Identifying source code programming languages through natural language processing

Juriaan Kennedy van Dam

January 20, 2016

Supervisor: dr. Vadim Zaytsev
Host organisation: Universiteit van Amsterdam

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

The identification of source code language is not always possible through metadata, like file extensions. Using parsers for each language is not a good solution, because of incorrect and unparseable code, performance and high maintenance. Using natural language processing techniques for classification could be a better approach. This thesis contains a comparison of different NLP classification techniques such as Naive Bayes, n-grams, skip-grams and normalized compressions distance. It also covers other parameters like tokenizers and training set size. With this dataset the best classifier can be determined. Besides that, there was also looked at other possible uses of this data: language family detection and detection of embedded code. Some of the classifiers have a very high F1 measure, with the highest being 0.97. Other results also show that it is possible to detect embedded code with an accuracy of 88%. Future work can use this research as a basis to find better classifiers or more applications of the dataset.

A condensed version of this thesis was accepted for publication in Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Early Research Achievement track [1].

Contents

1 Introduction **2**

2 Method & Experiment **5**
2.1 Classification methods 5
2.1.1 Language Models 5
2.1.2 Tokenizers 7
2.2 Data selection 8
2.3 Evaluation 8

3 Results **10**
3.1 Classifier evaluation 10
3.2 Parameter evaluation 12
3.3 Best classifier per language 12
3.4 Effects of looking at lower ranks 13
3.5 Looking at language families 15
3.6 Best classifier between two languages 17
3.7 Detecting embedded code 18

4 Discussion **20**
4.1 Analysis of results 20
4.2 Threats to validity 21
4.3 Future work 22
4.4 Conclusion 22

5 Bibliography **24**

Chapter 1

Introduction

Language Identification is a large and well explored field of natural language processing (NLP) with many different approaches [2, 3, 4, 5]. These methods are able to cover the identification of *natural* languages, but remain largely untested on *programming* languages. Programming language identification of source code is a way to categorize the large amount of code snippets found on the Internet. There are many blogs, forums and other websites that contain small examples of code. Correctly classifying those examples can greatly increase the search ability of code in search engines. Besides that it could also improve syntax highlighters, by more accurately deciding what highlighting rules should be used, and it could assist with automated analysis of source code [6].

Currently code hosting services like GitHub categorize source code by looking at the file extension [7] and tools like SyntaxHighlighter [8] or SearchCode [9] use language features, like keywords and syntax rules, to identify the language. However, these methods do not work very well on small snippets of code. Another way of finding what language a piece of code is written in could be to gather the grammars of programming languages and try to parse the code. This method has high cost, since every time a new version of the language is released, the grammar needs to be updated as well. It is also possible that a grammar can not recognize a snippet of code, because it is only a part of the entire file.

This thesis presents a dataset of multiple programming language identification methods. With this dataset, the question which classification method is the best for classifying source code can be answered. There has also been looked at what other information can be gained from this data. The data can be used to find clusters in programming languages, which can show which programming languages are alike and might be of the same family, like C and C++. The dataset can also determine what method is the best at finding 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 only has those languages, like web pages, which can only contain HTML, CSS and JavaScript. Finally this thesis tries to determine if it is possible to correctly classify a piece of embedded code. Embedded code is code that mixes two languages, for ex-

ample, PHP is often used with embedded HTML. This is done by making use of the dataset, through which we can determine the best classifying method between two languages.

Scope

There are many different methods in natural language processing for language identification and comparing all of them is not the goal of this thesis. Instead there will be looked at the most commonly used methods in NLP 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, for example how comments or strings are indicated in a language. Also particularly complex and heavy computational methods are left out. The goal of the thesis is to see how well NLP classification methods apply to programming languages, so it will start with the more general and basic methods, after which it could be a possibility to see how more specialized methods perform.

One well known method that is not being tested is the support vector machine (SVM) [10]. SVMs are not tested because they are computationally heavy. SVMs require that the documents are transformed into features that represent the document. In text classification these features are usually the words occurring in the text. This leads to a large amount of features per document. Since SVMs compare every feature to each other such a large amount of features would cause an extremely large amount of dimensions which is very heavy to process [11]. For this reason text classification with SVMs apply some kind of feature selection. Good feature selection is based on the data the SVM is being used on [12, 13]. Finding good features for identifying programming language is a large undertaking and therefore falls out of the scope of this thesis.

Related Work

The amount of research on applying natural language processing techniques on programming languages is scarce. There is some research that uses n-grams for malware detection [14, 15] and identifying source code authors [16]. Other research shows how code completion and code suggestions can be improved by using NLP techniques [17, 18].

The identification of source code language is also a largely unexplored topic. Merten et al. try to detect source code in natural language texts, but they do not try to identify the programming language itself [19]. Ugurel et al. tried to classify source code with the help of SVMs. They found decent results, but their methods were only tested on a small amount of test files. Their method also makes use of a very basic feature extraction method and depends on the README files in projects [20]. Klein et al. created a tool that uses statistical features like the amount of special characters and punctuation. The tool had an accuracy of 48% [21]. Lastly Khasnabish et al. used Naive Bayes classifiers

to classify source code. They used a large amount of training files and made use of a list of keywords for each language. Their best model had an accuracy of 93% [22].

Chapter 2

Method & Experiment

Creating a dataset that compares different techniques for language identification requires classification methods, data and a way to evaluate the methods. The following sections describe:

- Which language models were tested, how they work and what parameters they use.
- An explanation about the different tokenizers.
- How the data was selected.
- How the classification methods were evaluated.

2.1 Classification methods

Five language models were used for the comparison. Each of these models can have varying parameters: n-gram size (only for the models using n-grams), tokenizer (all language models except for NCD), size of the training set [23] and if out of vocabulary (OOV) words were weighted or skipped (only the SRILM models)

2.1.1 Language Models

Five language models were tested: the Naive Bayes language model, three n-gram language models using Good-Turing discounting, Knesser-Ney discounting and Witten-Bell discounting, the skip-gram language model and a Normalized Compression Distance model.

Multinomial Naive Bayes

Multinomial Naive Bayes (MNB) is a well known and often used classifier. It is a simple method and easy to implement, but gives very good results [2, 22, 24, 25].

MNB finds the probability of a document d belonging to a certain class c by:

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

where n is the number of all the features in the document and

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

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

Naive Bayes classifiers assume that all features are independent of each other, but for classifying programming 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 [26].

N-gram language model

N-gram language models try to predict the probability of an n-gram in a language. More specifically they predict the probability of a word occurring after a sequence of words. By using these probabilities one can calculate the probability that a document d belongs to a certain class c:

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

where:

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

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 which 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 [27] and Witten-Bell were used.

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 probability for unknown words.

The n-gram language models were implemented with the SRILM tool [28].

Skip-gram

A variant on n-grams are skip-grams, which are n-grams but with one or more words 'skipped' [29]. 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

Normalized Compression Distance (NCD) is a relatively new technique primarily created for clustering [3]. By compressing two files separately and combined, the NCD can show the similarity between the files. The NCD between two files x and y is calculated by:

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

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 LZIB.

2.1.2 Tokenizers

Two tokenizers were used: alpha-numerical and all-symbols. Alpha-numerical only tokenizes alpha numerical characters separated either by whitespace or all other non alpha numerical characters. All-symbols tokenizes all characters separated by whitespace. Also both tokenizers had the option of keeping upper -and lowercase letters or converting all characters to lowercase.

Language	Test	Large	Small
C	109	146	10
C#	121	171	10
C++	110	150	10
Closure	134	255	10
CSS	150	271	10
Go	120	185	10
Haskell	123	186	9
HTML	126	189	10
Java	131	189	10
JavaScript	125	181	10
Lua	128	197	10
Objective-C	128	199	9
Perl	133	200	10
PHP	122	179	10
Python	114	167	10
R	137	223	10
Ruby	133	197	9
Scala	119	178	10
Scheme	134	215	10
XML	141	272	10

Table 2.1: Number of projects per language

2.2 Data selection

All the test- 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 organized 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 projects. 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.

2.3 Evaluation

Each classifier was evaluated by running the classifier on the test data with the classifier trained using the training data. Per classifier this resulted in a list of all the files in the test set, the actual language of the files and a list ranking all languages, where the highest rank is the most likely language according to the classifier. When only looking at the highest ranking language, the evaluation is

done by calculating the balanced F1 measure:

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

for each language and taking the the average. The *Precision* is the fraction of the files that were correctly assigned to a certain language (also known as positive prediction value):

$$Precision = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

and the *Recall* is the fraction of the files of a certain language that were correctly classified (also known as sensitivity):

$$Recall = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

When not only looking at the highest scoring language, but at how high each language ranks, calculating the recall is not possible anymore, since there are no false negatives. In these cases the accuracy is used for evaluation instead:

$$Accuracy = \frac{\text{Correctly classified files}}{\text{Total files}}$$

Chapter 3

Results

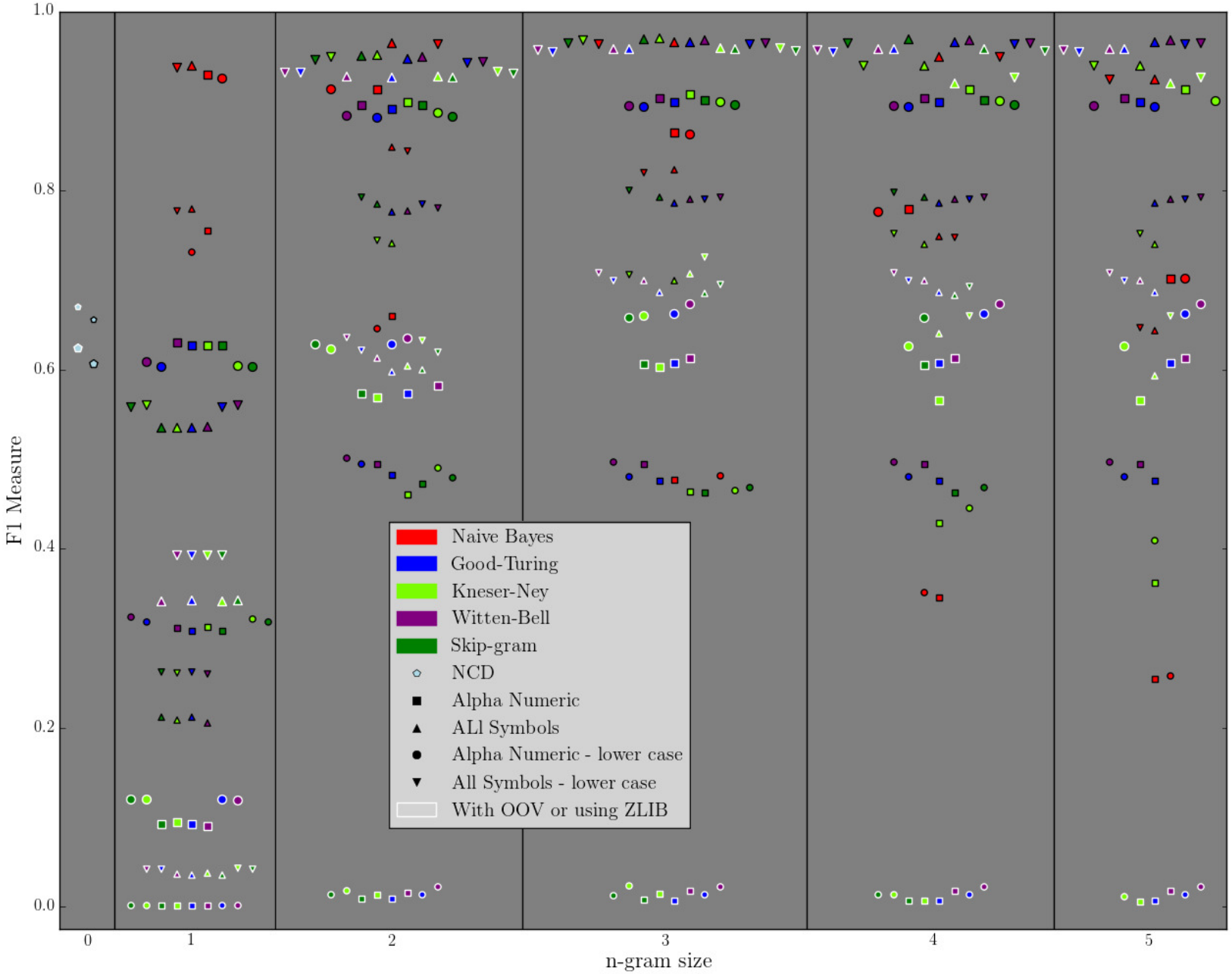
With the dataset of identification methods the following research questions can be answered:

- Which NLP classifier is the best at identifying programming languages?
- Which parameters used with the classifiers cause a significant difference?
- Is there a relation between programming language families and languages that are often mistaken for each other by the classifiers?
- What is the best classifier to differentiate between two given programming languages?
- Can the answer on the previous question help with identification of programming languages in files with multiple languages.

3.1 Classifier evaluation

In total all the classification methods and their varying parameters resulted in 348 different classifiers. The F1 measure of each of these classifiers can be seen in Figure 3.1. The classifier with the highest F1 measure is the Modified Kneser-Ney discounting classifier using the all-symbols tokenizer, 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 F1 measure of 0.969. Furthermore, the results show that, with almost all classifiers, having a larger dataset gives better results. Out of 174 classifiers, only two gave better results with a small dataset, both of which were the Normalized Compression Distance classifiers (both GZIP and ZLIB).

Figure 3.1: F1 Measure of all classifiers



The size of the symbol shows which training set was used: large symbols used the large training set and small symbols the small training set.

3.2 Parameter evaluation

To find which parameters cause a significant difference in the F1 measure multiple one-way ANOVAs were used with each parameter as a separate predictor variable. The ANOVA (ANalysis Of VAriance) is a statistical test that determines if the difference of the means of two or more groups is significant. For an ANOVA to work the data must be normally distributed. Since the data from the classifiers is not normally distributed, it was transformed using a rank transformation. Each parameter that had more than two categories (n-gram size and tokenizer) was also subjected to a Tukey's range test, to see find if there was a significant difference between the categories belonging to that parameter. Training set size, tokenizer, n-gram size and OOV all significantly affect the F1 measure (see table 3.1).

Table 3.1: One-way ANOVA on parameters.

Parameter	P-value
Training set size	**0.00**
n-gram size	**0.00**
2 - 1	0.00
3 - 1	0.00
4 - 1	0.00
5 - 1	0.00
3 - 2	*0.72*
4 - 2	*0.98*
5 - 2	*1.00*
4 - 3	*0.95*
5 - 3	*0.83*
5 - 4	*1.00*
Tokenizer	**0.00**
AllSymbols - AllSymbolsLowerCase	*0.99*
AlphaNumeric - AllSymbols	0.00
AlphaNumeric - AllSymbolsLowerCase	0.00
AlphaNumeric - AllSymbolsLowerCase	0.00
AlphaNumericLowerCase - AllSymbolsLowerCase	0.00
AlphaNumeric - AlphaNumericLowerCase	*0.95*
OOV	**0.00**

Parameters in *italic* show no significant difference.

3.3 Best classifier per language

Now it is known which classifier gives the best results for all the 20 programming languages, but what if only a specific language needs to be classified? By

calculating and comparing the accuracy of all classifiers for a given language L, the best classifier for correctly classifying the language L can be found. The result for each language can be found in table 3.2.

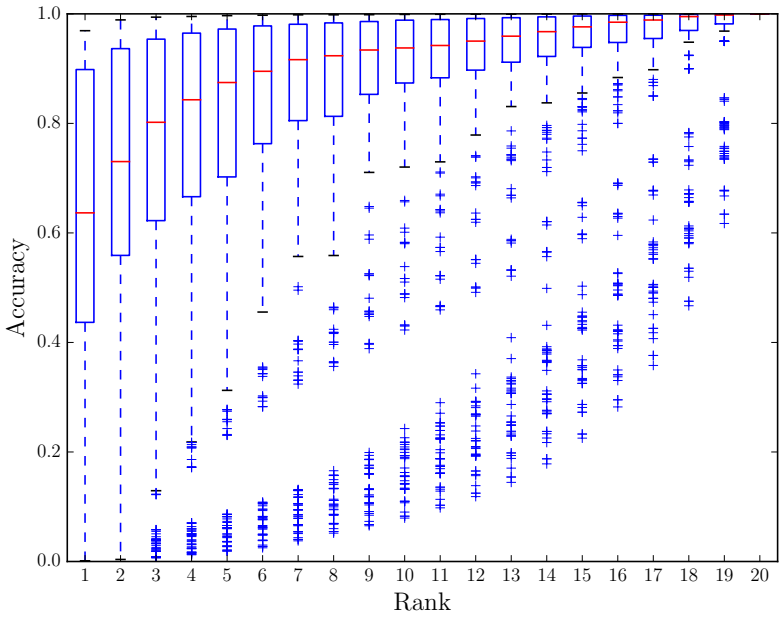
Table 3.2: Best classifier per language

Language	Model	Tokenizer	N	Training	OoV	Accuracy
C	Naive Bayes	AllSymbols	2	large	N/A	0.975
C#	Good-Turing	AllSymbols	3	large	Skip	0.99
C++	Skip gram	AllSymbols	3	large	Skip	0.865
CSS	Naive Bayes	AllSymbols	3	large	N/A	0.995
Clojure	Good-Turing	AllSymbols	3	large	Skip	0.99
Go	Skip gram	AllSymbols	3	large	Skip	0.995
HTML	Naive Bayes	AllSymbols	3	large	N/A	0.975
Haskell	Naive Bayes	AllSymbols	2	large	N/A	0.995
Java	Witten-Bell	AllSymbols	2	large	Skip	1
JavaScript	Modified Kneser-Ney	AllSymbols	3	large	Skip	0.98
Lua	Naive Bayes	AllSymbols	2	large	N/A	0.975
Objective-C	Good-Turing	AllSymbols	3	large	Skip	0.985
PHP	Good-Turing	AllSymbols	3	large	Skip	0.985
Perl	Naive Bayes	AllSymbols	2	large	N/A	0.995
Python	Good-Turing	AllSymbols	3	large	Skip	0.99
R	Good-Turing	AllSymbols	3	large	Skip	0.995
Ruby	Good-Turing	AllSymbols - LowerCase	3	large	Skip	0.98
Scala	Modified Kneser-Ney	AllSymbols	2	large	Skip	0.995
Scheme	Naive Bayes	AllSymbols	3	large	N/A	0.99
XML	Naive Bayes	AllSymbols	3	large	N/A	0.96

3.4 Effects of looking at lower ranks

To classify the files, the classifiers calculated the chance that a file belonged to each language. With these chances, an ordered list of rankings could be made for all the tested files. These rankings can be used to see the increase in accuracy of a classifier when not only looking at the highest ranking language. This essentially transforms the classifiers into recommenders, which can look at a file and then recommend the most likely languages the file belongs to. The difference between looking only at the highest ranking language and looking at lower ranks as well can be shown by counting how many more files would have been correctly recommended, when including lower ranking languages. This number can be used to calculate the increase in accuracy per rank per classifier method. The results are shown as a box plot in Figure 3.2.

Figure 3.2: Accuracy per rank



3.5 Looking at language families

Besides looking at which classifier is the best, the data can also be used to try and detect language families. A language that classifiers often wrongly classify as a different language might be related to the other language. For example, since the language C is more or less a subset of the language C++, one could assume that classifiers might have issues with correctly seeing the difference between C and C++. By looking if classifiers often incorrectly classify files of the language A as being a file from language B, it could be possible that languages A and B are closely related. Table 3.3 shows the percentage of files per language that are classified as a certain language. Applying a heat map to table 3.3 shows languages that are often incorrectly classified as each other, which can show possible language families.

Table 3.3: Percentage of files classified as a certain language

Actual language \ Classified as	C	C#	C++	CSS	Clojure	Go	HTML	Haskell	Java	JavaScript	Lua	Objective-C	PHP	Perl	Python	R	Ruby	Scala	Scheme	XML
C	48	4	16	7	0	2	1	0	4	2	0	5	2	1	1	1	4	1	1	1
C#	2	62	3	4	0	3	1	1	8	2	0	8	1	0	0	1	2	1	1	1
C++	16	5	45	6	1	2	1	0	5	2	0	6	2	0	1	1	4	1	2	1
CSS	1	3	4	71	0	1	1	0	1	0	0	4	2	1	5	0	3	1	1	0
Clojure	1	3	3	4	60	2	1	1	2	1	0	10	2	0	1	2	3	1	3	1
Go	1	5	2	6	0	64	1	0	3	3	0	7	1	0	0	1	2	2	0	1
HTML	1	5	1	1	0	2	68	0	1	3	1	4	4	0	0	1	2	1	0	3
Haskell	1	5	9	6	1	2	1	52	1	1	1	6	3	0	1	1	4	2	1	1
Java	1	5	3	4	0	2	1	0	70	1	0	5	1	0	0	1	2	4	0	0
JavaScript	1	5	10	3	1	3	2	1	4	48	1	7	5	0	0	1	4	1	1	0
Lua	1	7	3	2	0	2	0	1	1	2	55	7	3	0	2	2	7	3	1	1
Objective-C	4	2	3	6	0	3	1	1	2	1	1	65	3	0	1	2	3	1	1	0
PHP	1	4	5	4	0	2	4	1	2	1	0	5	63	1	1	1	3	1	1	0
Perl	0	3	6	6	0	1	0	0	2	0	0	8	4	64	0	1	3	1	0	0
Python	0	4	6	6	0	2	1	1	3	1	2	12	2	1	49	2	6	2	0	0
R	1	4	4	4	1	2	1	1	2	2	1	5	4	1	1	56	6	2	1	1
Ruby	0	4	1	2	0	2	1	4	1	1	3	7	1	1	2	2	65	3	0	1
Scala	0	2	1	6	1	2	1	1	10	1	1	9	1	0	1	1	4	57	0	1
Scheme	1	4	3	3	4	2	1	1	2	1	1	6	4	0	1	1	4	1	59	1
XML	0	2	2	1	1	2	10	3	3	0	0	5	2	1	1	1	6	3	0	56

Table 3.3 shows that many files are classified as Objective-C. This is probably caused by one of the weaker classifiers, which classifies almost all files as Objective-C. Since weak classifiers will usually classify files wrongly, even if the actual language is not related to the predicted language, the result might be better if these classifiers are left out. In table 3.4 only classifiers with a F1 measure higher than 0.9 were used.

Table 3.4 shows a possible relationship between the languages C and C++; both languages were mistaken for the other fifteen percent of the time. Besides

Table 3.4: Percentage of files classified as a certain language with F1 measure higher than 0.9

Actual language \ Classified as	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

those languages no other relationship is seen. Four percent of HTML files are classified as JavaScript files, but only one percent of JavaScript files are classified as HTML files. Also three percent of XML files are found to be HTML files, but it is only one percent the other way around.

Section 3.4 showed the possible impact of looking at lower ranked languages. This can also be applied in the search for language families. Table 3.5 shows all classifiers with a F1 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. For example Java seems to be related to C#, C++ and Scala. Also JavaScript is ranked highly among almost every language.

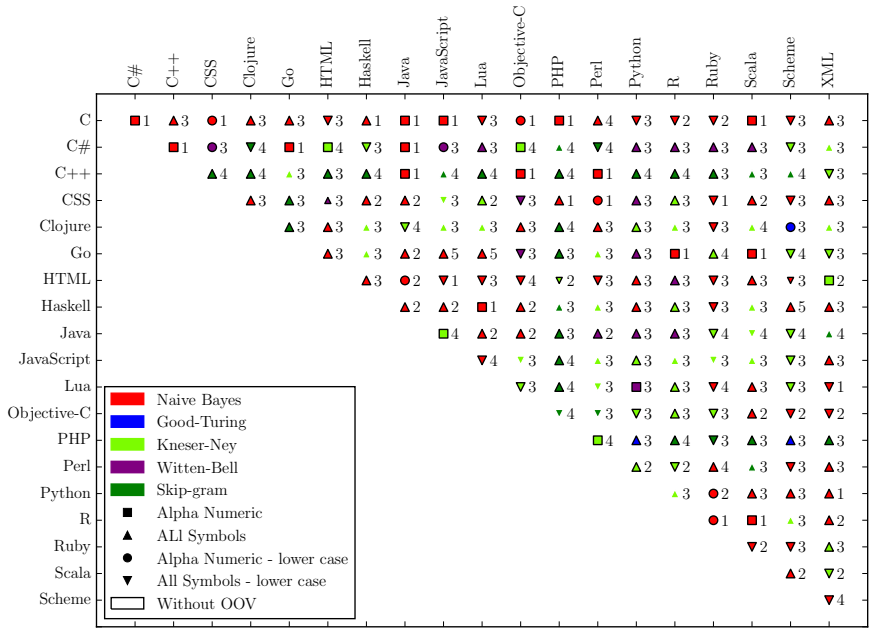
Table 3.5: Percentage of files ranking in top 5 as a certain language with F1 measure higher than 0.9

		Classified as																		
		C	C#	C++	CSS	Clojure	Go	HTML	Haskell	Java	JavaScript	Lua	Objective-C	PHP	Perl	Python	R	Ruby	Scala	Scheme	XML
Actual language	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

3.6 Best classifier between two languages

Section 3.5 shows that classifiers occasionally make mistakes differentiating between two languages. This is especially prevalent between the languages C and C++, as seen in table 3.4 and 3.5. This raises the question which classifier is the best at differentiating between two specific languages. To answer this question classifiers had to be scored. The score is calculated for a classifier C differentiating between languages A and B. For each file belonging to language

Figure 3.3: Best classifier per language combination



The number next to each symbol denotes the n-gram size. The size of the symbol shows which training set was used: large symbols used the large training set and small symbols the small training set.

A we looked at what rank language B was predicted to be, according to the method described in 3.4. The inverse of these ranks was combined. The same was done for files belonging to language B. The results of both scores was combined as well. This gives a score which is lower if the classifier C does better in differentiating between languages A and B. Since some classifiers have a very bad accuracy, it is important to also take the actual accuracy of the classifier in consideration. For this reason the accuracy of the classifier was added to the score. This means the accuracy of the classifier is weighted more heavily than the actual differentiating power, because bad classifiers should be avoided. With this scoring method each classifier could be rated for each language combination. The results are shown in figure 3.3

3.7 Detecting embedded code

To detect which piece of code belongs to what language in a document, the data from section 3.6 is very useful. Five source code files of well known websites were collected. These files are labeled as HTML files, but there is JavaScript embedded within the HTML. To find out what code is JavasScript and what

code is HTML, each line of code was classified using the Naive Bayes classifier, with n-grams of length 1 and the all-symbols lower case tokenizer (which is the best at differentiating JavaScript and HTML, see section 3.6). This resulted in 3187 lines that were correctly classified out of 3605 lines of code, 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.

Chapter 4

Discussion

The following section contains:

- The answers to the research questions through analysis of the results.
- Threats to the validity of the research.
- Possible future work based on the research.
- The conclusion.

4.1 Analysis of results

The comparison of the different classifying methods showed that Modified Kneser-Ney discounting with the all-symbols tokenizer, n-gram size of 3, trained on a large dataset and skipping OOV is the best all round classifier. It is very closely followed by other language models, except for NCD, which scores much lower. The Naive Bayes language model scores very well with most parameters, as long as it has a big enough training set. Since the Naive Bayes language model requires less complex computations this model should be preferred if performance is an issue.

The all-symbols tokenizer (both original case and lower case) scores significantly higher than the alpha-numerical tokenizer. This means that it is better to look at all symbols in classifying programming languages, unlike natural languages, where punctuation and other non alpha numeric symbols are usually ignored. The difference between original case and lower case is not significant for both the all-symbol and alpha-numerical tokenizers. The methods that use a n-gram size of 1 are generally worse than the methods that use higher n-gram size. There is significant difference between a n-gram size of one and two or higher. However there is no significant difference between n-gram sizes larger than one, so in general n-gram sizes of two should be used, because they require less computational power and storage. Skipping OOV words scores a bit higher

than its counterpart and shows a significant difference. Finally a larger training set always gives better results than a smaller training set, except for NCD.

There is no evidence that there is a family relationship between languages that are often mistaken between each other. Only C and C++ are often classified as the same, while being in the same language family. Other found relationships are related to embedded languages instead of language families, for example with JavaScript and HTML. Looking at lower ranked languages shows many more possible relationships, but many of those combinations do not belong to the same language family.

The best classifiers between two languages differ greatly. Out of the 190 language combinations, only 7 combinations are best classified by the best all-round classifier, found in section 3.1. The Naive Bayes language model with the all-symbols tokenizer, n-gram size of 3 and trained on the large dataset is the classifier that occurs most often with 20 combinations. In total 42 different classifiers are seen when searching for the best classifier between two languages.

Detecting embedded languages with a specifically chosen classifier worked decently, with an accuracy of 0.88. However it was tested on a rather small sample size and with only one type of language combination (HTML and JavaScript)

4.2 Threats to validity

There was a selection bias in the selection of programming languages. If there is a large difference between the tested languages, it is easier to differentiate between the languages and it could be easier for the classifier to identify a certain language.

The main language of a file was based upon the language GitHub assigned to it. Since the files were not manually verified it is possible that GitHub also misclassified some files. Also some files may belong to a certain language, but because of embedded languages, the file might contain more code of another language. This could happen for example with PHP and HTML, where a file is a PHP file, but it is classified as HTML, because it exists mostly out of HTML code.

There is a threat to the construct validity, since there might be situations where the tested classifiers would not be able to differentiate between languages. For example a file that contains many keywords of another language in comments. This file belongs to a certain language, but the classifiers might wrongly classify it as the language found in the comments.

The classifiers were trained and tested only on projects from GitHub, which means they were only trained and tested on open source projects. It is possible that the classifiers perform differently on closed source projects.

4.3 Future work

One possible area of future work is adding more classifiers and different parameters. By comparing with more classifiers the data will be more accurate representation of what classifier is the best for a specific job. Different parameters can show possible improvements to classifiers and other differences between natural and programming languages. One language model that especially needs research are SVMs, since these generally have very strong results with natural languages. It might be possible that SVMs have an even higher accuracy than what is currently found. Besides SVMs more compressors could be tested for NCD, since the tested compressors were found to be lacking. A parameter that could be looked in to is the removal of comments. Since comments are not actual source code, these might confuse the classifiers. Removing comments, could increase the accuracy of the classifiers.

Some of the classifiers have really good accuracy, so good, that only a couple of files are misclassified. Looking into these files to find why they were misclassified, could lead to an even higher accuracy. This is also to make sure there are not some edge cases in which the classifier always fails.

The tested classifiers have proven that they are able to identify programming languages and maybe even multiple languages per file, but it is not clear if they can identify source code in a file combined with natural language. There are many texts where snippets of code are included, but not actively labeled. It might be possible to use the classifiers to automatically detect those snippets. This can be combined with the research done by Merten et al. [19].

Improving the different applications of the dataset is another area. For instance, finding the best classifier between two languages was tested with classifiers that were trained on all 20 languages. Maybe the results would be different if the classifiers were only trained on the two languages that are being looked at. More work is also possible with the research on detecting embedded languages. It was tried to detect the language of each line of a file, but it is also possible that there are multiple languages embedded in one line (this happens often with in-line styling in HTML with CSS). Another possibility would be that there are more than two languages in one file, which might require another classifier.

Finally the data can be adjusted. There are many different programming languages and adding more might change the results of the thesis. Looking into different versions of languages could also be very interesting, especially when different versions are not backward compatible, like python 2 and python 3. Other research has shown that adding more training data will keep increasing the accuracy in NLP [23]. Using more source code files, for example all of GitHub, for training the classifiers could result in an higher accuracy.

4.4 Conclusion

A dataset has been presented which contains 348 different programming language identification methods and their accuracy. Some of these methods have

shown very good accuracy with the highest having a F1 measure of 0.969. Using this dataset other questions could be answered as well. All parameters showed significant differences, although some options can be left out. No evidence was found that shows a relation between language families and language that are often mistaken for each other. By using the best classifier between two given languages, one can try to detect which lines of a file are of what language. Future work can test more advanced classifiers or apply the information presented in this thesis.

Chapter 5

Bibliography

[1] Juriaan Kennedy van Dam and Vadim Zaytsev. Software Language Identification with Natural Language Classifiers. In Katsuro Inoue, Yasutaka Kamei, Michele Lanza, and Norihiro Yoshida, editors, *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering: the Early Research Achievements track (SANER ERA)*, 2016. In print.

[2] Timothy Baldwin and Marco Lui. Language Identification: The Long and the Short of the Matter. *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, (June):229–237, 2010.

[3] Rudi Cilibrasi and P.M.B. Vitanyi. Clustering by Compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.

[4] Thomas Gottron and Nedim Lipka. A comparison of language identification approaches on short, query-style texts. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5993 LNCS:611–614, 2010.

[5] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, March 2002.

[6] David Binkley. Source Code Analysis: A Road Map. *Proc. FOSE*, pages 104–119, 2007.

[7] Joshua Peek, Arfon Smith, Ted Nyman, Paul Chaignon, Adam Roben, and Brandon Keepers. Linguist. https://github.com/github/linguist.

[8] Alex Gorbatchev. SyntaxHighlighter. http://alexgorbatchev.com/SyntaxHighlighter/.

[9] Ben Boyter. SearchCode. https://searchcode.com/.

[10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[11] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. *Machine Learning: ECML-98*, 1398(2):137 – 142, 1998.

[12] Zhi-hong Deng, Shi-wei Tang, Dong-qing Yang, Ming Zhang Li-yu Li, and Kun-qing Xie. A Comparative Study on Feature Weight in Text. *APWeb 2004*, pages 588–597, 2004.

[13] Yaoyong Li, Kalina Bontcheva, and Hamish Cunningham. Adapting SVM for data sparseness and imbalance: a case study in information extraction. *Natural Language Engineering*, 15(1):241, 2009.

[14] Jason Upchurch and Xiaobo Zhou. First byte: Force-based clustering of filtered block N-grams to detect code reuse in malicious software. In *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, pages 68–76. IEEE, October 2013.

[15] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, 2(1):2–3, 2004.

[16] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Source code author identification based on N-gram author profiles. *IFIP International Federation for Information Processing*, 204:508–515, 2006.

[17] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings - International Conference on Software Engineering*, volume 20, pages 837–847, 2012.

[18] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 1–13, 2014.

[19] Thorsten Merten, Bastian Mager, Simone Bürsner, and Barbara Paech. Classifying Unstructured Data into Natural Language Text and Technical Information. *MSR 2014: Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 300–303, 2014.

[20] Secil Ugurel, Robert Krovetz, C. Lee Giles, David M. Pennock, Eric J. Glover, and Hongyuan Zha. What's the code? Automatic Classification of Source Code Archives. *Proceedings of the Eighth Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, 2:632 – 638, 2002.

[21] David Klein, Kyle Murray, and Simon Weber. Algorithmic Programming Language Identification. *Computing Research Repository*, page 11, 2011.

[22] Jyotiska Nath Khasnabish, Mitali Sodhi, Jayati Deshmukh, and G. Srinivasaraghavan. Detecting Programming Language from Source Code Using Bayesian Learning Techniques. In *Machine Learning and Data Mining in Pattern Recognition, Mldm 2014*, volume 8556, pages 513–522. 2014.

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

[24] Am Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. *In AI 2004: Advances in Artificial Intelligence*, pages 488–499, 2005.

[25] Andres McCallum and Kamal Nigam. A Comparison of Event Models for Naive Bayes Text Classification. *AAAI/ICML-98 Workshop on Learning for Text Categorization*, pages 41–48, 1998.

[26] Marco Lui and Timothy Baldwin. Cross-domain Feature Selection for Language Identification. *Proceedings of the 5th International Joint Conference on Natural Language Processing*, (1967):553–561, 2011.

[27] 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* -, number August, pages 310–318, Morristown, NJ, USA, 1996. Association for Computational Linguistics.

[28] Andreas Stolcke. SRILM – An extensible language modeling toolkit. *Proc. (ICSLP 2002)*, 2:901–904, 2002.

[29] David Guthrie, Ben Allison, and Wei Liu. A closer look at skip-gram modelling. *Proceedings of the 5th international Conference on Language Resources and Evaluation (LREC-2006)*, pages 1222–1225, 2006.