BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

SOURCE CODE AUTHORSHIP ATTRIBUTION

SOURCE CODE AUTHORSHIP ATTRIBUTION

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	TOMÁŠ PRUŽINA
SUPERVISOR VEDOUCÍ PRÁCE	Doc. RNDr. PAVEL SMRŽ, Ph.D.

BRNO 2017

Brno University of Technology - Faculty of Information Technology

Department of Computer Graphics and Multimedia Academic year 2016/2017

Bachelor's Thesis Specification

For:	**Pružina Tomáš**
Branch of study:	Information Technology
Title:	**Source Code Authorship Attribution**
Category:	Algorithms and Data Structures

Instructions for project work:

1. Study existing methods of source code authorship attribution.
2. Familiarize yourself with previous work in field of machine learning approaches to source code authorship attribution and feature learning techniques that can be used for source code authorship attribution.
3. Design and implement a system, which, given the source code, will attribute it to its original author with a high probability based on prior knowledge of author's source codes.
4. Evaluate performance of the system on a representative dataset.

Basic references:

- According to supervisor's recommendations.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Smrž Pavel, doc. RNDr., Ph.D.**, DCGM FIT BUT
Beginning of work:	November 1, 2016
Date of delivery:	May 17, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

[illegible]

Jan Černocký
Associate Professor and Head of Department

Abstract

With the rise of the Internet, the trend of source code plagiarism and copyright infringement has been increasing and it has become a problem in both academic and corporate environments. With that in mind, it becomes important to be able to identify author of a source code based on stylomentry techniques to prevent unethical violations of the intelectual property of the author. This thesis summarizes modern approaches to source code authorship attribution and presents a general purpose authorship attribution tool that attempts to detect author of the source code.

Abstrakt

Se vzestupem internetu se trend plagiátorství zdrojového kódu a porušování autorských práv zvyšuje a stal se problémem jak v akademickém, tak v podnikovém prostředí. S ohledem na to je důležité, aby bylo možné identifikovat autora zdrojového kódu založeného na stylomentrických technikách, aby se zabránilo neetickému porušování autorského intelektuálního vlastnictví. Tato práce shrnuje moderní přístupy k přiřazení autorského zdrojového kódu a představuje nástroj pro přiřazení autorů obecného určení, který se pokouší odhalit autora zdrojového kódu.

Keywords

authorship attribution, machine learning, feature detection, source code authorship verification, code stylometry

Klíčová slova

určování autorství, strojové učení, detekce příznaků, ověření autorství zdrojového kódu, stylometrie zdrojových kodů

Reference

PRUŽINA, Tomáš. *Source Code Authorship Attribution.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. RNDr. Pavel Smrž, Ph.D.

Source Code Authorship Attribution

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. Pavel Smržp The supplementary information was provided by Pavel Smrž. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Tomáš Pružina
May 18, 2017

Contents

Chapter 1

Introduction

Source code authorship attribution can be defined as identification of the author of a source code based on stylistic features of the source code.

In its principle this problem is closely linked to the problem of well researched field of text authorship attribution which uses natural language processing (NLP) techniques for a problem of text classification.

To date, vast majority of machine learning techniques of text classification are based on simple statistical model of ordered word combinations which are used to great effect and accuracy.

This thesis presents a classifier capable of determining the most likely author(s) of a source code file trough application of machine learning methods and various stylometry features.

1.1 Motivation

Source code authorship attribution can be used in wide variety of situations to detect the original author of the source code. These use-cases include, but are not limited to copyright investigation, ghostwriting detection, authorship verification, software forensics and programmer deanonymization.

1.1.1 Copyright Investigation

A dispute of authorship between two parties (or a small group of authors) can lead to serious copyright problems and could affect the whole software project. This dispute can be resolved by comparing known samples of code from both (multiple) parties and attributing authorship of disputed parts of code to their authors. It is crucial that this is to be done with high degree of confidence in the result of such code attribution procedure.

1.1.2 Ghostwriting Detection

Ghostwriting is a subtype of plagiarism in which source code is used or published without disclosing involvement of the original author. Unlike traditional plagiarism, the original author of the code is aware of code being used by third party, often at an incentive.

1.1.3 Authorship Verification

Authorship verification is a problem of deciding whether an (assumed) author is the real author of the source code when presented with source code samples allegedly written by the (assumed) author.

1.1.4 Programmer Deanonymization

Programmer deanonymization attempts to reveal the identity of an anonymous author. One notable example of this process is attempt to use machine learning techniques to deanonymize an author going by a virtual identity named **Satoshi** who created original code for reference Bitcoin protocol implementation. While this attempt by the US Federal Bureau of Investigations (FBI) haven't been successful (as far as public knows), this is a serious threat to open-source developers who wish to preserve their code contributions anonymous and their identities hidden.

1.2 History

Historically, authorship attribution methods have been applied to works of art such as paintings and written books.

Stylometry, the study of linguistic style, was used to extract authors features. Up until middle of the 20th century this work was done by hand, by people attempting to recognize stylometric features from the text common to the the author. Principal method of stylometry is called author invariant, which can be described as a property of a text that is invariant of its author. Early applications of author invariant method used features such as average sentence lengths, average word lengths, frequency of usage of common words and vocabulary richness. Soon it became apparent that none of these features were sufficient enough to be truly author-invariant.

First truly successful attempts at using computers for machine learning based authorship attribution came to fruition in 1960's. In 1962 Mosteller and Wallace studied the problem of disputed Federalist papers and came up with a machine learning method based on simple mathematical model of Bayesian inference.

The need for source code authorship attribution arose sometime in 1970s when modern and portable (hardware independent) source codes were becoming more common and source code could be reused across variety of platforms. At this time, there were no legal frameworks that classified source code as an intellectual property or a gave it status of literary work.

With advances in computation power, text classification via modern classifiers such as support vector machines (SVM) and neural networks (NN) became feasible in 1990's.

This trend continued well into the 21th century, with another major step in computing power advancement that came with general purpose programming on graphical processing units of modern computers and application specific accelerating units [5].

Chapter 2

Code stylometry features

A stylometric feature captures distinctive aspects of someone's coding style and its output should remain consistent (for given author) even if the functional purpose of the evaluated code differs.

Most commonly used methods in modern source code authorship attribution tools use representations of code based on word or character level sequences that capture preference of keywords, naming conventions and code formatting features.

2.1 Types of Features

There are 4 basic types of source code stylometry features in contemporary literature, lexical features, layout features, syntactic features and semantic features[3].

2.1.1 Lexical and Layout Features

Lexical features represent authors preference of certain features in the source based on their statistical occurrence. Examples of lexical feature are keyword frequency per line of source code or average number of comments per each line of code. These features can generally be easily computed from source code by using regular expressions and their outputs can be represented as scalar values which can then be fed directly into the classifier.

Layout features represent authors preference for code indentation (white-space characters that serve no other purpose than making code more readable).

The main disadvantage of both lexical and layout features is that their output is often of limited use across multiple programming languages, e.g. Python source code differs from that of C-like programming languages.

2.1.2 Syntactic Features

Semantic features represent source code structure and their extraction from the source code requires use of a parser. Examples of syntactic features are average level of code nesting and branching.

Much like lexical and layout features, syntactic features are largely programming language dependent.

2.1.3 Semantic Features

Semantic features represent code intentions and measure features such as use of recursion, authors tendencies to split code into functions and so on. Semantic features are *technically* language-independent stylometric features, their processing however typically requires an advanced parser which needs to understand the programming language.

Some semantic features cannot be represented as a scalar value(s), and Celiskan-Islam [3] suggests using abstract syntax trees (ASTs) for their representation, but other graph theory based models are also used in practice. Semantic features can be also naturally appearing in complex models of neural networks [7].

2.2 Feature Extraction Methods

Feature extraction is a process of extracting stylometric features from raw input (source code file) into feature or vector of features that can be used as an input for the classifier.

There are dozens of possible features extracting methods proposed in the existing literature, some of which are detailed in following subsections.

2.2.1 Bag of Words

Bag of words (BoW) is simple and yet powerful model used for unsupervised feature detection. The Bag of words model learns a vocabulary of inputs (texts), then models each input by counting the numbers of each dictionary word appearing in the input. BoW algorithm has been used for tasks such as spam filtering [10] and tagging [23].

A variant of Bag of words method called Continous Bag of Words (CBOW) can predict the word given it's context and is used as a building block of a word2vec algorithm described in more detail in section 2.2.3.

2.2.2 N-grams Features

In the context of code stylometry, n-gram features are continuous sequences of N tokens (words, special characters) of a source code. N-gram of size 1 is referred to as a *unigram*, size 2 n-gram is bigram, size 3 n-gram is trigram, and so on.

N-gram based models have been used in the past for purposes of text classification and sentiment learning. One subset of n-gram models known as skip-grams appears to be well suited for text classification task [8].

Skip-grams are n-grams that are formed by allowing adjacent sequences of words to be skipped. 3-skip-n-gram would for example contain n-grams with 3,2,1 adjacent words skipped.

2.2.3 Word Embeddings

Word embeddings are a set of unsupervised language modeling and feature learning techniques in Natural Language Processing (NLP) that map phrases from the vocabulary to vectors of real numbers.

Word embeddings are low-dimensional representation of text sequences. Each row of the resulting matrix corresponds to one token (word or phrase) of the dictionary created from the text. Naive implementation of word embedding algorithm could represent each unique word in the text as a one-hot encoding of a dictionary created from a text.

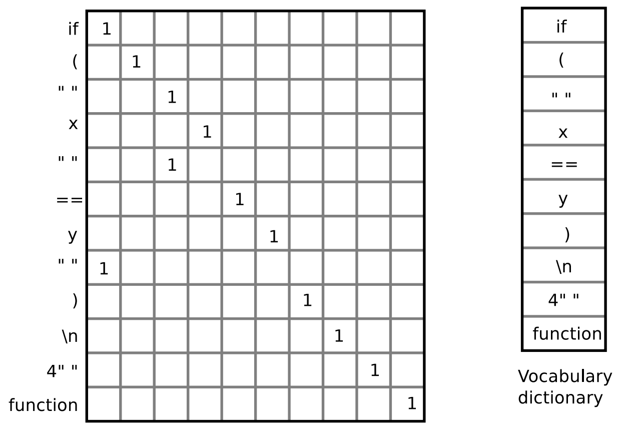

Figure 2.1: Word embedding input vector creation

One-hot Vector

One hot vector embedding is a simple word embedding algorithm based on encoding vector representation of the text by mapping them onto indices of a dictionary built from words of the corpus. This results in a sparse two dimensional matrix of binary values [words, dictionary_words] where 1 represents matched word in the dictionary (at appropriate dictionary index) and 0 a non-matched word. Figure 2.1 explains the idea on how this is done, given artificial source code sequence as follows:

```
if( x == y )
function
```

In this simplified example, vocabulary is being built as the code snippet is being parsed to exemplify the properties of one-hot encoding.

GloVe

GloVe, or *Global Vectors* algorithm is an unsupervised learning algorithm for obtaining vector representations for words [17]. Training is performed on aggregated global word-word co-occurrence statistics from a text corpus.

While originally developed for natural language processing by Computer Science Department of Stanford university, it has been experimentally applied to text classification tasks.

word2vec

Word2vec is an embedding algorithm that has been applied on the problem of text authorship classification (among other problems such as sentiment analysis).

It was originally created in by a team of researchers led by Tomas Mikolov at Google [16]. Word2vec can be described as a shallow two layer neural network model that represent linguistic context of words. Each word is assigned a corresponding vector in feature space. Each word vector is positioned in the vector space in such a way that words with common context are located in close proximity to each other in the space.

Word2vec has been successfully used for text classification tasks such as sentiment analysis and sentence classification [11].

2.2.4 Low Level Data Neural Network Based Feature Detectors

While it is technically possible to represent words and sentences on byte level (ASCII characters), such representation is not particularly suited for machine learning tasks due to its high dimensionality.

In his work, Baisa[2] have researched low level models (morphemes, characters or bytes) and found them competitive with word level approach used by vast majority of modern modeling methods. He introduces low-level models that solve some of the shortcomings of character-level input representation.

Neural networks have been experimentally used as a low level feature extractors that work on character (byte) level for purposes of text classification with some success [24].

2.2.5 Abstract Syntax Trees

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of a source code. ASTs have an advantage over models based on character/word level features in that they are almost entirely programming language independent and can express complex features such as authors tendency to write deeply nested functions or use of a recursion. Another notable advantage of using ASTs as features is that they are more difficult to obfuscate with tools such as indent.

Principal disadvantage of AST based feature detectors is that their parsing is entirely language agnostic and requires a fairly complex fuzzy parser that can handle unseen code constructs or even even syntax errors.

Nevertheless, AST based feature detectors have achieved excellent results [3] in source code authorship attribution tasks.

Chapter 3

Classification

Classification in context of machine learning is the problem of identifying to which category or set of categories a new observation belongs to on basis of a training set of observed data (dataset).

In this process, a set of training data consists of a set of training examples of which each example is a pair of an input object and desired output value of the final model.

In context of source code classification an input object is a source code file (or it's fragment) and desired output value associated with it is an original author of the code.

Process of training the model on labeled data is referred to as *supervised learning*.

Input object is passed to feature detectors which typically produce a N-dimensional feature matrix that is called a feature vector.

Feature vector of each input is then used as an input of a classifier model which outputs classification outcome. Classification outcome is typically either a matched classification class (most likely author) or a vector of all classes, where each value of the vector stands for likelihood of the author being the real author.

3.1 Naive Bayes Classifier

Naive Bayes classifier is a simple, but powerful algorithm based on predictive model of Bayes' theorem.

Naive Bayes classifiers work by correlating the token's input data (typically words) with labeled data and then using Bayes' theorem to calculate a probability that an input sample belongs to classification class.

Naive Bayes classifiers are widely used for text classification [18] and a spam filtering [22]. They have also been used for authorship attribution [4], [13].

3.2 Random Forest

Random forests or *Random decision forests* are an ensemble learning classification method that operate on multitude of decision trees set at training time outputting the classification class that appears most often as a result in set of decision trees [9].

Random forests have already been successfully used for source code authorship classification [14].

3.3 Support Vector Classifier

Support vector machines (SVMs) are commonly used as classifiers (also referred to as Support Vector Classifiers) in field of machine learning. They represent data as a points in N-dimensional feature space. Originally developed in 1960's as a linear classifiers, with application of kernel trick proposed by Aizerman [1] they can be used for classifying non-linear problems. Modification of SVM method that became known as Support Vector Clustering (SVC) allows for unsupervised learning and data-mining by means of clustering the data in feature space.

When compared to Neural Network based classifiers, SVM based classifiers are typically order of magnitude faster and less memory demanding while being able to achieve similar results.

SVCs have been successfully used for text classification and experimentally for authorship analysis of compiled program binaries [19].

3.4 Neural Networks Classifiers

Neural network based classifiers have been used extensively in recent papers on NLP text classification [11], [25], [13]. Neural networks are more closely detailed in following chapters.

Chapter 4

Artificial Neural Network

Artificial neural networks (ANNs) are a machine learning model which is based on a large collection of connected simple units called nodes or neurons, loosely analogous to axons in a biological brain. Neurons are connected to each other by connections that carry an activation signal of varying strength in a network of connected nodes. If the combined input of incoming signals is strong enough, the node becomes activated and the signal travels to other node connected to it.

ANNs can be represented as a directed graph in which nodes represent nodes and directed edges represent connections.

ANNs that resemble directed acyclic graph are commonly referred to as Feedforward Neural Networks, while graphs that contain directed cycles are called Reccurent Neural Networks.

Recurrent neural networks (RNNs) are models that have been naturally applied to NLP problems mainly because RNNs can use their internal memory to process arbitrary sequences of inputs which makes them useful for tasks such as speech recognition or authorship classification.

Convolutional neural network model implemented in this thesis will be briefly explained in following sections as well as its basic building blocks.

Note that full overview is outside the scope of this work.

4.1 Basic Building Blocks of a Neural Network

4.1.1 Node

Node of a neural network, also referred to as a *Perceptron* and *Neuron* is a basic building block of a neural network. It represents a function that takes multiple input parameters (typically outputs of from a previous layer of NN) and outputs a result value.

Input of each node is multiplied by the weight (of each connection), results of this operations are summed and evaluated with activation function (4.1.1). Afterwards bias value of the node is added to the output which then serves as an input for node in next layer.

Figure 4.1 shows example node with 3 inputs along with their associated weights and single output.

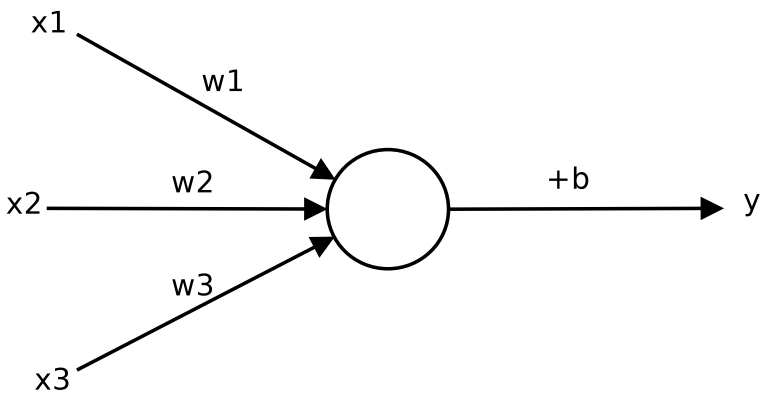

Figure 4.1: Perceptron

Activation Function

Activation function of a node defines the output of that node given an input or set of input data. Two of the most commonly used activation functions that are were experimented with in this thesis and are commonly used with CNNs are ReLU and logistic function (*sigmoid*).

$$output = f(\sum x_i * w_i) + bias \tag{4.1}$$

Logistic Function

Logistic function is a function defined with equation:

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{4.2}$$

ReLU

Rectifier linear unit (*ReLU*) is an activation function defined with equation:

$$f(x) = \max(0, x) \tag{4.3}$$

Upon initialization of the model, weights and biases of nodes are initialized to pseudo-random values. During training weights and bias values are adjusted by running the optimizer based on gradient descend (4.2.3).

4.1.2 Network and evaluation

A typical feed-forward Neural Network used for supervised classification contains several layers of nodes, with one or more input layers (nodes have single input, for example scalar value of a feature detector output), several inter-connected hidden layers and an output layer which maps output value of each classification class (given the inputs in input layer).

Figure 4.2 shows such a network with input vector of 4, two hidden layers of size 5 and output layer with 3 classes.

During an evaluation of this model, the network is presented with inputs x_i which are passed trough the Nodes in input layer, outputs of input layer are passed to nodes of first hidden layer and so forth until outputs reach the output layer.

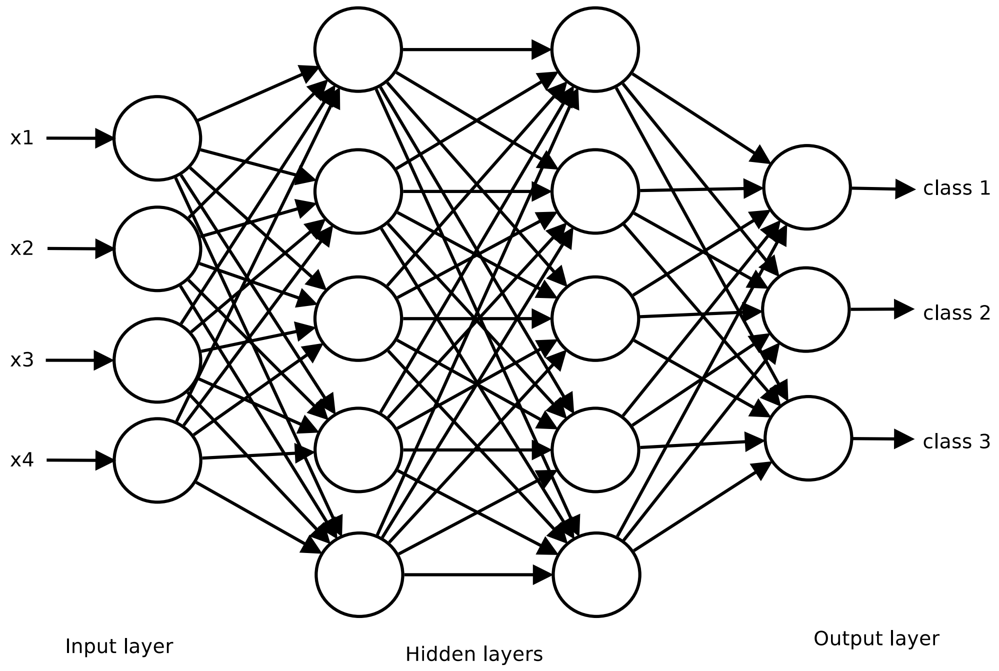

Figure 4.2: Example of feed-forward classification network

On a trained model, input features presented to the model will output a likelihood prediction of each classification class. Subsections that follow will detail how model learns to classify the input data.

4.2 Model learning

Learning is a supervised process that occurs every time a neural network is presented new input during the training phase (learning epoch). Feedforward neural networks often use backpropagation, which is an abbreviation of for *Backward Propagation of Error* [20].

4.2.1 Backpropagation

Backpropagation works by adjusting the model (weights and biases of nodes) based on knowledge of the true classification classes of inputs during the training. Backpropagation works in two phases, forward pass and backward pass that are further detailed in following subsections.

4.2.2 Forward pass

Forward pass evaluates current model on set of input data and outputs predictions for each classification class. On untrained network, these prediction values will have no correlation to the true classes of the input (labels).

Loss function

Loss function, also referred to as *cost function*, is a function that returns a real number representing how well (badly) the neural network performs during training by comparing predicted output values (classification classes) to correct values (labels).

Since output values of model are real numbers while true values (labels) are a *binary truth values* (1 or 0), normalizing function has to be applied onto output layer in order to calculate loss function and error of the model. Most commonly used method for this purpose is softmax.

Softmax function

Softmax function, also also referred to as *normalized exponential function* is a generalization of the logistic function that normalizes K-dimensional vector $\vec{v}$ of arbitrary real values into K-dimensional vector of real values that range [0,1] and add up to 1.

$$softmax(\vec{v}) = \frac{e^{v_j}}{\sum_{d=1}^{K} e^{v_k}} for j = 1, ..., K \tag{4.4}$$

When adjusted for our problem of multi-class neural network it takes shape of:

$$x_i = \frac{e_i^s}{\sum_c^{nclass} e_c^s} \tag{4.5}$$

where x_i is normalized value for each output class.

Figure 4.3 shows how softmax is applied over output layer of the neural network.

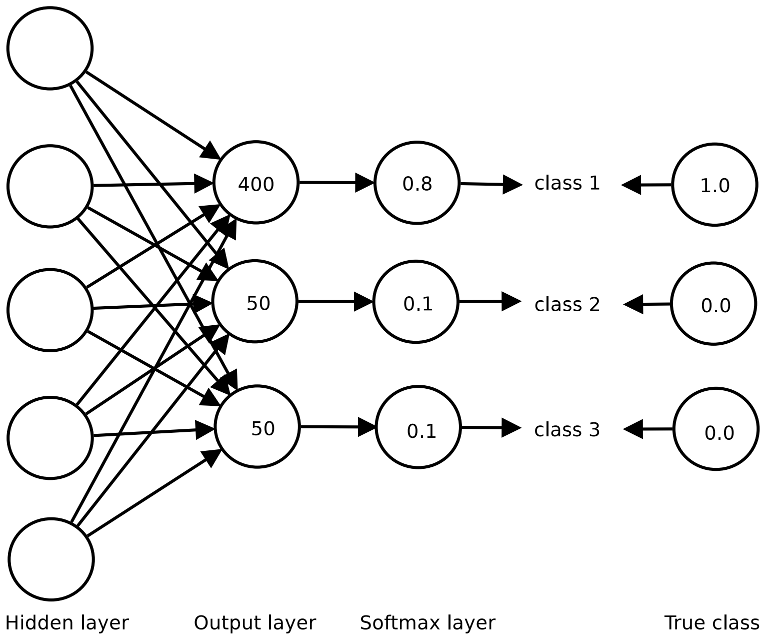

Figure 4.3: Softmax layer applied to output layer of a NN

Cross entropy

Cross entropy cost function used for multi-class neural networks [20] can be defined as follows:

$$E = -\sum_{i}^{nclass} t_i log(x_i) \tag{4.6}$$

This signifies total error of our model and is required for weight adjustment in backward pass of backpropagation during learning phase.

4.2.3 Backward pass

Backward pass, sometimes referred to as *weight update*, recursively (from output layer, trough hidden layers onto input layer) adjusts the weights in the model.

The goal of backward pass is to update each of the weights in the network so that they cause the current output to be closer to the target output. This is done by minimizing the error for each Node of the network. This is done in two steps for each weight:

1 The weights output delta and input activation are multiplied to find the gradient of the weight

2 A fraction of the weight's gradient is subtracted from the weight

The fraction mentioned in step 2 is also known as learning rate of the training phase. This parameter is a value in range [0,1] that suggest how much should weights change during each training iteration. If set to 1, network would always correctly classify the current inputs of the network, but wouldn't generalize onto unseen inputs.

Weight update

Following example shows how weights are updated by computing their gradients on an output node of a network.

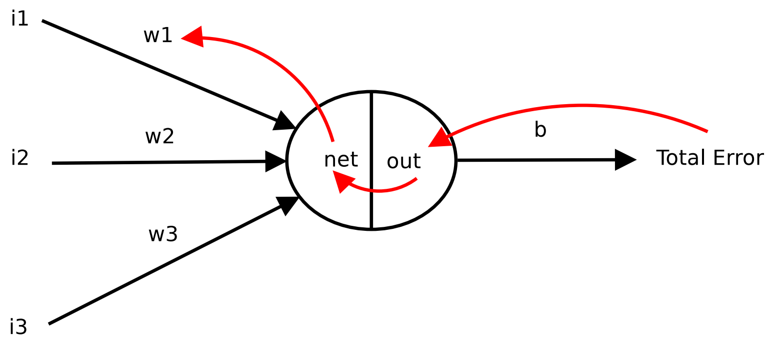

Figure 4.4: Simplified adjustment of weights of a single node

In order to find out how much a change in w_1 will affect total error, we need to find the gradient with respect to w_1 which is defined as:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out} * \frac{\partial out}{\partial net} * \frac{\partial net}{\partial w_1} \tag{4.7}$$

E_{total} stands for summed error of the node (since we already have E_total from forward pass, we don't need to compute it), **out** stands for activation value of the node adjusted by bias (output), **net** is a sum of all inputs (sometimes referred to as net input).

Final formula for updating weights is:

$$w_1update = w_1 - learning\_rate * \frac{\partial E_{total}}{\partial w_1} \tag{4.8}$$

This calculation is done on for each Node of the network in reverse order of that used to evaluate the model during the forward pass.

Gradient descend

Gradient descend is an iterative algorithm which computes the gradient of a function and uses it to update the parameters of the function in order to find a maximum or minimum value of the function. It is used to optimize (minimize) the loss function which parameters are the weights and biases in the network. Gradient descend algorithms belong to one of 3 categories based on the input size that is presented to the model when updating its weights.

1. **Batch gradient descend** use every example in each iteration
2. **Stochastic gradient descend** uses single example in each iteration
3. **Mini-batch gradient descend** uses b examples in each iteration where $b > 1, b < numberofexamples$

This work uses Adam algorithm [12] implemented in TensorFlow library. Stochastic Gradient Descend (SGD) algorithm was also experimented with, but found inferior.

4.2.4 Overfitting

Neural networks are very expressive models that can learn complicated relationships between input and output data. This can lead to sampling noise problem in the model (relationships that exist in training data but not in test data).

To deal with this problem, several techniques have been used.

Dev/Train division of data

One simple technique is the division of the (labeled) training data into separate **dev** and **training** sets, where **training** set is used for training the model and the **dev** set (training phase test set) is used for periodic evaluation of accuracy. When the classification accuracy of the model on the **dev** set starts getting worse, the training is stopped.

For method to work, (multi-class) input data needs to be shuffled in the dataset to provide optimal coverage of each class in both training and dev set.

L1 and L2 regularization

Other commonly used method is L1 and L2 regularization [15] which works by penalizing weight values of NN nodes to the calculation of loss function (4.2.2).

Dropout

The key idea behind dropout [21] is to randomly drop nodes along with their connections from the neural network during training. This is meant to prevent nodes (typically within layer) from co-adapting to each other.

4.3 Convolutional Neural Networks

Convolutional Neural Networks were originally developed for use with image classification in mind and differ from classic architecture in that they use convolution operation on inputs, taking advantage of image pixels binned in a neighborhood correlating to each other.

A typical CNN consist of 3 main types of layers of nodes, Convolutional Layer, Pooling layer and Fully connected layer. In CNN models, the convolutional layer is generally referred to as *feature detector* and fully connected layer is referred to as a classifier.

Main advantage of CNNs over other types of NNs is that they are fast to train and generalize very well on well suited input data.

CNNs utilize have already been applied to NLP problems such as semantic parsing and sentence classification [7]. It has been shown[11] that convolutional neural networks can be used for the task of sentence classification with promising results when used with pre-trained word vectors as inputs.

4.3.1 Convolutional Layer

Convolutional layer consists of set of trainable filters (kernel windows) that are convolved across the input matrix computing the dot product between the elements of filter and input data. This is optionally followed by activation function that adds non-linearity to the model (4.1.1). Afterwards, outputs of the convolutional layer are subsampled by a max-pooling layer (or other subsampling method).

Figure 4.5 shows an example step of convolution process with 3x3 filter sliding over 5x5 input matrix resulting in 3x3 convolved feature matrix.

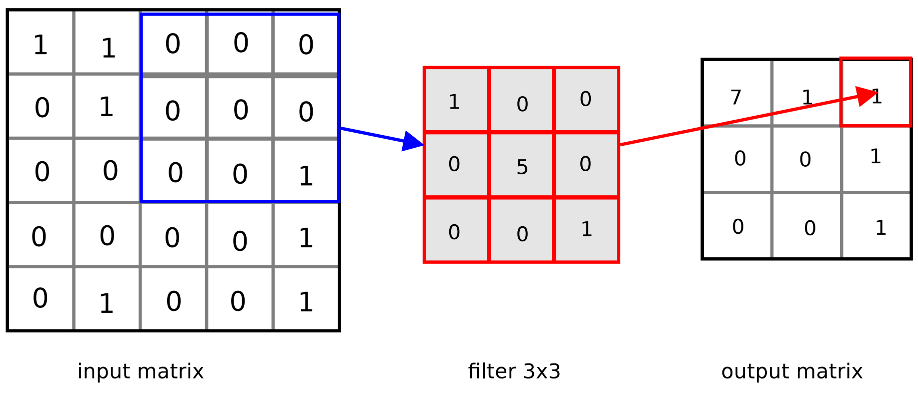

Figure 4.5: Convolution of 5x5 image and 3x3 image filter

4.3.2 Pooling Layer

Pooling layer, sometimes referred to as *subsampling* or *spatial pooling layer*, is a layer of a CNN that reduces the dimensionality of each feature map while retaining most of the information.

Pooling layers define a spatial neighborhood (for example a 3x3 window) and produce a single value. This value is produced in several ways that define the type of the pooling layer, for example max-pooling layer takes the highest value from the neighborhood, sum pooling sums all the values and average pooling averages all elements of a neighborhood. In practice, Max Pooling is most commonly used method.

4.3.3 Fully connected layer

Fully connected (FC) layer is a traditional multi-layer perceptron layer that uses a softmax activation function in the output layer as described in figure 4.2. *Fully connected* means that each node in the previous layer is connected with to the each node in following layer.

Fully connected layers are meant to learn non-linear combinations of features from feature layer.

In CNN models they typically follow convolutional layers with one or more FC layers being stacked together with final layer serving as output layer of the model (with activation function being omitted) and optional dropout layer.

Chapter 5

Dataset

Dataset used for development and training of the model that implements source code authorship classification in this thesis was obtained by parsing public source code repositories on GitHub.

5.1 GitHub

GitHub[1]is a source code hosting platform for source code version control, management and open source collaboration. As of April 2017, GitHub reports „almost 20 million users, 57 million repositories, and 100 million pull requests“[6].

5.2 Dataset gathering

In order to obtain relevant data from GitHub, *PyGitHub* library was used for *scraping* (web data extraction process) the metadata about source code repositories.

PyGitHub3[**?**] is an open source library written in Python programming language[2] that allows access the GitHub API (Application Programming Interface)[3] and its features such as listing existing users, their repositories, source code size and and others.

It was used to list existing GitHub users and their repositories and downloading them if following conditions are met for each repository:

- Size: <10,1000> kB
- Programming language: C, C++
- Number of commits: > 1
- Number of contributors: 1
- License: permissible (MIT, GPL, ...)

These restrictions are meant to sanitize the downloaded dataset as much as possible, although optimal dataset (in which all authors would be the real authors of all source files associated in their respective repositories) couldn't be achieved. Further details are discussed in chapter (7).

[1]`https://github.com`
[2]Python language reference available at `https://python.org`
[3]GitHub API v3 reference available at `https://developer.github.com/v3`

Dataset scraping has proven fairly difficult and slow mostly due to GitHub API limitation of number of requests that one public IP address can process per hour (5000). Since every requests (listing users, listing repositories, getting repository details) counts into this limit and filtering parameters were fairly strict, final dataset gathered over two days of constant parsing is still fairly small.

5.3 Dataset preprocessing

Downloaded repositories with source codes were automatically unpacked and everything except source codes (as identified by file extension) were deleted.

Whole dataset was further partitioned by size, repositories of less than 256 words (151 files, 11% of the dataset) were separated from the main dataset used for training and evaluation of the model during experiments. Similarly, biggest repositories with over 1500 lines of code (30 files, 21% of the dataset) were separated from the main dataset for further experiments.

Standard UNIX tool *iconv*[4] was further used to change input file encoding into ASCII text.

After manual inspection of the dataset, repositories were separated by size into three categories:

- **big-mt-1500l** Repositories bigger than 1500 lines of code (300 files)
- **small-lt-256w** Repositories smaller than 256 words (e.g. ˜less than twice the size of input sequence 6.3.2)
- **data** Everything else (960 files)

For experimental purposes, clones of **data** repositories were created:

- **no-comment** Repository data without C-style comments
- **no-words** Repository with words replaced by a single character **X** (words can defined by regular expression [a-Z][a-Z|0..1]+), e.g. a string that stars with a letter of the alphabet followed by optional alphanumeric letters)

5.4 Data statistics

Following table captures statistics of obtained from raw dataset:

Table 5.1: Dataset statistics

Feature	Value	Note
Number of authors	648	
Number of repositories	1412	
Size	81MB	(data=19MB, big-mt-1500l=71MB, small-lt-256w=640kB)

[4]iconv reference available at http://pubs.opengroup.org/onlinepubs/009695399/functions/iconv.html

Chapter 6

Authorship attribution tool architecture

6.1 Architecture

Figure 6.1 shows high level overview of the reference implementation of the source code authorship tool work. Firstly, the dataset is created from directory of formatted source and stored into the memory. During this step the data is processed as detailed in section 6.3 and vocabulary of the input data is stored as detailed in section 6.3.2.

the model is trained on labeled examples source code in training phase of the classification. After the model is trained, the model can evaluate unknown (unlabeled) examples of source code in prediction phase and it outputs a vector of classification classes (authors from training dataset) with their respective probabilities per each class.

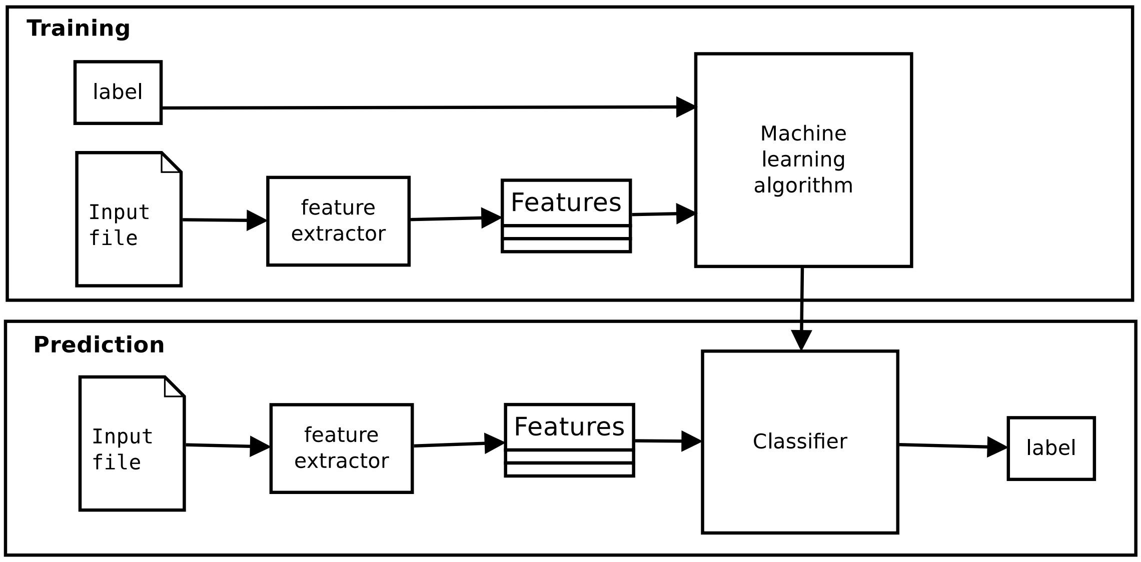

Figure 6.1: High level design of the source code authorship detection tool

6.2 Technologies used

6.2.1 TensorFlow

The neural network software framework library of choice used in this thesis is TensorFlow[1].

TensorFlow is an open source library released in 2015 by Google and is meant to make it easy for developers to design, build and train deep learning models. It originated as in-house library meant for internal Google use.

From high level point of view, TensorFlow is a Python[2] library that computes on arbitrary data by using graph of data flows. Nodes of this graph represent mathematical operations and edges represent the data being communicated from one node to the next. Data itself is represented by multi-dimensional vectors also known as *Tensors*.

For visualization of the model and suite of visualization tools distributed with TensorFlow named TensorBoard[3] was used.

6.3 Data Processing

6.3.1 Input Tokenization

Tokenization converts a raw source file into a single space delimited sequence of strings (row) which has original whitespace/symbolic characters replaced with arbitrary tokens (for example, group of 8 spaces is replaced with token **###8S**.

The idea behind token replacement is logical separation of adjacent words to ease vocabulary dictionary creation in subsection 6.3.2. For example, the C expression **var1==var2** would be an unique single dictionary word. After token replacement, resulting sequence yields 3 tokens **{var1, ###EQUALS###, var2}**.

List of selected replacement token categories is listed in table 6.1.

Table 6.1: List of replacement tokens

Token	Description	Number of tokens
###nS###	Adjacent spaces ###4S###)	8
###TAB###	Tokenized tab	1
###NL###	Newline character	1
###OPERATOR###	various C operators (+=,+,-,&,*,++,–,!,~,{,},[,],(,), ...)	33

6.3.2 Data splitting

As discussed in chapter 4, CNNs expect fixed sized inputs. To achieve this, tokenized data sequence is split into fixed size vector of 128 input tokens. Input sequence is further padded with padding tokens as necessary.

While this is necessary to make word embeddings based feature detector work, it also has an added benefit of dividing each source code into moderately sized sequences of tokens (128 tokens represent approximately 15 lines of code on average).

Each of these tokenized input sequences (further referred to as *chunks*) is classified independently along with lexical and layout feature vector (which is calculated on per-file basis). This is further discussed in following sections 6.4.2 and 6.4.1.

[1] https://www.tensorflow.org/
[2] https://www.python.org
[3] https://www.tensorflow.org/get_started/summaries_and_tensorboard

Vocabulary dictionary is created during training phase from the training set after the input data has been tokenized.

6.4 Feature extraction

Feature extraction process is two-fold, first lexical and layout features are extracted during parsing for each source file. Secondly, features are extracted during via convolutional neural network and vocabulary based word embeddings.

Data flow of feature extraction process is captured by figure 6.2 and further detailed in subsequent sections 6.4.1 and 6.4.2.

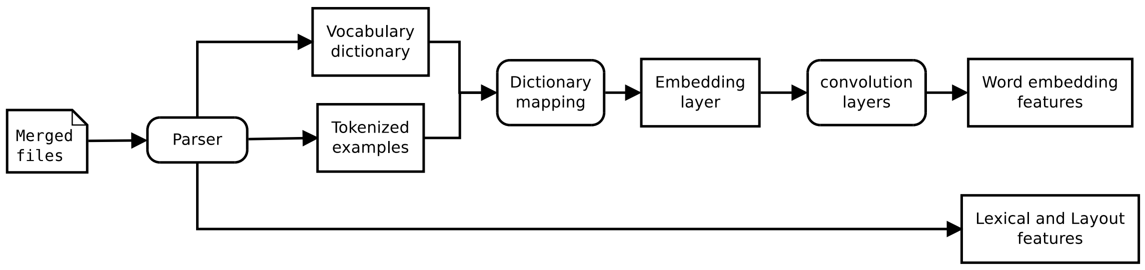

Figure 6.2: Feature extraction data flow

6.4.1 Lexical and layout features

Lexical and layout feature detection is being done on per-file basis during file parsing stage together with tokenization of inputs. Output result of a lexical and layout feature detector is a 1D feature vector of numerical values per each file.

Table 6.2 lists the selected features detected during this process.

Table 6.2: Basic lexical and layout features

Feature	Class	Description	number of features
kw_occurrence_<KW>	lexical	number of keywords (KW) per line of source code	~30
avg_line_len	lexical	average length	1
tabs_per_line	layout	basic source code metric	1
file length	layout	another basic source code metric	1
<N>spaces_per_line	layout	features describing width of indentation (4 consecutive spaces are represented by token 4spaces_per_line)	8
tabs_per_line	layout	average number of tabs per line of source code	1

6.4.2 Word embeddings

Features themselves are extracted by passing embedding layer to the set of convolutional layers in parallel. These convolutional layers extract features vectors by applying kernel function (filter) onto input vector.

Embedding Layer

Embedding layer maps vocabulary word indices into low-dimensional vector representation via vocabulary dictionary created by parser during training of the model.

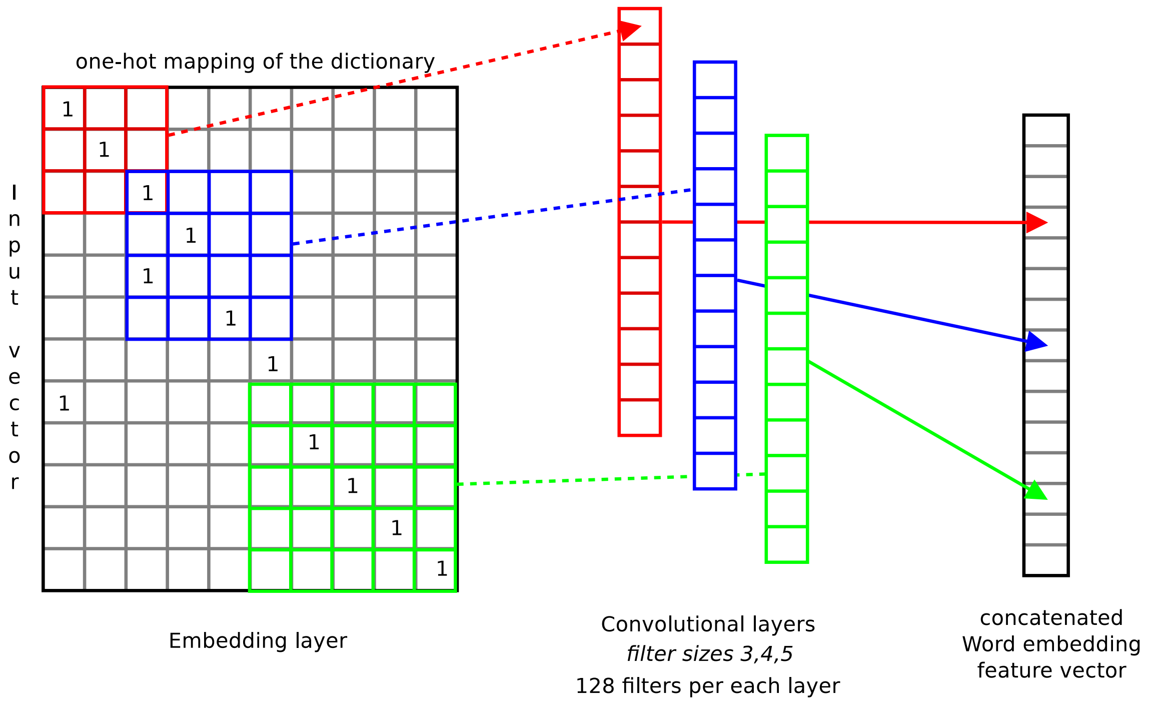

Figure 6.3: Word-embedding feature extraction process

In the actual model input is represented by an input batch comprised of multiple chunks of input data. During training phase, input batch contains randomly shuffled input sequences from the dataset.

Convolutional Layer

Each convolutional layer is represented by simple 1-layer CNN.

As an input it takes expanded 4-dimensional vector [batch_size, chunk_size, embedding_size].

Kernel *window* continually moves (strides) over the input matrix and applies convolution which results in single scalar value. Result of this operation is a matrix of all convolutions.

Afterwards, an ReLU (rectified linear unit) activation function is applied over the results of the convolution. Logistical function (sigmoid) was considered as well, but experimentally ReLU seems to generalize faster.

Since the input vector contains relatively sparse values, convolutional layer is followed by max-pooling, reducing it's dimensionality and allowing for better generalization.

Each of convolutional layers outputs a feature vector which is then concatenated into final convolution layer output vector.

To prevent overfitting of the model, dropout layer(4.2.4) can be added behind the max-pooling layer.

6.5 Classification

Using the output of **features detectors** from **feature extraction phase**, classifier part of the model can be evaluated and model outputs can be generated. Classification output per each class can be evaluated by measuring output signals of the output layer (output vector with size equal to number of classification classes).

Softmax function is further applied to normalize this vector of resulting classes and this operation yields a vector of probabilities per each class.

This vector is presented for each tokenized input sequence (128 tokens by default) and can be either reduced to single file by using the normalization or further used in attempt to detect plagiarism of a fraction of a source file (*stolen function*).

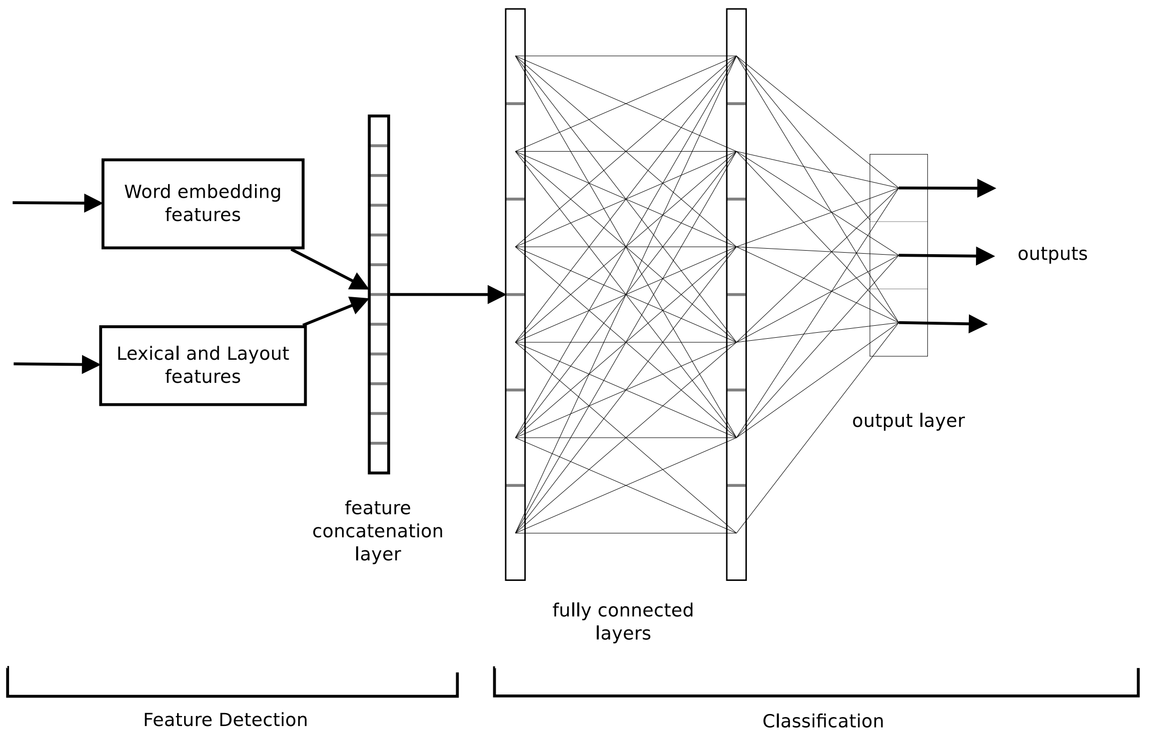

Figure 6.4: Classification data flow

Chapter 7

Model Evaluation

During development and experiments, the *current* model was continually tweaked and adjusted in order to best generalize on a dev dataset. For this purpose, TensorBoard output showing the model weights and biases was continually evaluated along with training and testing accuracy. Model underwent several revisions described in following sections.

7.1 Feature selection

Initially Convolututional layers 6.4.2 were combined with with *lexical and layout feature layer* in the manner depicted in following figure 7.1.

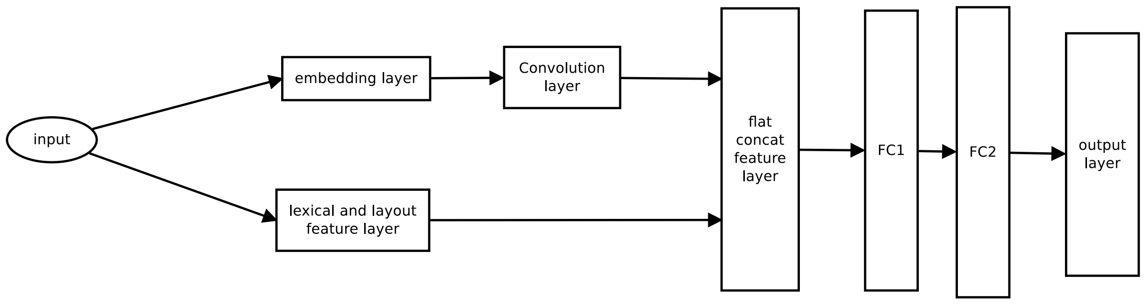

Figure 7.1: Discarded NN architecture model

After initial evaluation on moderately sized dataset, analysis of model weights in the first fully connected layer (*classifier*) has showed that that after training, these features have almost no impact on the model.

It became clear that this model can be simplified into following final model:

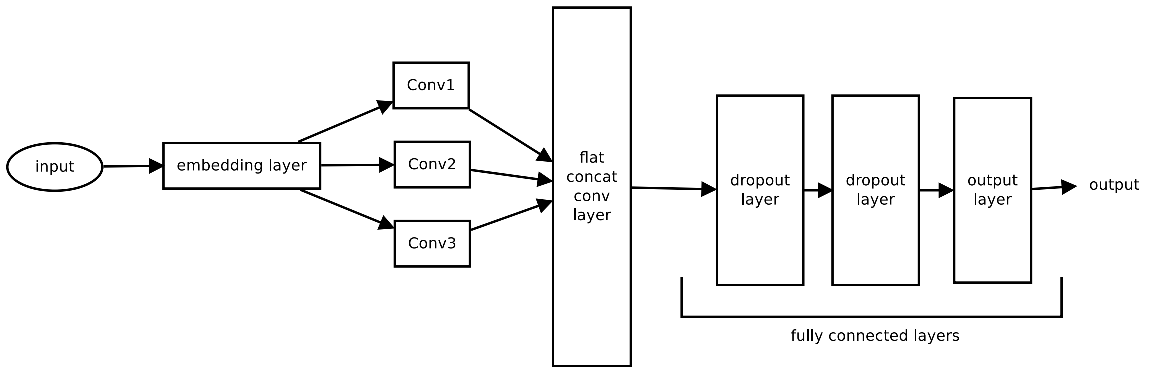

Figure 7.2: final NN architecture model

7.2 Model evaluation

Model can either output the raw output vector per each classification class, a softmax normalized vector per each class or a *hard prediction* of a class with highest output value. Reference implementation of the model output its results as a CSV file[1].

CSV format is defined as <file>,<line>,<class>,[optional vector of softmax values per each class].

Note that the **line** stands for the position in the input source code where the sample chunk used for evaluation was taken from.

Example output of the model is as follows (note: sequence on lines <79,89> was miss-classified as *jhunt* instead of *mattn* GitHub author.

```
../data/data/mattn.c-bucketsort.raw,1,mattn
../data/data/mattn.c-bucketsort.raw,17,mattn
../data/data/mattn.c-bucketsort.raw,32,mattn
../data/data/mattn.c-bucketsort.raw,48,mattn
../data/data/mattn.c-bucketsort.raw,63,mattn
../data/data/mattn.c-bucketsort.raw,79,jhunt
../data/data/mattn.c-bucketsort.raw,89,mattn
```

All experiments and accuracy measurements were done with main „/data" dataset as described in section 5.3 and default model parametersA unless specified otherwise.

7.3 Model accuracy

For measuring the accuracy of the model during development, labeled main dataset was used and split into training (90%) and test (10%) sets.

After each 100 evaluations of the 64 chunk sized minibatches, evaluation on the test set was measured and true values were compared to output class values. Output class is for purpose of this measurement defined as the class with highest output signal from the output layer.

After each test set evaluation on 100th iteration, the model was saved if the accuracy has improved7.4.

[1] http://www.ietf.org/rfc/rfc4180.txt#page-1

It should be noted that whole dataset divided into chunks of 128 tokens with their respective true class values was shuffled and as such some correlation is present between dev and test set (e.g. separate repositories coming from the same author haven't been used).

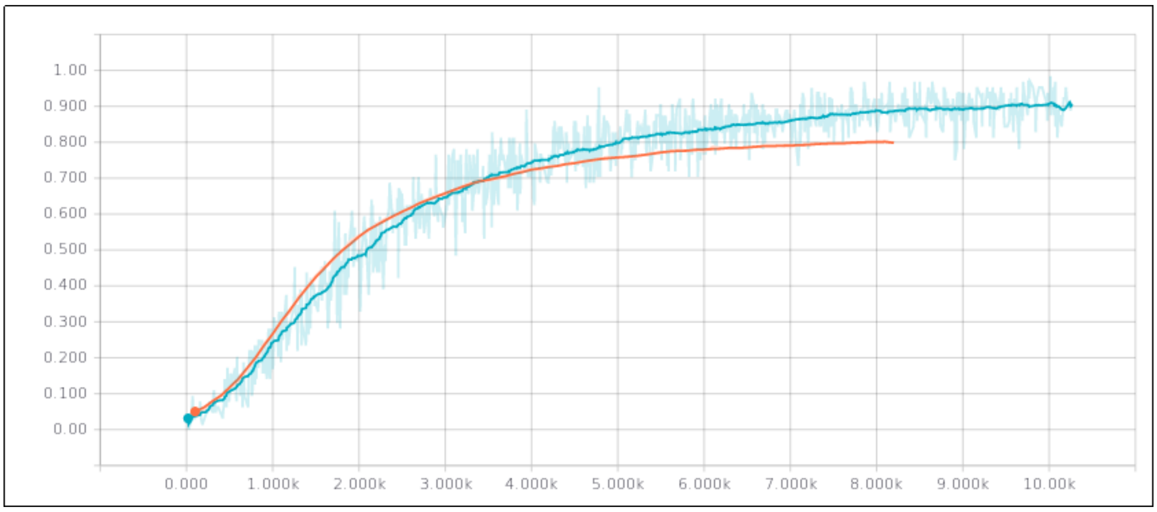

Figure 7.3: Model accuracy as measured on dev/test set of the main dataset, X axis stands for evaluation step, Y axis for accuracy, blue signal is accuracy on the training set, orange signal is accuracy on the test set, note that test accuracy stepped updating when model started overfitting

Reference accuracy of the model on is **83%** on the test set and **91%** on the training set.

If the model was evaluated on previously unseen data, it was forced to make a wrong prediction by choosing the class with highest output signal (argmax[2] operation on output layer).

7.4 Scalability

The model implemented in this thesis scales very well on the testing dataset, but proper cross-validation with method such as k-fold cross-validation[3] haven't been measured. Test accuracy on main dataset stops after only 5 epochs when the model starts overfitting.

[2]https://www.tensorflow.org/api_docs/python/tf/argmax
[3]https://www.cs.cmu.edu/~schneide/tut5/node42.html

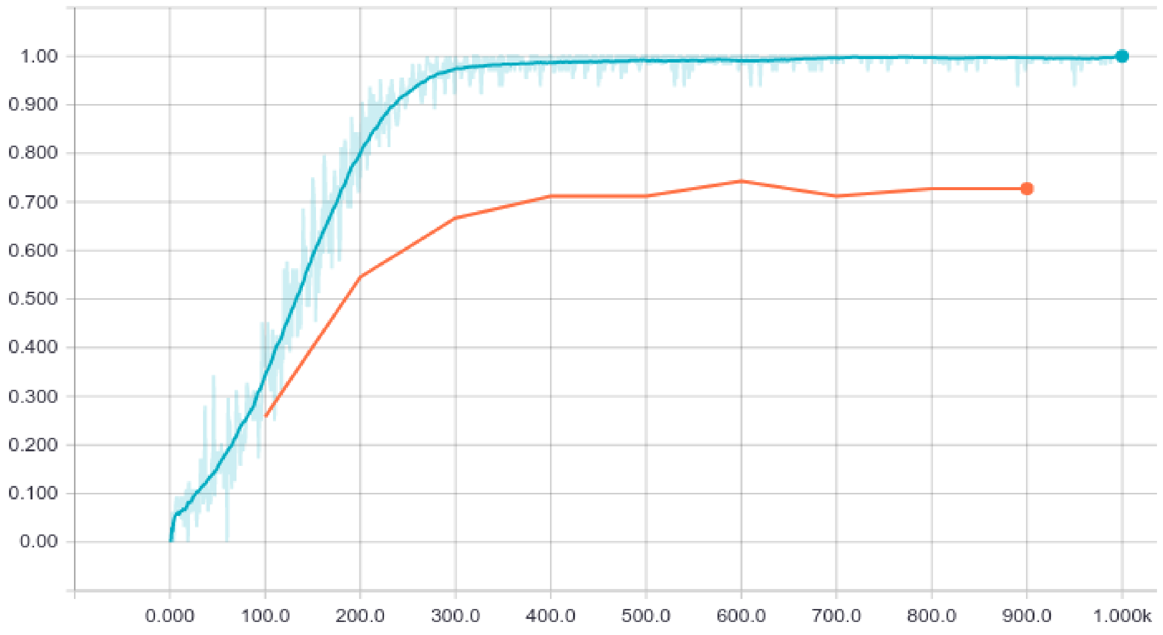

Figure 7.4: Model accuracy as measured on dev/test set of the main dataset, X axis stands for evaluation step, Y axis for accuracy, blue signal is accuracy on the training set, orange signal is accuracy on the test set, note that test accuracy stepped updating when model started overfitting

One serious scalability deficiency that *significantly* hurts classification performance on large sets of evaluation inputs is that due to regular expression based token replacement it becomes impossible to keep track of of where did the input string originate in input batch. At the moment this is done by matching tokenized input sequences back to the tokenized evaluation dataset however this substring search infers heavy performance penalty.

7.5 Model Tuning

Model has several configurable parameters that affect performance, scalability and accuracy. For complete list and implementation details on each parameter refer to A.

7.5.1 Number of Epochs

This model parameter signifies number of global iterations that model undergoes during training. As discussed in 7.4 on a small dataset the model can start overfitting and stop generalizing well very rapidly. Even on moderately sized dataset, in practice there was little improvement of accuracy of the test set past 7 or 8 full epochs.

7.5.2 Mini-batch size

Minibatches allow to reduce the variance of gradient updates and allow bigger step sizes **??** at the cost of introducing averaging noise introduced to the model. On the main dataset, various minibatch sizes have been experimented with and apart from batches of extreme sizes (>1000) when classification accuracy dropped bellow 50% or small size(<16) when accuracy dropped slightly and performance worsened significantly they didn't seem to affect classification accuracy on the test set. Size of 64 was chosen artificially for evaluation of the model.

7.5.3 Learning rate

Various learning rate values were evaluated for the main dataset, high values (>0.1) combined with low minibatch sizes prevented the model from generalizing on the dataset by adjusting the weights by big margins. Extremely small $1e^{-6}$ on the other hand slowed down the learning process significantly. Values suggested in the literature and existing NN implementations typically range between $1e^{-4}$ and $1e^{-3}$ and so the latter was artificially chosen.

7.5.4 Convolutional Layers

Originally, single convolutional layer was considered. After further research, 3-parallel layer design was adapted from [11]. This allows for skip-gram like model expressiveness even when using a simple one-hot vector encodings. Further evaluation has shown that as little as 3 convolutional layers with filters of varying sizes can significantly improve feature accuracy of the model (45% -> 75%+ on the test set). Evaluations done with increasing number of filters and convolution kernel sizes and number of filters per each layer had no significant impact on the accuracy performance of the main dataset.

7.5.5 Fully Connected Layers

Several combinations of number of connected layers and varying sizes were tested during model evaluation phase. Apart from a single layered design where a layer was directly connected to convolutional network and output layer (*mapping layer*), 2 or more layers of sufficient (200 nodes or more for main dataset) size seem to have made a significant enough difference in terms of performance on a test set. One layered approach with layer of 4096 in size resulted in good training accuracy (>93%) but fairly worse test accuracy (61%).

7.5.6 Sequence Length

Different sequence lengths (sizes of tokenized input sequences of the evaluated file) were tested and optimal size was found to be between 75 and 150 of tokens. Higher number of tokens in input sequence caused model to fit too closely onto the training data, whereas shorter sequences were causing worsened accuracy on both training and test set.

7.6 Experiments

Apart from supervised evaluation of the network, an unsupervised plagiarism detection approach was attempted by. For this purpose, a source file that contained 50% of source codes coming from one author and 50% coming from second author were concatenated into single file and presented to the network during evaluation phase with the idea that the network should react differently to each input. Manual inspection of source codes has shown that this was indeed the case, with input sequences from first author predominantly matching classes of two authors from training dataset, whereas second author author predominantly matched a single author in the dataset. This evaluation was done repeated with different source code files with similar results.

Second experiment dealt with experimenting with evaluation accuracy on pre-processed inputs. The main dataset was stripped of all C and C++ style comments and evaluated for test set accuracy. This significantly reduced the vocabulary size from over 30000 input tokens into just over 13000, but the accuracy on the testing set has worsened to 69% after

Chapter 8

Conclusion and Future Work

Theoretical part of this thesis researches and evaluates modern authorship attribution techniques used for source code authorship attribution with special attention given to neural network based techniques.

For implementation and evaluation part of this thesis, a combined approach of statistical analysis of code stylometry features and word embedding feature detector backed by a powerful feed-forward neural network based classifier was chosen.

Resulting implementation of this thesis presents a tool for authorship attribution of program source codes based on convolutional neural network and word embedding approach.

Furthermore, this tool can be easily extended by adding new feature detectors to the model or be extended to work with other tools.

Resulting source code authorship tool is capable of identifying the original author of a source code with high degree of probability upwards of 75% on unseen data in diverse dataset originating from GitHub online repositories.

Bibliography

[1] Aizerman, M.; Braverman, E.; Rozonoer, L.: *Theoretical foundations of the potential function method in pattern recognition learning*. 1964.

[2] Baisa, V.: Byte Level Language Models. 2016. [Online; retrieved 17.05.2017].
Retrieved from: https://is.muni.cz/th/139654/fi_d/thesis.pdf

[3] Caliskan-Islam, A.: De-anonymizing Programmers via Code Stylometry. online. [Online; retrieved 17.05.2017].
Retrieved from: https://www.princeton.edu/~aylinc/papers/caliskan-islam_deanonymizing.pdf

[4] Coyotl-Morales, R. M.; Villaseñor-Pineda, L.; y Gómez, M. M.; et al.: Authorship Attribution using Word Sequences. online. [Online; retrieved 17.05.2017].
Retrieved from:
http://users.dsic.upv.es/~prosso/resources/CoyotlEtAl_CIARP06.pdf

[5] Farabet, C.; Martini, B.; Akselrod, P.; et al.: Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems. online. [Online; retrieved 17.05.2017].
Retrieved from: http://yann.lecun.com/exdb/publis/pdf/farabet-iscas-10.pdf

[6] Firestine, B.: Celebrating nine years of GitHub with an anniversary sale. [Online; retrieved 17.05.2017].
Retrieved from: https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversary-sale

[7] Grefenstette, E.; Blunsom, P.; de Freitas, N.; et al.: A Deep Architecture for Semantic Parsing. online. [Online; retrieved 17.05.2017].
Retrieved from: http://yoavartzi.com/sp14/pub/gbfh-sp14-2014.pdf

[8] Guthrie, D.; Allison, B.; andLouise Guthrie, W. L.; et al.: A Closer Look at Skip-gram Modelling. online. 2014. [Online; retrieved 17.05.2017].
Retrieved from:
http://homepages.inf.ed.ac.uk/ballison/pdf/lrec_skipgrams.pdf

[9] Ho, T. K.: Random Decision Forests. online. [Online; retrieved 17.05.2017].
Retrieved from:
http://ect.bell-labs.com/who/tkh/publications/papers/odt.pdf

[10] Ioanis Kanaris, I. H., Konstantinos Kanaris; Stamatatos, E.: WORDS VS. CHARACTER N-GRAMS FOR ANTI-SPAM FILTERING. [Online; retrieved 17.05.2017].

Retrieved from:
http://www.icsd.aegean.gr/lecturers/Stamatatos/papers/IJAIT-spam.pdf

[11] Kim, Y.: Convolutional Neural Networks for Sentence Classification. online. [Online; retrieved 17.05.2017].
Retrieved from: https://arxiv.org/pdf/1408.5882

[12] Kingma, D. P.; Ba, J. L.: Adam. [Online; retrieved 17.05.2017].
Retrieved from: https://arxiv.org/abs/1412.6980

[13] Kjell, B.: *Authorship attribution of text samples using neural networks and Bayesian classifiers*. IEEE. 1994. ISBN ISBN-0-7803-2129-4. [Online; retrieved 17.05.2017].

[14] Maitra, P.; Ghosh, S.; Das, D.: Authorship Verification - An Approach based on Random Forest. online. [Online; retrieved 17.05.2017].
Retrieved from: https://arxiv.org/abs/1607.08885

[15] McCaffrey, J.: L1 and L2 Regularization for Machine Learning. 1 2015. [Online; retrieved 16.05.2017].
Retrieved from: https://msdn.microsoft.com/en-us/magazine/dn904675.aspx

[16] Mikolov, T.: Word2vec. online. [Online; retrieved 17.05.2017].
Retrieved from: https://arxiv.org/abs/1301.3781

[17] Pennington, J.; Socher, R.; Manning, C. D.: GloVe. online. 2014. [Online; retrieved 17.05.2017].
Retrieved from: https://nlp.stanford.edu/pubs/glove.pdf

[18] Raschka, S.: Naive Bayes and Text Classification. online. [Online; retrieved 17.05.2017].
Retrieved from:
http://sebastianraschka.com/Articles/2014_naive_bayes_1.html

[19] Rosenblum, N.; Zhu, X.; Miller, B. P.: Who Wrote This Code? online. [Online; retrieved 17.05.2017].
Retrieved from:
http://ftp.cs.wisc.edu/paradyn/papers/Rosenblum11Authorship.pdf

[20] Sadowski, P.: Notes on Backpropagation. online. [Online; retrieved 17.05.2017].
Retrieved from: https://www.ics.uci.edu/~pjsadows/notes.pdf

[21] Srivastava, N.; Hinton, G.; Krizhevsky, A.; et al.: Dropout. 2014.

[22] Sun, T.: Spam Filtering based on Naive Bayes Classification. online. 2009. [Online; retrieved 17.05.2017].
Retrieved from:
http://www.cs.ubbcluj.ro/~gabis/DocDiplome/Bayesian/000539771r.pdf

[23] Zając, Z.: Classifying text with bag-of-words. online. [Online; retrieved 17.05.2017].
Retrieved from:
http://fastml.com/classifying-text-with-bag-of-words-a-tutorial

[24] Zhang, X.; Zhao, J.; LeCun, Y.: Character level convolutional networks for text classification. [Online; retrieved 17.05.2017].
Retrieved from: https://papers.nips.cc/paper/5782-character-level-convolutional-networks-for-text-classification.pdf

[25] Zhang, Y.; Wallace, B.: A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification. online. [Online; retrieved 17.05.2017].
Retrieved from: https://arxiv.org/abs/1510.03820

Appendix A

Model Parameters

Model parameters are defined in their respective source code files - *cnn.py* for **model parameters**, *train.py* for **training parameters** and finally *eval.py* for **evaluation parameters**.

They can be either modified directly in source codes or by passing a command-line argument (*–param=value*) when executing scripts *train.py*, *eval.py*.

A.1 Train/Dev Split

During training of the model, input data (inputs, labels) are split into two sets of input data - **training set** and **dev set**. Split parameter *OPTS.data_split_ratio* defines the ratio as a real value <0..1>, with for example **0.8** value signifying 80%:20% split.

Unlike training set, dev set is never used to train the model (adjust weights) and is merely used to evaluate model accuracy on unseen data. Since the dataset is made of merged source files of a repository, there is inherent functional correlation between the test and dev set, nevertheless this data was used to compare different experiments for sake of accuracy comparison.

During the experimenting with the model, this ratio was kept constant at *0.9*.

A.2 Number of Epochs

Number of epochs signifies how many times will the model evaluate on the whole training dataset.
This parameter is defined as *OPTS.num_epochs* in source file *train.py*.
Default value chosen is 10.

A.3 Mini-batch Size

Mini-batch size (4.2.3) specifies the size of the mini-batch (number of tokenized input sequences) used for model evaluation.
Default value chosen is *64*.
This parameter is defined as *OPTS.batch_size* in source files *train.py*,*eval.py*.

A.4 Learning Rate

Learning rate parameter is the rate at which the model updates the weight given the input of a mini-batch (4.2.3).
This parameter is defined as *OPTS.learning_rate* in source files *train.py,eval.py*. Default value chosen is $1e - 3$.

A.5 Output Directory

Output directory specifies the directory into which to save the vocabulary, author names and other relevant model data during training phase. Furthermore, *neural network state* is being considered for saving each 100 iterations of the training, depending on whether the dev set accuracy has improved from previous 100step cycle or not.
This parameter is defined as *OPTS.out_dir* in source file *train.py*.
Default value chosen is *„/model/<system-date>*.

A.6 Input Directory

Input directory specifies the directory containing saved model from training phase (A.5).
This parameter is defined as *OPTS.model_dir* in source file *eval.py*

A.7 Convolutional Filter Sizes

Convolutional filter size is a comma separated string containing size of convolution filters. This parameter is defined as *OPTS.filter_sizes* in *cnn.py*. Default value chosen is *3,4,5* (creates 3 convolutional layers with kernel sizes of 3x3,4x4 and 5x5 respectively).

A.8 Number of Convolutional Filters

This parameter defines the number of convolutional filters per each convolutional layer. This parameter is defined as *OPTS.num_filters* in *cnn.py*. Default value chosen is 128.

Appendix B

Usage

The reference implementation uses two main python programs:
train.py for model training on a dataset represented with a directory a labeled files and *eval.py* which uses learned model to classify files in the evaluation directory and outputs a CSV formatted file with results (predicted class and vector of classification likelihood per each classified sequence in each file of the dataset).

B.0.1 train.py

train.py trains the model on provided dataset directory that contains labeled files. Labels are extracted from the filename base, if multiple files are to be assigned same label this can be done as label.<string> where **<string> part of the filename is ignored**. *Model output is automatically stored in directory model/<timestamp> or can be specified by switch out_dir.*

Example usage:

```
python ./train.py --dataset=<dir>
python ./train.py --dataset=<dir> --out_dir==./out
```

B.0.2 eval.py

eval.py evaluates the model and stores predictions in CSV formatted file. Model is loaded with the switch model_dir and input data directory is specified switch dataset.

Example usage:

```
python ./eval.py --dataset=<dir> --model_dir=<dir>
python ./eval.py --dataset=<dir> --model_dir=<dir> -o output.csv
```