**University Institute of Lisbon**

Department of Information Science and Technology


# Stepwise API Usage Assistance based on N-gram Language Models


Gonçalo Queiroga Prendi


A Dissertation presented in partial fulfillment of the Requirements for the Degree of

**Master in Computer Engineering**


**Supervisor**

Doctor André Leal Santos, Assistant Professor

ISCTE-IUL

**Co-Supervisor**

Doctor Ricardo Daniel Santos Faro Marques Ribeiro, Assistant Professor

ISCTE-IUL


September, 2015

*"The only source of knowledge is experience"*

Albert Einstein

# *Resumo*

O desenvolvimento de software requer a utilização de *Application Programming Interfaces* (APIs) externas com o objectivo de reutilizar bibliotecas e *frameworks*. Muitas vezes, os programadores têm dificuldade em utilizar APIs desconhecidas, devido à falta de recursos ou desenho fora do comum. Essas dificuldades provocam inúmeras vezes sequências incorrectas de chamadas às APIs que poderão não produzir o resultado desejado. Os modelos de língua mostraram-se capazes de capturar regularidades em texto, bem como em código.

Neste trabalho é explorada a utilização de modelos de língua de $n$-gramas e a sua capacidade de capturar regularidades na utilização de APIs, através de uma avaliação intrínseca e extrínseca destes modelos em algumas das APIs mais utilizadas na linguagem de programação Java. Para alcançar este objectivo, vários modelos foram treinados sobre repositórios de código do GitHub, contendo centenas de projectos Java que utilizam estas APIs. Com o objectivo de ter uma avaliação completa do desempenho dos modelos de língua, foram seleccionadas APIs de múltiplos domínios e tamanhos de vocabulário.

Este trabalho permite concluir que os modelos de língua de $n$-gramas são capazes de capturar padrões de utilização de APIs devido aos seus baixos valores de perplexidade e a sua alta cobertura, chegando a atingir 100% em alguns casos, o que levou à criação de uma ferramenta de *code completion* para guiar os programadores na utilização de uma API desconhecida, mas mantendo a possibilidade de a explorar.

**Palavras-chave:** $N$-gram Language Models, Usabilidade das APIs, Perplexidade, Code Completion.

# *Abstract*


Software development requires the use of external Application Programming Interfaces (APIs) in order to reuse libraries and frameworks. Programmers often struggle with unfamiliar APIs due to their lack of resources or less common design. Such difficulties often lead to an incorrect sequences of API calls that may not produce the desired outcome. Language models have shown the ability to capture regularities in text as well as in code.

In this work we explore the use of $n$-gram language models and their ability to capture regularities in API usage through an intrinsic and extrinsic evaluation of these models on some of the most widely used APIs for the Java programming language. To achieve this, several language models were trained over a source code corpora containing several hundreds of GitHub Java projects that use the desired APIs. In order to fully assess the performance of the language models, we have selected APIs from multiple domains and vocabulary sizes.

This work allowed us to conclude that $n$-gram language models are able to capture the API usage patterns due to their low perplexity values and their high overall coverage, going up to 100% in some cases, which encouraged us to create a code completion tool to help programmers stay in the right path when using unknown APIs while allowing for some exploration.



**Keywords:** $N$-gram Language Models, API usability, Perplexity, Code Completion.

# *Acknowledgements*

In first place I would like to acknowledge my supervisors for all the motivation and support throughout the whole work as well as all the knowledge they have passed, that will be critical in my professional future.

Secondly I would like to acknowledge my parents, for helping and motivating me to accomplish all my objectives. Furthermore for never giving up even when the hardest times strike, which was essential specially during my academic life.

Finally, an acknowledgement to my girlfriend and my closest friends for their unconditional support.

# Contents

**Resumo** v

**Abstract** vii

**Acknowledgements** ix

**List of Figures** xiii

**Abbreviations** xvii

**1 Introduction** 1
- 1.1 Context . . . . . 1
- 1.2 Motivation . . . . . 2
- 1.3 Approach . . . . . 3
- 1.4 Research Questions . . . . . 4
- 1.5 Objectives . . . . . 5
- 1.6 Research Method . . . . . 5
- 1.7 Document Structure . . . . . 6

**2 Related Work** 7
- 2.1 N-Gram Language Models . . . . . 8
  - 2.1.1 Smoothing techniques . . . . . 9
    - 2.1.1.1 Witten-Bell . . . . . 9
    - 2.1.1.2 Kneser-Ney . . . . . 10
- 2.2 Method Call Recommenders . . . . . 11
- 2.3 Snippet/Sequence Recommenders . . . . . 14
- 2.4 Recommenders for a desired object type . . . . . 17
- 2.5 Summary . . . . . 18

**3 API Sentence Models** 19
- 3.1 Tokens . . . . . 19
- 3.2 Sentences . . . . . 20
- 3.3 Extraction Process . . . . . 21
  - 3.3.1 Composite expressions . . . . . 21
  - 3.3.2 Token dependencies . . . . . 22
  - 3.3.3 Selections and loops . . . . . 24

3.3.4 Sentence sample vocabulary . . . . . 25
3.4 Definition of API Sentence Models . . . . . 25
3.5 Limitations . . . . . 25

**4 Measuring API Perplexity 27**
4.1 Setup . . . . . 28
4.2 SWT API . . . . . 30
4.3 Swing API . . . . . 30
4.4 JFreeChart API . . . . . 31
4.5 JSoup . . . . . 32
4.6 JDBC Driver for MySQL . . . . . 33
4.7 Jackson-core . . . . . 33
4.8 Discussion . . . . . 34

**5 Recommendations Evaluation 37**
5.1 Method . . . . . 37
5.2 Results . . . . . 39
5.3 Discussion . . . . . 41

**6 Stepwise Code Completion tool 47**
6.1 Recommendation (APISTA Tool) . . . . . 47
6.2 User interface . . . . . 49
6.3 Integration in software development . . . . . 50
6.4 Limitations . . . . . 51

**7 Conclusions and Future Work 53**

**Appendices 57**

**A Overall Coverage and Average Hit Index Results 57**
A.1 JDBC Driver for MySQL . . . . . 57
A.2 Jackson-core . . . . . 58

**Bibliography 61**

# List of Figures


1.1 Eclipse standard code completion system. . . . . . . . . . . . . . . . . . 2
1.2 Code Recommenders code completion system [8]. . . . . . . . . . . . 2
1.3 Overview of components and stakeholders involved in our approach. The three main components are depicted in gray. The recommender component is a plugin to an IDE. . . . . . . . . . . . . . . . . . . . . 3

2.1 Recommendation systems for code completion. . . . . . . . . . . . . 7
2.2 Call completion feature example, on the Code Recommenders intelligent code completion [1]. . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3 The BCC tool sorts the available methods by declaration type first and by hierarchy type secondly. . . . . . . . . . . . . . . . . . . . . . 12
2.4 The available methods are sorted by popularity.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.5 RASCAL prototype overview [24]. . . . . . . . . . . . . . . . . . . . 14
2.6 Partial program using multiple APIs. . . . . . . . . . . . . . . . . . 15
2.7 Holes automatically replaced with SLANG's proposals. . . . . . . . 15
2.8 Overview on the recommender component of the MAPO tool [41]. . 16
2.9 PARSEWeb tool interface. . . . . . . . . . . . . . . . . . . . . . . . 17

3.1 API Sentence model based on $n$-gram language models with $n = 2$. 26
3.2 Sample of observations $\Omega = (\omega_0, \omega_1, \omega_2)$. . . . . . . . . . . . . . . . . 26

4.1 Perplexity values for the SWT API. . . . . . . . . . . . . . . . . . . 30
4.2 Perplexity values for the Swing API. . . . . . . . . . . . . . . . . . 31
4.3 Perplexity values for the JFreeChart API. . . . . . . . . . . . . . . 31
4.4 Perplexity values for the JSoup API. . . . . . . . . . . . . . . . . . 32
4.5 Perplexity values for the JDBC Driver API. . . . . . . . . . . . . . 33
4.6 Perplexity values for the Jackson-core API. . . . . . . . . . . . . . 34
4.7 Perplexity values comparison for all the APIs. . . . . . . . . . . . . 34
4.8 Relation between vocabulary size and perplexity. . . . . . . . . . . 35

5.1 $N$-gram overall coverage for the SWT API with Kneser-Ney smoothing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.2 $N$-gram overall coverage for the SWT API with Witten-Bell smoothing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

5.3 Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the SWT API. . . . . 41
5.4 Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the SWT API. . . . . 41
5.5 $N$-gram overall coverage for the Swing API with Kneser-Ney smoothing. . . . . 42
5.6 $N$-gram overall coverage for the Swing API with Witten-Bell smoothing. . . . . 42
5.7 Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the Swing API. . . . . 43
5.8 Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the Swing API. . . . . 43
5.9 $N$-gram overall coverage for the JFreeChart API with Kneser-Ney smoothing. . . . . 43
5.10 $N$-gram overall coverage for the JFreeChart API with Witten-Bell smoothing. . . . . 43
5.11 Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the JFreeChart API. . . . . 44
5.12 Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the JFreeChart API. . . . . 44
5.13 $N$-gram overall coverage for the JSoup API with Kneser-Ney smoothing. . . . . 44
5.14 $N$-gram overall coverage for the JSoup API with Witten-Bell smoothing. . . . . 44
5.15 Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the JSoup API. . . . . 45
5.16 Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the JSoup API. . . . . 45

6.1 Stepwise API usage assistance for writing API sentences using the code completion proposals of our APISTA tool in Eclipse. . . . . 48
6.2 User interaction with the APISTA tool. Parameters are matched with context variables (for instance, $c$). Unmatched parameters have to be completed manually (for instance, parameter *style*). . . . . 49

A.1 $N$-gram overall coverage for the JDBC Driver for MySQL API with Kneser-Ney smoothing. . . . . 57
A.2 $N$-gram overall coverage for the JDBC Driver for MySQL API with Witten-Bell smoothing. . . . . 57
A.3 Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the JDBC Driver API. . . . . 58
A.4 Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the JDBC Driver API. . . . . 58
A.5 $N$-gram overall coverage for the Jackson API with Kneser-Ney smoothing. . . . . 58

A.6 $N$-gram overall coverage for the Jackson API with Witten-Bell smoothing. . . . . . 58
A.7 Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the Jackson API. . . . . . 59
A.8 Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the Jackson API. . . . . . 59

# Abbreviations

**API** **A**pplication **P**rogramming **I**nterface (see page 1)

**IDE** **I**ntegrated **D**evelopment **E**nvironment (see page 1)

**OOV** **O**ut **O**f **V**ocabulary (see page 29)

**SWT** **S**tandard **W**idget **T**oolkit (see page 30)

# Chapter 1

# Introduction

## 1.1 Context

Modern software development requires the use of several thousands of classes and methods distributed along library or framework components, that provide functionalities to implement certain features. These components provide an interface commonly referred to as Application Programming Interface (API) that abstracts implementation details, and is used by developers to interact with these components, with the objective of reusing code in order to increase developers' productivity and diminish coding errors.

An Integrated Development Environment (IDE), like Eclipse[1], is a software application that provides programmers certain functionalities, like an intelligent code completion system, that facilitate software development. These code completion systems assist on API usage by showing possible interactions to the programmer through context pop-up panes. These interactions on the Eclipse's default code completion system are limited to method calls and variables (see Figure 1.1). There are however some approaches that increase these system's potential by implementing other features like relevance-ordered method calls as seen in Figure 1.2.

[1] www.eclipse.org

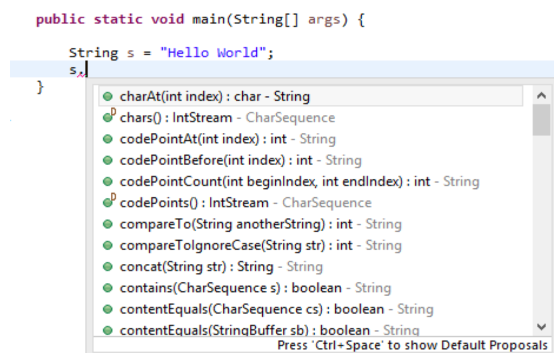


FIGURE 1.1: Eclipse standard code completion system.

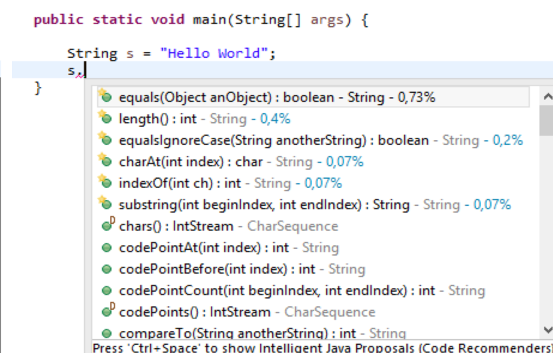


FIGURE 1.2: Code Recommenders code completion system [8].

Figure 1.1 presents Eclipse's standard code completion system, that orders all method calls alphabetically. This approach although it shows every possible interaction with the object, it might hinder the programmer's task since he/she has to search for relevant methods from a possibly long list that can exceed 100 different methods. In Figure 1.2, method calls are shown by their relevance which advices the programmer, on what he/she might be looking when using a new API, increasing the usability of the IDE and consequently the developer's productivity.

## 1.2 Motivation

According to Robillard [33], when developers are learning how to use an API, the major obstacle are its resources, that are often inadequate or absent, which may result in an inefficient API use or atrocious code structure. Often, the documentation to learn how to use a new API is unclear or incomplete, which results in usability flaws [27]. Examples can also be an obstacle if there is a mismatch between their purpose and the developer's goal. Regarding API design, developers often experience difficulties they cannot answer like the relationship between two types, how an object of a certain type can be created without a public constructor, and the existence of a helper-type to manipulate a certain object [12]. Even when the API provides factories to instantiate objects, this is significantly more time consuming than using a constructor [13]. These case studies show the lack of code recommendation tools that assist programmers overcoming API usability

hurdles. Enhanced code completion systems are a possible solution to help mitigate these obstacles consequently reducing programmers' learning curve, since it requires them less time browsing documentation or searching code examples.

## 1.3 Approach

There are several components and stakeholders involved in our approach (see Figure 1.3). Providing a recommendation system for a given API requires mining source code repositories where the API is used. Several processes have to be carried out for achieving robust and accurate recommendations. Namely, determining the API vocabulary in order to extract API sentences from source code repositories, building the API language model using the extracted information, and integrating the model in a code completion system that is able to recommend proposals of API usage that take into account the code that is being edited as context.

*API developers* write and maintain libraries and frameworks, whose API is used by several projects that are developed by a *programmer community*. Each project that uses the API is a candidate for being part of the source code corpora that is used to build the API *sentence model*. The broader the source code corpora is in terms of API coverage, the more complete the language model will be. The API

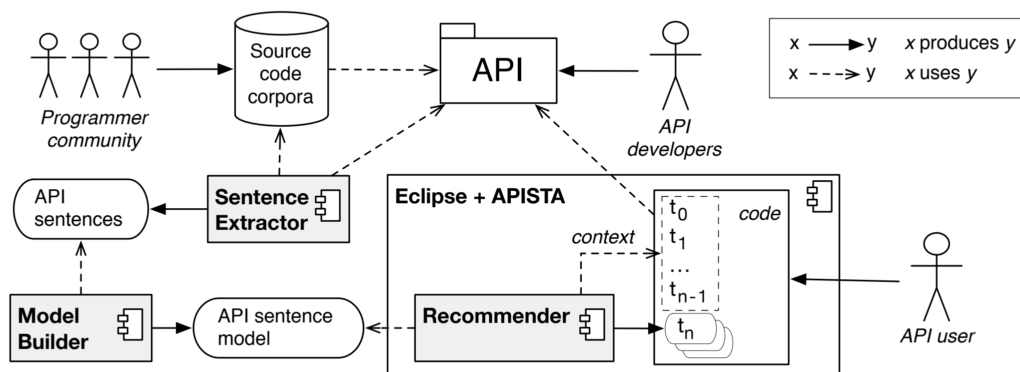


FIGURE 1.3: Overview of components and stakeholders involved in our approach. The three main components are depicted in gray. The recommender component is a plugin to an IDE.

source is used to determine its boundaries, in terms of which types belong to the API when mining the repositories.

There are two core processes for setting up our recommendation system for a particular API. The *sentence extractor* component mines a set of API sentences from the source code corpora. The *model builder* component uses the sentences set to build an API sentence model, which is based on $n$-gram language models. Moreover, the model is used by a *recommender component* in an IDE in order to assist API users with *proposals* that augment the API sentence that is being written in the code editor. The recommender is integrated with the code completion system of Eclipse, using the available facilities as the means to present the API usage proposals. When requesting code completion, the context in which the API user is writing code is given as input to the recommender, namely the tokens contained in the lines of code that precede the line where code completion is requested. API usage assistance is provided in a stepwise manner, through single-token proposals to augment the context.

## 1.4 Research Questions

This work was conducted around the following research questions that focus on enhancing IDE usability with the objective of decreasing the difficulties experienced by the programmers when learning and using an unfamiliar API.

1. How statistical language models for analysing source code corpora can be used to assist programmers in a step by step API usage through code completion proposals?
2. Which models best fit each API in order to produce the most accurate code completion proposals?
3. Which cases of API usage assistance cannot be effectively addressed by (1)?

## 1.5 Objectives

With this work, we intend to increase programmers productivity and decrease error insertion in code, due to the lack of knowledge, on the use of unknown APIs, by recommending valuable instructions in the programmer's context. To achieve this, a code recommendation prototype was implemented but with a different approach from the ones described in the related work. The prototype uses $n$-gram language models, created from the analysis of source code acquired from repositories, to obtain the recommendations according to the programmer's context. The models are intrinsically evaluated with a common measurement named perplexity. Regarding the performance, the models are evaluated with a cross-validation scheme in order to assess its assistance's quality and coverage by testing its proposals against the code found in the repositories. This evaluation will give us useful information on the prototype's performance and how its models can be tuned in order to provide better recommendations as well as information about how well it performs compared to other approaches.

## 1.6 Research Method

The Design Science Research method [15] is a problem solving process with different models of approach. The model [26] that will be used, is divided in the following steps:

1. Problem identification and motivation
2. Objectives of a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

The first three steps are briefly described in this Chapter in Sections 1.1, 1.2 and 1.5 respectively, and show the importance of this area of research and which problems this approach will address. The next steps consist of designing and implementing a solution based on a code completion system and evaluating how it answers the research questions. Finally, the communication will be addressed with a paper [30] and this dissertation.

## 1.7 Document Structure

This document provides more detailed information on each part of the approach and is divided as follows:

- Chapter 2 provides background information regarding our approach, namely $n$-gram language models and respective smoothing techniques and an overview of the existing tools, with a comparison and an explanation on how they differ from our approach;
- Chapter 3 formally describes how the API calls are represented in the language models, in the form of tokens and sentences as well as the source code extraction process and its importance when used with $n$-gram language models;
- Chapter 4 intrinsically evaluates the language models and discusses and compares the results;
- Chapter 5 extrinsically evaluates these models, as well as a discussion regarding the relation with the previous evaluation;
- Chapter 6 shows an example of how these language models can be applied in code completion systems;
- Chapter 7 concludes this work and proposes future directions.

# Chapter 2

# Related Work

Similarly to our approach, most of the tools presented in this chapter are built on top of the following three pillars (see Figure 2.1): (1) Source code miner to extract relevant code from the repository; (2) Source code analyser that analyses and organizes the extracted code in a way to be processed into relevant recommendations; (3) Recommendation system that ranks and displays the most relevant suggestions to the programmers, a feature typically implemented as an IDE extension.

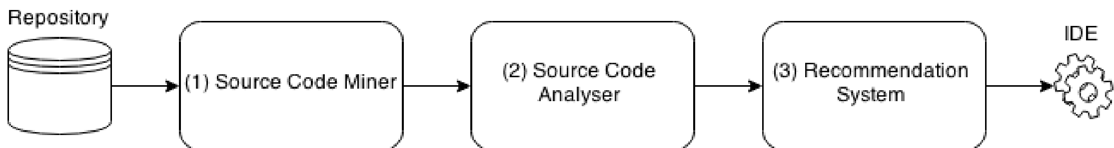


FIGURE 2.1: Recommendation systems for code completion.

Between the different approaches what changes is the implementation and detail of each one of the steps previously presented. The next section provides some background knowledge on $n$-gram language models, since some of the tools described in this chapter (as well as our approach), use these models to organize the extracted examples. Sections 2.2, 2.3 and 2.4 describe and aggregate the tools by their general goals.

## 2.1 N-Gram Language Models

Language models are widely used on several written and spoken language processing tasks, such as speech recognition [32], spell-checking and correction [28] and machine-translation [6]. These models allow the computation of the probability of a sentence or the estimation of how likely a history of tokens will be followed by a certain token. These probabilities are described as the product of a set of conditional probabilities. Hence, the probability of a sentence $\omega = (t_0, t_1, \cdots, t_n)$ is given by:

$$P(\omega) = P(t_0)P(t_1|t_0)P(t_2|t_0t_1)\cdots P(t_n|t_0t_1\cdots t_{n-1}) \tag{2.1}$$

Equation 2.1 represents a chain of conditional probabilities $P(t|h)$, where $t$ is the token and $h$ is the history or the previously written tokens. The maximum likelihood estimate is used to compute these probabilities, where $C()$ is the number of times the sequence appears in the training set (Equation 2.2).

$$P(t_n|t_0t_1\cdots t_{n-1}) = \frac{C(t_0t_1\cdots t_n)}{C(t_0t_1\cdots t_{n-1})} \tag{2.2}$$

Since it is not reliable to estimate probabilities for long histories, it is common to set a limit $N$ for these long histories using the Markov assumption. The probabilities are then computed as follows:

$$P(t_n|t_{n-N+1}\cdots t_{n-1}) = \frac{C(t_{n-N+1}\cdots t_n)}{C(t_{n-N+1}\cdots t_{n-1})} \tag{2.3}$$

$N$-gram language models are some of the several existing statistical models and have been previously used to capture regularities in source code in general [16]. Our intuition was that $n$-gram language models would work well for API client code, since APIs usually have common usage patterns or regularities with relatively small histories, which can be captured with these models.

### 2.1.1 Smoothing techniques

Language models suffer from a problem called data sparseness. This problem especially arises when the corpora is too small and does not contain all possible sentences, which will result in a zero probability for those sentences when querying the model, due to the nature of the maximum likelihood estimate. Smoothing techniques are used to address this problem by producing more accurate probabilities. This is done by adjusting the maximum likelihood estimate of probabilities [10] by attempting to increase the lowest probabilities (such as zero) and decrease the highest ones resulting on a more uniform probability distribution, and hence, producing a more accurate model. The SRILM toolkit [36] supports the smoothing techniques that we have experimented in our evaluation, namely the Witten-Bell and Kneser-Ney methods.

#### 2.1.1.1 Witten-Bell

If a given $n$-gram occurs in the training set, it is reasonable to assume that we should use the highest order $n$-gram in order to calculate the probability of the next token. The model in this situation will be much more accurate than using the lower order ones that may recommend a larger set of possible tokens. However, when the $n$-gram does not appear in the training set, we can backoff to the lower order ones. Equation 2.4 shows how the Witten-Bell smoothing addresses this problem.

$$P_{\mathrm{WB}}(t_i|t_{i-n+1}^{i-1}) = \lambda_{t_{i-n+1}^{i-1}} P(t_i|t_{i-n+1}^{i-1}) + (1 - \lambda_{t_{i-n+1}^{i-1}}) P_{\mathrm{WB}}(t_i|t_{i-n+2}^{i-1}) \tag{2.4}$$

This equation is based on a linear interpolation between the maximum likelihood estimate of the $n$-order model and the $(n-1)$-order smoothed model. The weight $\lambda$ given to the lower order models is proportional to the probability of having an unknown word with the current history (in our experiments, this value was estimated by the SRILM toolkit).

#### 2.1.1.2 Kneser-Ney

A very frequent token $t_j$ would be represented in the model with a very high unigram probability. However, if this token is only observed after another token $t_k$ in the training set, it is very unlikely it will appear after another token $t_l$. A smoothed probability estimate may be high if the unigram probability of $t_j$ is taken into account when backing-off. Equation 2.5 shows how the Kneser-Ney [20] smoothing technique addresses this problem, where D is a constant given by Equation 2.6 ($n1$ and $n2$ are the total number of $n$-grams with exactly one and two counts) that is subtracted to the $n$-gram count, when computing the discounted probability and $\gamma(t_{i-n+1}^{i-1})$ is used to make the distribution sum to 1 [11][1].

$$P_{\mathrm{KN}}(t_i|t_{i-n+1}^{i-1}) = \begin{cases} \frac{\max\{C(t_{i-n+1}^{i-1})-D,0\}}{\sum_{ti} C(t_{i-n+1}^{i-1})} & \text{if } C(t_{i-n+1}^{i-1}) > 0 \\ \gamma(t_{i-n+1}^{i-1}) P_{\mathrm{KN}}(t_i|t_{i-n+2}^{i-1}) & \text{if } C(t_{i-n+1}^{i-1}) = 0 \end{cases} \tag{2.5}$$

$$D = n1/(n1 + 2 * n2) \tag{2.6}$$

This method consists of taking into account the number of different tokens that precede a token $t_j$ to calculate the unigram probabilities, and not only the number of occurrences of that token.

While learning an unknown API, programmers tend to search for resources in order to complete a task. If most of the programmers use a certain example, that pattern will have a very high probability in the model. However, when executing a more specific task, programmers might use some uncommon operations, that will be reflected in the model with a low probability, since they will occur less often in the source code corpora. A code recommendation tool based on this type of models will keep the programmer on the correct path, but would allow for some exploration of the API by providing some of the uncommon operations.

[1] For more details on the computation of these values see http://www.speech.sri.com/projects/srilm/manpages/ngram-discount.7.html

## 2.2 Method Call Recommenders

The code completion system developed by Marcel Bruch et al. [9], uses a modified $k$-nearest-neighbours [5] called "Best Matching Neighbour", that recommends method calls for particular objects, by extracting the context of the variable, searching the codebase for similar situations and synthesize method recommendations. Figure 2.2 gives an example of the "Call Completion" feature.

This code completion system led to an industry-level Eclipse Project named Code Recommenders [8], along with some extra features like "Override Completion" that recommends which methods are usually overridden when extending a certain API class, "Chain Completion" suggests chains of method calls that return the desired type and "Adaptive Template Completion" that recommends multiple methods that frequently occur together on an object. Being this tool the most complete we have found and although the system provides several different recommendations programmers usually need with a certain object, it does not assist the programmer with the next object he might need in order to correctly interact with the API, which limits the tool because the programmer must know at least which classes exist in the API. This is also a problem if the documentation is unclear.

Hindle et al. [16] apply language models to code and try to find evidence that software is far more regular than, for example, English, and that these techniques can be applied to code completion systems. To test this, they implemented a

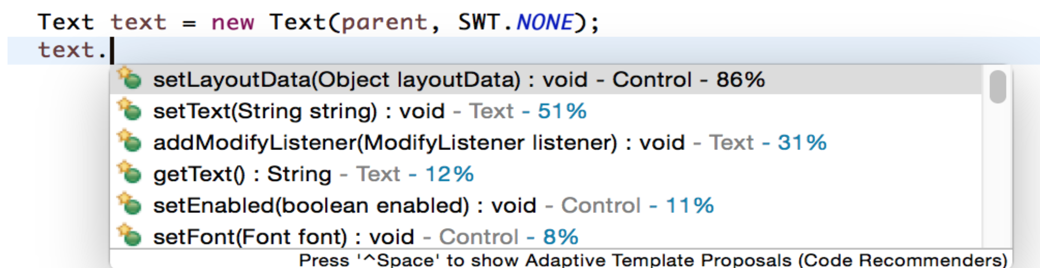


FIGURE 2.2: Call completion feature example, on the Code Recommenders intelligent code completion [1].

system called "corpus-based $n$-gram models suggestion engine" that tries to guess the next token based on a static corpus of source code. Their approach differs from ours in the aspect that we focus only on creating models that can represent the API usage, which is much more restrict and regular than source code in general, thus allowing to produce more accurate recommendations. Although their work also measures the models' perplexity, we argue that an extrinsic evaluation is required in order to fully assess the prototype. Nevertheless, it was one of the approaches that motivated the use of $n$-gram language models on our approach.

Better Code Completion (BCC) [19, 29] sorts, filters, and groups API methods. Sorting has two options; the first is a type-hierarchy-based sorting, which proposes the methods from the declared type before its super type, which is very useful since in certain contexts methods like `wait()` are never used; the second is a popularity-based sorting, that sorts the recommendations based on the frequency of a method call, similar to our and other approaches. Figures 2.3 and 2.4 show how the sorting feature works [19].

Their approach also allows to manually filter API methods, i.e. define context-sensitive filters since certain methods have to be public because the sub-classes that invoke them are located outside the package. BCC allows developers to configure groups of methods that will be displayed together in the code completion pane.

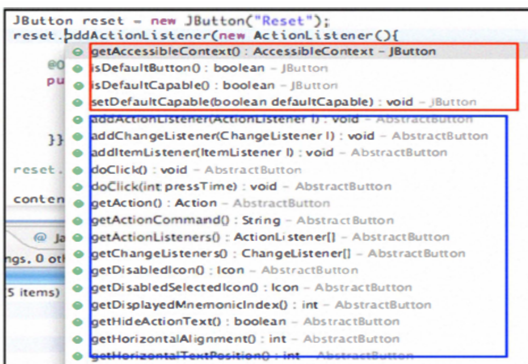


FIGURE 2.3: The BCC tool sorts the available methods by declaration type first and by hierarchy type secondly.

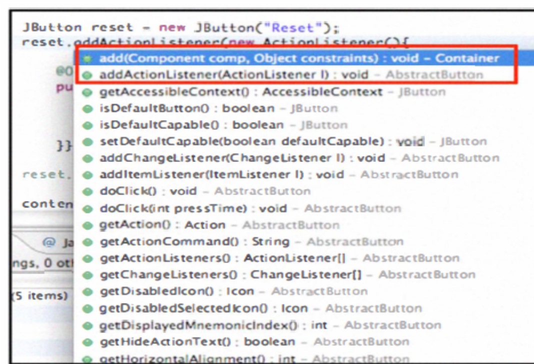


FIGURE 2.4: The available methods are sorted by popularity.

Again, this approach is limited to the methods of a certain object and does not provide any extra knowledge on how to use the API.

There are two types of relevant information available in a callgraph [35]. The relation between the functionality of two functions, and that a certain degree of layering exists. The FRAN algorithm consists of two phases. The first is based on a query function, it provides a set of related functions and then orders the result based on the relevance of each function. Although this might be useful, an important sequence of method calls might not be structurally related and still be important to the developer in order to interact correctly with the API. There is also the FRIAR algorithm developed by the same authors that mines sets of functions that are commonly called together in order to recommend functions related to a particular query function. With this, the developer might get an appropriate sequence of method calls, however, it might not make sense if the code that the programmer wrote is not taken into account, which may result in an inappropriate use of the API.

Another approach is RASCAL [24, 23] which is a tool that recommends a set of task-relevant methods, by trying to predict the next method the programmer will call. To do this, the authors employed a Collaborative Filtering algorithm using the Vector Space Model and compare that approach with another technique called Latent Semantic Indexing. Figure 2.5 shows how to interact with the tool.

These tools are the ones we consider most related to our goals, but none of them has a stepwise instruction recommendation. These tools, as stated before, can be difficult to use properly if the programmer has no knowledge about the API, or its resources are unclear or absent, which will decrease the programmer's productivity since he/she will have to look for examples. As stated in Section 1.2, these examples sometimes hinder the programming task if they mismatch the developer's goal and may lead to an inefficient or incorrect use of the API, consequently decreasing code quality. Nevertheless, developers often rely on examples and try to adapt them to their tasks. However, it is important to show the benefits of example recommendation tools like the ones that follow.

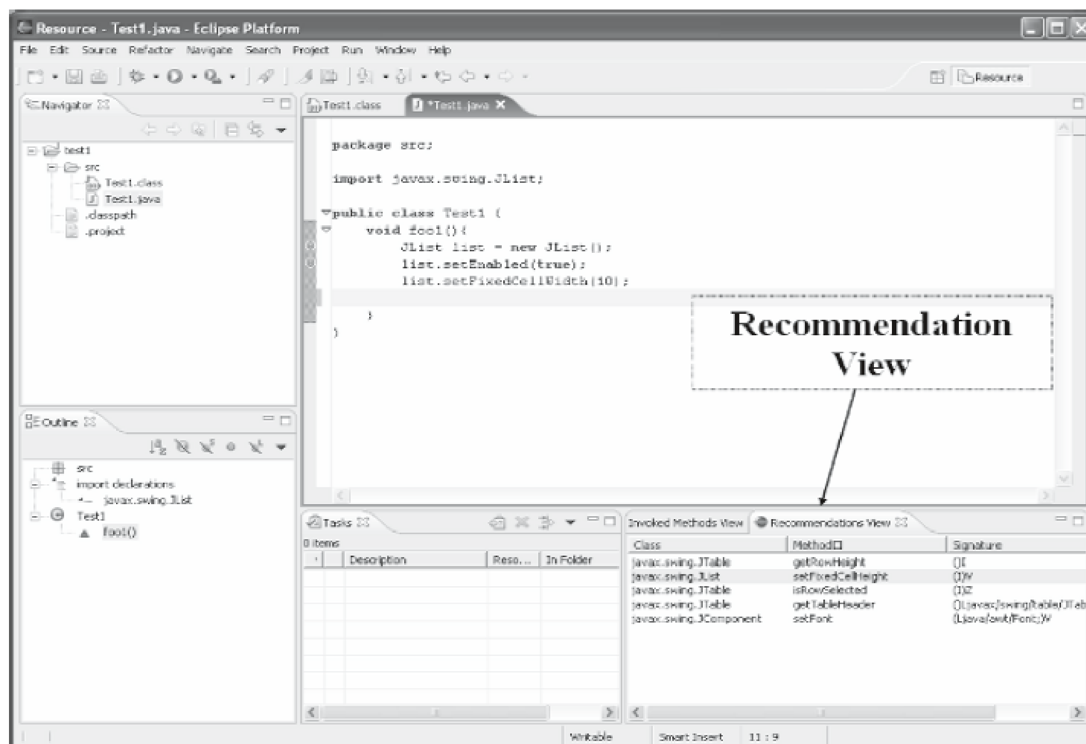


FIGURE 2.5: RASCAL prototype overview [24].

## 2.3 Snippet/Sequence Recommenders

Recently, a tool called SLANG used $n$-gram language models and recurrent neural networks to fill holes in partial programs that use a certain API. Figure 2.6 shows a partial program with holes to be filled with the tool as in Figure 2.7 [31].

Even though our work only focuses on the language models, we intend to achieve a stepwise assistance tool based on a code completion system that adapts and helps the programmer using and discovering the API and not by filling holes in partial programs. This may require the programmer to know where those holes are, which can be a difficult task if there are no apparent compilation or execution errors.

Software developers often search for examples when using an unfamiliar API which can usually be found in its documentation or on the web. XSnippet [34] is a code assistant tool that allows programmers to query repositories for relevant code snippets according to their context without having to explicitly formulate a query. Although this approach abstracts the developer from the query formulation,

```
void exampleMediaRecorder()
throws IOException {
Camera camera = Camera.open();
camera.setDisplayOrientation(...);
? // (H1)
SurfaceHolder holder = getHolder();
holder.addCallback(...);
holder.setType(...);
MediaRecorder rec = new MediaRecorder();
? // (H2)
rec.setAudioSource(...);
rec.setVideoSource(...);
rec.setOutputFormat(...);
? {rec} // (H3)
rec.setOutputFile(...);
rec.setPreviewDisplay(...);
rec.setOrientationHint(...);
rec.prepare();
? {rec} // (H4)
}
```

FIGURE 2.6: Partial program using multiple APIs.

```
void exampleMediaRecorder()
throws IOException {
Camera camera = Camera.open();
camera.setDisplayOrientation(...);
camera.unlock();
SurfaceHolder holder = getHolder();
holder.addCallback(...);
holder.setType(...);
rec = new MediaRecorder();
rec.setCamera(camera);
rec.setAudioSource(...);
rec.setVideoSource(...);
rec.setOutputFormat(...);
rec.setAudioEncoder(1);
rec.setVideoEncoder(3);
rec.setOutputFile(...);
rec.setPreviewDisplay(...);
rec.setOrientationHint(...);
rec.prepare();
rec.start();
}
```

FIGURE 2.7: Holes automatically replaced with SLANG's proposals.

programmers still need to know what to search for which implies some knowledge about the API.

The Strathcona [18, 17] is an example recommendation tool to help programmers locate source code examples by formulating a query based on the code's structural context. The drawback of this approach is the necessity of having some knowledge of where to start writing code before querying an example, which could be obtained by some documentation. Furthermore, if the resources are absent, the tool would be unusable.

MAPO is another tool [41, 40] used to mine API usage patterns to recommend associated code snippets (see Figure 2.8). Their approach differs from ours, since it returns code snippets based on the class and method the programmer is implementing, i.e., the code's actual structure and not on the instructions the programmer has written. Although snippets show interactions with different objects, if they mismatch the developer's goal, it can become an obstacle [33].

Selene [37] is a code recommendation system that uses the editing code as a search query to look for similar program fragments from an example repository. This, again, requires the developer to write some code and have a starting point to make a query. However, this approach has a similarity to ours since it takes into account the editing code, i.e., the code the programmer has written. Although the code's structural context is very important, we argue that the previously written code

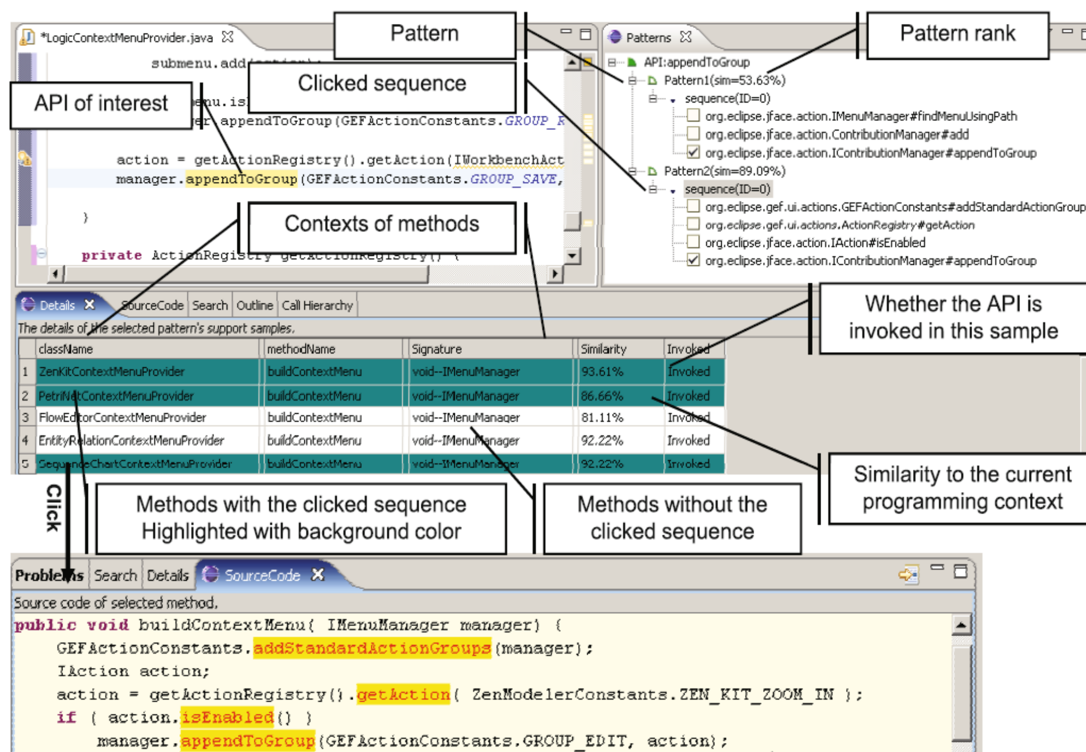


FIGURE 2.8: Overview on the recommender component of the MAPO tool [41].

should also be taken into account to recommend more relevant instructions, since that code is more likely to influence the next instruction the programmer might want than the structural context of the code. Nevertheless, a combination of both structural and local contexts should be a plus.

Although the snippet matching also recommends the next instruction, in the approaches that we analysed, they recommend a set of method calls for the same object, or multiple objects but the developer always needs to have some knowledge about where to start which is a drawback of these approaches.

As stated in Section 1.2, one of the biggest difficulties programmers have to face is the creation of an object that does not own a public constructor. The tools presented in following section were created in order to mitigate this problem.

## 2.4 Recommenders for a desired object type

The Prospector tool [22] helps programmers by returning code snippets to obtain a certain type given another type the programmer already has through a chain of objects and methods calls. That is referred to as a jungloid.

Thummalapenta and Xie [38] developed a tool that allows programmers to formulate queries by providing a source and destination object type. The tool, PARSEWeb, will use this query to suggest chain calls of methods that will return the destination type from the source type. Their approach uses a code search engine to gather code samples and then analyse it statically to return the relevant method sequences. Figure 2.9 presents the interface of PARSEWeb.

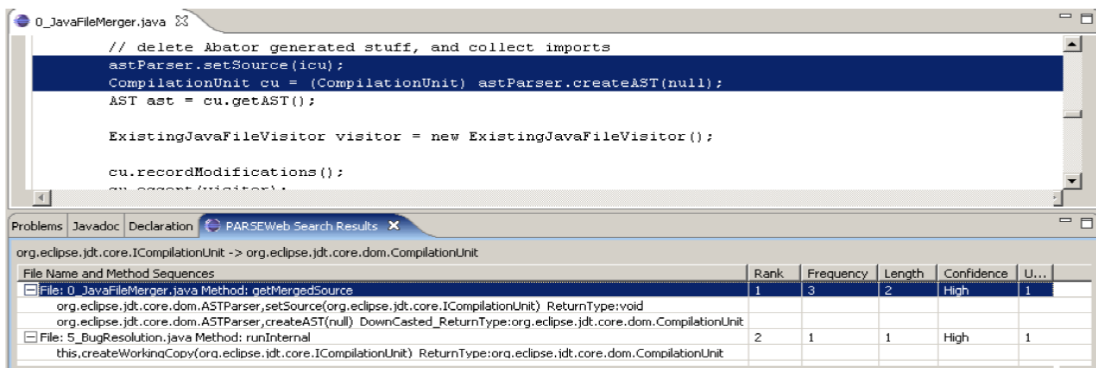


FIGURE 2.9: PARSEWeb tool interface.

Another alternative is RECOS [4], which consists of an object-instantiation and recommendation system, i.e., it composes a chain of method calls that returns a certain type, given an input type. Moreover, this system does not require a repository of sample code to mine snippets nor a source code search engine and it takes into account not only the structural context but the local context as well.

The drawback these tools have in common is the requirement of having some knowledge about the API in order to be able to query the system, i.e., the programmer has to know which object type to input and which one to expect, otherwise the tools are unusable.

## 2.5 Summary

In order to summarize, all the tools were generalized into five different categories according to their purpose (one tool can fit more than one category) as seen in Table 2.1.

| Tools | Snippet/Sequence Recommendation | Method Call Recommendation | Return Desired Object Type | Structural Context Sensitive | Local Context Sensitive |
|---|---|---|---|---|---|
| Code Recommenders | x | x | x | x | x |
| Better Code Completion | | x | | | |
| Corpus-based n-gram,models suggestion engine | | x | | | |
| FRAN algorithm | x | | | | x |
| FRIAR algorithm | x | | | | x |
| RASCAL | | x | | | x |
| SLANG | | x | | | x |
| XSnippet | x | x | | | x |
| Strathcona | x | | | | x |
| MAPO | x | | | x | |
| Selene | x | | | | x |
| Prospector | | | x | | |
| PARSEWeb | | | x | | x |
| RECOS | x | | x | x | x |

TABLE 2.1: Tool categorization table.

The first category, "Snippet/Sequence Recommendation" is assigned to the tools that recommend examples or sequence of instructions with one or multiple objects. The tools categorized with "Method Call Recommendation" are the one's that filter the available methods for a given object by ranking the most relevant. The third category "Return Desired Object Type" is assigned to the tools that, given an expected output type, show the chain of method calls that return that expected type. The last two categories are used on the tools that take into account the code's structural or local context. For our approach it is important to emphasize the fact that most of the tools are local context sensitive. We believe that this has to do with their goal since they require a query mechanism or the suggestions are based on previous written code they need to have this level of sensitivity in order to suggest in the best way possible. However, structural context sensitivity can also be important since some suggestions might be unusable inside a certain class and that can be taken into account in order to filter the proposed suggestions.

# Chapter 3

# API Sentence Models

Typically, APIs have regular usage patterns that describe valid sequences of API calls. $N$-gram language models provide a simple and efficient way of capturing these regularities. However, these instructions need to be abstracted in order to have a simple representation in the model. SLAMC [25] is a statistical semantic language model for source code, where Nguyen et al. introduce semantic information into language models. This additional information helps to predict the next token using the global context of source files. Tu et al. [39], however, argue that code tokenization is enough for $n$-gram language models. On the one hand this supports our decision regarding the abstraction of the API calls. On the other hand, this simple representation causes other limitations (see Section 3.5).

## 3.1 Tokens

Within the realm of an API, we refer to *token* as a possible code instruction that involves a public API type and a public operation therein, considering both static operations and constructors as operations too. Tokens are semantic "chunks" that abstract the representation of code instructions. Currently, method overloading is not handled, and therefore, there is no distinction amongst methods of the same class whose name is equal but their parameters differ. The same applies to

alternative public constructors of the same class. A token is uniquely identifiable by a tuple formed by an API type and an operation of that type. We consider the API *vocabulary* to be formed by a set containing all the possible tokens of the API[1], denoted by $\mathcal{V}$:

$$\mathcal{V} = t_0, t_1, \ldots, t_n : \forall t \in \mathcal{V}, t = \langle type, operation \rangle$$

The extraction process considers three kinds of tokens involving invocations on constructors, static operations, and instance operations. Constructor invocation is in fact a static operation, but it has to have special treatment in Java and we use the keyword "new" to name the operation. Table 3.1 summarizes the kinds of tokens that are considered when extracting sentences.

| **Invocation** | **Token type** | **Token operation** |
|---|---|---|
| Class constructor | owner class | "new" |
| Static operation | owner class | operation name |
| Instance operation | target expression type | operation name |

Table 3.1: Token kinds in API sentences.

Tokens are extracted from assignment statements and expressions. The token is considered as part if its type is part of the API, whereas all other tokens are ignored. The API boundaries are given by the API source code that takes part as input in the extraction process.

## 3.2 Sentences

The projects that are considered for building the language model are mined to extract API sentences that are used to build an API sentence model. An API sentence is a sequence of tokens of $\mathcal{V}$, whose instructions are related. A sentence consists of an observation ($\omega$) and is treated as a vector of tokens from the vocabulary:

[1] A member of a vocabulary is normally referred to as *type*, but such terminology is not adopted here to avoid confusion with the types of the API.

$\omega = t_0, t_1, \ldots, t_n : \forall t \in \omega, t \in \mathcal{V}$

The extracted sentences are mined by parsing every method instruction block found in the source code corpora. For instance, the following method would result in the extraction of the sentence below.

```
void method(...) {
Label label = new Label(...);
String s = ...;
label.setText(s);
}
```

Extracted sentence:

$\omega = (\langle Label, new\rangle, \langle Label, setText\rangle)$

The instantiation of `Label` was considered as an API token, the `String` assignment was ignored, and the operation `Label.setText` was also identified as a token.

## 3.3 Extraction Process

The degree of robustness in the process of extracting API sentences from source code corpora affects the quality and accuracy of the recommendations. Poorly extracted snippets result in highly noisy data, which in turn will lead to less accurate language models. Therefore, the extraction process is handled with special care using different tactics for sentence extraction. We developed a sentence extractor, which is in essence a parser of Java source files based on the infrastructure of Eclipse's Java Development Tools.

### 3.3.1 Composite expressions

Instructions may contain composite expressions that involve API tokens. Given that our goal is to capture sentences in terms of their semantics, rather than how

they are expressed syntactically, we treat composite expressions so that the extracted sentences are decomposed as if having separate instructions. For instance, the two following code snippets result in the same extracted sentence.

```
Composite composite = new Composite(...);
composite.setLayout(new GridLayout(...));
```

```
Composite composite = new Composite(...);
GridLayout layout = new GridLayout(...);
composite.setLayout(layout);
```

Extracted sentence:

$\omega = (\langle Composite, new\rangle, \langle GridLayout, new\rangle, \langle Composite, setLayout\rangle)$

The token order reflects the execution order, namely, the instantiation of `GridLayout` occurs prior to the invocation of `setLayout(...)`.

Analogously, when having chained invocations, the expression is separated into different instructions. The following code snippet illustrates this case, where the two cases also result in the same extracted sentence.

```
new Label(...).getParent().getParent();
```

```
Label label = new Label(...);
Composite parent = label.getParent();
Composite parentParent = parent.getParent();
```

Extracted sentence:

$\omega = (\langle Label, new\rangle, \langle Label, getParent\rangle, \langle Composite, getParent\rangle)$

As with the previous case, the token order reflects the execution order.

### 3.3.2 Token dependencies

API calls are normally interleaved with other instructions that do not relate to the API, while unrelated API sentences might also be interleaved. The criteria for relating the tokens that form a sentence is based on dependency graphs between instructions. An instruction $a$ is considered to depend on another instruction $b$ in the following cases:

```
input  : block : ∀t ∈ block, t ∈ V_Ω
         (a list of block instructions)
output : out ⊆ S_V
         (a set of unrelated sentences)

out ← ∅
foreach t : block do
    sv ← s ∈ out : variables(s) ∩ variables(i) ≠ ∅
    if sv.isEmpty() then
        sentences.add(new Sentence(t))
    end
    else if sv.size() = 1 then
        sv.get(0).append(t)
    end
    else
        out.removeAll(sv)
        s ← merge(sv)
        s.append(t)
        out.add(s)
    end
end
```

**Algorithm 1:** Algorithm to compute a set of unrelated sentences contained in an instruction block.

1. $b$ is an assignment and $a$ uses the assigned variable;

2. $a$ is an invocation and $b$ is used in its arguments.

Algorithm 1 describes in pseudo-code how the instruction blocks are processed with respect to token dependencies. Each instruction block will result in a set of sentences (possibly empty, if no API tokens are found therein). We start with an empty set of sentences. For every instruction of the block, the variables used therein are obtained and the sentences where at least one of those variables is used is computed ($sv$). If there are no matching sentences in the set, then a new sentence containing the instruction is created and added to the set. If there is only one matching sentence, then the instruction is appended to the end of that sentence. Finally, if there are more than one matching sentence, these are considered related and merged into a new sentence, using the instruction order. The merged sentences are removed from the set and the new sentence is added.

Consider the following example, where we have two unrelated sentences in a block. The instructions that form the two `Composite` objects are unrelated because the variables that are involved do not overlap, and hence, two sentences are extracted from the block.

```
Composite composite1 = new Composite(...);
composite1.setLayout(new RowLayout());
Composite composite2 = new Composite(...);
composite2.setLayout(new FillLayout());
Button button = new Button(composite1, ...);
Text text = new Text(composite2, ...);
button.setText(...);
text.setText(...);
```

Extracted sentences:

$\omega_1 = (\langle Composite, new\rangle, \langle RowLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Button, new\rangle, \langle Button, setText\rangle)$

$\omega_2 = (\langle Composite, new\rangle, \langle FillLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Text, new\rangle, \langle Text, setText\rangle)$

### 3.3.3 Selections and loops

Selections (`if-else` blocks) are treated so that a sentence is extracted for each possible branch, using a similar strategy to the MAPO tool [41]. Once again, the extracted sentences follow the execution order, but in this case, with alternative execution sequences. Loops are treated in the same way as if they were conditionals, preserving the possible execution sequences, but ignoring repetition.

Consider the following code snippet with an `if-else` block to illustrate this case. Given that there are two possible executions of the block, two sentences are extracted.

```
if(composite.isVisible()){
composite.setFocus();
}
else {
composite.setVisible(...);
composite.redraw();
}
composite.layout();
```

Extracted sentences:

$\omega_1 = (\langle Composite, isVisible\rangle, \langle Composite, setFocus\rangle, \langle Composite, layout\rangle)$

$\omega_2 = (\langle Composite, isVisible\rangle, \langle Composite, setVisible\rangle, \langle Composite, redraw\rangle, \langle Composite, layout\rangle)$

### 3.3.4 Sentence sample vocabulary

Consider $\mathcal{S}_\mathcal{V}$ as the set of all possible sentences using vocabulary $\mathcal{V}$, i.e. the sample space. The set of extracted sentences is denoted by $\Omega$, consisting of a sample of observations that is a subset of the possible sentences ($\Omega \subseteq \mathcal{S}_\mathcal{V}$).

$\Omega = (\omega_0, \omega_1, \ldots, \omega_n) : \forall \omega \in \Omega, \omega \in \mathcal{S}_\mathcal{V}$

Given that the sentences of the sample may not use every token of the vocabulary, the vocabulary of the extracted sentences $\mathcal{V}_\Omega$ is a subset of the API vocabulary ($\mathcal{V}_\Omega \subseteq \mathcal{V}$). $\mathcal{V}_\Omega$ is used to build the language model, and the accuracy of the latter depends on the quality of the former.

## 3.4 Definition of API Sentence Models

Formally, an API Sentence model is an $n$-gram language model built from a set of extracted API sentences $\Omega$ from the source code corpora, which are composed by tokens from a subset of the API's vocabulary $\mathcal{V}_\Omega$. Consider the examples on Figures 3.2 that illustrate how a sample of observations $\Omega$ containing the extracted sentences $\omega 0, \omega 1, \omega 2$ would be organized on an API Sentence Model with order $n = 2$ with their respective probabilities.

## 3.5 Limitations

As stated before, this simple representation of the instructions and respective sentences creates a number of limitations namely:

| **Tokens** | $\langle Composite, new\rangle$ | $\langle RowLayout, new\rangle$ | $\langle Composite, setLaytout\rangle$ | $\langle Button, new\rangle$ | $\langle Button, setText\rangle$ | $\langle FillLayout, new\rangle$ | $\langle Text, new\rangle$ | $\langle Text, setText\rangle$ |
|---:|---|---|---|---|---|---|---|---|
| $\langle Composite, new\rangle$ | | $^1/_3$ | | | | $^2/_3$ | | |
| $\langle RowLayout, new\rangle$ | | | $^1/_1$ | | | | | |
| $\langle Composite, setLaytout\rangle$ | | | | $^2/_3$ | | | $^1/_3$ | |
| $\langle Button, new\rangle$ | | | | | $^2/_2$ | | | |
| $\langle Button, setText\rangle$ | | | | | | | | |
| $\langle FillLayout, new\rangle$ | | | $^2/_2$ | | | | | |
| $\langle Text, new\rangle$ | | | | | | | | $^1/_1$ |
| $\langle Text, setText\rangle$ | | | | | | | | |

FIGURE 3.1: API Sentence model based on $n$-gram language models with $n = 2$.

$$\omega_0 = (\langle Composite, new\rangle, \langle RowLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Button, new\rangle, \langle Button, setText\rangle)$$
$$\omega_1 = (\langle Composite, new\rangle, \langle FillLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Text, new\rangle, \langle Text, setText\rangle)$$
$$\omega_2 = (\langle Composite, new\rangle, \langle FillLayout, new\rangle, \langle Composite, setLayout\rangle, \langle Button, new\rangle, \langle Button, setText\rangle)$$

FIGURE 3.2: Sample of observations $\Omega = (\omega_0, \omega_1, \omega_2)$.

**Tokenization ignoring overloading.** Our tokenization of API sentences does not take into account method and constructor overloading. Our intuition was that abstracting overloaded methods and constructors into a single token would not have a significant impact on the results. However, we are aware that making the distinction in certain cases could yield value.

**Long-distance relations among tokens.** The ability to recommend useful API tokens is constrained by a relatively small number of previous of tokens (e.g., previous 4 tokens using a 5-gram model). This limitation is inherent to $n$-gram models [7]. The downside is that some recommendations could benefit from tokens written far back in the context, i.e. distance greater that the order of the $n$-gram.

# Chapter 4

# Measuring API Perplexity

To evaluate the performance and nature of $n$-gram language models, we produced a model for each API and measured its perplexity. This measurement is one of the most common methods for evaluating language models. Equation 4.1 formally describes how the perplexity is computed, where $p(T)$ is the probability assigned to a test set $T$ with a length $W_T$ (number of tokens).

$$P(T) = 2^{-\frac{1}{W_T} log_2 p(T)} \tag{4.1}$$

This technique is an intrinsic evaluation approach since it provides information about the models' nature and behaviour. Moreover, it provides information such as the average number of choices for each word [14], which is very important since we want the developer to stay in the right path but also allow for some API exploration. Furthermore, it allows us to determine which $N$ value and which smoothing technique best fits each API, consequently producing the most accurate model.

## 4.1 Setup

Allamanis and Sutton [3] collected a large corpus from GitHub, containing thousands of Java projects that were filtered according to the Github's social fork system, in order to assume an adequate code quality level, since low quality projects are rarely forked. Posteriorly, we made a second filter in order to obtain only the projects that contain references to the APIs that we wanted to evaluate. In order to accomplish this task, we searched for import statements and fully qualified names that contain the APIs' root package in all the projects' source files. The ones that do not contain a reference to the desired API, do not use it, therefore we can ignore them, since they would not produce any API sentences and would increase the time required to extract the sentences. After determining which projects contain these references, we used the method described in Chapter 3 to extract the API sentences. To perform a 5-fold cross-validation scheme [21], the set with the extracted API sentences was divided into 5 subsets (containing both training and test sets). Note that sometimes each subset may contain more or less tokens, since the size of the sentences may vary from one subset to another. After this process, the SRILM toolkit was used to build the $n$-gram language models from each subset of the extracted API sentences and apply the different smoothing techniques available, as well as to evaluate the models' performance.

In order to evaluate the performance of language models, we have chosen APIs addressing both similar and different domains, with small and large corpus sizes, and finally, with different vocabulary sizes in order to compare results and fully assess the usage of $n$-gram language models on capturing usage patterns from different APIs. Table 4.1 shows the APIs that were evaluated, the number of projects containing at least one reference to the API's root package, the total number of sentences extracted from those projects, and finally, the size of the API's vocabulary, which will help us to draw some conclusions from the results obtained.

The performance evaluation for each API was made with both the Witten-Bell and Kneser-Ney smoothing methods, as well as without any smoothing method,

| **API** | **Projects** | **Sentences** | **Vocabulary** |
|---|---:|---:|---:|
| SWT | 501 | 105,718 | 4,085 |
| Swing | 2,294 | 160,961 | 8,105 |
| JFreeChart | 248 | 22,948 | 3,788 |
| JSoup | 115 | 1,508 | 370 |
| JDBC Driver | 559 | 1,666 | 176 |
| Jackson-core | 71 | 9,306 | 253 |

Table 4.1: APIs under analysis: number of client projects, number of extracted sentences, and vocabulary size.

in order to evaluate how differently the model performs[1].

There are some words in the test set that are not found in training sets, which are called Out Of Vocabulary (OOV) words. When the perplexity values are computed, these words are not taken into account. For this we need to use a special token, `<unk>`, representing the OOV words.

In our case, the interpolate option is used only when using a smoothing method, Witten-Bell and Kneser-Ney. This option interpolates the $n$-order estimates with the lower order ones, which, in general, are more accurate.

After computing the results, we concluded that the interpolation option alone, produces better results in 80% of the cases. The baseline was the perplexity without any smoothing methods, compared with the Witten-Bell and Kneser-Ney smoothing methods with the interpolation option. Also, we do not show the results for unigrams since they do not support the existence of an history when estimating the probabilities of the next token. The results presented are the averages of the 5-fold cross validation scheme.

[1] In SRILM, for each of the smoothing methods, there are 4 different options (no options, `-unk`, `-interpolate` and `-interpolate -unk`).

## 4.2 SWT API

SWT[2] is a cross-platform library to develop graphical user interfaces. There are several examples on the web on how to interact with its API. Even though the interface highly depends on the application and might differ a lot from one another, some of the method calls must always be performed in order to correctly build and customize the user interface.

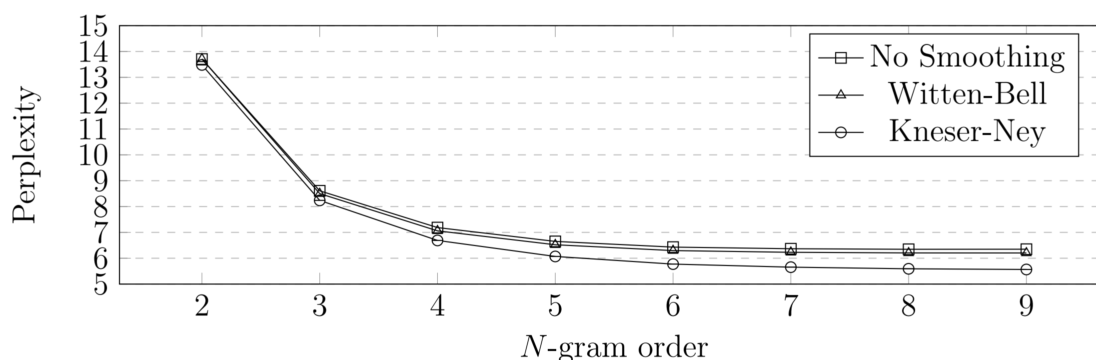


FIGURE 4.1: Perplexity values for the SWT API.

Figure 4.1 shows that as we increase the value of $n$, the perplexity decreases and stabilizes at $n = 6$. Thereon it decreases less than 0.2 in total. We can also see that the Kneser-Ney smoothing performs slightly better than all the others. Although the SWT library is used to produce graphical user interfaces, it shows a relatively low perplexity value, with an average of 6 different recommendations for each history of length 5 or higher.

## 4.3 Swing API

Similarly to the SWT, the Swing library is also used to create graphical user interfaces, however it produces a more similar appearance independently of the operating system in use. Again, the graphical user interfaces are very customizable, and there are several different widgets and components the developer can use, which produces a larger number of recommendations for the same history.

[2]Standard Widget Toolkit: Graphical User Interface library developed and used by Eclipse — `www.eclipse.org/swt`

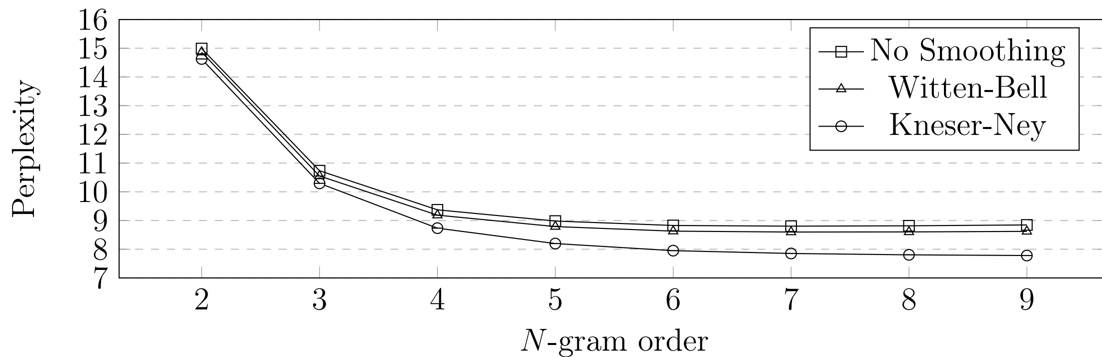


FIGURE 4.2: Perplexity values for the Swing API.

Regarding the objective, it is reasonable to assume that the Swing API is similar to the SWT API. This API may be used for the same domain as the previous one, and although the perplexity values are somehow similar (see Figure 4.2), they are slightly higher. Nevertheless, the values stabilize at $n$=6, and the Kneser-Ney smoothing is also the best method as with the SWT API.

## 4.4 JFreeChart API

The JFreeChart library[3] allows developers to create and customize different kinds of charts in a graphical application. Although the library allows programmers to customize charts, it is not as flexible as the previous ones.

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

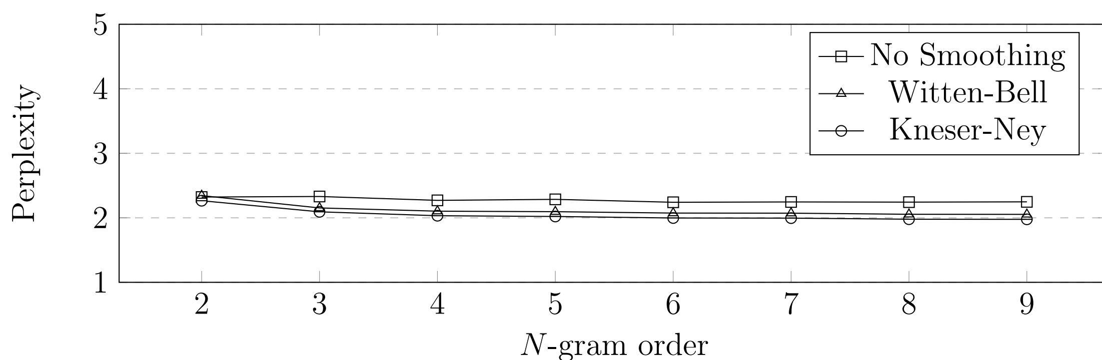


FIGURE 4.3: Perplexity values for the JFreeChart API.

As expected, and since the JFreeChart API is much more regular and has a lot less different paths to follow on each word, the perplexity values are much lower than the ones obtained on the previous APIs. Figure 4.3 shows that the perplexity values range from 1.97 to 2.34. Although the Kneser-Ney smoothing still achieves the best results, in this case the difference is not significant in comparison with the Witten-Bell method.

## 4.5 JSoup

The JSoup[4] library is used to fetch and parse HTML, manipulate and extract data from it, and clean it in order to prevent XSS attacks. It was expected that the JSoup API would have a relatively low perplexity in comparison with the APIs for creating graphical user interfaces.

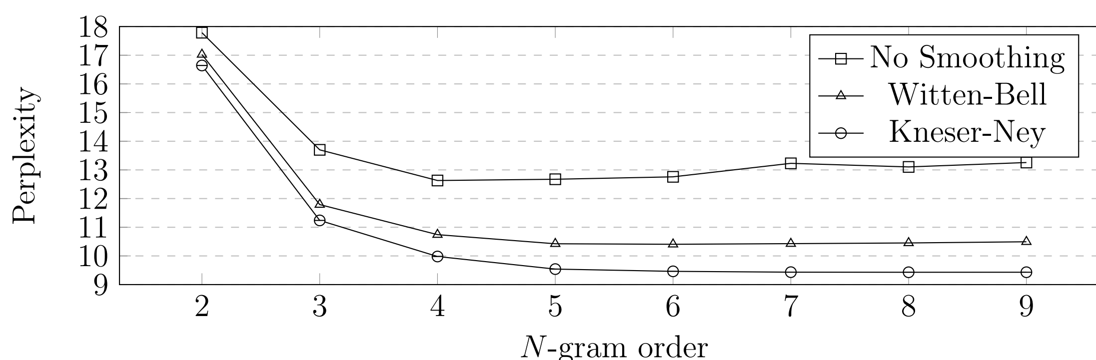


FIGURE 4.4: Perplexity values for the JSoup API.

The high perplexity values (see Figure 4.4) have to do with the fact that it does not require a very predictable way to be interacted with. We can also observe that without applying a smoothing technique, with $n \geq 5$, the perplexity values tend to increase. This is where the smoothing techniques take advantage and produce more accurate models, as we can see by the perplexity values achieved with the Witten-Bell and Kneser-Ney methods.

[4]http://jsoup.org/

## 4.6 JDBC Driver for MySQL

MySQL[5] provides drivers in several programming languages to make connections and execute statements in a database, enabling developers to integrate their applications with the MySQL databases.

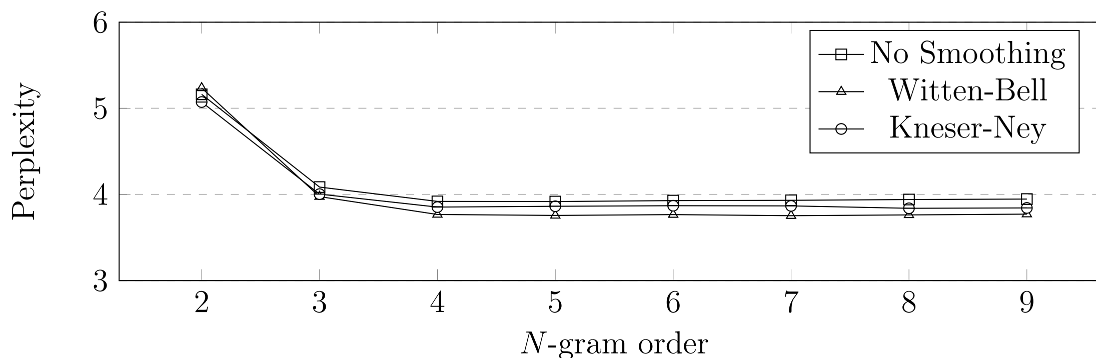


FIGURE 4.5: Perplexity values for the JDBC Driver API.

Our intuition is that the low perplexity values (see Figure 4.5) are explained with the relatively simple usage of the API, since in most of the cases it basically requires to open a connection (with more or less properties), execute an SQL statement, and iterate over the results.

## 4.7 Jackson-core

This library[6] is used to process JSON data format. Its core contains a parser and an abstract generator used by its data processor.

Again, a low perplexity (see Figure 4.6) was expected due to its simple usage, that mainly requires a factory or a mapper, and a parser to read and a generator to write. In this API the smoothing methods did not improve accuracy significantly.

[5] http://dev.mysql.com/downloads/connector/j/
[6] http://wiki.fasterxml.com/JacksonHome

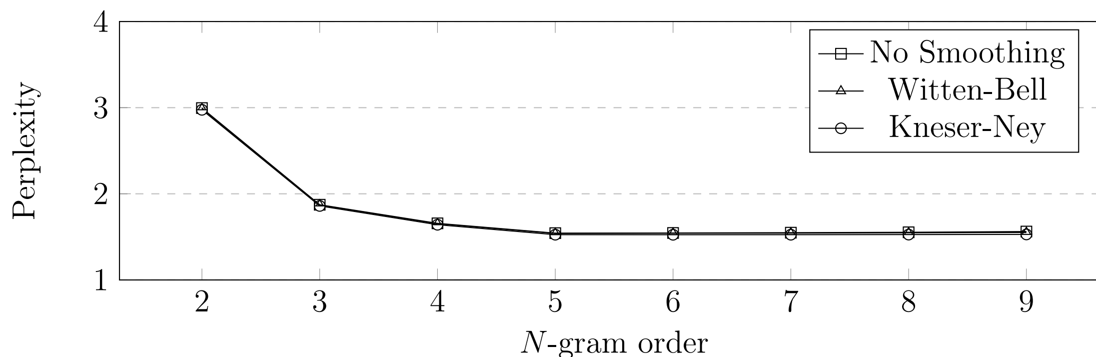


FIGURE 4.6: Perplexity values for the Jackson-core API.

## 4.8 Discussion

We now compare and discuss the obtained results in order to draw some conclusions regarding the language models and their applicability in the domain of the APIs. Since in most of the cases the Kneser-Ney smoothing technique produced better perplexity values, Figure 4.7 compares those values across all the analyzed APIs.

Regarding the perplexity, it is reasonable to conclude that APIs have a similar nature, i.e., as we increase the value of $n$, the perplexity decreases and stabilizes

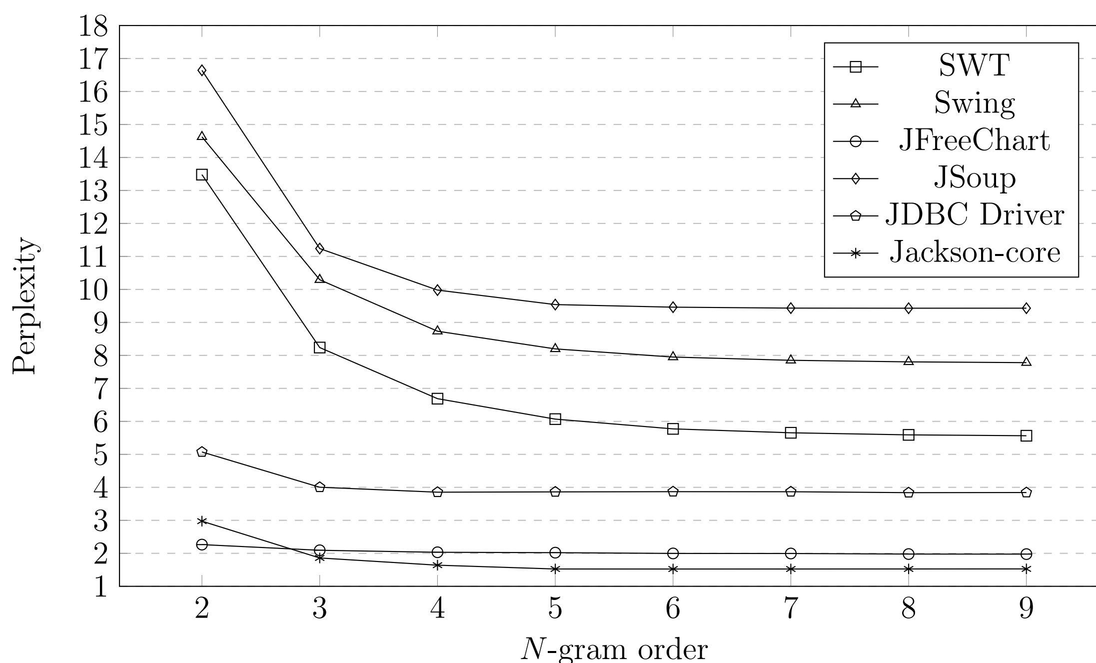


FIGURE 4.7: Perplexity values comparison for all the APIs.

at $n$=6 in most cases. Even though these APIs may have very different purposes, this perplexity decrease was expected since as history increases in the $n$-gram models, there are less options for each history, but with more accurate estimates for that history.

Figure 4.8 presents the relation between the API's vocabulary size and the obtained perplexity values. Even though there are some outliers in this chart, the

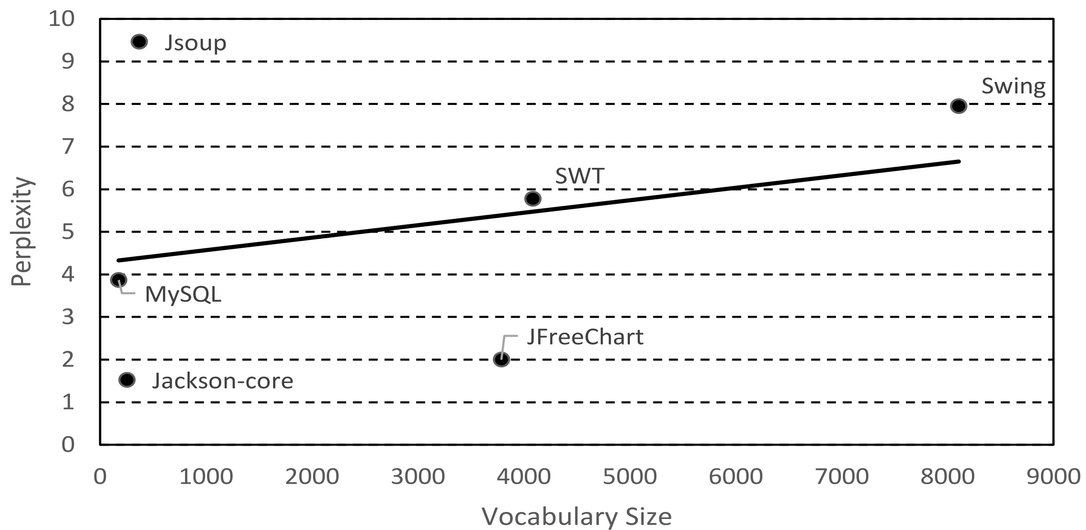


FIGURE 4.8: Relation between vocabulary size and perplexity.

trendline suggests that as the size of the API's vocabulary increases, in general, the perplexity tends to increase as well. These results are not very surprising, because as the size of the API increases there may be more possible usage patterns. Nevertheless, this is not true in all the cases: the JFreeChart API, which has a vocabulary size similar to the SWT API, achieves a lower perplexity. This has to do with the nature of the API, which has a relatively simple usage, and with the fact that the corpora may not cover a considerable part of the API.

$N$-gram language models achieve a perplexity that goes from 50 to 1000 on English text. Hindle et al. [16] obtained perplexity values that vary from approximately 3 to 32 for Java source code in general. These results are very encouraging because they show that API usage is far more regular than source code in general, which motivated us to create a tool to help programmers find the next token they need to correctly interact with the API.

One of the main threats to the use of $n$-gram language models is the size of the training corpus, which may not contain all the possible sentences to populate the model. Although the smoothing techniques help mitigate this question, the model will not be accurate when predicting sentences that do not occur in the training corpus. This is one of the most important factors that impact the generation of more accurate language models, and consequently better recommendations to the programmer.

It is a fact that the API's vocabulary is not as extensive as for example the English vocabulary. Even though the source code corpora might not include all the possible tokens, we argue that it contains enough to create models that will help introduce the programmer to the API and still allow for some extra exploration.

Programmers often look for examples to learn an unknown API. These usually represent possible usage patterns and are often required to correctly interact with it [2]. When the resources are ill-formed, programmers might not correctly use the proper constructs of these patterns in their code, and even if the code compiles, and apparently does what it is supposed to, internally it might work incorrectly, thus being more likely to produce errors in the future. There is no viable way of manually filtering out these patterns from the source code corpora, but an adequate dimension of the corpora helps to mitigate this question. This point is related to the previous one in the sense that in order to allow for some exploration, the uncommon patterns must be taken into account. Just because they are not common, does not mean they are not correct. Smoothing techniques produce a more uniform probability distribution in language models, i.e., they will increase the probability of rare tokens and decrease the probability of the very common ones.

# Chapter 5

# Recommendations Evaluation

The intrinsic evaluation described in Chapter 4 only provides information about the model's nature. Even though the relatively low perplexity values hint that the models will achieve good coverage and accuracy values, a deeper and extrinsic evaluation is required in order to fully assess this question. To perform this task we evaluate the coverage and average proposal hit ranks of the recommendations provided by the model, for the top 20 proposals. The models evaluated are the same of the previous chapter in order to produce a full evaluation over the models' performance and maintain consistency throughout the whole process. The following sections describe the evaluation method, present the most relevant results (see the Appendices for more information) and discuss the results obtained.

## 5.1 Method

In order to extrinsically evaluate the models, we have built a system[1] that allows to import the models produced by the SRILM tool. To proceed with the necessary evaluation, the models are loaded into the system. Recall that a 6-gram model requires a 5-gram model to be loaded which in turn requires a 4-gram model and

[1]https://github.com/andre-santos-pt/apista

so on. We do not load the 1-gram models since it is not possible to query them with a history. After this, the test set is loaded into the system.

To evaluate the proposals generated by the models, the tokens of each sentence in the test set are iteratively accumulated and used as context to query the model with the number of tokens required by the $n$-gram model. For example, a 3-gram model would require two tokens to be used as context (history), where the third would be the recommendation to be considered. However, if the size of the context is smaller than the size required by the model, we backoff to the correct $n$-order model, in order to be able to query it. For example, assuming that we are evaluating 6-gram models, this would require five tokens to be used as context. Since the tokens are iteratively accumulated, there are situations where the context does not have the correct size to query the model. To overcome this issue, higher order models can be filled with multiple copies of "start sentence" tokens (also represented as <s>) which would allow the model to be queried with a smaller context. However, the models do not contain these tokens and instead, a backoff mechanism is used to overcome this issue similarly to the SRILM tool. If the token in the sentences that follows the context is found among the predictions generated by the model, its proposal index is analysed in order to understand how good the prediction was. On the other hand, if the token that follows the context is not found in the predictions it may be due to multiple reasons: (a) the token did not appear in the training set, which explains why the model would not be able to recommend it; (b) the token did appear in the training set, but the context that precedes it is not the same; (c) the token was recommended by the model, but not in the top 20 proposals.

The following section provides information about:

- The overall coverage of the proposals, that show the percentage of recommended tokens that the model was able to propose correctly in the top 20 ranks.

- The average hit index allows to understand the quality of the recommendations generated by the model by providing an average of the index where the correctly recommended token was found in the proposals.

## 5.2 Results

Since the perplexity values presented in 4 tend to stabilize at $n = 6$, in order to simplify and show only the most relevant results, this section only provides information for $n$-grams with $2 \leq n \leq 6$. In order to reduce the amount of charts presented, and since all the APIs have a similar behaviour, one should refer to Appendix A for more details regarding other APIs. Nevertheless, there may be references to those APIs when it is relevant. The results presented are the average of a 5-fold cross validation scheme.

Figures 5.1 and 5.2 present the overall coverage of expected tokens for the SWT API for both Kneser-Ney and Witten-Bell smoothing techniques for the different hit ranks and values of $n$.

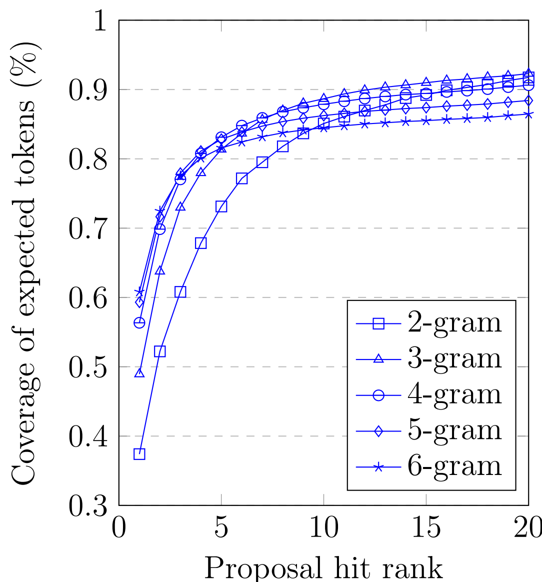


FIGURE 5.1: $N$-gram overall coverage for the SWT API with Kneser-Ney smoothing.

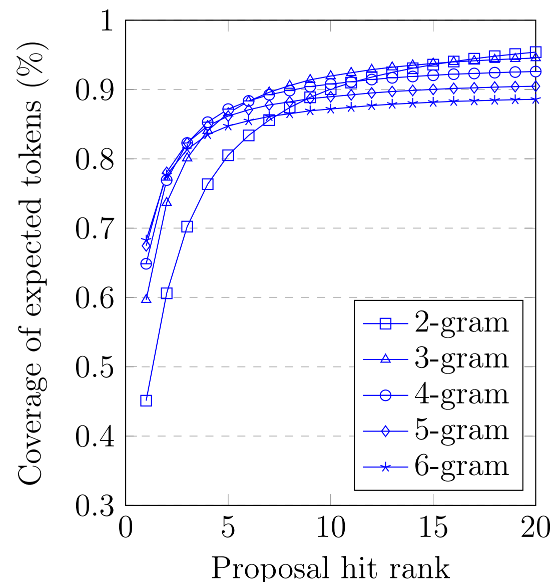


FIGURE 5.2: $N$-gram overall coverage for the SWT API with Witten-Bell smoothing.

The first hit rank shows a relatively low coverage that ranges from approximately 37% (Kneser-Ney) to 45% (Witten-Bell) of correct hits with $n = 2$. The low coverage obtained with $n = 2$ may be explained by the fact that some of the proposals may no be found in the top 20 ranks, since this $n$ order has the highest amount of proposals. As the order of $n$ increases, the model is much more accurate recommending, which results on an increase of the overall coverage, achieving approximately 68% in the first rank.

If we look at the top-5 hit ranks, the coverage increases and ranges from approximately 73% to 87%, since it is possible to gather more recommendations. As the number of proposals increases we can achieve values that range from approximately 86% to 95% of correct recommendations in the top 20 hit ranks. As a side note, it is important to emphasize that although as the order of $n$ increases the model is more accurate in the first ranks, it has a lower overall coverage since the model is much more constrained, thus not being able to generate tokens for long histories. Moreover, the Witten-Bell smoothing provides a higher coverage in most cases.

This evaluation allows us to draw some conclusions regarding the coverage of the recommendations on the top-20 ranks of the proposals for each value of $n$. Clearly, the best model would recommend the next token in the first rank, however this is hard to achieve, since there are multiple valid choices that may appear for the same context. Therefore, Figures 5.3 and 5.4 show more detailed information regarding the accuracy of the recommendations where the predicted token is found (tokens that were not found are discarded, since they do not have an hit rank). The median value is represented in the middle of the box with the respective upper and lower quartiles and the minimum and maximum values in the whiskers.

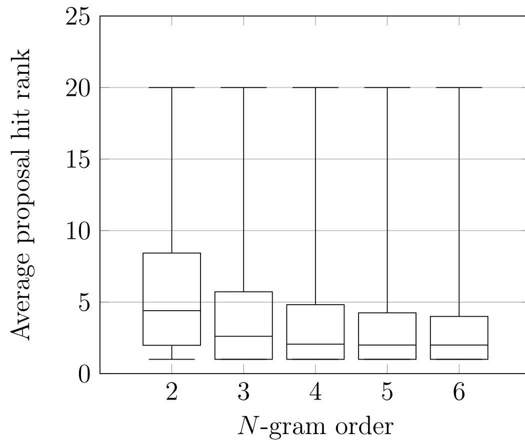


FIGURE 5.3: Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the SWT API.

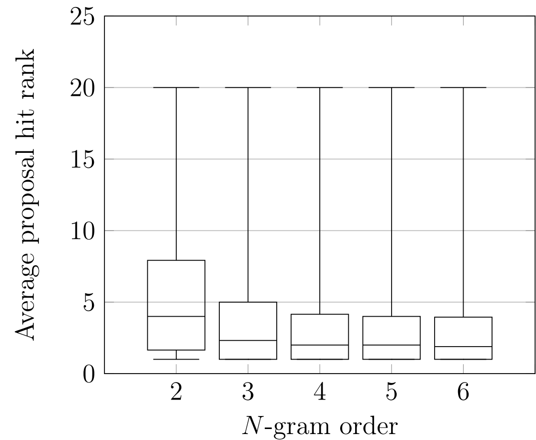


FIGURE 5.4: Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the SWT API.

Again, it is possible to observe that as the order of $n$ increases, the model is much more accurate, thus being more likely to recommend the next token in the first hit ranks. With $n = 2$ the median value ranges from 4 to 4.4 for the Kneser-Ney and Witten-Bell smoothing techniques. With $n = 3$ the median achieves a value of 2.32 and decreases until 1.89 ($n = 6$) with the Witten-Bell smoothing. Although the median value (and the quartiles) decreases as the order of $n$ increases resulting in more accurate recommendations, it is important to note that the overall coverage decreases. We argue that there should be a balance between the coverage and precision in order to keep the programmer in the right path while maintaining good quality recommendations.

Nevertheless these values are good and very encouraging since that a programmer requesting a recommendation, would most likely have a correct proposal in the top 20 ranks, thus not being required to browse for the correct instruction from the whole documentation of an unknown API.

## 5.3 Discussion

In this section we discuss the values obtained for the overall coverage as well as the average hit index and correlate them with the perplexity values obtained in

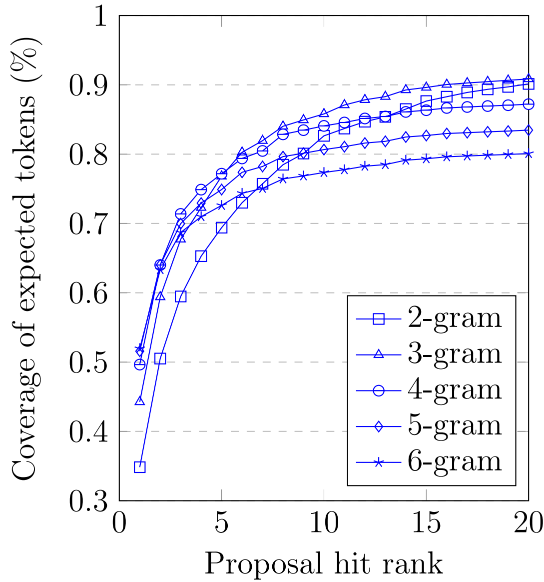


FIGURE 5.5: $N$-gram overall coverage for the Swing API with Kneser-Ney smoothing.

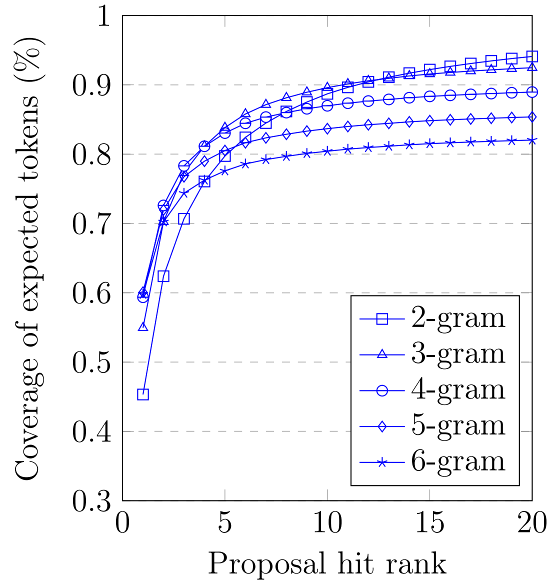


FIGURE 5.6: $N$-gram overall coverage for the Swing API with Witten-Bell smoothing.

Chapter 4. As stated before, the perplexity measure provides information about the average number of choices for each token, which is very important in the context of code recommendations since we want the programmer to be able to access the most relevant instructions in the top ranks.

The SWT API models have an approximate perplexity value of 6 for $n = 6$. Therefore, a maximum of 6 proposals should be enough for most of the recommendations on the SWT API. If we assume that a system is able to recommend 10 instructions in a code completion menu without the need for scrolling, from Figures 5.1 and 5.2 we can see that the increase in the overall coverage of the models is minimal above 10 proposals and still achieves overall coverage values of more than 80%.

As observed in the previous chapter, the nature of the Swing API is similar to the SWT API. When comparing these two APIs, we can see that the coverage (Figures 5.5 and 5.6) is lower for the Swing API, however, the overall coverage results are still acceptable and range from approximately 82% ($n = 6$) to 94%($n = 2$).

This may be due to the fact that the Swing API models have a slightly higher perplexity than the SWT API. Since the Swing API has more possibilities for each token, this may explain the decrease on accuracy and overall coverage in the first

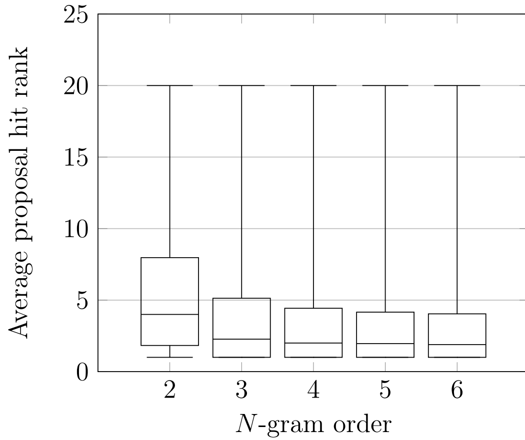


FIGURE 5.7: Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the Swing API.

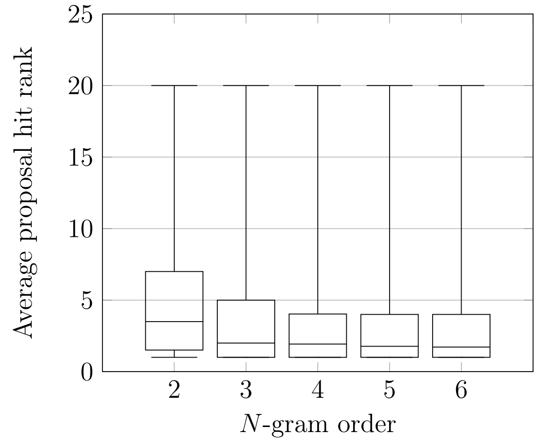


FIGURE 5.8: Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the Swing API.

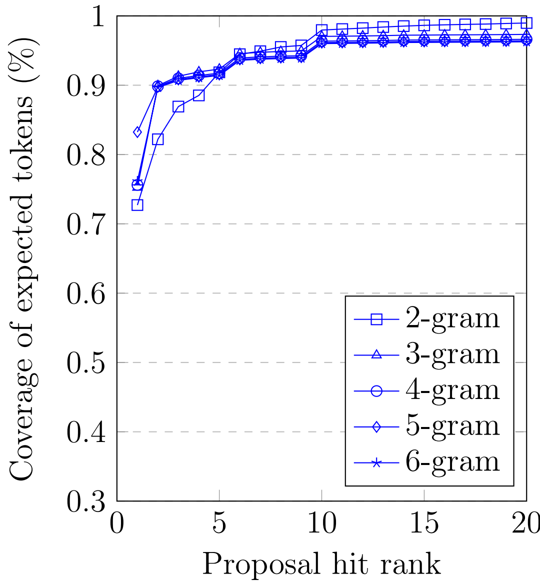


FIGURE 5.9: $N$-gram overall coverage for the JFreeChart API with Kneser-Ney smoothing.

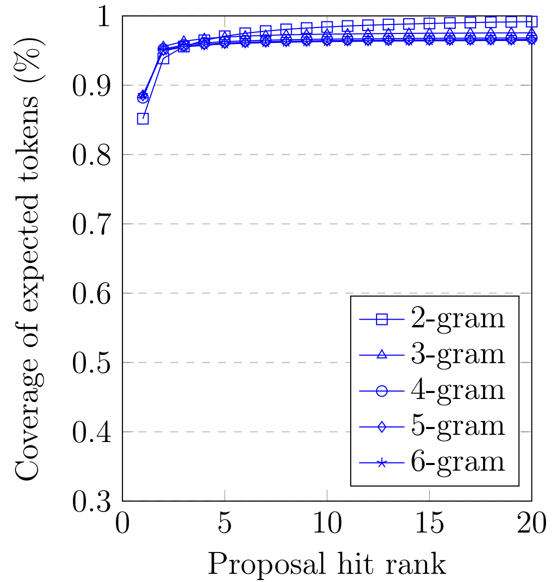


FIGURE 5.10: $N$-gram overall coverage for the JFreeChart API with Witten-Bell smoothing.

ranks, due to the fact that it will be more difficult for the models to correctly predict the next token.

Recalling the perplexity values of the JFreeChart API, that ranges from 1.97 to 2.34, a high overall coverage (up to 99% as observed in Figures 5.9 and 5.10) as well as a low average hit index (Figures 5.11 and 5.12) were expected due to the small number of different choices for each token. The difference between both smoothing techniques is also significant going over 10% with $n = 2$. Although the

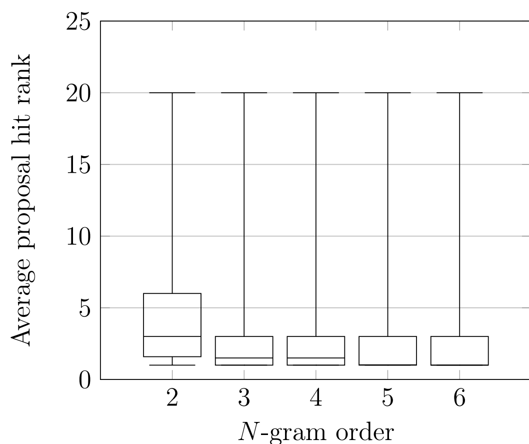


FIGURE 5.11: Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the JFreeChart API.

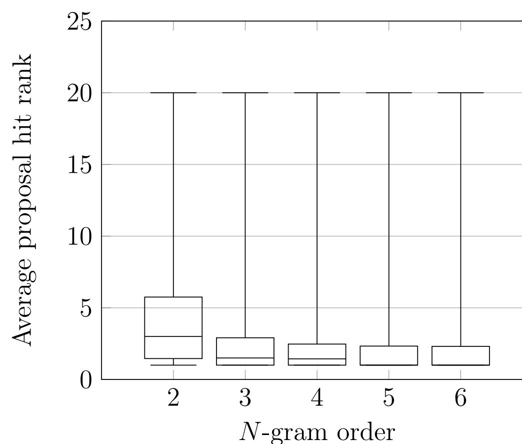


FIGURE 5.12: Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the JFreeChart API.

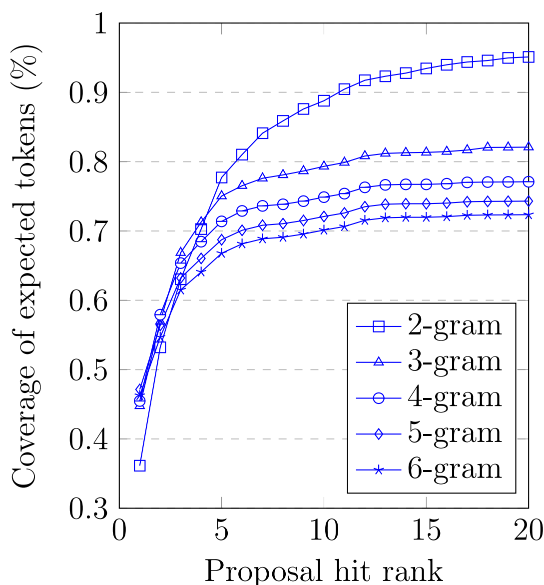


FIGURE 5.13: $N$-gram overall coverage for the JSoup API with Kneser-Ney smoothing.

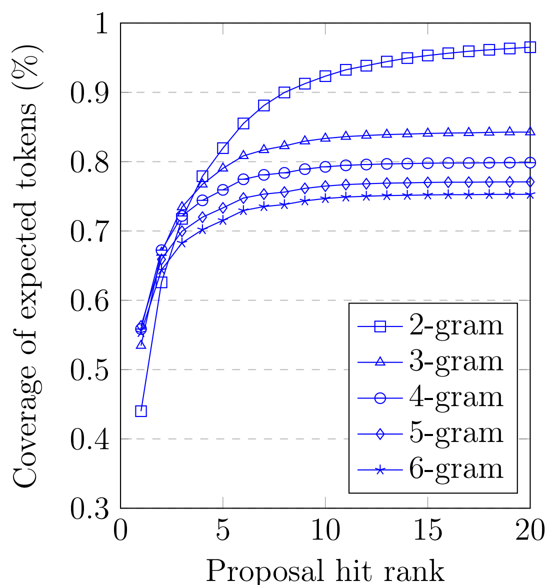


FIGURE 5.14: $N$-gram overall coverage for the JSoup API with Witten-Bell smoothing.

average hit index can go up to 20, the median values range from 1 to 3, which is in accordance with the perplexity values previously obtained.

This relation between the perplexity values and the accuracy/coverage of the recommendations is extended to the JSoup API. Figures 5.13 and 5.14, show the results obtained for the JSoup API. This API had the highest perplexity values, ranging from approximately 9.5 ($n = 6$) to 17 ($n = 2$).

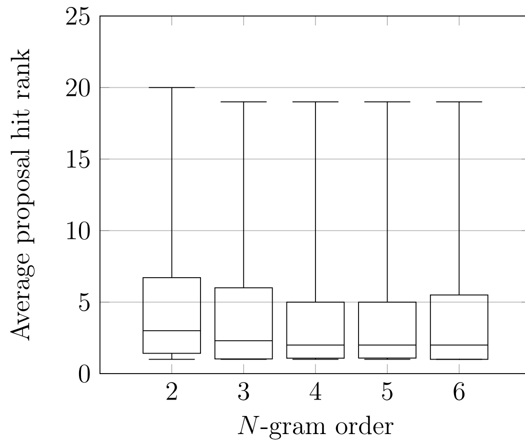


FIGURE 5.15: Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the JSoup API.

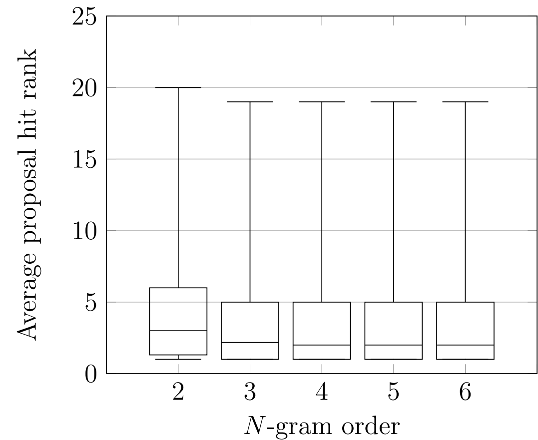


FIGURE 5.16: Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the JSoup API.

Even though the overall coverage exceeds 95% in the top 20 ranks for the 2-grams, the accuracy is lower than any other API, going below 80% in some cases. However, the average hit index (Figures 5.15 and 5.16) is not higher than, for example, the Swing API (see Figures 5.7 and 5.8). The absence of a relation between both evaluations may be due to the corpus, which may not be large enough to represent the API usage correctly.

Regarding the other APIs (one should refer to Appendix A for more details), their low perplexity values explain the results obtained that achieve 100% overall coverage in some cases, as well as a low average hit index. These APIs have small vocabularies which may explain these low average hit indexes, since the model is able to correctly capture the usage patterns. Nevertheless a repetitive usage pattern and a small corpus might be influence these results as well.

Similarly to evaluation in Chapter 4, it is possible to make a relation between the values obtained and model's perplexity. Since the difference between the perplexity values for both smoothing techniques is minimal, this is reflected with a minimal difference in the model's accuracy, e.g. the Witten-Bell smoothing has a lower perplexity which is reflected with a lower median value for the proposals hit rank. As the order of $n$ is increased, the accuracy increases (proposal hit rank decreases), however, the minimum and maximum values stay constant in most

cases independently of the value of $n$. This is due to the nature of the API which contains, in some cases, several different options for the same context.

The obtained results are interesting and hint that a code completion tool would help programmers to correctly interact with the API, while allowing some exploration. However, it is important to note some threats that may be influencing these results. Assuming that the corpus is extracted from projects with a relatively good code quality [3], it is still one of the biggest threats. For example, if the API is mostly used for specific and repetitive tasks, the corpus will contain very few different patterns with a high probability. Since the models are being tested over the same corpus (even though it is a cross-validation scheme), it will most likely recommend the next instruction correctly, thus resulting in a very high overall coverage and very low average hit indexes. Another important threat may be the size of the training corpus which in some cases might no be enough to represent the whole API.

# Chapter 6

# Stepwise Code Completion tool

This chapter describes an application of $n$-gram language models on a recommendation system for stepwise API usage assistance through code completion, based on an API sentence model built from source code corpora. We implemented the proposed approach for Java and the recommendation system was developed as a plugin to the Eclipse IDE. The plugin makes use of the available code completion facilities, namely the pop-up menus that are available when writing code in the editor. Figure 6.1 presents a usage scenario of our APISTA tool[1] where we can see that the code completion proposals adapt to the context, suggesting the use of related API types. Notice that as the context changes, proposals are progressively adapted to the previously written instructions (steps 1 to 3 in the figure). In addition to this kind of recommendations, our system is also suitable for recommending operation calls on a type given its variable (steps 4 and 5 in the figure).

## 6.1 Recommendation (APISTA Tool)

IDEs typically offer facilities for code completion, commonly through pop-up menus in code editors containing proposals that the user may select. The selection of a code completion proposal inserts code at the caret position where the

[1]APISTA stands for *API Sentence Token Assistance*

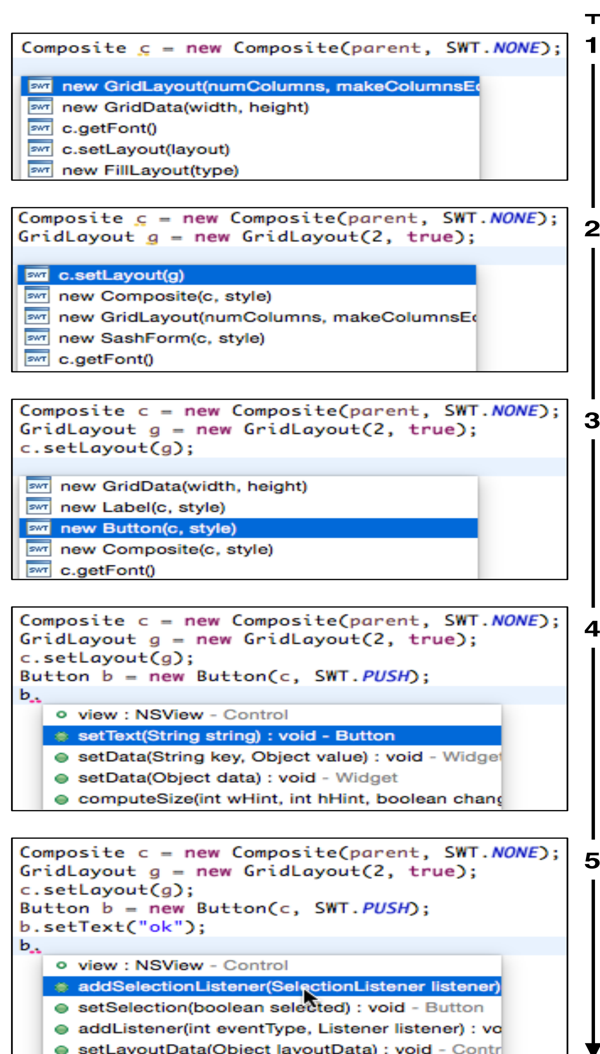


FIGURE 6.1: Stepwise API usage assistance for writing API sentences using the code completion proposals of our APISTA tool in Eclipse.

developer is typing, as well as additional code elsewhere in some cases (e.g., adding required import statements). An IDE such as Eclipse allows third-party plugins to contribute with code completion proposal engines. Using this mechanism, we extended Eclipse with code completion proposals provided by our API sentence model in a tool that we refer to as APISTA.

## 6.2 User interface

Figure 6.2 presents the user interface of our APISTA tool integrated with the code completion menu of Eclipse. As illustrated, the proposals are adapted to the context variables. Each proposal is accompanied by the JavaDoc documentation text, as in regular settings, in order to help the user to select among the available proposals. We believe that having proposals that are adapted to the context variables helps the user to realize how the API objects are composed, besides accelerating code typing. For convenience, import statements are also automatically inserted if necessary, as in other IDE automatization features. We extract the lines of the block in which the user is writing code that precede the line where code completion is requested. Depending on the $n$ order of the trained $n$-gram models that are being used, different context tail lengths will be used to query the model. A list of context variables is also extracted in terms of *identifier* and *type* so that this information can be used to adapt the code completion proposals to match them when possible.

In addition to the illustrated usage mode, another possible way of using the recommendations is through the existing code completion menu of Eclipse for displaying the available operations of a type given a variable (recall Figure 6.1, steps 4 and 5). We sort the Eclipse-provided results according to the ranking of our recommendations, as in Code Recommenders. The same ranking is used as in the other recommendation model, but only the hits pertaining to types that are compatible with the object are considered.

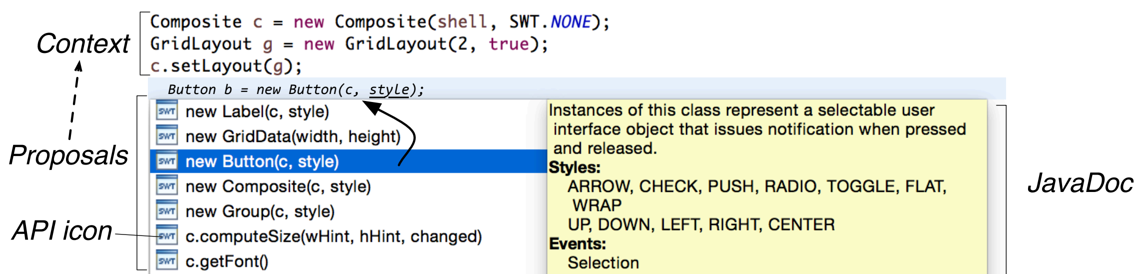


FIGURE 6.2: User interaction with the APISTA tool. Parameters are matched with context variables (for instance, *c*). Unmatched parameters have to be completed manually (for instance, parameter *style*).

The most common situation is that a software project makes use of several APIs. Therefore, it is relevant to consider that the proposals of different APIs have to coexist in the IDE. In our solution for Eclipse, the support for each API can be plugged independently, defining a proposal category for each API with its own icon (as illustrated with SWT). The activation/deactivation of the proposal categories is managed by the IDE. Given that the proposals for each API are determined by the context, if the latter has no tokens of a certain API there will be no proposals with respect to that API.

## 6.3 Integration in software development

The described process for building an API language model is always targeted at a well-defined API. We envision that it should be a responsibility of the API developers to build a sentence model for it, and possibly package the model together with the libraries of that API (for instance, included as part of the JAR files of the libraries). Another option could be based on independent extensions that are installed separately. In either case, the owners of a software artifact should in principle have a good notion of which projects to select for building a model for their API. Although we currently have not yet designed a well-defined format for serializing the API model, this would be straightforward to achieve. The most important aspect is to have a serialized model that is IDE-agnostic, so that different IDEs could seamlessly make use of the same artifact. As with our APISTA tool for Eclipse, a recommender component that is specific to an IDE has to load the model and implement the necessary behaviour to query the model correctly. The recommender component has to extract the context of the code editor to a token format that is compatible with the model's vocabulary.

## 6.4 Limitations

We expect that our approach will aid programmers in the use of unknown APIs through the recommendation of the most likely token the programmer might need. However, there are other obstacles that programmers often face that our approach cannot help with namely:

**Structural context.** As opposed to other approaches (e.g., [17, 41, 9]), ours does not take into account the structural context in which the code is being written. For instance, if the programmer is overriding a certain method of a particular class. We argue that using the structural context is relevant when facing the API of a framework, which requires specialization through inheritance or interface realization, as opposed to a library.

**Parameter values.** Although our system may match parameters of the context whose type belongs to the API vocabulary, it is not able to provide example values for parameters of other types. We believe that one of the strengths of snippet matching recommenders (e.g., [17, 34, 41]) relies on this aspect, given that some parameter values might not be obvious to find.

# Chapter 7

# Conclusions and Future Work

This work presents an approach that exploits $n$-gram language models, widely used in natural language processing, in the context of API usage, in order to produce recommendations for a stepwise API usage assistance tool. Despite the existence of several recommendation tools, to our knowledge, this is the first that allows a stepwise assistance. The objective of this tool is to guide the programmer through the use of an API, even when using an unknown one, by adapting to the previously written code, thus producing the most accurate recommendations.

The results allow us to conclude that the $n$-gram language models are able to capture the regularities for any kind of APIs, whether they are used to create user interfaces or to parse HTML. The models can accurately recommend in over 90% of the cases in the top 20 proposals, which is a very encouraging result. Nevertheless, these models have some limitations that when overcome, will produce much more accurate recommendations.

Despite the existence of several tools, code recommendation is a relatively under-developed area and there is still a lot of room for improvement regarding both the tool's usability as well as the recommendation systems. Regarding our approach, future work and improvements are as follows:

**Model tuning.** Although $n$-gram language models perform well, there are several other modelling techniques that may be interesting to explore.

**Fine-grained tokenization.** A possible future development could be to extract parameter information from the source code repository, so that overloaded methods and constructors have a distinct token. We argue that the significance of this issue relates to the design of a particular API. In some APIs, for instance SWT, where overloading is not much used, improvements should not be significant. On the other hand, given an API with many overloaded methods or constructors the situation might be different.

**Code completion system usability.** A user study through a controlled experiment is necessary to evaluate the usability of our code completion system. The effectiveness of the mechanism from a human-centered perspective may reveal other shortcomings that we did not anticipate, as well as give rise to new ways of improvement. For example, if the results of a user study would reveal that dealing with parameter values consists of a significant hurdle, the sentence extractor process could be enhanced to collect common parameter values, in order to use them in the proposals.

# Appendices

# Appendix A

# Overall Coverage and Average Hit Index Results

## A.1 JDBC Driver for MySQL

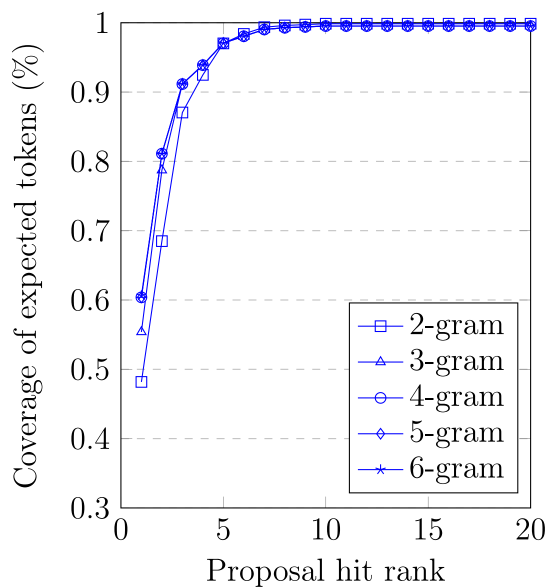


Figure A.1: $N$-gram overall coverage for the JDBC Driver for MySQL API with Kneser-Ney smoothing.

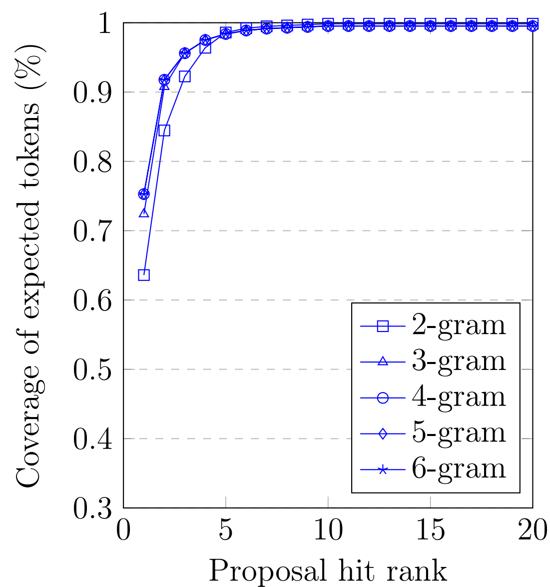


Figure A.2: $N$-gram overall coverage for the JDBC Driver for MySQL API with Witten-Bell smoothing.

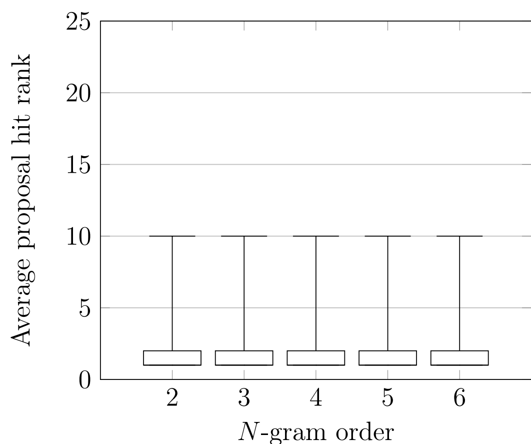


Figure A.3: Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the JDBC Driver API.

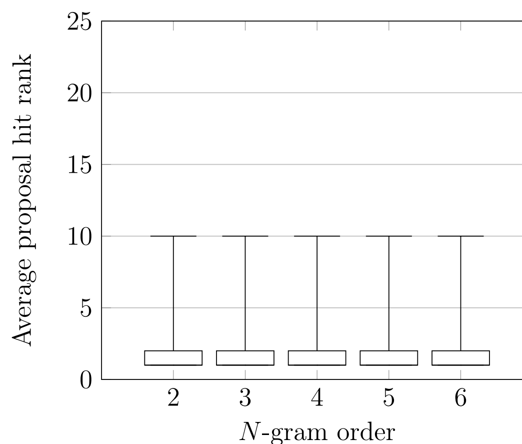


Figure A.4: Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the JDBC Driver API.

## A.2 Jackson-core

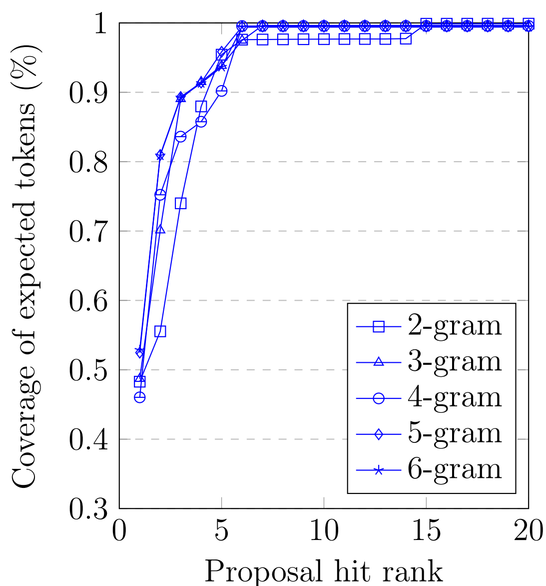


Figure A.5: $N$-gram overall coverage for the Jackson API with Kneser-Ney smoothing.

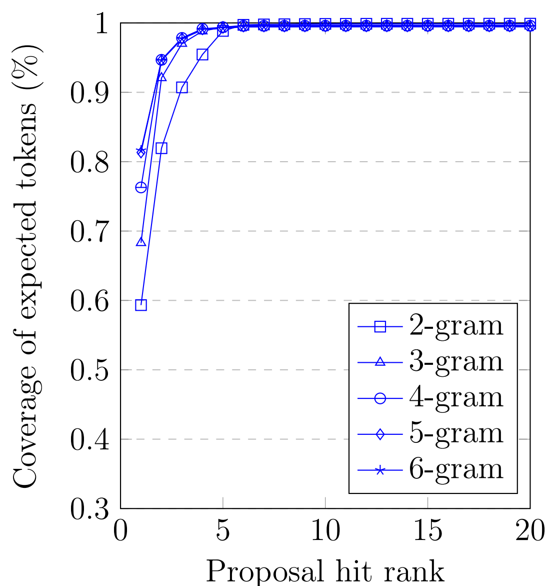


Figure A.6: $N$-gram overall coverage for the Jackson API with Witten-Bell smoothing.

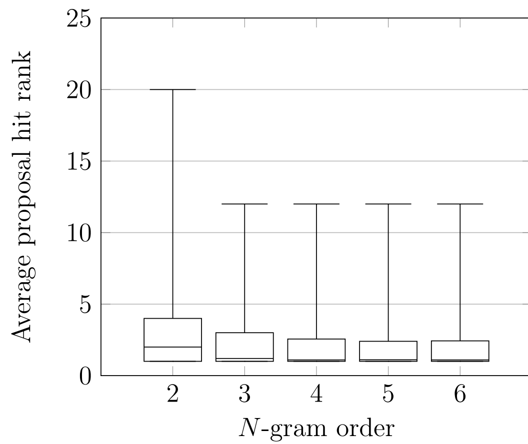


FIGURE A.7: Average proposal rank for each $n$-gram with the Kneser-Ney smoothing for the Jackson API.

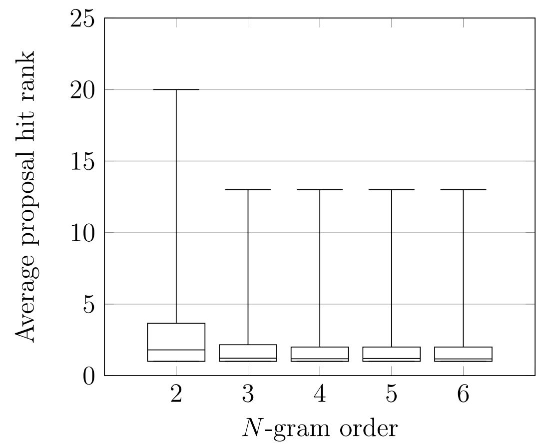


FIGURE A.8: Average proposal rank for each $n$-gram with the Witten-Bell smoothing for the Jackson API.

# Bibliography


[1] Code recommenders. http://www.eclipse.org/recommenders/manual/. Accessed September 11th, 2015.

[2] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.

[3] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 207–216. IEEE, 2013.

[4] Awny Alnusair, Tian Zhao, and Eric Bodden. Effective API navigation and reuse. In *Information Reuse and Integration (IRI), 2010 IEEE International Conference on*, pages 7–12. IEEE, 2010.

[5] Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Abraham Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 2013.

[6] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large Language Models in Machine Translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 858–867. Association for Computational Linguistics, 2007.

[7] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, December 1992.

[8] Marcel Bruch. *IDE 2.0: Leveraging the Wisdom of the Software Engineering Crowds.* PhD thesis, Technical University Darmstadt, 2012.

[9] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[10] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[11] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. 1998.

[12] Ekwa Duala-Ekoko and Martin P Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 266–276. IEEE Press, 2012.

[13] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007.

[14] Joshua T Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434, 2001.

[15] Alan Hevner and Samir Chatterjee. *Design science research in information systems.* Springer, 2010.

[16] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[17] Reid Holmes and Gail C Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, pages 117–125. ACM, 2005.

[18] Reid Holmes, Robert J Walker, and Gail C Murphy. Strathcona example recommendation tool. *ACM SIGSOFT Software Engineering Notes*, 30(5):237–240, 2005.

[19] Daqing Hou and David M Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 233–242. IEEE, 2011.

[20] Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *Proceedings of 1995 International Conference on Acoustics, Speech, and Signal Processing*, ICASSP'1995, pages 181–184. IEEE, 1995.

[21] Ron Kohavi. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, pages 1137–1143. AAAI, 1995.

[22] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[23] Frank Mccarey, Mel Ó Cinnéide, and Nicholas Kushmerick. Rascal: A recommender agent for agile reuse. *Artificial Intelligence Review*, 24(3-4):253–276, 2005.

[24] Frank McCarey, Mel O Cinneide, and Nicholas Kushmerick. Recommending library methods: An evaluation of the vector space model (VSM) and latent semantic indexing (LSI). In *Reuse of Off-the-Shelf Components*, pages 217–230. Springer, 2006.

[25] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542. ACM, 2013.

[26] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[27] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. An empirical study of API usability. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 5–14. IEEE, 2013.

[28] Tommi A Pirinen and Sam Hardwick. Effect of Language and Error Models on Efficiency of Finite-State Spell-Checking and Correction. In *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing*, pages 1–9. Association for Computational Linguistics, 2012.

[29] David M Pletcher and Daqing Hou. BCC: Enhancing code completion for better API usability. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 393–394. IEEE, 2009.

[30] Gonçalo Prendi, Hugo Sousa, A.L. Santos, and Ricardo Ribeiro. Exploring APIs with n-gram language models. In *INForum 2015 - Atas do 7º Simpósio de Informática*, pages 296–310. UBI - Universidade da Beira Interior. Serviços Gráficos, 2015.

[31] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 44. ACM, 2014.

[32] Brian Roark, Murat Saraclar, and Michael Collins. Discriminative n-gram language modeling. *Computer Speech & Language*, 21(2):373 – 392, 2007.

[33] Martin P Robillard. What makes APIs hard to learn? answers from developers. *Software, IEEE*, 26(6):27–34, 2009.

[34] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. *ACM Sigplan Notices*, 41(10):413–430, 2006.

[35] Zachary M Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 15–24. ACM, 2007.

[36] Andreas Stolcke. Srilm-an extensible language modeling toolkit. 2002. In *Proceedings of International Conference on Spoken Language Processing (ICSLP)*, pages 901–904.

[37] Watanabe Takuya and Hidehiko Masuhara. A spontaneous code recommendation tool based on associative search. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pages 17–20. ACM, 2011.

[38] 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*, pages 204–213. ACM, 2007.

[39] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM, 2014.

[40] Tao Xie and Jian Pei. Mapo: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[41] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.