Technical Journal, University of Engineering and Technology (UET) Taxila, Pakistan *Vol. 28 No. 1-2023*
ISSN:1813-1786 (Print) 2313-7770 (Online)

A Stylometric Fingerprinting Method for Author Identification Using Machine Learning

M. M. Iqbal[1], A. Raza[2], M. M. Aslam[3], M. Farhan[4], S. Yaseen[5]

[1]*Department of Computer Science, University of Engineering and Technology, Taxila, Pakistan,*
[2,3,4,5]*Department of Computer Science COMSATS University Islamabad, Sahiwal Campus, Pakistan*

munwar89@gmail.com

Abstract- Identifying authors relies on their unique writing patterns, also known as stylometry. However, as each individual's stylometry may differ, it can be challenging to determine the true author, especially when multiple documents exist. This difficulty is compounded by similarities in writing styles, such as font and language, which can obscure the author's identity. To address this issue, machine learning techniques can be employed to identify human attitudes in written documents. By analyzing patterns in human behavior, it is possible to enhance privacy and security by detecting malicious users and malware programs. However, the use of behavior analysis may raise privacy concerns, particularly for individuals seeking to conceal their identity. Authorship attribution involves using stylometry techniques to identify the authors of multiple documents, and can aid in accurately identifying authors. In this research, we propose using stylometry to extract the number of programmers from a given database, and to analyze different datasets to determine whether a program's coding style remains consistent. This analysis can enhance the reliability and quality of programming, ultimately improving the overall efficiency of programming tasks.

Keywords- Stylometry, author identification, programmer deanonymization, machine learning, deep learning

I. INTRODUCTION

Language is a crucial aspect of communication, and any coherent sentence is considered a part of a language. Various techniques, such as natural language processing (NLP) and machine learning (ML), can be used to analyze the behavior of programmers. Each developer has their unique writing style and coding patterns. For example, a developer may prefer a for loop over a while loop, a switch statement over multiple if-else statements, or more elaborate code structures than a simple code. These variations can raise privacy concerns as many programmers wish to maintain their anonymity, even if they are the creators of popular software such as Bitcoin. This aspect of coding can be beneficial in anonymizing donor names, pilgrims, forensic reports, and malware issues, among others.

Computer code can be identified by matching its features, such as control structures and data types. Programming offers many opportunities for variation and innovation, and each programmer has a unique writing pattern. Using stylistic elements can help programmers reuse, produce, and debug code. During programming, programmers often combine elements of others' styles into their documents. Identifying the authorship of code requires an adequate body of code and the identification of features for comparison. However, it can be challenging to identify the author if they have attempted to conceal their authorship or if the code sample is unavailable. Nonetheless, essential features may still be present for analysis. Analyzing code attributes may lead to the identification of suspects for further investigation. Moreover, if sufficient background research is conducted to establish a statistical base and if large code samples are available, statistical methods can be applied to find the authorship. However, it is uncertain whether these features can be combined with stylistic features to provide clues for the authorship of code.

Author identification is a valuable technique for determining who is most likely to have written messages, articles, code, or news. This task is commonly viewed as a text categorization problem with multiple classes and a single label. It is a fascinating topic in natural language processing, with numerous applications, including identifying anonymous authors, detecting plagiarism, assisting in crime investigation and security, and locating ghostwriters.

Most previous efforts to classify authors have relied on n-grams, which are character constants of varying lengths. In this paper, we use multiple text-based models and machine learning methods with increasing

feature engineering at various stages to address the problem. To gain a better understanding of the models, our suggested approach evaluates multiple stylometric aspects and selects individual features with strong performance. We test the methodology on a portion of the Reuters news corpus, which consists of works by 50 different authors on the same topic. Our experiments show that using document fingerprinting features improves the classifier's accuracy, and principal component analysis (PCA) enhances the outcomes. Additionally, we compare our findings to previous studies in the authorship identification field.

II. LITERATURE REVIEW

The earliest attempt at authorship identification focused on [1] attribute counting-metric systems, which featured metrics for counting the number of code lines, unique operands, or declared variables; and [1, 2] structure metrics, which contrasted abstract representations of the program structure [3]. Machine learning methodologies are used in most of the publications in the bibliography today. For example, the authors created an authorship identification method that extracts statistical information like word n-grams and some hand-crafted aspects like code structure. According to the authors, some hand-crafted features reflect "explicit and implicit personal programming preference patterns of and between keywords, identifiers, operators, statements, methods, and classes." Without using hand-crafted characteristics, Bander et al. used Recurrent Neuronal Networks (RNNs) based on traditional and bidirectional Long-Short Term Memory networks (BiLSTM) from the Abstract Syntax Tree (AST) in [4]. Another RNN study is detailed in [5]. The authors used a Gated Recurrent Unit (GRU) tested on two datasets and achieved an accuracy of 69.1 percent and 89.2 percent, respectively. Another method is described in [6], in which the authors compare Latent Semantic Analysis (LSA) with re-use detection models to determine source code cross-language similarity. The writers' contribution in [6] is two-fold. On the one hand, they show two language-independent models that outperformed language-specific models on datasets from three common programming languages. On the other hand, they point out several flaws in source-code datasets for authors profiling, emphasizing how the environment in which programmers write code (which they regard as the work context) influences their style by compelling them to make certain decisions, such as naming conventions. They further claim that (1) existing datasets ignore equitable code collaboration and (2) the reality that author styles can change over time. The authors of [7] advocated working on authorship of source code segments to address the issue of fair code collaboration. They used a stacking ensemble strategy that combined deep neural networks and machine learning classifiers to provide promising results. [8] takes an exciting approach to authorship attribution by reversing the problem and a black-box attack for authorship identification of source code is presented by performing semantics-preserving code transformations to create variations of the source code that fool machine-learning solutions into inducing false attributions. The goal of this method is to generate source code for use in adversarial learning.
The author's work in the software sector has gotten a lot of attention at scientific seminars and conferences. The shared task detection of source code Re-use [9] was suggested as a PAN shared task in 2014, and it consisted of identifying source code re-use from an unbalanced dataset of C and Java code. This challenge was completed by five teams, with 17 runs. [10] describes another shared task in which participants were asked to predict the author's personality based on four significant qualities extracted from Java source code. 48 runs from 11 participants were sent at the end of the challenge.
So, there is an overlook to authorship identification and attribution to discover exciting ways to help us build our models.

Stylometric Methodologies

Stylometry is focused on recognizing features in written text that are related to an user's stylistic decisions. It is the earliest technique in code authorship attribution. Krsul and Spafford [11] proposed 49 different features in three major areas: (1) programming style such as comment naming style, variables and functions. (2) layout specific metrics such as indentation and comment style; and (3) program structure such as presence of debugging identifiers or assertions, the use of specific data structures, and error handling.

Abstract Syntax Tree Approaches

Abstract Syntax Trees (ASTs) are tree structures representing the abstract structure of a source code and are a most important source of information in the analysis of programs. Caliskan-Islam [12] proposed combining a stylometric feature set recovered from a source code's AST with lexical and layout features elicited directly from the source code. Caliskan-Islam [12] discovered that AST node bigrams were the significant feature for determining different authors.
CodeBERT [13] is a multi-layer bidirectional Transformer model that has been pre-trained and is built on the same architecture as RoBERTa. CodeBERT, unlike RoBERTa, is intended for usage in both natural language and programming language

applications, such as code documentation and natural language code search. Instead of leveraging information from the code AST, as models like code2vec [13] and code2seq [14] do, CodeBERT produce feature vectors for code segments using contextual information from surrounding words.

Qiqqa is a free referencing and research tool. It is used to find, read, and annotate PDF files. We can quickly go through our work, write it up, and make bibliographies. It is an Excellent document and reference handling. It imports PDF files into different libraries. OCR and tag extraction are done automatically. Qiqqa can help fill in the gaps in millions of research publications' metadata. Full-text search, duplicate document identification, inbound and outbound linkages, and much more are just a few of the features available.

Built-in PDF reader with annotation, highlighting, and automated jump links, among other features. We may produce printable summaries of our notes within the Microsoft Word processor, mind maps of our thoughts, immediately credit our references, and automatically build bibliographies.

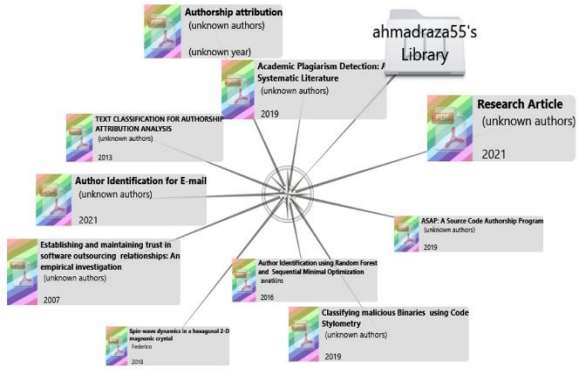


Fig 1: Related work for Authorship Contribution

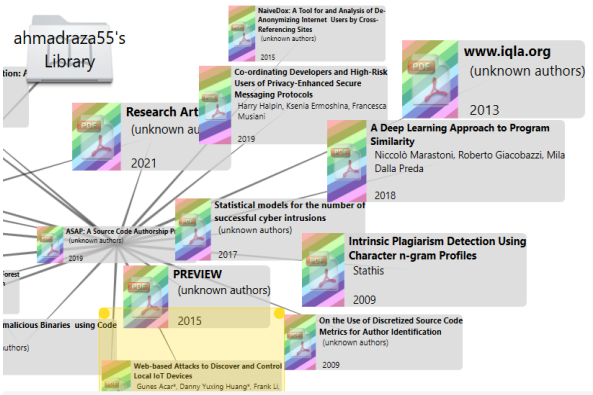


Fig 2: Related work based on expansion of proposed work

III. Research Methodology

We treat a source code file as sample data to identify the programmer of a document or source code file. We used numerous source code files to train our model. We used the test on other source code files when the training was completed.

Dataset

The Google Code Jam dataset programming competition, which ran from 2008 to 2021, provides a collection of solutions and code from previous Code Jam rounds. The dataset aims to facilitate experimentation with problems of varying levels of difficulty. Some files may be missing special characters and encodings, particularly among Chinese contestants. Additionally, due to modifications in the structure of the contest pages by Google, the file names for rounds held from 2018 to 2021 are slightly different.

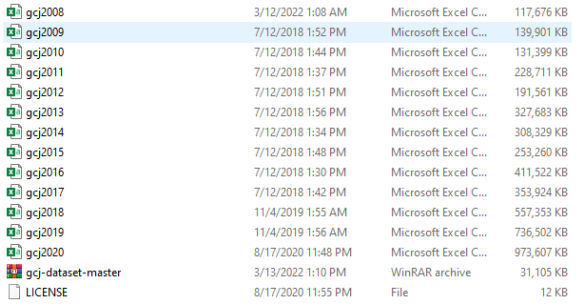
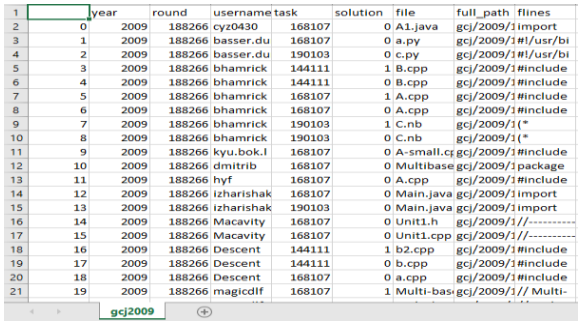


Fig 3: Dataset of programmers' competition containing source code files

Each .csv file consists of Year, Round, Username, Task, Solution, File, Full Path, and Felines, respectively. Flines data indicates the source code. Using that source code, we are working on our pre-processing steps. A sample view of gcj2009 is attached in the fig 4.



	year	round	username	task	solution	file	full_path	flines
2	0	2009	188266	cyz0430	168107	0	A1.java	gcj/2009/1	import
3	1	2009	188266	basser.du	168107	0	a.py	gcj/2009/1	#!/usr/bi
4	2	2009	188266	basser.du	190103	0	c.py	gcj/2009/1	#!/usr/bi
5	3	2009	188266	bhamrick	144111	1	B.cpp	gcj/2009/1	#include
6	4	2009	188266	bhamrick	144111	0	B.cpp	gcj/2009/1	#include
7	5	2009	188266	bhamrick	168107	1	A.cpp	gcj/2009/1	#include
8	6	2009	188266	bhamrick	168107	0	A.cpp	gcj/2009/1	#include
9	7	2009	188266	bhamrick	190103	1	C.nb	gcj/2009/1	(*
10	8	2009	188266	bhamrick	190103	0	C.nb	gcj/2009/1	(*
11	9	2009	188266	kyu.bok.l	168107	0	A-small.cp	gcj/2009/1	#include
12	10	2009	188266	dmitrib	168107	0	Multibase	gcj/2009/1	package
13	11	2009	188266	hyf	168107	0	A.cpp	gcj/2009/1	#include
14	12	2009	188266	izharishak	168107	0	Main.java	gcj/2009/1	import
15	13	2009	188266	izharishak	190103	0	Main.java	gcj/2009/1	import
16	14	2009	188266	Macavity	168107	0	Unit1.h	gcj/2009/1	//----------
17	15	2009	188266	Macavity	168107	0	Unit1.cpp	gcj/2009/1	//----------
18	16	2009	188266	Descent	144111	1	b2.cpp	gcj/2009/1	#include
19	17	2009	188266	Descent	144111	0	b.cpp	gcj/2009/1	#include
20	18	2009	188266	Descent	168107	0	a.cpp	gcj/2009/1	#include
21	19	2009	188266	magicdlf	168107	1	Multi-bas	gcj/2009/1	// Multi-

gcj2009

Fig 4: Dataset view for source code files

Pre-processing

Pre-processing is the process of doing some tasks before model implementation on the dataset.

In this article, following techniques have been used as pre-processing for the dataset.

1. Selecting required columns from csv i.e. Username and Flines.
2. Punctuation removal.
3. Tokenization of the punctuation removed dataset.
4. Replacing null with 0.
5. Calculation of TF-IDF values from tokens.
6. Rounding up the decimal values.

Simply we can take the pre-processing as cleaning, transforming and selection of data. It breaks the PDG into tokens and then calculates each token's frequency. The pre-processing steps include cleaning, instances, selection, transformation. As model implementation doesn't need noise in the data. TF-IDF values indicate the number of occurrences of each element indifferent source codes [15].

In the first step of working on the dataset, we read our CSV files using the python libraries. We extract two main features we have to work with and are concerned with. These features are username and Flines. Flines is the column in .csv files which contains the source code of different languages of different programmers. These two features are put into a list and then stored in a new CSV.

Punctuation Removal

The next step in the pre-processing of the dataset is punctuation removal. This step is done on the CSV files created with the two features having a username and flines (source code). The library used for performing punctuation removal is mentioned in fig 5.

```
#library that contains punctuation
import string
string.punctuation
```

Fig 5: Punctuation removal using Python

The output of this code having punctuation-free source codes corresponding to each username is saved to a new CSV named "Punctuation_free.csv." The output view is showed as in the figure 6.

```
public class EndlessKnight
    public static void mainString args
        Scanner in  new ScannerSystemin
        int numCases  innextInt
        for int i  0 i  numCases i
            doCase1  i in

    private static void doCaseint caseNum Scanner in
        int H  innextInt
        int W  innextInt
        int R  innextInt
        int memoized  new intWH
        for int arr  memoized
            for int i  0 i  arrlength i
                arri  1
        for int i  0 i  R i
            int rockY  innextInt
            int rockX  innextInt
            memoizedW  rockXH  rockY  0
        int total  findMovesW  1 H  1 memoized
        SystemoutprintlnCase  caseNum  total
    private static int findMovesint i int j int memoized
```

Fig 6: Punctuation removal output

Tokenization

The breaking of large text bodies into smaller chunks or words is knows as tokenization.

Tokenization applied to GCJ 2009 file is shown in the fig 7:

```
In [3]: #apply_tokenization
        import csv
        from nltk import word_tokenize

        data=pd.read_csv(r"D:\Thesis work\Dataset\output\2009 full data\punctuation_free2009.csv",encoding='latin1')
        dict1={"username":[],"tokens":[]}

        for i in range(0,len(data)):
            dict1["username"].append(str(data["username"][i]))
            var=word_tokenize(str(data["flines"][i]).replace("\n"," ").replace("\t"," "))
            var_list=[]
            for j in var:
                if j.isdigit():
                    pass
                else:
                    var_list.append(j)
            dict1["tokens"].append(var_list)

        token_dataframe=pd.DataFrame(dict1)
        token_dataframe.to_csv(r"D:\Thesis work\Dataset\output\2009 full data\tokens2009.csv')
```

Fig 7: Python implementation for Tokenization

In tokenization the desired output is the tokens of whole code, and these tokens are considered to be the stylometric features. On the basis of these features the TF-IDF values are generated and model is trained output CSV sample is shown the fig 8:

```
64202,marek.cygan,"['include', 'cstdio', 'include', 'iostream', 'include', 'algorithm', 'include', 'set', 'include', '
64203,marek.cygan,"['include', 'cstdio', 'include', 'iostream', 'include', 'algorithm', 'include', 'set', 'include', '
64204,marek.cygan,"['include', 'cstdio', 'include', 'iostream', 'include', 'algorithm', 'include', 'set', 'include', '
64205,marek.cygan,"['include', 'cstdio', 'include', 'iostream', 'include', 'algorithm', 'include', 'set', 'include', '
64206,AdrianRuegel,"['include', 'stdioh', 'include', 'iostream', 'include', 'sstream', 'include', 'algorithm', 'includ
64207,AdrianRuegel,"['include', 'stdioh', 'include', 'iostream', 'include', 'sstream', 'include', 'algorithm', 'includ
64208,AdrianRuegel,"['include', 'stdioh', 'include', 'iostream', 'include', 'sstream', 'include', 'algorithm', 'includ
64209,Vitaliy,"['import', 'javaio', 'import', 'javautil', 'imports', 'javamath', 'public', 'class', 'Main', 'int', 'n',
64210,Vitaliy,"['import', 'javaio', 'import', 'javautil', 'public', 'class', 'Main', 'Scanner', 'in', 'PrintWriter', '
64211,Vitaliy,"['import', 'javaio', 'import', 'javautil', 'public', 'class', 'Main', 'Scanner', 'in', 'PrintWriter', '
64212,Vitaliy,"['import', 'javaio', 'import', 'javautil', 'public', 'class', 'Main', 'Scanner', 'in', 'PrintWriter', '
64213,Vitaliy,"['import', 'javaio', 'import', 'javautil', 'public', 'class', 'Main', 'Scanner', 'in', 'PrintWriter', '
64214,Vitaliy,"['import', 'javaio', 'import', 'javautil', 'public', 'class', 'Main', 'Scanner', 'in', 'PrintWriter', '
64215,hos.lyric,"['GCJ', 'World', 'Finals', 'A', 'Author', 'hoslyric', 'include', 'cstdio', 'include', 'cstdlib', 'inc
64216,hos.lyric,"['GCJ', 'World', 'FInals', 'E', 'Author', 'hoslyric', 'include', 'cstdio', 'include', 'cstdlib', 'inc
64217,hos.lyric,"['GCJ', 'World', 'FInals', 'D', 'Author', 'hoslyric', 'include', 'cstdio', 'include', 'cstdlib', 'inc
64218,hos.lyric,"['GCJ', 'World', 'Finals', 'C', 'Author', 'hoslyric', 'include', 'cstdio', 'include', 'cstdlib', 'inc
64219,hos.lyric,"['GCJ', 'World', 'Finals', 'B', 'Author', 'hoslyric', 'include', 'cstdio', 'include', 'cstdlib', 'inc
64220,nika,"['import', 'javaioFile', 'import', 'javaioFileNotFoundException', 'import', 'javaioPrintWriter', 'import',
64221,nika,"['import', 'javaioFile', 'import', 'javaioFileNotFoundException', 'import', 'javaioPrintWriter', 'import',
64222,nika,"['include', 'cstdio', 'include', 'cstdlib', 'include', 'cmath', 'include', 'string', 'include', 'vector',
64223,nika,"['include', 'cstdio', 'include', 'cstdlib', 'include', 'cmath', 'include', 'string', 'include', 'vector',
64224,nika,"['include', 'cstdio', 'include', 'cstdlib', 'include', 'cmath', 'include', 'string', 'include', 'vector',
64225,nika,"['include', 'cstdio', 'include', 'cstdlib', 'include', 'cmath', 'include', 'string', 'include', 'vector',
64226,nika,"['include', 'cstdio', 'include', 'cstdlib', 'include', 'cmath', 'include', 'string', 'include', 'vector',
```

Fig 8: Output of Tokenization for source code files

TF-IDF

Based on the information retrieved from the code segments, we can generate feature vectors.

It's only the frequency of each word or keyword and the document frequency in reverse (TF-IDF). We can use TF-IDF for determining that which keyword is more essential for the author who used it. The TF-IDF value is the multiplication of the ratio of a word in a document by the reciprocal ratio used in all documents.

Bag of Words Extraction

The bag-of-words model is a best natural language processing model. In this model, text is considered to be collection of words. In other words, it's the representation of missing document attributes in the form of frequencies that appears in the document to build a dictionary. In this dictionary, characters, character n-grams, words, words n-grams, and other text characteristics may be found.

Steps in calculating TF/IDF calculation

First, it was necessary to input the data in the processed form. For that purpose, I have pre-processed the dataset explained in the above sections, i.e.,

Selecting the relevant features from the CSV, punctuation removal, and tokenization. Tokenization was a challenging task because tokenizing the data with nearly 175000 rows and then storing it into a new CSV was difficult as tokens are comma-separated objects. In CSV, each column is also a comma-separated index. SO for that purpose, I have searched out more ways to store and finally did it using the tokenization function in python. The next task to calculate TF/IDF more challenging. Many ways can do it.

The first idea was to put all the tokens into a txt corpus and then give it input to the TF/IDF function to process and then calculate it. The output will be in the form that there would be two columns. One for the tokens list, and the next would be the TF/IDF values against each token. Then I would have to apply the transformation function to change the view of CSV to adjust the username against tokens and their TF/IDF values. The output would be that the columns will be a features list, i.e., the tokens used by each programmer, and under them would be their values. Next, I have to save the TF/IDF values against each programmer in the form of a text file so that a check would have to be applied. The purpose of the check is that csv has multiple codes against the same author so that the text files would be against each row. The check would determine if the same username exists, then append the txt with the previous and if it does exist, then create a new txt. In this case, more text would have been generated. But to handle all this, we did this in a single CSV. Path for tokens generated csv is given as input for TF-IDF calculation and result generates as shown in the fig 9.

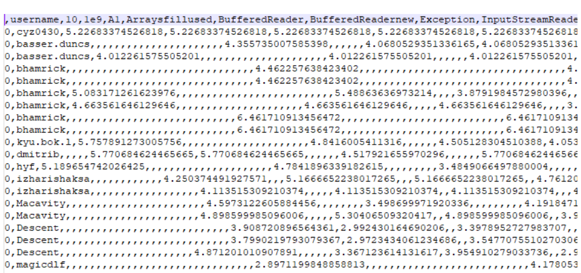


Fig 9: Output of TF-IDF csv file

But the issue was to handle such a large dataset to train the algorithm as the output file for TF-IDF contains nearly 64000 features and 3000 rows. So, a mechanism required to handle the situation.

Iteration on 5 Programmers

To handle the situation, I took up the data of % programmers and passed it into whole procedure i.e., Selecting up two required columns (username and flines code), punctuation removal, Tokenization, Tf-IDF values and parsing up into H20 tool for calculation. The overall interface was shown in the fig 10.

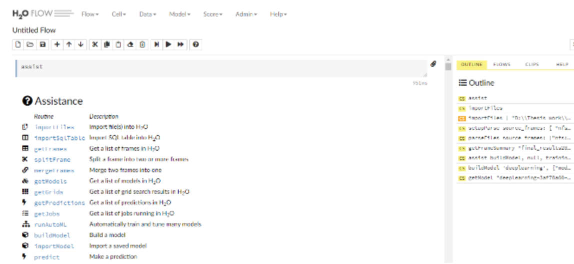


Fig 10: H2O programmers' tool overview

Steps for implementation on H20

1.Importing files from device.

Fig 11: Importing files using H2O

2.Parsing of files.

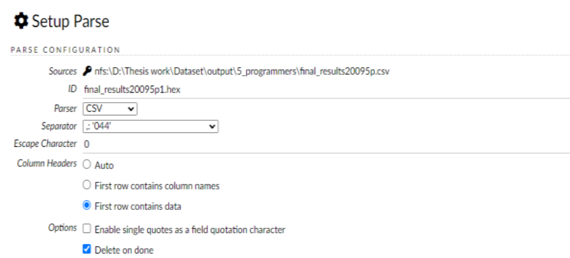


Fig 12: Parsing files using H2O

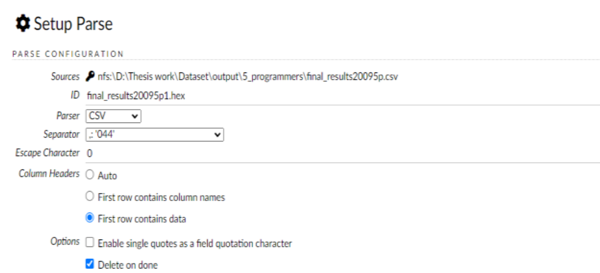


Fig 13: Set-up Parsing files using H2O

Model Building

Next step is to apply model for training. First of all select model parameters and then apply model building.

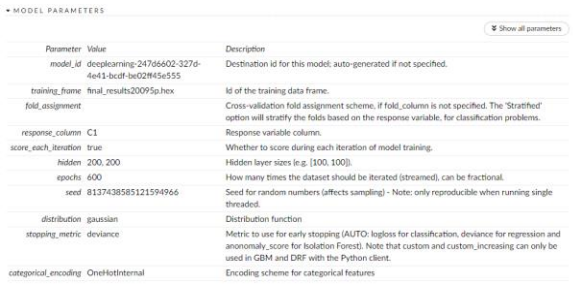


Fig 14: Model Parameters for dataset

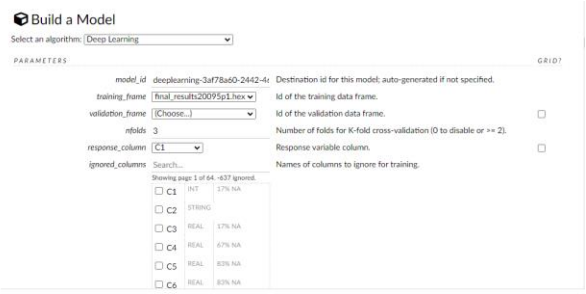


Fig 15: Building model using Deep Learning

Iteration on 15 Programmers
First of all import dataset and these repeat the steps as mentioned above. The purpose is to check on each iteration that when the training and prediction stops and overall result is generated.

OUTPUT - CROSS-VALIDATION METRICS SUMMARY

	mean	sd	cv_1_valid	cv_2_valid	cv_3_valid
mae	6.3398	1.0606	7.5535	5.5915	5.8744
mean_residual_deviance	58.8374	24.9004	87.4999	46.4761	42.5364
mse	58.8374	24.9004	87.4999	46.4761	42.5364
r2	-0.1756	0.2098	-0.0179	-0.0952	-0.4136
residual_deviance	58.8374	24.9004	87.4999	46.4761	42.5364
rmse	7.5645	1.5569	9.3541	6.8173	6.5220
rmsle	0.7991	0.3459	1.1560	0.7761	0.4653

Fig 16: Cross Validation Summary for 15 programmers

Iteration on 50 Programmers
When we apply all the pre-processing steps and generate a final TF-IDF csv file for 50 programmers then it can be considered as input for all the training metrices.
Variable importance for each of the metrices for 50 programmers is shown in Fig 17.

OUTPUT - VARIABLE IMPORTANCES

variable	relative_importance	scaled_importance	percenta
C143	1.0	1.0	0.00
C722	0.9801	0.9801	0.00
C1577	0.9539	0.9539	0.00
C3837	0.9533	0.9533	0.00
C2321	0.9507	0.9507	0.00
C2766	0.9487	0.9487	0.00
C683	0.9480	0.9480	0.00
C224	0.9461	0.9461	0.00
C719	0.9450	0.9450	0.00
C1402	0.9445	0.9445	0.00
C528	0.9435	0.9435	0.00
C937	0.9427	0.9427	0.00
C2.cedricl	0.9423	0.9423	0.00
C124	0.9420	0.9420	0.00
C67	0.9411	0.9411	0.00
C1079	0.9393	0.9393	0.00
C2822	0.9392	0.9392	0.00
C35	0.9386	0.9386	0.00
C1843	0.9378	0.9378	0.00

Fig 17: Output variable importance for 15 programmers

Overall methodology of the proposed work is to take sample data, then pre-process it accordingly and then finally train it using Deep Learning Algorithm and then match the result on testing data.

IV. Results and Discussion

In this section, we will discuss findings and results of our methodology proposed in the above section. As discussed in the methodology section, dataset is broken into number of programmers and based in that division, results are discussed for each of the iteration.

Iteration on 5 programmers

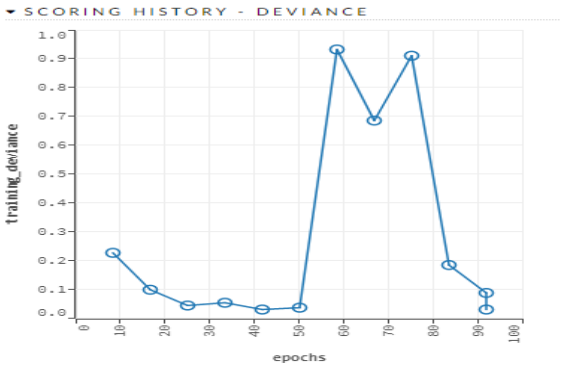


Fig 18: Scoring history for 5 programmers

OUTPUT - SCORING HISTORY

timestamp	duration	training_speed	epochs	iterations	samples	training_rmse	training_deviance	training_mae	training_r2
2022-03-26 00:18:34	0.000 sec		0	0	0	.	.	.	.
2022-03-26 00:18:34	0.823 sec	289 obs/sec	0.3333	1	50.0	0.4776	0.2281	0.4370	0.8860
2022-03-26 00:18:35	1.466 sec	195 obs/sec	16.6667	2	100.0	0.3168	0.1004	0.2124	0.9498
2022-03-26 00:18:36	2.084 sec	189 obs/sec	25.0	3	150.0	0.2135	0.0456	0.1429	0.9772
2022-03-26 00:18:36	2.725 sec	173 obs/sec	33.3333	4	200.0	0.2350	0.0552	0.1527	0.9724
2022-03-26 00:18:37	3.249 sec	179 obs/sec	41.6667	5	250.0	0.1781	0.0317	0.1129	0.9841
2022-03-26 00:18:37	3.868 sec	178 obs/sec	50.0	6	300.0	0.1953	0.0381	0.1213	0.9809
2022-03-26 00:18:38	4.422 sec	182 obs/sec	58.3333	7	350.0	0.9655	0.9321	0.8434	0.5339
2022-03-26 00:18:39	5.023 sec	178 obs/sec	66.6667	8	400.0	0.8283	0.6860	0.7167	0.6570
2022-03-26 00:18:39	5.606 sec	179 obs/sec	75.0	9	450.0	0.9542	0.9106	0.8253	0.5447
2022-03-26 00:18:40	6.167 sec	176 obs/sec	83.3333	10	500.0	0.4311	0.1858	0.3670	0.9071

Fig 19: Scoring History-2 for 5 programmers

OUTPUT

original_names	·
cross_validation_predictions	·
cross_validation_holdout_predictions_frame_id	·
cross_validation_fold_assignment_frame_id	·
model_category	Regression
validation_metrics	·
status	·
start_time	1647872325970
end_time	1647872327294
run_time	1324
default_threshold	0.500000
weights	·
biases	·
normmul	·
normsub	·
normrespmul	·
normrespsub	·
catoffsets	·

Fig 20: Output Metrices for 5 programmers

Cross Validation

CROSS_VALIDATION_MODELS

cross_validation_models
deeplearning-2242aa2e-2be7-4e58-977f-81fef5e7a109_cv_1
deeplearning-2242aa2e-2be7-4e58-977f-81fef5e7a109_cv_2
deeplearning-2242aa2e-2be7-4e58-977f-81fef5e7a109_cv_3

Fig 21: Cross Validation for 15 Programmers

VARIABLE IMPORTANCES

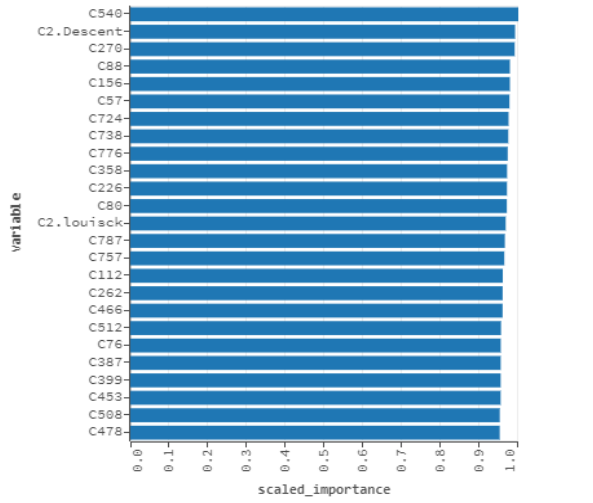


Fig 22: Variable importance of each variable for 15 programmers

OUTPUT - VARIABLE IMPORTANCES

variable	relative_importance	scaled_importance	percentag
C540	1.0	1.0	0.00
C2.Descent	0.9918	0.9918	0.00
C270	0.9907	0.9907	0.00
C88	0.9784	0.9784	0.00
C156	0.9784	0.9784	0.00
C57	0.9772	0.9772	0.00
C724	0.9749	0.9749	0.00
C738	0.9742	0.9742	0.00
C776	0.9727	0.9727	0.00
C358	0.9711	0.9711	0.00
C226	0.9708	0.9708	0.00
C80	0.9705	0.9705	0.00
C2.louisck	0.9676	0.9676	0.00
C787	0.9653	0.9653	0.00
C757	0.9640	0.9640	0.00
C112	0.9601	0.9601	0.00
C262	0.9596	0.9596	0.00
C466	0.9595	0.9595	0.00
C512	0.9554	0.9554	0.00
C76	0.9547	0.9547	0.00

Fig 23: Variable importance_2 of each variable for 15 programmers

V. Conclusion and Future Work

This paper discusses the use of code smells and aesthetic features in authorship detection. Author attribution is a technique that involves extracting features to identify the author of a piece of code. In this study, the effectiveness of both stylistic features, which are related to the author's writing style, and code smells, which are a novel contribution, are evaluated for feature representation in source code. To determine the author's style and code smells, two test cases are examined. The first test case determines the author's stylistic proficiency, while the second test case focuses on the combined performance of style-related features and code smells. The extracted features were found to be useful in authorship attribution when paired with code smells and stylistic traits, resulting in higher classification accuracy. In this paper, deep learning is utilized for model training and to analyze the results. Based on the findings of this study, future attempts will be made to perform multi-author attribution of source code.

References

[1] J. A. García-Díaz and R. Valencia-García, "UMUTeam at AI-SOCO'2020: Source Code Authorship Identification based on Character N-Grams and Author's Traits," in *FIRE (Working Notes)*, 2020, pp. 717-726.

[2] J. Gravill and D. Compeau, "Self-regulated learning strategies and software training," *Information & Management*, vol. 45, no. 5, pp. 288-296, 2008.

[3] K. L. Verco and M. J. Wise, "Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems," in *ACM International Conference Proceeding Series*, 1996, vol. 1, pp. 81-88.

[4] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, "Source code authorship attribution using long short-term memory based networks," in *European Symposium on Research in Computer Security*, 2017, pp. 65-82: Springer.

[5] C. Qian, T. He, and R. Zhang, "Deep learning based authorship identification," *Report, Stanford University*, pp. 1-9, 2017.

[6] E. Flores Sáez, L. A. Barrón-Cedeño, L. A. Moreno Boronat, and P. Rosso, "Cross-language source code re-use detection using latent semantic analysis," *Journal of Universal Computer Science*, vol. 21, no. 13, pp. 1708-1725, 2015.

[7] P. Mahbub, N. Z. Oishie, and S. R. Haque, "Authorship identification of source code segments written by multiple authors using stacking ensemble method," in *2019 22nd International Conference on Computer and Information Technology (ICCIT)*, 2019, pp. 1-6: IEEE.

[8] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 479-496.

[9] D. Ganguly, G. J. Jones, A. Ramírez-De-La-Cruz, G. Ramírez -De - La - Rosa, and E. Villatoro-Tello, "Retrieving and classifying instances of source code plagiarism, "*Information Retrieval Journal*, vol. 21, no. 1, pp. 1-23, 2018.

[10] F. Rangel, F. González, F. Restrepo, M. Montes, and P. Rosso, "PAN@ FIRE: overview of the PR-SOCO track on personality recognition in SOurce COde," in *Forum for Information Retrieval Evaluation*, 2016, pp. 1-19: Springer.

[11] I. Krsul and E. H. Spafford, "Authorship analysis: Identifying the author of a program," *Computers & Security*, vol. 16, no. 3, pp. 233-257, 1997.

[12] Q. Song, Y. Zhang, L. Ouyang, and Y. Chen, "BinMLM: Binary Authorship Verification with Flow-aware Mixture-of-Shared Language Model," *arXiv preprint arXiv:2203.04472*, 2022.

[13] Z. Li, "Cross-Lingual Transfer Learning Framework for Program Analysis," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1074-1078: IEEE.

[14] D. Vagavolu, K. C. Swarna, and S. Chimalakonda, "A Mocktail of Source Code Representations," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1296-1300: IEEE.

[15] E. Haddi, X. Liu, and Y. Shi, "The role of text pre-processing in sentiment analysis," *Procedia computer science*, vol. 17, pp. 26-32, 2013.