Software Language Identification with Natural Language Classifiers

Juriaan Kennedy van Dam[†] (juriaankennedy@gmail.com) and Vadim Zaytsev[†⋆] (vadim@grammarware.net)
[†]University of Amsterdam, The Netherlands
[⋆]Raincode, Belgium

***Abstract*—Software language identification techniques are applicable to many situations from universal IDE support to legacy code analysis. Most widely used heuristics are based on software artefact metadata such as file extensions or on grammar-based text analysis such as keyword search. In this paper we propose to use statistical language models from the natural language processing field such as n-grams, skip-grams, multinominal naïve Bayes and normalised compression distance. Our preliminary experiments show that some of these models used as classifiers can achieve high precision and recall and can be used to properly identify language families, languages and even deal with embedded code fragments.**

I. Introduction

Software language identification (SLI) is a problem of determining correctly which software language was a source code fragment written in. (We use "code" to deliberately limit ourselves to textual sequential representations of software artefacts. If they happen to be purely graphical, some parsing in a broad sense [1] in the form of image/object recognition [2], [3] must happen first to lift their perception to the structural level, at which point we can use canonical or even ad hoc representation of such "parsed" structures).

Example scenarios of SLI application include:

- **IDE support**: syntax highlighting, code completion, deployment environments, refactoring, etc. Many IDEs offer such support for several different languages, and SLI can help choosing which environment variant to use.
- **Code interaction** as the developer-artefact interface can be affected by the language, determining simple things like what should happen when an Enter key is pressed, as well as more global issues like aiding code navigation.
- **Reverse engineering** a legacy code base, written in an unknown language or a collection of languages, has at some point to step up beyond simple language-agnostic methods to heavyweight reverse engineering activities, most of which are fundamentally language-specific.
- Code search in **unstructured data** such as legacy documentation, email archives, blogosphere, discussion boards, wiki-websites, can be optimised if code fragments are reliably identified and classified into languages.

Following Conway's law, heuristics that use SLI for the purpose of keyword highlighting often are implemented as thresholded statistical keyword counters (prone to misclassifying domain terms); file storing versioning system managers focus on file extensions (limited: does a `.h` file contain C or C++ code? Is a `.pl` file written in Perl or Prolog?); and grammarware-based approaches rarely step beyond attempts to parse everything available with anything that fits (slow and inapplicable to program fragments). Some of these heuristics are computationally heavy, others are unreliably imprecise, and none ever work on small embedded code fragments. In this paper, we investigate whether natural language identification techniques are applicable to software language identification.

Natural language identification is a large and well explored field of natural language processing with many different approaches [4], [5]. In the next section we present a set of SLI methods which are used against one another in the section after that. With the dataset collected, the question which classification method is the best for classifying source code, can be answered. We also look at what other information can be gained from this data and find clusters in software languages, which can show which languages are alike and may belong to the same family. The dataset can help to determine what method is the best at identifying a specific language. It can also be used to find the best method between two specific languages, which could be very helpful if we are in a domain that is limited to a selection of languages — like web pages, which usually only contain HTML, CSS and JavaScript. Finally, we try to determine if it is possible to correctly classify a piece of embedded code (e.g., HTML within PHP; CSS within HTML; any other language within Markdown).

Instead of comparing all natural language identification methods, we will look at the most commonly used ones that do not require any specific knowledge of features of the languages. This means that these methods work with only the training data and **no** additional information. This is a very limiting requirement since many gains can be obtained from comment information, indentation, alphabets, quoting rules, escaping policies or by applying supervised learning. We also leave out particularly complex and heavy computational methods like Support Vector Machines (SVM), since they are usually highly customisable and require substantial research on good feature selection.

II. Language Identification Methods

Five language models will be compared among themselves: Naïve Bayes, n-grams with Good-Turing discounting, n-grams with Knesser-Ney discounting, n-grams with Witten-Bell discounting, the skip-gram language model and a Normalized Compression Distance model. Each of these models can have varying parameters: n-gram size (only for the models using n-grams), tokeniser (all language models except for NCD), size of the training set [6] and if out of vocabulary (OoV) words are weighted or skipped (only the SRILM models).

Multinominal Naïve Bayes (MNB) is a well-known classifier, a simple and easy to implement method, still giving very good results [4], [7]. MNB finds the probability of a document d belonging to a certain class c by:

$$P(c|d) = P(c)\prod_{i=1}^{n} P(x_i|c)$$

$$P(x_i|c) = \frac{\text{freq}(x_i, c) + 1}{\sum\limits_{x\in V} \text{freq}(x, c) + |V|}$$

where n is the number of all the features in the document; freq gives the frequency of x in class c; V is the vocabulary of all classes. To avoid a zero probability in case of not occurring feature, Laplace smoothing is used.

Naïve Bayes classifiers assume that all features are independent of each other, but for classifying software languages some feature dependency can be important. For example, the word "`system`" occurs often in both Java and C#, but the combination of "`system`", "`out`" and "`println`" will almost never appear in C#, while it is standard in Java. To cover these possible dependencies this implementation of MNB uses the frequencies of word n-grams, with a length of one to five words, as the features, instead of the characters or words that are typically used in MNB [8].

N-gram language model is used to predict the probability of a n-gram in a language. More specifically, it predicts the next word after a sequence of words. By using these probabilities, one can calculate the probability that a document d belongs to a certain class c:

$$P(c|d) = \prod_{i=1}^{n} P(w_i|w_{i-n+1}^{i-1})$$

$$P(w_i|w_{i-n+1}^{i-1}) = \frac{\text{freq}(w_{i-n+1}^{i-1} w_i)}{\text{freq}(w_{i-n+1}^{i-1})}$$

where freq gives the frequency of the n-gram in the class.

Since training data is finite, it is very likely that some n-grams will occur in a document are not seen in the training data. These n-grams will cause the above model to assign a probability of zero to the document. These zero-probability n-grams need to be changed to ensure an accurate probability can be calculated. This is done by smoothing. Smoothing changes zero-probabilities and other low probabilities by redistributing some probability of high probabilities. There are different kinds of smoothing techniques; Good-Turing, Modified Kneser-Ney and Witten-Bell were used [9].

Smoothing gives probabilities to n-grams of which the smaller n-grams have occurred in the training data. Words that do not occur in the training data require a different approach. These out of vocabulary (OoV) words can either be skipped or get assigned a probability. Skipping OoV words does not calculate the probability for any OoV word. Assigning a probability is done by changing the first occurrence of each word in the training data to a special `<unk>` token. The language model will then have a unified probability for unknown words.

The n-gram language models were implemented with the SRILM tool [10].

Skip-gram is a variant of n-grams with one or more words "skipped" [11]. For example, `private int x` would form a skip-gram like $\{private, x\}$, causing it to also match `private string x` The skip-gram language model was also implemented with the SRILM tool.

Normalized Compression Distance is a relatively new technique primarily created for clustering [12]. By compressing two files separately and combined, the NCD can show the similarity between the files. The NCD between two files is calculated by:

$$\text{NCD}(x, y) = \frac{C(xy) - \min(C(x), C(y))}{\max(C(x), C(y))}$$

where C gives the size of the file after compression. The resulting number is the similarity score; lower scores means more similar files. In this implementation all training files of a language are concatenated, this combined file is then used to compare each test file with NCD using a compressor. The language that scores the lowest NCD is the most likely language. The compressors tested were GZip and ZLib.

Two tokenisers were used: alpha-numerical and all-symbols. Alpha-numerical only tokenises alpha-numerical characters separated by whitespace; it discards all kinds of punctuation and uses it to slice up the character stream into tokens. All-symbols tokenises all characters separated by whitespace. Also both tokenisers had the option of keeping uppercase and lowercase letters or converting all characters to lowercase.

III. Preliminary Experiments

All the test data and training data for the classifiers are source code files. The source code files were selected from GitHub. Since GitHub already determines the programming language a project is written in, the files were already organised per programming language. Three data sets were created. One set for testing, containing 4000 (200 per language) source code files and two sets for training, small and large, containing 200 (10 per language) and 10000 (500 per language) files respectively. The source code files were collected from multiple GitHub projects. For obvious reasons, none of the projects used in the test data set were used in either of the training sets. To keep the amount of data per language equal each file had to be between 5 and 10kb.

As traditional for natural language identification research, we have used the F_1 measure for ranking:

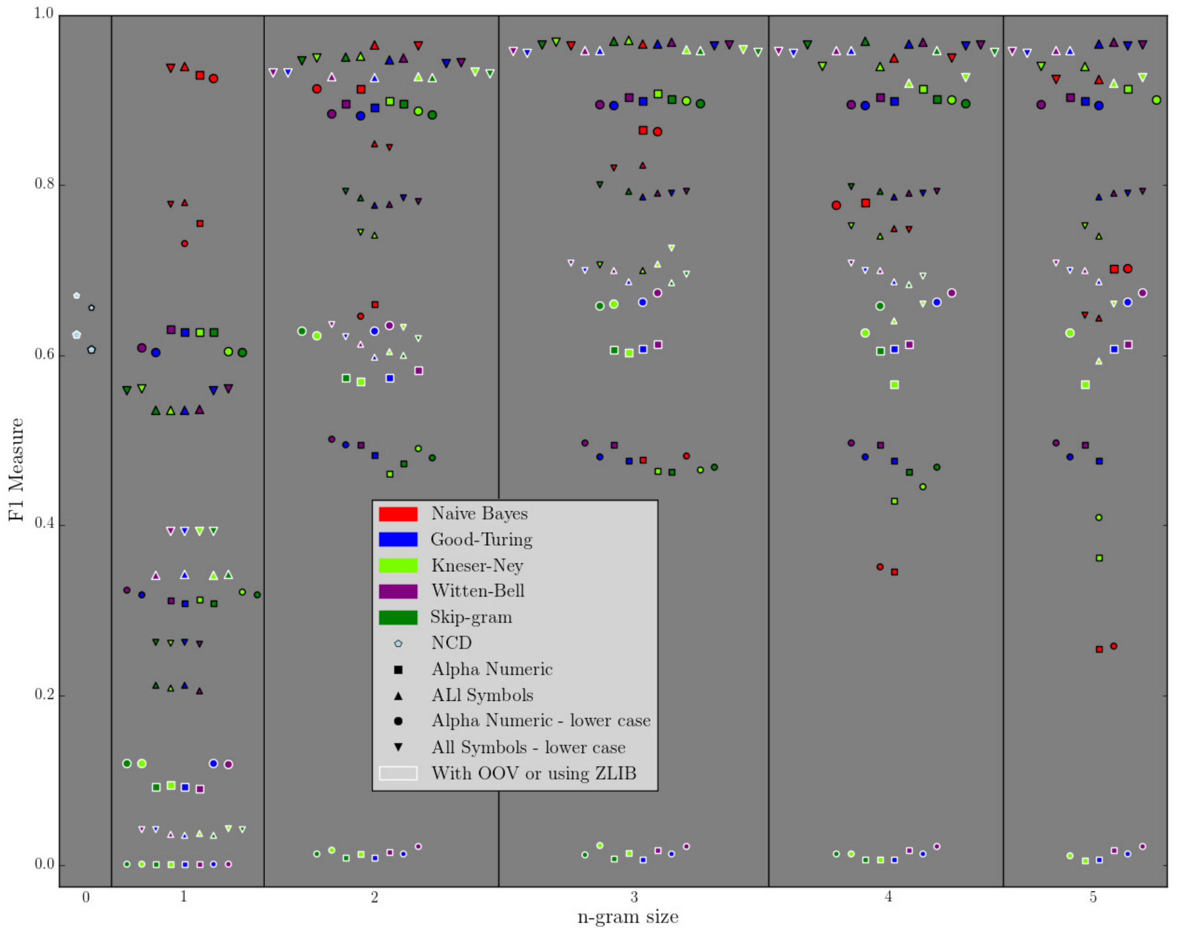

Fig. 1. Language model is coded by colour: red for Naïve Bayes; blue for Good-Turing; lime for Kneser-Ney; purple for Witten-Bell; green for skip-grams. Geometric form denotes a tokeniser: alpha-numeric (■); alpha-numeric lowercase (●); all symbols (▲); all symbols lowercase (▼). ⬠ represents NCD. A white border marks OoV methods. Larger signs refer to large training sets, smaller signs to smaller ones.

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

A. Classifiers Comparison

In total all the classifiers and their varying parameters resulted in **348** different methods. The F_1 measure of each of them can be seen on Figure 1. Many perform adequately, but the classifier with the highest F_1 measure is Modified Kneser-Ney discounting using the all-symbols tokeniser, n-grams with a maximum size of three, the large training set and skipping OoV words. This classifier had a recall of 0.969 and a precision of 0.969 resulting in an F_1 measure of **0.969**, much better than anticipated. The results also clearly show that, with almost all classifiers, having a larger dataset yields better results: only 2 out of 174 classifiers (GZip NCD and ZLib NCD) gave slightly higher F_1 with a small dataset, but they were not competitive with alternatives in all other aspects.

	C#	C++	CSS	Clojure	Go	HTML	Haskell	Java	JavaScr	Lua	Objecti	PHP	Perl	Python	R	Ruby	Scala	Scheme	XML
C	■1	▲3	●1	▲3	▲3	▼3	▲1	■1	■1	▼3	●1	■1	▲4	▼3	▼2	▼2	■1	▼3	▲3
C#		■1	●3	▼4	■1	■4	▼3	■1	●3	▲3	■4	▴4	▼4	▲3	▲3	▲3	▲3	▼3	▴3
C++			▲4	▲4	▴3	▲3	▲4	■1	▴4	▲4	■1	▲4	■1	▲4	▲4	▲3	▴3	▴4	▼3
CSS				▲3	▲3	▴3	▲2	▲2	▾3	▲2	▼3	▲1	●1	▲3	▲3	▼1	▲2	▼3	▲3
Clojure					▲3	▲3	▴3	▼4	▴3	▴3	▲3	▲4	▲3	▲3	▴3	▼3	▴4	●3	▴3
Go						▲3	▴3	▲2	▲5	▲5	▼3	▲3	▴3	▲3	■1	▲4	■1	▼4	▼3
HTML							▲3	●2	▼1	▼3	▼4	▾2	▼3	▲3	▲3	▼3	▲3	▾3	■2
Haskell								▲2	▲2	■1	▲2	▴3	▴3	▲3	▲3	▼3	▴3	▲5	▲3
Java									■4	▲2	▲2	▲3	▲2	▲3	▲3	▼4	▾4	▼4	▴4
JavaScript										▼4	▾3	▲4	▴3	▲3	▴3	▾3	▴3	▼3	▲3
Lua											▼3	▲4	▾3	■3	▲3	▼4	▲3	▼3	▼1
Objective-C												▾4	▾3	▼3	▲3	▼3	▲2	▼2	▼2
PHP													■4	▲3	▲4	▼3	▲3	▲3	▲3
Perl														▲2	▼2	▲4	▴3	▼3	▲3
Python															▴3	●2	▲3	▲3	▲1
R																●1	■1	▴3	▲2
Ruby																	▼2	▼3	▲3
Scala																		▲2	▼2
Scheme																			▼4

TABLE I

Best classifier per language combination, in the same notation.

B. Best Classifier per Language

If the problem is limited to identifying *one* specific language, we can advise to use the classifier that was especially good for that language and not on average: the accuracy is upwards of 0.975 with C++ (0.865) as the worst case outlier.

	Classified as																			
Actual language	C	C#	C++	CSS	Clojure	Go	HTML	Haskell	Java	JavaScript	Lua	Objective-C	PHP	Perl	Python	R	Ruby	Scala	Scheme	XML
C	82	0	15	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
C#	0	96	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
C++	15	1	80	0	0	0	0	0	1	0	0	2	0	0	0	0	0	0	0	0
CSS	0	0	0	98	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Clojure	0	0	0	0	97	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
Go	0	0	0	0	0	98	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HTML	0	0	0	0	0	1	93	0	0	4	0	0	1	0	0	0	0	0	0	1
Haskell	0	1	0	1	0	0	0	96	0	0	0	0	0	0	0	0	0	0	0	0
Java	0	0	1	0	0	0	0	0	98	0	0	0	0	0	0	0	0	0	0	0
JavaScript	0	1	1	0	0	1	1	0	1	93	0	0	0	0	0	0	1	0	0	0
Lua	0	0	0	0	0	1	0	0	0	1	93	1	0	0	1	0	1	0	1	0
Objective-C	1	0	1	0	0	0	0	0	0	0	0	97	0	0	0	0	0	0	0	0
PHP	0	0	0	1	0	0	2	0	0	0	0	0	95	0	0	0	0	0	0	0
Perl	0	0	0	0	0	0	0	0	0	0	0	0	1	98	0	0	0	0	0	0
Python	0	0	0	0	0	0	0	0	0	0	0	0	1	0	96	0	1	0	0	0
R	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	96	0	0	0	0
Ruby	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	95	0	0	0
Scala	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	98	0	0
Scheme	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	96	0
XML	0	0	0	0	0	0	3	0	0	0	0	0	1	0	0	0	0	0	0	93

TABLE II
Percentage of files classified as a certain language with F_1 measure higher than 0.9.

	Classified as																			
Actual language	C	C#	C++	CSS	Clojure	Go	HTML	Haskell	Java	JavaScript	Lua	Objective-C	PHP	Perl	Python	R	Ruby	Scala	Scheme	XML
C	20	8	20	0	1	3	1	0	10	7	4	16	3	5	1	1	0	1	1	0
C#	5	20	17	1	0	4	1	1	17	10	1	7	1	6	0	0	1	4	0	2
C++	19	12	20	0	1	3	1	0	10	6	2	14	4	5	0	1	0	1	1	0
CSS	1	4	1	20	1	4	10	2	2	5	4	2	2	1	4	5	9	7	17	1
Clojure	2	8	2	2	20	0	0	18	2	10	6	1	2	15	1	1	8	0	1	0
Go	3	13	6	1	0	20	1	0	6	10	10	2	1	1	3	3	5	14	1	0
HTML	2	5	3	4	1	3	20	1	1	4	7	1	7	3	7	5	9	10	5	0
Haskell	0	2	1	0	2	3	0	20	2	15	12	0	2	18	1	1	2	0	9	9
Java	5	18	14	1	1	2	0	1	20	10	0	6	0	5	0	0	0	15	0	1
JavaScript	3	9	5	1	0	8	1	13	8	20	8	1	3	8	2	2	4	3	1	0
Lua	3	4	3	2	1	9	5	1	1	11	20	2	3	2	12	6	12	3	1	0
Objective-C	13	10	14	0	0	3	1	2	3	13	5	20	4	4	2	0	3	1	0	0
PHP	3	2	5	1	0	1	6	2	1	13	2	2	20	16	4	5	13	1	2	0
Perl	7	2	8	1	1	0	1	8	5	17	3	1	15	20	1	1	3	3	1	1
Python	1	3	1	1	1	4	2	1	1	11	14	3	5	3	20	4	15	10	1	0
R	2	3	2	2	1	7	5	1	2	10	10	0	6	3	8	20	10	8	2	0
Ruby	0	2	1	4	0	2	3	1	1	9	16	2	11	3	13	6	20	4	1	0
Scala	2	9	4	3	1	11	4	1	13	10	2	3	1	1	7	2	7	20	1	1
Scheme	4	3	5	18	2	2	12	2	1	4	4	2	2	2	4	3	3	7	20	0
XML	0	6	2	2	1	1	2	13	5	4	8	3	7	10	6	2	3	3	4	19

TABLE III
Percentage of files ranking in top 5 as a certain software language with the F_1 measure higher than 0.9.

C. Best Classifier per Language Combination

More interestingly, we can look into what classifiers that are best capable of differentiating between two languages, because there are groups that are particularly easy to mix up, and distinguishing between, say, C and C++ or HTML and XML is a realistic application scenario. By only analysing language identification results for pairs of languages, we rerun the experiments for each pair of languages, calculated precision and recall and combined those ranks in Table I.

D. Viability of Reasonable Classifiers

There are a couple of particularly weak classifiers in our set that blindly classified everything as Java or everything as Objective C — we exclude them from the final analysis, setting the threshold at $F_1 > 0.9$. Table II shows the percentages of files which software language they were able to identify correctly. As we can see, the results are rather promising except for C and C++, which are mistaken for each other quite often (15% of the time). Together with the fact that there were languages that were never mistaken for one another, this hints at the existence of families of related software languages.

E. Software Language Families

Table III shows all classifiers with a F_1 measure higher than 0.9 and uses the top five languages a file is predicted to belong to. With this method many possible relationships can be seen: e.g., Java seems to be related to C#, C++ and Scala. Also JavaScript is ranked highly among almost every language.

F. Detecting Embedded Software Languages

To detect which piece of code belongs to what software language within a document, we used the "best classifier per language combination" to try to find JavaScripts snippets in HTML files collected from big websites, and checked results automatically obtained using the Naïve Bayes classifier, with n-grams of length 1 and the all-symbols lower case tokeniser (which was the best at differentiating JavaScript and HTML in subsection III-C). Our preliminary experiment on 3605 lines of HTML resulted in 3187 lines classified correctly, which gives an accuracy of 0.88. The actual language of a line was determined by looking at the tags and the tag its content in that line. If more than half of the line consisted out of `<script>` tags and `<script>` content, the line was classified as JavaScript, otherwise the line was classified as HTML.

IV. Related Work

Computer scientists have been fascinated by what they can learn from natural linguistics for a long time [13]. Combining natural language processing, information retrieval and software analysis is promising [14], even though still underexplored.

The most usual place to find natural language processing methods in software engineering is requirements engineering: there have been uncitably many papers, luckily with some surveys available [15], [16]. Somewhat closer to our topic, there have been several reports on successful near-clone detection in natural language artefacts such as defect reports [17], [18].

Blosseville et al. combine natural language analysis and statistical analysis to design a system of supervised learning that classifies project descriptions [19]. If our approach is combined with theirs, it could be possible to improve SLI accuracy further by switching to semi-automation (supervision).

Nakamura et al. propose to facilitate understanding of graphical software models by annotating them with links to concepts known from a given vocabulary as well as other natural language artefacts [20]. Unfortunately they do not propose any automation strategy which naturally limits the usefulness of their solution to *forward* engineering.

SLI can be seen as a very specific form of fact extraction, recovering only one extremely trivial fact (what is the language?) yet applied on such an early stage that nothing else can be achieved until this fact is settled. Thus, we treat fact/concept extraction with natural language techniques as related work [21], [22]. One step farther lie other techniques that use extracting identifiers, literals and comments and leveraging them to improve software maintenance tools without performing fact/concept extraction in a strict sense [23]. It is also possible to generate natural language summaries of related code fragments once their identification was completed using static code analysis [24], [25], but nobody seems to have been able yet to connect the two ends and apply numerous natural language summarisation techniques directly to code.

One of the experiments most closely related to our project was performed by Merten et al. last year: combining white space tokenisation, text heuristics and agglomerative hierarchical clustering, they were able to tell natural text from code $F_1 = 0.84$ [26]. Their notion of "code" was very similar to ours, covering actual code fragments but also log files and stack traces. We see our contribution as complementary to theirs since we essentially focus on refining this classification as identifying language families within code and then languages within language families. Interestingly, judging from techniques' accuracy, automatic identification of software language families within code is easier than identification of code within unstructured data, but harder than identification of software languages within families.

V. Conclusion and Future Work

We have presented early results of researching the applicability and viability of SLI with methods available in natural language identification. 348 different classifiers were obtained from variations of lightweight natural language methods and compared on a modest size corpus of 20 languages (per language we had 109–150 test projects and 146–272 training projects). While results on smaller training sets were close to being consistently wrong, there were several methods that, given a large training set, were able to correctly identify software languages with the accuracy of 0.975 and higher.

The results obtained so far clearly show that natural language methods of language identification are definitely worth considering in the scope of software language reverse engineering. In the future we plan to run more convincing experiments positioning best NLP-based methods among other approaches to SLI, highlighting corner cases of pairs of software languages commonly confused by one family of methods but not by the other. We also expect certain types of software (e.g., compiler sources) to confuse many methods. False positives and false negatives should be inspected manually to determine causes for misclassification, possibly followed by recalibration of the chosen classifier.

We also consider extending the set of chosen methods (including SVMs which perform very well for natural languages) as well as the training set (covering software language variants, versions and dialects).

References

[1] V. Zaytsev and A. H. Bagge, "Parsing in a Broad Sense," in *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, ser. LNCS, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfrán, Eds., vol. 8767. Springer, 2014, pp. 50–67.

[2] O. Augereau, N. Journet, and J.-P. Domenger, "Semi-structured Document Image Matching and Recognition," in *Proceedings of the 20th Conference on Document Recognition and Retrieval (DRR)*, ser. SPIE Proceedings, vol. 8658. SPIE, 2013.

[3] V. M. Kiyko, "Recognition of Objects in Images of Paper Based Line Drawings," in *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR v.II)*. IEEE, 1995, pp. 970–973.

[4] T. Baldwin and M. Lui, "Language Identification: The Long and the Short of the Matter," in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2010, pp. 229–237.

[5] T. Gottron and N. Lipka, "A Comparison of Language Identification Approaches on Short, Query-style Texts," in *Proceedings of the 32nd European conference on Advances in Information Retrieval (ECIR)*, ser. LNCS, vol. 5993. Springer, 2010, pp. 611–614.

[6] M. Banko and E. Brill, "Mitigating the Paucity-of-Data Problem: Exploring the Effect of Training Corpus Size on Classifier Performance for Natural Language Processing," *Computational Linguistics*, pp. 2–6, 2001.

[7] A. Kibriya, E. Frank, B. Pfahringer, and G. Holmes, "Multinomial Naive Bayes for Text Categorization Revisited," in *Advances in Artificial Intelligence*, 2004, pp. 488–499.

[8] M. Lui and T. Baldwin, "Cross-domain Feature Selection for Language Identification," in *Proceedings of the Fifth International Joint Conference on Natural Language Processing*, 2011, pp. 553–561.

[9] S. F. Chen and J. Goodman, "An Empirical Study of Smoothing Techniques for Language Modeling," in *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ser. ACL. Association for Computational Linguistics, 1996, pp. 310–318.

[10] A. Stolcke, "SRILM — an extensible language modeling toolkit," in *Proceedings of the Seventh International Conference on Spoken Language Processing (ICSLP)*, J. H. L. Hansen and B. L. Pellom, Eds. ISCA, 2002, pp. 2:901–904.

[11] D. Guthrie, B. Allison, and W. Liu, "A Closer Look at Skip-gram Modelling," in *Proceedings of the 5th international Conference on Language Resources and Evaluation*, ser. LREC, 2006, pp. 1222–1225.

[12] R. Cilibrasi and P. M. B. Vitányi., "Clustering by Compression," *IEEE Transactions on Information Theory*, vol. 51, no. 4, pp. 1523–1545, 2005.

[13] P. Naur, "Programming Languages, Natural Languages, and Mathematics," in *Conference Record of the Second Symposium on Principles of Programming Languages (POPL)*, R. M. Graham, M. A. Harrison, and J. C. Reynolds, Eds. ACM Press, 1975, pp. 137–148.

[14] L. L. Pollock, "Leveraging Natural Language Analysis of Software: Achievements, Challenges, and Opportunities (Keynote)," in *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*. IEEE, 2012, p. 4.

[15] D. Falessi, G. Cantone, and G. Canfora, "A Comprehensive Characterization of NLP Techniques for Identifying Equivalent Requirements," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2010, pp. 18:1–18:10.

[16] U. S. Shah and D. C. Jinwala, "Resolving Ambiguities in Natural Language Software Requirements: A Comprehensive Survey," *SIGSOFT Software Engineering Notes*, vol. 40, no. 5, pp. 1–7, Sep. 2015.

[17] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 499–510.

[18] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 461–470.

[19] M. J. Blosseville, G. Hébrail, M. G. Monteil, and N. Pénot, "Automatic Document Classification: Natural Language Processing, Statistical Analysis, and Expert System Techniques used together," in *Proceedings*

of the 15th International Conference on Research and Development in Information Retrieval (SIGIR). ACM, 1992, pp. 51–58.

[20] Y. Nakamura, R. Furukawa, and M. Nagao, "Diagram Understanding Utilizing Natural Language Text," in *Proceedings of the Second International Conference on Document Analysis and Recognition (ICDAR).* IEEE, 1993, pp. 614–618.

[21] J. Nilsson, W. Löwe, J. Hall, and J. Nivre, "Natural Language Parsing for Fact Extraction from Source Code," in *Proceedings of the 17th International Conference on Program Comprehension (ICPC).* IEEE, 2009, pp. 223–227.

[22] S. L. Abebe and P. Tonella, "Natural Language Parsing of Program Element Names for Concept Extraction," in *Proceedings of the 18th International Conference on Program Comprehension (ICPC).* IEEE, 2010, pp. 156–159.

[23] L. L. Pollock, K. Vijay-Shanker, D. C. Shepherd, E. Hill, Z. P. Fry, and K. Maloor, "Introducing Natural Language Program Analysis," in *Proceedings of the Seventh Workshop on Program Analysis for Software Tools and Engineering (PASTE).* ACM, 2007, pp. 15–16.

[24] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating Natural Language Summaries for Crosscutting Source Code Concerns," in *Proceedings of the 27th Conference on Software Maintenance (ICSM).* IEEE, 2011, pp. 103–112.

[25] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Automatic Generation of Natural Language Summaries for Java Classes," in *Proceedings of the 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 23–32.

[26] T. Merten, B. Mager, S. Bürsner, and B. Paech, "Classifying Unstructured Data into Natural Language Text and Technical Information," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR).* ACM, 2014, pp. 300–303.