Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Code stylometry, a metric learning approach

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Andrea Gurioli

Correlatore:
Chiar.mo Prof.
Stefano Zacchiroli

Sessione III
Anno Accademico 2021/2022

Un sincero grazie ai Professori Maurizio Gabbrielli e Stefano Zacchiroli per avermi concesso questa fantastica opportunità, rendendo il lavoro non solo stimolante, ma anche piacevole. Ringrazio inoltre tutte le persone che mi sono state vicine durante gli studi, i miei genitori Sonia e Orlando, i miei amici di sempre, e i compagni di Erasmus per aver condiviso momenti unici.

Abstract

Authorship attribution, also recognized as code stylometry, has always been a milestone in obtaining important information for what concerns plagiarism and de-anonymization tasks, assessing the author in several different ways through the years. The proposed work revolves around the whole problem, starting with the mining of a new dataset which faces data scarcity and domain bias problems that afflicted the former works. Diving then into a new machine learning model design, derived from the former state-of-the-art techniques, which tries to gain advantages from Natural language process practices adopted by the newest language models. The problem is then tackled by moving through a metric learning technique, dealing for the first time with the stylometry problem as a querying snippet mechanism that allows a zero-shot inference over authors not present in the training set.

Sommario

L'elaborato presentato si pone al termine del progetto di ricerca intrapreso all'interno del programma Erasmus+ presso l'università Tèlècom Paris. Il soggetto della ricerca riguarda lo sviluppo di un tool volto a individuare la paternità del codice sorgente, usando come chiave di lettura lo *stile* dell'autore. Il lavoro viene frammentato in due sezioni principali:

- Sviluppo del Dataset
- Sviluppo del tool di stilometria

ricalcando il processo delineato durante la parte sperimentale della tesi.

Lo sviluppo del dataset si è reso necessario al fine di sopperire alle maggiori esigenze di dati poste dai nuovi modelli di machine learning. Inoltre si è posto l'obiettivo di utilizzare fonti che provenissero da progetti appartenenti a casi d'uso reali, mentre la quasi totalità degli sviluppi allo stato dell'arte sono basati su codice proveniente da competizioni algoritmiche. Il modello inerente alla soluzione del task di inferenza della paternità del codice sorgente mostra elementi di innovazione sotto diversi punti di vista. Partendo dal design della struttura, rimodellata con uno sguardo alle metodologie più moderne dal mondo del natural language processing, sino ad arrivare all'utilizzo del metric learning per osservare possibili potenzialità di una tecnica finora non utilizzata nel campo della stilometria. Gli esperimenti sono stati svolti seguendo un ablation study al fine di poter trarre conclusioni riguardanti la bontà delle soluzioni proposte.

Contents

1 Introduction **3**
Problem statement 4

2 Background and related work **5**
Background, a metric learning overview 5
Related work 7
A.S.T. dependent methodologies 8
The rise of the embeddings 10
Code2vec, a new approach for snippet embedding techniques . 11
Code2seq as an advanced technique for code embedding 15
Metric learning over code representation 17

3 Methodology **19**
Data mining 19
First repositories URLs mining 21
Lookup for Top authors over the initial repositories list 23
Final repos list expansion 24
Repositories cloning and experiment reproducibility 25
Dataset RAW preparation - data mining process 25
Author pruning and data undersampling 27
Model design 28
An adapted version of code2seq 28
Token embedding summation and Byte pair Encoding technique 28
AST embedding - from monolithic structures to LSTM reprojection 30
From combined context vectors to the attention module, adding an M.L.P. layer 31

Training loss 33
Classification baseline 35
Data pre-processing 36
Evaluation phase 40
Cosine similarity as embedding evaluation 41

4 Results **43**
Dataset 44
Hyperparameters 44
Dataset dimensions 46
Stylometer 49
Hyperparameters 49
Stylometer results 53

5 Discussion **59**
Dataset 59
Threats to Validity 61
Stylometer 61

Future works **63**

Conclusions **64**

Bibliography **67**

Chapter 1

Introduction

The code stylometry task is a problem stated to overcome anonymization and plagiarism, putting its roots over forensics applications, its use cases can spread across different domains. Nowadays, the problem is well related to machine learning methodologies, addressing the model's capability to infer the person behind the written code. As techniques improved over time, the task moved forward too, getting through different code vectorization techniques. The feature extraction phase for code vectorization, settles a big problem as much as the snippet encoding technique itself. Moving through time, three main kinds of features identify the data:

- Lexical features
- Syntactical features
- Layout-based features

Splitting the code representation by means of these three traits, allows us to infer the author's fingerprint by analyzing the stream of tokens in the source code(*lexical features*), by getting information about the graphic layout of the source code(*layout-based features*), or by analyzing the AST structure(*syntactical features*) which involves a code parsing phase. Several approaches exploit all of the mentioned features, obtaining a vector which represents snippets of code through all its nuances. Over time, when it comes to solving the stylometry task, syntactical and lexical features seem to overcome the layout-based ones, getting a better, not obfuscable snapshot of

what it's called the author's 'fingerprint'. In this work, we deep into a novel approach to code stylometry, trying to analyze the performances with a metric learning approach over the task and to overcome the typical class-constrained classification techniques related to the problem. We introduce a stylometer capable of addressing the task by exploiting a contrastive objective, modelling the latent space to achieve code embeddings closer by style, and opening the view to a similarity function author recognition over the snippets.

We outline then the model to a *zero-shot* author recognition, where the *zero-shot* label stands for a model capable of large flexibility, indeed, this kind of classification works even over unseen authors, differentiating itself from previous classical models which need fine-tuning when it comes to adding a class(in our case the author), making the model unfeasible without this further training step. Previous work[11] showed great results coupling semantically similar snippets of code for zero-shot clone detection, highlighting a path to other code metric learning domains like the one described above. We can then pose a first research question:

Is it possible to achieve effective latent space representations that group snippets of code closer by stylistic terms exploring metric learning techniques?

Problem statement

De-anonymization techniques spread across different use cases, making the task a valuable opportunity to integrate technologies that could overcome the problem from different points of view. Companies that migrate their system or buy source code that no one could address; open source works with several source code authors that anonymize themself by changing the account are just a few of the problems that can occur and could need the use of a model that helps on the author recognition phase. The focus has been addressed in two different steps:

- Creating an author-labelled dataset
- Creating a model to solve the Stylometry task

Chapter 2

Background and related work

Background, a metric learning overview

Before deepening into the problem, it's necessary to introduce the metric learning field and how it differs from standard classification techniques based on classification heads. As neural network techniques expanded in the years, research on latent space modelling has been tackled, observing how the input is projected over the embedded space and trying to reshape it on the assigned task. Standard classification techniques exploit as a final layer, a projection head channelled with a softmax activation function, modelling the output into a probability distribution. Architectures of this kind can be pruned by the final softmax layer, obtaining a model exploitable for feature extraction tasks. Major concerns for these techniques are scalability and flexibility. What happens if a new class is added to the dataset? The first answer would be exploiting transfer learning methodologies, enlarging the classification space to a new set of classes. This technique can yes, be part of a solution, but it does also need a large number of elements per class in order to be effective. Here comes the need for a model that can tackle open-world problems, being able to classify by clustering the elements, and generalise over seen and unseen classes. The softmax cross entropy trained model, pruned of the last classification layer can become an efficient embedding generator for a k-NN classifier, but could lead to arbitrarily large or small distances between each other, obtaining a latent space not useful for a nearest neighbour search space, especially when it comes to a high number of different classes task Shota

et. al [17]. Here comes the need for a new training paradigm, moving the loss from the classical Cross Entropy to a loss which takes into account the distances between the embeddings themselves. The distance in the embedded space should preserve the objects' similarity — similar objects get close and dissimilar objects get far away. Various loss functions have been developed for Metric Learning. For example, the contrastive loss guides the objects from the same class to be mapped to the same point and those from different classes to be mapped to different points whose distances are larger than a margin. Given $x \in X$ be an input data and $y \in \{1, ..., L\}$ be its output label we use x^+ and x to denote positive and negative samples of x, meaning that x and x^+ are from the same class and x is from a different class. The kernel f is the module that is responsible for the embedding, takes x and generates an embedding vector. m is the margin parameter which stops pushing clusters of different classes apart from each other.

$$L(x_i, x_j; f) = 1\{y_i = y_j\}\|f(x_i) - f(x_j)\|_2^2 + 1\{y_i \neq y_j\} \max(0, m - \|f(x_i) - f(x_j)\|_2)^2$$

Figure 2.1: Contrastive margin loss

Triplet loss is also popular, which requires the distance between the anchor sample and the positive sample to be smaller than the distance between the anchor sample and the negative sample; here the margin works as a distantiator between positive and negative samples Sun et. al.[18].

$$L(x, x^+; x^-; f) = \max(0, \|f(x) - f(x^+)\|_2^2 - \|f(x) - f(x^-)\|_2^2 + m)$$

Figure 2.2: Triplet margin loss

A common problem in these losses is the slowness of convergence, as a matter of fact, each sample is moved away from one single adversarial class at a time, the multi-class N-pair loss introduced by Sohn [19] propose a way to, given a batch of elements, bring the positive embeddings together while distancing the adversarial ones. With this technique, a constraint over batch construction is that there must be only one positive sample per class.

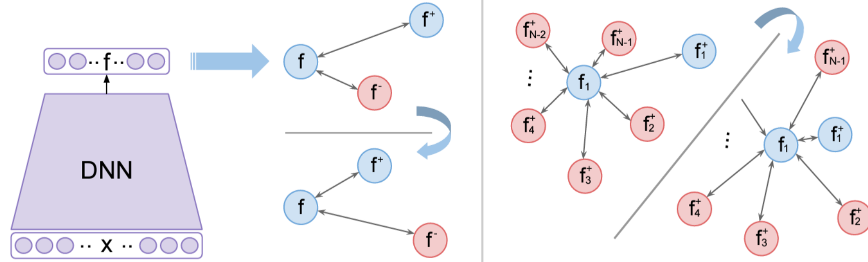


Figure 2.3: Difference between Triplet loss and N-pair loss approaches.

This proposal has been generalized over contrastive representation learning by Oord et al. [20] as InfoNCE loss.

$$L(x, x^+; x_i; f) = -\mathbb{E}_X \log \frac{\exp(f(x^+) \cdot f(x)/t)}{\sum_i^N \exp(f(x_i) \cdot f(x)/t)}$$

Figure 2.4: infoNCE loss, where N is the batch dimension and t is a temperature hyperparameter proposed by Wu et al.[21] and exploited by Jain et. al [22] for the code summarization task

Related work

Approaches to code stylometry have increased and developed through time, posing a step-by-step evolution of the solution by exploiting the edge of current machine learning technologies. The key point of all these approaches is the vectorization of the source code, indeed, obtaining a good representation in the latent space of the code's snippet poses the strengths of the model itself. As the need for a good representation income, feature selection and encoding techniques changed over time, moving the perspective to different techniques, trying to understand and improve the code understanding from another point of view.
Oman et. al.[1] work introduced the idea of human *fingerprint* over code. Starting from a human-driven classification, they demonstrated that it's possible to recognize who has written the source code, in the first place, just by looking at common visual patterns that occur in the code. This

experiment has driven the feature choice to a cluster-based classification, trying an unsupervised technique to infer the code writer. The main gap between this work and the further ones is the lack of Syntactic features, where, by syntactic features we mean the Abstract syntax tree-derived data.

Before going deeper, defining what an AST is becomes fundamental; we follow then the definition stated by the code2vec[8] work:

"An Abstract Syntax Tree (AST) for a code snippet C is a tuple $\langle$N,T,X, s,δ,$\phi\rangle$ where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of values, s $\in$ N is the root node, δ : N $\rightarrow$ (N $\cup$ T) is a function that maps a nonterminal node to a list of its children, and ϕ : T $\rightarrow$X is a function that maps a terminal node to an associated value. Every node except the root appears exactly once in all the lists of children."

A.S.T. dependent methodologies

Caliskan et. al. [2] stepped into the authorship problem by introducing the A.S.T. features usage, capturing the snippet vectorization not only by lexical and layout-based features but with syntactical features too. The paper tackles the problem from different points of view, showing how syntactical features are less sensible to obfuscation processes, obtaining a more reliable model. As the main AST features in this work we have the tree AST depth, showing how much the author tends to nest the code; the AST bigram frequency is then one of the key features in this research, stating that the model with the only usage of AST bigram features can have results close to the 'full feature' model. This shows the importance of the AST features over the others in the authorship task and lays the groundwork for further analysis of the feature magnitude in the classification process.

These features have been analyzed by exploiting the Information Gain technique, evaluating which one is more incisive over the class prediction. As the results highlights, the most important features(percentage of the number of features exploited) are:

- 54% lexical features
- 44% syntactic
- 1% layout based

Clearly opening a new way of thinking about the stylometry task, showing how syntactic features affect the inference process. Caliskan et. al. work was grounded over the Random forest technique, addressing the problem as a 229 classes classification over a Python Corpus; the average accuracy was 53.91% for top-1 classification, 75.69% for top-5 relaxed attribution over the GCJ[3] dataset, introducing the idea of the 'problem derived' model learning. The concept was grounded on the design of the Corpus itself; for instance, modelling a machine learning algorithm over a Dataset that contains snippets of code belonging to one single project, can lead the model to learn how to classificate the author by project, and not by style. The GCJ dataset is taken by the Google code competition, obtaining a corpus modelled by several problems per author, deflecting the previously mentioned problem.

A major interest point is to address these works over real use-case scenarios. Open source/big companies projects work with multiple authors over each single script of code, it's necessary to move the context to arbitrarily small snippets of code, checking the authorship. Dauber et. al. [4] address this problem by working over 'one line minimum' snippets of code. As the lines of code decrease, the overall accuracy drops heavily, highlighting how this factor afflicts the results. The Corpus taken into account was composed of C++

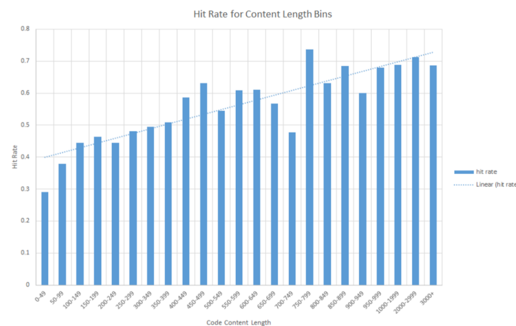


Figure 2.5: This plot shows accuracy, or hit rate, for samples grouped by the number of characters.

code from over 104 different programmers, posed by starting from 14 seed authors, moving then to contributors in order to increase the overall number of classes and samples, exploiting the Git blame functionality to address the code. The resulting dataset was composed of 150 samples per author with at least 1 line of code. The work was pursued with the Random forest classifier

as the Caliskan work, obtaining a single sample attribution accuracy of 48.8%

The rise of the embeddings

Since Mikolov[6] introduced the usage of word embeddings, outperforming previously SOTA NLP tasks, the overall techniques which could involve token features moved toward this path. The main idea behind Mikolov's work was about moving from sparse token representations, to dense ones, obtaining inputs that can be semantically valuable. In order to obtain the semantically rich embeddings, the approaches used were led by unsupervised language modelling techniques, masking part of the sentence and obtaining an output probability of the addressed masked word.

This approach outperformed all the previously stated works and delined the path for further approaches.

As the Mikolov approach was being used over NLP techniques, this approach is starting to be used over code stylometry tasks.

An initial approach to modelling tokens using this technique was brought by Alsulami et. al.[7]. The reference model is structured using recurring modules based on LSTM; specifically the results obtained were tested through uni and bi-directional networks, obtaining a metric of comparison between a simple model and a model capable of carrying out the computation also taking into account the future values within the context.
The architecture differs from the previous ones also thanks to its use of only syntactic features derived directly from the AST. We, therefore, have an an iterative algorithm that will initially move through a mining step of the AST tokens; tokens which will then be mapped to the optimized embedding space, trained during the backpropagation phase, represented therefore through a pre-established dimensionality as a hyperparameter. The embedding values will then be used as input for the LSTM, creating a hidden state tree useful for the final classification phase.

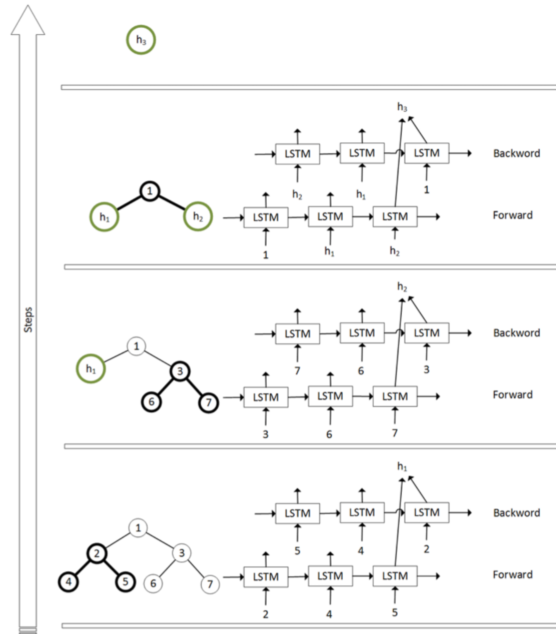


Figure 2.6: Example of the LSTM embeddings in the Alsulami et. al. work.

In this case, the model was trained over the Google code Jam dataset[3] as the Caliskan et. al. paper, working over C/C++ and Python source code, taking into account 10 authors for the C/C++ language and 70 authors for the Python language. The results show outstanding values, breaking the SOTA with 88.86% accuracy regarding the Python dataset, and 85.00% for what concerns the C/C++ code. These results demonstrate the capabilities of both, embeddings and Syntactic features in the code stylometry task.

Code2vec, a new approach for snippet embedding techniques

Alon et. al.[8] work aims to obtain a richer snippet representation combining syntactical and lexical features.
The novelties that this model proposes are:

- Novel use of path structure as a syntactic feature
- Triplet encoding between two terminal nodes and their path embedded

- Novel use of attention mechanism on snippet's embeddings

As mentioned before, the model works with triplets of embeddings, each triplet is defined by two terminal nodes extracted from the AST and the path between these terminal nodes. The path exploited for the embedding is mined starting from the terminal nodes taken into account, looking for the lowest common ancestor in the Abstract Syntax Tree. The extracted path is then represented as a string containing non-terminal nodes and the direction, also mined during the processing phase. The embeddings used for the models are so the representations of the two connected terminal nodes and the embedding of the path String.

$$\langle x, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), 7\rangle$$

Figure 2.7: embedded triple (terminal,path,terminal) for the statement "$x = 7$"

The embedded Triple is then reprojected in a defined space with a fully connected layer, with, as an activation function, the hyperbolic tangent function. The aim of this layer is to learn a combined representation of the three embedded features, moving them into a new latent space.
As it has been obtained the new representations of the arbitrarily chosen Triplets, the aim moves now into leading to a combined representation of the given embedded triplets(context vectors or path contexts). Here the concept of soft and hard attention has been introduced obtaining a weighted representation of the contexts as a final Vector for the entire snippet of code. Soft and hard attention were defined as:

- Soft attention: *weights are distributed "softly" over all path-contexts*
- Hard attention: *selection of a single path-context to focus on at a time*

Both of these attention mechanisms were tested during an ablation study; the soft attention weighting mechanism obtained better results over the code summarization task, and it has been used over the final version of the model. Here are the formulas for the Soft-attention mechanism:

$$attentionWeight\ \alpha_i = \frac{\exp{(c_i^T \cdot \alpha)}}{\sum_{j=1}^{n} \exp{(c_j^T \cdot \alpha)}}$$

Figure 2.8: Computation of the attention weight, where alpha is the attention matrix and c are the context-path vectors

$$codeVector\ v = \sum_{i=1}^{n} (\alpha_i \cdot c_i)$$

Figure 2.9: Computation of the final snippet representation

The final 'Snippet vector' can be then exploited as the embedding of the addressed snippet, adapting it depending on the task.

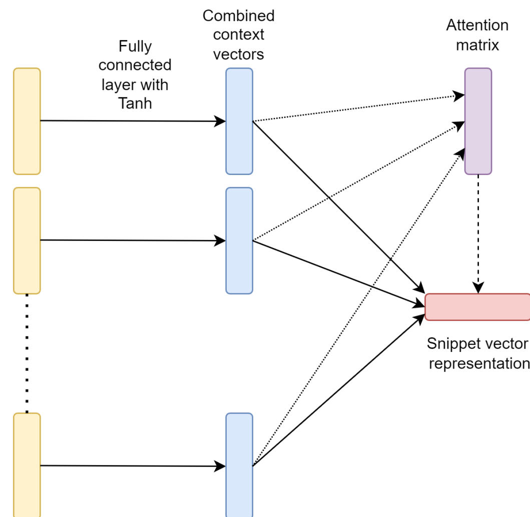


Figure 2.10: Code2vec model

The main problems addressed in this model are given by the sparseness of the data, making the model extremely Data hungry. Terminal values are represented as whole symbols, obtaining different embeddings for semantically similar tokens(eg. *"OldArray"*, *"NewArray"*). The monolithic representation of the paths suffers from the same problem, having different

embeddings for similar paths. These techniques suffer also from the problem of out-of-vocabulary(OoV) tokens. In this domain, where the variable names are so variegated and depending on the programmer itself, it's easy to encounter semantically equal names but written in different ways.
These problems are tackled by the model Code2Seq[13], which we will talk about later.

As the Snippet embedding task has been improved by the development of code2vec, Code stylometry research moved in this direction with the works described by Bogomolov et. al.[9] and Kovalenko et. al.[11].

The first aim of Bogomolov's work is to compare standard machine-learning techniques to the code2vec approach. The first results that were highlighted by this work are the outperforming accuracies compared to models without syntactic features, confirming the valuable impact that these kinds of features have.
Another important result is about how the Random Forest model outperformed code2vec with smaller amounts of data, bearing out even this time the concerns about the data hungriness of the code2vec model. Bogomolov's work with code2vec reached good results, pushing the accuracies over the GCJ dataset up to the SOTA level(previously 88.86% with the LSTM Alsulami work) for what it regards the Random forest technique(95.9%), and lower accuracies for the code2vec model(72.3%).
Kovalenko et. al.[11] work shifted slightly from previous approaches by looking at the code differences between authors; a context change like this one needed a specific dataset, mined appositely for the task from the IntelliJ Community taking Java snippets of code. The model indeed extracts from the AST only the differences between the previous code and the newest one, addressing them to the pointed author. The main focus of this work is the time-labelled snippet, showing the weight and the impact of self-improvement over years over the latent space, grounding the evaluation phase, not over the author recognition, but over the author's embeddings distances from each other.

These techniques opened solutions for modern tasks. Azcona et. al.[10] propose a vectorization technique of the user aimed at showing students' criticalities in terms of school learning, by embedding student assignments. It is therefore possible get a meaningful representation of the users engaged within of a latent space of dimensionality decided a priori, showing how the

style of programming is fundamental in the differentiation of users.

Author	Dataset-S.Dim.	N. Authors	Technique	Language	Accuracy (%)
Caliskan	GCJ-Source	229	R.F.	Python	53.91
Dauber	Custom-Fragment	104	R.F.	C++	48.80
Alsulami	GCJ-Source	70	L.S.T.M.	Python	88.86
Alsulami	GCJ-Source	10	L.S.T.M.	C++	85.00
Bogomolov	GCJ-Source	70	code2vec	Python	72.30
Bogomolov	GCJ-Source	70	R.F.	Python	95.90

Above the table of accuracies per dataset type (if Source it has been used the whole file, otherwise fragment), number of authors, technique and language.

Code2seq as an advanced technique for code embedding

Considering the previous code2vec model, tackling the highlighted problems was the principal concern for an improvement over code embedding tasks. As we can infer from the name, Code2seq [13] is a model mainly designed for having as an output a sequence, not anymore the single vector representation of the embedded snippet; these design decisions are mainly addressed to the code summarization task.

In order to solve the OoV problem over the monolithic path representation and the terminal nodes, Code2seq changes completely the Context-Path embedding, moving from monolithic embeddings to composed representations of the inputs.

The first shrewdness is about the Terminal tokens embeddings. As we mentioned before, *"SortArray"* and *"ArraySort"* over the code2vec proposal have two different embeddings for two semantically equal tokens. When it comes to the code2seq model, the proposal was to split the tokens by exploiting commonly used programmers' habits like camel notation and underlining character values through regex techniques; once obtained two different tokens, the embedded values of the tokens are summed, obtaining semantically rich embeddings representation composed by split elements. In this case, *vector("Array") + vector("Sort")* and *vector("Sort") + vector("Array")* lead to the same representation in the latent space, improving the data sparsity problem mentioned before; even semantically similar embeddings like *vector("Array") + vector("Old")* and *vector("Array") + vector("New")* are led by design to closer embeddings.

This technique takes up from the findings made by Mikolov over the embeddings where the results showed the chance of vectorial operations over embeddings, highlighting that *"simple algebraic operations are performed on the word vectors, it was shown for example that vector("King") - vector("Man") + vector("Woman") results in a vector that is closest to the vector representation of the word Queen"*[6]

The second point of weakness over the code2vec model was about the path representation, even though the paper showed how almost all the monolithic paths appear more than one time, it remains still a source of sparsity, which is tackled by splitting the path into pieces(non-terminal nodes) and representing it exploiting a bidirectional LSTM, based upon non-terminal token embeddings.

As the model is designed for the code summarization task, the final part of code2seq is based on a decoder modelled with an attention mechanism which selects dynamically which path-context representation influences the most the output.

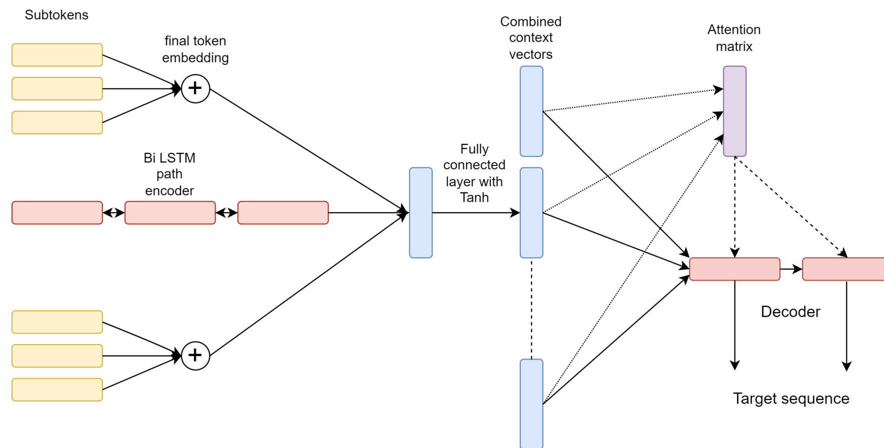


Figure 2.11: Code2seq model for code summarization task.

Code2seq's improvements over the code2vec model are the modules which inspire the final model used in this work for the code stylometry task, remodelling this work to obtain a vector representation as the code2vec model.

There have been attempts to recreate language model pretraining with the work of Feng et. al.[15] with CodeBERT, tested over the code summarization task, obtaining worse results than the code2seq model. This has been addressed to the absence of the AST syntactic data, highlighting even this time the importance of the AST features during the code representation in the latent space.

Metric learning over code representation

As metric learning techniques expanded and research over it improved the overall results through time, the representation of code in the latent space has followed this path with the introduction of Contracode[22]. The first aim of this project was to explore the concept of program representation learning based on functional equivalence, moving the embeddings closer in a matter of semantic similarities, and not just syntactic ones. For this purpose, the loss exploited in the metric learning environment was the infoNCE loss proposed by Oord et al. [20], obtaining a representation of the snippets which converges between one similar class at a time, diverging from N-1 classes where N is the dimension of the batch during the training phase. The proposal, works in an unsupervised way, exploiting a data augmentation phase per sample, keeping the semantic structure of the code unaltered and changing the syntactical structure with different techniques. Results showed an apparent model capacity of embedding code snippets by semantic similarities, grouping functionally-wise snippets.

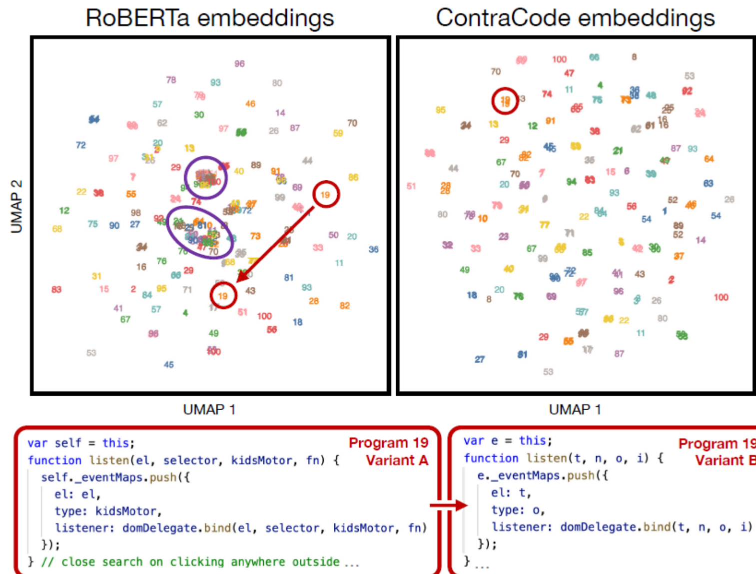


Figure 2.12: A UMAP visualization of JavaScript method representations learned by RoBERTa and ContraCode, in $\mathbb{R}^2$. Programs with the same functionality share colour and number.

The proposed work was based on the idea that functionally-wise snippet embeddings could lead to a valid pre-trained model for downstream tasks, obtaining a prior latent space beneficial for a further fine-tuning phase. An initial proposed test to validate the model is a zero-shot clone detection, exploiting the cosine similarity between the embedded snippets with a thresholding technique indicating what is cloned and what is not. Bui et. al. [23] followed the same path, led by unsupervised learning techniques, but focusing the project on the code retrieval task, embedding the snippets with code2vec and code2seq encoders. Bui et. al. pose an important Research Question which led the expectations of this work with the results of [22]

Are the code vectors generated by the pre-trained models useful in the unsupervised space-searching task?

the positive results of this research question open to us the view for a domain change, moving from code semantic similarity to code style similarity.

Chapter 3

Methodology

The proposed work is composed of four main steps, assessing a work path for the development phase:

- Data mining of the source code dataset, labelled per author-name
- Model design
- Data pre-processing, with a parsing step to retrieve the AST
- Experimental validation

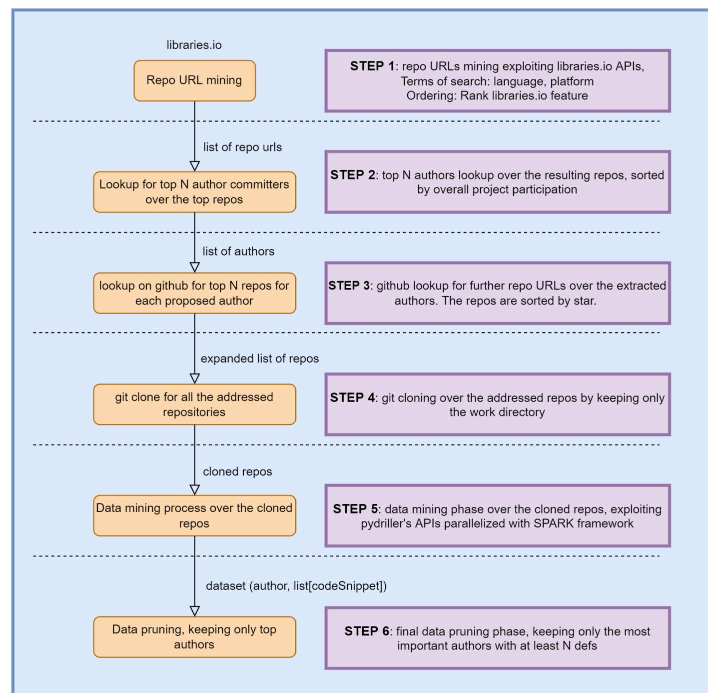
Data mining

Creating a good Corpus for this approach is one of the cornerstones in order to get a well-performing model. Data should be mined taking into account several incoming problems; as the pre-processing phase needs data that must be parsable, getting valuable input for this statement is a key point.

Other key points that have to be taken into account are the milestones to obtain a great Corpus: *"In building a corpus of a language variety, we are interested in a sample which is maximally representative of the variety under examination, that is, which provides us with as accurate a picture as possible of the tendencies of that variety, including their proportions"*[12]. Golden rules for the Corpora-making process should guide our work even on our domain-shifted task from typical NLP problems. As we mine data

we should project our emphasis on what results represent. It's easy to make mistakes when it comes to selecting the repositories to look into(where we mine data); people with low-profile experience in coding could lead to rapid stylistic changes over time, biasing the latent space with noisy data. Indeed, expert authors who follow the quality of code guidelines should have more consistent representation over time; this poses constraints over our variety representation, moving from a full range to a narrower, well-defined Corpus. Another essential feature to take care of is the representation of the style of the programmer rather than the functionalities shown in the code. The mining should take care of different repositories about the same author, tackling the problem stated by Caliskan[2].

Obtaining a consistent dataset which reflects the points highlighted above, requires a clear design methodology to follow, the schema below shows the six main steps used to represent our population:



First repositories URLs mining

Having as the first objective the need to work with Python's projects which follow the standardized guidelines of quality of code and to make sure to work with well-outlined and consistent coding styles, we have to define a good metric in order to rank the projects to mine. Munaiah et. al.[25] discerns the idea of the quality of a project, differentiating it from its popularity. Here the first definition of an engineered software project becomes crucial for an initial ranking proposal:

An engineered software project is a software project that leverages sound software engineering practices in one or more of its dimensions such as documentation, testing, and project management.

Settled a good starting point for a Corpus lookup, Libraries.io[24] makes a good fit for an initial repositories URL proposal, exploiting a ranking algorithm described below. Libraries.io indexes data from more than 6M packages from 32 package managers. It monitors package releases, analyses each project's code, community, distribution and documentation, and maps the relationships between packages when declared dependent. The 'dependency tree' that emerges is the core of the services that it provides.

We took the indexed data from libraries.io as the starting lookup point for repos since the indexing is sorted by '*SourceRank*', defined as: the name of the algorithm used to rank search results. The maximum score for SourceRank is currently 30 points.

The analysis is broken down into:

- Code
  - Affects the score if the package has outdated dependencies: $Score = Score - 2$
- Community
  - How many 'stars' does the project have?
    $Score = Score + log(stars)/2$
  - How many contributors does the project have?
    $Score = Score + log(contributors)/2$

- How many ‘subscribers’ does the project have?
  $Score = Score + log(subscribers)/2$
- Has there been an update within the last six months?
  $Score = Score + 1$

- Distribution
  - Is there a link to the source code?
    $Score = Score + 1$
  - Does the project use versioning?
    $Score = Score + 1$
  - Does every version use semantic versioning?
    $Score = Score + 1$
  - Has the project reached version 1.0.0 yet?
    $Score = Score + 1$
  - Is the project more than six months old?
    $Score = Score + 1$
  - Has the project had a release within the last six months?
    $Score = Score + 1$
  - Are all published versions marked as ‘pre-release’ by the maintainer?
    $Score = Score - 2$
  - Has the project been removed from the package manager?
    $Score = Score - 5$

- Documentation
  - Does the project have a readme file?
    $Score = Score + 1$
  - Does the project have a valid license?
    $Score = Score + 1$
  - Does the project have a description, homepage, repository link or keywords?
    $Score = Score : +1$

- Is the project marked as deprecated by the owner?
  $Score = Score : -5$
- Is the project marked as unmaintained by the maintainer?
  $Score = Score - 5$

- Usage
  - How many Projects are dependent on this project?
    $Score = Score + log(dependent_projects) * 2$
  - How many Repositories are dependent on this project?
    $Score = Score + log(dependent_repositories)$

Libraries.io gives the possibility through APIs for querying, exploiting the SourceRank score by pre-established filters like platform and main programming language.

Having the need for a large number of repositories for the initial look-up list, it's easy to exceed API query limits; indeed, the initial mining URL repositories model design is based on a waiting algorithm which prevents the system to crash, freezing the miner until a positive answer from the query is reached. The repositories mined, elaborated page per page as the framework of libraries.io needs, are filtered by regular expression to match the presence of the word *github* inside of the URL.

Filtering the projects for being part of the GitHub hosting service is a constraint given to move later over the author and repository queries, exploiting only one API service, and moving through a thinner and more reliable system to expand the initial repositories list.

Lookup for Top authors over the initial repositories list

The second mining process stage is designed to expand the initial number of repositories by looking up to the prominent project authors. This stage is directly channelled to the next one and it takes as input the initial repositories and it outputs the authors' list.

The process is pursued as the former phase with the libraries.io APIs which ranks the authors by importance, where the importance is given by the overall participation in terms of project commits. Obtaining a list of authors gives

us the chance to move over a high-level quality of code authors, addressing the further look-up in a well-defined direction.

Final repos list expansion

The third stage is the one which tackles the data scarcity problem by expanding the initial repository list with an author-driven look-up. Github[26] APIs are in this case the framework exploited for the querying process. As mentioned before, moving through the only usage of GitHub as a query framework is designed to ease the repository URL extraction phase.

For each user, mined by the previous stage, a ranked look-up repository is applied. In the beginning, the first query is modelled to retrieve the user's meta-data and extract the repositories URL list addressed to the user himself. Once the repositories URL list is mined, the next step is to extract the list's meta-data, giving us the chance to mine and sort the repositories by importance and filter them by need. The second query, as described above, gives us a list of repositories and meta-data for each one; at the beginning, an initial filtering phase is applied, pruning the repositories that are:

- Private repositories
- Not in Python language

Populating then a list of tuples $(RepoURL, RepoStars)$. The decision to keep the stars for each repository is given by the need of sorting the results by importance, keeping then high-profile repositories. The final repositories list is finally sorted by GitHub stars and pruned to the maximum amount of repositories per user given previously as input.

Extracting different repositories per author helps to differentiate the author's repositories, creating a heterogeneous dataset by means of semantic content, and increasing the overall variance.

The overall process is parallelized by exploiting the Pyspark framework; the framework gives us the chance then to apply a *distinct* method, pruning the duplicates by terms of URL.

Repositories cloning and experiment reproducibility

The fourth step is the one that prepares the data for the final data mining process. As we want to assure the experiment's reproducibility, the need of tracking the timestamp of the repositories comes, assuring a way to extract the data from the same commit as the one extracted by us.

The cloning phase over the unique URLs repositories list obtained by the previous steps is processed initially by cloning the repository with the *–bare* command, assuring to extract a lighter version of the folder itself [27]. With the git clone –bare command

instead of creating <directory> and placing the administrative files in <directory>/.git, make the <directory> itself the git directory. When this option is used, neither remote-tracking branches nor the related configuration variables are created

The lightening is given by the absence of the working tree, getting a repository useless for the development phase, but with the needed data for further mining. As mentioned before, after the repository has been cloned, the next step is to write in a separate file *.txt* the URL of the repository with the relative timestamp. As the step before, the whole phase is parallelized through the usage of PySpark, fastening the whole process.

Dataset RAW preparation - data mining process

As we dive inside the core of the data mining phase, it's necessary to review the ideas behind this process. In order to obtain data which could work over several different models, the ones that need the snippet's AST requires *parsability*, instantiating a big constraint over the process itself. Once one of the prior tying factors has been settled, a decision step has to be tackled; the idea of getting through open-world problems, where big snippets of code are developed by multiple programmers, leads us to achieve parsable data structures as thin as possible, keeping the semantic validity of the code and the fingerprint style of the author.

These considerations that we get through led us to shrink the search space starting with *function definitions* as a primary point of disambiguation. The last point, but not for importance, is the author's univocity. During the mining phase, when it comes to code stylometry, making sure that the script

comes from only one author is mandatory. The tool exploited during the process is the git diff command, which prevents us from adversarial labels (wrong class assumption) by posing the author who changed every single line.

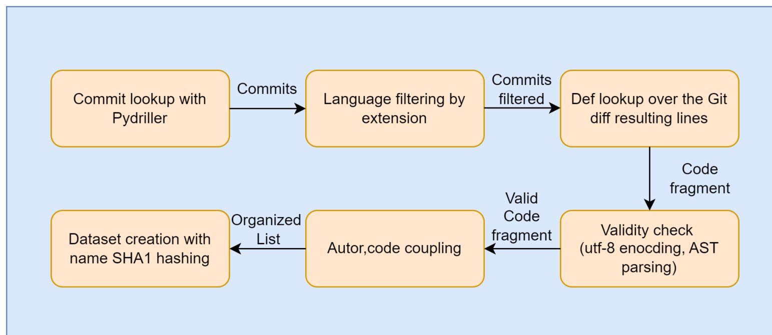


Figure 3.1: Phase design of the step5

The combination of the git diff procedure and the function definition assumption gets us as close as possible to a great, univocal per-style dataset.

In order to retrieve the diffs, the need to extract each commit from the repository comes. As a background framework, it has exploited the pydriller package [28], allowing us to move easily through project commits, filtering them by language(Python).

As the commit has been extracted, the process can be mainly split into two different parts:

- Not congruent extension pruning
- Data extraction

By checking the file extension, the first part prunes all the files that don't belong to the language addressed.

The second part is the core of the mining by which each modified file is initially split into lines, exploiting a dictionary that associates each line number to its content. The following part tackles the function definition lookup by exploiting regular expressions, if the inspected line contains a function definition, a temporary list is created, inserting all the following contiguous lines till the dictionary key has a missing index, which indicates that the snippet comes to a breaking part or another function definition.

After the definition in the snippet has been spotted, a validity check needs to be accomplished. The extracted snippet is at first utf-8 encoded, making sure to have a snippet in a valid, standardized format. The snippet is then parsed with the AST native Python package as an inspection for parsability. If all constraints are exceeded, the code is added in its initial form(not encoded and not parsed) to the list of methods, containing all the parsable fragments of the inspected file. All the invalid repositories are tracked down, guaranteeing the knowledge of the code provenience. All the code fragments obtained during the mining process are collected and grouped as couples *(authorName,code-fragment)* obtaining a good format for a labelled format dataset.

The final part of this stage points out how to manage the persistence of the dataset as a distinctive factor of univocity. Obtaining a dataset with a high level of variance is one of the main points, leading to a representative and not redundant set.

The design of the former mining phases, including this one, can suffer from clone functions from diff. to diff. The problem has been tackled by exploiting the fragment ASTs as names for the addressed files. Given the need for name univocity in the filesystem, syntactically equal fragments are automatically discarded. To compress the final name, the AST dump has been hashed through the SHA1 algorithm, guaranteeing equal outputs for equal strings.

Author pruning and data undersampling

The final stage is settled to highlight the most relevant authors in the dataset itself. Authors with a limited amount of code fragments can lead to classes with less differentiated data.

The initial point of the pruning is then developed as a dimension-checking snippet, looking at the total amount of elements contained in each class(directory), if the classes don't satisfy the dimensional constraint, the addressed directory is pruned.

Once it has been obtained the final dataset dimension in terms of the class number, the final part of this stage consists of the dataset undersampling to obtain a balanced dataset. In order to speed up the process, the undersampler script works with the multiprocessing Python framework, looking for the file number constraint over all the class directories.

Model design

As the model design affects the quality of the snippets' embedding, this step poses another cornerstone in the problem. Code embeddings through time, as the related work section outstands, changed a lot by means of model structure and the kind of features that were exploited. Recent(and not so much) works [2,13,15] highlight the importance of syntactical features, narrowing our perspective over AST-based models, and opening up possibilities for comparisons. Nghi et. al.[16] work has already compared the most modern approaches for code encoding in a representation learning environment, focusing on snippets' semantic similarity instead of stylistic and syntactic similarities, leaving an open window over the code stylometry metric approach evaluation.

An adapted version of code2seq

Code2seq is a model designed mainly for code summarization tasks, posing a decoder as the final part of the model. This task-driven design differentiates itself from code2vec not only in the final decoder but even in the initial embedding phase, tackling the problems highlighted by the former version. In this work, the code2seq model has been modelled to generate a final vector, but keeping the improvement points that the code2seq work showed.

Token embedding summation and Byte pair Encoding technique

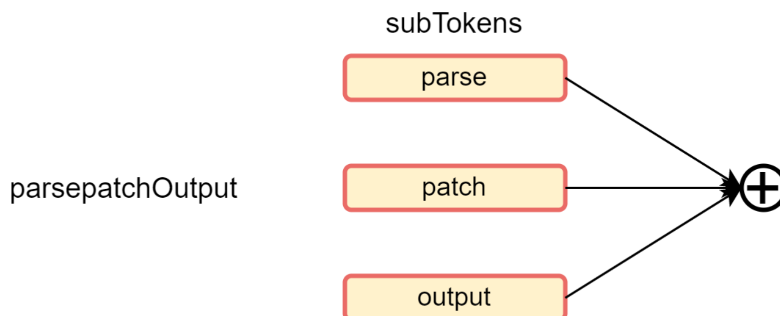


Figure 3.2: sub-tokenization and embedding summation.

The Terminal tokens embedding in code2seq is one of the major design improvements. The former code2vec model is used to embed terminal tokens exploiting only the embedding layer; here, an abstraction layer has been added. As Mikolov et. al. outstand [6] it's possible to obtain meaningful embedding representation over algebraic operations.

Here, the concept is incorporated to bypass the sparsity problem highlighted in the code2vec [8] paper. The idea is to split tokens into subtokens, inspired by programmers' habits who tend to name functions and variables by merging two or more names with camel notation paradigms or underscores between words. The resulting embedding values have to be summated, obtaining a unique, semantically valuable token representation.

$$tokenRepr_{\ i} = \sum_{s=1}^{n} Et_s$$

Figure 3.3: Subtoken summation where each subtoken is taken from the terminal embedding layer Et

In the original code2seq approach, the sub-tokenization is handled only with regex, leaving room for sparse token embeddings led by out of habits namings like *parsepatchoutput* in which the resulting sub-tokens would be embedded in a monolithic way.

In order to bypass this problem the intuition was taken from the modern language and NLP models design [29,30], leveraging the novel use of the Byte Pair Encoding technique introduced by Sennrich et. al. [31]. With the B.P.E, compression over the vocabulary dimension has been applied, exploiting an algorithm which works between word and character levels.

The idea behind the algorithm was taken from Gage [32] and adapted for word tokenization usage by merging characters and word fragments that have more frequency in the text. The final symbol vocabulary size is equal to the initial vocabulary size, plus the number of merge operations, the latter is the only hyper-parameter of the algorithm. This enables us to choose between different vocabulary sizes, testing the trade-off between many sub-tokens and monolithic words.

The terminal-token Corpus exploited is written by moving through the whole

dataset and creating a text that contains all the terminal tokens present in the Dataset. As the idea was to have meaningful word representations, the tokens were at first regex split with the code2seq regex technique before placing them in the corpus and normalized by lower casing the characters. This enables the algorithm to recognize better the syntactically valuable words, enforcing a more meaningful character merge.

AST embedding - from monolithic structures to LSTM reprojection

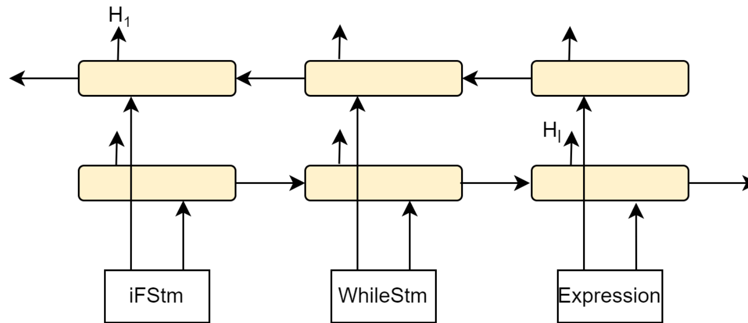


Figure 3.4: A.S.T. embedding with Bidirectional LSTM

The code2vec structure shows data sparseness impairments even over the AST projection, exploiting monolithic representations of the path between two different terminal nodes, embedded as the string composed by the elements of the non-terminal nodes and their path direction.

Code2seq aims to tackle this problem by using embeddings for each non-terminal node and then exploiting a bidirectional LSTM to project the AST on the latent space.

The two final hidden states are then both used in the following triplet projection. In this work, the two hidden states are pre-processed before the path context by projecting them in the same initial dimensionality, the idea is to obtain an antecedent matrix W_lstm specifically designed to extract semantic information from the two different LSTM outputs.

$$h_1, ..., h_l = LSTM(Ent_1, ..., Ent_l)$$

$$ASTrepr\ _i = [h_l^{\rightarrow}, h_1^{\leftarrow}] \cdot Wlstm$$

Figure 3.5: Where the Ent values are the resulting projected non-terminal tokens from the non-terminal embedding matrix; h_l are the hidden output states of the LSTM; $Wlstm$ is the projection matrix that mixes up the two final hidden states

From combined context vectors to the attention module, adding an M.L.P. layer

The triplet resulting from the first embedding, as expressed before, has to be elaborated, gaining semantic information derived from the AST and both the terminal values in between them. The upcoming data from the previous step can be synthesized as *'overall dimensionality * 3'*. As the code2seq model moves at this point, we apply a projection layer W_p with an activation function F_p. In the code2vec and code2seq works, the activation function used was the tanh, which normalizes the output in a codomain between $]-1, 1[$, and it can be formulated as:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Figure 3.6: Tanh activation function

Deep metric learning leads to slower convergence time than typical classification tasks. As we experienced during training time, the tanh activation function led to too much slow convergence times; replacing it with the ReLU activation function led to better gradient propagation speeding up the overall training process and gaining sparseness advantage [33].

$$ReLU(x) = \max(0, x)$$

Figure 3.7: ReLU activation function

The combined context vector is then the ReLU activation function application over the projection in the initial dimensionality of the previous states, given by the Terminal tokens and AST embeddings. For each context path P_i, composed by the concatenated triplet $[tokenRepr_{iS}, ASTrepr_i, tokenRepr_{iE}]$ where the S and E subscripts stand for Starting and Ending token of the addressed path, the resulting output from this stage is given by:

$$contextVector\, CV_i = ReLU(P_i \cdot W_c)$$

Figure 3.8: Context vectors (CV) obtained exploiting ReLU function where W_c is the Context projection matrix and P_I is the context Path as described above

As we obtain N combined context vectors, where N is entirely arbitrary, code2seq and code2vec models entangle the results with an attention module. As the needs of a single context vector income, the attention mechanism is led by the code2vec guidance, obtaining a final result that does a weighted average between the softmax-like function results with the attention matrix and the combined context vectors resulting from the previous steps, summing then the results weighted by the attention weight obtained before.

$$attentionWeght\, \alpha_i = \frac{\exp{(CV_i^T \cdot \alpha)}}{\sum_{j=1}^{n} \exp{(CV_j^T \cdot \alpha)}}$$

Figure 3.9: Computation of the attention weight, where alpha is the attention matrix and CV are the context vectors

$$codeVector\, v = \sum_{i=1}^{n} (\alpha_i \cdot CV_i)$$

Figure 3.10: Computation of the final snippet representation

The resulting codeVector is suitable as a final snippet representation. As Appalaraju et. al. [34] suggest, adding a source of non-linearity can lead to overall improvements in deep metric learning techniques. A Multi-layer

perceptron is then applied to the model with ReLU functions as an activation function, gaining the non-linearity mentioned before. The network with the non-linear projection head is structured as shown below:

$$fc \rightarrow ReLU \rightarrow fc \rightarrow ReLU \rightarrow fc \rightarrow ReLU \rightarrow fc$$

Figure 3.11: structure of the MLP head where fc is a fully connected layer

This structure aims to gain non-linearity, stated this, it's important to not compress the data in its dimensionality. The dimension chosen in the first layer is the same as the overall dimension used in the model; the dimensions of the hidden layer are then doubling the inner dimension, closing the model with a final layer projected on the initial dimension.

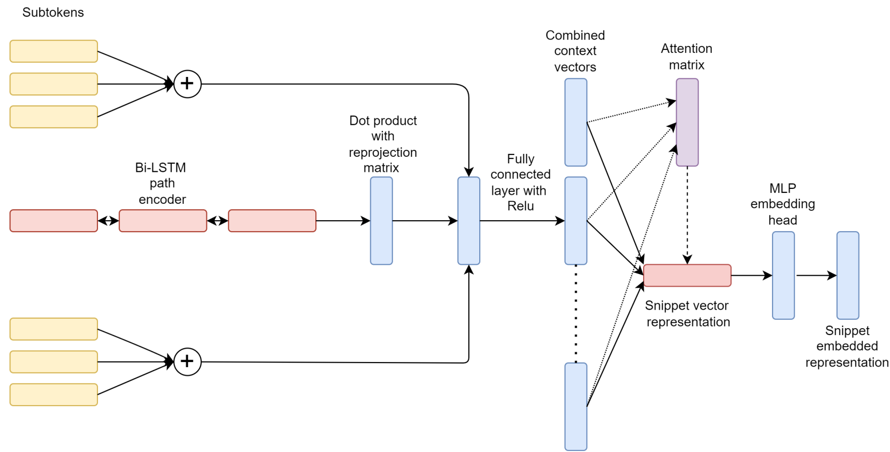


Figure 3.12: Code2vec model adapted with code2seq improvements.

Training loss

The loss choice leads to different approaches over various factors, as we move on to deep metric learning techniques, different losses must use different batching processes.

The loss exploited in this work is derived from the infoNCE loss proposed by Oord et al. [20], modified by Zang et. al. [35] exploiting the cosine similarity notion:

$$\langle v_i, u_i \rangle = \frac{v_i^T u_i}{||v_i||\,||u_i||}$$

Figure 3.13: Cosine similarity between two different code embeddings(u,v), exploiting the L2 norm

Used in the following loss function:

$$\ell_i^{(v \rightarrow u)} = -\log \frac{\exp(\langle v_i, u_i \rangle / \tau)}{\sum_{k=1}^{N} \exp(\langle v_i, u_k \rangle / \tau)}$$

$$\ell_i^{(u \rightarrow v)} = -\log \frac{\exp(\langle u_i, v_i \rangle / \tau)}{\sum_{k=1}^{N} \exp(\langle u_i, v_k \rangle / \tau)}$$

$$Loss = \frac{1}{N} \sum_{i=1}^{N} (\ell_i^{(v \rightarrow u)} + \ell_i^{(u \rightarrow v)})/2$$

Figure 3.14: InfoNCE loss function modified by Zang et. al.

The final loss is the average between the two losses obtained before as Radford et. al. [36] settled. The τ represents the temperature parameter which controls the range of logits in the softmax function.

As the negative and positive samples are driven by the batch, customizing the sampler to obtain only different classes is mandatory to avoid adversarial labels. This poses a constraint in terms of batch dimensionality during training, imposing the max batch dimension as the number of total classes present in the dataset.

	Snippet 1/2	Snippet 2/2	Snippet ...	Snippet N/2
Snippet 1/1	1/1 1/2	1/1 2/2	...	1/1 N/2
Snippet 2/1	2/1 1/2	2/1 2/2	...	2/1 N/2
Snippet ...	...	...	⋱	...
Snippet N/1	N/1 1/2	N/1 2/2	...	N/1 N/2

Figure 3.15: Computation of cosine similarity over a batch of N elements; the ground truth is highlighted in green

Classification baseline

In order to obtain a metric of comparison aiding to evaluate the deep metric learning model performances, it has been developed a model which has a classification head on top of the final M.L.P. module. The classification baseline is implemented with the CrossEntropy as a loss function over the softmax function measuring the dissimilarity between the predicted probability distribution and the true probability distribution, with the goal of minimizing this dissimilarity:

$$\ell(x, y) = L = \{l_1 ... l_N\}^T, \;\; l_n = -\log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} exp(x_{n,c})}$$

$$\ell(x, y) = \frac{\sum_{n=1}^{N} l_n}{N}$$

Figure 3.16: Cross entropy loss function having C as the class number and N as the batch dimension.

Data pre-processing

The data pre-processing step tackles data-transformation-related problems; A main goal is to render the input in a model-readable form. Since the input encoder works with code tokens and AST paths, the source code previously mined needs to be parsed, obtaining a token representation of the code's AST.

Both of the most used snippet encoders work with AST tokens, but from code2vec[8] to code2seq[13] the usage paradigm has changed, adopting techniques that take into account differently the position of terminal nodes within the AST. Code2vec based models work with AST paths between two random terminal nodes connected to each other, represented and embedded in a monolithic string format, instead, code2seq works with the same kind of path, but embeds each node separately, exploiting an LSTM technique for a further reprojection.

Taking into account the main differences between these approaches, data-balancing the Corpus is another fundamental step for further training; as we move into metric learning, the batch creation process needs to work over N differentiated samples, belonging to different authors. This process has a model-design-related problem too, indeed, batches should be instantiated by taking into account the contrastive loss exploited over the training phase. Recent works[14] showed how the loss over metric learning could impact the overall convergence of the model, old fashioned triplet losses tend to get one positive sample closer to the one taken into account, and push one negative sample away. These kinds of approaches don't handle other negative samples, slowing the time of convergence. These are the main factors that we should look into for both data preprocessing(batch-making process) and model design(loss choice).

As we move through the data transformation in a model readable form, we can distinguish two main steps:

- Code tokenization
- Code vectorization

During the code tokenization step, the goal is, as we can infer by the name, to split the code into pieces, obtaining a set of three different outputs lists returned as a triplet; the first is given by the subtokenization of the first terminal node over the path, the second is the list of nodes in the A.S.T. path

and the third is the list of subtokens for the ending terminal node in the path.

Algorithm 1 Function definition → ProcSnip(snippet,numContext):

```
# initial snippet parsing to retrieve the A.S.T.
Tree = ast.parser(snippet)

#T as list of terminal nodes
T = ast.retrieveTerminal(Tree)

# PathCouples as list of couples containing indexes for T list
PathCouples = getRandomCouples(len(T),numContext)

# mineT and mineAST respectively as terminal node extractor
# and AST node extractor
for all PathCouples do
  Result.append((mineT(T[path[0]]),mineAST(path[0],path[1],T),
  mineT(T[path[1]])))
end for
return Result
```

This process starts by parsing the code as the first step, obtaining a tree containing terminal and non-terminal nodes already tokenized. Given the complete tree, the next step is meant to retrieve all the terminal nodes, listing them for further random picking. As mentioned by Alon et. al. [8], exploiting random terminal nodes for path picking leads to a regularization effect during training. The parser, apart from the entire A.S.T. representation, lists all the terminal tokens; the resulting list is used both to get the overall number of terminal nodes, generating valuable couples for the path picking and to move further in the algorithm with the terminal and non-terminal token mining.

Algorithm 2 Function definition → mineT(terminalNode):

```
# initial node filtering, pruning strings and numerical values
if terminalNode is string || terminalNode is numericalValue then
    terminalNode = filter(terminalNode)
else
    terminalNode = regexSubTokenization(terminalNode)
end if
terminalNode = lowerCasing(terminalNode)
return terminalNode
```

The terminal nodes are extracted by a node abstractor which at first filters comments, strings and numbers rendering them as:

- comment
- string
- integer
- float

Simplifying the vocabulary by pruning the natural language and unifying all the numbers in the script as integers or float as the Python AST abstracts.

Then an initial subtokenization process is applied exploiting regex functions that split the word with camel notation and underscore as explained before. The resulting list of tokens is then lowercased to normalize words.

Algorithm 3 Function definition → minePath(T1,T2,TerminalList):

```
Path1 = [ ]
Path2 = [ ]
node1 = TerminalList[T1].getParent()
node2 = TerminalList[T2].getParent()
if node1 then
  Path1.append(node1)
end if
if node2 then
  Path2.append(node2)
end if
# seeking for the lowest common ancestor
while !((node1 in path2) || (node2 in path1)) && (node1||node2) do
  if node1 then
    node1 = node1.getParent()
  end if
  if node2 then
    node2 = node2.getParent()
  end if
  if node1 then
    Path1.append(node1)
  end if
  if node2 then
    Path2.append(node2)
  end if
end while
return merge(Path1,Path2)
```

The AST path between the two addressed terminal nodes is then retrieved by looking for the lowest common ancestor between them in the tree, moving up from the leaves, and seeking the nodes' parent. The two resulting lists of paths are then merged keeping the order from the first terminal node to the ending one.

The number of paths to retrieve from each snippet is given a priori such as Alon. et. al.[8] proposed.

After the paths have been retrieved, the tokens have to be rendered as vectors, gaining a model-readable form. The vectorization of the input has been handled by the usage of a vocabulary for the Not terminal tokens, associating each token present in the corpus with a unique number. The terminal tokens vectorization has been handled instead with the Byte pair encoding technique explained above, creating a further fragmentation of the subtokens obtained with the initial regex usage, and then, as the non terminal vocabulary, assigning a unique value to the subtokens.

Evaluation phase

The first comparison term highlights the embedding quality differences between a standard classifier and our metric learning-based one. Obtaining good cosine similarity top-1 ranked accuracy results, is by far the first key point to look into, showing the embedding-by-style capabilities of the model.

Another strategy of evaluation is given by Caliskan et. al. [2] introducing the concept of *relaxed classification*. In these terms, the utility is to check if the correct class is present over the top 5 results, giving a chance to shrink the search space. In a scenario with hundred of different programmers, this usage would have therefore a beneficial impact.

As the evaluation is embedding quality driven, the overall accuracy will be directed to evaluate the retrieval properties of the model, checking with a query vector if in the top-k element ranked per similarity is present at least one element which belongs to the correct class. This evaluation term is introduced by Jégou et. al.[37] as Recall@k and it has been used to compare it with the acc@1 and acc@5 relaxed classification introduced by Caliskan by testing the Recall@1 and Recall@5 accuracies.

Addressing then our approach to a zero-shot capable classification model sets the need for a different evaluation phase than the classic ones. The training-validation-test approach needs to be redesigned for an in and out-distribution analysis. It's important, in order to evaluate the zero-shot capabilities of the model, to keep a set of authors as an out-distribution test set, allowing us to evaluate no anymore 'unseen data', but 'unseen classes'. We can in this way assess the model by comparing results over two different test sets, one with authors' snippets belonging to the training set and one with authors' snippets

out of the training set, obtaining a full picture of the model performances.

Furthermore, the chance to experience an 'outlier' misclassification by adopting this technique, poses an open problem; thresholding a 'cosine similarity' level to obtain a 'not in the shown sample' level, could lead to other multi authors' related problems too, misclassifying a snippet written by two different authors to a 'not in the list' class, placing the work in a trade-off for the zero-shot use case.

Cosine similarity as embedding evaluation

As the loss function indicates, the tightening of the cosine similarity between similar classes is the main goal of the model, underlying how snippets from the same author style should be placed on the latent space.

The evaluation phase working with the top 1 and top 5 relaxed retrieval embeds the snippets in the latent space exploiting the trained encoder and, as a further computation, all the cosine similarities are calculated, seeking then for the highest values for each snippet. What we do obtain is a representation of the embedding goodness resulting from the encoder; this solution can lead to a search space by which, having samples from the classes, an authorship distance metric could be settled, gaining information about the possible authors in the search space, with and without the need of fine-tuning the model.

Given the snippets samples S: $S_1, ..., S_n$ with the encoder E, the cosine similarity is computed at first normalizing the embeddings with an L2 norm:

$$Cosine\ similarities\ matrix : CS = \frac{E(S)^T \cdot E(S)}{||E(S)|| \cdot ||E(S)||}$$

Figure 3.17: Notion of cosine similarities in the latent space

After the cosine similarities computation, the final step consists in checking if correct classes are present over the top 1 and top 5 most similar embeddings, assessing the goodness of the model. When it comes to evaluating the baseline, the same procedure has been applied, but this time the classification head

has to be removed, obtaining the same latent space dimensionality as the deep metric learning trained model.

Chapter 4

Results

The Results chapter's aim revolves around posing a disambiguation term over the methodologies, tackling the highlighted problem in a concrete way by showing the experiments' outcomes. As deepening into the results, the first step must show how the methodologies have been handled by highlighting the work hyperparameters.

As the work moves through two main phases, the chapter is split into two main sections:

- Dataset
- Stylometry task

These two sections will be split then into two different subsections, showing at first the hyperparameters setup and later, the results obtained with the experiments. The discussion of outcomes will be discussed in the latest chapter, showing the strengths and weaknesses of the methodologies applied.

Dataset

The data mining process posed the fundamentals for the tackled code stylometry task. Obtaining a dataset with enough data from different projects per author was, as highlighted before, one of the main cornerstones to dive into.

Hyperparameters

The first step leverages over where to span for the initial lookup, and it gains importance as the first retrieved repositories and authors are the ones that should lead the data mining phase.

First stage

Using libraries.io APIs as the first lookup space simplifies the process. The first hyperparameter to set for this stage is the number of overall pages to look up. In the first stage, repositories that don't belong to the GitHub platform will be pruned, shrinking the resulting URL list. The number of pages retrieved from the libraries.io API is **10** with **100** results per page.

Second stage

The second stage, meant to retrieve a list of valuable authors for the further process, seeks the top contributors over the previous project list, sorting them by the percentage of contribution over the project itself. In this phase, the hyperparameter to fix is the maximum number of authors per project to retrieve, settled as **10** authors per project.

Third stage

The third stage is the one that tackles the expansion of the URLs list by looking for projects on each mined author's GitHub personal page. Projects need to be sorted by a quality metric, assessed as the number of stars. Given the list of repositories retrieved with the GitHub APIs, the maximum number of repositories mined per author is settled to **4**.

Fourth stage

The fourth stage handles the repositories' cloning, it doesn't have hyperparameters that influence the results of the experiments. The usage of the *bare* command for the repositories cloning doesn't affect the overall results, but it only lights the space in the local memory.

Fifth stage

The fifth stage is the one, as mentioned above, responsible for the snippets manipulation, filtering them for the further saving and labelling process. As this stage moves on with the aid of the Pydriller APIs, part of the setup is addressed to filter the results in the exploited framework. Pydriller moves through commits in time, giving us a first customization chance that doesn't really affect the final result, which is if the commits should be ordered from the oldest to the newest or vice-versa. For our purpose, the miner has been settled to work in **reverse** mode, which means moving from the oldest commit to the newest one.

The Mining process' slowness is an important step to tackle, pydriller APIs offer the chance to check directly if, in the commit, modifications over the code for a specific language are present, lightening the search process by pruning all the files that don't belong to the addressed language. In this case, the file extension for the commit lookup is settled to **.py**.

Another important hyperparameter that can be settled during this stage is the minimum amount of lines to be present for addressing a fragment as valid. Settling this hyperparameter can lead to avoiding functions that work as placeholders or, getter and setter functions which are not majorly beneficial in identifying the author's fingerprint. The minimum length for a fragment is settled to an amount of **3** lines.

Sixth stage

The sixth stage aims to prune the authors that don't have enough data, this process is useful even for addressing authors who tend to have data from multiple projects, increasing the overall class data variance. The minimum number of snippets per author is settled to **1100**.

To differentiate authors for the zero-shot dataset, a different range of snippets has been used, assuring to have zero overlap results over the authors. In this case, all the authors with a number of snippets between **500** and **700** have been kept.

Dataset dimensions

In this subsection, final results have been highlighted, showing how each step leads to obtaining a certain number of authors and repositories to look into.

Following the steps described during the methodology, the initial schema has been reproposed:

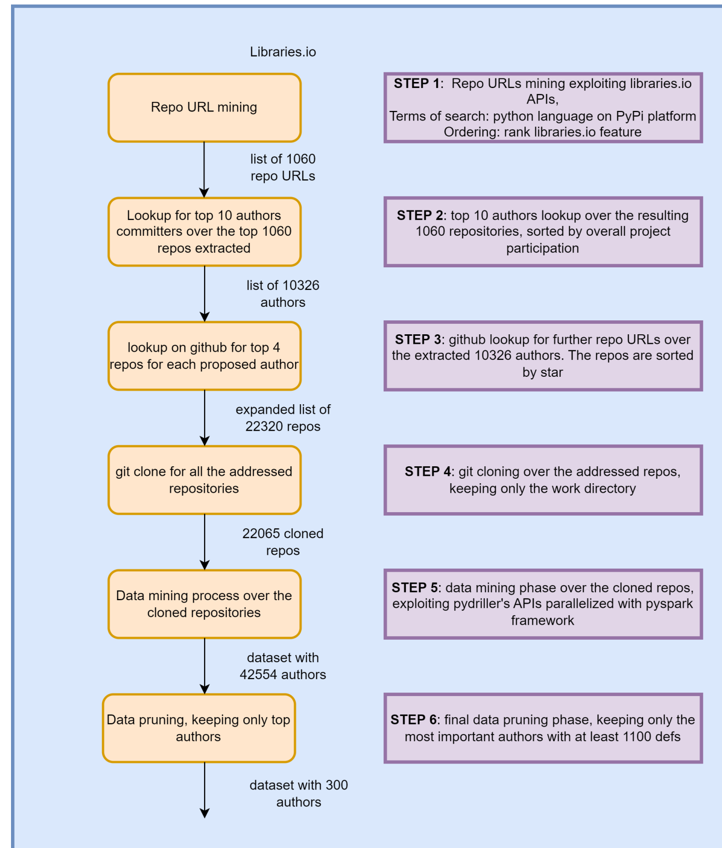


Figure 4.1

As mentioned in figure 4.1, the **step 1** produced a list of **1060** repositories' URLs ranked with the libraries.io standard rank, mined from the *PyPi* platform and filtered with the *Python* language.

The **step 2** produced a list of **10326** authors, obtained with the lookup over the initial repositories' URLs, having the authors sorted by overall participation, pruning the ones after the *10*th element.

The initial repositories list, expanded with the **step 3**, reaches an overall dimension of **22320** units, used for further repositories cloning.

Step 4 tracks down all the repositories that have been cloned, listing the invalids in a specific list. The cloning stage, lighted with the *–bare* command, worked over **22065** repositories writing down on a local file each valid one cloned, with its cloning timestamp.

Moving through **step 5**, tackling indeed the data mining phase, determines the most important stage, extracting data from the cloned repositories, obtaining snippets from **42554** different authors that will be filtered after. As in the previous step, all the repositories invalid for the data mining procedure will be tracked into the invalid repositories list.

The resulting RAW dataset shows a huge amount of noisy authors in terms of not having a sufficient amount of data; **step 6** works by pruning all the authors with less than *1100* snippets, gaining the chance to obtain authors with more variance in terms of the number of projects by which the data has been extracted. The resulting authors are **300** for the training validation and testing set.

As mentioned in the methodologies, another dataset has been extracted from the RAW one, in order to test the zero-shot capabilities of the model. Tackling the **step 6** with boundaries between *500* and *700* fragments per author, the resulting dataset contains **70** different classes that don't overlap with the previous ones.

Stage	Results
Stage 1	1060 repositories
Stage 2	10326 authors
Stage 3	22320 repositories
Stage 4	22065 cloned repositories
Stage 5	42554 authors dataset
Stage 6	300 authors dataset

Figure 4.2: Summary table for the data mining process' results

Balancing and splitting the dataset

Both the regular dataset and the zero-shot one have been balanced following the *random undersampling* technique, lowering the dataset dimensionality to **1100** snippets of code per author and splitting it by keeping respectively:

Type of set	Authors	Dataset portion(%)	code fragments per author
Training	300	80	880
Validation	300	10	110
Test	300	10	110

Figure 4.3: Dataset dimensionality after the splitting phase

The dataset for the zero-shot testing has been under-sampled as the previous test set to **110** snippets per author, keeping the same testing dimensionality. The number of authors has been randomly undersampled to reach the amount of **70** different authors, obtaining the same number of classes as the problem stated by [7,9]

Type of set	Authors	code fragments per author
Zero-shot test	70	110

Figure 4.4: Zero shot test set dimensionalities

Stylometer

In this section, the results of the code stylometry task have been tackled, the first part, following the former section's pattern, highlights the hyperparameters settled during the training phases, showing how different types of training require different setups. The second part of this section's aim is then to show the overall results, gaining evidence for the discussion in the next chapter.

Hyperparameters

Batch contruction

Given the loss involved during the training phase, the batch dimensionality directly influences the results by setting how many negative labels to drive away for each iteration. As said in the methodology chapter, the ideal scenario is to drive away every negative class per iteration while getting the positive one closer, fastening the convergence of the model and gaining full advantages over the old triplet losses[14]; due to memory constraints, the batch dimensionality exploited during training is of **64** elements per batch, which involves 128 different snippets encoded at a time. Experiments with a higher number of elements per batch have been tackled, but bigger batches led to a lower overall dimensionality, restraining the learning capabilities of the model.

For the classification baseline, the same batch dimension of **64** elements has been used

Model dimensionalities

The model learning capacity is correlated to the overall dimensionality of the model, starting from the embeddings to the final representation dimension, the designing process of the dimensionality is moved by a tradeoff between high learning capabilities that could lead to overfitting, where bigger models need more memory, and lighter models that could lead to shallow generalization capacities, directing to poor results.

Following what outstands from code2seq[13] in the high learning capabilities setup, the embeddings of both vocabularies and LSTM hidden sizes are kept all with the same dimension, fixing it at **256** weights in Single-precision floating-point format as the whole model's weights formats. The final MLP

head is then maintaining the same dimensionality, doubling it in the two hidden layers to **512** and resizing it over the last one to **256**.

Vocabulary compression with the B.P.E.

The model introduced in this work shows a novel usage of the byte pair encoding technique [31] over the code2seq model, modified to produce a vector, to embed the terminal tokens then summated.

In order to check the differences between different terms of compression, the B.P.E. vocabulary has been designed with **32000**, **64000** different tokens and without compression, keeping the non-terminal tokens vocabulary to its initial dimensionality of **147967** embeddings. The non-terminal tokens vocabulary isn't compressed with the B.P.E. technique and the overall amount of non-terminal tokens is of **174**.

Path bounding and overall number of context paths

As Bogomolov et. al. [9] stated, paths of a higher length are less frequent and cause the model to overfit. As the models exploited in the Bogomolov work are simpler and more inclined to overfit, an ablation study has been tackled, working with boundaries:

Model	max path-width distance	A.S.T. max path length
Model bounded	4	7
Model not bounded	-	-

Figure 4.5: Levels of settled boundaries

Where the term max path-width distance, refers to the maximum distance between two terminal nodes, and A.S.T. max path length refers to the maximum distance in terms of path lengths between the first two non-terminal nodes of the terminal ones addressed.

Later, over the model results, the constrained models will be referred to as *bounded*.

As code2vec and code2seq[8,13] highlight, sampling randomly arbitrary paths from the snippets covers the representativess of the snippet itself, working even

as a regularization factor during training. Gained by scientific evidence[13], using a number of paths of **200** is enough for the sampling process.

Temperature parameter, dropout and other hyperparameters

Working with the infoNCE loss poses the need for a temperature parameter τ as Wu et al.[21] suggests; the temperature parameter controls the concentration level of the distribution[38], for this process, the work of Zhang et al.[35] has been followed, using a temperature parameter of **0.1**.

During the training phase, as a regularization factor, dropout over the L.S.T.M. and over the context path embeddings has been applied, using values of **0.50** for the L.S.T.M. and **0.25** for the context embeddings. These values follow the results obtained in the code2seq[13] work.

Metric learning training processes have been tackled differently from the classification baseline, highlighting how different losses led to distinct solutions to the problem with different convergence times. Starting from the optimization algorithm, the model trained with the baseline cross-entropy loss showed a lack of convergence using the Adam algorithm, settling the need to try different optimization algorithms. From the initial tests, the two models followed immediately different training patterns, obtaining the best results with **Adam** as the optimization algorithm for the deep metric learning model, and the **stochastic gradient descent** when it comes to the classification baseline.

Both models have been trained with a learning rate scheduler which starts from a learning rate of **0.01** and decreases itself by multiplying the value every **50** epochs by a gamma value of **0.95**, helping to find the best local minima during training.

Hyperparameter	M.L. model	Classifier baseline
Terminal Vocabulary	32000,64000,147969	32000,64000,147969
Non Terminal Vocabulary	174	174
Optimization algorithm	Adam	S.G.D.
Path-contexts	200	200
Loss	InfoNCE	Cross Entropy
τ	0.1	–
Boundaries ablation	yes	yes
l.r.	0.01 with scheduler	0.01 with scheduler
Batch dimension	64	64

Each model has been trained for **700** epochs, taking as the optimal, the one with the lowest loss during validation. Early stopping was not a valuable solution due to unstable validation results and slow convergence times during the training process.

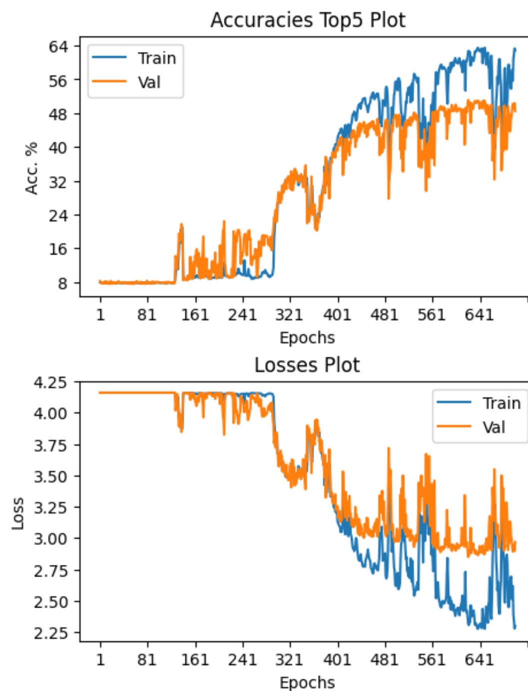


Figure 4.6: Training process of the Metric learning bounded model for the 64000 terminal-vocabulary. Example of high convergence instability.

Stylometer results

In this section, the results of the experiments are outlined, trying to highlight a path to answer the research question posed in the introduction:

Is it possible to achieve effective latent space representations that group snippets of code closer by stylistic terms exploring metric learning techniques?

In order to tackle the "effectiveness" point, models have been confronted with their standard classification counterparts, gaining a practical term of comparison. Given all the random sources that appear during the experiment, the training was performed by fixing the random seeds in order to ensure reproducibility, with a seed value of **7**. The tests follow the same procedure, but, are repeated three times as the snippets paths are sampled randomly. A set of three random seeds **{7,8,9}** has been used. Outcomes are then displayed as mean and standard deviation. As a metric of comparison, the Recall@k has been used on both metric learning models and classification models by which, to obtain the same embedding dimensionality, the classification head has been pruned.

$$Recall@k = \frac{PQO}{Q}$$

Figure 4.7: Recall@k where PQO stands for positive query occurrences, i.e. if in a query result of k elements, at least one positive sample appears. Q is instead the overall number of queries

All the experiments are carried out on the test set, which is a subset of the initial dataset, containing known authors; outlying the model capabilities to generalize over already seen classes but unseen snippets of code. The second test aim is to check the *zero-shot* capabilities of the model, looking for the ability to generalize over unseen snippets of unseen authors. During the querying phase(higher rates of cosine similarity between query embedding and target embeddings), all the encoded snippets have been L2 normalized in both metric learning and classification model. The design of this procedure has been applied to follow the training pattern for what concerns the metric learning model, and as the Horiguchi et al[17] outlines to obtain better representation over softmax-driven(classification) models.

In figure 4.8, the table shows the outcomes for the metric learning model over the testing set, highlighting the ablation study over the vocabulary compression and the bounded models in terms of A.S.T. path length and terminal nodes distance. The table shows the Recall@k accuracies, indicating the goodness of the models for k=1 and for k=5.

T. vocab dimension	Recall@1	Recall@5	Bounded
32000	0.253(±0.002)	0.422(±0.004)	no
32000	0.262(±0.001)	0.471(±0.001)	yes
64000	0.249(±0.001)	0.412(±0.002)	no
64000	0.311(±0.001)	0.526(±0.001)	yes
147967 no B.P.E.	0.246(±0.001)	0.417(±0.001)	no
147967 no B.P.E.	**0.324(**±0.001**)**	**0.544(**±0.001**)**	**yes**

Figure 4.8: Recall@1 and Recall@5 results over the deep metric learning models for the test set

As expected from the results obtained by Bogomolov et. al. [9], boundaries on context paths, taking into account the A.S.T. path length and the distance from terminal nodes, lead to better snippet representation when it comes to diving stylometry tasks. Lowering compression terms (rising the merging in the B.P.E .algorithm) increases the overall model's generalization capabilities.

The table in figure 4.9 highlights instead the results of the models pruned of their classification head and trained with a standard softmax, cross entropy-based classification technique. Even here, the outcomes confirm the Bogomolov et. al. results when it comes to bounded models, but in this case, more and less compressed models have closer Recall@k values than the deep metric learning ones.

T. vocab dimension	Recall@1	Recall@5	Bounded
32000	0.215(±0.002)	0.361(±0.001)	no
32000	**0.257(**±0.001**)**	**0.450(**±0.001**)**	**yes**
64000	0.218(±0.002)	0.364(±0.002)	no
64000	0.255(±0.001)	0.449(±0.001)	yes
147967 no B.P.E.	0.166(±0.001)	0.317(±0.001)	no
147967 no B.P.E.	0.185(±0.001)	0.359(±0.001)	yes

Figure 4.9: Recall@1 and Recall@5 results over the baseline classification models for the test set

The evidence from the test set experiments shows that deep metric learning models can obtain better results than classification baselines when it comes to generalizing over already-seen authors, outlying better retrieval performances over the Recall@1 and Recall@5 metrics. In this case, rising the terminal tokens vocabulary led to better generalization results.

In figure 4.10 the same experiments as in figure 4.08 have been tackled, this time up to unseen authors, tackling out-of-distribution data and so highlighting the zero-shot capabilities of the model.

T. vocab dimension	Recall@1	Recall@5	Bounded
32000	0.272($\pm$0.002)	0.476($\pm$0.007)	no
32000	0.297($\pm$0.001)	0.503($\pm$0.001)	yes
64000	0.234($\pm$0.003)	0.438($\pm$0.002)	no
64000	**0.320(**$\pm$0.001**)**	**0.522(**$\pm$0.001**)**	**yes**
147967 no B.P.E.	0.315($\pm$0.005)	0.502($\pm$0.004)	no
147967 no B.P.E.	0.311($\pm$0.001)	0.517($\pm$0.003)	yes

Figure 4.10: Recall@1 and Recall@5 results over the deep metric learning models for the zero-shot set

As can be seen, results take benefit from the vocabulary compression, obtaining the best Recall@1 and Recall@5 accuracies over the 64000 BPE bounded model. This could be due to a major presence of unseen tokens(mostly out of vocabulary in models without BPE) that in the case of the BPE model lead to more fragmented and semantically meaningful embeddings.

An important factor that needs to be taken into account is the difference in terms of classes number between the two datasets. For instance, higher accuracy over the zero-shot testing set is achieved by almost all the zero-shot experiments, but the dataset contains 70 classes instead of 300, gaining a comparison value between the old accuracies results in the state-of-the-art models, but easing the task if compared to the initial dataset which is trained for 300 different authors.

T. vocab dimension	Recall@1	Recall@5	Bounded
32000	0.335($\pm$0.003)	0.511($\pm$0.002)	no
32000	0.369($\pm$0.001)	0.566($\pm$0.001)	yes
64000	0.346($\pm$0.002)	0.532($\pm$0.004)	no
64000	**0.371(**$\pm$0.002**)**	**0.569(**$\pm$0.002**)**	**yes**
147967 no B.P.E.	0.231($\pm$0.002)	0.404($\pm$0.004)	no
147967 no B.P.E.	0.296($\pm$0.002)	0.467($\pm$0.003)	yes

Figure 4.11: Recall@1 and Recall@5 results over the baseline classification models for the zero-shot set

The table in figure 4.11 keeps slight differences between BPE models differentiated by the vocabulary dimension but still maintains an interesting gap between the unbounded models and the bounded ones. The absence of the BPE led to a high weakness in the baseline zero-shot embedding, highlighting even this time the strengths of the BPE technique when it comes to generalising over unseen, out-of-distribution authors.

Surprisingly, comparing the baseline and metric learning models show inverted results when it comes to generalizing over out-of-distribution classes, leading to better results over the softmax classification-based models.

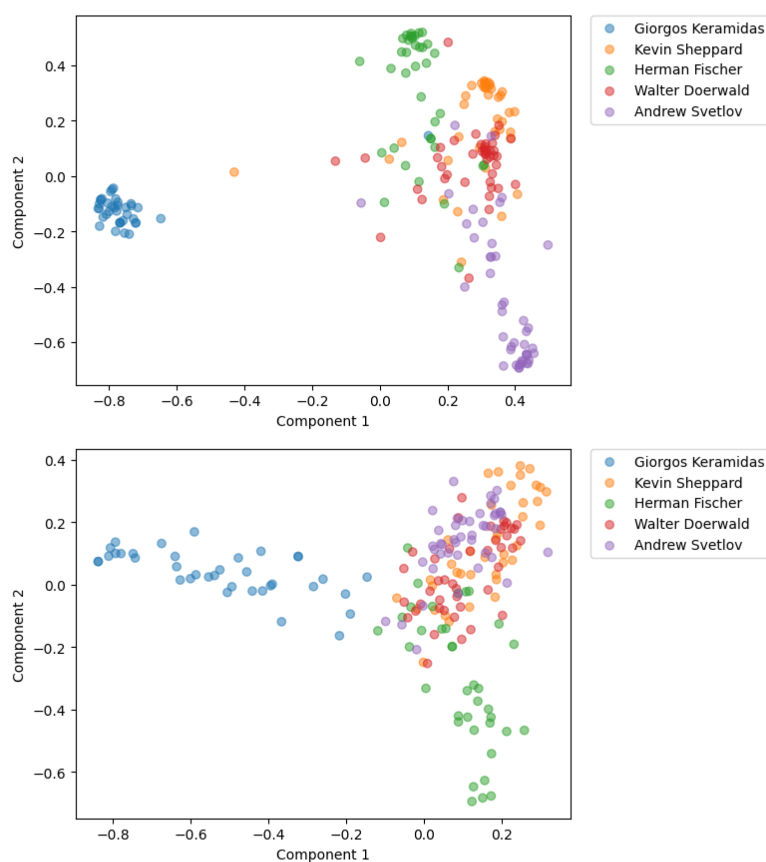


Figure 4.12: Snippets test-set Embeddings of five different authors using as an encoder the metric learning bounded model without the BPE technique (first image) and the baseline classification bounded model with 32000 Terminal tokens vocabulary(second image)

Figure 4.12 shows an embedding representation, using the Principal Component Analysis (PCA), projected to the first two principal components. The representation has been obtained by getting five random authors from the test set with an overall amount of 200 different snippets randomly picked from the five selected authors.

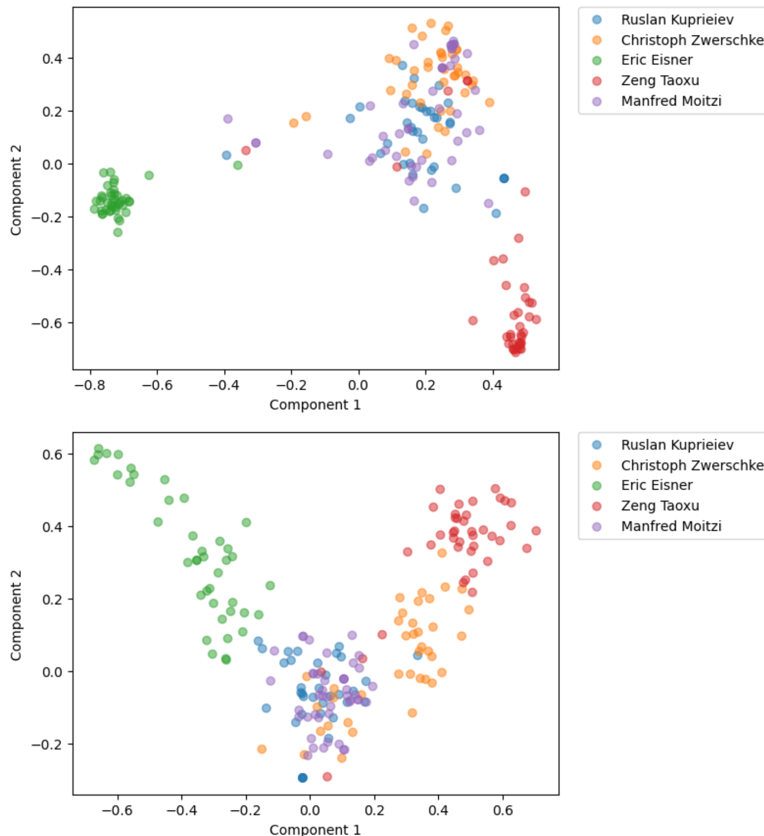


Figure 4.13: Snippets of zero-shot authors from five different authors using as an encoder the metric learning bounded model with 64000 Terminal tokens vocabulary(first image) and the baseline classification bounded model with 64000 Terminal tokens vocabulary(second image)

Figure 4.13 shows the embeddings of five different authors coming from the zero-shot dataset, outlying the still good capabilities of the model to generalize over unseen, out-of-distribution data.

Chapter 5

Discussion

This section's aim is to analyze the results outlined in the previous chapter, moving through the advantages and disadvantages of the novelties introduced over the new dataset and authorship attribution technique, keeping an eye on the State of the art results. The second, but not ranked for importance aim of this chapter, is to answer the research question:

Is it possible to achieve effective latent space representations that group snippets of code closer by stylistic terms exploring metric learning techniques?

An initial background for this answer has been tackled in the results chapter, here the answer is led to put in the discussion of the exploited technique.

Dataset

Obtaining a dataset which respects the constraint given by Caliskan et. al.[2] was one of the main properties to take into account; as the data-mining process outlines, this first constraint has been tackled spanning through several repositories from different authors. The data pruning phase then assures to keep only authors with a sufficient amount of snippets, which rise up the probability to have in these terms a higher overall variance. The results obtained during the zero-shot testing for the stylometry task are proof of the goodness of the dataset in this sense. The model indeed kept the retrieval ability even over unseen authors, highlighting that the first Caliskan's constraint has been tackled.

The final resulting dataset, compared to the GCJ dataset, used by most of the SOTA models[2,9,7] for the stylometry task, gains notability as it deals with open world real problems. This approach of mining the data from real-world projects has been tackled previously by Dauber et. al. [4] keeping the data over fragments of code as this work does, and crawling it from GitHub.

In this work the spotlight was turned to gaining more valuable seed authors; the libraries.io API drives the work over projects with a higher level of quality of code, differentiating this work from the one outlined by Dauber et. al.

This work differentiates itself not only from the seed authors' importance but even by the main mining technique, indeed, Dauber et al. extracted the data exploiting the git blame command; in this work, we benefit from the novelty use of the git diff. technique, supporting the data's univocity with an AST hashing persistence technique, assuring to not obtaining semantically and syntactically equal fragments of code which could result in an involuntary preliminary data augmentation. This technique helped obtain a higher amount of data compared to Dauber's work, given the chance to mine snippets from the whole project exploiting all the commits.

Dataset	Number of authors	Training snippets per author
This work	300	880
Dauber et. al.	104	100

Figure 5.1: Summary for dataset dimensionalities

Having a higher amount of data compared to the GCJ dataset and the one from the Dauber et al work gives us a chance to tackle the stylometry problem with models that are more data-hungry than others. Bogomolov et. al.[9] showed that Random forest techniques gained a big gap in terms of general accuracies compared to code2vec models when it comes to training over a little amount of data, dealing with the data sparseness problem highlighted by Alon et. al.[8].

Threats to Validity

The proposed data-mining technique is based on syntactic data reduction discussed above, which prunes all the data having the same AST, leading to a data deletion that doesn't take into account comments. Indeed, the same snippet with or without comments would be collapsed into one single snippet; this could lead to an earlier, not commented version of the same fragment having the mining phase starting from the first commits in terms of temporal order.

This technique is also based on the author's username labelling, which leads to a possible non-univocity term, obtaining the same label for different authors, heading to adversarial labels that could affect the final result when it comes to the stylometry task.

Stylometer

Aiming to answer the Research question posed at the beginning of this work, a comparison between the baseline classification model aids answering to the *'effectiveness'* point of the proposed training methodology. As this work settles itself as a novelty in terms of code stylometry, approaching these results to the ones proposed by previous metric learning works over other tasks[16,22] is a possibility that would not gain valuable evidence to be discussed. However, the model, having a set of snippets as ground truth from different authors, can be used as a classificator looking for the closest snippets in terms of cosine similarity. For this reason, the zero-shot dataset has been created with 70 different authors as the GCJ python dataset used in the latest SOTA stylometry papers and gaining the chance to compare the current setup to the SOTA classification works.

The experiments moved through two different datasets, testing the model's capability of generalizing in-distribution data and out-distribution data; initial evidence can be highlighted:

Different terms of compression work differently between different datasets and training techniques, but data boundaries show, as Bogomolov et. al.[9] describes, a clear strength in this task, obtaining almost everywhere better results over the used Recall@k metric and faster convergence times.

The metric learning technique driven by the infoNCE loss works better than the classification baseline when it comes to in-distribution code retrieval capabilities, showing higher capacities to handle bigger terminal token vocabularies than the classification baseline, which, without the byte pair encoding outlines a gap between models with the compressed vocabularies. The difference between compressed and uncompressed vocabularies in the classification baseline highlights a potentially key point for models trained with classification softmax techniques, posing a point for future work.

Out-of-distribution data leads to different results, outlying how compressed vocabularies over unseen authors work better than uncompressed ones, this could be due to better semantic properties for unknown tokens' embeddings that, with the byte pair encoding technique are handled differently, exploiting fragments that keep higher semantically relevance than the unknown ones. This evidence is more relevant over the classification baseline which shows a bigger gap between compressed and uncompressed models. The model trained with the infoNCE loss is less affected even this time by the vocabulary compression, keeping good results even with the uncompressed models.

The BPE works therefore better where in-vocabulary tokens are less frequent; the zero-shot testing set hence, lights up the strengths of this technique and poses itself as a milestone in the baseline models, showing how, as in language models works as a foundation for the State of the art results [29,30] even for code2vec and code2seq models could become a major point of interest, updating the former designs.

Metric learning-based models, when it comes to in-distribution authors' testing, highlight a big gap between baseline models, showing capabilities to generate more effective representations after L2 normalization. The models' convergence is though slower and unstable when compared to classification baselines. Classification baseline representations, after an L2 normalization, outline better representations over out-distribution authors and shows higher efficiency in terms of stability and time of convergence.

Comparing the model's retrieval techniques Recall@1 as a classification method to the SOTA accuracies reveals still a big difference, which could be due to the higher difficulty of the acquired dataset. Methodology's strengths can be spotted as the zero-shot chance, which allows the final user to not fine-tune the model. The usage of the metric learning training technique can lead to other paths which will be discussed in the *future work* chapter.

Future works

This work has presented a new author-labelled dataset with a model that produces snippets embeddings as code2vec, and heads to the novelties of code2seq, engaging the B.P.E. exploited over the newest SOTA work on language models.

Given these novelties, an outlined path comes to attention, highlighting the need for a comparison between this newly proposed model, the code2vec structure, random forest model proposed by Bogomolov et. al.[9] and the LSTM-based model proposed by Alsulami et. al.[7], showing how results differ with the mined dataset.

Our results show that using the metric learning technique over the code stylometry task leads to different and in some cases more effective embeddings. As Jain et. al.[22] shows, metric learning could improve the overall model accuracies if exploited as a pretraining method for downstream tasks. Trying the model as a pretraining for downstream tasks is one of the future work to take into account. Few-shot training is then an interesting possible gained capability that the metric learning pre-trained model could have achieved, showing the possibility to obtain effective results with little amounts of data.

Conclusions

In this work, we dived into the task of code stylometry by moving on to three principal components:

- The introduction of the new author-labelled dataset
- The modified code2vec model
- The novelty use of metric learning for the code stylometry task

The introduction of the new author-labelled dataset

Capturing the author's style has always been a fundamental task to detect plagiarism, deanonymizing authors or simply easing the task of migrating the source code and trying to address it quickly. Several works outlined the possibility to tackle this task by operating on a dataset that resembles the real-world scenario but keeping the constraints mentioned by Caliskan et. al. for a good code stylometry environment. The work of Dauber et. al. headed to a more realistic dataset, obtaining a model that can generalize over real use case scenarios. In this work, the resulting dataset is inspired by Dauber et. al.'s aim, exploring different mining techniques that give the chance to extract bigger amounts of data from the same project, keeping the snippets univocity.

The results obtained by the stylometer in the zero-shot setting highlight that the major Caliskan's constraint of differentiating the author's data through different projects have been tackled effectively, showing that the model trained with the new dataset is capable of generalizing over unseen authors, outlying a high variance in terms of author's project differentiation.

The modified code2vec model

The Code2vec model has shown high capabilities for snippets representation over the latent space, placing itself with the SOTA models for several tasks. As the code2seq paper highlighted, the code2vec model showed weaknesses in its design, tackled by the code2seq model as a natural evolution of the former design. In this work, we adapted the code2seq idea to represent a vector as output, and it has been tested for modification over the terminal tokens tokenization, exploiting the B.P.E. technique. Other minor adjustments have been applied to the model, using the ReLU activation function instead of the tanh to speed up the model convergence which with the metric learning trained models didn't occur.

In order to test the goodness of the model, an ablation study over this major adjustment has been done, highlighting how rising the vocabulary compression by modifying the number of overall merges in the byte pair encoding algorithm could lead to a loss of information or a gain in generalization when it comes to the stylometry task.

As mentioned in the discussion chapter, the Byte pair encoding technique showed its strengths mostly over the classification baseline model and over the out-distribution authors' test set. We obtained for instance evidence that BPE can lead to improvements in latent space representation opening the path for future works models' design which involves this method.

A comparison between the newly designed model and the previous techniques needs to be done, inspecting the strengths of the BPE and code2seq novelties over the other techniques with the new dataset.

The novelty use of metric learning for the code stylometry task

In the presented work, the use of metric learning led the experiments ablation study, being the subject of the research question posed in the introduction. Given the outcomes, the potentiality of this technique has been acknowledged, outlying how metric learning can lead to better snippets representations.

The efficacy of the methodology has been experimented showing how the infoNCE loss works over different model designs, gaining efficacy in terms of Recall@k metrics, but slowing the convergence time in comparison to the classification baseline for the in-distribution authors. The model proposed could be then a valuable solution when it comes to looking up code authorship by style, easing the investigation when it comes to search spaces with several different authors. As models showed good capabilities with out-of-distribution data, the strength of this technique is headed by the possibility of not fine-tuning the model for further authors, obtaining a flexible tool for the code authorship investigation.

The lack of accurateness of the proposed technique compared to the SOTA models has been acknowledged but keeping an eye on testing the other designs with the proposed dataset, indeed, diving into real-world problems could lead to a harder stylometry task to solve than the typical GCJ dataset.

Bibliography

[1] Programming style authorship analysis - Oman, Cook.

[2] De-anonymizing Programmers via Code Stylometry - Aylin Caliskan, Richard Harang, Andrew Liu.

[3] Google code jam.

[4] Git Blame Who?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments - Edwin Dauber, Aylin Caliskan.

[5] Authorship attribution of source code by using back propagation neural network based on particle swarm optimization - Yang, Xinyu Xu, Guoai Li, Qi Guo, Yanhui Zhang, Miao.

[6] Efficient Estimation of Word Representations in Vector Space - Mikolov, Chen, Corrado, Dean.

[7] Source Code Authorship Attribution Using Long Short-Term Memory Based Networks - Alsulami, Dauber, Harang.

[8] code2vec: Learning Distributed Representations of Code - Alon, Zilberstein, Levy, Yahav .

[9] Authorship Attribution of Source Code: A Language-Agnostic Approach and Applicability in Software Engineering - Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, Timofey Bryksin .

[10] user2code2vec: Embeddings for Profiling Students Based on Distributional Representations of Source Code - David Azcona, Piyush Arora, I-Han Hsiao, Alan Smeaton .

[11] Building Implicit Vector Representations of Individual Coding Style -

Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, Alberto Bacchelli.

[12] Corpus Linguistics - Tony McEnery.

[13] code2seq: Generating Sequences from Structured Representations of Code - Uri Alon, Shaked Brody, et. al.

[14] Improved Deep Metric Learning with Multi-class N-pair Loss Objective - Kihyuk Sohn.

[15] CodeBERT: A Pre-Trained Model for Programming and Natural Languages - Zhangyin Feng, Daya Guo et. al.

[16] Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations - Nghi D. Q. Bui, Yijun Yu, Lingxiao Jiang

[17] Significance of Softmax-Based Features in Comparison to Distance Metric Learning-Based Features - Shota Horiguchi, Daiki Ikami, Kiyoharu Aizawa

[18] Road Network Metric Learning for Estimated Time of Arrival - Yiwen Sun, Kun Fu, Zheng Wang, Changshui Zhang, Jieping Ye

[19] Improved Deep Metric Learning with Multi-class N-pair Loss Objective - Kihyuk Sohn

[20] Representation Learning with Contrastive Predictive Coding - Aaron van den Oord, Yazhe Li, Oriol Vinyals

[21] Unsupervised Feature Learning via Non-Parametric Instance-level Discrimination - Zhirong Wu, Yuanjun Xiong, Stella Yu, Dahua Lin

[22] Contrastive Code Representation Learning - Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, Ion Stoica

[23] Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations - Nghi D. Q. Bui, Yijun Yu, Lingxiao Jiang

[24] Libraries.io - packages index from package managers

[25] Curating GitHub for engineered software projects - Nuthan Munaiah, Steven Kroh, Craig Cabrey, Meiyappan Nagappan

[26] Github.com

[27] Git protocol - official documentation

[28] PyDriller: Python Framework for Mining Software Repositories - Spadini Davide, Aniche Maurício and Bacchelli Alberto

[29] Language Models are Unsupervised Multitask Learners - Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever

[30] RoBERTa: A Robustly Optimized BERT Pretraining Approach - Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyerand Veselin Stoyanov

[31] Neural Machine Translation of Rare Words with Subword Units - Rico Sennrich, Barry Haddow and Alexandra Birch

[32] A new algorithm for data compression - Philip Gage

[33] Deep Sparse Rectifier Neural Networks - Xavier Glorot, Antoine Bordes and Yoshua Bengio

[34] Towards Good Practices in Self-supervised Representation Learning - Srikar Appalaraju, Yi Zhu, Yusheng Xie and István Fehérvári

[35] Contrastive Learning of Medical Visual Representations from Paired Images and Text - Yuhao Zhang, Hang Jiang, Yasuhide Miura, Christopher D. Manning and Curtis P. Langlotz

[36] Learning Transferable Visual Models From Natural Language Supervision - Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger and Ilya Sutskever

[37] Product Quantization for Nearest Neighbor Search - Herve Jégou; Matthijs Douze and Cordelia Schmid

[38] Distilling the Knowledge in a Neural Network - Geoffrey Hinton, Oriol Vinyals and Jeff Dean