Exploring APIs with N-gram Language Models

Gonçalo Prendi[1], Hugo Sousa[1], André L. Santos[1], and Ricardo Ribeiro[1,2]

[1] Instituto Universitário de Lisboa (ISCTE-IUL), Av. das Forças Armadas, 1649-026 Lisboa

[2] INESC-ID Lisboa, R. Alves Redol 9, 1000-029 Lisboa

goncalo.prendi25@gmail.com, hugossousa92@gmail.com
andre.santos@iscte.pt, ricardo.ribeiro@iscte.pt

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. Such difficulties often lead to 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 paper we explore the use of *n*-gram language models and their ability to capture regularities in APIs. We explored some of the most widely used APIs with the Java programming language, training several language models over hundreds of GitHub Java projects that use these APIs. The evaluation shows perplexity values for these language models that hint the possibility of using them to produce a tool to assist developers with code completion when using an unfamiliar API. Such a tool may help developers to write correct API call sequences more efficiently; moreover, allows them to explore the features offered by the API.

Keywords: APIs, Java, *n*-gram language models, perplexity, source code mining, code completion

1 Introduction

The use of third-party APIs is inherent to modern software development, since it provides a reliable and efficient way of producing software. The correct use of an API can sometimes be hard to perceive if the provided documentation is insufficient or inadequate. The API design itself may cause discoverability hurdles related to the relationships between API types and the sequence of calls that are necessary to obtain an instance of a certain type. These are some of the difficulties programmers often find when using an unknown API [18, 9, 14]. These difficulties may lead to an incorrect use of an API and slow down the realization of programming tasks.

In order to overcome the API usability problems, extensions to Integrated Development Environments (IDEs) have been developed to assist programmers on their API-related tasks (e.g., [6]). Often these tools use Natural Language Processing (NLP) techniques to create statistical models, in order to recommend a set of valid API calls. We refer to these sequences as API *sentences*, which are composed of *tokens* from the API *vocabulary*. These *sentences* are usually extracted from source code corpora. The main contribution of this paper is an overview on the performance of n-gram language models [5] in capturing usage patterns in the context of APIs and which techniques best suit each API.

This paper is organized as follows. Section 2 describes the API sentence extraction method. Section 3 gives an overview over the language models and respective smoothing techniques, Section 4 presents the evaluation methodology and results achieved for each of the APIs. Section 5 discusses the results achieved and draws some conclusions for other APIs as well as a description of threats to validity that might influence the results obtained. Section 6 addresses related work. Section 7 presents our conclusions and outlines future work towards improving the models.

2 API Sentences and Tokens

Language models are highly dependent on how the information is represented in order to produce satisfactory recommendations. Therefore, the extraction process of the API sentences from source code corpora is crucial. A poor extraction process will result on an inaccurate or noisy model, thus producing low quality recommendations that might become more of an obstacle than an aid to the programmer. In order to minimize this noise, our extraction process is handled very carefully. We developed a Java parser on top of the Eclipse's Java Development Tools for this purpose. Next, we present an overview of how API tokens and sentences are handled in the extraction process.

2.1 Tokens

A token is a valid API call that involves a public type and a public operation therein. Static methods and constructors are considered operations as well. Currently, operation parameters are not taken into account (which is a possible improvement for future work), therefore method overloading is not handled. Since these tokens are abstractions of the actual code instruction, they are represented by a tuple consisting on the API's type and the operation. Take the following instructions from the SWT[3] API as an example:

```
Label label = new Label(...);
label.setText(...);
```

The first instruction is represented as the token $\langle Label, new \rangle$ and the second as $\langle Label, setText \rangle$. As stated before, constructors are handled as operations, so we decided to use the "new" keyword to differentiate it from the other operations.

[3] https://www.eclipse.org/swt/

2.2 Sentences

An API sentence is a sequence of tokens, representing related instructions. Formally, a sentence ω is a vector of tokens from the API's vocabulary $\mathcal{V}$:

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

Take the following code snippet as another example:

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

The resulting extracted sentence would be:

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

In this code snippet there are three API calls. The instantiation of the types `Composite` and `GridLayout` and the operation `Composite.setLayout`.

Any instructions that do not represent valid calls on the SWT API are ignored in the parsing process. The tokens' order reflects the execution path, where chained invocations are separated into individual instructions.

Token dependencies are also considered when building the API sentences. Since instructions can be interleaved, an instruction a only depends on another instruction b in the following cases:

1. b is an assignment and a uses the assigned variable;
2. a is an invocation and b is used in its arguments.

The above guarantees that all the dependencies will be represented in the same sentence. Finally, similarly to the MAPO tool [21] (which is used for mining API usage from repositories), `if-else` blocks result in a set of sentences for each possible branch. As for the loop blocks, repetitions are ignored and they are treated as regular selection blocks.

3 API Sentence Models

Language models are widely used on several written and spoken language processing tasks, such as speech recognition [17], spell-checking and correction [15] and machine-translation [4]. Typically, APIs have regular usage patterns that describe valid sequences of API calls. Language models provide a simple and efficient way of capturing these regularities.

3.1 Overview

Language models are statistical models that 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:

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

Equation 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. To compute these probabilities the maximum likelihood estimate is used, where $C()$ is the number of times the sequence appears in the training set, which is given by:

$$P(t_n|t_0t_1\cdots t_n - 1) = \frac{C(t_0t_1\cdots t_n)}{C(t_0t_1\cdots t_n - 1)} \tag{2}$$

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:

$$P(t_n|t_{n-N+1}\cdots t_{n-1}) = \frac{C(t_{n-N+1}\cdots t_n)}{C(t_{n-N+1}\cdots t_{n-1})} \tag{3}$$

N-gram language models are some of the several existing statistical models and have been previously used [11] to capture regularities in source code in general. Our intuition was that n-gram language models would work well in this context, since APIs usually have common usage patterns or regularities with relatively small histories, which can be captured with these models.

After extracting the API sentences from the source code corpora, the SRILM toolkit [19] was used to build the n-gram language models and apply different smoothing techniques available, as well as to evaluate the models' performance.

3.2 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 [7] 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 supports smoothing techniques that we have experimented in our evaluation, namely the Witten-Bell and Kneser-Ney methods.

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 back off to the lower order ones. Equation 4 shows how the Witten-Bell smoothing addresses this problem.

$$P_{\mathrm{WB}}(t_i|t_{i-n+1}^{i-1}) = \lambda_{t_{i-n+1}^{i-1}} P(t_i|t_{i-n+1}^{i-1}) + (1 - \lambda_{t_{i-n+1}^{i-1}}) P_{\mathrm{WB}}(t_i|t_{i-n+2}^{i-1}) \tag{4}$$

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 λ 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).

Kneser-Ney A very frequent token t_1 would be represented in the model with a very high unigram probability. However, if this token is only observed after another token t_0 in the training set, it is very unlikely it will appear after another token t_x. A smoothed probability estimate may be high if the unigram probability of t_1 is taken into account when backing-off. Equation 5 shows how the Kneser-Ney [12] smoothing technique addresses this problem, where D is a constant given by Equation 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 (for more details on the computation of these values see `http://www.speech.sri.com/projects/srilm/manpages/ngram-discount.7.html`) [8].

$$P_{\mathrm{KN}}(t_i|t_{i-n+1}^{i-1}) = \begin{cases} \frac{\max\{C(t_{i-n+1}^{i-1})-D,0\}}{\sum_{ti} C(t_{i-n+1}^{i-1})} & \text{if } C(t_{i-n+1}^{i-1}) > 0 \\ \gamma(t_{i-n+1}^{i-1}) P_{\mathrm{KN}}(t_i|t_{i-n+2}^{i-1}) & \text{if } C(t_{i-n+1}^{i-1}) = 0 \end{cases} \tag{5}$$

$$D = n1/(n1 + 2 * n2) \tag{6}$$

This method consists of taking into account the number of different tokens that precede a token t_1 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 will allow for some exploration of the API by providing some of the uncommon operations.

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

4 Measuring API Perplexity

To evaluate the performance 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 7 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).

$$P(T) = 2^{-\frac{1}{W_T} log_2 p(T)} \tag{7}$$

This technique provides very useful information about the model, such as the average number of choices for each word [10], which is very important since we want the developer to stay in the right path but also allow for some API exploration. Further, it allows us to determine which N value and which smoothing technique best fits each of the APIs, consequently producing the most accurate model.

4.1 Setup

Allamanis and Sutton [2] collected a large corpus from GitHub, containing thousands of Java projects. To perform our evaluation we selected the projects containing references for each of the APIs from the same GitHub Java Corpus. Table 1 shows the APIs that were evaluated, and the number of projects containing at least one reference to the API's root package, either via import statement or qualified type references. After determining which projects contain these references, we used the method described in Section 2 to extract the API sentences.

The performance evaluation for each API was made with a 5-fold cross validation scheme, using both the Witten-Bell and Kneser-Ney smoothing methods, as well as without any smoothing method, in order to evaluate how differently the model performs[4].

There are some words that may not be found in test sets, which are called Out Of Vocabulary (OOV) words. When the perplexity values are computed,

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

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.

4.2 SWT API

SWT 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 is application-specific 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.

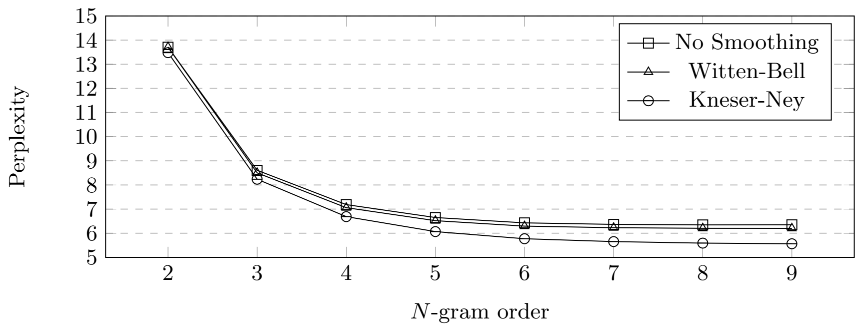

Fig. 1. Perplexity values for the SWT API.

Figure 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.

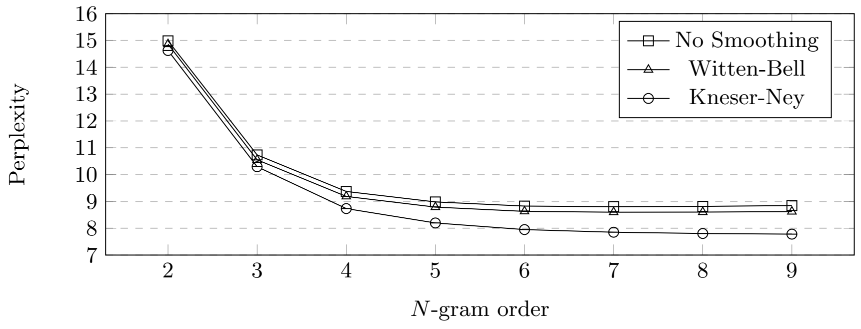

Fig. 2. Perplexity values for the Swing API.

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 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[5] 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.

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 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.

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

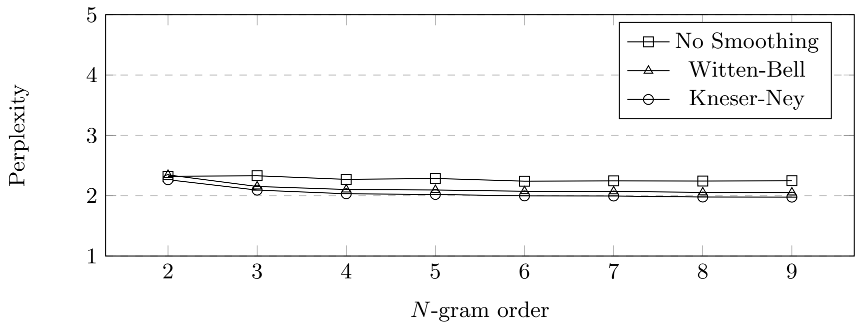

Fig. 3. Perplexity values for the JFreeChart API.

4.5 JSoup

The JSoup[6] 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.

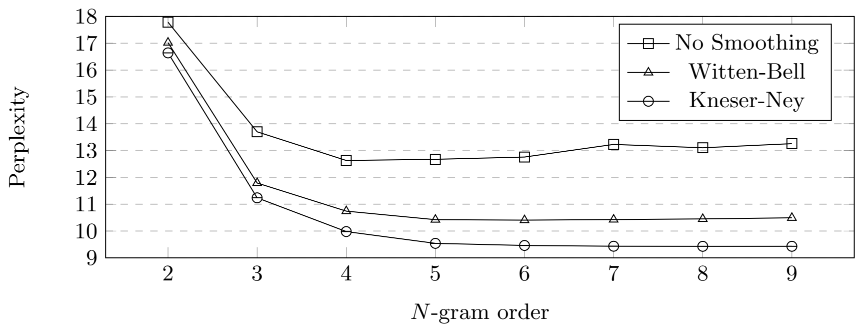

Fig. 4. Perplexity values for the JSoup API.

The high perplexity values (see Figure 4) have to do with the fact that it does not require a very standard 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.

[6] http://jsoup.org/

4.6 JDBC Driver for MySQL

MySQL[7] 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.

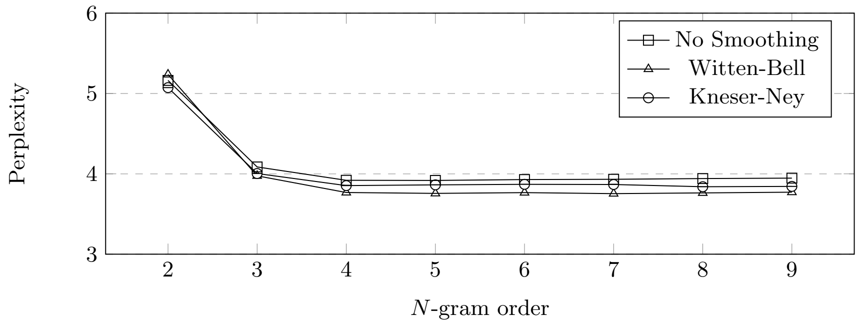

Fig. 5. Perplexity values for the JDBC Driver API.

Our intuition is that the low perplexity values (see Figure 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[8] 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 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 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 7 compares those values across all the analyzed APIs.

[7] http://dev.mysql.com/downloads/connector/j/
[8] http://wiki.fasterxml.com/JacksonHome

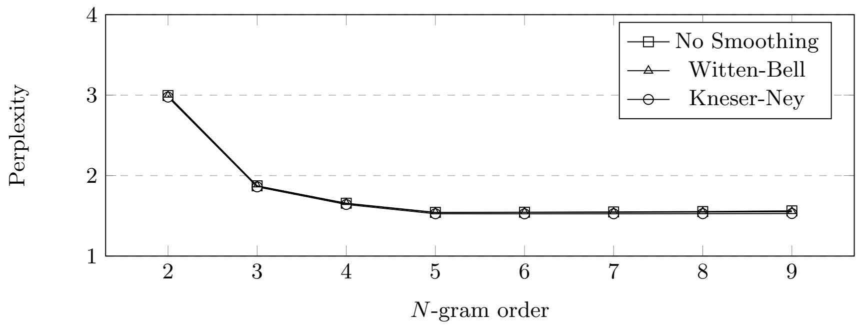

Fig. 6. Perplexity values for the Jackson-core API.

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 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 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 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 such as the JFreeChart API, that has a vocabulary similar to the SWT API and 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. [11] 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 motivates our future work of creating 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 is not accurate when predicting sentences that did not occur in the training corpus. This is one of the most important factors to produce 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

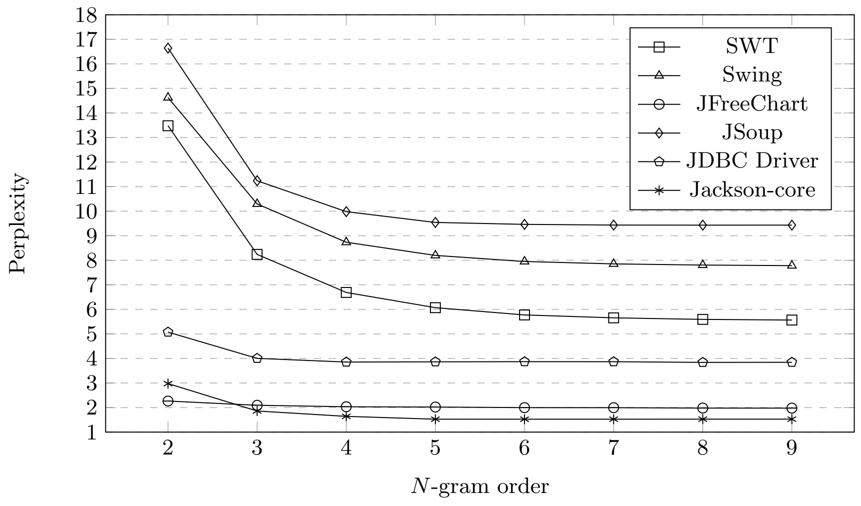

Fig. 7. Perplexity values comparison for all the APIs.

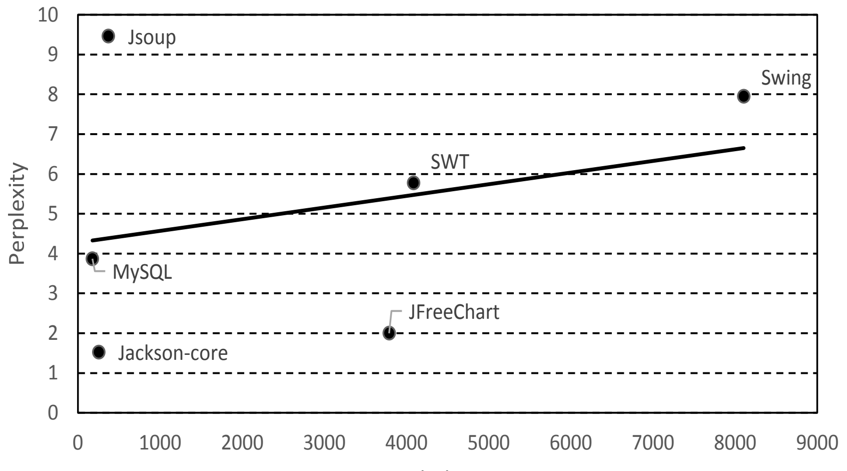

Fig. 8. Relation between vocabulary size and perplexity.

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 [1]. 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.

6 Related Work

Hindle et al. [11] describe that source code in general can be treated as a natural language, thus allowing to create language models to produce recommendation tools to help programmers. It was one of the papers that motivated our work on API usability and exploration using language models. 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.

Recently, a tool called SLANG [16] used n-gram language models and recurrent neural networks to fill holes in partial programs that use a certain API. They also replaced very rare API calls with an unknown token in order compact the language model. Since their work focuses on evaluating the tool and not the language models, we cannot compare the impact of this replacement. 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.

SLAMC [13] is a statistical semantic language model for source code, where Nguyen et al. introduce semantic information into the language models, which represents additional information in the language model that would allow, for example, the global context of source files to help predict the next token. Tu et al. [20], however, argue that code tokenization is enough for n-gram language models. They also state that n-gram models will not help when a particular context is not present in the source code corpora used to train the model. On the one hand, we agree that these models cannot provide accurate recommendations when the code is not present in the training set. On the other hand, we argue that if they do not appear regularly, they are context specific and the probabilities computed with the smoothing techniques should be enough to recommend these tokens.

Although it uses a different approach to produce the recommendations, it is important to mention the Code Recommenders [6] tool. This is an Eclipse Project that uses a modified k-nearest-neighbours [3] called "Best Matching Neighbours", that recommends method calls for objects in the context. It also provides features like "Override Completion" which recommends methods that should be overridden; "Chain Completion" returns a chain of method calls that return a desired type, which is useful when using for example object factories, that require invoking static methods to obtain the desired object; and, finally an "Adaptive Template Completion" that suggests a sequence of methods that usually occur together on a method.

7 Conclusions

In this paper we explored how language models perform on capturing regularities when using APIs, and which smoothing methods provide the best performance results. The obtained results show that language models can successfully be used to capture APIs usage patterns, with perplexity values ranging from 1.52 to 9.46 with the Kneser-Ney smoothing technique at $n = 6$, using interpolation.

There are some code completion systems that help the programmers explore and correctly use unknown APIs. However, these approaches can only recommend method calls for a certain object, or return the desired object type through a sequence of API calls. This requires the programmer to have some knowledge about the API.

As future work, we plan to develop a code completion tool to recommend the most likely tokens the programmer might need, according to the history (the code previously written). Additionally, user studies with programmers should be carried out in order to better assess the usability of such tool.

Regarding the language models, we intend to take into account the method's parameters in order to produce a more accurate model, since a token found in a block of code might depend on these variables.

References

1. Acharya, M., Xie, T., Pei, J., Xu, J.: 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. pp. 25–34. ACM (2007)
2. Allamanis, M., Sutton, C.: Mining source code repositories at massive scale using language modeling. In: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on. pp. 207–216. IEEE (2013)
3. Bobadilla, J., Ortega, F., Hernando, A., Gutiérrez, A.: Recommender systems survey. Knowledge-Based Systems 46, 109–132 (2013)

4. Brants, T., Popat, A.C., Xu, P., Och, F.J., Dean, J.: 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. pp. 858–867. Association for Computational Linguistics (2007)
5. Brown, P.F., deSouza, P.V., Mercer, R.L., Pietra, V.J.D., Lai, J.C.: Class-Based *n*-gram Models of Natural Language. Computational Linguistics 18(4), 467–479 (1992)
6. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. pp. 213–222. ESEC/FSE '09 (2009)
7. Chen, S.F., Goodman, J.: An empirical study of smoothing techniques for language modeling. In: Proceedings of the 34th annual meeting on Association for Computational Linguistics. pp. 310–318. Association for Computational Linguistics (1996)
8. Chen, S.F., Goodman, J.: An empirical study of smoothing techniques for language modeling (1998)
9. Duala-Ekoko, E., Robillard, M.P.: Asking and answering questions about unfamiliar APIs: An exploratory study. In: Proceedings of the 34th International Conference on Software Engineering. pp. 266–276. ICSE '12 (2012)
10. Goodman, J.T.: A bit of progress in language modeling. Computer Speech & Language 15(4), 403–434 (2001)
11. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering. pp. 837–847. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012)
12. Kneser, R., Ney, H.: Improved backing-off for m-gram language modeling. In: Proceedings of 1995 International Conference on Acoustics, Speech, and Signal Processing. pp. 181–184. ICASSP'1995, IEEE (1995)
13. Nguyen, T.T., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: A statistical semantic language model for source code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 532–542. ACM (2013)
14. Piccioni, M., Furia, C.A., Meyer, B.: An empirical study of API usability. In: ESEM'13. pp. 5–14 (2013)
15. Pirinen, T.A., Hardwick, S.: 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. pp. 1–9. Association for Computational Linguistics (2012)
16. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 44. ACM (2014)
17. Roark, B., Saraclar, M., Collins, M.: Discriminative n-gram language modeling. Computer Speech & Language 21(2), 373 – 392 (2007)
18. Robillard, M.P.: What makes APIs hard to learn? answers from developers. Software, IEEE 26(6), 27–34 (2009)
19. Stolcke, A., et al.: Srilm-an extensible language modeling toolkit. In: INTERSPEECH (2002)
20. Tu, Z., Su, Z., Devanbu, P.: On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 269–280. ACM (2014)
21. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and recommending API usage patterns. In: Proceedings of the 23rd European Conference on Object-Oriented Programming. pp. 318–343. ECOOP '09 (2009)