Multi-Channel Convolutional Neural Network for Source-Code Authorship prediction and Error Detection

Saurabh Chavhan

Department of Computer Science
Syracuse University Syracuse,
NY 13210
sachavha@syr.edu

Abstract— There are several successful studies on source code authorship prediction. However, they often disregard some important data points. The use of many tokens in the source code is overlooked while generating the signature for any author. Taking this into consideration we propose a convolutional neural network architecture and we will compare it with several other techniques[1]. Furthermore, we will use the output of this neural network and will associate the author with a certain set of errors. This way, given a source code we will be able to predict the author and check if he/she has made an error.

I. Introduction

Authorship prediction and stylometric analysis were originally introduced for text documents and literature. It has gained popularity as stylometric analysis can identify the author and their signature and hence can uniquely associate documents, text or literature with their authors. We focus on a very specific field of computer science for authorship prediction which is identifying the authors of the source code. Source code can be written in any programming language and for our implementation, we need the source code in the same programming language for all the authors. Previous studies have proposed many solutions involving feature extraction using the N-gram approach which turns out to yield the best results. This N-gram dataset is fed to neural network architecture (LSTM or convolutional) to predict the author. For the LSTM-based network, training is slow considering we will require quite a large dataset[1]. Whereas 1-Dimensional convolutional neural networks are used with a kernel size of 2 in many other cases. Despite the satisfactory result of previous studies what they lack is the tokens used for feature selection or the N-gram selection. In most of the cases, special characters are often ignored, space and tab chars are removed and trailing white spaces and new line characters are often truncated. In our implementation, we will consider these tokens. Another aspect of our study is to identify the errors made by authors in a programming language. Studies in this area are mostly on syntactical errors[11] and some on the single token syntax errors, their major goal is "Given a source code file with a syntax error, how can one accurately pinpoint its location and produce a single token suggestion that will fix it?"[12], Feed-forward models are used for mapping input and output patterns but it hardly can preserve the association between the elements of each input pattern[13], some others use the recurrent neural network is often used for this purposes.

We start by introducing our token creation approach followed by an in-depth analysis of our multichannel neural network. Later in Section VI, we will define our approach for error detection. Finally, we will compare our results with the metrics of state of the art and mentioned in [1].

II. Token Generation

N-grams of words grasp the relevant linguistic information in stylometric analysis, short words often belong to the group of function words[2]. Usually while generating tokens for a dataset we often ignore some stop words, punctuations, and other keywords the are essential in programming languages. For example, a 'full stop' or a 'comma' character can be insignificant for normal text file but they are important in any many programming languages.

In our implementation, we consider all the characters that are present in source code. An argument can be made that given a particular programming language there will be many similar tokens, also some Ide`s have certain configuration for many languages. We can answer these questions by taking positions of the token into account and giving equal weight to all the tokens. We take 150 top tokens and all tokens are given equal priority therefore even though there are similarities at the same position such cases will be less.

Another major aspect that we touch is the indentation and new line character used by developers and programmers. Many programmers often prefer to start new statement on a new line that produces a new line character at the end of each statement, similarly for languages like C++ and C semi-colon characters are the end of the statement, thus token generated by semi-colon character is important for our implementation of stylometric analysis.

Representation for data (source code)

```
int main() {
  while(i < 0) {
      i++;
    }
}
```

Figure 1.

A. *Tokens generated by the tokenizer*

1	2	1	3	5	6	4	0	0
7	8	5	14	13	16	6	4	0
7	7	14	17	17	10	0	0	0
7	21	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0

Figure 2.

From figure 1, consider there is a space char before the first statement it has a token in the token matrix shown in figure 2.

Sample tokens generated by our tokenizer:

{' ': {'value': '1'}, '\t': {'value': '2'}, '\n': {'value': '3'}, ';': {'value': '4'}, '(': {'value': '5'}, ')': {'value': '6'}, '[': {'value': '7'}, ']': {'value': '8'}, '=': {'value': '9'}, ',': {'value': '10'}, 'i': {'value': '11'}, '<': {'value': '12'}, 'int': {'value': '13'}, '>': {'value': '14'}, '0': {'value': '15'}, '.': {'value': '16'}, '1': {'value': '17'}, '#': {'value': '18'}, 'j': {'value': '19'}, 'x': {'value': '20'}, '"': {'value': '21'}, 'if': {'value': '22'}, 'n': {'value': '23'}, 'for': {'value': '24'}, 'define': {'value': '25'}, '"': {'value': '26'}, 'a': {'value': '27'}, 'include': {'value': '28'}, '/': {'value': '29'}, 'd': {'value': '30'}, 'rep': {'value': '31'}, 'v': {'value': '32'}, 'b': {'value': '33'}, 's': {'value': '34'}, 'typedef': {'value': '35'}, 'vector': {'value': '36'}, '2': {'value': '37'}, 'N': {'value': '38'}, 'll': {'value': '39'}, 'cin': {'value': '40'}, '!': {'value': '41'}, 'return': {'value': '42'}, ':': {'value': '43'}, 'y': {'value': '44'}, 'fi': {'value': '45'}, 'k': {'value': '46'}, 'g': {'value': '47'}, 'se': {'value': '48'}, 'u': {'value': '49'}, 'w': {'value': '50'}

III. Dataset

In this study, we are working with code samples of 4 authors taken from Google code jam competition. A similar dataset has been used in many other studies[1] with which we will compare our results. The dataset contains 15-20 samples for each author in the C++ programming language.

IV. Convolutional Layer

As we can see that we get a 2-dimensional matrix after tokenization we concluded that a convolutional neural network would work the best. The problem can be visualized as detecting the signature of an image. For example, consider images of 2 different persons with the same background, we can say that the signature generated is different because of the 2 persons but not because of the background.

We use a 2-Dimensional Convolutional neural network for this purpose. We shall discuss the kernel size in the next subsection. Below we have shown how a convolutional layer with kernel size 3x3 will operate on the data matrix obtained in section II.

1	2	1	3	5	6	4	0	0	0	0
7	8	5	14	13	16	6	4	0	0	0
7	7	14	17	17	10	0	0	0	0	0
7	21	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0

Figure 3.

1	2	1	3	5	6	4	0	0	0	0
7	8	5	14	13	16	6	4	0	0	0
7	7	14	17	17	10	0	0	0	0	0
7	21	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0

Figure 4.

V. MULTI-LAYER CONVOLUTIONAL LAYER

Till now it seems like we have solved the problem but the accuracy that we get is around 87% (with kernel size 2x2 of the convolutional layer). Obviously, because the window size is small we are not taking enough tokens together for analysis. But this does not mean we can ignore smaller parts of the matrix.

Therefore we introduce 2 strategies here, one layer with a smaller scope which concentrates on the limited part of the dataset to understand the relationship between the tokens which are close enough to each other and other with large window size to retain the relationship between the tokens that are further away from each other.

These 2 strategies can be implemented using a variable kernel size of 2x2, 3x3 and 5x5. Therefore in our implementation, we have used the above-mentioned kernel sizes in 3 different channels. These 3 channels merge into one channel to generate the output which is the author label. We observed that with one channel of kernel size 2x2 we get an accuracy of around 87% whereas if use another channel with a kernel size of 3x3 accuracy increases to 90%. Adding one more channel of 5x5 kernel size increases the accuracy to 90.98%.

On top of this, we use 'Nadam' optimizer for our network as it provides momentum as an in-build functionality. The above-mentioned accuracies are observed with optimizer 'Adam'. With 'Nadam' our overall accuracy jumps to 94%.

Network architecture is shown below in Figure 5 below followed by the results.

VI. ERROR DETECTION

To address this problem we use the same architecture as above. But we change our input dataset. We add author bits with each input matrix to associate the author with each line of code. Labels, in this case, would the 0 or 1 (error or no-error). The idea behind this is to associate each author with the error they make. We divide our dataset into testing and training with the split of 1:9.

Network architecture is shown below in Fig 6 below followed by the results.

VII. LIMITATIONS

After several rounds of testing, it was found that if we increase the number of authors and their data samples our network starts performing poorly. The optimal number of authors would be a value of less than 10. For future updates, we can try building a deeper channel for our network as the variables increase drastically when we increase the number of authors and their source code sample.

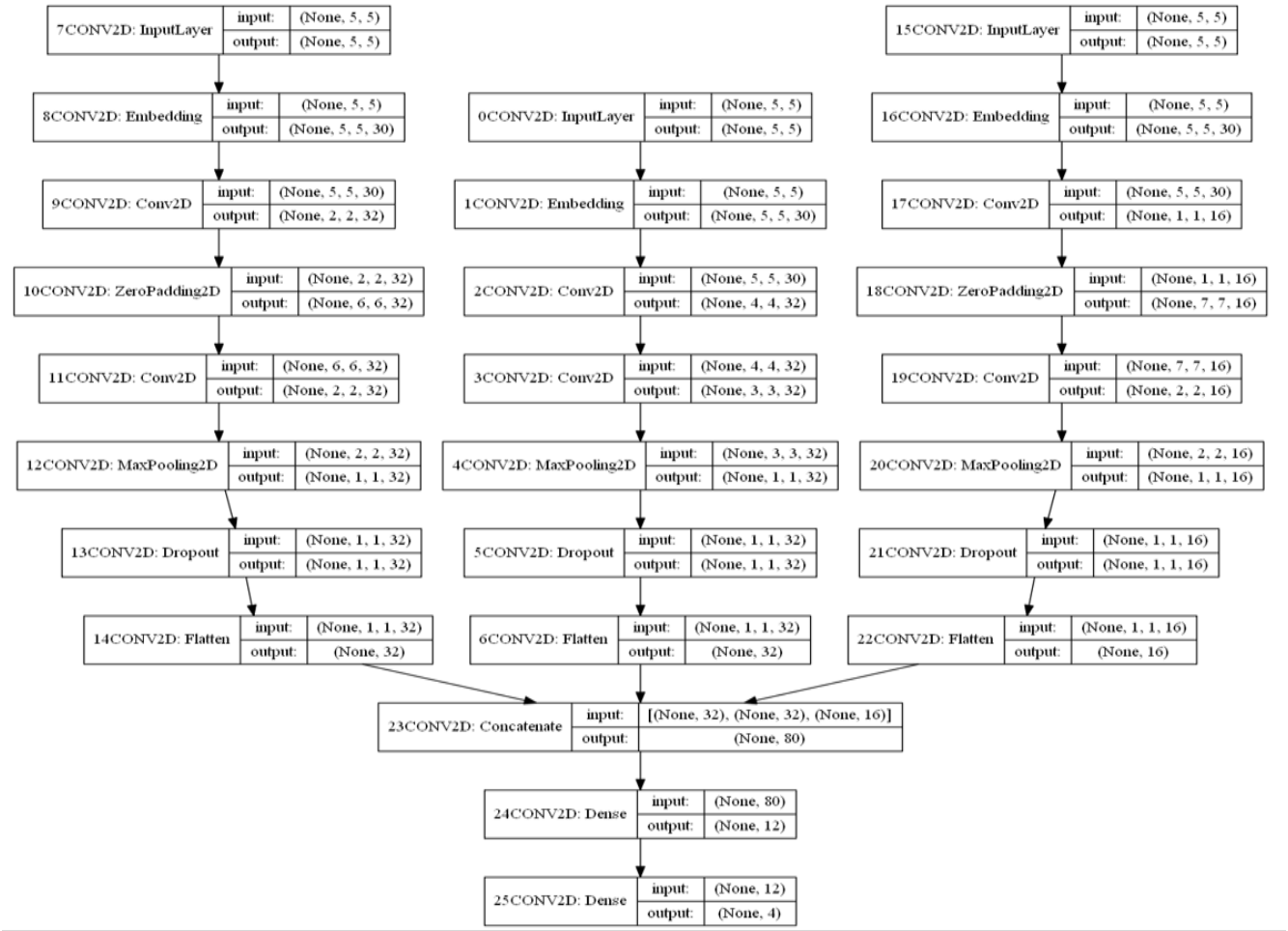


Figure 5.

Results when compared to results with[1].

Method	Output Dim.	Metric	Driving Force	Acc (%)	Similarity Threshold	PPV (%)	NPV(%)
Random guess	NA	NA	NA	1.23	NA	0.5	0.5
PCA	5	Euclidean	Maximum variance	11.11	0.2	62.65	59.53
Siamese Network	5	Euclidean	Contrastive Loss(8)	60.49	1.7	86.72	84.05
SRFNN	81/72	Cosine	Cross-Entropy Loss	95.06	0.35	87.34	86.42
Multi-channel Conv-2D	4	Cross-Entropy	Cross-Entropy	94.3	NA	84	96

Training: Specificity: 97% and sensitivity: 87%
Testing Accuracy: 90.04%, PPV: 80% and NPV: 94%

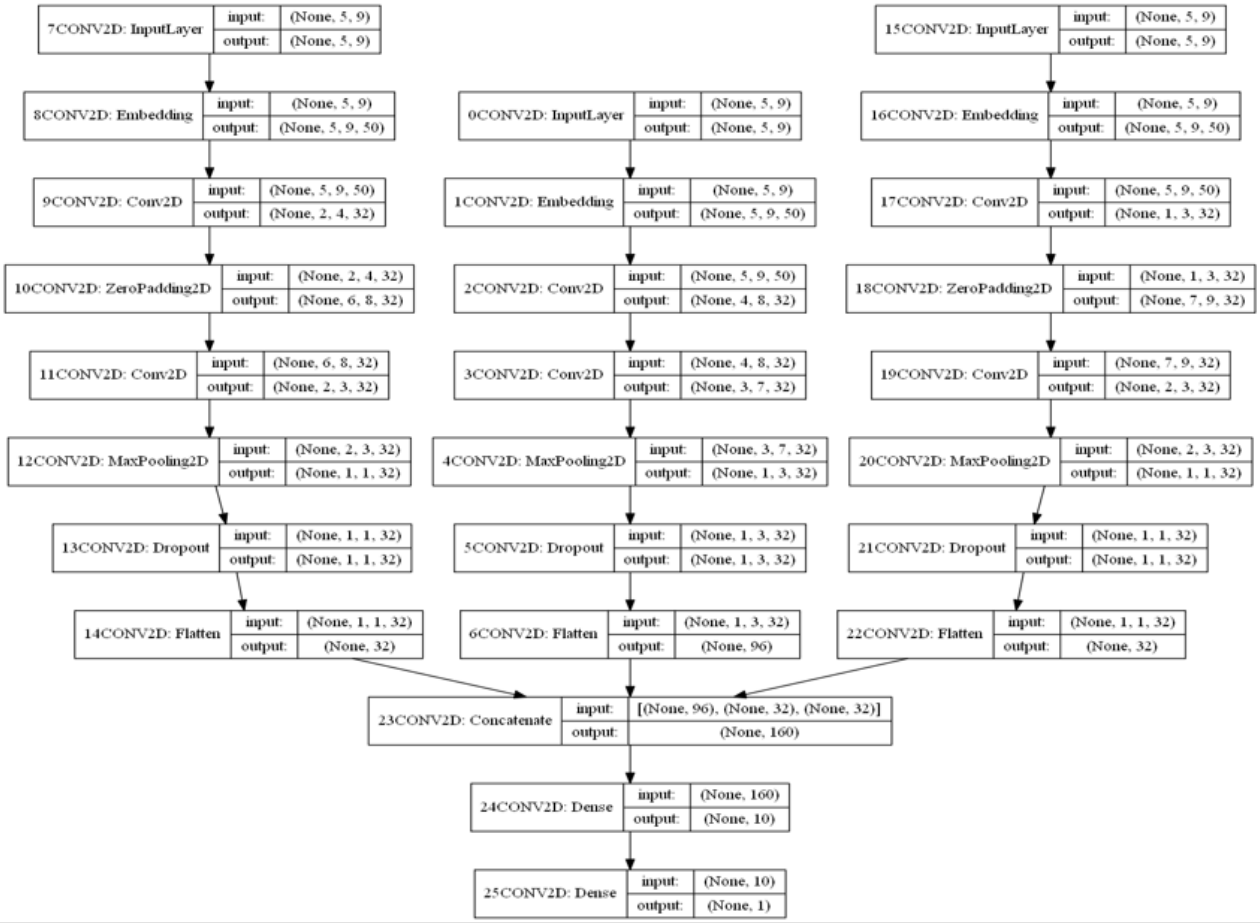
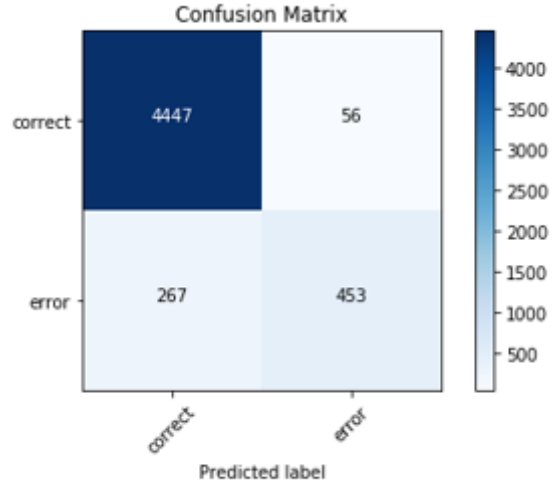


Figure 6.

Training : sensitivity: 60% specificity: 98.99% acc: 93.17%



Testing: accuracy 88.68%

REFERENCES

[1] Pegah Hozhabrierdi, Dunai Fuentes Hitos, Chilukuri K. Mohan, Nested Bigrams and Stylometric Embedding for Source Code Authorship Attribution

[2] Łukasz Gągała, Authorship attribution with neural networks and multiple features Notebook for PAN at CLEF 2018.

[3] Dylan Rhodes, Author Attribution with CNN's.

[4] Liuyu Zhou, Huafei Wang, News Authorship Identification with Deep Learning.

[5] Elisa Ferracane , Su Wang and Raymond J. Mooney, Leveraging Discourse Information Effectively for Authorship Attribution.

[6] Bander Alsulami , Edwin Dauber , Richard Harang , Spiros Mancoridis , and Rachel Greenstadt, Source Code Authorship Attribution using Long Short-Term Memory Based Networks

[7] Steven Burrows and Seyed MM Tahaghoghi., Source Code Authorship Attribution using n-grams, Australia, RMIT University, Citeseer, 2007

[8] Pegah Hozhabrierdi, Dunai Fuentes Hitos, and Chilukuri K Mohan. Python source code deanonymization using nested bigrams. In 2018 IEEE International Conference on Data Mining Workshops (ICDMW), pages 23–28. IEEE, 2018.

[9] Ivan Krsul and Eugene H Spafford. Authorship analysis: Identifying the author of a program. Computers & Security, 16(3):233–257, 1997.

[10] Brian N Pellin. Using classification techniques to determine source code authorship. White Paper: Department of Computer Science, University of Wisconsin, 2000.

[11] Lingchen Huang, Huazi Zhang, Rong Li, Yiqun Ge, Jun Wang, AI Coding: Learning to Construct Error Correction Codes.

[12] Eddie Antonio Santos , Joshua Charles Campbell , Abram Hindle , and Jose Nelson Amaral, Finding and correcting syntax errors using recurrent neural networks

[13] Hossam Abdelbaki, Erol Gelenbe, Said E. El-Khamy, Random Neural Network Decoder for Error Correcting Codes