Thomas Corcoran

Capstone Project

Machine Learning Engineer
Nanodegree

Convolution Neural Networks on Wikipedia Author Detection

This project focuses on authorship analysis from Wikipedia talk pages. As everyone knows Wikipedia is the online encyclopedia unique in that articles can be authored by a virtually unlimited number of contributors. Determining authorship of these articles is non-trivial, as other researches have shown[1]. This is due partially to the fact that articles have multiple authors, so it's not always clear who can be labeled the true author of any particular sentence without using some heuristics. Unlike the article pages, the Wikipedia talk pages contain a wealth of texts (individual posts, like a message board) clearly written by single authors. Most talk pages are empty, but some contain more text than the original article. Most posts in the talk pages are signed by the author, and it is thus relatively simple to extract the author from the text of the post. In this report I'll show how I built a classifier to determine authorship of Wikipedia talk page posts. In future work, the information from this classifier might be useful in a the more complex case of article authorship analysis.

The classifier will be constructed using a convolution neural network implemented with Google's TensorFlow. Even though convolutional neural networks (CNNs) were originally conceived with computer vision in mind, they typically perform fantastically in text classification problems as well[2]. CNNs differ from traditional neural networks in that they parse each input (a talk page post in this case) in chunks, rather than all at once. This turns out to be especially useful when we know the data has a local structure e.g. images and natural language. The CNN will directly be able to make predictions about authorship once full trained on a subset of the talk page post data. Multiple models will be trained using the same training data using different model hyperparameters. In the end I'll have several fully trained models, the best of which will be determined using several metrics.

In a classification setting, we must worry about precision (the ability of the classifier not to label a post wrongly) and recall (the ability of the classifier to label a post in a particular class). For example if the classifier guess that all posts were authored by *JoeWikipedia* the recall for *JoeWikipedia* would be 100%, while the precision would be much lower (dependent on the number of other authors). Precision and recall are more intuitive in the binary class case, but can be made to work in a muticlass case, such as this when there are a variable number of authors. In our case, we'd like to maximize both precision and recall, rather than one or the other. Luckily there is a metric that does just this, the harmonic mean of precision and recall also known as the F1 score. The F1 score is defined as:
$F1 = 2 * (precision * recall) / (precision + recall)$. The difficulty in this that that the precision and recall are defined only for binary classification. To generalize the F1 score to this mutilabel case, it's necessary to take some kind of average across all labels. The way this will be done is by taking a one against the rest approach for each label, compute the precision and recall treating only one label as positive and the rest negative. The F1 scores for each of the “1 against the rest” rounds are then combined in a weighted average where the weight of the class is informed by the number of true

1 https://cs224d.stanford.edu/reports/MackeStephen.pdf
2 http://emnlp2014.org/papers/pdf/EMNLP2014181.pdf

positive examples for that class. I'll also use the accuracy score of the model on a subset of validation data, but I will not use accuracy alone to determine the 'best' model. This is because accuracy alone can be subtly misleading due to the accuracy paradox in some edge cases.

Analysis

Data Exploration

Using the latest Wikipedia metadata dump, I was able to collect nearly 2.5 GB of texts from 1547773 authors (including bots). I've decided to limit my experiments to a subset of authors due the limit of my computational resources (2.5 GB is a lot of text to process). For this experiment I've chosen a subset of the authors by the total number of characters in all of their posts. Because bot posters are uninteresting in terms of authorship analysis, any author name with the substring 'bot' was removed. Finally the top 10 most prolific authors were chosen, and I've determined by hand that none of them are automated bots. From these 10 authors there are a combined total of 18,687 posts. Below is a chart of the number of posts per author:

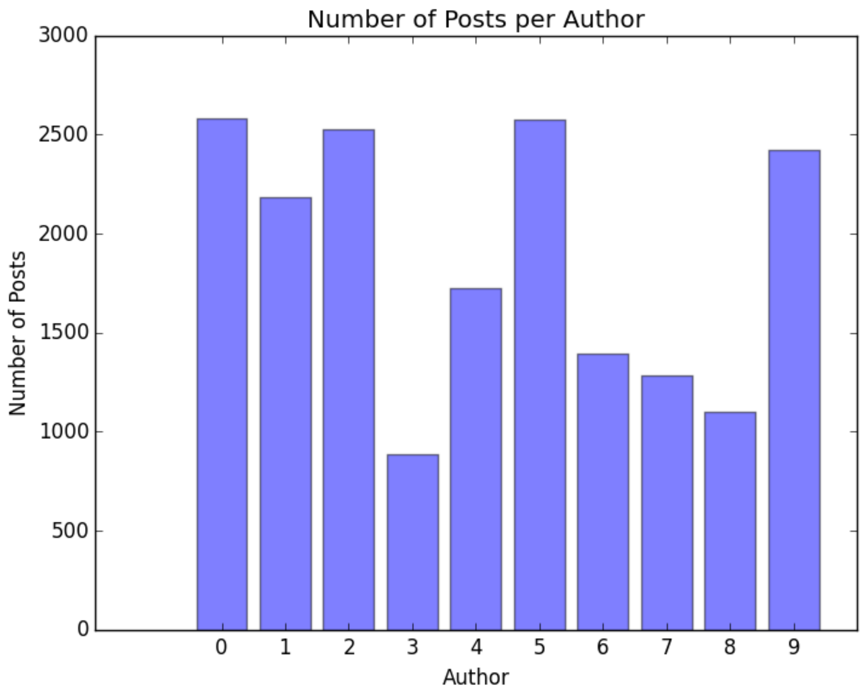


Table 1

The number of posts are not evenly distributed between authors, as would be expected in a real-world data set. The final analysis of the models will take this into consideration with a weighted average of the F1 scores. The total vocabulary size for this sub-corpus is 58,235. The longest post has 4,046 tokens. Both of these figures will be used when creating the CNN. The average post length is just under 127 tokens. Below is an sample taken directly from the raw Wikipedia dump files:

```
==What is N.O.?==
Nevermind, this was already discussed on this page. Anyway, I think something in
the article should be added that explains that N.O. has no meaning (much like the
title).  --[[User:Cyde|Cyde]] 00:04, 12 November 2005 (UTC)
:All I know is that N.O. refers to a &quot;hyperspace channel&quot; through the
brain, so maybe N.O. is a japanese acronym of this, or something to do with the
brain, specifically right vs. left. Speculation, maybe someone who knows japanese
can find out.         -Dunce(2005)
::I have a feeling that the answer to that is spelled N.O. --[[User:Yar Kramer|Yar
Kramer]] 07:53, 13 December 2005 (UTC)
```

This sample represents one topic of conversation within a talk page. Each talk pages may contain many different topics relating to the original article. This topic contains three posts by three different authors. The authors in this example all sign their posts either with a Wikipedia link to their own user talk pages e.g. *[[User:Cyde|Cyde]]* or with a simple plaintext signature *-Dunce(2005)*. Additionally the post contains a reply structure indicated by the colon. In this case each post is a reply to the post directly above. However, reply structure information will not be used in the classifier. Also not considered is the topic under which the post was made or the article to which the topic belongs. Instead only the tokenized post text will be used as input to the classifier. This is the most straightforward way to get a baseline for the classifier, and also the most generalizeable.

Algorithms and Techniques

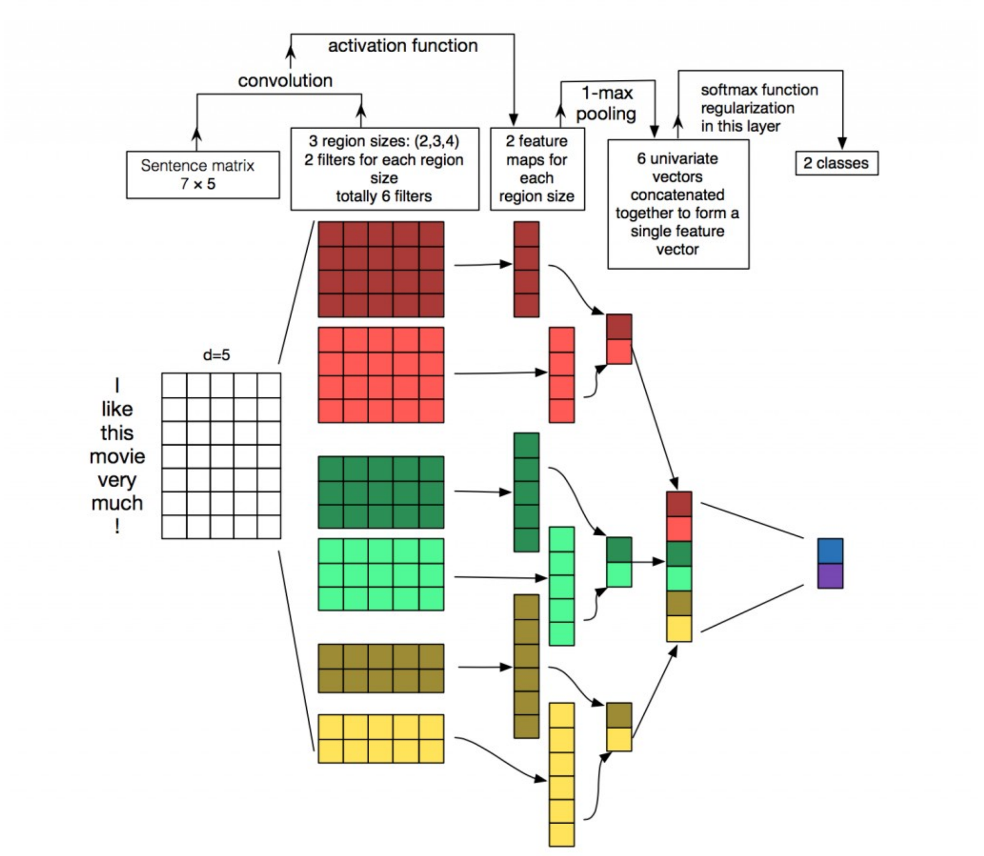

activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I
like
this
movie
very
much
!

Illustration 1

I'll be using a convolutional neural network as the classifier. Other types of classifiers typically used for NLP tasks assume a 'bag of words' model. A model in which the order of words does not matter, these classifiers rely heavily on vocabulary rather than any deeper structure or more interesting features in the text. Even though there may be better models for NLP tasks, such as recurrent neural networks, CNNs turn out to work pretty well. They are also relatively fast[3].CNNs work on text by using word embedding vectors to get a multidimensional array, similar to the structure of an image. Consider this helpful diagram of a CNN from Zhang and Wallace's paper on CNNs for sentence classification[4].

The sentence is turned into a sxd matrix where *s* is the length of the sentence and *d* the length of the word embedding vector. The word vectors can be pretrained on other corpora such as Google's word2vec or Stanford GloVe. Or, as in the case for this experiment, they can be generated using the Wikipedia talk page corpus. TensorFlow has methods useful for generating custom word embeddings which is what is used in this experiment. With each post an sxd matrix, we can now slide our convolving filters over the matrix as we would in a picture sxd pixels large. The convolving filters are then turned into feature maps via a non-linear activation function (a ReLU in this case). Then a max function is applied to these feature maps to gather the most salient feature from each, these most salient features are then concatenated, dropout is then applied followed by a standard fully connected layer onto which softmax is applied to turn the logits into classifications. This high-level overview of the implementation will be discussed further in a later section.

Benchmark

Why is it necessary to use such a complex model on what seems like a relatively straightforward problem, other than the fact that CNNs are really really cool? Let's take a look at a benchmark off-the-shelf classifier using the bag of words model. A very common, fast, and not-unreasonable classifier for general NLP tasks is Naive Bayes. The Multinomial Naive Bayes model takes in a vector of word counts. Using the Multinomial Naive Bayes model on a training set of containing 10 authors and 14949 examples, a weighted F1 score of **0.8471** was obtained on a held out test set containing 1870 examples. This is a very reasonable result using a simple bag of words model, and should stand as a de facto baseline for further experiments.

Methodology

A significant amount of data preprocessing is needed for this experiment, as with most NLP tasks. The data underwent several steps before being sent to the classifiers. First the talk pages were separated from a large (>100 GB) file of the latest Wikipedia metadata database dump. This resulted in millions of files corresponding to different articles talk pages. Only talk pages that contained interesting posts were selected—that is, some talk pages contain only boilerplate tags with no discussion—these were not selected. The talk pages were then processed to extract the reply structure (although that is not utilized in these experiments) through an heuristic method. The topic of each post was also extracted (again not used in these experiments). Finally the author was extracted using several regular expressions for author signatures. The literal post text, extracted author and other metadata are then stored in json format. After this a list of authors is created, sorted by the total number of characters in all their talk page posts. Users were determined to be automated bots if they had the substring 'bot' anywhere in the user name. This is not entirely accurate, and there does exist a list of current and

3 http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/
4 http://arxiv.org/pdf/1510.03820v3.pdf

former bots that post to Wikipedia. Further work is needed to filter the author list. We'll use this list of authors later to create a subcorpus to do smaller scale learning experiments.

Each post is then processed to remove artifacts from Wikipedia markup, HTML, and most importantly the author signature. Finally each post is tokenized using the default tokenizer in spaCy[5] and stored in a file with the author label. These tokenized post files are then converted to integer representations such that each token is assigned a corpus-wide unique ID. Each post is then seen as a whitespace separated list of integers, posts delimited by the newline character—these correspond to the .corpus.vector files used in the CNN classifier. One additional step needs to be taken before the corpus is ready for the classifier. We need to ensure that each post is of uniform dimension. To do this an extra padding characters are added to all posts that are shorter than the longest post. For example in the corpus with 10 authors, as will be used in all further experiments discussed here, each row contains a vector of length 4,046 with padding characters appended to make up for the length of the actual post. Thus the corpus of 10 authors is an input matrix of shape (18687, 4046)

Implementation

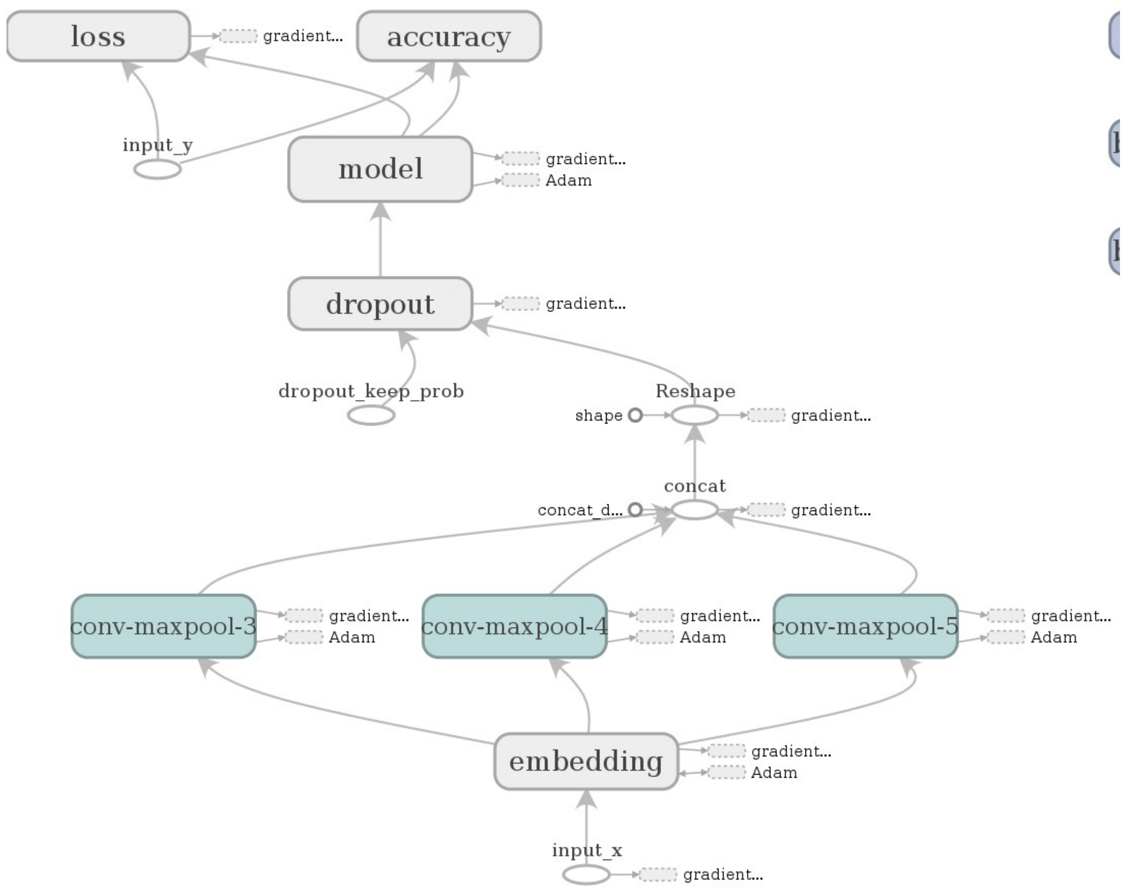
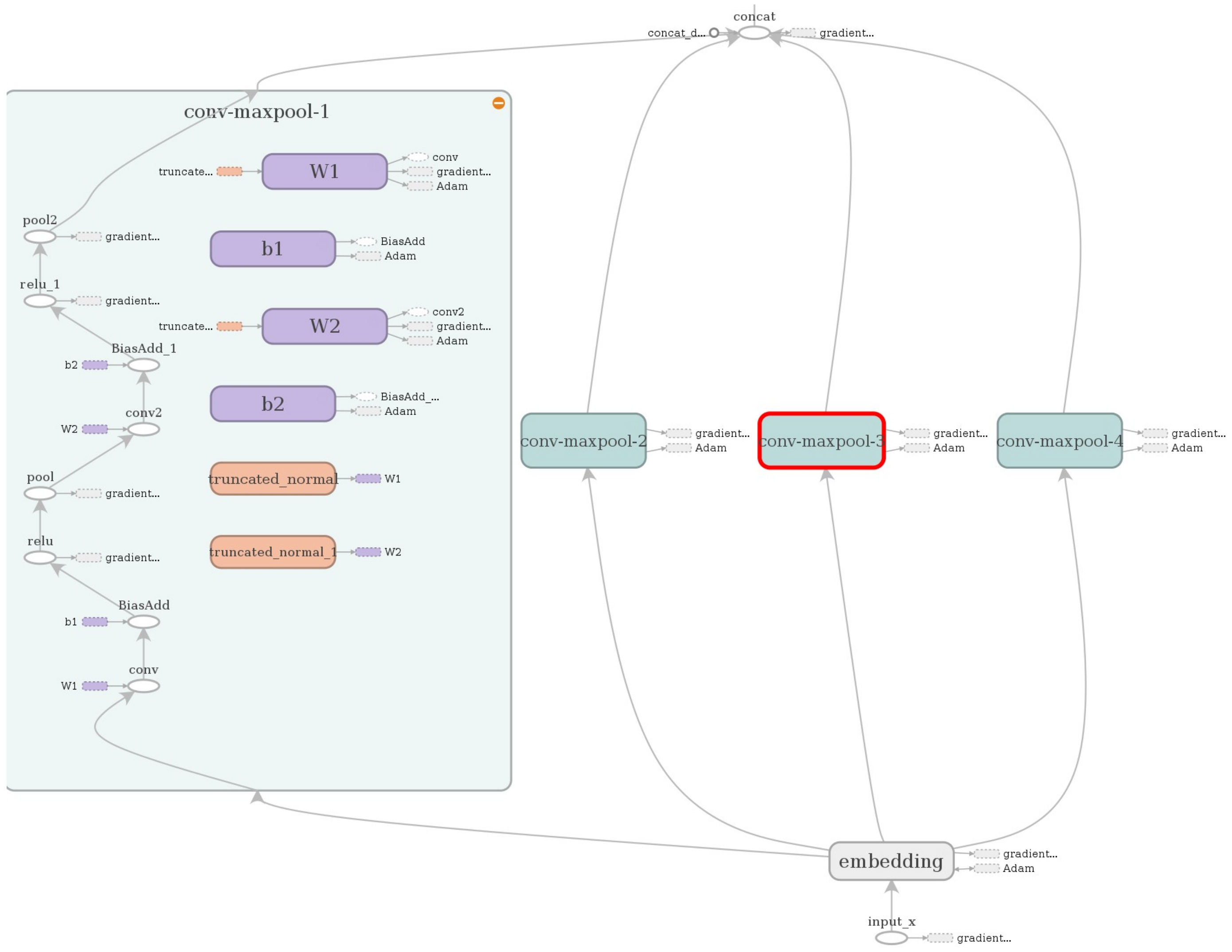


Illustration 2

5 https://spacy.io/

Illustration 2 is great visualization of the TensorFlow graph used in several of the experiments. This graph show the embedding lookup generated from the post vectors, input_x. This is done with a TensorFlow function nn.embeddinglookup. The model hyperparameter associate with the embedding is the embedding size, how long each word vector is. For all the models I'll leave this at 128.

For this experiment several filter or kernel sizes were used at once, following from Zhang and Wallace. The embedding is used in the first the convolution for each filter size. For each filter size the filter is convolved about the (128x4046) input matrix representing a post. The width of each filter is the same as the length of the word embedding vector. Here the number of filters per filter size is a model hyperparameter., though in these experiments the number of filters was frozen at 64. Then a non-linear activation function applied to the convolved matrix plus the bias. Max pooling is then applied in turn to each filter size, taking the most salient feature of each. In this implementation max pooling function uses a stride size of 1 with valid padding for all experiments. The max pooled features are then concatenated and flattened. For some models 2 convolutional layers were used. In those cases the pooled output from the first convolutional layer are connected to a second convolutional layer. The outputs of the 2nd convolutional layer are max pooled and for each filter size concatenated, exactly as in the case where there is only 1 convolutional layer. This 2nd convolutional layer is shown in the graph below on the expanded view of the conv-maxpool-1 node.



Dropout is applied to the concatenated feature vector such that randomly for every iteration some percentage of the activations are removed. This forces the learner to never rely too heavily on particular features, that is dropout induces redundancy in the network. The dropout probability is another model hyperparameter, for now it's set at 0.5. Following dropout there is a final fully connected layer. The fully connected layer produces logits which are then turned into class predictions via a softmax function. Loss is then calculated using cross entropy. Accuracy is reported against the training set at each step. Loss is optimized using AdamOptimizer which handles learning rate decay automatically—the initial learning rate is set at 1e-4.

Training is done in batches the size of which is set via the batch size training parameter. The dataset used in these experiments is a corpus posts by 10 different authors. The data is split once into three groups: training, development, and test. The training set is a random shuffle of 80% of the data, development and test are a random shuffle of 10% each of the data. The random shuffle is set with a random seed so that all experiments use the same set of data. The alternate method would be to use cross-validation, however, this was not implemented. During training the test set is never looked at or used at all. After a specified number of steps, set by a training parameter, the model is evaluated on the development set and the accuracy is reported. In these experiments development evaluation occurs every 40 steps. It's important to note that while during training a certain amount of of activations are dropped, but during evaluation (of development and test sets) all activations are kept—dropout keep probability is increased to 1.

Training was performed on a Google Cloud Compute Engine instance with 8 cores and 52 GB of memory. The large amount of memory is needed to perform the evaluation on the ~1800 input vectors. In the future it makes more sense to code a more memory efficient evaluation function. Training batch sizes were kept at 128. The number of filter was also kept constant throughout the experiments at 64. Increasing the amount of filters radically increased the memory usage of the learner. Training time takes around 20 hours per model on 8 CPU cores. It would most surly be faster on a GPU, but none were available for this experiment[6], as GPUs are better suited to these kind of large matrix operations. Training was stopped after 8560 steps because it was obvious that the initial model was underfitting the data around this point. This was also a good practical stopping point because it took around 20 hours to reach 8500 steps (slightly longer for more complex models, with the final model needing around 26 hours of training time on 8 CPU cores).

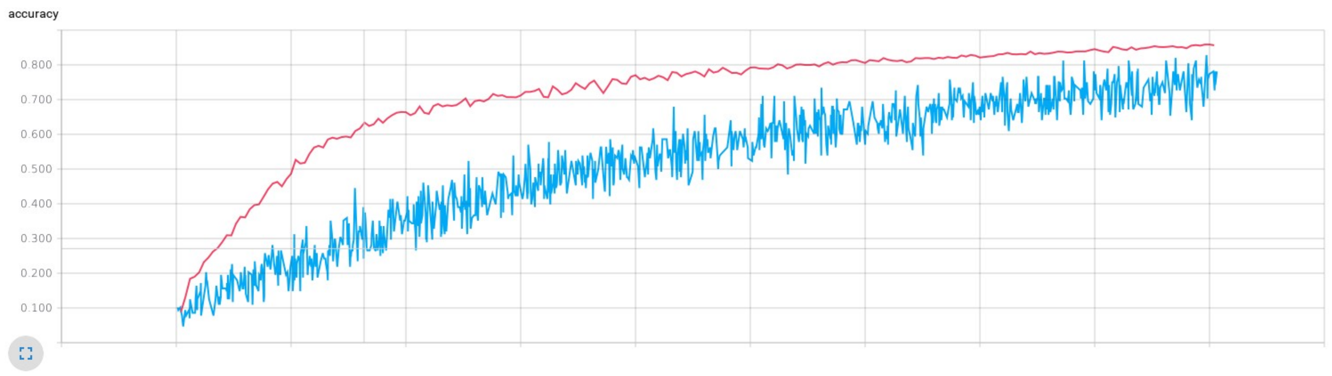
Refinement

The initial model was changed in two distinct ways. The initial results and model are summarized below:

conv_layers	1
BATCH_SIZE	128
DROPOUT_KEEP_PROB	0.5
FILTER_SIZES	3,4,5
NUM_FILTERS	64
Train/Dev/Test split	14949/1868/1870
Accuracy:	0.9170
F1 score	**0.9176**

This model has 3 filter sizes, and shows reasonable improvement over the baseline Naive Bayes classifier. However improvement could be made. It was clear that because the training accuracy had not surpassed the development accuracy the model was underiftting the data. Here the red curve is the accuracy on the development set, taken every 20 steps, the blue cure the accuracy on the training set, taken every step.

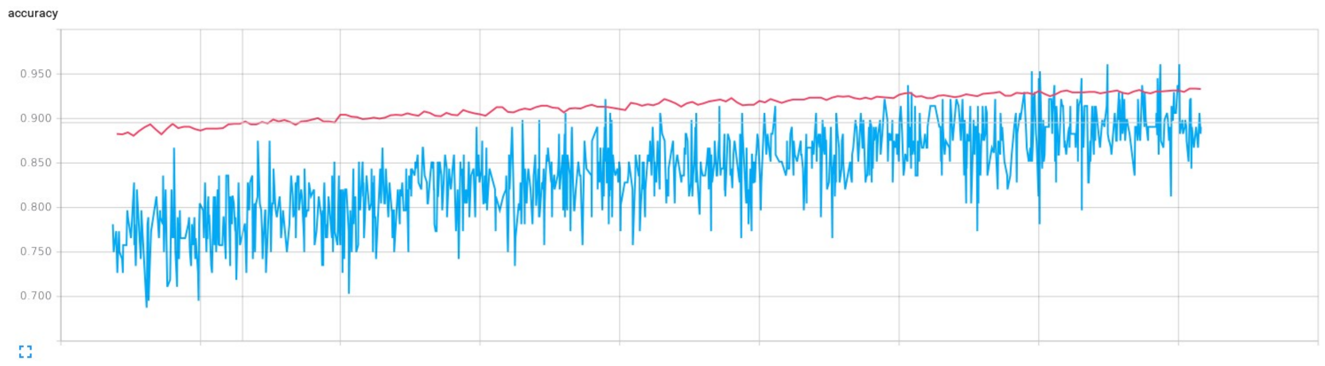
6 I used GCE free credits to run these experiments. They do not have GPU instances available, but they have very recently announced instances designed for machine learning. As of now it's invite only.



To increase the model's complexity and thus reduce underfitting, additional filter sizes were added. This proved to be a minor improvement over the initial model. The parameters and results of the 2nd model are shown below.

conv_layers	1
BATCH_SIZE	128
DROPOUT_KEEP_PROB	0.5
FILTER_SIZES	*1,2,3,4*
NUM_FILTERS	64
Train/Dev/Test split	14949/1868/1870
Accuracy:	0.9241
F1 score	**0.9245**

Although this is an improvement over the initial model, according to the training curve there is probably still a small amount of underfitting.

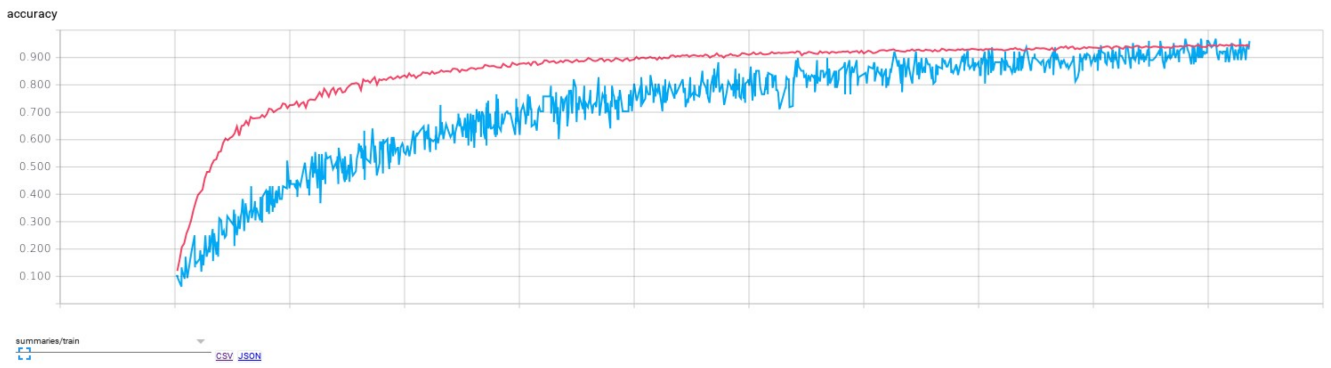


I tried increasing the maximum filter size to 5, but this proved to take up too much memory for even the 52 GB instances. Instead the number of convolutional layers was increased to 2. Below is the model summary.

conv_layers	*2*

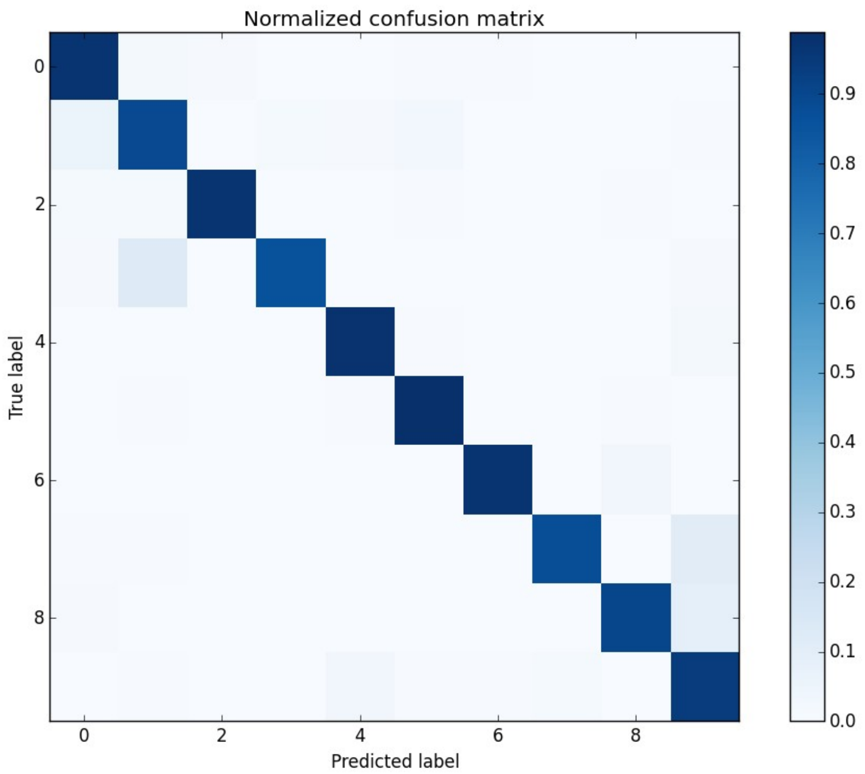
BATCH_SIZE	128
DROPOUT_KEEP_PROB	0.5
FILTER_SIZES	1,2,3,4
NUM_FILTERS	64
Train/Dev/Test split	14949/1868/1870
Accuracy:	0.9422
F1 score	**0.9421**

This model shows significant improvement over the initial model. This is the final model tested, but according to the training curve, it's possible that additional convolutional layers or ever additional fully connected layers could increase F1 score on the test set.



Results

Model Evaluation and Validation



Above is a normalized confusion matrix of the final model on the unseen test data set, scoring an F1 score of **0.9421**. This is an improvement over the baseline classifier and other less complex models. From the confusion matrix it seems the model has 2 authors the model has trouble disambiguating from author 9, namely author 8 and author 7. It's not clear why this is the case. From table 1 on page 2 both authors 7&8 have roughly half the posts of author 9. It could be the case that all three authors share some similarity, and because of the post imbalance it's more often the case that author 9 is the true author.

The model also has trouble telling author 1 and author 3 apart. It seems to predict author 1 when the true author is author 3. Again looking back at the table on page 2, we see the same type of imbalance. Author 3 is the least frequent poster by more than 100%. It could be the case that authors 1&3 are somehow similar, perhaps they post on the same articles, but because author 3 is a relatively infrequent class the model chooses author 1 some of the time on more ambiguous posts.

Otherwise the model seems pretty robust in that it can generalize very well to unseen data in the test set. While it's possible that better results could be attained by increasing model complexity, this was not done due to time and resources. There is one significant tradeoff between the baseline Naive Bayes classifier and the CNN classifier, the time and resources to train each classifier. Naive Bayes classifier takes a trivial amount of time to train, on the order of seconds, while the CNN classifier takes on the order of days to train (using CPU instances).

Justification

Let's use McNemar's test to determine if the improvement from the baseline classifier is a significant result. McNemar's test is given below along with the p-value calculation

$$\chi^2 = \frac{(b-c)^2}{(b+c)} \qquad p = -\binom{n}{b} 0.5^b (1-0.5)^c$$

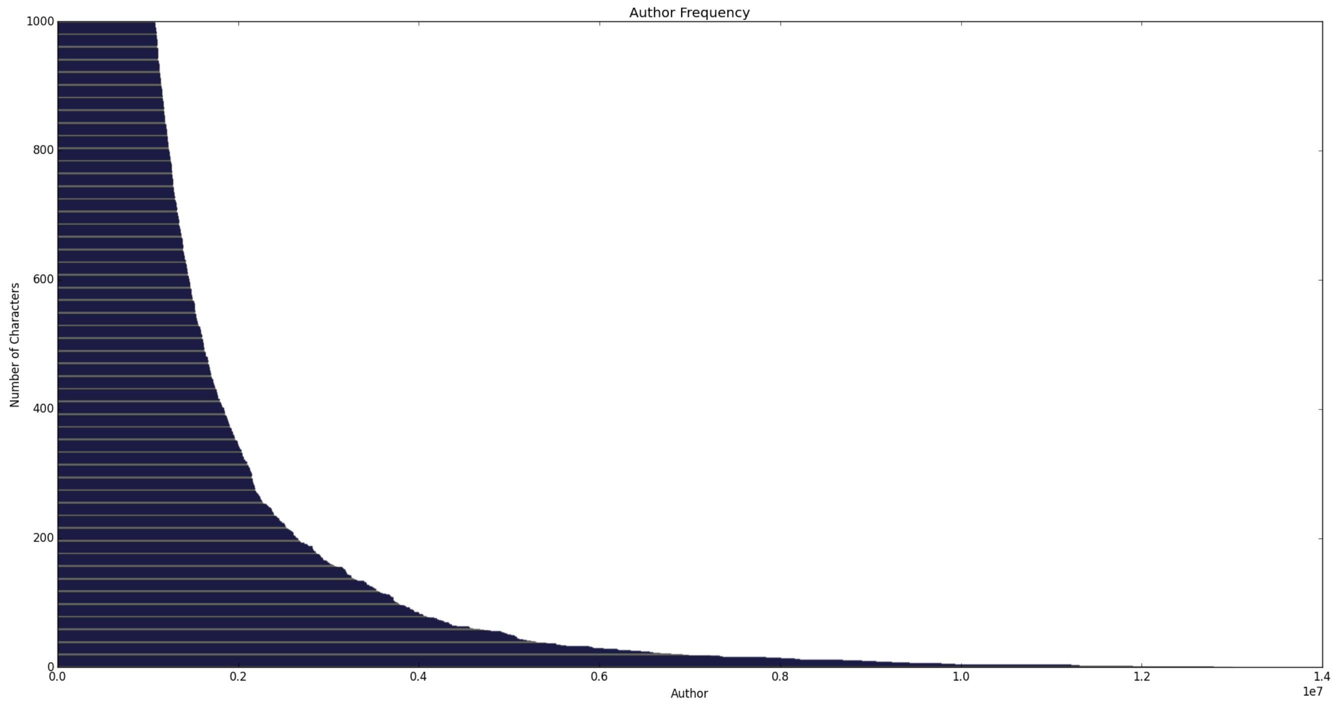
McNemar's test is used on nominal data, so I'll use the floor of F1 score of each classifier and the number of samples in the test set to retrieve nominal data.

	F1 score	floor(F1 x sample size)
NB	0.8471	1584
2-layer conv net	0.9421	1761

Using these definitions we find the following:

p-value	**< 2.2e-16**
McNemar's χ-squared	175.01

Indicating that a near 10% increase in F1 score on the same test set is indeed a very highly significant

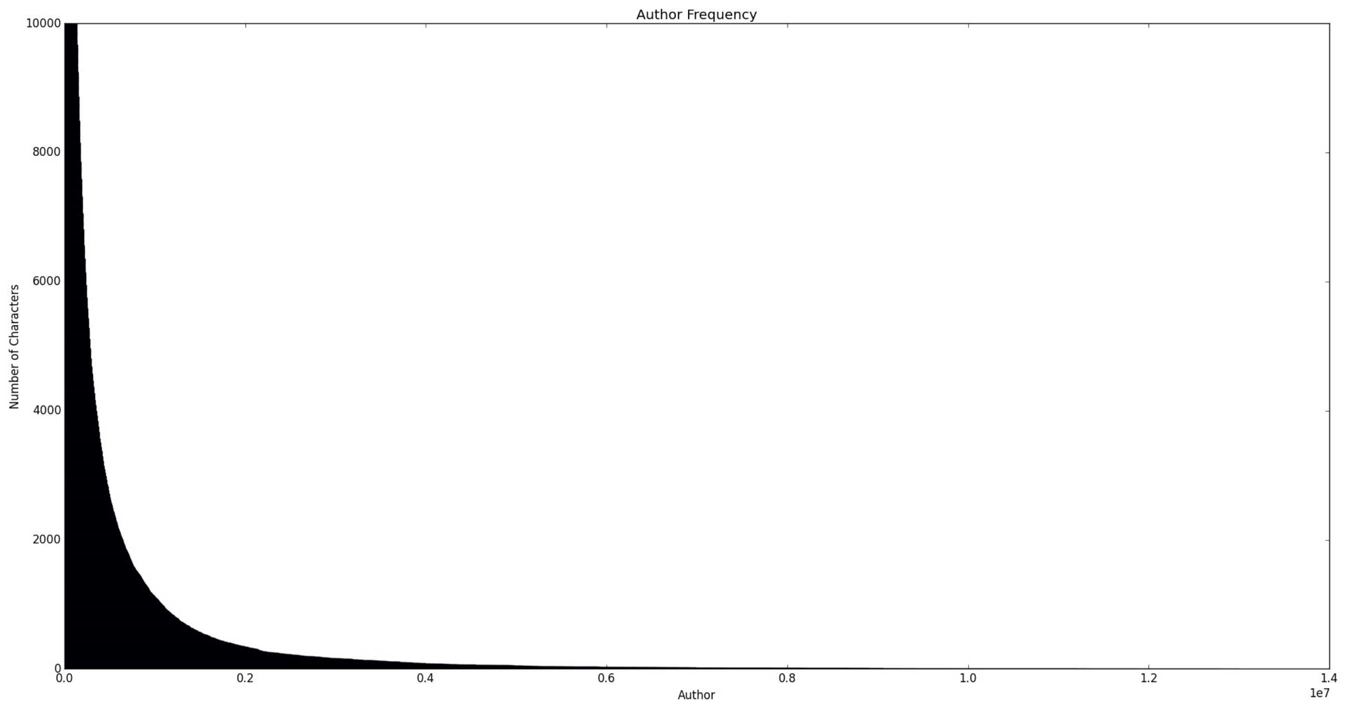


improvement. In fact the test set is reasonably large, large enough to conclude that the final classifier is an excellent solution to the problem of classifier the top 10 Wikipedia talk page authors.

Conclusion

Although the classifier works very well on the top 10 most prolific authors, it's worth noting that there is an ample amount of data for these 10 authors. In fact, the most prolific author are exponentially more prolific than the rest of the authors on wiki talk pages. See the chart below of the the number of total characters each of the top 1000 authors' post history. On the X-axis is the total number characters, the Y-axis the individual author sorted by number of characters in their post history. Given this more clear picture of the larger data set, it might well be the case that the classifier starts to perform exponentially worse at the number of authors in the corpus increases. And, unfortunately, this would be the much more interesting experiment.

This problem becomes even more clear when we look at the first 10,000 most prolific authors.



After around the top 5,000 authors the total number of characters reduces more slowly. A further interesting experiment would be to train the classifier on at least the first 5,000 authors. It's possible that the first 5,000 authors still post at a frequency that there remains strong disambiguating features. It's not immediately clear if the classifier will ever work on the authors who post with little frequency. Further work will need to be done to determine this, but for now it's clear that CNNs are a significant improvement to the baseline Naive Bayes. Though CNNs are an improvement over NB, there is significant overhead in training the CNN classifier. And I suspect that as additional authors are added to the training of the CNN, the time to train will increase exponentially.

Also worth pondering is the application of the classifier to Wikipedia articles themselves. Because of the final models rather opaque interpretability, it's not clear at all if the model will generalize well to the author's style on articles rather than the more informal talk pages. In other words, it's impossible to tell if the salient linguistic features present in the talk page posts carry over to the articles. Nonetheless I've demonstrated that using convolutional neural nets it's possible to correctly retrieve the author information given enough data.