Universidade Federal da Bahia
Instituto de Matemática e Estatística

Programa de Graduação em Ciência da Computação

A DEEP LEARNING APPROACH TO RECOGNIZE SOURCE CODE AUTHORSHIP

Roberto Sales Caldeira

TRABALHO DE CONCLUSÃO DE CURSO

Salvador
?? de dezembro de 2018

Universidade Federal da Bahia
Instituto de Matemática e Estatística

Roberto Sales Caldeira

A DEEP LEARNING APPROACH TO RECOGNIZE SOURCE CODE AUTHORSHIP

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Instituto de Matemática e Estatística da Universidade Federal da Bahia como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Maurício Pamplona Segundo

Salvador
?? de dezembro de 2018

Sistema de Bibliotecas - UFBA

Sales, Roberto.
A deep learning approach to recognize source code authorship / Roberto Sales Caldeira – Salvador, 2018.
21p.: il.

Orientador: Prof. Dr. Maurício Pamplona Segundo.
Monografia (Graduação) – Universidade Federal da Bahia, Instituto de Matemática e Estatística, 2018.

1. Exact algorithm. 2. Isometric embedding. 3. Maximum norm. I. Pamplona, Maurício. II. Universidade Federal da Bahia. Instituto de Matemática e Estatística. III Título.

CDD – XXX.XX
CDU – XXX.XX.XXX

TERMO DE APROVAÇÃO

ROBERTO SALES CALDEIRA

A DEEP LEARNING APPROACH TO RECOGNIZE SOURCE CODE AUTHORSHIP

Este Trabalho de Conclusão de Curso foi julgado adequado à obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Programa de Graduação em Ciência da Computação da Universidade Federal da Bahia.

Salvador, ?? de dezembro de 2018

Prof. Dr. Maurício Pamplona Segundo
Universidade Federal da Bahia

Prof. Dr. Professor 2
Universidade Federal da Bahia

Profa. Dra. Professora 3
Universidade Federal da Bahia

ACKNOWLEDGEMENTS

DIGITE OS AGRADECIMENTOS AQUI

DIGITE AQUI A CITACAO

—AUTOR (NOTA)

RESUMO

COLOQUE O RESUMO. Se preferir, crie um arquivo separado e o inclua via comando include.

Para evitar problemas de formato neste template (de uso geral), usamos acentuação mostrada abaixo.

```
\c{c} \~{a} \'{a} \^{e} \'{\i}
```

Não precisa fazer dessa forma, caso use pacotes adequados (latin1, etc.).

Palavras-chave: PALAVRAS-CHAVE.

ABSTRACT

Source code authorship identification is the task of deciding who is the author of a program given its source code. This is usually based on the analysis of previously collected samples from a set of candidate authors. There are several use cases for such a method, including attribution and detection of malicious codes, copyright infringement incident resolution, plagiarism detection, etc. As with texts in natural language, there are many distinguishing features in a piece code, like variable names, indentation style, etc. Some of these features are part of the coding style of a programmer. In this work, we investigate authorship attribution of C++ source codes based on the coding style of the authors. We also propose an end-to-end deep learning method for deciding if two source codes are from the same author, even if the involved authors are unknown to the system.

Keywords: software forensics, authorship identification, plagiarism detection

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Programmers often have to choose between tabs and spaces, between `while` loops and `for` loops, between positioning the open bracket in the next line or in the current line, etc. These are choices that can be regarded as their coding styles. Are these choices distinguishing features? In this chapter we will discuss what is style-based source code authorship identification, what challenges it poses, what has been done and what it is useful for. We will also introduce our contributions on the subject.

1.1 BIOMETRICS AND CODING STYLE

Biometrics is the field of computer vision that studies how certain human characteristics can be used to distinguish individuals. Even though physiological characteristics such as fingerprint, iris and face are probably the first ones that come to mind, the study of behavioral characteristics such as typing rhythm, voice, signature and writing style have brought new less intrusive means of distinguishing people. Even though using behavioral characteristics effectively has been made possible throughout the years, the fact that behavior is way more susceptible to change than physiological characteristics poses a big challenge.

Identifying authors of texts based on their writing styles is not a new topic (MENDENHALL, 1887). Throughout the years, computing has evolved and machine learning has reached its peek, making its way through writing style recognition. Narayanan et al. (2012) made it possible to identify the author of a text among tens of thousands of writers. It wasn't long until we were able to distinguish programmers by their coding styles (CALISKAN-ISLAM et al., 2015).

1.2 MOTIVATION

Throughout this work, we mainly consider the task of an investigator interested in deciding whether two anonymous pieces of code were authored by the same programmer or not. The actual authors of these pieces of code may be unknown to the investigator.

Also, these codes may aim to solve different problems, therefore the investigator intends to distinguish them solely based on stylistic features of such pieces.

We also consider easier scenarios where the set of possible authors are known to the investigator and labeled samples from this set are available.

We approach these problems from a deep learning perspective, training a deep neural model that can be subsequently used to resolve authorship of source codes and applying it to the different scenarios an investigator can face.

1.3 APPLICATIONS

Resolving source code authorship has a few real-world applications both in industry and academy. Although we haven't directly studied those, in this section we briefly describe a few of them.

1.3.1 Plagiarism Detection

Plagiarism can be generally defined as the unauthorized re-use of the work of another individual. Source code plagiarism is a widespread problem in academic institutions. Checking for plagiarism manually is time-consuming and not extremely effective, becoming impractical as the size of the code base increases.

Although automatic source code plagiarism detection is a recurring and well-studied problem (MARTINS et al., 2014), the approaches consolidated by widely used tools such as MOSS (SCHLEIMER et al., 2003), Sherlock (...) (tem uma TODO) and JPlag (PRECHELT et al., 2000) are mainly based on code similarity metrics which are greatly correlated to the task the code was written to solve.

For example, let's consider the specific case of *ghostwriting*, where the code the individual claims to be of his authorship was neither written by him nor copied from a colleague, but was actually written by another person (a former student, for example). It may not be possible to compare the suspicious code with another code by the same author, since the ghostwriter may actually be unknown. On the other hand, if pieces of code of the accused party are available, it's possible to determine if his coding style matches the coding style present in the suspicious code.

Also, an analyst may strongly suspect that a piece of code a programmer claims to be of his authorship is actually not, but have no clue of who the actual author might be. This can be modeled as a binary classification problem where positive samples are other pieces of code of the same author and negative samples are pieces from unrelated programmers.

1.3.2 Copyright Infringement

Software forensics is the science of examining source code and binary code in order to identify, preserve, analyze and present facts and opinions about pieces of software. Although it can also be used in civil proceedings, it's most often associated with the investigation of a wide variety of computer crimes, one of which is copyright infringement.

Code correlation analysis plays an important role at copyright disputes. In this case, an analyst has labeled codes from the involved parties and the task of determining if there was infringement or not.

1.3.3 Cyber Attack Identification

Cyber attack identification is a powerful application of software forensics in cyber security. Files left behind by an intruder during a cyber attack may have just enough information for an analyst to identify who the intruder is or to relate such an attack to a previous incident. Therefore, comparing the coding style of the attacker to those of authors of previous attacks and authors of public code repositories is of great interest to the analyst.

1.3.4 Exposing Anonymous Programmers

Although there are many helpful applications of source code authorship identification, systems capable of de-anonymizing programmers pose a threat to those who want to remain anonymous, in special for anonymous open source contributors (DAUBER et al., 2017). There may be good reasons for a programmer to be anonymous, like working in a software a hostile government doesn't like.

An example of a famous open source anonymous programmer is Bitcoin's creator, Satoshi Nakamoto. For example, if we had a set of labeled codes from programmers that are likely to be Nakamoto, we could try to match their coding style to the early versions of Bitcoin, of course, assuming that Nakamoto didn't try to obfuscate his own coding style.

1.4 CHALLENGES

Although comparison metrics have proved to work well for source codes, extracting features that encode the author's style and, therefore, are independent of the task being solved have proved to be challenging. For example, features such as methods and variables names can often be misleading. This task gets even more challenging as we need to select features that are steady across different programs and capable of distinguishing between programmers. In this work, we propose an end-to-end model that solves this problem.

Also, the environment the programmer is inserted can heavily affect the difficulty of the task. For example, in projects that have a rigid style guide to be followed, much less of the programmer's own coding style might prevail. We don't study the impact of such environments in this work. Moreover, in multi-contributor projects, usually powered by VCS (version control systems), certain pieces of code can contain contributions of many authors, turning the task of relating a single author to the style present on such piece very hard. Although we believe our contributions can be applied to multi-contributor environments, we leave this for future work.

In each of the mentioned applications, the claimed author may act adversarially and try to actively modify the coding style of the program. In the ghostwriting scenario, the involved parties may act together to make the style of the code written by one to match the other's. During a cyber attack development, an attacker may explicitly try to hide his own coding style. In a copyright infringement, the suspect may modify the code to match his own style. In this work, adversary interference is not considered.

1.5 RELATED WORK

Spafford and Weeber (1993) were among the first that suggested attributing source code based on style. Even though they suggested a lot of features, they did not propose an automated method nor a thorough analysis on how those features were useful. Hayes and Offutt (2010) examined the conjecture that programmers are unique and that this uniqueness can be observed in the code they write. They conducted an experiment with programmers and graduate students, and found that programmers do have distinguishable style features which they use consistently.

Ranking approaches to source code authorship attribution were proposed by Burrows and Tahaghoghi (2007), Burrows et al. (2007, 2009) and Frantzeskou et al. (2007). Burrows and Tahaghoghi used an information retrieval technique to solve the task, obtaining token-level n-gram representations of the source codes, building an index from these representations and querying that index for programs with unknown authors. The authors of the top-ranked programs were considered the authors of the queried program. Frantzeskou et al. used byte-level n-gram features to tackle the problem. An author profile is composed of the most frequent n-grams in training data of that author. Then, the author of an unclaimed program is considered the one with the most common n-grams to this code. Both works achieved high accuracy on very small suspect sets but didn't scale well.

Use of ASTs (abstract syntax trees) for authorship attribution was first introduced by Pellin (2000). Caliskan-Islam et al. (2015) have proposed using fuzzy ASTs and random forests to classify authorship of source code. Moreover, they proposed a coding style feature set for C/C++ source codes and a dataset for authorship attribution, based on Google Code Jam, which is a programming competition that resembles laboratory conditions. Dauber et al. (2017) showed that Caliskan-Islam et al. results could be extended to previously unexplored conditions, by adapting their techniques to work for small blocks of code of GitHub repositories.

Macdonell et al. (1999) introduced neural networks to the subject by using feed-forward neural networks and multiple discriminant analysis to attribute source codes. Bandara and Wijayarathna (2013) studied how deep neural networks could be competitive to previous methods given enough training data. Alsulami et al. (2017) applied LSTMs to the AST structure of a code.

1.6 CONTRIBUTION

In this work, we introduce the concept of *coding style descriptors*, which are compact representations that capture distinguishing stylistic features of a source code. We propose an end-to-end deep model that produces coding style descriptors from source code. Then, we study how the generated descriptors encode meaningful properties to the source code attribution problem by solving many of its variations.

We also introduce the Codeforces dataset for source code attribution, a C++ dataset with more than 30,000 samples extracted from Codeforces, a website specialized in holding online programming competitions. We briefly describe how the dataset was constructed and how it differs from previously published datasets.

Chapter 2

METHODOLOGY

In this chapter, we present a formulation for the source code attribution problem (Section 2.1), we describe how the Codeforces dataset was assembled (Section 2.2) and present a top-down approach to how the end-to-end model was developed in (Section 2.3).

2.1 PROBLEM FORMULATION

Although there are many variations of the source code attribution problem, in this chapter we will focus on one of them. In Chapter 3, we analyze other variations of the problem.

Given two source codes A, B, we want to determine if A and B were written by the same programmer. For that end, we have a dataset of source codes labeled with their authors, which can be used to train a classifier. However, the authors of A and B are not necessarily represented in this set.

2.2 DATASETS

The first step to develop an effective deep learning model is to gather enough training data. In this work, we decided to work with C++ source codes written in a laboratory environment – we assume the whole code is written by the author under no external style enforcement such as a style guide.

2.2.1 Google Code Jam

Although there are many public C++ laboratory datasets, the Google Code Jam[1] dataset (CALISKAN-ISLAM et al., 2015) is probably the biggest of them all. Samples from this dataset are collected from previous editions of Google Code Jam, an annual programming competition held by Google. In this competition, participants are given algorithmic tasks to be solved in a limited amount of time. Thus, it's very likely that code written by a participant manifests his own coding style.

[1]⟨https://codingcompetitions.withgoogle.com/codejam⟩

Google Code Jam holds nearly 10 rounds every year. Most of these rounds are eliminatory. Thus, the availability of samples from less experienced participants is expected to be low. If we want to build a balanced training set not biased by the way experienced participants code, we are limited by the small amount of code less experienced participants wrote.

Although this dataset was not extensively used throughout the development phase, it was a reference for the Codeforces dataset introduced in Section 2.2.2.

2.2.2 Codeforces

Codeforces[2] is a website specialized in holding online programming contests. Contest format is similar to Google Code Jam's, but they are not eliminatory. Thus, we are able to find a lot of samples from both non-experienced and experienced users.

We wrote a Python script that receives target constraints for the dataset and scrapes Codeforces for samples. Using this script, we assembled a balanced dataset with more than 30,000 C++ samples from nearly 2,000 authors, meaning that we have around 15 samples per author. This dataset was packaged and made public[3].

2.3 SOURCE CODE EMBEDDING MODEL

In this section, we propose a deep learning model that embeds source codes, from their string representations, into a denser latent space (Fig. 2.1). In Section 2.3.1, we describe what is a style descriptor. In Sections 2.3.3 and 2.3.4, we describe network architectures used in our work. In Section 2.3.5, we describe how our embedding network was trained to generate meaningful descriptors.

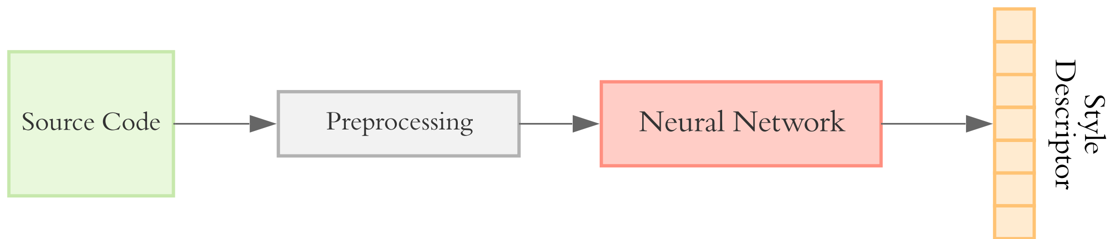


Figure 2.1: Overview of style descriptor generation pipeline.

2.3.1 Coding Style Descriptor

The performance of machine learning methods is heavily affected by the choice of data representation. Thus, much of the effort of the machine learning community has been

[2]⟨http://codeforces.com⟩
[3]link-pro-dataset

put into developing algorithms that transform otherwise unmanageable data into representations that can be effectively used by learning methods (BENGIO et al., 2013).

A Coding Style Descriptor (hereon referred simply as *style descriptor*) is a d-dimensional representation of a source code in a latent space. A latent space is a space where representations of similar objects lie close to each other. Therefore, the latent space of style descriptors should capture stylistic similarities of source codes. Ideally, style descriptors should encode everything a machine learning model needs to solve the problem posed in Section 2.1 and its variations. Thus, we can build simpler classifiers for these problems if we can provide a good embedding function $f(x) \in \mathbb{R}^d$, which maps source codes to d-dimensional descriptors.

Deep feed-forward networks are a natural approach to representation learning. In the remainder of this chapter, we will mainly study deep learning embedding techniques and apply them to our problem.

2.3.2 Preprocessing

The models we propose are end-to-end. Thus, the code is minimally preprocessed. Using Tensorflow static graph, it is not possible to support arbitrary input sizes in a batch. Therefore, we must crop our source code to a maximum line length M and a maximum number of lines L, converting it to a $L \times M$ char matrix. We chose to pick the last L lines of the code and the first M characters of each line. The positions that do not correspond to a char in the source code are masked out both during inference and during optimization. For the models we propose, we chose values of M and L that incurred the best improvement while keeping the training time and memory consumptions affordable.

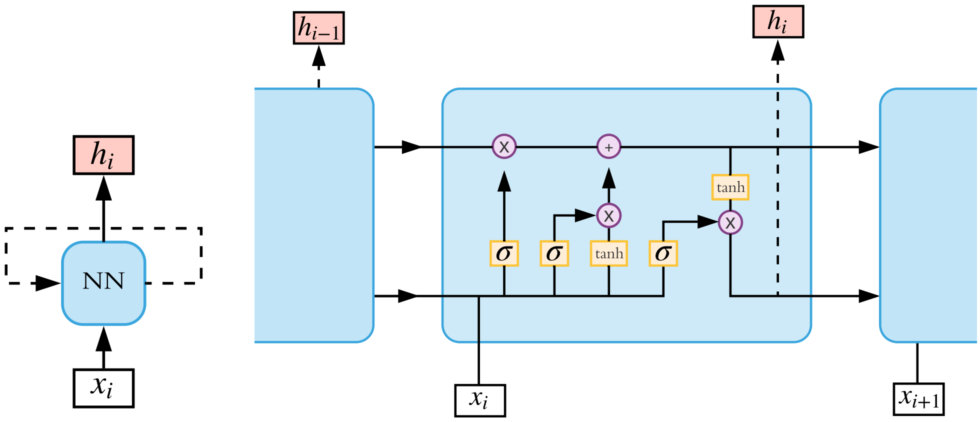
2.3.3 Model based on Convolutional Networks

Network Architecture

2.3.4 Model based on Long Short-Term Memory Networks

Recurrent neural networks (RNNs) were introduced to solve the lack of persistence of feed-forward networks. They are networks with loops in them. They are fed from an external input – a sequence x – and from their own output h (Fig 2.2a). Although generic RNNs are powerful and in theory are capable of learning any kind of sequence dependency, in practice they struggle to handle those that are long-term. The problems of training RNNs with gradient descent were studied by Bengio et al. (1994).

LSTM Long Short-Term Memory (LSTM) network was a special kind of RNN introduced by Hochreiter and Schmidhuber (1997). LSTMs were specifically developed to avoid the long-term dependency problem. It accomplishes that with its special cell design (Fig. 2.2b). During sequence unrolling, it learns what to remember and what to forget through carefully regulated gates – depicted as sigmoid layers. Moreover, besides being fed with its own output, it maintains an internal cell state which helps it to remember such dependencies. Although LSTMs usually produce sequences, it is a common procedure to



(a) A generic RNN (b) An LSTM cell

Figure 2.2: A quick view on the internals of an LSTM.

take only the last produced element as its output.

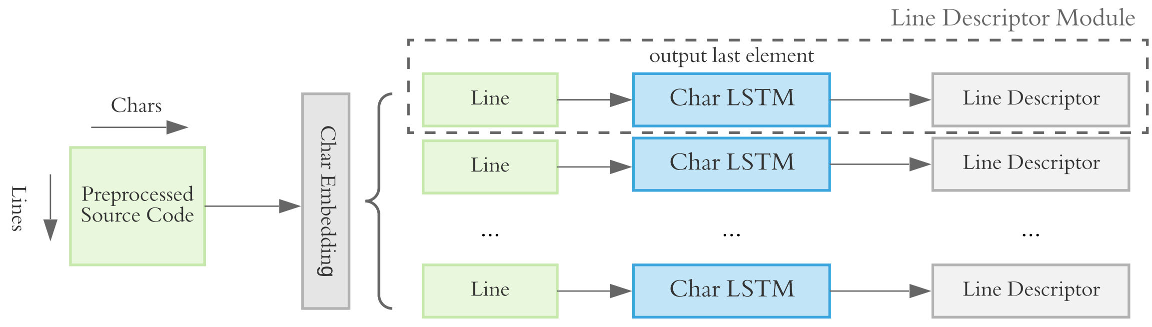
Given the efficiency of LSTMs, many researchers have focused on studying other variations of it (GREFF et al., 2015).

Bidirectional LSTM Long Short-Term Memory cells are very good at remembering. However, they can only infer based on previous elements of the input sequence. A bidirectional LSTM (IRSOY; CARDIE, 2013) is an extension of the usual LSTM that supports inferring based on both previous elements and subsequent elements of a sequence. It maintains two hidden layers: one for the left-to-right propagation and other for the right-to-left propagation. The results of these two passes are combined into a single result, usually through concatenation or averaging.

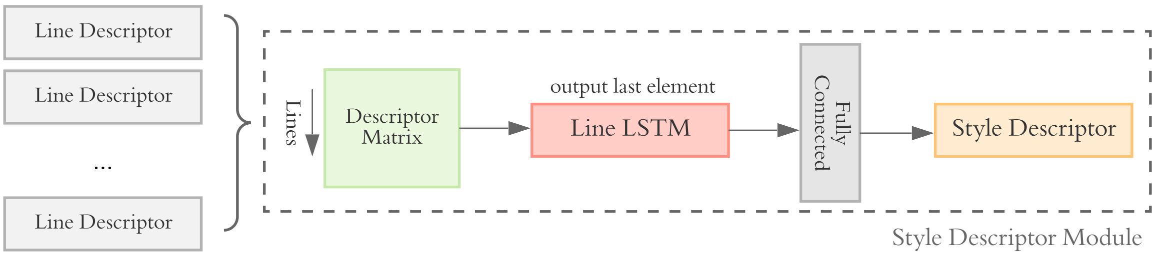
LSTM Stacking The sequence produced by a LSTM network can be re-used as input for another LSTM network. This is called LSTM stacking. As it is the case with other types of networks, deepening a LSTM network usually improves its performance (GRAVES et al., 2013). Intuitively, it allows each layer to independently learn different levels of abstraction.

Network Architecture Our proposed architecture is heavily based on bidirectional LSTM stacks and can be split in three parts: the char embedding layer, the line descriptor module (Fig. 2.3a) and the style descriptor module (Fig. 2.3b).

Char Embedding Layer Neural networks can't handle discrete types – like chars – naturally. Hence, we need to map the alphabet Σ of characters of source codes to real-valued vectors. We could simply convert each char to a $|\Sigma|$-dimensional one-hot vector.



(a) Char embedding layer and line descriptor module



(b) Style descriptor module

Figure 2.3: The architecture of the LSTM-based model.

As opposed to arbitrarily defining a mapping, we can also let the network learn it (GAL; GHAHRAMANI, 2016).

The char embedding layer is responsible for learning an embedding $f_c(x) \in \mathbb{R}^{d_c}$ that maps chars to d_c-dimensional vectors. Thus, each char in the source code is converted to a real-valued vector. Hereon, we will simply refer to char embeddings as chars.

Line Descriptor Module This module is responsible for learning an embedding $f_l(x) \in \mathbb{R}^{d_l}$ that maps lines of code to d_l-dimensional descriptors. Each line of the source code is fed – char by char – to the same char-level bidirectional LSTM stack. The last element of the sequence produced by this LSTM is taken as the line descriptor.

Style Descriptor Module The line descriptors generated by the line descriptor module are stacked back into a descriptor matrix. This module is responsible for learning an embedding $f_s(x) \in \mathbb{R}^{d}$ that maps descriptor matrices to d-dimensional style descriptors. Thus, the whole descriptor matrix is fed – line by line – to a line-level bidirectional LSTM stack. The last element produced by this LSTM is passed through a fully-connected layer and normalized to lie on the boundary of the d-sphere. The result is taken as the desired style descriptor.

We believe this architecture encourages the network to learn in a divide-and-conquer manner, by learning the individual features of each line and how to combine them into a single descriptor. Our selection of hyperparameter values for this architecture are given in Chapter 3.

2.3.5 Optimization

Although we decided the architecture of our models, we still have to make them learn. For that end, we review two optimization methods widely used in multi-class identification problems.

2.3.5.1 Softmax Cross-Entropy Loss The softmax function is commonly used in multi-class classification problems. In a d-class scenario, let $f(x) \in \mathbb{R}^d$ be the output of our neural network for a sample x. The softmax activation for this sample is given as

$$q(i) = \frac{e^{f(x)_i}}{\sum_j e^{f(x)_j}} \,. \tag{2.1}$$

$q(i)$ assume values ranging from 0 to 1, and $\sum_i a_i = 1$. Therefore, we can reinterpret $q(i)$ as the estimation of probability the sample x belongs to class i. The softmax cross-entropy loss is given as

$$\mathcal{L} = -\sum_{i=1}^{d} p(x,i) \log q(i) \,, \tag{2.2}$$

where $p(x,i)$ is the actual probability the sample x is from class i (usually a one-hot vector). Thus, by minimizing $\mathcal{L}$, we minimize the cross-entropy between the probability distribution p and an estimated distribution q.

Although softmax cross-entropy loss is a very powerful tool, it does not naturally account for the fact that the number of classes may be unknown. Although there are techniques to apply softmax in these scenarios (SUN et al., 2014; TAIGMAN et al., 2014), there are other optimization methods designed for such cases. Also, it is not inherently suited for generating descriptors. Hence, we restrict ourselves to use this method only when the number of classes is known.

2.3.5.2 Triplet Loss Schroff et al. (2015) introduced triplet loss for training embedding networks. In their work, the loss function is used in conjunction with a novel triplet mining algorithm to train an embedding network that maps images to descriptors. These descriptors are then used to solve face recognition. Moreover, triplet loss works in scenarios where the number of classes is unknown. Therefore, it is well-suited for deciding if two pieces of code are of the same person, even if they are unknown to the system. In this section, we will study the L_2 triplet loss.

The embedding is represented by $f(x) \in \mathbb{R}^d$. Additionally, we constrain this embedding to the boundary of a unit d-sphere, *i.e.* $\left\|f(x)\right\|_2 = 1$.

Let a, p, n (stand for anchor, positive and negative, respectively) be a triplet from the training set such that a and p have the same label (positive pair), but a and n have different labels (negative pair). Also, let $\mathcal{T}$ be the set of all possible said triplets. Then, the L_2 triplet loss is defined as

$$\mathcal{L} = \sum_{(a,p,n)\in\mathcal{T}} \max\Big(\left\|f(a) - f(p)\right\|_2 - \left\|f(a) - f(n)\right\|_2 + \alpha,\ 0\Big), \tag{2.3}$$

where α is a margin that is enforced between positive and negative pairs. If $\mathcal{L} = 0$, then for every triplet $(a, p, n) \in T$, it must be true that

$$\left\|f(a) - f(p)\right\|_2 + \alpha \leq \left\|f(a) - f(n)\right\|_2 . \tag{2.4}$$

When Eq. 2.4 is fulfilled, the negative pair of a triplet will be at least as far as the positive pair plus a margin α. Thus, by minimizing $\mathcal{L}$, we push the distance of positive pairs towards zero as we push the distance of negative pairs to be greater than the correspondent positive's by α. The advantage of this formulation is that, even though all training samples of the same class will form a cluster, they are not required to collapse to a single point. Fig. 2.4 shows a hypothetical scenario of optimization.

Generating all triplets from $\mathcal{T}$ would consider many triplets that easily satistfy Eq. 2.4. This would cause the training to converge slowly, since those triplets would still be fed to the network, but would not contribute to loss minimization. Therefore, it is crucial to select triplets that do not satisfy this condition to keep improving the model. These are called *hard* triplets.

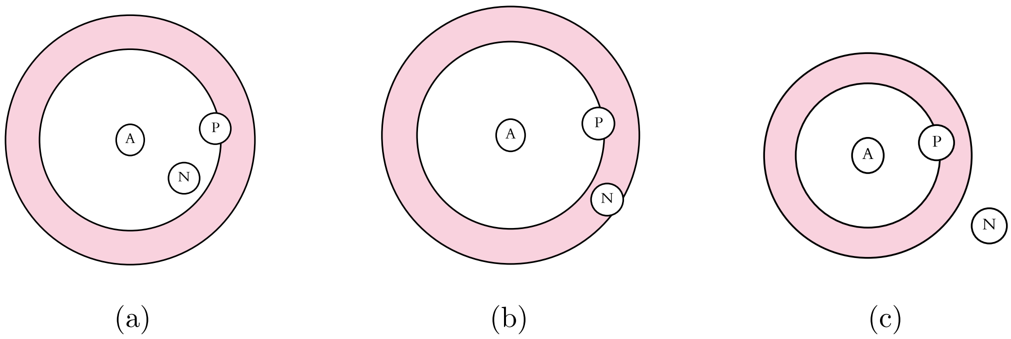


Figure 2.4: The region in red represents the margin area beyond p with diameter α. Before loss optimization (a), the negative pair is closer than the positive. During loss optimization (b), the negative pair is pushed further than the positive, but n is still in the margin area. After loss optimization (c), the positive pair is finally closer and n is beyond the margin.

Online Semi-Hard Triplet Mining One way to select hard triplets from the training set is to consider every sample as the anchor a. Then, select such p that minimizes $\left\|f(a) - f(n)\right\|_2$ and such n that maximizes $\left\|f(a) - f(n)\right\|_2$. This does not scale with the size of the training set. Moreover, it can cause outliers to dominate the selection process.

Schroff et al. suggested the online semi-hard triplet mining method to tackle both problems. Instead of picking hard triplets from the whole training set, we pick them from the mini-batch. Also, their work suggests that prioritizing triplets such that negatives lie in the margin area (Fig. 2.4b) helps avoiding local minima early in the training. Such triplets are called *semi-hard*.

Although in Chapter 3 we use softmax cross-entropy loss for comparison purposes, we mostly worked with triplet loss. Therefore, our main optimization flow is pictured in Fig. 2.5.

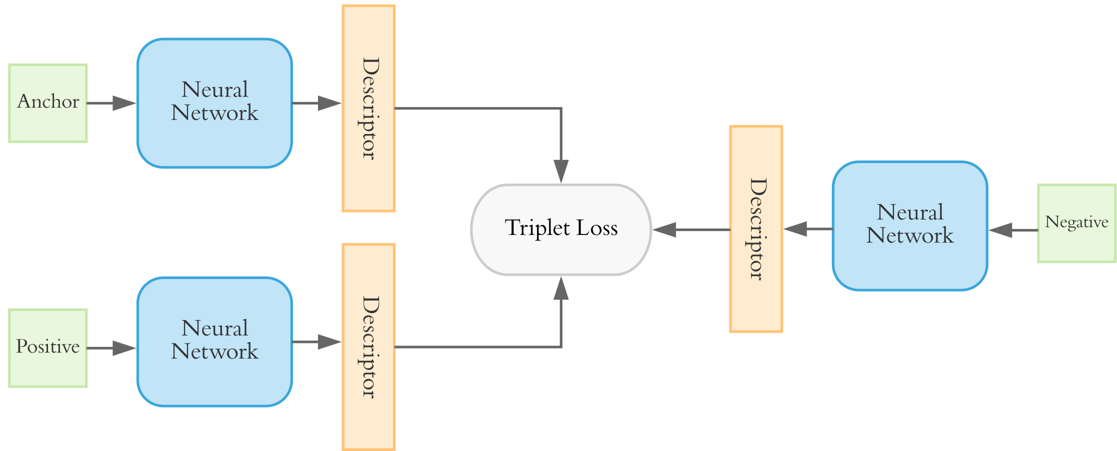


Figure 2.5: Overview of style descriptor generation pipeline.

Chapter 3

EVALUATION

Parameter	**Value**
maximum line length	(...)
maximum number of lines	(...)
d_c, char embedding size	(...)
d_l, line descriptor size	(...)
d, style descriptor size	(...)
char-level LSTM hidden units	(...)
line-level LSTM hidden units	(...)
fully-connected layer units	(...)

Table 3.1: Values for the hyperparameters of the LSTM-based model.

Chapter 4

CONCLUSION

BIBLIOGRAPHY

ALSULAMI, B. et al. *Source Code Authorship Attribution Using Long Short-Term Memory Based Networks*. 2017. 65-82 p.

BANDARA, U.; WIJAYARATHNA, G. Deep neural networks for source code author identification. In: *Proceedings, Part II, of the 20th International Conference on Neural Information Processing - Volume 8227*. New York, NY, USA: Springer-Verlag New York, Inc., 2013. (ICONIP 2013), p. 368–375. ISBN 978-3-642-42041-2. Available from Internet: ⟨http://dx.doi.org/10.1007/978-3-642-42042-9_46⟩.

BENGIO, Y.; COURVILLE, A.; VINCENT, P. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 35, n. 8, p. 1798–1828, ago. 2013. ISSN 0162-8828. Available from Internet: ⟨http://dx.doi.org/10.1109/TPAMI.2013.50⟩.

BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 5, n. 2, p. 157–166, mar. 1994. ISSN 1045-9227. Available from Internet: ⟨http://dx.doi.org/10.1109/72.279181⟩.

BURROWS, S.; TAHAGHOGHI, S. M. M. Source code authorship attribution using n-grams. In: *RMIT UNIVERSITY*. [S.l.: s.n.], 2007. p. 32–39.

BURROWS, S.; TAHAGHOGHI, S. M. M.; ZOBEL, J. Efficient plagiarism detection for large code repositories. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 37, n. 2, p. 151–175, fev. 2007. ISSN 0038-0644. Available from Internet: ⟨http://dx.doi.org/10.1002/spe.v37:2⟩.

BURROWS, S.; UITDENBOGERD, A. L.; TURPIN, A. Application of information retrieval techniques for source code authorship attribution. In: *Proceedings of the 14th International Conference on Database Systems for Advanced Applications*. Berlin, Heidelberg: Springer-Verlag, 2009. (DASFAA '09), p. 699–713. ISBN 978-3-642-00886-3. Available from Internet: ⟨http://dx.doi.org/10.1007/978-3-642-00887-0_61⟩.

CALISKAN-ISLAM, A. et al. De-anonymizing programmers via code stylometry. In: *Proceedings of the 24th USENIX Conference on Security Symposium*. Berkeley, CA, USA: USENIX Association, 2015. (SEC'15), p. 255–270. ISBN 978-1-931971-232. Available from Internet: ⟨http://dl.acm.org/citation.cfm?id=2831143.2831160⟩.

DAUBER, E. et al. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. *CoRR*, abs/1701.05681, 2017. Available from Internet: ⟨http://arxiv.org/abs/1701.05681⟩.

FRANTZESKOU, G. et al. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *IJDE*, v. 6, n. 1, 2007. Available from Internet: ⟨http://dblp.uni-trier.de/db/journals/ijde/ijde6.html\#FrantzeskouSGCH07⟩.

GAL, Y.; GHAHRAMANI, Z. A theoretically grounded application of dropout in recurrent neural networks. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. USA: Curran Associates Inc., 2016. (NIPS'16), p. 1027–1035. ISBN 978-1-5108-3881-9. Available from Internet: ⟨http://dl.acm.org/citation.cfm?id=3157096.3157211⟩.

GRAVES, A.; JAITLY, N.; MOHAMED, A. rahman. Hybrid speech recognition with deep bidirectional lstm. *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, p. 273–278, 2013.

GREFF, K. et al. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015. Available from Internet: ⟨http://arxiv.org/abs/1503.04069⟩.

HAYES, J. H.; OFFUTT, J. Recognizing authors: an examination of the consistent programmer hypothesis. *Softw. Test., Verif. Reliab.*, v. 20, n. 4, p. 329–356, 2010. Available from Internet: ⟨https://doi.org/10.1002/stvr.412⟩.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural Comput.*, MIT Press, Cambridge, MA, USA, v. 9, n. 8, p. 1735–1780, nov. 1997. ISSN 0899-7667. Available from Internet: ⟨http://dx.doi.org/10.1162/neco.1997.9.8.1735⟩.

IRSOY, O.; CARDIE, C. Bidirectional recursive neural networks for token-level labeling with structure. *CoRR*, abs/1312.0493, 2013. Available from Internet: ⟨http://arxiv.org/abs/1312.0493⟩.

MACDONELL, S. G. et al. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In: *ICONIP'99. ANZIIS'99 ANNES'99 ACNN'99. 6th International Conference on Neural Information Processing. Proceedings (Cat. No.99EX378)*. [S.l.: s.n.], 1999. v. 1, p. 66–71 vol.1.

MARTINS, V. T. et al. Plagiarism Detection: A Tool Survey and Comparison. In: PEREIRA, M. J. V.; LEAL, J. P.; SIMÕES, A. (Ed.). *3rd Symposium on Languages, Applications and Technologies*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014. (OpenAccess Series in Informatics (OASIcs), v. 38), p. 143–158. ISBN 978-3-939897-68-2. ISSN 2190-6807. Available from Internet: ⟨http://drops.dagstuhl.de/opus/volltexte/2014/4566⟩.

MENDENHALL, T. C. The characteristic curves of composition. *Science*, American Association for the Advancement of Science, ns-9, n. 214S, p. 237–246, 1887. ISSN 0036-8075. Available from Internet: ⟨http://science.sciencemag.org/content/ns-9/214S/237⟩.

NARAYANAN, A. et al. On the feasibility of internet-scale author identification. In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2012. (SP '12), p. 300–314. ISBN 978-0-7695-4681-0. Available from Internet: ⟨https://doi.org/10.1109/SP.2012.46⟩.

PELLIN, B. Using classification techniques to determine source code authorship. 2000.

PRECHELT, L.; MALPOHL, G.; PHILIPPSEN, M. *JPlag: Finding plagiarisms among a set of programs*. [S.l.], 2000.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: Local algorithms for document fingerprinting. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2003. (SIGMOD '03), p. 76–85. ISBN 1-58113-634-X. Available from Internet: ⟨http://doi.acm.org/10.1145/872757.872770⟩.

SCHROFF, F.; KALENICHENKO, D.; PHILBIN, J. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015. Available from Internet: ⟨http://arxiv.org/abs/1503.03832⟩.

SPAFFORD, E. H.; WEEBER, S. A. Software forensics: Can we track code to its authors? *Comput. Secur.*, Elsevier Advanced Technology Publications, Oxford, UK, UK, v. 12, n. 6, p. 585–595, out. 1993. ISSN 0167-4048. Available from Internet: ⟨http://dx.doi.org/10.1016/0167-4048(93)90055-A⟩.

SUN, Y.; WANG, X.; TANG, X. Deeply learned face representations are sparse, selective, and robust. *CoRR*, abs/1412.1265, 2014. Available from Internet: ⟨http://arxiv.org/abs/1412.1265⟩.

TAIGMAN, Y. et al. Deepface: Closing the gap to human-level performance in face verification. In: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA: IEEE Computer Society, 2014. (CVPR '14), p. 1701–1708. ISBN 978-1-4799-5118-5. Available from Internet: ⟨https://doi.org/10.1109/CVPR.2014.220⟩.