate the context of a developer who is familiar with Android. Once we trained the language model, we evaluate the familiarity of a given set of documents (testing set) that contains Android documents and other documents that are not related to Android.

The language model was able to evaluate the Android documents within the testing set as the most familiar. In this experiment, the language model approach satisfied our expectation in capturing the document familiarity.

In the second study we evaluate our approach in assessing documents by the comprehension effort. We asked developers to read a set of tutorials, and then we gave them a set of documents, where some documents are related to the tutorials and some are not, and we asked the developers to score the documents by the comprehension effort. and we compared their scores with our precomputed scores. The evaluation results suggest that our approach is promising in estimating the comprehension effort of developers.

1.3 Structure of the Thesis

This thesis consists of seven chapters:

1. **Introduction**: In this chapter we introduce the thesis work.
2. **State of the Art**: This chapter describes the existing related work as code search engines, and recommender systems.
3. **Approach**: This chapter describes our approach to calculate the comprehension effort.
4. **Study design**: This chapter discusses the research question, the data extraction process, the analysis method, and the replication package.
5. **Result**: This chapter shows the results and their implications.

6. **Threat to Validity**: This chapter describes the possible threats that could affect the results validity.

7. **Conclusion**: This chapter summarizes our work, presents ideas for possible future work, and concludes the thesis.

Chapter 2

State of the Art

We present the existing related work concerning the comprehension effort, and discuss related tools, such as code search engines and recommender systems.

2.1 Program Comprehension

There are several studies on program comprehension that are related to the thesis work.

Corbi [4] did a research on tools which could help developers in two key areas: static analysis (reading the code) and dynamic analysis (running the code). Corbi describes how program understanding relates to software renewal, and he indicates that more than half of the time is spent in understanding a system. Corbi concludes by mentioning that developer training and tools should assist the developer in combining different kinds of information in ways which can support the understanding of the system being investigated, and they should not favor or force the use of only one way of gathering information about programs. Corbis' work motivated us to investigate and develop a novel technique to calculate the comprehension effort.

Another related work is by Kushwaha and Misra [10] who claim that the required effort to understand a software depends on the difficulty in understanding the information, where the information is related to the number of operators and identifiers. They count the number of operators and identifiers per line of code and they multiply it by an associated weight of the identifier name (1 if the identifier name belongs to the problem domain, and 4 if the identifier name is selected arbitrarily).

They performed an experiment with 60 students, where 5 sample programs were given. One set of programs had meaningful identifier names related to the problem domain and the other used arbitrarily selected identifier name. They measured the required time to comprehend the program.

The result showed that programs with arbitrarily selected identifier names required about 4 times the time needed to comprehend the programs compared to programs with meaningful identifiers.

Scalabrino et al. [22] presented a metric able to assess the understandability of a given code snippet . In their work they consider three types of metrics:

1. **Code-related metrics** related to the code, (e.g., cyclomatic complexity, LOC, the number of identifiers, line length).

2. **Documentation-related metrics** to capture the quality of the internal documentation of a snippet (e.g, comments readability, and identifiers consistency).

3. **Developer-related metrics** to measure the programming experience of the developer in years, in any programming language.

They analyze whether code-related, documentation-related, developer-related metrics can be used to assess the understandability level of a snippet code. The authors asked 46 developers to read and to fully understand eight code snippets. Participants could, select the option *I understood the snippet* or *I cannot understand the snippet*, and the time was monitored. Once the participants chose *I understood the snippet* option, the authors asked questions about the code snippet to verify the actual level of understanding.

After an extensive statistical analysis Scalabrino et al. [22] could not find a significant correlation between the considered metrics and the understandability of code snippets. They assumed that the code complexity has a big influence on the developer' ability to understand the code, but they could not demonstrate it with a empirical evidence. They also mentioned that the code readability can have a direct impact on the understandability of the code.

As mentioned in the previous chapter, in this thesis we take in consideration the code readability to calculate the comprehension effort.

Buse and Weimer [3] introduced a code readability metric, and they investigated its relation to software quality. A part of their work was to run an experiment which compared readability of the code to the cyclomatic complexity,

and they were able to show that code readability is significantly independent of the traditional code complexity.

In this experiment, 120 developers were asked to individually score a sequence of 100 code snippets, based on their personal estimation of readability. From the result, they determined which code features were predictive of readability, and they constructed a readability model. They also tested the model performance on ten different classifiers, and on average the model classified correctly between 75% and 80% of the snippets. They found that factors like *average line length* and, *average number of identifiers per line* are very important to readability. In this thesis we use Buse and Weimer approach [3] to calculate code readability.

2.2 Semantic code search and code search engines

Reiss [18] presented a tool that generates specific functions or classes from the open source repositories, where these classes meet user's specifications. This tool uses the user's input, as keywords and other constraints, and suggests codes that meet the user's needs.

The tool takes a set of candidate solutions, and it transforms them into a more appropriate set. Both static and dynamic specifications can be used. The main goal of this tool is to satisfy the user's constraints, but it does not take into consideration the user's experience and the comprehension effort.

Thummalapenta and Xie [26] presented *PARSEWeb*, a tool similar to the one by Reiss [18], where they collect code from public sources and search engines to suggest it to developers based on their input query. In the query, the developers have to specify the *source object type* and the *destination object type*. For example, programmers know what type of object that they need to instantiate like *QueueConnectionFactory* (Source), but do not know how to write code to get that object from a known object type like *QueueSender* (Destination). Therefore, the proposed problem can be translated to a query of the form *QueueConnectionFactory →QueueSender*.

McMillan [12] created an application search system called *Exemplar*, which reduces the mismatch between the high-level intent reflected in the descriptions of software and low-level implementation details.

Exemplar differs from traditional search engines that match the keywords, by matching keywords with the descriptions of the various API calls in help documents. *Exemplar* has three components of Ranking:

1. **WOS** a component that computes a score based on word occurrences in project descriptions
2. **RAS** a component that computes a score based on the relevant API calls
3. **DCS** a score based on data-flow connections between calls

McMillan [12] concludes by mentioning that the performance of search engines can be improved if those engines consider the API calls that the software uses.

Bajracharya and Lopes [1] conducted an exploratory analysis of the usage log of Koders, the first commercially available Internet-Scale code search engine, and their goal is to answer the following three questions:

1. **Usage**: What kind of usage behavior can we see in Koders?
2. **Search Topics**: What are the users searching for?
3. **Query Forms**: How are users expressing their information need in their queries?

They analyzed the usage logs, finfing that:

- Most of the users did not use Koders again after using it for a day
- Sessions are short (a series of activities by a single user within a small duration of time constitutes a session)
- More than half of the sessions had no downloads
- There are sessions with no search activities
- Queries are very short
- Terms in queries are quite diverse
- Code queries are the mostl used types of queries
- Code queries lead to the most of the downloads

They conclude the analysis by mentioning that usage behavior is similar between Koders and search on the Web, and the majority of the users do not refine their existing queries.

The approaches described so far do not take into consideration any readability or familiarity metrics. We believe that our comprehension effort metric can be integrated in these search engines as complementary information to help the user in choosing the most relevant document.

2.3 Recommender Systems for Software Engineering

Čubranić and Murphy [27] built *Hipikat*, a tool that helps newcomers by recommending existing artifacts from the development that are relevant to a task that the newcomer is trying to perform. *Hipikat* infers links between the artifacts that may have been apparent at one time to members of the development team but that were not recorded. *Hipikat* uses those links to suggest possibly relevant parts of all artifacts that have been produced given information about a task a newcomer is trying to perform.

The authors performed two qualitative studies, where they showed that *Hipikat* helped newcomers to perform a task effectively on an unfamiliar system.

Holmes and Murphy [7] presented *Strathcona*, a plug-in for the Eclipse integrated development environment (IDE) that extracts the structural context of the code on which a developer is working, and selects from an example repositories a set of relevant code examples to be returned using a set of structural matching heuristics.

To understand if the structural matching heuristics can return examples that a developer finds useful, Holmes and Murphy [7] performed a case study in which they asked two developers to complete four programming tasks, where Subject 1 had less than one month of Eclipse plug-in programming experience but more than eight years of Java experience. Subject 2 had over six months of Eclipse plug-in programming experience but only eighteen months of experience with Java, and neither subject knew how to implement any of the assigned tasks. Subject 1 completed all four tasks successfully, finding relevant examples in all cases for which appropriate examples were returned; Subject 2 completed three out of four tasks, finding relevant examples in two of the three cases. These results show that *Strathcona* can suggest relevant examples to developers.

Ponzanelli et al. [14] presented *Prompter*, a plug-in for the Eclipse IDE that retrieves and recommends, with push notifications, relevant Stack Overflow discussions to the developer. *Prompter* analyzes the code context in the IDE

and searches for Stack Overflow discussions, and evaluates their relevance by taking into consideration *code aspects*, *conceptual aspects*, *community aspects*.

The authors performed a study to evaluate to what extent the use of *Prompter* can be useful to developers during a development or maintenance task. In the study they selected 12 participants that have at least 3 years of experience in programming, with a maximum of 12. The authors asked the participants to perform one maintenance task and one development task, and if they used the suggestions by *Prompter*. Three participants answered *absolutely yes*, eight *more yes than no*, and two *more no than yes*.

The authors concluded their work by mentioning that *Prompter* ranking model resulted to be effective in identifying the right discussions given a code snippet to analyze.

In a following work Ponzanelli et al. [16] created *Libra*, a holistic recommender system that supports developers in their information search in the web browser. The developers are tracked during their activities in the IDE and in the web browser, which allows *Libra* to collect information as web search results, perused pages, and code written and modified by the developer. *Libra* uses this information to model a knowledge context of the developer and, constructs a holistic meta-information model of their contents.

Libra is based on three important metrics :

- **Context Complementarity** measures the information intake provided by a resource in the current context of the developer. A low context complementarity indicates that the resource information is already a part of the context. In other words, the resource and the context are too similar, and the amount of new information that the developer can retrieve is low.

- **Result Prominence** identifies prominent results among the search engine result set. When a query matches a result, it does not tells much about the result relevance. If a result overlaps with many other results, it would be more prominent, since probably it provides diversified information in its contents.

- **Information Quantity** sums up the number of "elements" identified by *Libra*'s meta-information system. This metric is important to distinguish which resource has more information giving two resources with a similar number of characters. If the first resource has only text and the second one has text and code, obviously the second resource contains a higher information quantity.

16 third year CS Bachelor students were asked to evaluate *Libra* in terms of *its ability in correctly assessing for each query search result its prominence and complementarity with respect to the context, and usefulness to developers during a development or maintenance task.*

The study results indicated that both prominence and complementarity indicators reflect developers perception of such measures, and are considered as useful indicators. Moreover, students achieved a significantly better task completeness with *Libra*.

The work that has been done in this thesis can be integrated with RSSE as *Libra*, where the comprehension effort metric can be added to the other three metrics used by *Libra* to provide developers with the most relevant documents.

Chapter 3

Approach

The volume of online resources is huge as we can find millions of documents related to one search query. How can we suggest the most suitable one to developers? Several elements might determine which documents contain valuable information, and one of these elements is the comprehension effort. Code search has been a part of software development for decades, Singer et al. [24] reported in 1997 that the most frequent developer activity was code search, and most searches focus on code that is familiar to the developer [20]. Certainly, developers prefer to search for documents they are familiar with since they require less effort to be comprehended. We believe that the comprehension effort is strictly related to the document familiarity and readability.

In the literature there are several metrics that estimate the readability of a given text, (e.g., Flesch—Kincaid [8], Dale—Chall [5]), and other metrics that estimate code readability [2]. In this thesis, we introduce for the first time a new approach where we combine text readability, code readability and document familiarity, to estimate comprehension effort.

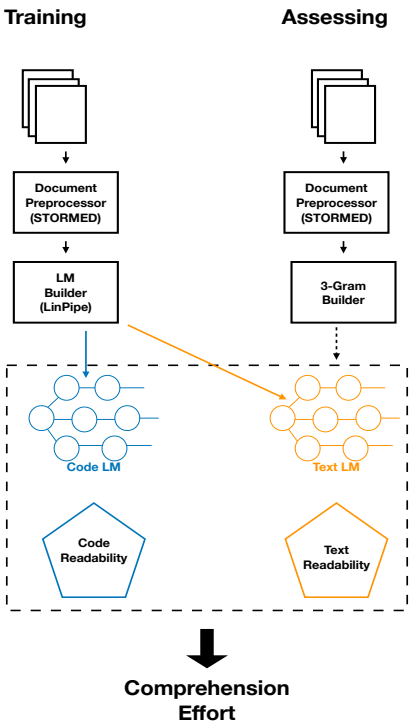

Figure 3.1. Overall architecture of our approach

3.1 Overview

Fig 3.1 shows the overall architecture of our approach. As we can see, there are two *language models*, one for code and one for text (natural language), and the whole process is divided in two phases that work in parallel:

- **Training phase**: In this phase we train the language models on a given set of documents, where each document is broken down in two segments: code and text. To extract code and natural language from a document, we use the Stormed Island Parser[1] [15]. The *code language model* will be trained with the code segment, and the *text language model* will be trained with the text segment. These language models will give us the familiarity of a given document in the assessing phase.

- **Assessing phase**: In this phase we give each document a *Comprehension Effort* score. As a first step we use Stormed [15] to brake down each doc-

[1] `https://stormed.inf.usi.ch`

ument in two segments: code and text, then we use the language models that we trained in the training phase to estimate the familiarity. In the same time we use Flesch—Kincaid[8] to calculate the text readability, and RayKernel [2] to calculate the code readability. In the last step we combine the familiarity and the readability to get the comprehension effort.

In the next sections we explain the training and assessing phase, and we give more details about the component that are used in the whole process.

3.2 Training Phase

In the training phase we use Stormed to distinguish between code and text, and we train two language models, one for text and one for code. We explain what Stormed is and how we use it, and then we explain what a language model is, and how we train it.

3.2.1 Stormed Island Parser

Stormed is a dataset and parser for Stack Overflow that models the posts by building a heterogeneous abstract syntax tree (H-AST) for each discussion in the data dump [15].

Stormed Parser

Stormed can parse any given data in form of text. For example, given the following text :

```
With a sorted array, the condition data[c] >= 128
```

Stormed can identify *"With a sorted array, the condition"* as text, and "$data[c] >= 128$" as code.

Stormed gives us more detailed information about the code, that can be one of the following possible H-AST nodes:

- **JavaASTNode**: Java code including incomplete fragments.
- **StackTracesASTNode**: Stack traces including incomplete stack trace lines.
- **XMLASTNode**: XML/HTML documents, tags and elements.
- **JSONASTNode**: JSON fragments.

Stormed allows us to distinguish between the textual part (natural language), and the code part. Moreover, Stormed parses the code part and identifies the Java code even if the code is incomplete.

Stormed Dataset

As shown in Fig 3.2 each Stack Overflow document is represented with HTML tag, Stormed extracts two types of information units:

- **Natural Language Tagged Unit** is the textual part of a discussion. Text units are all fragments that are not tagged as <code> or <pre><code>.
- **Code Tagged Unit** is the code part of a discussion. Structured Fragment Unit are every contents tagged as <code> or <pre><code>.

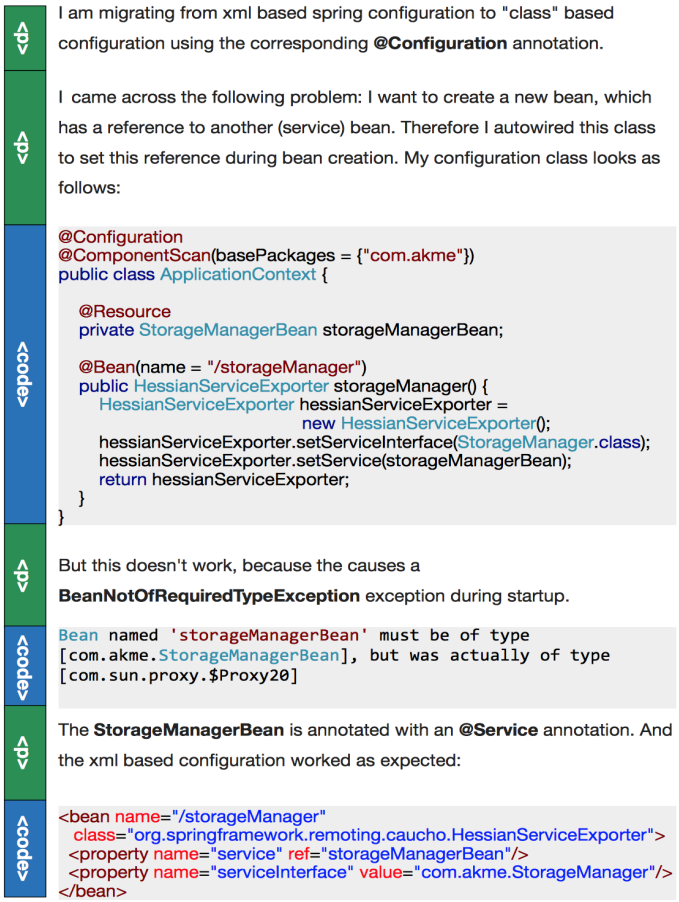

Figure 3.2. Example of Stack Overflow question with HTML tagging

The content of the extracted units are modeled as a H-AST, and added to the data set. The Stormed data contains a set of JSON files, one for each Stack Overflow discussion. The JSON files can be parsed to obtain objects corresponding to the H-AST.

3.2.2 Language Model

A **Probabilistic Language Model (LM)** is a probability distribution over sequences of words. Given such a sequence of length m, it assigns a probability $P(w_1, \ldots, w_m)$ to the whole sequence[2].

The goal of the language model is to compute the probability of a sentence or sequence of words.

$$P(W) = P(w_1, w_2, w_3, w_4, w_5 \ldots w_n)$$

A LM can also compute the probability of an upcoming word.

$$P(w_5|w_1, w_2, w_3, w_4)$$

A LM applies Markov Chain Assumption to compute $P(W)$

$$P(w_1 w_2 \ldots w_n) \approx \prod_i P(w_i|w_{i-k} \ldots w_{i-1})$$

Each component in the product is approximated

$$P(w_i|w_1 w_2 \ldots w_{i-1}) \approx P(w_i|wi-k \ldots w_{i-1})$$

Bigram model provides the conditional probability of a word given the previous word.

$$P(w_i|w_1 w_2 \ldots w_{i-1}) \approx P(w_i|w_{i-1})$$

The bigram model can be extended to trigrams, 4-grams, 5-grams, ..., n-grams. $P(w_i|w_{i-1}$ is the maximum likelihood estimation in a bigram where

$$P(w_i|w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

[2] https://en.wikipedia.org/wiki/Language_model

Example

Lets consider the following text:

```
Hi I am John
Hi John I am David
Hi I am looking for John
```

$$P(I|HI) = \frac{2}{3} = 0.67$$

The probabilities of the sentence "I like Italian food" estimated by a bigram model is:

$$P(like|I) \times P(Italian|like) \times P(food|Italian) = 0.00042$$

To avoid underflow and to make multiplication faster, we use log space.

$$log(p_1 \times p_2 \times p_3 \times p_4) = log(p_1) + log(p_2) + log(p_3) + log(p_4)$$

3.2.3 Training the Language Model

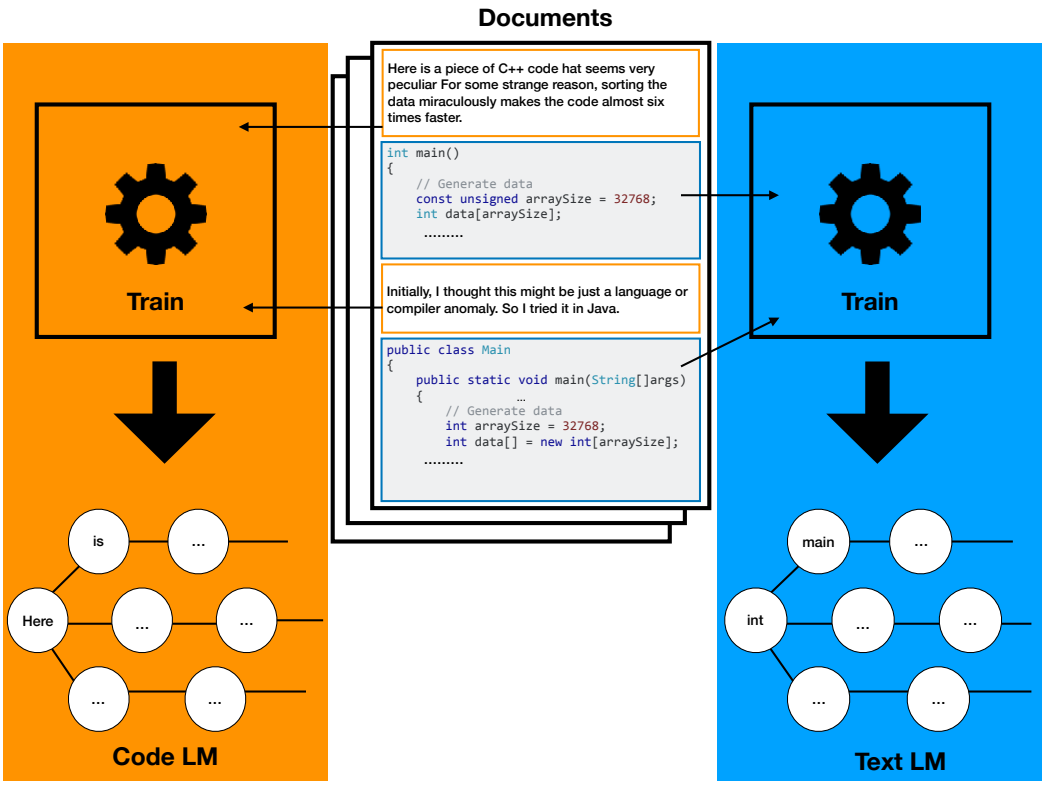

Figure 3.3. Training Language Model overview

In our approach we use the 3-Gram model, we tried the 4-Gram and the 5-Gram but there was no significant difference. As we said before, we differentiate between natural language and code, therefore, we create a language model for each as we can see in Fig 3.3.

- **Natural Language LM training**: Once we have identified the textual part of a document, we remove the stop words, since they are common words and they add noise to the LM, then we train the *Natural Language LM* with the filtered text.
- **Code LM training**: For the code part we use a similar approach. We use the ANTLR[3] parser to parse the code and to create tokens that we model as 3-Grams, and we pass them to the LM. But before training the LM on the code we remove all separators (see Fig 3.4).

  The reason why we remove separators is because they add noise. Similar to text punctuation, they do not tell much about the information that a document contains. For example, given a Java class code:

```
public class Example {
  void multiply(int a, int b) {
    if(a == 0 || b == 0){
    System.out.println(0);
    }
  else{
    int result = a * b;
    System.out.println(result);
    }
  }
}
```

  We can see that the code ends with three curly brackets **}}}**, and almost all Java calsses end with at least 2 curly brackets. If we do not remove the separators from code, the three curly brackets will form the most popular 3-Gram.

  Before removing the separators, we had several situations where the most familiar 3-Grams is composed by 2 brackets or 2 parentheses, for this reason we decide to remove this noise by filtering all the separators.

[3] http://www.antlr.org

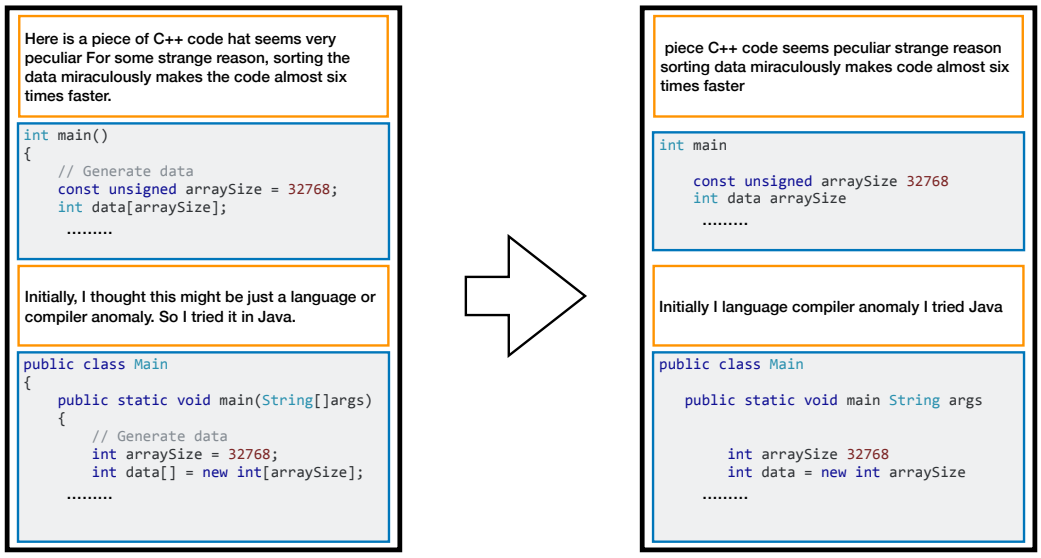

Figure 3.4. Filtering stop words and separators

3.3 Assessing Phase

In the assessing phase we estimate the familiarity of a given document, then we calculate the document readability and we combine them in a comprehension effort score.

3.3.1 Familiarity Estimation

We define the familiarity as the estimation of likelihood of a specified character sequence. But before estimating the familiarity, we need to do the following preprocessing steps:

- **Selecting & Filtering**: As we did in the training phase, we use the Stormed Island Parser[4] [15] to break down a document into text and code. We remove the stop words form the text, we parse the code[5] and we remove the separators, as shown in Fig 3.4

- **3-Grams generating** : In Fig 3.5 we can see that each chunk of code or text is transformed to a 3-Gram model, and then the language model evaluates the familiarity of each N-Gram.

[4] https://stormed.inf.usi.ch
[5] http://www.antlr.org

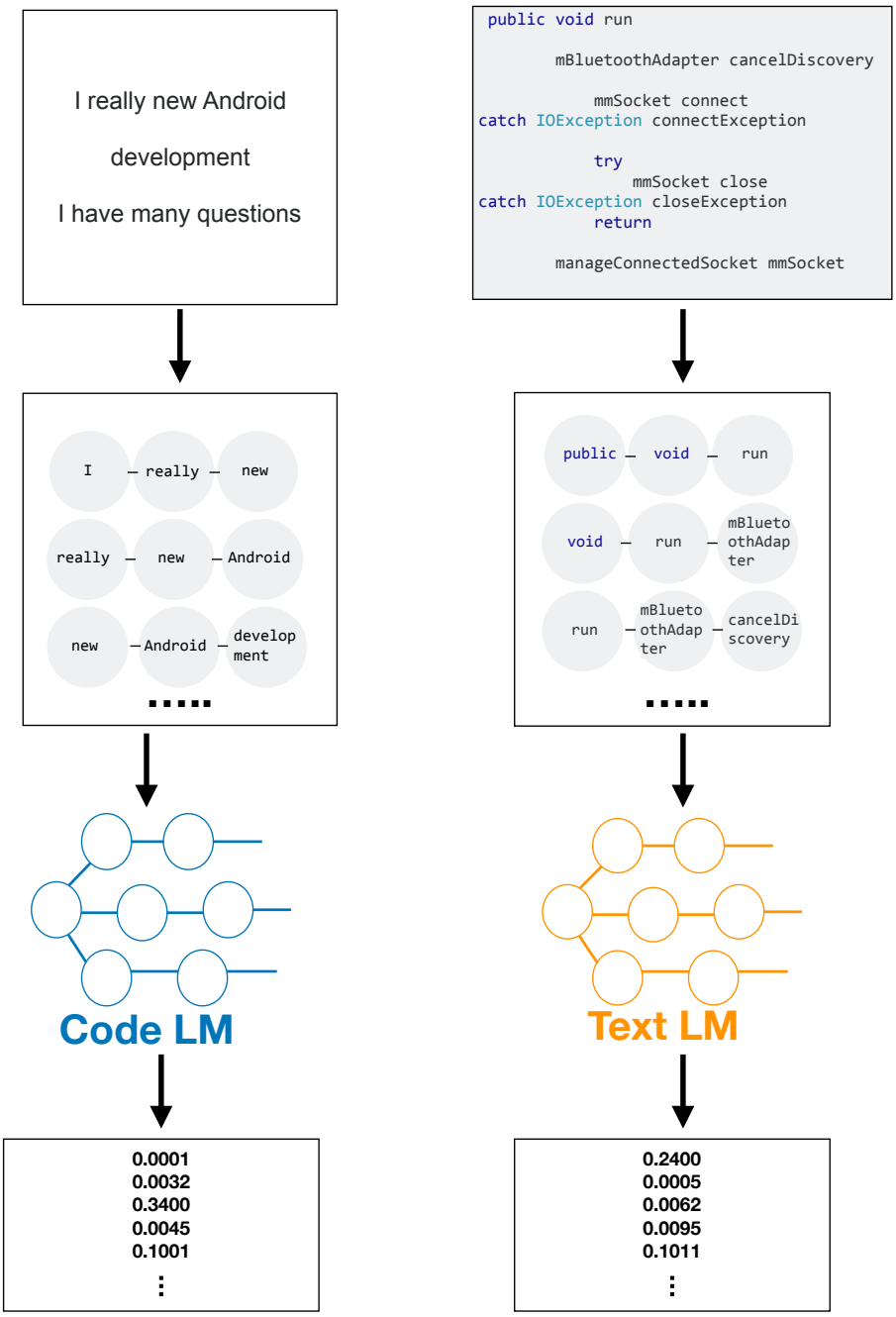

Figure 3.5. 3-Gram evaluating process

- **Familiarity aggregation**: Now that we have retrieved the familiarity of each 3-Gram of a given document, we need to aggregate them in one value. The trivial way is to multiply all the probabilities to get one value, but this approach is biased by the document length.

  As we can see in Fig 3.6 we have 2 documents, a and b, where *document b* contains twice the content of *document a*. We expect that the familiarity in both documents must be the same since they have the same text, even though in *document b* the text is duplicated.

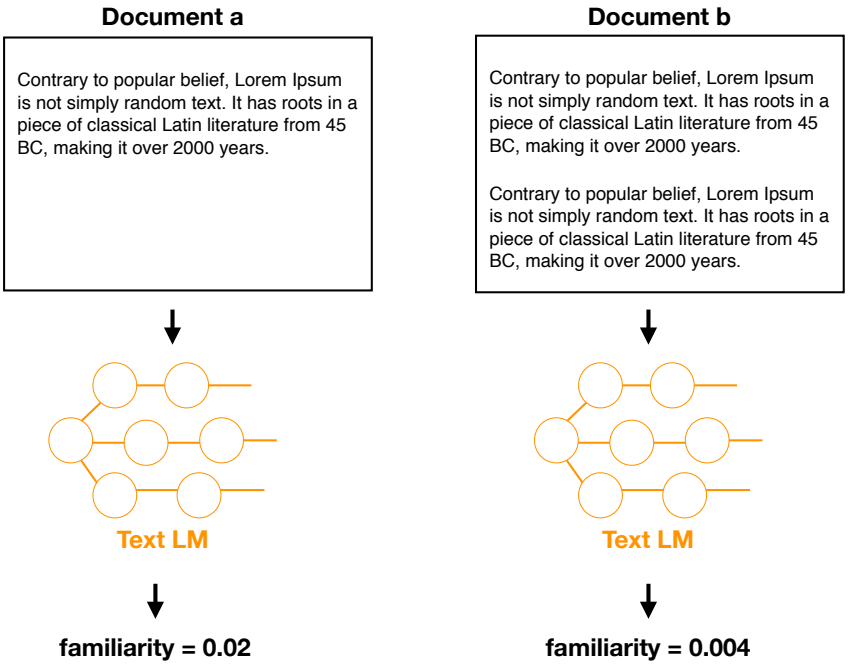

Figure 3.6. Aggregating probabilities by multiplication

Given a set of documents with different lengths, the shortest documents turns out to be the most familiar. This approach is valid if all documents have the same length, which is not the case with the documents that we are dealing with, since they have heterogeneous length.

We tackle this problem by introducing a new approach to aggregate the probabilities as shown in Fig 3.7. The approach consists in 5 steps:

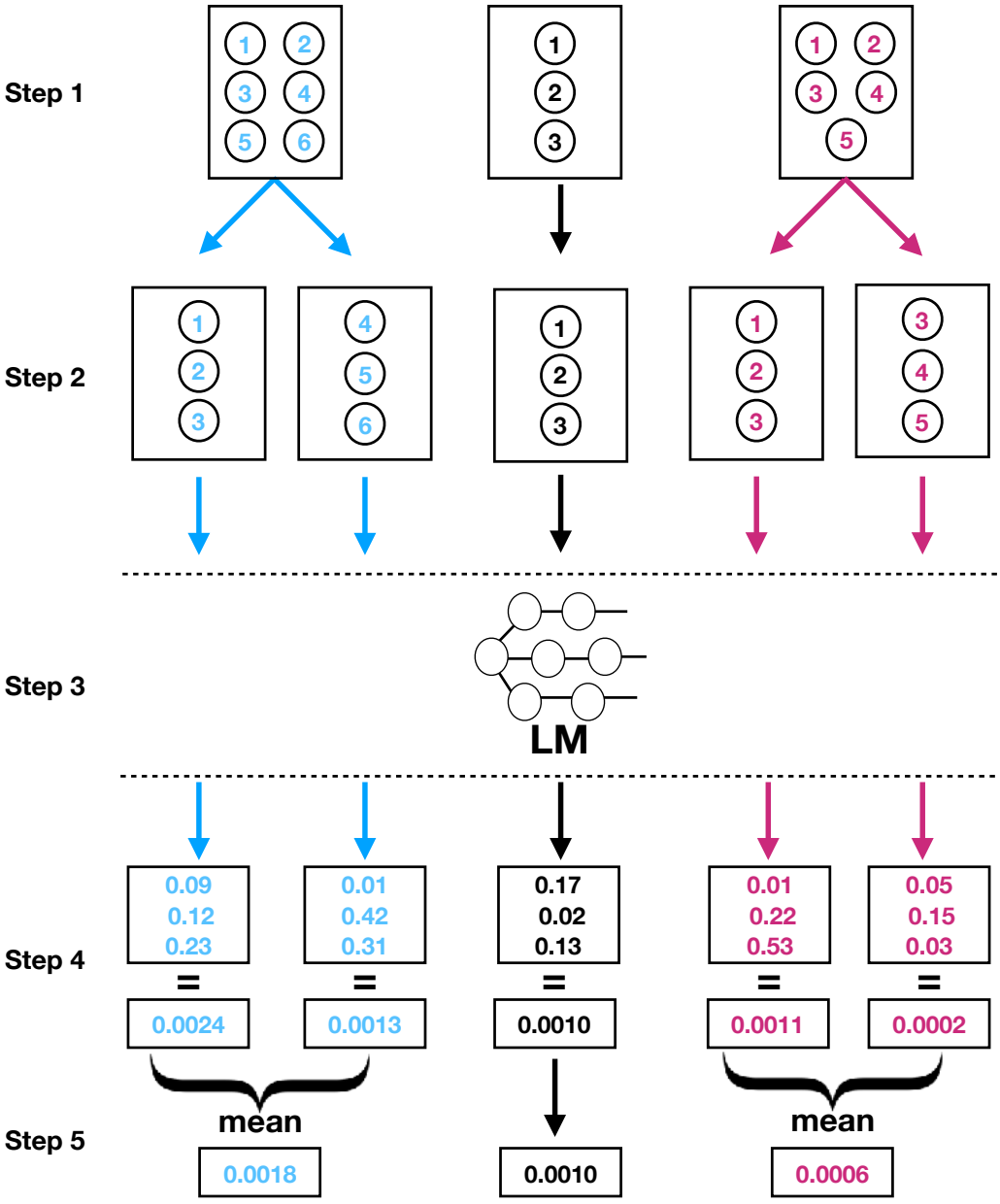

Figure 3.7. Aggregating probabilities by mean

- **step 1**: We select from the set the document with the smallest number of 3-Gram, say of length m.
- **step 2**: We split each document into chunks of length m.
- **step 3**: For each chunk we calculate the familiarity of its 3-Grams
- **step 4**: Now that all chunks have the same size, we calculate the familiarity of a chunk by multiplying the probability of all its 3-Grams
- **step 5**: Is the last step, we calculate the familiarity mean of each document chunks.

3.3.2 Readability

The document readability has a big impact on the comprehension effort. Two text documents can contain approximately the same information, but one text can be easy to read and the other one can be difficult to read, and it is the same for code. The following piece of code can be written in one line :

```
{"name":"mkyong.com","messages":["msg
   1","msg 2"],"age":100}
```

But most likely developers will find it easier to read the indented code:

```
{
        "name":"mkyong.com",
        "messages":["msg 1","msg 2"],
        "age":100
}
```

As mentioned so far we distinguish between *code readability* and *text readability*. Therefore, we use different tools to calculate the readability of each.

3.3.3 Accounting for Text Readability

The text readability score is calculated by the **Flesch-Kincaid** [6] formula:

$$0.39\left(\frac{\text{total words}}{\text{total sentences}}\right)+11.8\left(\frac{\text{total syllables}}{\text{total words}}\right)-15.59$$

[6] https://en.wikipedia.org/wiki/FleschKincaid_readability_tests

A higher scores indicate that the document is easier to read, and a lower number indicates that the document is more difficult to read.

Flesch-Kincaid score is scaled in 0-100 rage, but for convenience we normalize it to 0-1 range. The score can be interpreted as shown in the table below, Table 3.1.

Score	**School Level**	**Notes**
100.0 – 90.0	5th grade	Very easy to read. Easily understood by an average 11-year-old student.
90.0 – 80.0	6th grade	Easy to read. Conversational English for consumers.
80.0 – 70.0	7th grade	Fairly easy to read.
70.0 – 60.0	8th & 9th grade	Plain English. Easily understood by 13- to 15-year-old students.
60.0 – 50.0	10th to 12th grade	Fairly difficult to read.
50.0 – 30.0	College	Difficult to read.
30.0 – 0.0	College Graduate	Very difficult to read. Best understood by university graduates

Table 3.1. Flesch-Kincaid score grade

3.3.4 Accounting for Code Readability

To calculate the code readability we use the Buse and Weimer [2] code readability metric and tool[7]. Buse and Weimer determined a set of code features that are predictive of readability. The features are listed in the Table 3.2.

Buse and Weimer's readability score is scaled in 0-1 range, where a higher scores indicate that the code is easier to read and a lower number indicates that the code is more difficult to read.

[7] http://www.arrestedcomputing.com/readability

Feature Name
line length (# characters)
identifiers
identifier length
indentation (preceding whitespace)
#keywords
#numbers
#comments
#periods
#commas
#spaces
#parentheses
#arithmetic operators
#comparison operators
#assignments (=)
#branches (if)
#loops (for, while)
#blank lines
#occurrences of any single character
#occurrences of any single identifier

Table 3.2. Buse and Weimer code features

3.4 Accounting Comprehension Effort

To calculate the *Comprehension Effort* of a given document we calculate:

- The familiarity of the textual parts f_t
- The familiarity of the code parts f_c
- The readability of the textual parts r_t
- The readability of the code parts r_c

The r_t and r_c are on a scale of 0 to 1, the f_t and f_c are unbounded log_2 probabilities.
We propose two formulas to calculate the *Comprehension Effort*:

$$\text{Comprehension Effort} = \frac{(r_c \times f_c) + (r_t \times f_t)}{2}$$

In the above formula the *readability* and the *familiarity* have the same weight in calculating the *comprehension effort*.

$$\text{Comprehension Effort} = \frac{(r_c \times f_c) + (r_t \times f_t)}{r_c + r_t}$$

In the above formula the *familiarity* has a higher weight. In this thesis we explore both formulas to understand which one performs better.

Chapter 4

Study Design

In this chapter we present two studies aimed of assessing the LM capability to evaluate the familiarity of a given document, and to evaluate the effectiveness of our approach in assessing documents by the comprehension effort. Towards this goal, we run two tailored studies:

The first study evaluates the familiarity approach. In this study (Experiment-1) we select a set of Android documents representing the developer knowledge context, and we train the LMs with these documents (Chapter 3.2). Once we trained the language model, we evaluate the familiarity of a given set of documents about Android and other documents that are not related to Android (Chapter 3.3.1).

The second study evaluates our approach in assessing documents by comprehension effort. In this study (Experiment-2) we ask developers to read a set of tutorials, and then we give them a set of documents, where some documents are related to the tutorials and some are not, and we ask them to score the documents comprehension effort on a scale from 1 to 5. Then we compare their score with our precomputed comprehension effort score (Chapter 3.4).

4.1 Research Questions

To evaluate the effectiveness of our approach we have formulated the following research questions:

RQ1: *Can a language model correctly evaluate the familiarity of a given document?*

We have answered this research question by downloading 200,000 Stack Overflow questions tagged as Android. From these 200,000 documents we select 5 training sets with different size (10, 100, 1000, 10000, 100000), and we select 1000 documents for testing.

We extend the testing set by downloading:

- 1000 Stack OVerflow questions tagged as JavaScript.
- 1000 Stack OVerflow questions tagged as Swift.
- 1000 Stack OVerflow questions tagged as Java.

We create a code LM and a text LM, and we train them with the Android 10 documents training set, then we calculate the familiarity of the testing sets (Android, JavaScript, Swift, Java). We repeat this experiment with 100, 1000, 10000, 100000 android documents to understand how the training set size can influence the performance. Then we plot the familiarity results to check if the LMs classify Android documents as the most familiar.

RQ2: *What is the accuracy of our technique in assessing documents by the comprehension effort?*

To answer this research question we asked two developers with no experience with Android to read a set of introduction tutorials to Android, then we asked them to read two Android tutorials. The first one explains how to use the *Android-Bluetooth* API and the second one explains how to use the *Android-Camera* API. Once they finished reading the tutorials, we asked them to score the comprehension effort of 6 documents from Stack Overflow. The comprehension effort scores are on a scale from 0 to 5, where a higher score indicates a higher effort in comprehending the documents. The developers scored the following 6 documents:

- Android-Bluetooth discussion[1].
- Android-Camera discussion.[2]
- Android-AsyncTask discussions (not related to *Bluetooth* or *Camera*) [3].

[1] `https://stackoverflow.com/questions/9693755`
[2] `https://stackoverflow.com/questions/5991319`
[3] `https://stackoverflow.com/questions/31874990`

- Android-Database discussions (not related to *Bluetooth* or *Camera*)[4].
- Two Cordova discussions. (Cordova also known as PhoneGap is a mobile application development framework that enables software developers to build applications for mobile devices using CSS3, HTML5, and JavaScript)[5] [6].

We trained the LMs with the tutorials that we gave to the developers. Since the tutorials are not enough to train a LM, we augmented the set by adding several tutorials that have the same subject as the one that we gave to the developers. We also trained the code LM with as set of *Android-Bluetooth* and *Android-Camera* code snippets from *Gist*[7].

After the training phase we calculated the comprehension effort of the 6 Stack Overflow documents and we compare our results with the developers scores.

4.2 Data Collection

To run the two experiments we collect and extract data from different sources. In the next sections we explain how we collect data for each experiment.

4.2.1 Experiment-1

The Stormed dataset provides a set of JSON files, one for each discussion. We use the Stormed development kit to parse the JSON files and obtain the H-AST. As we mentioned in the previous section, in Experiment-1 for the training set we select a set of 100,000 Android discussions, and for the testing set we select a set of 1000 Android discussions. These sets are retrieved from the Stormed dataset, where we select discussions that are tagged only as Android, and the training and testing sets are disjoint.

We also use Stack Overflow dump[8] to select the JavaScript, Swift and Java discussions that we add to the testing set. We selected Java discussions that are not also tagged as Android to avoid having Android discussions in the Java set, since Android is implemented in Java.

[4] https://stackoverflow.com/questions/17451931
[5] https://stackoverflow.com/questions/20835768
[6] https://stackoverflow.com/questions/10023328
[7] https://gist.github.com/
[8] https://archive.org/details/stackexchange

4.2.2 Experiment-2

In Experiment-2 we have a set of 4 tutorials that introduce Android to developers, and give them an overview of how Android works. These tutorials are selected from the *tutorialspoint* website[9].

To augment the training set we select 4 Android introduction tutorials, 7 *Android-Bluetooth*, 6 *Android-Camera* tutorials from several websites. The tutorials are parsed by Stormed and passed to the LMs.

Besides the Android tutorials, we augment the training set by selecting from Gist 525 *Android-Bluetooth*, 2666 *Android-Camera* code snippets.

4.3 Replication Package

The Stormed data set that contains the Stack Overflow dump is available on Stormed website : `https://stormed.inf.usi.ch`.

The Thesis work is publicly accessible on this link : `https://github.com/talalelafchal/familiarity/tree/Aggregation`. The repository contains:

- Testing and training sets used in Experiment-1
- Tutorials, Stack Overflow discussions and Gist files used in Experiment-2
- Experiment-1 results in *.csv* files with the R script to plot the results.
- Experiment-2 results.
- The source code.

[9] `https://www.tutorialspoint.com/android/`

Chapter 5

Results

In this Chapter we present and discuss the evaluation results of our familiarity and comprehension effort estimating approach.

5.1 Familiarity Estimation Results

To evaluate our approach in estimating familiarity, we train two LMs (one for code and one for text) on a set of Android discussions from Stack Overflows' dump (training set). Then we evaluate the familiarity of 1000 Stack Overflow discussions on each of the following subjects: Android, Java, Swing and JavaScript.

We select 5 training sets with different size (10, 1000, 1000, 10000, 100000) to understand how the training set size can influence the performance. In the following sections we discuss the code LM results and the text LM results.

5.1.1 Code LM Results

As a first step in the experiment, we train the code LM with 10 Android documents and we evaluate the code familiarity of the testing set. In Fig 5.1, we show the box plot result. The y axis represents the *Log* estimated familiarity, where a higher box position indicates a lower familiarity. In this plot, we can see that JavaScript and Swing documents are estimated as more familiar than Android documents (JavaScript F_c mean $= -169$, Swing F_c mean $= -178$, Android F_c mean $= -212$). The reason why Android is not estimated as the most familiar is because 10 documents are too few to train a LM.

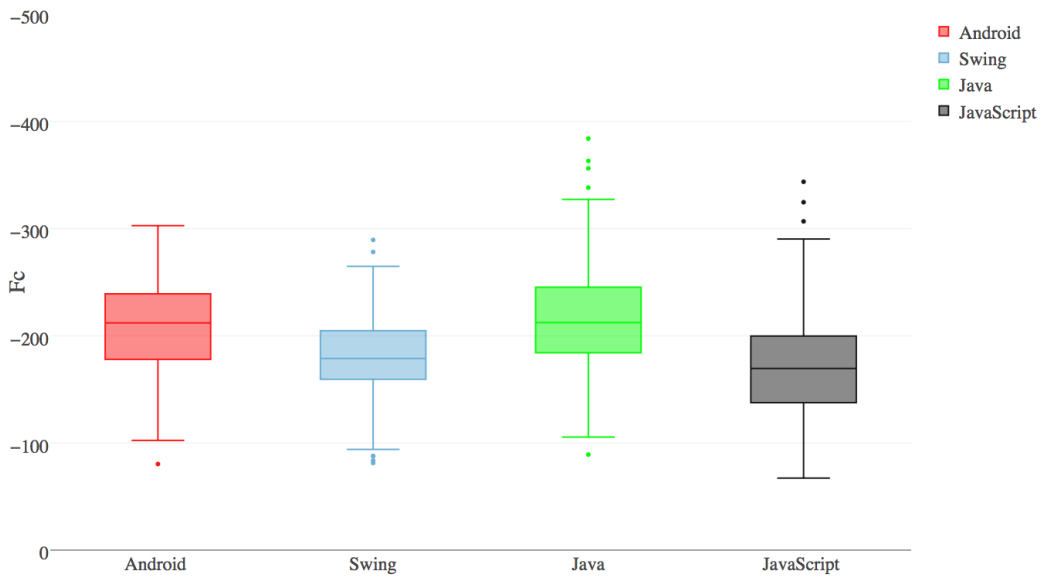

Figure 5.1. Code LM trained with 10 documents

Then we train the code LM with 100 Android documents, and as we can see in Fig 5.2, the Android testing documents are not estimated as the most familiar, since 100 documents are still not enough (Swing F_c mean $= -245$, JavaScript F_c mean $= -274$, Android F_c mean $= -280$, Java F_c mean $= -294$).

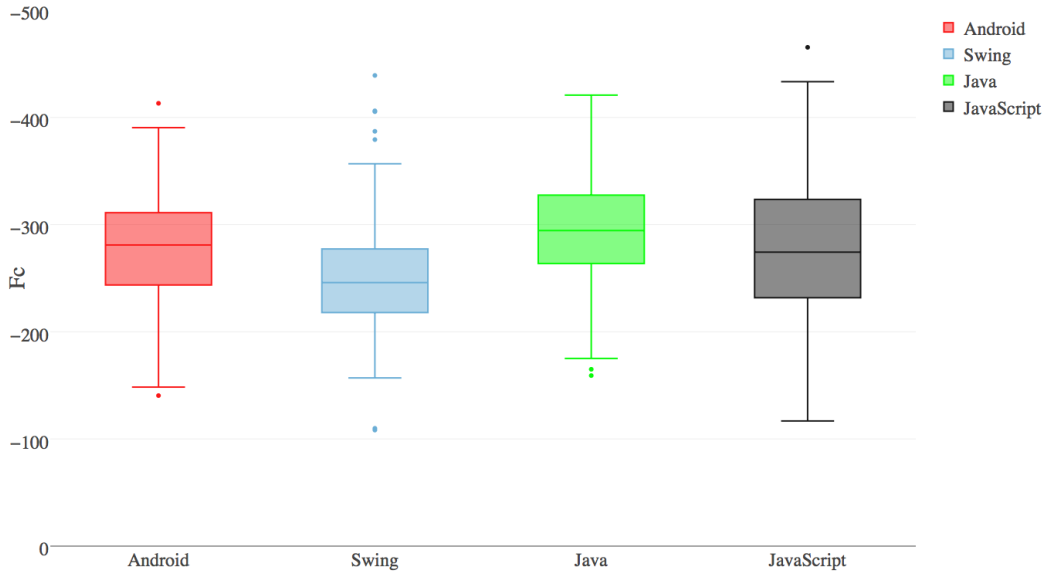

Figure 5.2. Code LM trained with 100 documents

With a training set of 1000 Android documents we start observing some changes, as we can see in Fig 5.3, Android testing documents are more familiar than Java and significantly more familiar than JavaScript (Swing F_c mean = −322, Android F_c mean = −331, Java F_c mean = −350, JavaScript F_c mean = −419).

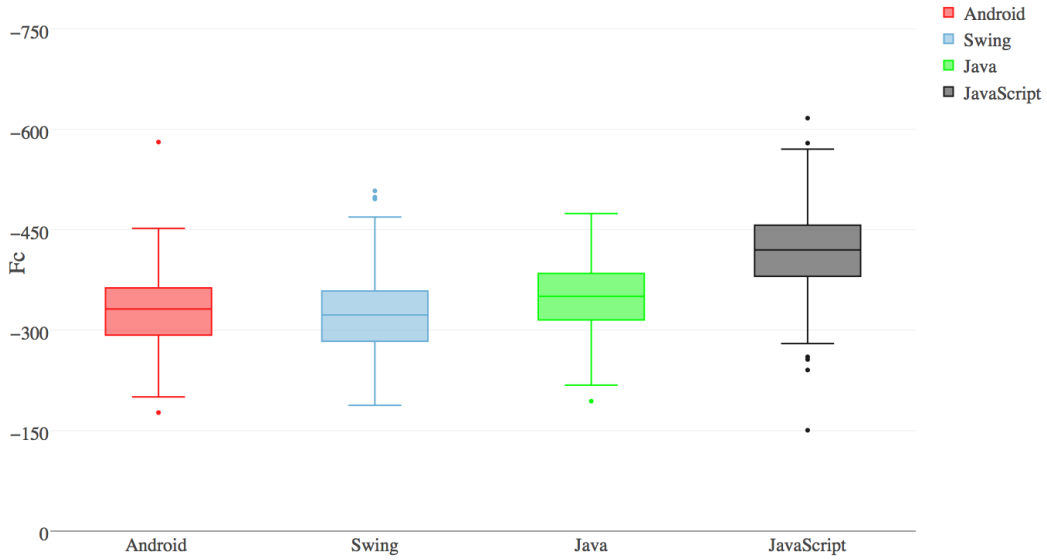

Figure 5.3. Code LM trained with 1000 documents

In Fig 5.4, we can see that with a training set of 10,000 documents, Android is estimated as the most familiar, followed by Java; then Swing and JavaScript, are clearly less familiar (Android F_c mean = −351, Java F_c mean = −378, Swing F_c mean = −405, JavaScript F_c mean = −483).

This result reflects our expectation, where Android is expected to be the most familiar, since the training set contains only Android documents, and as Android is implemented in Java we expected that Java should be more familiar than JavaScript and Swing. We have to mention that the familiarity result is logarithmic, therefore a small difference between the boxes position has to be interpreted as a big difference between the familiarity of the boxes.

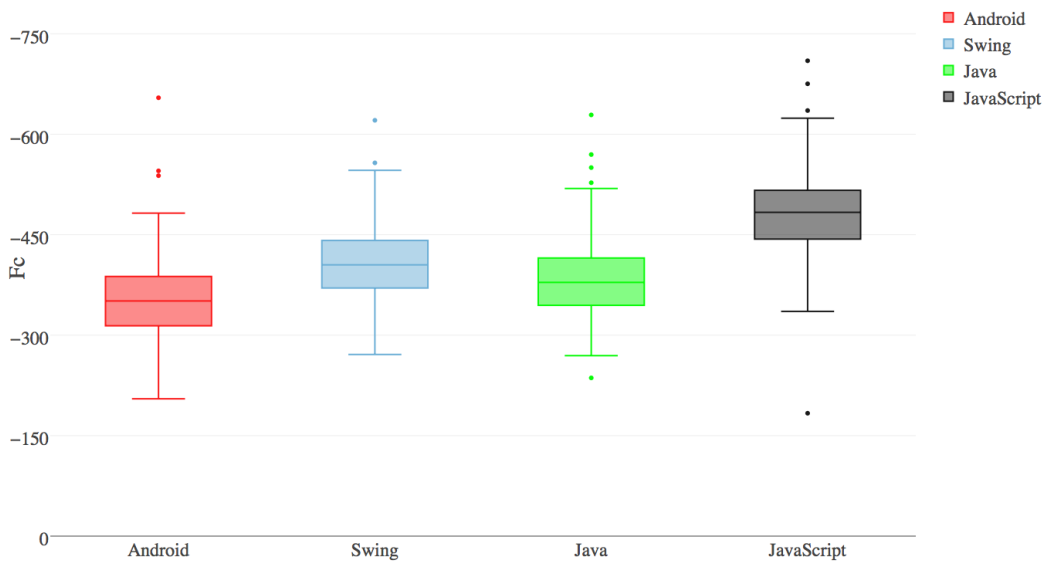

Figure 5.4. Code LM trained with 10000 documents

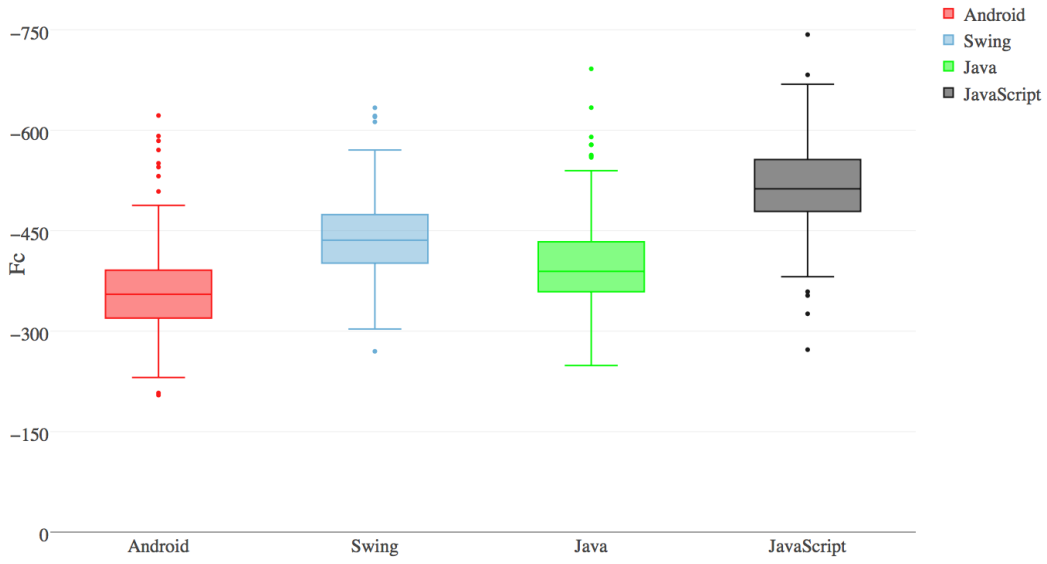

Figure 5.5. Code LM trained with 100000 documents

We also train the code LM with a set of 100,000 Android documents and as we can see in Fig 5.5, the result is similar to Fig 5.4 where the training set contains 10,000 documents (Android F_c mean $= -355$, Java F_c mean $= -389$, Swing F_c mean $= -435$, JavaScript F_c mean $= -512$).

These results infer that a LM can capture the code familiarity. In Fig 5.6, we can see that bigger is the training set, better is the performance. In this figure, the *y axis* represents the code familiarity mean, and the *x axis* represents the testing set size.

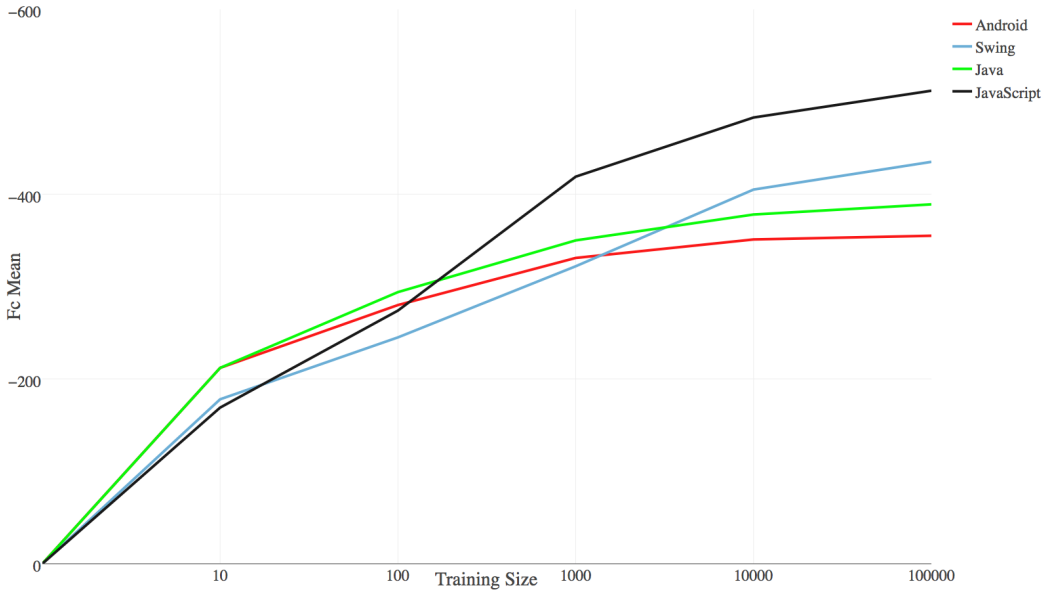

Figure 5.6. Code familiarity mean for each training set

5.1.2 Text LM Results

As we did for the Code LM, we train the text LM with 10, 100, 1000, 10,000 and 100,000 Android documents, and for each training set we estimate the familiarity of the testing set.

In Fig 5.7, we can see that Android documents are not the most familiar documents (Swing F_c mean $= -693$, JavaScript F_c mean $= -700$, Android F_c mean $= -752$, Java F_c mean $= -769$), and the reason is the small size of the training set.

Also, Fig 5.8, shows that Android documents are not the most familiar (Swing F_c mean $= -1140$, JavaScript F_c mean $= -1214$, Java F_c mean $= -1241$, Android F_c mean $= -1246$), and the reason still because the training set size is small.

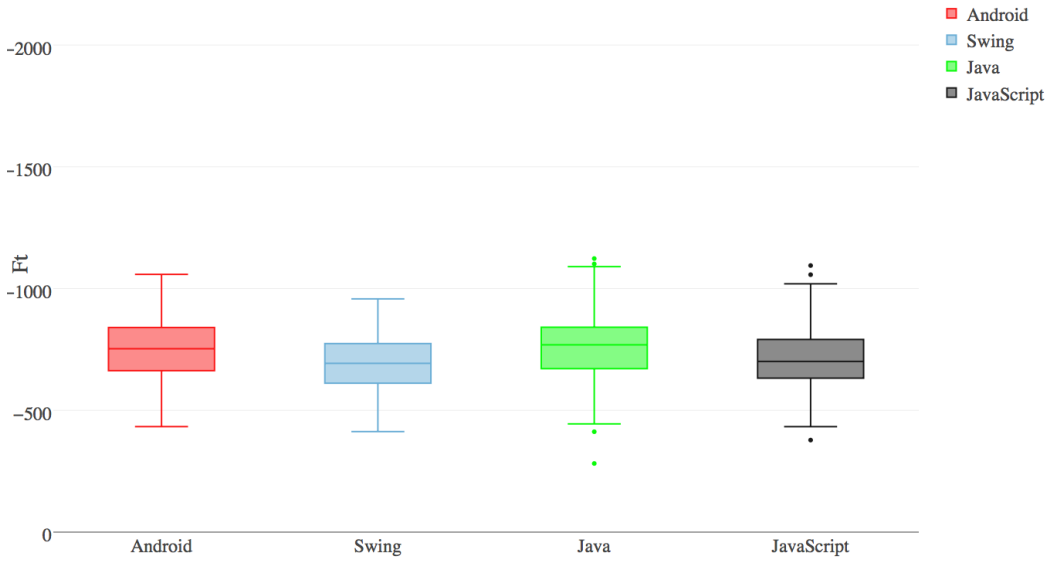

Figure 5.7. Code LM trained with 10 documents

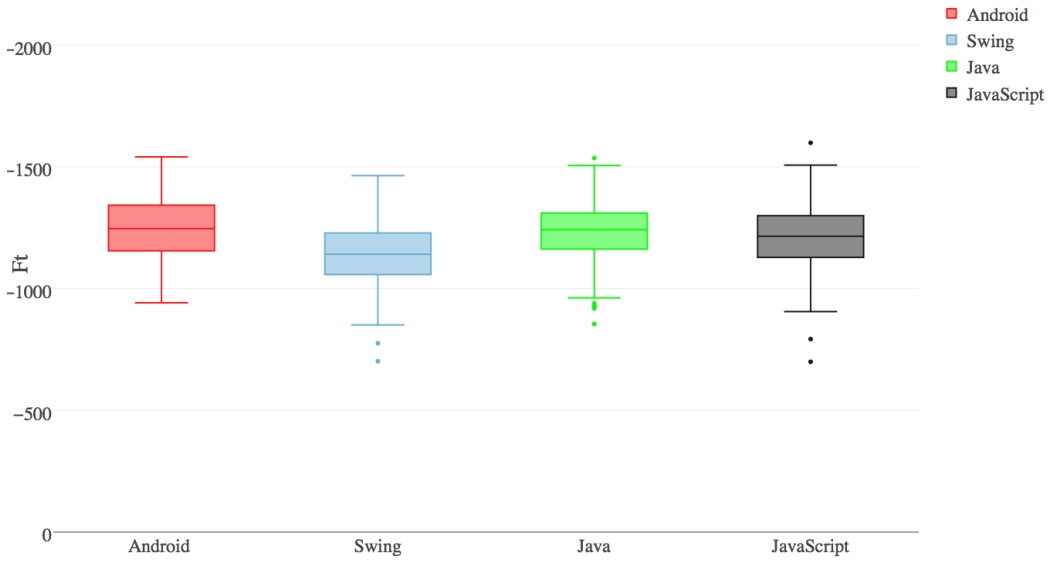

Figure 5.8. Code LM trained with 100 documents

In Fig 5.9, we can see that when we train the text LM with 1000 Android documents, Android is estimated as more familiar than Java and JavaScript (Swing F_c mean $= -1440$, Android F_c mean $= -1482$, Java F_c mean $= -1494$, JavaScript F_c mean $= -1497$), and in Figure 5.10 we train the code LM with

10,000 documents which estimate Android as the most familiar (Android F_c mean $= -1499$, Swing F_c mean $= -1535$, Java F_c mean $= -1546$, JavaScript F_c mean $= -1555$).

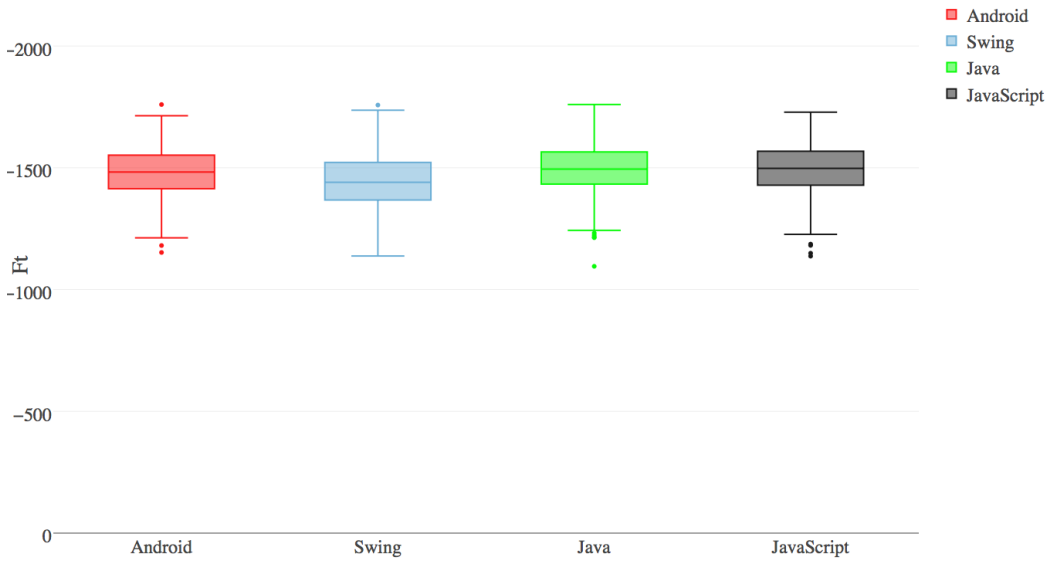

Figure 5.9. Code LM trained with 1000 documents

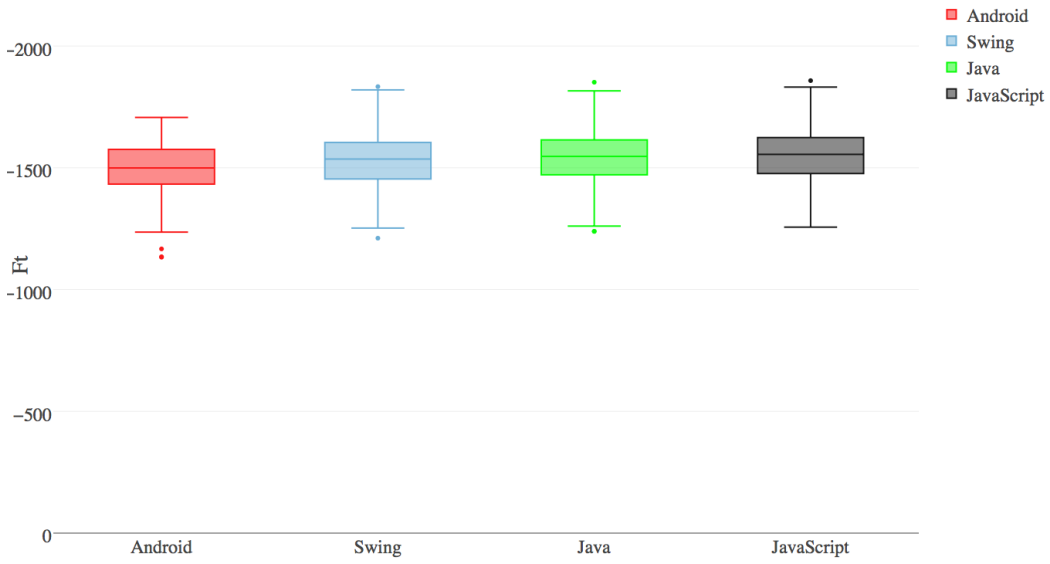

Figure 5.10. Code LM trained with 10000 documents

We get the expected result when we train the text LM with as set of 100,000 documents. In Fig 5.11, we can see that Android is the most familiar and then Java which is more familiar than JavaScript and Swing (Android F_c mean $= -1488$, Java F_c mean $= -1532$, Swing F_c mean $= -1599$, JavaScript F_c mean $= -1575$).

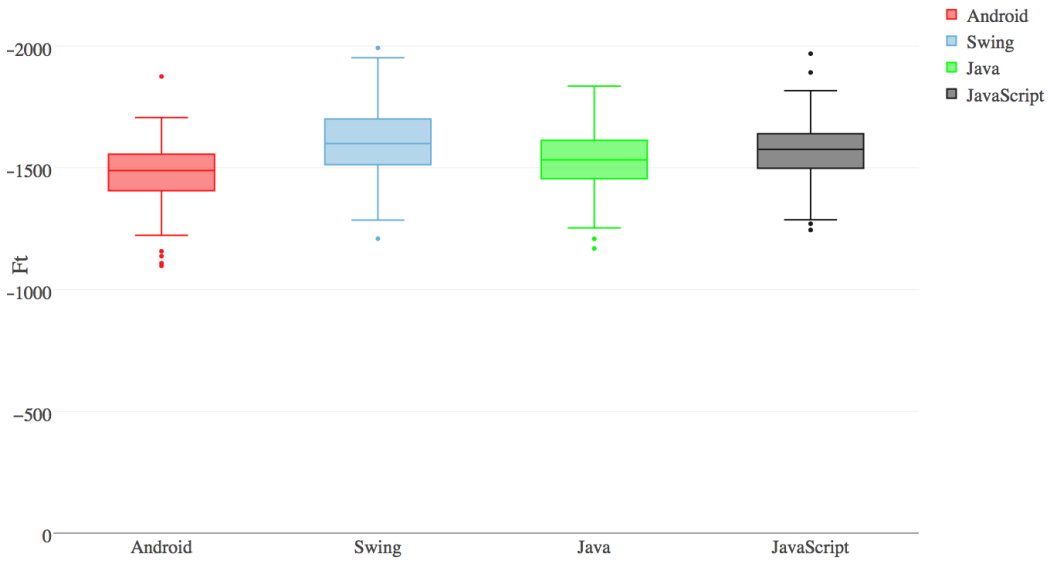

Figure 5.11. Code LM trained with 100000 documents

These results infer that the LM can capture the familiarity of text. Fig 5.12 shows that to get a better performance we need to train the LM with a large set of documents. In this figure, the *y axis* represents the text familiarity mean, and the *x axis* represents the testing set size.

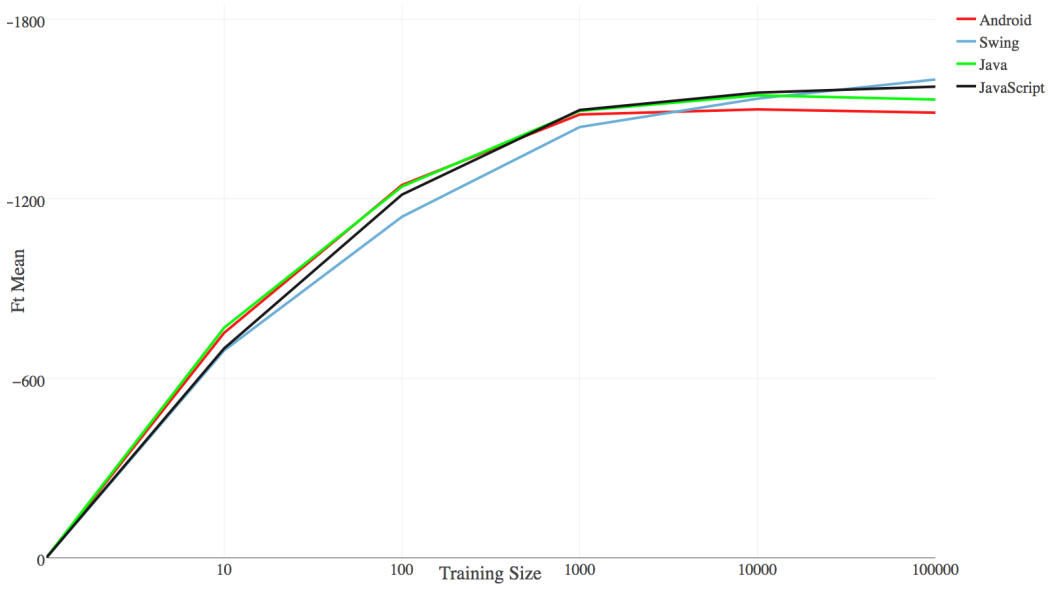

Figure 5.12. Text familiarity mean for each training set

5.2 Comprehension Effort Estimation Results

To evaluate our comprehension effort estimation, we ask two developers to read 4 introduction tutorials to Android, and one Android-Camera API tutorial and one Android-Bluetooth API tutorial, and as described in the Chapter 4.1, we ask them to score the comprehension effort of 6 documents on a scale of 0 to 5, where a higher score means a higher effort. In the meanwhile, we train the LMs and we estimate the comprehension effort that we compare with their score.

Tutorial	F_c	F_t	R_c	R_t	**Effort**	**Dev 1**	**Dev 2**
Bluetooth	-2694.64	-377.42	0.19	0.54	1166.6	1	1
Camera	-2794.77	-309.74	0.15	0.47	1259.8	2	2
AsyncTask	-3446.07	-265.87	0.25	0.53	1348.2	3	3
Database	-3491.46	-336.06	0.24	0.62	1377.4	4	3
Cordova-1	-4539.99	-313.44	0.33	0.49	1582.5	4	1
Cordova-2	-4429.52	-273.63	0.15	0.41	1944.5	2	1

Table 5.1. Comprehension effort evaluation. Formula-1

Tutorial	F_c	F_t	R_c	R_t	**Effort**	**Dev 1**	**Dev 2**
Camera	-2794.77	-309.74	0.15	0.47	1836.9	2	2
Bluetooth	-2694.64	-377.42	0.19	0.54	1854.6	1	1
AsyncTask	-3446.07	-265.87	0.25	0.53	2217.3	3	3
Database	-3491.46	-336.06	0.24	0.62	2439.5	4	3
Cordova-1	-4539.99	-313.44	0.33	0.49	2712.0	4	1
Cordova-2	-4429.52	-273.63	0.15	0.41	2729.8	2	1

Table 5.2. Comprehension effort evaluation. Formula-2

In Table 5.1 and Table 5.2 we show our estimation effort for each tutorial. As we mentioned in Chapter 3.4 we explore two formulas to compute the comprehension effort. Formula-1 :

$$\text{Comprehension Effort} = \frac{(r_c \times f_c) + (r_t \times f_t)}{2}$$

In the above formula the *readability* and the *familiarity* have the same weight in calculating the *comprehension effort*. Formula-2:

$$\text{Comprehension Effort} = \frac{(r_c \times f_c) + (r_t \times f_t)}{r_c + r_t}$$

In the above formula the *familiarity* has a higher weight. The tables contain also the developers scores (Dev 1, Dev 2). As we can see in In Table 5.1 and Table 5.2, our approach estimates *Bluetooth* and *Camera* tutorials as the documents that require less effort to be comprehended, and both developers gave them a low score (1 and 2), we also estimate that *AsyncTask* and *Database* require more effort, and also both developers gave them a higher score (3 and 4).

We estimate *Cordova-1* and *Cordova-2* to require the highest effort. The first developer gave them 4 and 2 which still a high score, but the second developer gave them 1, which is a low score. We asked the developer if he can justify the low score that he gave to the *Corodova* documents, and he pointed out that JavaScript is the programing language that he knows best. Therefore, he is familiar with JavaScript, and Cordova is implemented with JavaScript, which justifies the low score.

The results in Table 5.1 and Table 5.2 are similar, the only difference is that Formula-1 estimates that the *Camera* document is easier to comprehend compared to the *Bluetooth* document, which matches with the developers estimation. Therefore in this experiment Formula-1 performs better.

5.3 Summary

To evaluate our familiarity estimation approach, we ran an experiment where we evaluate the LM efficiency in estimating the familiarity of a given document. In this experiment we tried different training sets sizes, and the results showed that 10 or 100 training documents are not enough to build a LM able to capture the familiarity. The LM needs larger set to perform well. With a training set of 10,000 documents we got a very good performance.

To evaluate the comprehension effort estimation approach, we asked two developers to read a set of tutorials and then to score a set of StackOverflow discussions and we compare their score with our precomputed score. The results indicate that our score match with the developers score, therefore we believe that our approach is promising in estimating the comprehension effort.

Chapter 6

Threats to Validity

6.1 Experiment-1

Two potential threats to validity concern the training and testing sets in Experiment-1. The experiment is based on training LMs on Android discussions and evaluating the familiarity of different testing sets (Android, Java, Swing, JavaScript). The experiment considers only Android framework to show that LMs are efficient in estimating Android familiarity, but we did not run other studies to evaluate our approach on different frameworks. To generalize our approach we need to evaluate other frameworks or even other programming languages. We will investigate this further in future research.

The testing and training sets are StackOverflow discussions, where documents contain code snippets and text. In this experiment, the code LM is trained on small and some times incomplete code. If we run this experiment in a different context as any system where the code is complete and much bigger than Stack Overflow discussions, we might have different performance in estimating the familiarity.

6.2 Experiment-2

In Experiment-2, the main threats are related to the data set size. In this study, we asked two developers who have no experience with Android framework to read a set of 6 Android tutorials (training set) and to score the comprehension effort of a set of Stack Overflow discussions (testing set). To compare the developers' scores with our estimation we trained the LMs with the same training set. But as we mentioned in Chapter 5.1 the LM requires a large training set to

perform well, and 6 tutorials are too few. Therefore, we trained the code LM with a large set of Gist code snippets that have the same subject as the tutorials. This procedure can introduce some inconsistency since the training set that we gave to LMs is different form the one that we gave to the developers.

In this experiment, we compare our comprehension effort estimation with two developers scores. In order to generalize our approach, we need to run our experiment on a bigger sample. Furthermore, we could not run any statistical analysis as calculating the precision and accuracy of our approach because the sample size is too small, but this experiment is a starting point which suggests that our approach is promising in estimating the comprehension effort. We will investigate this further in future research.

Chapter 7

Conclusion

In this thesis, we have presented and evaluated the first approach in the literature able to assess documents by comprehension effort. The comprehension effort is based on the readability and the familiarity of the document. We have conducted a study over 100,000 Stack Overflow documents aimed to evaluate the language model efficiency to estimate the familiarity of a given document. The achieved results have shown that the language model is able to estimate the familiarity and we noticed that its performance is tightly related to the training set size. The best results have been achieved with a training set size $\geq$ 10,000 Stack Overflow documents.

We have conducted a study to evaluate our approach in estimating the comprehension effort. The sample size was not big enough to generalize the findings of your study, but the achieved results suggest that our approach is promising in estimating the comprehension effort. We beleive that the RSSE as *Libra* might take advantage of our comprehension effort metric to improve their suggestions.

Future Work

In the future studies aimed at replicating our work, we plan to run the familiarity evaluation experiment on a different framework or programing language. For example, we might train a LM on Java and to evaluate its efficiency in predicting the familiarity of Java and a C++ sets, where these two programing languages have a similar syntax.

We plan to run the comprehension effort evaluation experiment on a larger sample in order to generalize our estimating approach. Moreover with a large sample we can have a better evalaution to understand which effort estiamting folmula performs better.

Finally, it is important to note that the comprehension effort metric described in this thesis is not intended as the final model where only the readability and familiarity of a document determine the required human effort to comprehend it. Other metrics, (e.g., cyclomatic complexity) can be explored and added to readability and familiarity metrics to compute the comprehension effort.

Bibliography

[1] Sushil Krishna Bajracharya and Cristina Videira Lopes. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, 17(4): 424–466, Aug 2012. ISSN 1573-7616. doi: 10.1007/s10664-010-9144-6. URL https://doi.org/10.1007/s10664-010-9144-6.

[2] Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Trans. Softw. Eng.*, 36(4):546–558, July 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.70. URL http://dx.doi.org/10.1109/TSE.2009.70.

[3] R.P.L. Buse and W.R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, jul. 2010.

[4] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, June 1989. ISSN 0018-8670. doi: 10.1147/sj.282.0294. URL http://dx.doi.org/10.1147/sj.282.0294.

[5] J.Chall E.Dale. *Educational Research Bulletin*.

[6] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337322.

[7] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062491. URL http://doi.acm.org/10.1145/1062455.1062491.

[8] Rogers.RL Chissom.BS Kincaid.JP, Fishburne.RPJr. *Research Branch Report 8-75, Millington, TN: Naval Technical Training, U. S. Naval Air Station, Memphis, TN*.

[9] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes. Automatically locating relevant programming help online. *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2012)*, 00:127–134, 2012. ISSN 1943-6092. doi: doi.ieeecomputersociety.org/10.1109/VLHCC.2012.6344497.

[10] Dharmender Singh Kushwaha and A. K. Misra. Improved cognitive information complexity measure: A metric that establishes program comprehension effort. *SIGSOFT Softw. Eng. Notes*, 31(5):1–7, September 2006. ISSN 0163-5948. doi: 10.1145/1163514.1163533. URL http://doi.acm.org/10.1145/1163514.1163533.

[11] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-442440-6.

[12] Collin McMillan. Finding relevant functions in millions of lines of code. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1170–1172, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1986032. URL http://doi.acm.org/10.1145/1985793.1986032.

[13] Roberto Minelli, Andrea Mocci and, and Michele Lanza. I know what you did last summer: An investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 25–35, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2820282.2820289.

[14] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of MSR 2014 (11th Working Conference on Mining Software Repositories)*, pages 102–111. ACM, 2014.

[15] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. In *Proceedings of MSR 2015 (12th Working Conference on Mining Software Repositories)*, pages 474–477. ACM Press, 2015.

[16] Luca Ponzanelli, Simone Scalabrino, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Supporting software developers with a holistic recommender system. In *Proceedings of ICSE 2017 (39th ACM/IEEE International Conference on Software Engineering)*. to be published, 2017.

[17] Mohammad Masudur Rahman and Chanchal K. Roy. Recommending relevant sections from a webpage about programming errors and exceptions. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 181–190, Riverton, NJ, USA, 2015. IBM Corp. URL http://dl.acm.org/citation.cfm?id=2886444.2886471.

[18] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070525. URL http://dx.doi.org/10.1109/ICSE.2009.5070525.

[19] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, July 2010. ISSN 0740-7459. doi: 10.1109/MS.2009.161.

[20] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: A case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 191–201, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786855. URL http://doi.acm.org/10.1145/2786805.2786855.

[21] Nicholas Sawadsky and Gail C. Murphy. Fishtail: From task context to source code examples. In *Proceedings of the 1st Workshop on Developing Tools As Plug-ins*, TOPI '11, pages 48–51, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0599-0. doi: 10.1145/1984708.1984722. URL http://doi.acm.org/10.1145/1984708.1984722.

[22] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, 2017.

[23] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.*, 21(1):4:1–4:25, December 2011. ISSN 1049-331X. doi: 10.1145/2063239.2063243. URL http://doi.acm.org/10.1145/2063239.2063243.

[24] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 21–. IBM Press, 1997. URL http://dl.acm.org/citation.cfm?id=782010.782031.

[25] Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding api components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2586-5. doi: 10.1109/VLHCC.2006.32. URL http://dx.doi.org/10.1109/VLHCC.2006.32.

[26] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321663. URL http://doi.acm.org/10.1145/1321631.1321663.

[27] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. URL http://dl.acm.org/citation.cfm?id=776816.776866.

[28] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. URL http://dl.acm.org/citation.cfm?id=998675.999460.