Source Code Plagiarism Detection using Machine Learning

Utrecht University

Daniël Heres

August 2017

Contents

Abstract

In this thesis we study how we can apply machine learning techniques to improve source code plagiarism detection. We present a system, InfiniteMonkey, that can identify suspicious similarities between source code documents using two methods. For fast retrieval of source code similarities, we use a system based on n-gram features, tf-idf weighting and cosine similarity. The second part focuses on applying more complex neural network models trained on a large synthetic source code plagiarism dataset to classify source code plagiarism. This dataset is created using an automatic refactoring system we developed for learning this task. The methods are evaluated and compared to other tools on a number of different datasets. We show that the traditional approach compares well against other approaches, while the deep model on synthetic data does not generalize well to the evaluation tasks. In this thesis we also show a simple technique for visualization of source code similarities.

Chapter 1

Introduction

Learning programming skills often requires a lot of practice, time and effort. For some programmers it becomes tempting to cheat and submit the work of another person. This may occur when they feel they can not solve a task before the deadline, or don't want to invest the time required to complete the task. Students can also work together "too much", and partially re-use the work of each other when this is forbidden. Plagiarism is often considered unwanted, and universities and other organizations have rules in place to deal with plagiarism. Detecting cases of plagiarism when working with large groups quickly becomes infeasible. The number of unique pairs that could contain plagiarism grows quadratically with the number of submissions n using this formula:

$$\text{number-of-pairs}(n) = \frac{n \times (n-1)}{2}$$

. The number of pairs to compare grows very quickly, as is visible in Figure 1.1. Without support from plagiarism detection tools, identifying plagiarism takes too much work, especially in big classes, in MOOC environments and in courses that span multiple years.

Source code plagiarism detection tools work as follows: they take in a collection of source code documents, and sort all unique pairs according to their measured similarity. Tools can also give an explanation of this ordering, by visualizing the similarities in the two programs. An abstract view of this task can be seen in Figure 1.2.

In an earlier project we performed a comparison of a number of existing source code plagiarism detection tools. While the tools often work pretty well, it occurred to me that a lot of tools fail to incorporate domain knowledge and/or remove useful information from the source code which results in a bad ranking. In a certain programming task and programming language, some similarities should be expected (e.g. in import statements and language keywords), while other similarities should be reason to report a high level of similarity. The frequency of those patterns could be used to improve the results of a plagiarism detection system.

Figure 1.1: Relation between number of submissions and number of pairs to check for plagiarism

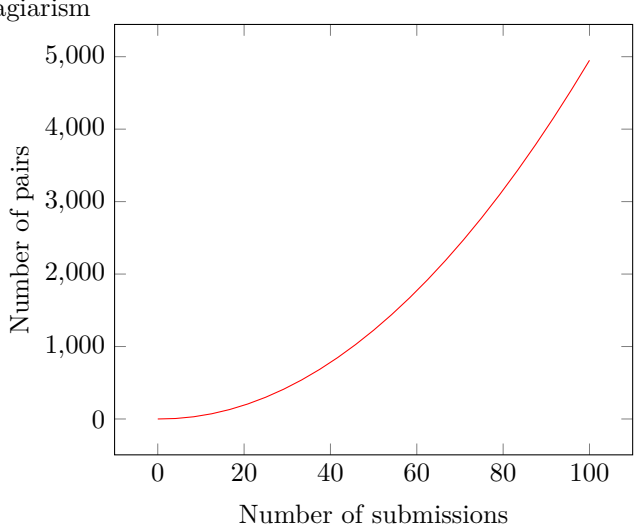


Figure 1.2: Source Code Plagiarism Task

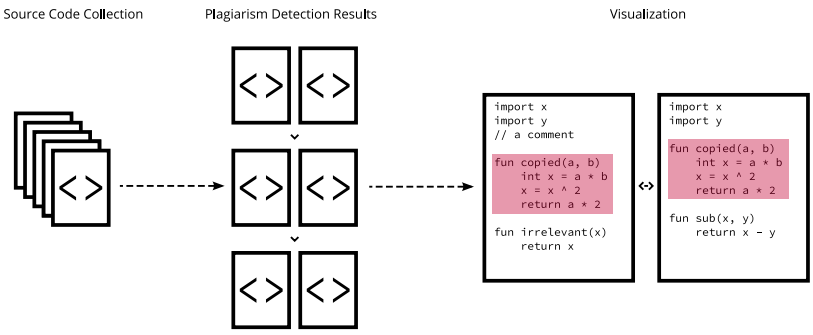


Figure 1.3: Moss document comparison view

Besides differences in detection performance, some tools also were more helpful than other programs: Moss for example shows for each tool exactly *where* between the two documents are the highest similarities. This can make visual identification of plagiarism a lot easier, as can be seen in a files comparison by Moss in Figure 1 . The same information that could be used to improve the quality of similarity detection can also be used to improve the visualization of source code similarity. We could not only show that certain parts are similar, but what features contribute the most.

1.1 Formal Description

Given a set of documents D, we want to calculate the similarity matrix (or distance matrix) between the elements of D. The similarity between two source files is a binary value in the case of plagiarism detection: either plagiarism can be identified or not. For each set of programs D there exists a ground truth G_D which consists of the triplets:

$$G_D = \{\, (d_1, d_2, s) \,|\, d_1 \in D, d_2 \in D, s \in \{0, 1\} \,\}$$

The ground truth is a (human) annotated dataset of pairs which are considered either positive or negative. The ground truth contains the set of cases of plagiarism pairs

$$G_D{}^+ = \{\, (d_1, d_2) \,|\, (d_1, d_2, s) \in G_D, s = 1 \,\} \tag{1.1}$$

and the set of pairs where no plagiarism could be identified.

$$G_D{}^- = \{\, (d_1, d_2) \,|\, (d_1, d_2, s) \in G_D, s = 0 \,\} \tag{1.2}$$

The problem is to create or learn a function that predicts a similarity $S_E : (d_1, d_2) \rightarrow \mathbb{R}$ such that most, preferably all, pairs in $\{\, S_E(d_1, d_2) \,|\, (d_1, d_2) \in G_D{}^+ \,\}$ result in a higher value than those in $\{\, S_E(d_1, d_2) \,|\, (d_1, d_2) \in G_D{}^- \,\}$. This function returns a measure of relative or absolute similarity between two

documents. We measure the performance of this function on this set using a performance metric P that computes a score using the sequence of ordered similarities and the ground truth G_D. In this thesis we use the average precision as metric for comparing the quality of the results. This computes the average for the precisions at each correctly predicted example.

1.2 Thesis Overview

This thesis starts in chapter 2 with an overview of tool comparisons, tools and methods. In chapter 3 we will define our research goals and goals of the plagiarism detection tool InfiniteMonkey. Also we show what dataset we use, how we model and train models and how we evaluate and compare tools against each other. In chapter 4 we will show a text retrieval based baseline, using n-grams, tf-idf weighting, cosine similarity and a grid search on hyperparameters. In chapter 5 we present a learned similarity model on synthetic data based on code Github repositories. We show how we apply random refactorings to Java code using a refactoring tool. We then show a neural network model and how it performed on a hold out set of this data. In chapter 6 we show how we use the tf-idf weighted features in combination with cosine similarity to produce visualizations of source code similarity. In chapter 7, a large number of plagiarism detection tools are evaluated and compared using average precision scores on 9 different sets. We evaluate the results of those tools. Chapter 8 discusses our contributions, lists limitations and future research. Finally in chapter 9 we list our conclusions.

Chapter 2

Literature Overview

In this chapter I give a literature overview of research and tools for plagiarism and source code similarity detection.

2.1 Tool Comparisons

In the evaluation of Hage [1] and others five Java tools are compared by comparing the 10 features of the tools, by sensitivity to different kinds of changes in source code (refactorings) and by comparing the results of the top 10 of each of the tools to the top 10 results of others. Those top 10 results are classified as being either plagiarism, a false alarm, similar, a resubmission, self compare (tool found code in same submission). The difference between similar and false alarm in this study is that in the case of the qualification similar the code may have some similarity but is found to be no case of plagiarism, while pairs qualified as false alarm are not even similar.

In the work of Flores and others [2] [3] a binary plagiarism detection task is presented. The dataset is based on data from Google Code Jam in 2012. The resulting SOCO dataset includes a training dataset on which some testing/-training can be done. The classification of this dataset is based on the results of the tool JPlag. The submissions for the SOCO challenge are compared based on relevance metrics like the F_1-score.

In the evaluation of Modiba and others [4], 12 plagiarism detection tools are compared against each other based on a set of features. The 5 tools that support C++ code and are freely available, namely AC, CodeMatch, CPD, Moss and NED are compared by the quality of the results. The results are reported by percentage of agreement, false positives and false negatives. The tool NED resulted in the result closest to the ground truth as annotated by the authors. The other tools also gave reasonable results. The authors recommend Moss for more advanced courses as it detects similarities between two documents that are not present in other documents: when a sequence occurs in more documents, those are not considered as similarity.

2.2 Tools and Methods

2.2.1 Moss

Moss [5] is a tool developed to detect source code similarity for the purpose of detection of software plagiarism. It is available as a web-service: documents can be uploaded using a script and the results are visible through a web interface after processing. Moss uses character level n-grams (a contiguous subsequence of length n) as features to compare documents. Instead of comparing all n-grams, only some of the features are compared for reasons of efficiency. A commonly used technique to select textual features is to calculate a hash value for each feature but selecting only a subset of those features using $0 \bmod p$ for a fixed p. The authors observe that this technique often leaves gaps in the documents, making the probability of missing matches between documents higher. To prevent this from happening they use an algorithm they call *winnowing*: instead of randomly selecting n-grams from the document, they select for each window at least one feature. Furthermore they use a large value for n to avoid noisy results and remove white space characters to avoid matching on white space.

2.2.2 JPlag

JPlag [6] is a tool to order programs by similarity given a set of programs. The authors argue that comparing programs based on a feature vector alone throws too much away of the structural similarity. Instead they try to match on what they call *structural* features. Instead of using the text directly they convert the Java source code first to a list of tokens, such as BEGINCLASS, ENDCLASS and BEGINMETHOD and ENDMETHOD. Then, they use an algorithm to find matches between documents using the list of tokens, from the largest to the smallest matches. There is some parameter for the minimum size of matches, otherwise small matches would occur too often. They apply a few runtime optimizations to the basic comparison algorithm with worst case $O(n^3)$ time complexity (where n is the size of the documents) using the Karp-Rabin [7] algorithm. They compare different cut-off criteria using different methods to create a threshold value given the similarity values on a dataset. They also compare the influence of the minimum match length and the set of tokens to use by performing some measurements on datasets. They also show some possible attacks against JPlag.

2.2.3 Sherlock

Sherlock [8] is a simple C-program to sort text document pairs like source code according to their similarity. The program first generates signatures. While generating the signature, it drops whitespace characters and drops a fraction of the other characters from the text file in a somewhat random fashion. Finally all the signatures are compared against each other.

2.2.4 Plaggie

Plaggie [9] is another tool that supports checking for similarities between Java source code documents. It works similar to JPlag. At the time of publication the main differences were that Plaggie was open source and could be run locally. Currently, JPlag is both open source and can also be run locally.

2.2.5 SIM

SIM [10] [11] is a software and text plagiarism detection tool written in C. It works by tokenizing the files first and searching for the longest common subsequence in the file pairs.

2.2.6 Marble

Marble [12] is a tool that is developed with simplicity in mind. The tool consists of three phases: normalization, sorting and detection. The normalization phase converts source code from raw text to a more abstract program. Keywords like `class`, `extends`, `String` are maintained, names are converted to `X` and numeric literals to `N`. Operators and symbols are also left in place. Import declarations are discarded in this transformation. After normalization, classes and class members are sorted lexicographically, this makes the tool insensitive to reordering these program constructions. The simplified program finally is compared by the Unix line-based diffing tool `diff` using the number of changed lines normalized by total length of the two files.

2.2.7 GPlag

GPlag [13] uses a program dependence graph (PDG) analysis to measure similarity between two programs. The idea of this is that the dependencies between program parts often remain the same, even after refactoring of code. After the programs are converted to a PDG, all subgraphs larger than some "trivial" size are tested on graph isomorphism relaxed with a relaxation parameter $\gamma \in (0, 1]$. Graph G is said to be γ-isomorphic to G' if the subgraph $S \subseteq G$ is subgraph isomorphic to G' and $|S| \geq \gamma|G'|$. To avoid testing every pair of sub-graphs, they reject pairs of graphs that don't have similar histograms of their vertices.

2.2.8 Evolving Similarity Functions

Wu and others [14] suggest to use a genetic approach for learning similarity functions. The first contribution consists of finding good parameters for the OkapiBM25 similarity function using Particle Swarm Optimization. They find that the default parameters are suboptimal for plagiarism detection, and can be optimized using a genetic optimization algorithm. They show the parameters result in about the same performance as JPlag, the differences are not too big. The second contribution is learning a similarity function by optimizing a similarity function using a Genetic Programming approach. The genetic algorithm

creates new programs based on a pool of the best current functions. The possible functions are restricted by a simple grammar supporting for example addition and multiplication and some terminals like within-document term frequency, within-query term frequency and document length. They add a penalty for the size of the similarity functions. They optimize three different fitness functions. The resulting functions perform worse compared to the other functions, which the authors suggest may be due to the functions returning negative similarity scores as well as overfitting to the training set.

2.2.9 Plague Doctor, Feature-based Neural Network

The plagiarism detection system Plague Doctor described in [15] uses features derived from the source code as input for a Neural Network model. The model also can output the relative importance of each feature. It uses 12 numerical features such as: result from Moss, comment similarity, ratio of misspelled words, etc. Using the features, they train a classifier using a small Neural Network with 7 hidden units. When analyzing the weights in the connections of the neural network, the relative importance of each feature can be identified. After learning, more important features will have higher weights for their connections than features of lower importance. The output from Moss has the highest normalized weight of 0.2390, followed by the ratio of misspelled comments (0.1418) and string literal similarity (0.1147). The F-measure on the hold-out set is higher than using Moss alone, but the authors say this could in part be explained by the fact that the system is trained on that particular dataset.

2.2.10 Callgraph Matching

Callgraph Matching is the method used by [16] for a plagiarism detection tool for the Haskell programming language. The program first converts each program into call graphs and preprocesses these graphs. Then both an edit distance algorithm using A* search and a subgraph isomorphism algorithm are used to find matches for all subtree pairs. They compare the total similarity scores after applying different kinds of source modifications against the token-based tools Holmes and Moss, which use character level n-grams combined with a fingerprinting technique. The comparison of a set of 59 programs took about a day on a desktop PC (from 2011 or before). The program can make use of multiple threads and multiple machines to speed up the comparison.

2.2.11 Holmes

Holmes [17] is a plagiarism detection tool for Haskell programs. A number of different techniques are used by the tool for comparison of two programs: a fingerprinting technique similar to Moss, a tokenstream and three different ways of comparing degree signatures: based on Levenshtein edit distance of the two vertice and Levenshtein distance of two vertices combined with position. Local functions are not considered for the call graph. Holmes performs a reachability

analysis and removes code that is not reachable from the entry point to some extent are removed as well. Comments are entirely removed in the transformation and most identifiers. Template code can be added by a teacher to prevent matching on code that students are permitted to use.

2.2.12 DECKARD Code Clone Detection

DECKARD[18] uses a tree based method for detecting code clones within a software project. They use what they call *characteristic vectors* to capture information about trees in the program which consists of the element-wise sum of occurrences of types of nodes in the program part, where each node is an instance of the AST (Abstract Syntax Tree) of the program. To prevent matching on small vectors, a minimum token count can be specified (a minimum ℓ_1-norm). Occurrences of some nodes are not included in the vector, such as `[`, `]`, `(` and `)`. Furthermore, they use a sliding window approach to merge program subtrees. Similar vectors are clustered based on their euclidean distance. They use Locality Sensitive Hashing to map similar vectors to equal hash values with high probability.

Chapter 3

Research Questions

In this thesis, we want to answer the following questions:

- Can we achieve better results than existing tools for plagiarism detection tasks and source code similarity using machine learning techniques?
- How can we visualize the similarities between two programs and how does this compare to e.g. the Unix `diff` tool and MOSS? [5]

By evaluating existing literature and experimenting with techniques from Machine Learning and Information Retrieval domains, we will provide empirical evidence whether plagiarism detection using techniques from these domains can result in improved detection performance.

3.1 Plagiarism Detection Tool goals

In this research, we will develop a tool *InfiniteMonkey*, that can be applied on a large range of programming tasks, languages and dataset sizes. While developing and experimenting with this tool, we will keep the following goals in mind:

- *Detection Performance*: The detection performance should be better than that of other tools, and we should be able to demonstrate this on a variety of tasks.
- *Visualization*: To be usable as plagiarism detection tool, it is important to show the locations of the similarities in each program pair or the source of other features. We should be able to visualize the influence of certain features in the document pairs to make comparing easier.
- *Generalization*: It is important that a model trained on one or a few datasets will generalize to other datasets (as they may have a very different distribution of features). Features occurring in one dataset may be absent or occur with a different frequency in another dataset.

Dataset	origin	annotated as similar	annotated as dissimilar	total nr. files
a1	SOCO	54	n/a	3241
a2	SOCO	47	n/a	3093
b1	SOCO	73	n/a	3268
b2	SOCO	35	n/a	2266
c2	SOCO	14	n/a	88
mandelbrot	UU	111	85	1434
prettyprint	UU	9	135	290
reversi	UU	117	83	1921
quicksort	UU	81	94	1353

Table 3.1: Dataset Characteristics

- *Language Support*: Porting the tool to other languages should be relatively easy. This will mean that very language-specific approaches will not be used. Whenever some language-specific features are added to the model, they should not be too hard to develop for other languages, or these should not degrade the detection performance when evaluated on other programming languages.
- *Runtime Performance*: Finding plagiarism in a set of source code documents should not take days, but preferably less than an hour. Besides the inconvenience for the user, slow performance makes evaluation of the tool and hyperparameter optimization harder. However, improvements in detection performance may justify a higher runtime.

3.2 Research Approach

In this section we show how we try to answer our research questions and how I plan to build the plagiarism detection tool InfiniteMonkey.

3.2.1 Dataset

For building, validating and testing our models we need a few datasets with an annotated ground truth to optimize and evaluate our models. During a previous experiment we used data from the SOCO [19] challenge and manually annotated a few datasets from UU courses. During this thesis we extended this set with one extra dataset. The characteristics (name, origin of data, nr. similar, nr. dissimilar, total number of files) of these datasets are listed in table 3.1. In both the SOCO and UU sets, documents can occur in multiple pairs because multiple people may re-use code from another document. In the SOCO datasets, only the number of documents annotated as similar is given. The total number of documents annotated as similar or dissimilar may be higher than listed in the table, as in all sets only a subset of the document pairs are evaluated.

3.2.2 Modeling and Training

In this thesis, we experiment with and evaluate different kinds of feature extraction methods and machine learning models.

We consider using these approaches:

- Information retrieval based model using n-grams + tf-idf weighting and a hyperparameter search.
- Supervised learning using deep neural networks.

We try these approaches as they seem the most promising approach for our task. Other approaches could be unsupervised learning (which we experimented with) and semi-supervised learning. For the models we use the machine learning tool Keras [20], in combination with the computation graph backend TensorFlow [21] or Theano [22] in combination with scikit-learn [23] for creating models, text processing, numerical computation, model evaluation and hyperparameter optimization.

3.2.3 Evaluation

We can evaluate our model by reporting metrics on a collection of human annotated validation sets, i.e. the SOCO [19] dataset and data from Computer Science courses at Utrecht University.

These metrics and results are compared against those of other source code similarity tools that are both publicly available and support comparing Java source code.

Chapter 4

Infinitemonkey Text Retrieval Baseline

In this chapter we show the methods we used to build a baseline for Infinitemonkey. We show how we compute a representation of documents using n-grams, tf-idf weighting, and cosine-similarity. Further we show how we find a reasonable setting for our hyperparameters using grid search.

4.1 Text Representations

To build a model from unstructured text with variable size, the raw text is often first converted to a numeric representation (feature vector). This vector contains for example the number of occurrences of tokens in a document or other features.

4.1.1 N-grams

A n-gram is a contiguous sequence with length n in sequential data, often in text. For example, the Haskell program `main = putStrLn "hello word" >> putStrLn "exit"` split on and including non alphanumeric characters except spaces contains the 1-grams visible in Table 4.1. 1-grams are also called unigrams or "bag-of-words". The downside of the bag-of-words model is that it throws away all information about word order: all possible permutations of a sequence result in the same vector. To overcome the issue of local ordering, n-grams split the text into all sequences with length n: bigrams ($n = 2$), trigrams ($n = 3$), etc. For an example of bigrams ($n = 2$) see Table 4.2. All possible tokens not found in this document, but that occur in other documents have a value of 0.

Instead of splitting the text into words, n-grams can also be computed at the character level. For example, the character level 3-grams in the text `range(2)`

	Count
`main`	1
`=`	1
`putStrLn`	2
`"`	4
`hello`	1
`world`	1
`>>`	1
`exit`	1

Table 4.1: 1-gram of Haskell program

	Count
`main =`	1
`= putStrLn`	1
`putStrLn "`	2
`" hello`	1
`hello world`	1
`world "`	1
`" >>`	1
`>> putStrLn`	1
`" exit`	1
`exit "`	1

Table 4.2: 2-gram of Haskell program

is represented by the set {`ran`, `ang`, `nge`, `ge(`, `e(2`, `(2)`}. The representation of this can be seen in Table 4.3.

To determine the full mapping from n-gram to vector index, a dictionary of all the n-grams in the text must be computed. A common way to avoid the need of building a dictionary of the mapping from n-gram is applying the so-called "Hashing Trick": instead of building up a dictionary, a hash function is used to convert n-grams to an index of the feature vector. The size of the vector can be tuned to keep the number of expected collisions low, while also not being unnecessarily big. The values for n and which choice or combination of features representations to use can be optimized using hyperparameter optimization.

4.1.2 Tf-idf Weighting

For some applications or models it is important that the values in the feature vector are scaled based on the relative importance of each feature. In the case of n-grams, some sequences occur more often in a domain than other sequences, and are often less relevant than sequences that occur less frequently in a dataset. For example: in programs written in the language C, the word `void` probably has a higher frequency than the word `swap`, so when a document contains the word `swap` this is probably more relevant than the word `void`.

	Count
`ran`	1
`ang`	1
`nge`	1
`ge(`	1
`e(2`	1
`(2)`	1

Table 4.3: Character-level 3-gram of `range(2)`

To scale the features, the tf-idf (term frequency-inverse document frequency) weighting scheme is often used. Although variations exist, the main idea is the same: the frequency of the term t in document d is multiplied with the inverse of the term frequency *idf* in the domain D:

$$\text{tf-idf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D) \tag{4.1}$$

The term frequency tf can simply be the raw frequency $f_{t,d}$, but can also be scaled logarithmically, using $\text{tf}(t, d) = 1 + log f_{t,d}$ and binary (1 if the term is present in the document and 0 otherwise).

4.1.3 Cosine-similarity

Cosine similarity is used to compute a scalar value in the interval $[0, 1]$:

$$\text{cosine-similarity}(d_1, d_2) = \frac{d_1 \cdot d_2}{\|d_1\|\|d_2\|} \tag{4.2}$$

Because we normalize the vectors before computing the distance matrix, we only need to calculate the dot product between the two vectors. This operation is very cheap to compute: for example, the pairwise distance matrix of more than 3000 sparse feature vectors (which results in a distance matrix of size 3000×3000) can be computed in about 10 seconds on a modern CPU, depending on the amount of features.

4.2 Grid Search

Grid Search is an optimization technique for finding a global optimimum for a finite set of hyperparameters in a model. It exhaustively searches for the best performing model for each combination of hyperparameters.

For comparing against other approaches, we developed a baseline tool based on information retrieval methods. This baseline consists of character level n-gram features which are weighted using tf-idf on the dataset. The hyperparameters are optimized on the dataset mandelbrot using grid search optimization. The hyperparameter space is defined in Table 4.4. The meaning of the hyperparameters is:

Hyperparameter	values
ngram_range	$\{(3,3),(3,4),(3,5),(3,6),(4,4)...(6,6)\}$
binary	$\{0,1\}$
smooth_idf	$\{0,1\}$
sublinear_tf	$\{0,1\}$
lowercase	$\{0,1\}$
max_df	$\{0.5,0.75,1.0\}$
min_df	$\{1,2,3\}$
analyzer	$\{$"*char*","*char_wb*","*word*"$\}$
min_threshold	$\{400.0,600.0\}$

Table 4.4: Hyperparameter search space

- **ngram_range**: Range of values for n in n-grams. For example, the range $(3,4)$ computes both 3-grams and 4-grams.
- **binary**: Whether to only count first occurrence of term in document
- **smooth_idf**: Whether to add 1 to the document occurences (to prevent zero divisions for some operations)
- **sublinear_tf**: Whether to use raw frequencies or log-scaled frequencies as term frequency
- **lowercase**: Whether to transform sequences to lowercase before counting
- **max_df**: Remove terms that have a *higher* document frequency than max_df
- **min_df**: Remove terms that occur less than min_df times in the dataset. Compared to max_df, this uses the absolute number of occurrences instead of the fraction in which the feature occurs.
- **analyzer**: Whether to split on characters (char), split on word boundaries but use character level n-grams, or use word level n-grams
- **min_threshold**: Minimum sum of weighted token frequencies. Documents that are below this threshold are removed from comparison.

The best performing hyperparameters on the mandelbrot set are visible in Table 4.5. We use these hyperparameters for evaluation of infinite-monkey on other datasets. To confirm the robustness of these parameters found on the mandelbrot dataset, we also performed a grid search with the same search space on the b2 subset of SOCO. This leads to a slightly different set of optimal parameters, which can be seen in Table 4.6. The values for the optimal parameters *ngram_range*, *smooth_idf*, *lowercase*, *min_df* and *analyzer* are different. It has to be noted however that the differences of the final score for most of these parameter settings may be very small. For example, the highest score for a hyperparameter setting with analyzer="char" had an average precision of 0.9348

Hyperparameter	Value
ngram_range	$(5, 5)$
binary	0
smooth_idf	0
sublinear_tf	1
lowercase	0
max_df	1.0
min_df	2
analyzer	"*char*"
min_threshold	400.0

Table 4.5: Best performing hyperparameters found using grid search on the mandelbrot dataset

Hyperparameter	Value
ngram_range	$(6, 6)$
binary	0
smooth_idf	1
sublinear_tf	1
lowercase	1
max_df	1.0
min_df	1
analyzer	"*char_wb*"
min_threshold	400.0

Table 4.6: Best performing hyperparameters found using Grid Search on the b2 dataset

on this task, while the highest with the "char_wb" optimizer has 0.9353 as its average precision.

Using our hyperparameter set the grid search is performed on a total of 8064 combinations. Performing a grid search on a single dataset consumes more than 24 hours for both the mandelbrot and b2 dataset.

Chapter 5

A Source Code Similarity Model

In this chapter we show how we learn a similarity model using a neural network model and a dataset retrieved from open source code retrieved from Github.

5.1 Machine Learning Techniques

Machine Learning lets a computer program learn a task from data. In this section we will talk about the techniques we will need for learning the source code plagiarism task.

For a more complete overview of Machine Learning and Deep Learning, please refer to the book Deep Learning [24] by Goodfellow and others.

5.1.1 Word and Character Embeddings

Instead of converting words to some vector representing the occurrence of tokens or token combinations, the mapping from words to word vectors or character to character embeddings can be learned as well. This can be learned jointly with the main task using a lookup table where each item corresponds to a word or character, or can be initialized using existing word vectors learned using unsupervised learning [25].

In our model we use character embeddings. The source code document is converted to a sequence of integers, which represent the set of characters in our document. The integers are used as index into the lookup table, which contains for each index a vector with a 32-dimensional vector. This table is initialized with uniform distributed random numbers and optimized during training of the network. This can be seen as a more efficient way of encoding text than one hot encoding, where each character or word is represented by a vector with 1 at the integer index and the rest zeros and is multiplied by a linear layer. This is

however very inefficient compared to using a lookup table, as the vector grows with the

5.1.2 Gradient Descent

A popular and often used algorithm to optimize models is gradient descent. Gradient descent finds a (local) minimum of a function by taking small steps towards it using the negative of the *gradient* given the loss function and values of all model parameters. The gradient is denoted by $\nabla f(x)$. The model parameters are iteration computed by the formula $x_{t+1} = x_t - \gamma \nabla f(x_t)$ where γ is the learning rate, a positive scalar value. In most training settings and in this work, stochastic gradient descent (SGD) is used to decrease the time to train models using limited memory. Instead of using the entire dataset at a time, the gradient is computed using a small subset (mini-batch) of the data. Commonly used mini-batch sizes are 32, 64 or 128 data samples. There are alternatives to plain SGD, that require less hyperparameter optimization and (often) achieve faster convergence such as Adam [26].

5.1.3 Neural Network Layers

Neural networks consists of one or more hidden layers which contain both the computations and the parameters that are to be minimized given a loss function.

A densely connected layer we use in our network is for example denoted like this:

$$y = b + W^\intercal x \tag{5.1}$$

where_ is a bias vector with size n, W a matrix of weights with dimension $m \times n$ that are multiplied with the input vector x with dimension n. This results in a output vector m-dimensional output vector y.

This can be used for example for a logistic regression model by combining it with the logistic sigmoid function σ:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5.2}$$

When the output of the densely connected layer is a scalar value, it can be combined with the sigmoid function to build a logistic regression model:

$$p(y = 1|x) = \sigma(b + W^\intercal x) \tag{5.3}$$

The output of this function is in $(0, 1)$, a prediction of the probability of the sample x belonging to class $y = 1$ or to $y = 0$ with probability $1 - p(y = 1|x)$.

To model non-linear relationships inside a model, the hidden layers within a neural network are often combined with activation functions, such as the sigmoid. Other commonly used activation functions are for example the ReLU (rectified linear unit or rectifier) function: $ReLU(x) = max(0, x)$ and $tanh$ in recurrent neural network models.

5.1.4 Early Stopping

Overfitting is a problem in learning models where there is a (big) difference in error on a training set and the error on a validation set. This can be avoided using a bigger training set and regularization methods.

Early stopping is a simple technique for preventing overfit by stopping the optimization procedure when the validation loss does no longer improve. We use a variant of early stopping that only writes a model to disk when this has a lower error on a hold out set than the model that is already saved.

5.1.5 Loss Function

A loss function is the function we need to minimize. For training our model, we use the binary cross entropy loss, which is defined as follows:

$$L(W) = -\sum_{i=1}^{n} \Big[y_i \log p_i(W_i x_i) + (1 - y_i) \log(1 - p_i(W_i x_i) \Big] \tag{5.4}$$

Where $y_i \in \{0, 1\}$ is the binary class label, in our case similar and non-similar and n is the number of samples in our training set. We optimize the difference between the true labels and the predicted conditional probabilities given by our model $p(y_i = 1|x_i)$. Minimizing this loss means we better capture the underlying probability distribution.

5.1.6 Batch Normalization

Batch normalization [27] is a recently proposed method to speed up model training and also improves generalization. The method normalizes layer inputs for each batch: the items in each batch are normalized to have zero mean and a standard deviation of one. They show it enables higher learning rates, speeding up the training of the model. Because the normalization is calculated per (random) mini-batch, the precise values of each unit given a training sample can shift depending on the mini-batch in which they occur. It has been shown this has a positive effect on the generalization of the model, similar to Dropout [28] which removes entire units in the network during training.

5.2 Github Training Dataset

Learning a complex model with high capacity often needs a big set of training data when the model is trained from scratch. That is why we consider retrieving a big dataset from open source projects on Github, and applying refactorings to this dataset using an obfuscation tool we make for this purpose. We hope this dataset and obfuscations generalize well to our source code similarity task.

We retrieve a large number of source code documents on Github using the Github API. The datasets consists of 300 popular open source projects, which

are tagged as Java by Github. The files ending on the `.java` extension are copied to another directory. In addition to this automatically retrieved source code set, we use source code from the RosettaCode project. The original data is available on the following URL: `https://github.com/acmeism/RosettaCodeData`.

The dataset is split into a set of similar source code document pairs and a set of unique documents. After processing, the dataset contains 58548 file pairs that are considered as similar and 57170 file pairs that are considered as dissimilar.

5.2.1 Random Code Obfuscation

We made a tool that automatically applies refactorings to Java source code in order to hide certain similarities, such as identifiers, global orderings and code removal, while keeping other high level similarities intact.

For example, the following similarity between Listing 5.1 and Listing 5.2 is harder to detect on first sight by applying these refactoring. The order of member variables is changed and identifiers are replaced with new ones. Also, the order of a commutative operation is swapped: `top + 1` is changed in `1 + Br7Bo8c5`. However, on a high level, these programs are very similar.

Listing 5.1: Original Java Code

```
public class SumStack {
   private int[] items;
   private int top = -1;
   private int sum = 0;
   public SumStack (int size) {
       items = new int[size];
   }
   public void push (int d) {
       if (top < items.length) {
           top = top + 1;
           items[top] = d;
           sum = sum + d;
       }
   }
   public int pop () {
       int d = -1;
       if (top >= 0) {
           d = items[top];
           top = top - 1;
           sum = sum - d;
       }
       return d;
   }

   public int sum () {
       return sum;
   }
}
```

Listing 5.2: Obfuscated Java Code (formatted)

```
public class O {
   public void B9a310 (int
        u34n9qK) {
       if (Br7Bo8c5 <
            vM.length) {
           Br7Bo8c5 = 1 +
                Br7Bo8c5;
           vM[Br7Bo8c5] =
                u34n9qK;
           pVn_ = pVn_ + u34n9qK;
       }
   }
   private int pVn_ = 0;
   public int pVn_ () {
       return pVn_;
   }
   public O (int VTX) {
       vM = new int[VTX];
   }
   private int[] vM;
   public int Ga () {
       int CfluOXd3 = -1;
       if (Br7Bo8c5 >= 0) {
           CfluOXd3 =
                vM[Br7Bo8c5];
           Br7Bo8c5 = Br7Bo8c5 -
                1;
           pVn_ = pVn_ -
                CfluOXd3;
       }
       return CfluOXd3;
   }
   private int Br7Bo8c5 = -1;
}
```

The obfuscation tool randomly applies the following modifications to code:

- Reordering and removal of import statements
- Reordering of class members
- Removal of class methods
- Renaming of identifiers (i.e. class names, member names, local variables, function parameters)
- Swapping of commutative expressions: addition, multiplication, logical or (`||`) and logical and (`&&`).

The Java parser we use removes both comments and formatting. Because of this, after printing (and formatting) of the code we remove all spacing and newline characters as they no longer give any additional information. Furthermore, the implementation of the tool does not support the entire Java-language, some parts in the code may be unchanged after the obfuscation.

All code samples that are larger than 256 characters and smaller than 3000 characters after removing whitespace are converted to similar pairs and "unique" pairs, 114338 Java files in total.

The obfuscation tool makes sure that there is at least some similarity by keeping at least one class method.

5.3 Neural Network for Source Code Similarity

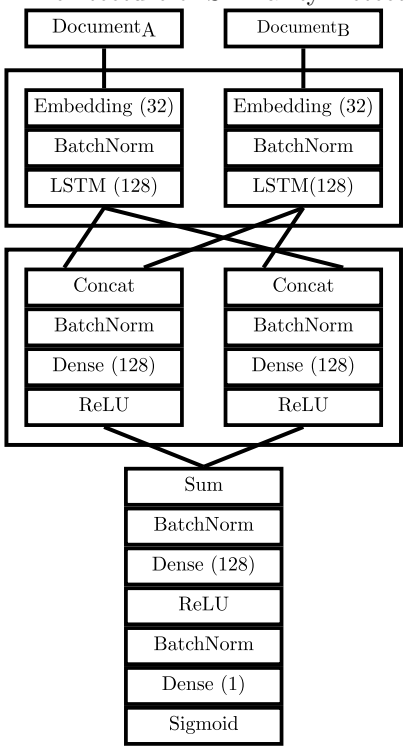
We learn a Neural Network for recognizing similarities in code using the collected dataset that is refactored using code obfuscation.

We learn the network to classify whether two documents are similar or not. In Figure 5.1 we give a schematic overview of the neural network architecture. The network consists of the following parts:

- **Embedding**: A lookup table of character indices to a numeric representation of those characters. These representations are learned while optimizing the neural network.
- **LSTM**: Recurrent Neural Network for conversion of the document to a representation of fixed length. The RNN uses the LSTM (Long short-term memory) [29] architecture.
- **Dense**: Fully connected hidden layer.
- **ReLU**: ReLU activation function: $ReLU(x) = max\{0, x\}$.
- **Sigmoid**: Standard logistic activation function: $sigmoid(t) = \frac{1}{1+e^{-t}}$
- **Sum**: element-wise sum of inputs.
- **BatchNorm**: Batch Normalization: Normalizes the previous layer given the statistics (mean and variance) of the current batch.

The network consists of three main parts: an encoder, which transforms the character embeddings into a representation of fixed size using an LSTM-layer. The documents are then concatenated in two ways and processed by another "comparison" module. The left and right parts of the network share the same parameters. By concatenating the output of the encoder in both ways, sharing parameters and summing the outputs of the "comparison" modules, the learned similarity output of this network will be symmetric: $sim(Document_a, Document_b) = sim(Document_b, Document_a)$. The model constains a total of 136,929 parameters.

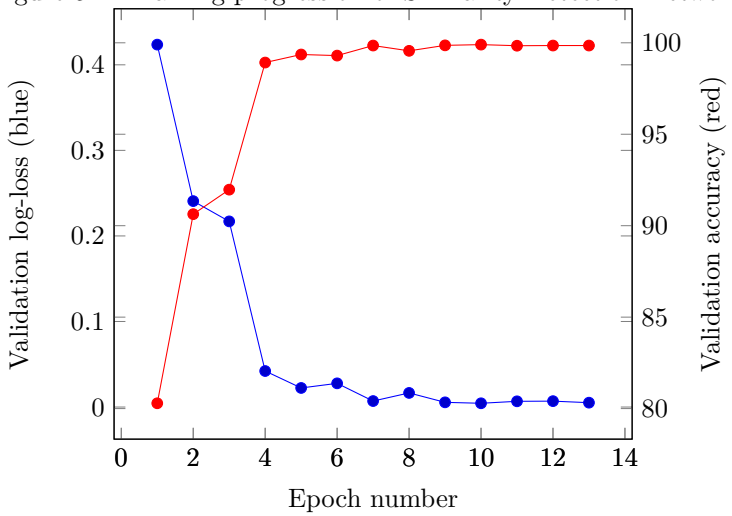
Figure 5.1: Architecture of Similarity Detection Network



For training the network we randomly split the data in a training and validation set using respectively 80% (91470 samples) and 20% (22868 samples) of the data. The model is trained using the SGD method Adam with default learning rates. The model is saved automatically at each iteration when it improves the accuracy on the validation set. The progression of the training of the neural network can be seen in Figure 5.2. The blue line shows the cross-entropy loss of the network on the validation set (lower is better) after each full iteration on the training data, the red line shows the accuracy on the validation set. The high performance of the model on the validation set shows that the model separates the similar and dissimilar classes well, on this synthetic dataset.

For using this large model in practice for plagiarism tasks we reorder the highest similar results of the cosine-similarity model using the predictions of the neural network model. The output of both the first model are combined by taking the average of the two.

Figure 5.2: Training progression of Similarity Detection Network



5.3.1 Other Experiments

Supervised Model using textual features

The first experiment we did for learning a model for source code plagiarism was by training a model on annotated data on the mandelbrot dataset and validating the model on the reversi dataset. Two models we tested on this dataset were logistic regression and a small feed forward neural network. However, the amount of features with textual features is high while the number of annotated samples is very low. This quickly leads to overfitting and little generalization. During these experiments we also tried using L_1 and L_2 regularization and Dropout. In this case, it did not really have a measurable effect on model performance. Another problem of this approach may be that the feature distribution can be different for each task, e.g. terms occurring in one dataset may not even be present in the other dataset and vice versa.

Unsupervised LSTM models

During this thesis we also experimented with two different methods for unsupervised representation learning. Two different models we used for unsupervised representation learning are an LSTM next character prediction model and an Auto Encoder. The next character based model predicts the probability distribution of the characters given the character sequence before it. As part of learning this distribution, it also learns a feature representation of the document in the hidden layers of the model. A second model we experimented with was a LSTM Auto encoder. This model learns to reconstruct the sample X given X, i.e. learn the identity function. For both models the latest layer before the

final layer is used as vector representing each document. All the vectors are compared using cosine similarity, under the assumption that similar documents have vectors in a similar direction. Both models did not perform very well on the mandelbrot set. They seem to predict high similarity values when documents are almost the same, but fail on samples that are harder to predict.

Chapter 6

Visualization of Source Code Similarity

Our tool InfiniteMonkey includes a web-based graphical interface where a user can check programs for plagiarism. It can visualize the similarities between two documents by showing the highest similar fragments between two documents. The application starts the http service using the command `python serve.py;`, this will start a simple web application. After uploading a set of documents (see figure 6.1), the tool will perform the comparison and show the most similar results.

6.1 Visualization

We use the following method to visualize the similarities between two source code files:

- Calculate the relative importance of all the features for each pair
- Locate these features in the document pairs
- Display the fragments with the highest cosine similarity value.

An example of a result in InfiniteMonkey can be seen in figure 6.2. The darker the color, the more that sequence contributes to the similarity score. In this example it can be seen that the sequence: `newText` is much more important than the sequence `else`, as the first sequence occurs less frequently in the dataset than the second. The color of a feature is computed by comparing each n-gram against the maximum value present in the fragments. For each character, the maximum possible color value is used.

The number next to the result is calculated using the similarity score of our model multiplied by 100. The results are ordered by this value from high to low. In figure 6.3 the top 5 of the results are shown.

Figure 6.1: InfiniteMonkey upload screen

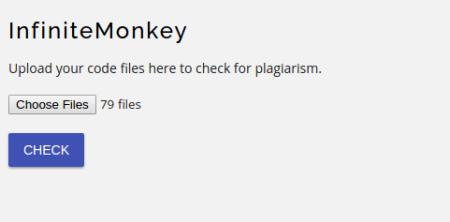
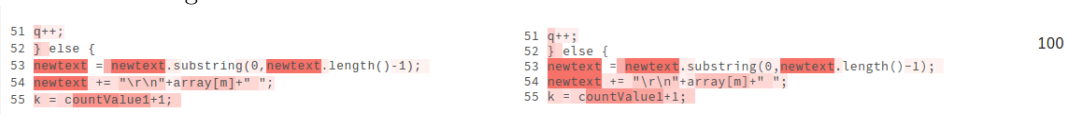


Figure 6.2: Visualization of source code similarities



We can also show the top 5 results within two file pairs, which can be seen in figure 6.4. In our detail view, we show the top 5 combinations between two file pairs with the highest cosine similarity.

Compared to the line based tool `diff` and plagiarism tool Moss, we show the importance of individual features rather than showing respectively identical lines or similar text blocks.

Figure 6.3: Top 5 results on the b2 dataset

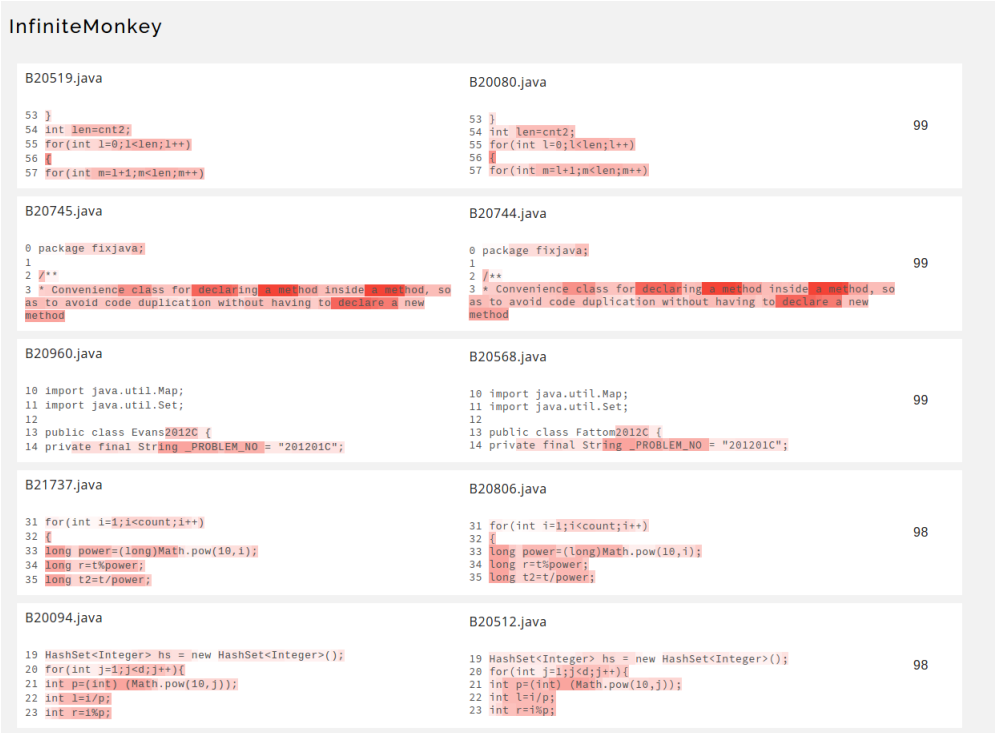


Figure 6.4: Detail view: top 5 similarities for a single file pair

Chapter 7

Evaluation of Results

We evaluate the tools (cpd, InfiniteMonkey, `diff`, `difflib`, Jplag, Marble, Moss, Plaggie, Sherlock and Sim) by running them on different datasets and comparing the performance of the top results using the average precision metric. For the UU datasets we only evaluate the top 50 results for the reversi, mandelbrot and quicksort datasets, and only the top 10 for prettyprint. The reason for only including the top 50 and top 10 of the datasets, is that we annotated only the top 50 and top 10 of the results for each tool, as annotating them consumes a lot of time. Prettyprint is a much smaller dataset, most of the cases must end up in the top of this list. The area under the precision recall curve metric is used to compare the tools on the datasets. The curve is defined as the average of the fraction of correct predictions (precision) at each positively predicted item in the list when the results are sorted by descending similarity. This means that the number of correctly predicted items and the ordering of these predictions are both important for this metric. For the SOCO datasets the score is calculated using the number of pairs evaluated as similar in this dataset. Because the mandelbrot dataset is used for optimizing the hyperparameters of InfiniteMonkey, it is shown in italic.

We excluded all files with less than 512 characters in this analysis for all the tools. We do this to make sure that all small files are almost always not considered as similar, even though they are textually relatively similar. Some tools do remove small files and some tools don't, so this preprocessing step makes them more equal and the comparison is more fair to the methods not doing this preprocessing step.

The scores of the tools are listed in Table 7.1. The first thing that is clear is that the tool `difflib` (a Python diffing library) has the highest scores on all SOCO datasets, and also quite high scores for other sets. Difflib also has the highest mean average precision of all tools. Simple diffing tools like `diff` and especially `difflib` work well on the datasets. The reason for this could be that most of the cases of plagiarism in our datasets are quite easy to detect, i.e. they contain at least some identical parts. The tool `diff` can be sometimes fooled by identical lines with only formatting characters, like newlines, spaces and curly

Dataset	cpd	infinitemonkey	diff	difflib	jplag	marble	moss	plaggie	sherlock	sim	infinitemonkey-rerank
a1	0.611	0.928	0.929	**0.94**	0.842	0.766	0.775	0.857	0.877	0.778	0.754
a2	0.574	0.873	0.919	**0.939**	0.829	0.727	0.83	0.877	0.855	0.755	0.683
b1	0.676	0.965	0.979	**0.981**	0.872	0.71	0.829	0.864	0.77	0.783	0.819
b2	0.522	0.92	0.912	**0.951**	0.83	0.624	0.888	0.911	0.553	0.72	0.903
c2	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
prettyprint	0.0	0.767	0.1	**1.0**	0.764	0.604	0.579	0.633	0.0	**1.0**	0.071
reversi	**1.0**	**1.0**	0.99	0.945	0.964	0.962	**1.0**	0.962	0.744	0.71	**1.0**
quicksort	0.998	**1.0**	0.966	0.965	0.92	0.968	0.995	0.93	0.849	0.833	0.976
mandelbrot	0.975	**1.0**	0.995	0.998	0.996	**1.0**	**1.0**	0.963	0.978	0.59	**1.0**
nr. best	2	3	1	6	1	1	2	1	1	1	3
mean	0.706	0.939	0.866	0.969	0.898	0.82	0.872	0.89	0.792	0.792	0.763

Table 7.1: Average precision scores on source code plagiarism datasets

Dataset	cpd	infinitemonkey	diff	difflib	jplag	marble	moss	plaggie	sherlock	sim
synthetic	0.919	0.983	0.921	0.815	0.985	**0.987**	0.91	0.983	0.806	0.96

Table 7.2: Average precision scores on synthetic dataset

braces, which is the case in the quicksort dataset.

Our baseline version of InfiniteMonkey does achieve good performance on most datasets. It achieves better average precision scores compared to other popular plagiarism tools like JPlag and Moss.

The reranking performed by our learned similarity model did not improve but degrade the performance on most datasets. From this we can conclude that the model did not generalize well to these datasets.

The tools CPD, SIM and Sherlock seem to be relatively weak tools in this comparison, while the tools JPlag, Marble, Moss and Plaggie perform quite well.

We performed also a comparison on a synthetic dataset generated by the random code obfuscation tool listed in Chapter 5. It consists of 200 files with 50 file pairs annotated as similar. For this test we don't list inifinitemonkey-rerank, as it uses the same tool for creating training and test data. In this test, the tools marble, Plaggie, JPlag and InfiniteMonkey have an average precision above 0.98, while the other tool perform worse. Remarkably, difflib performs much worse in this test than on the previous tasks, which suggests that it is sensitive to the refactorings performed by our tool.

Chapter 8

Discussion

We tried two main different approaches towards source code plagiarism detection using machine learning. The first approach is an unsupervised approach using cosine similarity, n-gram features and tf-idf ranking. The main contribution is that the baseline tf-idf approach does well on a large set of plagiarism tasks, and achieves better scores than other popular tools. Systems that also use textual features like Moss, can likely add tf-idf scaling of features to improve the ranking quality.

Our second contribution is a model based on a deep recurrent neural network. This model shows the possibility to learn similarities in source code using our model and a large synthetic dataset. However, this model does not generalize well to our evaluation data.

Finally we compare all the tools on a number of different datasets using the average precision metric and evaluate the results.

8.1 Limitations and Future Research

Our approaches have several limitations. The first approach using character n-gram features and cosine similarity does not need training on a dataset before using it. That does mean that that approach can only know the importance of certain similarities using the frequency of the features within the dataset at run time. That can mean that when the dataset is small, the distribution of certain features may not be representative for the ”true” distribution for the task. This does not seem to be very problematic, as can be seen from the scores. The character level + tf-idf approach seems to lead to fairly good results.

The main approach using machine learning did not result in better scores in the evaluation. The most likely reason of this is that the synthesized dataset using data from open source repositories and automatic refactorings could be too different from the evaluation data. The approach did generalize well to the hold out set in the synthetic data, but failed at the tasks in the evaluation.

For future research directions one obvious approach would be to build a

simpler supervised model using multiple similarity metrics or outputs from different tools. This model could learn the importance of each similarity metric. Another interesting direction would be applying semi-supervised learning: first learn a good representation of source code on a large dataset using an unsupervised learning algorithm and then using these representations as features for a supervised model.

Chapter 9

Conclusion

In this thesis we compared a number of tools and developed our own tool, InfiniteMonkey. We experimented with a more traditional information retrieval approach and machine learning models. We also performed a comparison based on datasets based on two different sources: 9 in total. The n-gram model with tf-idf weighting and cosine similarity works well for this problem and scores well across different datasets. The tf-idf weighted features can also be used as part of a visualization method, as more unique parts in the word pairs will show up as more important than other similarities. We developed a web based GUI based on this simple method. We also tried a deep neural network model on synthetic data generated from open source repositories. This model trained on the synthetic open source dataset did not generalize well to the source code plagiarism tasks we evaluated. This approach needs more work to overcome this problem.

Bibliography

[1] J. Hage, P. Rademaker, and N. van Vugt. Plagiarism detection for Java: a tool comparison. In G. C. van der Veer, P. B. Sloep, and M. C. J. D. van Eekelen, editors, *Computer Science Education Research Conference, CSERC 2011, Heerlen, The Netherlands, April 7-8, 2011*, pages 33–46. ACM, 2011.

[2] Enrique Flores, Paolo Rosso, Lidia Moreno, and Esaú Villatoro-Tello. On the detection of source code re-use. In Prasenjit Majumder, Mandar Mitra, Madhulika Agrawal, and Parth Mehta, editors, *Proceedings of the Forum for Information Retrieval Evaluation, FIRE 2014, Bangalore, India, December 5-7, 2014*, pages 21–30. ACM, 2014.

[3] Enrique Flores, Paolo Rosso, Esaú Villatoro-Tello, Lidia Moreno, Rosa Alcover, and Vicente Chirivella. Pan@fire: Overview of CL-SOCO track on the detection of cross-language source code re-use. In Prasenjit Majumder, Mandar Mitra, Madhulika Agrawal, and Parth Mehta, editors, *Post Proceedings of the Workshops at the 7th Forum for Information Retrieval Evaluation, Gandhinagar, India, December 4-6, 2015.*, volume 1587 of *CEUR Workshop Proceedings*, pages 1–5. CEUR-WS.org, 2015.

[4] Phatludi Modiba, Vreda Pieterse, and Bertram Haskins. Evaluating plagiarism detection software for introductory programming assignments. In *Proceedings of the Computer Science Education Research Conference 2016*, CSERC '16, pages 37–46, New York, NY, USA, 2016. ACM.

[5] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 76–85. ACM, 2003.

[6] L. Prechelt, G. Malpohl, and M. Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. *J. UCS*, 8(11):1016, 2002.

[7] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[8] R. Pike. Sherlock. `http://www.cs.usyd.edu.au/~scilect/sherlock/`.

[9] S. Surakka A. Ahtiainen and M. Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In A. Berglund and M. Wiggberg, editors, *6th Baltic Sea Conference on Computing Education Research, Koli Calling, Baltic Sea '06, Koli, Joensuu, Finland, November 9-12, 2006*, pages 141–142. ACM, 2006.

[10] D. Grune and M. Huntjens. Het detecteren van kopieën bij informatica-practica. *Informatie*, 31(11):864–867, 1989.

[11] D. Grune and M. Huntjens. Sim. `http://dickgrune.com/Programs/similarity_tester/`.

[12] J. Hage. Programmeerplagiaatdetectie met Marble. Technical Report UU-CS-2006-062, Department of Information and Computing Sciences, Utrecht University, 2006.

[13] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In T. E. Rad, L. H. Ungar, M. Craven, and D. Gunopulos, editors, *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 872–881. ACM, 2006.

[14] V. Ciesielski, N. Wu, and Seyed M. M. Tahaghoghi. Evolving similarity functions for code plagiarism detection. In Conor Ryan and Maarten Keijzer, editors, *Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008*, pages 1453–1460. ACM, 2008.

[15] S. Engels, V. Lakshmanan, and M. Craig. Plagiarism detection using feature-based neural networks. *ACM SIGCSE Bulletin*, 39(1):34–38, 2007.

[16] M.L. Kammer. *Plagiarism detection in Haskell programs using call graph matching*. PhD thesis, 2011.

[17] J. Hage, B. Vermeer, and G. Verburg. Research paper: Plagiarism Detection for Haskell with Holmes. In M. C. J. D. van Eekelen, E. Barendsen, P. B. Sloep, and G. C. van der Veer, editors, *Proceedings of the 3rd Computer Science Education Research Conference, CSERC 2013, Arnhem, The Netherlands, April 04 - 05, 2013*, pages 19–30. ACM, 2013.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 96–105. IEEE Computer Society, 2007.

[19] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello. On the detection of source code re-use. In *Proceedings of the Forum for Information Retrieval Evaluation*, pages 21–30. ACM, 2014.

[20] F. Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[22] R. Al-Rfou, H. Alain, A. Almahairi, and et al. Theano: A Python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *CoRR*, abs/1201.0490, 2012.

[24] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, pages 3111–3119, 2013.

[26] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[27] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015.

[28] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[29] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.