

UoB at AI-SOCO 2020: Approaches to Source Code Classification and the Surprising Power of n -grams

Alexander Crosby, Harish Tayyar Madabushi

University of Birmingham, Edgbaston, Birmingham, B15 2TT, United Kingdom

Abstract

Authorship identification of source code is the process of identifying the composer of given source code. Code authorship identification plays an important role in many real-world scenarios, such as the detection of plagiarism and ghost writing in both education and workplace settings. Additionally, it can allow the identification of individuals or organisations that produce and distribute malware programs.

In this paper we describe the experimentation and submission by team UoB to the AI-SOCO track at FIRE 2020, which achieved first place. We first perform extensive testing on a variety of techniques used in source code authorship identification including n -gram, stylometric, and abstract syntax tree derived features. We also investigate the application of CodeBERT, a new pre-trained model that demonstrates state-of-the-art performance in natural and programming language tasks. Finally, we explore the potential of ensembling multiple models together to create a single superior model. Our winning model utilises byte-level n -grams extracted from source codes to build feature vectors that represent an author's programming style. These feature vectors are then used to train a densely connected neural network model to carry out authorship classification on previously unseen source codes, achieving an accuracy of 95.11%.

Keywords

authorship identification, source code, machine learning, n -grams

1. Introduction

Source code authorship attribution is the task of identifying the author of a given piece of code [1]. The main concept behind authorship attribution is that each author uses a number of stylistic traits when writing code that can be used as a fingerprint to distinguish one author from another [2]. Authorship identification, therefore, is accomplished by identifying these stylistic fingerprints and using statistical and machine learning models to attribute source code to an author [3].

There are a number of real world applications of source code attribution, such as the detection of plagiarism [4], or the use of a ghost-writer in academic, workplace and other environments. Additionally, code authorship attribution can be used to identify authors of malware [5], who may attempt to conceal their identity or may obfuscate their code to hide its function and origin [6].

Forum for Information Retrieval Evaluation, December 16-20, 2020, Hyderabad, India

EMAIL: AlexCrosby@live.co.uk (A. Crosby); Harish@HarishTayyarMadabushi.com (H. T. Madabushi)

URL: <https://HarishTayyarMadabushi.com/> (H. T. Madabushi)

ORCID: 0000-0001-5260-3653 (H. T. Madabushi)



© 2020 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

Authorship Identification of Source Code (AI-SOCO) 2020 is a track organised for the Forum for Information Retrieval Evaluation (FIRE) 2020 [7]. The track asks for the identification of effective strategies that solve the problem of source code authorship attribution in issues related to “cheating, in academic, work and open source environments” and that help in the detection of authors of malware software [7].

The track involves a pre-defined set of 100,000 source codes made up from 1,000 unique authors who had submitted code as part of Codeforces online programming competitions [8]. This data set was broken down to create a training, development, and test set of source codes. The test set contained 25,000 source codes which did not have an attributed author, leaving 75,000 pieces of source code for training and development. Of this, 50,000 source codes were used for training and 25,000 for development. Using these data sets, participants were required to build a system that determined the author of each unlabelled source code. In the development stage, participants are ranked based on their model’s performance on the development set. In the evaluation stage, participants are ranked on their model’s performance on the 25,000 unlabelled source codes.

This paper describes our submissions to the AI-SOCO 2020 track¹. Our multi-faceted approach evaluates a variety of different techniques previously used in authorship identification tasks and additionally creates an ensemble model which pulls on the strengths of each individual model. We find that our n -gram-based approach described in Section 3.3.2 outperformed all other models we investigated, including modern state-of-the-art approaches, and achieved first place on both the development and test datasets.

2. Related Work

This section provides an overview of different approaches related to authorship identification attribution to identify promising techniques that inform the construction of our models.

2.1. Stylometric Approaches

Stylometry is perhaps the oldest technique in code authorship attribution and is based on the identification of features in written text that could be linked to stylistic choices in a person’s writing. One such example is the work of Krsul and Spafford [9], who proposed a total of 49 different features in 3 main areas: (1) layout specific metrics such as indentation and comment style; (2) programming style such as variable, function and comment naming style; and (3) program structure such as usage of specific data structures, presence of debugging identifiers or assertions, and error handling.

2.2. n -gram Approaches

n -grams have been used successfully in a variety of Natural Language Processing (NLP) tasks, including spell checking, language modelling and authorship attribution of text and have successfully been ported to the field of authorship attribution.

¹Source code and data are published at <https://github.com/AlexCrosby/AI-SOCO>

Frantzeskou et al. [10] introduces the profile-based Source Code Author Profile (SCAP) method. This method is a profile-based approach that creates a profile of frequently used byte-level n -grams for each author. Classification can then be carried out by finding the profile that best matches an unlabelled source code by counting common n -grams between source code and profile.

Kothari et al. [11] demonstrated that character-level n -grams could be combined with stylistic features to boost prediction accuracy, suggesting that combining different authorship identification techniques could be a promising approach to use in this track.

2.3. Abstract Syntax Tree Approaches

Abstract Syntax Trees (ASTs) are a representation of the abstract syntactic structure of a source code, represented in a hierarchical tree structure, that are a common source of information in program analysis tasks.

Caliskan-Islam et al. [12] proposed a stylometric feature set extracted from a source code's AST and combining that with lexical and layout features elicited directly from the source code, in line with more classical stylometry methodologies and running them through a random forest classifier. Caliskan-Islam et al. [12] then found that the feature that best discriminated between authors was AST node bigrams. Each bigram was made up from a node and the parent node it was directly connected to. The term frequencies of these bigrams, when used for classification, demonstrated an accuracy comparable to using these bigrams in combination with the extracted lexical and layout features.

2.4. CodeBERT

CodeBERT [13] is a pre-trained multi-layer bidirectional Transformer model based on the same architecture used in RoBERTa. Unlike RoBERTa, CodeBERT is designed to be used in both natural language and programming language applications, such as code documentation generation and natural language code search. CodeBERT produces word-level vectors for a source code using contextual information taken from surrounding word contexts, as opposed to information from the code AST as is done by models such as code2vec [14] and code2seq [15].

3. Methodology

This section outlines the techniques used to generate each model used for authorship attribution in the AI-SOCO 2020 track. Each model detailed was devised to identify meaningful vector embeddings for source codes and assess their overall effectiveness, with the end goal being the ensembling of a variety of different techniques to provide an overall superior model.

3.1. Preprocessing

Depending on the features required by each model, the source codes were preprocessed to make extraction of said features possible. Two different preprocessed datasets were generated for this purpose. The first of these involved the removal of all comment lines and the unfolding of

“#define” preprocessor directives which may otherwise obfuscate features extracted from the program structure and is a requirement for AST extraction.

The second preprocessed dataset was the ASTs for each source code extracted directly from the first preprocessed dataset.

Both of these preprocessing stages were carried out using the tool `astminer` [16] using Google Cloud Platform’s Compute Engine.

3.2. Initial Experimentation

Our initial experimentations were expansions on the character count and bag-of-words term frequency–inverse document frequency (TF-IDF) vectorisation techniques used in the baseline models; plus a variant on the character count where only A-Z characters were considered, ignoring case. These vectors were all used to train four different machine learning models: Logistic Regression; k -nearest neighbours; Naïve Bayes classifier; and a support vector classifier.

While these models did not perform particularly well, achieving a best accuracy of 74.996% using the TF-IDF vectors in a Logistic Regression model shown in Section 3, they did identify that the Naïve Bayes classifier was both quick to train and had moderate accuracy, meaning it was a good candidate model to evaluate vectorisation techniques used in further models.

3.3. n -gram Models

3.3.1. Source Code Author Profile Method

The SCAP method discussed in Section 2.2 was implemented based on its success in cases where only very short pieces of code were available, such as in our dataset [17].

In this method, character or byte-level n -grams are extracted from the raw source codes and used to create vector representations. This method involves the generation of a profile for each author by creating a set of the L most commonly occurring n -grams for each author throughout their source codes. Then, to calculate an unseen source code’s similarity with each profile, the Simplified Profile Intersection (SPI) is measured.

Letting P_a be the author profile n -gram set and P_s being the n -gram set of the previously unseen source code, the SPI value between P_a and P_s is given by the magnitude of the intersection of the two profiles (see Equation 1).

$$SPI = |P_a \cap P_s| \quad (1)$$

The unseen source code is hence attributed to the author profile achieving the highest SPI value.

To select n -gram size n , profile length L and whether to use character or byte n -grams, an exhaustive grid search was carried out to identify the best performing settings. In addition to a set L value, we also investigated using an unlimited profile length and excluding n -grams that only have a frequency of 1 as proposed by Tennyson and Mitropoulos [18]. Overall, this technique managed to achieve an accuracy of 92.212% on the development dataset as shown in Section 4.1.1.

3.3.2. Instance-based n -gram Models

A second n -gram based approach was proposed based on the success of the SCAP method (Section 3.3.1). In this model, the raw source codes were decomposed into their constituent character or byte-level n -grams. Unlike SCAP however, these n -grams were represented as a bag of n -grams which would be used to train machine learning models to predict authorship through the co-occurrence of n -grams in any given source code.

A Naïve Bayes classifier was used to identify the best candidate representations that would later be used to train the neural network.

The best candidate n -gram representation identified at this stage was then used downstream as the input in a neural network classifier model.

In addition to character and byte-level n -grams, a bag-of-Words (BoW) model was also conceived. In this model, vectors represented word-level n -grams extracted from the training dataset. Unlike the other n -gram models, only word n -grams of size 1 were used. However, the entire vocabulary of the training dataset, a total of 60,770 words, is used rather than limiting them to a specified max size as discussed in Section 5.5.

This model achieved an accuracy of 95.416% on the development dataset as detailed in 4.1.2. Due to the increased accuracy of this instance-based model over the profile-based SCAP method, going forwards only this instance-based n -gram model was used. The final accuracy of this model on the test dataset was 95.11% as mentioned in Section 4.4.

3.4. Abstract Syntax Tree Model

In this approach, features were derived from the preprocessed dataset containing the AST structures derived from the source codes. This model was developed based on its successful implementation by Caliskan-Islam et al. [12] and Wisse and Veenman [19].

These ASTs contain information relating to the node type, the code that relates to said node, and the relationships between each node. These nodes are then tokenised by substituting each unique node with a numerical representation. Likewise n -grams of nodes are tokenised, in which a node bi-gram would be a node and its parent node and so on.

These tokens are then used to create a vector representation that describe the occurrence of the top most commonly occurring tokens in the training dataset. Like the n -gram model, multiple AST vectors variants were evaluated in order to identify best performing vector parameters. These parameters were then to generate the vectors used downstream to train a neural network classifier model. The final accuracy achieved by this model on the development dataset was 80.052% as shown in Section 4.2.

3.5. Stylometry Model

In this model, 136 different stylistic and layout features were extracted from both the raw and preprocessed source codes without comments and “#define” directives. 100 of these 136 features were counts of the printable characters as used in the baseline model.

The remaining 36 features are documented in the repository released alongside this paper and were collated due to their common use in multiple papers on this topic [12, 20, 21].

136-dimensional vectors representing these features from each source code were then used to train a densely-connected neural network classifier model. This model achieved an accuracy of 75.376% on the development dataset as shown in Section 3.

3.6. CodeBERT Model

This model introduces CodeBERT to the task of authorship attribution, a domain that, to this author's knowledge, the model has not been applied to previously.

We fine-tuned the CodeBERT model (provided at <https://github.com/microsoft/CodeBERT>) for authorship attribution using an NVIDIA K80 GPU on an Amazon Web Services p2.xlarge instance for 10 epochs using the Adam optimiser at a learning rate of 2×10^{-5} . As shown in Section 3, this model achieved an accuracy of 86.724% on the development dataset.

3.7. Weighted Average Ensemble

The idea behind ensembling is that each model, when independently trained, is likely to have different strengths while classifying source codes. By combining individual models their strengths can be pulled together to get a more accurate classification than by any one model alone [22].

Following the creation of the previously mentioned models, five candidates were selected based on performance and difference in feature representations. These models were: The n -gram, BoW, AST, stylometry and CodeBERT models.

One ensembling technique experimented with was a weighted averaging procedure. In an average ensemble this is achieved by simply averaging the SoftMax outputs from each model to get the average prediction of all models. A common problem with this method is that if one model performs significantly worse, it can drag the overall prediction accuracy down. To combat this, each model is given a different weighting, allowing some models to contribute more to the pooled classification, and others less [22].

Two different optimisation techniques were tested in identifying the best weight values: Powell's conjugate direction method [23], and Differential Evolution [24]. These two methods were used as an attempt to avoid getting stuck in a local minimum due to the reliance on any single optimisation method. The best performing weights on the development dataset were selected to be used in the final ensemble. This model outperformed all other models on the development dataset, achieving an accuracy of 95.715% as discussed in Section 4.1.2. However, it did not manage to outperform the instance-based n -gram model on the test dataset, achieving an accuracy of 95.11% as shown in Section 4.4.

3.8. Discarded Methods

3.8.1. Convolutional Neural Network Model

Another deep-learning model evaluated was a Convolutional Neural Network (CNN) model. In this model, each word in the source codes are given a numerical token representation. This ordered list of tokens is then fed into the CNN model.

This model contains five main layers. The first is the embedding layer, that learns a multidimensional vector representation for each word during training specific to our task.

This sequence of embeddings is then fed into a 1-dimensional convolutional layer which extracts features useful for classification decisions over the sequence of word embeddings. This is then fed into a MaxPooling layer to down sample the convolutional layer’s output feature map to reduce model size and training time.

It is this pooled layer that is then fed into the LSTM layer of the CNN to generate a final source code vector than is then classified with a final SoftMax layer. Unfortunately, in the initial testing of this model the results were poor, failing to surpass any of our previous models with an accuracy of 73.880% as shown in Section 3. As we did not have enough time to fully explore the capability of this model, we decided to discontinue further research on this model in favour of our better performing models.

3.8.2. Stacked Neural Network Ensemble

In addition to the weighted average ensemble discussed in Section 3.7, another stacked neural network ensembling technique was trialled.

In this model, a single multi-headed neural network model was constructed. This model uses the same architectures as the individual models but concatenates them all at their final hidden layer before the final SoftMax classification layer. This also allows all models to be trained simultaneously, allowing for differences in the models from the original versions that allow for improved ensembling.

Due to the size of this model, a few concessions were made to reduce its size: first, the CodeBERT model was excluded from the ensemble since it was the largest individual model by a considerable margin. Instead, the final hidden layer values from the CodeBERT model were pre-calculated and input directly into this model at the concatenation layer. Additionally, only the single best n -gram model from those discussed in Section 3.3 was used.

Due to the size of this model, it experienced significant overfitting issues during development which would have required significant changes to overcome. This model was hence discarded in favour of the more promising weighted average ensemble strategy.

4. Results

This chapter outlines our findings and analysis of results of the application of our models defined in Section 3.

The highest accuracies achieved at each stage of experimentation on the development dataset are presented in Table 1. The n -gram-based neural network model achieved the best individual model accuracy at 95.416%. The top ensemble accuracy achieved was 95.716%, achieved through the weighted averaging ensemble of the five models, discussed in Section 3.7.

All models were evaluated based on their accuracy on the development dataset as that was the only metric considered in evaluating and ranking submitted systems in the AI-SOCO 2020 track.

Table 1
Final Model Results on the Development Dataset.

Model	Accuracy (%)
Weighted Average Ensemble	95.715
<i>n</i> -gram Model	95.416
SCAP Model	92.212
Stacking Neural Network Ensemble	89.160
CodeBERT Model	86.724
BoW Model	82.960
AST Model	80.052
Stylometry Model	75.376
Word TF-IDF Logistic Regression Initial Model	74.996
Convolutional Neural Network Model	73.880

4.1. *n*-gram Results

4.1.1. Source Code Author Profile Method

The highest accuracy achieved by the SCAP method on the development dataset was 92.212% was the highest accuracy achieved using $n = 9$ and $L = 8000$ on character-level n -grams.

4.1.2. *n*-gram Model Results

As outlined in Section 3.3.2, initial vector parameter exploration was carried out using a Naïve Bayes Model to find the best combination to feed to the neural network classifier model.

The best overall accuracy at this stage was achieved using character-level n -gram vectors made up of the normalised feature count of the top 10,000 occurring 8-grams in each of the source code and achieved an accuracy of 85.648% on the development dataset.

During the neural network hyperparameter optimisation stage, we found that different vector parameters were more effective. Experimentations on these parameters in the neural network model are shown in Table 2.

The final vector parameters selected used byte-level 6-grams, using binary representation of the top 20,000 most commonly occurring 6-grams. This achieved the final accuracy of 95.416% on the development dataset as shown in Table 1. The architecture of the neural net used two hidden layers containing 3,000 and 2,000 neurons respectively. The model also used dropout layers with a dropout rate of 0.5 between the two hidden layers and between the second hidden layer and final output layer. The model used an initial learning rate of 1×10^{-4} using the RMSProp algorithm.

4.1.3. Bag-of-Words Model Results

For the BoW model, we found that the highest accuracy achieved was 82.96% on the development dataset, using a binary enumeration representation.

Table 2*n*-gram Vector Exploration Results Using Neural Network Classifier on the Development Dataset.

<i>n</i> -gram Size	Enumeration Type	<i>n</i> -gram Level	Accuracy (%)
6	Binary	Byte	95.416
7	Binary	Byte	95.292
5	Binary	Byte	95.240
4	Binary	Byte	95.204
6	Binary	Character	95.135
8	Binary	Byte	95.072
9	Binary	Byte	95.000
10	Binary	Byte	94.860
8	Count	Byte	93.336
8	Count	Character	93.191
6	Count	Byte	93.180
8	TF-IDF	Byte	92.080
6	TF-IDF	Byte	91.264

4.2. Abstract Syntax Tree Model Results

The highest accuracy achieved using ASTs used node unigrams. These nodes were enumerated with a binary count representation, in which only the top 20,000 most commonly occurring nodes were represented. Using a Naïve Bayes classifier model, this representation achieved an overall accuracy of 64.308% on the development dataset. This vector representation was then used to train a neural network classifier, achieving an accuracy of 80.052% on the development dataset as shown in Table 1.

4.3. Weighted Average Ensemble Results

Both Powell’s conjugate direction method and Differential Evolution were used as optimisation techniques to find the best weights for the models of the ensemble as discussed in Section 3.7. Both algorithms concurred on the weights for each model, that when used in ensembling, gives an accuracy of 95.716% on the development dataset, as shown in Table 1. The weights derived by Powell’s method and Differential Evolution for each model were: *n*-gram - 0.3079; CodeBERT - 0.19504; BoW - 0.09437; AST - 0.31464; and stylometry - 0.08805.

4.4. Submitted Results

For the development phase of the AI-SOCO 2020 track, we submitted the results from our *n*-gram model which achieved an accuracy of 95.416%. The weighted average ensemble result was not submitted as this phase ended prior to its completion. In this phase we achieved first place.

For the evaluation phase, we again submitted the *n*-gram model predictions on the test set alongside the predictions from the weighted average ensemble. The results of our models in this phase are shown in Table 3.

With these results, we again managed to take first place in this phase of the track.

Table 3

Model performance in the evaluation phase.

Model	Accuracy (%)
<i>n</i> -gram Model	95.11
Weighted Average Ensemble	93.82

5. Analysis and Discussion

In this section, we discuss some observations made in two of our most powerful models that subverted the author’s expectations, as well as analysing the models strengths and weaknesses.

5.1. Weighted Average Ensemble

While the weighted average ensemble was the best model overall, its results do highlight some issues. Firstly, there is only a 0.3% accuracy increase over the *n*-gram model alone. From our ensemble analysis, it is clear that whatever other models can classify correctly, the *n*-gram model can typically make the same correct predictions, in addition to making predictions that other models cannot, resulting in the ensemble’s component models providing little benefit. It is also evident that when models share some degree of overlap, they are less effective in the final ensemble. This indicates that the final ensemble would likely benefit from having a more diverse set of constituent models rather than models that extract features from the same raw textual data.

5.2. *n*-gram Model

The *n*-gram model was ultimately our most powerful author classifier, with only the similar SCAP method coming close to the same accuracy. This demonstrates that character and byte-level *n*-grams are likely the best individual representation for the source codes in our dataset. Frantzeskou et al. [17] discuss how their SCAP method has strong performance, even when there is very limited training data per author available. Our results suggest that perhaps the good performance has less to do with the SCAP method itself, and more to do with *n*-gram features being more powerful than other vector representations in these limited feature datasets, hence why our *n*-gram neural network model outperformed other models as well.

One perplexing observation is how the optimised vector representation used in the Naïve Bayes model differed significantly to the optimal vector used in the neural network mode. In addition to a smaller *n* value being used, a binary count performed better than the frequency count values. This is an odd observation, as the initial assumption was that more information was contained in a normalised count, as it detailed how often an author used a specific *n*-gram rather than if they just used it at least once or not. Similarly, TF-IDF of *n*-grams performed worse, even more so than just normalised counts. Combined with our findings in Section 4.1.2, this suggests that commonly occurring words, that would typically be weighted down in TF-IDF, are indeed important in making classification decisions, as opposed to more unique features which would not be represented in the final vectors. This poses a possible new avenue of approach for

Table 4

Ensemble combination accuracies (%) on the Development Dataset. Accuracies along the diagonal reflect the individual model accuracy prior to ensembling.

\times	<i>n</i> -gram	CodeBERT	BoW	AST	Stylometry
<i>n</i> -gram	95.416				
CodeBERT	95.528	86.724			
BoW	95.472	90.564	82.96		
AST	95.524	90.52	87.864	80.052	
Stylometry	95.472	88.944	86.5	85.872	75.376

future work for this model in which a better selection of *n*-grams could be identified to make up our vector representation that could potentially improve accuracy.

5.3. CodeBERT Model

Despite having demonstrated state-of-the-art performance in a number of NL-PL tasks [13], CodeBERT failed to outperform *n*-gram-based models. This is perhaps unsurprising as CodeBERT’s focus is on NL-PL understanding tasks. It may be the case that classifications are being made on the basis of vocabulary used or that the classification token does not encapsulate enough information to distinguish between 1000 authors.

5.4. Ensemble Analysis

An ensemble analysis was carried out to investigate how each individual model contributed to the overall ensemble.

This study was carried out by analysing all ensemble combinations of any two given models used in the final weighted average ensemble. Table 4 shows the accuracies of all these combinations. This table shows that any combination involving *n*-grams will typically be the best performing and that overall, other models do not make a significant impact above the base *n*-gram accuracy. The highest accuracy achieved by an *n*-gram combination was *n*-gram \times CodeBERT models, achieving 95.528% accuracy, a 0.112% increase over *n*-grams alone.

Other combinations, however, can significantly increase the accuracy over the individual components. For example, the AST \times Stylometry ensemble accuracy is improved by 5.82% over the AST model alone and 10.496% better over the stylometry model alone.

An ablation study was also carried out, investigating the effect of removing individual models from the final weighted average ensemble. Table 5 shows the effects of these ablations on the final ensemble accuracy.

Whilst it is unsurprising that the BoW and stylometry models had little impact on the final accuracies given their small weights deduced in Section 4.3, the results displayed by the AST model are curious. Much like the BoW and stylometry models, it had a relatively small impact on final ensemble accuracy, however in the ensemble this model is given a higher weight than any of the other constituent models despite being second worst in terms of raw accuracy.

It is not entirely clear why a model with such a significant weight has so little contribution, but it could be attributed to the diversity of the information captured by the AST vectors. Unlike

Table 5

Ablation of Models from Weighted Average Ensemble on the Development Dataset.

Model Removed	Accuracy (%)	Difference (%)
None	95.715	0
<i>n</i> -gram	92.648	-3.067
CodeBERT	95.568	-0.147
BoW	95.672	-0.043
AST	95.632	-0.083
Stylometry	95.672	-0.043

the other models that extract their features from the raw source code, the AST model features are discovered as the result of syntax analysis carried out by a compiler and represents the actual working structure of the code. In other words, these other models have a higher degree of overlap in the information they contain, since their features all come from the same source, while the AST model has features that no other model has access to, and it is this diversity that could confer the higher weighting.

5.5. Error Analysis

To investigate why our *n*-gram model significantly outperformed all other models and to explore potential avenues of work that could lead to increased accuracy, an error analysis was carried out.

The first step in this process was to identify mistakes consistently made by all models. By intersecting the errors made by each individual model, a set of 822 source codes were identified that were never predicted correctly by any model belonging to 430 different authors. Over half of these authors were one-off misclassifications, leaving only 189 authors that had multiple misclassifications. By analysing the source codes from some of these authors, some consistent characteristics were identified that were shared by these misclassifications. A case study of the author labelled 376 provides a clear example of some of these features.

Source codes from author 376 which are correctly identified in the ensemble model contain the same “#define” pre-processor directives and comment signature at the top of the source codes. Figure 1 however, exhibits three source codes from the same author that were not classified correctly. Notable in these source codes, is that the comment and “#define” pre-processor directives were absent, or in the case of Figure 1(a), altered from the authors typical usage. These features were also missing in all other misclassified source codes from this author. Additionally, the actual code contained is both short, typically being less than 30 lines, and lacking complex variable names, preferring to use single letter identifiers. This is also consistent with a significant proportion of misidentified source codes from other authors and it may be this lack of extractable information that leads to the models consistently getting them wrong.

Next, an *n*-gram specific error analysis was carried out where $n = 6$. To analyse the weaknesses of this model, a number of author source codes were again analysed. Figure 2 contains three such source codes from the author labelled 672.

All of these source codes, along with the remaining correctly identified source codes from this author, share 52 common *n*-grams and, often, a unique comment at the start of the code.

(a)	<pre> 1 #include<bits/stdc++.h> 2 3 #define ff first 4 #define ss second 5 #define maxn 2000006 6 #define pb push_back 7 #define ll long long 8 #define lll __int128 9 #define vll vector<ll> 10 #define mll map<ll,ll> 11 #define MOD 1000000007 12 #define pll pair<ll,ll> 13 #define ull unsigned long long 14 #define f(i,x,n) for(int i=x;i<=n;i++) 15 16 int dx[] = { -1, 0, 1, 0, -1, -1, 1, 1}; 17 int dy[] = {0, 1, 0, -1, -1, 1, 1, -1}; 18 19 using namespace std; 20 21 int main() { 22 ios_base::sync_with_stdio(false); 23 cin.tie(NULL); 24 cout.tie(NULL); 25 26 int t; t = 1; 27 while (t--) { 28 29 ll c, x; cin >> c; 30 x = 2; 31 ll ans = 0; 32 if (c < x or x == 1) ans = c; 33 else if (c == x) ans = 1; 34 else { 35 while (c > 0) { 36 if (c % x == 0) c /= x; 37 else { 38 ans += (c % x); 39 c -= (c % x); 40 } 41 } 42 } 43 cout << ans << endl; 44 } 45 46 return 0; 47 } </pre>
(b)	<pre> 1 #include <bits/stdc++.h> 2 3 using namespace std; 4 5 int main() { 6 int n; 7 cin >> n; 8 for (int i = 0; i < n; i++) { 9 long long k, x; 10 cin >> k >> x; 11 cout << (k - 1) * 9 + x << endl; 12 } 13 } </pre>
(c)	<pre> 1 #include <bits/stdc++.h> 2 using namespace std; 3 4 int t, a, b, c, d; 5 6 int main() { 7 for (cin >> t; t; t--) { 8 cin >> a >> b >> c >> d; 9 cout << b << " " << c << " " << c << endl; 10 } 11 return 0; 12 } </pre>

Figure 1: Misidentified Source Codes from User 376.

Curiously however, the source code Figure 2(c) is misidentified by the n -gram model despite the shared n -grams and comment. Upon investigating the shared features, a notable pattern emerged where the 52 common n -grams exclusively corresponded to five repeating features found in the authors source codes:

```

#include<bits/stdc++.h>
using namespace std;
int main() {
scanf("%
return 0;

```

A significant note is that none of the 52 common n -grams were derived from the unique comment at the start of the source codes. Several conclusions can be made from this: firstly, features

```

(a)
1  /*jai mata di
2  let's rock*/
3  #include <bits/stdc++.h>
4  using namespace std;
5  const int N=200004;
6  long long sum[N];
7  int a[N];
8  int main()
9  {
10     int n,m;
11     scanf("%d %d",&n,&m);
12     int i;
13     for(i=1;i<=n;i++)
14     {
15         scanf("%d",&a[i]);
16     }
17     sort(a+1,a+n+1);
18     long long ans=0;
19     for(i=1;i<=n;i++)
20     {
21         sum[i%N]+=a[i];
22         ans=ans+sum[i%N];
23         printf("%lld ",ans);
24     }
25     printf("\n");
26     return 0;
27 }

(b)
1  /*jai mata di
2  let's rock */
3  #include <bits/stdc++.h>
4  using namespace std;
5  int main()
6  {
7     int t;
8     scanf("%d",&t);
9     while(t--)
10    {
11        int n;
12        scanf("%d",&n);
13        int cnt=0;
14        int i,j;
15        for(i=1;i<10;i++)
16        {
17            long long val=0;
18            for(j=1;j<=10;j++)
19            {
20                val=val*10+i;
21                if(val<=n) cnt++;
22            }
23        }
24        printf("%d\n",cnt);
25    }
26    return 0;
27 }

(c)
1  /*jai mata di
2  let's rock*/
3  #include <bits/stdc++.h>
4
5  using namespace std;
6  const int N=5e5+10;
7
8  char s[N], t[N];
9  int nxt[N];
10 int a[7];
11 int n;
12
13 void build() {
14     nxt[0]=-1;
15     for(int i=1; i<=n; i++) {
16         int j=nxt[i-1];
17         while(j!=-1&& t[j+1]!=t[i]) j=nxt[j];
18         nxt[i]=j+1;
19     }
20 }
21
22 int main() {
23     scanf("%s", s+1);
24     int len=strlen(s+1);
25     for(int i=1; i<=len; i++) {
26         a[s[i]-'0']++;
27     }
28     scanf("%s", t+1);
29     n=strlen(t+1);
30     build();
31     int cur=0;
32     string ans;
33     for(int i=1; i<=len; i++) {
34         int need=t[cur+1]-'0';
35         if (a[need]) {
36             a[need]--;
37             ans+=need+'0';
38             cur++;
39             if (cur==n) cur=nxt[n];
40         }
41         else {
42             break;
43         }
44     }
45     for(int i=1; i<=a[0]; i++) ans+='0';
46     for(int i=1; i<=a[1]; i++) ans+='1';
47     cout<<ans<<endl;
48     return 0;
49 }

```

Figure 2: Source Codes from User 672.

such as unique comments were largely unused in making classifications in this n -gram model. This can largely be explained by the fact that only the top 20,000 most commonly occurring n -grams were used in the model. These unique comments simply did not appear often enough to be included in the model. Secondly, the remaining 52 common n -grams alone were not enough to uniquely identify this author as they were commonly used in source codes by many other authors. This means that while these n -grams may have made some contribution; other

n -grams that were not shared between these source codes were certainly considered in making the classifications.

It was the lack of consideration of the unique comments used by many authors discovered during this analysis that prompted the creation of the BoW model. This model was designed to use every word found in the dataset, so it would be able to make determinations on features not common enough to be considered by the n -gram model. The BoW model does successfully classify the source code shown in Figure 2(c) misidentified by the n -gram model. However, this is not reflected in the final weighted average ensemble, where it is again misclassified, likely due to the BoW only having a small weight and hence small contribution to the final result.

Finally, to identify the strengths of the n -gram model, source codes that only the n -gram model successfully identified were analysed.

Surprisingly, it was found that the n -gram model can identify short source codes, seemingly typical of those that are regularly misidentified by itself and other models, that lack distinct features to the naked eye. Clearly the n -gram model is still able to identify combinations of features that can be attributed to an individual author in these source codes regardless, unlike other models. Unfortunately, due to the black-box nature of neural networks, it is not entirely clear what the specific features are, but this could provide a focus for future research.

6. Conclusion

In this work we found that a simple n -gram based neural network classifier model was capable of outperforming all of the other individual models that we investigated. We also found that our n -gram-based model outperformed our weighted average ensemble, which was unexpected as our weighted average ensemble had performed better on the development set.

With this n -gram based model, we achieved first place in the track with an accuracy of 95.11%.

Further research into this work could focus on other potential ensembling techniques that may better translate to unseen data and deep learning approaches that could better leverage the power of n -grams. Additionally we believe that a deeper analysis into what features are being used as the basis of classification in our n -gram model could provide noteworthy insight into how best to improve the model and why it performs so well compared to other models.

Acknowledgments

Thanks to the NVIDIA Deep Learning Institute for providing GPU resources through Amazon Web Services.

References

- [1] P. W. Oman, C. R. Cook, Programming style authorship analysis, in: Proceedings of the 17th conference on ACM Annual Computer Science Conference, 1989, pp. 320–326.
- [2] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, A. Matyukhina, Code authorship attribution: Methods and challenges, ACM Computing Surveys (CSUR) 52 (2019) 1–36.

- [3] R. C. Lange, S. Mancoridis, Using code metric histograms and genetic algorithms to perform author identification for software forensics, in: Proceedings of the 9th annual conference on Genetic and evolutionary computation, 2007, pp. 2082–2089.
- [4] B. Stein, N. Lipka, P. Prettenhofer, Intrinsic plagiarism analysis, *Language Resources and Evaluation* 45 (2011) 63–82.
- [5] G. Frantzeskou, S. G. MacDonell, E. Stamatatos, Source code authorship analysis for supporting the cybercrime investigation process, in: *Handbook of Research on Computational Forensics, Digital Crime, and Investigation: Methods and Solutions*, IGI Global, 2010, pp. 470–495.
- [6] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, R. Greenstadt, Source code authorship attribution using long short-term memory based networks, in: *European Symposium on Research in Computer Security*, Springer, 2017, pp. 65–82.
- [7] A. Fadel, H. Musleh, I. Tuffaha, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, P. Rosso, Overview of the PAN@FIRE 2020 task on Authorship Identification of SOURCE CODE (AI-SOCO), in: *Proceedings of The 12th meeting of the Forum for Information Retrieval Evaluation (FIRE 2020)*, CEUR Workshop Proceedings, CEUR-WS.org, 2020.
- [8] Codeforces, 2020. URL: <http://codeforces.com/>.
- [9] I. Krsul, E. H. Spafford, Authorship analysis: Identifying the author of a program, *Computers & Security* 16 (1997) 233–257.
- [10] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. E. Chaski, B. S. Howald, Identifying authorship by byte-level n-grams: The source code author profile (scap) method, *International Journal of Digital Evidence* 6 (2007) 1–18.
- [11] J. Kothari, M. Shevertalov, E. Stehle, S. Mancoridis, A probabilistic approach to source code authorship identification, in: *Fourth International Conference on Information Technology (ITNG'07)*, IEEE, 2007, pp. 243–248.
- [12] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, R. Greenstadt, De-anonymizing programmers via code stylometry, in: *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 255–270.
- [13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, *arXiv preprint arXiv:2002.08155* (2020).
- [14] U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning distributed representations of code, *Proceedings of the ACM on Programming Languages* 3 (2019) 1–29.
- [15] U. Alon, S. Brody, O. Levy, E. Yahav, code2seq: Generating sequences from structured representations of code, *arXiv preprint arXiv:1808.01400* (2018).
- [16] V. Kovalenko, E. Bogomolov, T. Bryksin, A. Bacchelli, Pathminer: a library for mining of path-based representations of code, in: *Proceedings of the 16th International Conference on Mining Software Repositories*, IEEE Press, 2019, pp. 13–17.
- [17] G. Frantzeskou, E. Stamatatos, S. Gritzalis, S. Katsikas, Effective identification of source code authors using byte-level information, in: *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 893–896.
- [18] M. F. Tennyson, F. J. Mitropoulos, Choosing a profile length in the scap method of source code authorship attribution, in: *IEEE SOUTHEASTCON 2014*, IEEE, 2014, pp. 1–6.
- [19] W. Wisse, C. Veenman, Scripting dna: Identifying the javascript programmer, *Digital*

Investigation 15 (2015) 61–71.

- [20] H. Ding, M. H. Samadzadeh, Extraction of java program fingerprints for software authorship identification, *Journal of Systems and Software* 72 (2004) 49–57.
- [21] C. Zhang, S. Wang, J. Wu, Z. Niu, Authorship identification of source codes, in: *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*, Springer, 2017, pp. 282–296.
- [22] F. Chollet, *Deep Learning with Python*, Manning, 2017.
- [23] M. J. Powell, An efficient method for finding the minimum of a function of several variables without calculating derivatives, *The computer journal* 7 (1964) 155–162.
- [24] R. Storn, K. Price, Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces, *Journal of global optimization* 11 (1997) 341–359.