Author identification with Neural Networks
Író azonosítás Neurális Hálókkal

Budapest University of Technology and Economics

Plagizálók Team

Ragács Attila
Urbanics András
Pfeifer Dániel

Abstract

During the task of Author Identification we predict the most probable author from a group of possible authors for academic articles, news, books or emails. In this project we worked with the Spooky Authors dataset from Kaggle where we had to find the original writer of sentence level entries. First we did some data manipulation to have a more usable dataset (stopword removal, turning all letters lowercase). To tackle this problem we established statistical baselines, then applied Fully Connected and Convolutional Neural Networks on the data, also making use of the pretrained FastText word embeddings. In the end we managed to reach 76% accuracy with 3 possible authors.

Kivonat

Az Író Azonosítási feladat alatt a legvalószínűbb írót jósoljuk meg egy néhány lehetséges akadémiai cikk, hír, könyv vagy e-mail írója közül. Ebben a projektmunkában a Kaggle-ről származó Spooky Authors adathalmazzal dolgoztunk, amelyben mondat-szintű bejegyzéseknek kellett kitalálnunk az eredeti íróját. Először némi adatkezelést hajtottunk végre használhatóbb adathalmaz előállítása érdekében (stopword eltávolítás, kisbetűssé alakítás). A probléma megoldásáért statisztikai alapokat fektettünk le, majd Teljesen Összekötött és Konvolúciós Neurális Hálózatokat alkalmaztunk az adaton, előre tanított FastText szó-beágyazást is alkalmazva. Végül 76%-os pontosságot sikerült elérni 3 lehetséges író esetén.

1 Introduction

Authorship Identification determines the probability of a text, programming code or any type of writing belonging to a particular author by taking certain describing attributes into account. It can be used for cybercriminal analysis, to detect plagiarism or to identify spam emails just to mention a few applications[1].

There is a rich literature on using stylistic features for this task [2]. Although with the use of neural networks we can avoid the difficult problem of feature engineering (choosing the best set of features). The d Several architectures have been tried before, these include Recurrent Neural Networks or Convolutional Neural Networks with varying degrees of success [3][4].

In this paper, in Section 2 we first give a short description of the dataset used for our project and introduce the basic statistical methods that served as our baseline. Then in Section 3 we showcase the different neural network systems we tried. Finally a short summary and future possibilies in Section 4.

2 Data and simple methods

2.1 The data

Our data is from the Kaggle competition Spooky Author Identification. The training set contains sentences from three authors: Edgar Allan Poe, HP Lovecraft and Mary Shelley and the task is to correctly classify the entries. They also gave a test set without labels for evaluation on Kaggle, but since we used different scoring measures we only used the training set. From that we created a smaller training set and a validation set for evaluation.

2.2 Data preparation

One of the first things we looked at was the distribution of words in the data and to no surprise we found the most frequent words to be these very common words also known as stopwords. These words tend to appear in all entries similarly, so to combat this, we removed them from the data, although in some cases it worked better to leave them in and only remove punctuation. Afterwards we continued to work with sentences that did not make too much sense to us, but they were more distinct to the neural networks which helped during training.

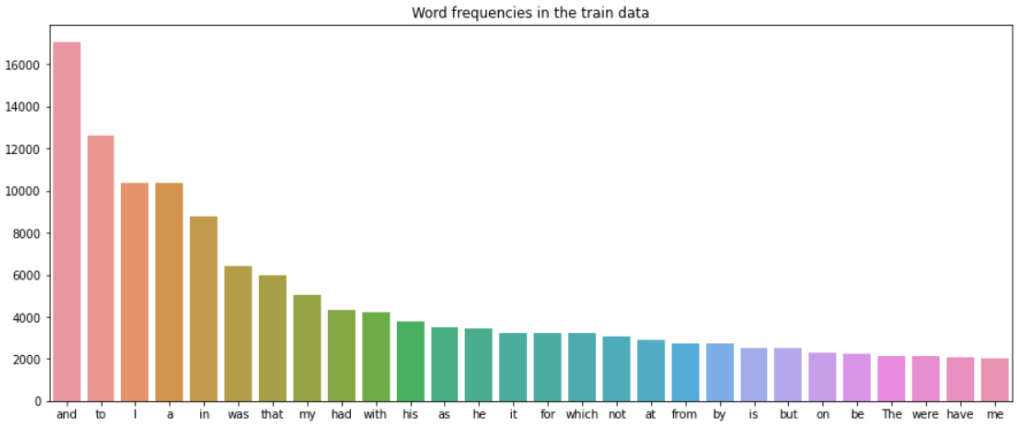


Figure 1: The most frequent words and their frequencies

Below on Figure 2 we can take a look at the remaining words (the bigger the more frequent) from entries written by Edgar Allen Poe and Mary Shelley.

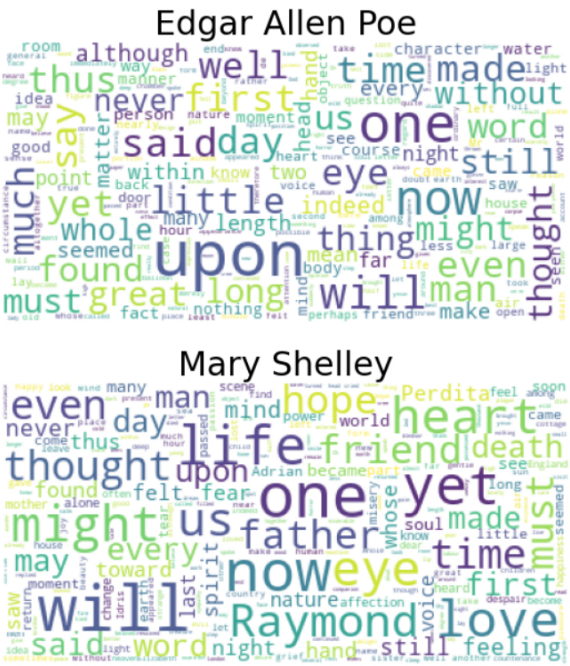


Figure 2: The most frequent words from Edgar Allen Poe and Mary Shelley in a wordcloud

2.3 Statistical methods

First of all, it is important to note, that predicting 0 for all given sentences still gives an accuracy of 40%, since 40% of the train data was written by Edgar Allan Poe. We clearly need to do better than this.

As our first idea we created a couple simple models to serve as a baseline. Even the accuracy that they gave needed to be surpassed by other, more complicated models. These models were based on simple statistical methods. The most basic idea one could have is a word appearance rate - some authors use some words more frequently, so that could be a deciding factor of who we choose. However, with slightly more thought, we can get even better estimators, based on which words appear next to each other. We introduce the following simple idea:

$$\begin{aligned} \text{Trigram} &= \text{A continuous sequence of 3 words} \\ T(A) &= \{\text{Trigrams in text } A\} \end{aligned}$$

With this, we can define the number of common trigrams in text A and B. Specifically, how many of B's trigrams appear in A as well. This is what we call the "Containment Measure":

$$\text{CM}(A, B) = \frac{|T(A) \cap T(B)|}{|T(A)|}$$

Clearly, if this measure is large, then text A and text B are similar in some sense. If this measure is 1, then every trigram contained in B also appears in A, so B is an extract of A.

Another measure we used is called the "Longest Common Subsequence", which its name explains. This refers to sequences of words commonly found in both texts, and is also normalized by the number of words of the shorter text:

$$\text{LCS}(A, B) = \frac{\max_{n \in \mathbf{N}}(w_1 w_2 \dots w_n \in A \cap B)}{\min(|A|, |B|)}$$

If two texts have an LCS score of 1, then the shorter text is entirely contained in the longer one.

2.4 Baseline predictions with CM and LCS scores

We've used the above measures for all pairs of sentences in our training data (70% of the actual train data given on Kaggle), and validated them on the remaining 30% of our training data with known labels. If a given sentence is more similar to ones written by a given author (according to one of our measures above), then we predicted that it was written by that author. We denote the set of sentences written by author a by $S(a)$. Then our prediction for sentence s is the following:

$$\text{Prediction}_{CM}(s) = \max_{a \in \{EAP,HPL,MWS\}} CM(s, S(a))$$
$$\text{Prediction}_{LCS}(s) = \max_{a \in \{EAP,HPL,MWS\}} LCS(s, S(a))$$

Running these for all s sentences in our validation set and reflecting the result against the actual labels we obtained the accuracies of

Accuracy scores	CM	LCS
with stopwords	45.71%	40.28%
without stopwords	63.55%	43.00%

Our best estimator gave an accuracy of 63.55%, which means we need to obtain a better estimator with the upcoming, Neural Network-based models.

3 Methods using Neural Networks

3.1 Word2vec and Sentence2vec

For our methods we used Neural Networks, namely the Keras infrastructure in Python [5]. Therefore, we needed to encode our word/sentence data into vectors, so they could be appropriate inputs for the networks.

To perform this issue, we loaded existing methods. The first solution used only our data. With this method we have automatically detected the common phrases - also known as word bigram collocations - from a stream of sentences. There was another solution, when we downloaded pre-trained word vectors with large datasets to obtain vector representations for words, here we used the FastText and GloVe wordembeddings [6][7]. In each case a 300-long vector was constructed from the words. With the obtained 300 dimensional vectors, it became possible to form suitable inputs for Neural Network.

One possible use of these is to calculate the average of the vectors belonging to the words of each sentence. In this way we came to a Sentence2vec method. Thus, the 300-long real vector inputs of the network are a given sentence. There are other ways to use Word2vec inputs, which will be described later, before using them.

3.1.1 Basic Word2vec model without Neural Network

The method we described earlier and with which we have automatically detected the common phrases, will be able to build a Word2vec network based on sentences for only that particular author. With this procedure, we constructed 3 different word2vec representations from the train sentences of the 3 authors.

In the testing, we used the words 'bigram' of the sentence to be examined to see how far apart the two words are in the word2vec network belonging to each author, how similar they are, for example how characteristic the author is. These values were averaged at the end of the sentence per author. And the prediction was the author with the highest value.

The method performed a little better, giving an accuracy of 66%. This performance could be improved by increasing the train set, since the model has one flaw: the constructed word2vec

representation does not include all the words, so there were bigrams in the testing that were incomprehensible in a given network. In such cases, the similarity of the words was 0. There is another flaw: train sets are not the same size, which also affects performance.

3.2 Simple dense Network

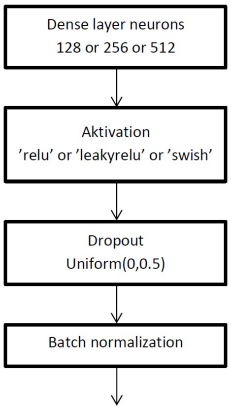
With this model our intention was to try one of the most simple neural systems on the data before moving on to more complicated ones. Interestingly, it turned out in the end that this method produced close to the best accuracy during the project. The model is structured the following way: the model receives the 300 dimensional vectors as its input, then there are 3 300 node dense layers with dropout between the first two. Finally a dense layer with 3 nodes and softmax activation gives the prediction.
What we could change before optimizing the hyperparameters, was the input. The 300 dimensional vectors ended up being the mean vectors of the words in a sentence described earlier in Section 3.1. However we also tried feeding the fully connected network the minimum and maximum vectors, where these were taken componentwise and also the concatenation of these two resulting in a 600 dimensional vector. In the end, the models underperformed the mean-vector method with these inputs. During the training of this model the highest validation accuracy we achieved was 75.3%. Also at the final stage, after the training stopped because of the early stopping condition, the built-in Keras evaluation function returned 75.1% accuracy.

3.2.1 Hyperparameter optimalization

Hyperparameter optimization was performed on the previous Simple dense Model. The first layer of the network could consist of 128, 256, 512 neurons, followed by a uniform (0,0.5) distributed Dropout. This was followed by a Batch normalization and these were repeated twice. Finally, a Dense layer of 3 neurons was added to the model with a softmax activation function.

The other activation functions of the network could be 'relu', 'leakyrelu' and 'swish', and the optimizer could choose between 'rmsprop', 'adam' and 'sgd'. The 300-dimensional input vectors could be 64, 128, or 256 batch sizes. This optimization was performed on 300-long vectors, generated two different ways, up to 100 epochs (Earlystopping with patience parameter 3 was used) with a maximum evaluation parameter of 100. This resulted in the best accuracy of 76.40%.

Figure 3: Hyperparameters and their possible values



3.3 Convolutional Network

3.3.1 Sentence encoding for convolution

In this section, recognizing the fact that the adjacency of words holds immense information, we use a Convolutional Neural Network with a sliding window that reads each input sentence

a couple words at a time. As previously mentioned, each of these words have been encoded as 300-long real vectors, so a given sentence of length n can be interpreted as n adjacent 300 dimensional vectors, or an input matrix with shape $(n, 300)$. However, convolution will only accept an input matrix with equal row-lengths, and our data obviously contains sentences with differing lengths.

We'll propose two solutions for this issue: A cutoff of 100 words for each sentence (longer sentences will be left out, but that only took up about 0.8% of our data), and repeating the sentence if it's shorter than 100 words.

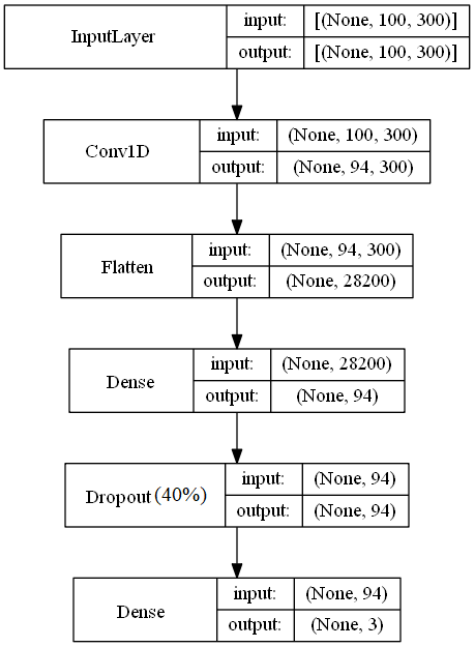
To be exact, we've also included an empty character at the end of the sentence to mark the beginning of the new one, so a short sentence of let's say 7 words will first be appended by an empty word that maps to the **0**-vector in the word2vec encoding, then the resulting 8-long word will be repeated 12 times, since $100 \div 8 = 12.5$, and the remaining 4 spaces will be filled out by the first part of the sentence.

This way we obtain an encoded input sentence with shape $(100, 300)$. There were a total of $19,000$ input sentences so our total input shape in this model is $(19000, 100, 300)$.

3.3.2 One-type convolution window model

The most basic Convolutional Network we can build is one with just a single Convolution Layer, with a single, non-changing kernel size, which was chosen to be 7. That means, that we forward the information of 7 adjacent words into nodes in the second layer. Then on the third layer, to obtain information from all parts of the Network, we use a Dense (a.k.a. Fully Connected) Layer with an output of 3 nodes and *softmax* activation that maps into the actual authors of the input sentences.

Figure 4: Layout of the One-type convolution window model



Due to technicalities, a flattening layer had to be included between the 1D-Convolution and Dense layers, and finally, a 40% Dropout rate has been used to avoid overfitting. This parameter has been chosen after some testing.

This model gave an accuracy of up to 74%. Multiple other options have been tried out, namely:

- Several Convolutional Layers one after another: This model wasn't actually capable of learning, it constantly hovered around $34 - 35\%$ accuracy, which is only slightly

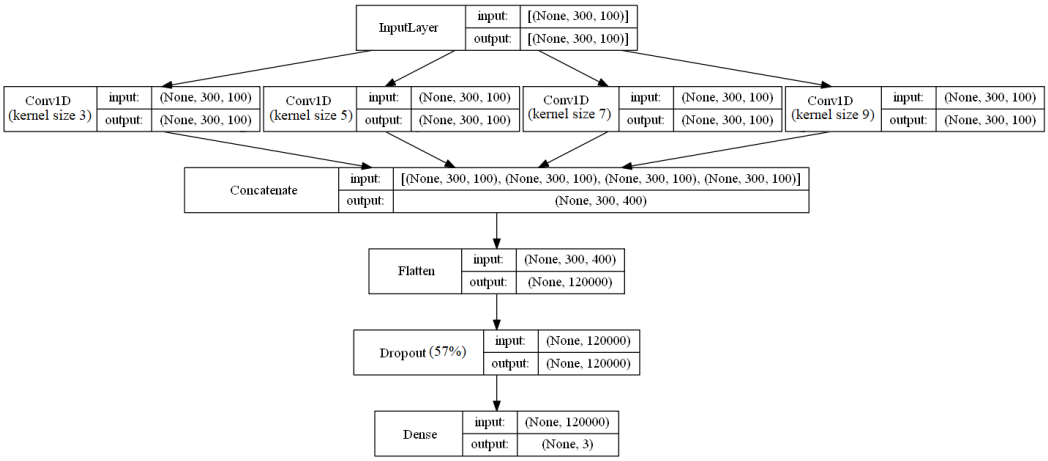
better than random, however predicting 0 for each sentence already gave a 40% accuracy (there were slightly more sentences by Edgar Allan Poe than others).

- A Max Pooling Layer after the Convolutional Layer. This simply made the accuracy worse. It seems keeping all the data from the Convolution is the best option.
- Multiple Dense Layers at the end. This also didn't help the accuracy. It looks like keeping the model simpler helps.

3.3.3 Multiple-type convolution window model

One way we can increase the complexity of the previous model is by including multiple Convolutional Layers next to each other, just like in [8] with differing Convolutional Window sizes. This can be seen on Figure 5.

Figure 5: Layout of the Multiple-type convolution window model



This model was capable of learning, however, it was by far the slowest model, so optimizing it would take a long time. Max Pooling on each separate Convolutional Layer could be used, which did increase the convergence rate and reduced the runtime, because even around 40 epochs without Max Pooling took around 11 hours, with various $(2\times -4\times)$ Max Pooling the runtime went down to about 8 hours. These numbers have been gotten by running the model on a 2.8 GHz Dual Core Processor, not a GPU, which could have significantly decreased the runtime, however, we didn't have access to that.

The best accuracy this model gave was also around 74%, but it's possible that with more optimized parameters and higher computational capacity, we could get an even better result.

4 Summary

After learning about the task of Author Identification we have developed statistical, but mainly neural network models for this problem. We have reached 76% accuracy on our data from Kaggle using Fully Connected and Convolutional Neural Networks. For a future project one might be interested in training the model with more data and more possible authors. Also there are plenty of other networks that were not used in our work that might result in better scores [9][10].

References

[1] LZ Wang. News authorship identification with deep learning, 2017.

[2] Efstathios Stamatatos. A survey of modern authorship attribution methods. *Journal of the American Society for information Science and Technology*, 60(3):538–556, 2009.

[3] Dylan Rhodes. Author attribution with cnns. *Avaiable online: https://www. semanticscholar. org/paper/Author-Attribution-with-Cnn-s-Rhodes/0a904f9d6b47dfc574f681f4d3b41bd840871b6f/pdf (accessed on 22 August 2016)*, 2015.

[4] Douglas Bagnall. Author identification using multi-headed recurrent neural networks. *arXiv preprint arXiv:1506.04891*, 2015.

[5] François Chollet et al. Keras. `https://keras.io`, 2015.

[6] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[7] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[8] Mohammed Abuhamad, Ji-su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. Code authorship identification using convolutional neural networks. *Future Generation Computer Systems*, 95:104–115, 2019.

[9] Zhenzhou Tian, Qing Wang, Cong Gao, Lingwei Chen, and Dinghao Wu. Plagiarism detection of multi-threaded programs via siamese neural networks. *IEEE Access*, 8:160802–160814, 2020.

[10] Magnus Stavngaard, August Sørensen, Stephan Lorenzen, Niklas Hjuler, and Stephen Alstrup. Detecting ghostwriters in high schools. *arXiv preprint arXiv:1906.01635*, 2019.