“I Have the Best Classifiers”: Identifying Speech Imitating the Style of Donald Trump

Michael Dickens
Stanford University
mdickens@stanford.edu

Abstract

These days, people love to write comments that imitate Donald Trump’s speaking style. But inevitably, a few folks won’t get the joke. It’s tragic how these people end up getting left out of all the fun.

This project is an attempt to rectify the situation. If there existed a program where you could give it a comment and it would tell you if the comment is written in Trump’s style, this would solve everyone’s problems. No longer will people miss the joke: they can simply run a comment through the Donald Trump recognizer to determine if what they’re reading is meant to imitate that great orator’s style.

This paper describes a text classifier based on a convolutional neural network (CNN) that determines whether an input is spoken by Donald Trump (or meant to imitate his style).

1 Introduction

This classifier should be able to identify text spoken by Donald Trump, or designed to imitate Trump. I am primarily using text from 2016 presidential debates (The American Presidency Project, 2016) because this provides a plentiful source of text–about 250,000 words. I considered including other sources in my corpus to get more non-Trump examples, but text from debates may look substantially different than text from e.g. news articles or novels. A good classifier should be able to distinguish Trump’s speech during a debate from other people’s speech.

In particular, a classifier should be able to identify text that is not written by Trump, but that imitates Trump’s style. This sort of text is much harder to come by, so I only have a few examples which I am saving for the test set. I will primarily evaluate the results using development/test sets drawn from the primary corpus, and include the imitation-Trump text as an extra check.

I develop a convolutional neural network (CNN) to classify text as spoken by Donald Trump, including a basic model and some modifications. I find that a basic model performs well here, achieving 84% accuracy on a test set; and no major modifications allowed the model to exceed this performance.

2 Literature survey

Our first objective is to identify a neural network structure that can perform well at text classification. Several different techniques have been historically used with some success.

Some, such as Wermter (2000), have used recursive neural networks for text classification or for similar problems in natural language processing (e.g., Socher et al., 2012). Using recursive neural networks requires creating some sort of structure on top of flat sentences, which can make them relatively difficult to implement effectively.

More recently, researchers have had success with applying convolutional neural networks to process text (Kalchbrenner et al., 2014, Santos & Gatti, 2014, Shen et al., 2014). Convolutional neural networks apply convolutions over windows of tokens in a text sequence, which means that phrase meanings are invariant with location in the sentence. At the same time, convolutions allow us to operate over n-grams for $n > 1$ without requiring $O(m^n)$-size matrices, where m is the number of unique tokens in our corpus.

Kim (2014) performed a useful survey, looking at a series of prior approaches for sentence classification on a small set of problems and then applying convolutional neural networks to them. He found that CNN's outperformed the previous state of the art on 4 out of 7 problems. Other machine learning techniques that performed well on these text classification problems include modified logistic regression (Le & Mikolov, 2014), Naive Bayes (Wang & Manning, 2012), and SVM with hand-coded rules (Silva et al., 2011). These techniques might have good success here (I suspect that at least one of these would outperform neural networks given the relatively small data sample) but that goes beyond the scope of this paper.

3 Primary model

A baseline logistic regression algorithm achieved 77% validation accuracy, so a well-tuned neural network must perform better than this to add value. I developed a primary model first as a neural network with a single convolution, and then with multiple convolutions; I later implemented a secondary non-convolutional model that performed more poorly.

3.1 Embedding

Before running words through a neural network, we need to produce an embedding. Here we have two primary choices: we can use pre-trained word vectors, or we can add an embedding layer to the network to produce a custom embedding. Here I frequently refer to word embeddings, although strictly speaking the input takes tokens, not words. I used NLTK's `TwitterTokenizer` to parse out tokens including punctuation, on the assumption that different speakers may use punctuation differently and therefore a model ought to have access to information about punctuation.

Initially, I used a trained embedding rather than pre-trained input word vectors. A system like word2vec captures word meanings, but we care about capturing semantic information other than the meanings of words. I suspected that a built-in embedding layer would capture this more effectively than pre-trained word vectors would. Consider for example that word2vec can model the relationships between word meanings (Deeplearning4j 2016), but this is not exactly what we want. We want to identify information about words that's relevant to classifying speakers, which relates more to word choice and syntax than word meanings. For instance, Trump is well-known for his frequent use of the word "best" (although, perhaps surprisingly, he has in fact only used the word three times total during all the presidential debates to date). Although the word "greatest" means the same thing as "best" in most contexts, if we had text from a speaker who frequently used the word "greatest", this would not constitute good evidence in favor of Trump being the speaker.

I converted each token into a unique positive integer and used these to index into an embedding in the neural network's trained embedding layer. The network takes in sentences matrices and sentences do not have a defined length. For simplicity, I truncated each sentence to a fixed size of 30 tokens. For sentences with fewer than 30 tokens, I added a special dummy token to the end to expand the sentence to length 30, which allows the network to "know" the length of the sentence. We do lose information for sentences with more than 30 tokens, but 30 should still be enough to capture almost all of the useful information.

I did not have strong confidence in my hypothesis that a trained embedding would outperform pre-trained word vectors, so I tested the latter as well. I used pre-trained GloVe word embeddings and then modified the CNN to remove the embedding layer and instead take three-dimensional matrices as input (where input[i, j, k] gives the i^{th} sentence, j^{th} word, k^{th} index in the word representation).

The modified neural model using GloVe performed a bit worse than the original model but not substantially so. This suggests that the GloVe word embeddings did encode useful information about tokens, but that we could get enough information out of the training data to produce similarly good embeddings.

3.2 Single-convolution network

I began by writing the simplest possible convolutional neural network. It has a single embedding layer and only performs a single convolution over bigrams in the input. I ran just a single epoch across all my training data. Across multiple runs, this gets a training accuracy of around 75-85% and a somewhat lower development accuracy. Even this simple neural network outperforms the benchmark about half the time. Empirically it seems to take about 30 epochs before training accuracy converges to about 90%, although using this many epoch causes overfitting, and the neural network performs somewhat worse on the development set than it does with only a single epoch–it fairly consistently gets 78-79% dev accuracy.

3.3 Multiple-convolution network

A model that only performs convolutions over bigrams is somewhat limited because it does not capture information about syntactic structures larger than two tokens. I improved the neural network to include convolutions with larger filter sizes and applied max pooling to flatten the result into a single matrix, which then went through a final output layer with an affine transformation and a softmax activation function.

This more complex model that uses convolutions over bigrams and trigrams rather than just bigrams performs somewhat better. Training accuracy increases to over 90%, although perhaps this isn't particularly meaningful–it just means the CNN has a tighter fit to the training data, which might not be what we want.

Development accuracy does not improve when we only train for a single epoch, but when we train over 10 epochs, it increases to 81-83%.

3.4 Hyperparameter optimization

On top of the primary model, I experimented with a number of hyperparameters, including learning rate, the random initialization function for weight matrices (uniform vs. truncated normal), convolution filter sizes, embedding size, and number of epochs. Training ran fairly quickly on the relatively small data set, so I was able to test many combinations of hyperparameters. I used a greedy search to move through the (large) space of possible hyperparameters and converged on the best combination. These hyperparameters listed only had a small effect on the results; the classifier did perform poorly when trained on fewer than about five epochs, but the random initialization function, embedding size, and learning rate did not appear to affect the results.

Filter sizes did matter somewhat more. My initial model with only a single convolution over bigrams performed somewhat more poorly than models with more convolutions and a variety of filter sizes. I found that the classifier performed best when using convolutions with filter sizes (2, 3, 4) or (2, 3, 4, 5). The latter appeared to perform slightly better, but the difference could easily be the result of non-meaningful random variations. I also tried applying multiple filters of each size, but this did not perform better than a single filter for each size.

In addition to tuning basic hyperparameters, I added two common features to the model in an attempt to improve accuracy. First, I added dropout after the convolution pooling and before the output layer. I found that this improved performance by about two percentage points for dropout rates of around 20-30%, making it the most significant improvement to the model other than adding more convolutional filters.

Second, I added L2 regularization for the weight matrices used by the model in an attempt to possibly reduce overfitting. I tried a variety of regularization constants and applied regularization to in different ways (i.e., to different weight matrices) but could not find any configuration that improved the performance of the classifier.

With filters over 2-, 3-, 4-, and 5-grams, and with dropout applied, the CNN classifier achieves 84% validation accuracy.

4 Secondary model

Convolutional neural networks perform well for some tasks in natural language processing because convolutions allow the network to treat structures in the input as invariant by location. This appears especially relevant for image classification, where we want to be able to recognize patterns no matter where they occur in the image, but it has benefits for sentence classification as well.

We can produce a similar benefit by taking a matrix representation of a sentence and summing along the columns to produce a single vector. If we represent a sentence as a series of word vectors, that means we encode a sentence as a single vector that is the sum of its constituent word vectors. If we represent tokens as one-hot vectors then we can represent a sentence as a vector containing 0's for each token that does not occur in the sentence, and 1's for each token that does occur (or 2's, 3's, etc. for tokens that occur repeatedly). (If we use a more complex vector representation for tokens then the sum of a series of tokens does not have a straightforward meaning.)

This method has the downside that loses all information about which words occur near other words, unlike convolutions which preserve some of this information. We can fix this by encoding a sentence as a sum of its constituent n-grams for $n > 1$. But it becomes increasingly complex if we want to use n-grams for $n > 1$ because the possibility space increases exponentially.

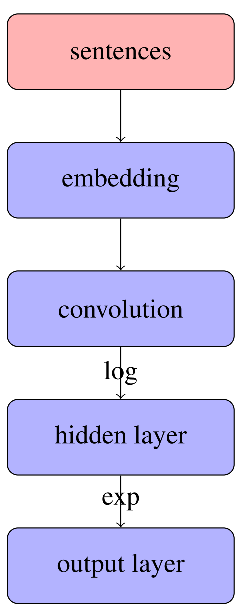
I implemented a simple one-layer neural network using this representation that simply takes sentences as input and then applies a softmax activation function to a weighted sum of the input. This model works similarly to logistic regression–it essentially takes a linear combination of the input and then applies a logistic function–but performs a bit better: I was able to get 81% validation accuracy with this neural network, versus 77% for logistic regression.

5 Attempted improvements

5.1 Multiplicative hidden layer

I had been toying with the idea of developing a neural network for the purpose of predicting stock prices as a function of company fundamentals. One innovation I had considered there could potentially be applied to sentence classification.

For predicting stock prices, we want to be able to take weighted products of company fundamentals as well as weighted sums. We can add a multiplicative layer to a neural network by taking the logarithm of inputs, applying an affine transformation, and exponentiating the result.



I see no good theoretical reason why this should work well for text classification, but I have limited experience and my intuitions are poorly trained (my mental neural network about neural networks only has a few months of training data) so I decided to try it anyway. I added an additional hidden layer to the network after the convolution layer and before the output layer. First I tried making this a simple affine transformation with a softmax activation function and found that, unsurprisingly, this did not improve the classification accuracy. Then I modified it to take the logarithm of inputs to the layer, apply a linear transformation, and then exponentiate the result. This is mathematically equivalent to taking a weighted product:

$$y_i = x_1^{W_{i1}} \cdot ... \cdot x_n^{W_{in}} \cdot b_i \tag{1}$$

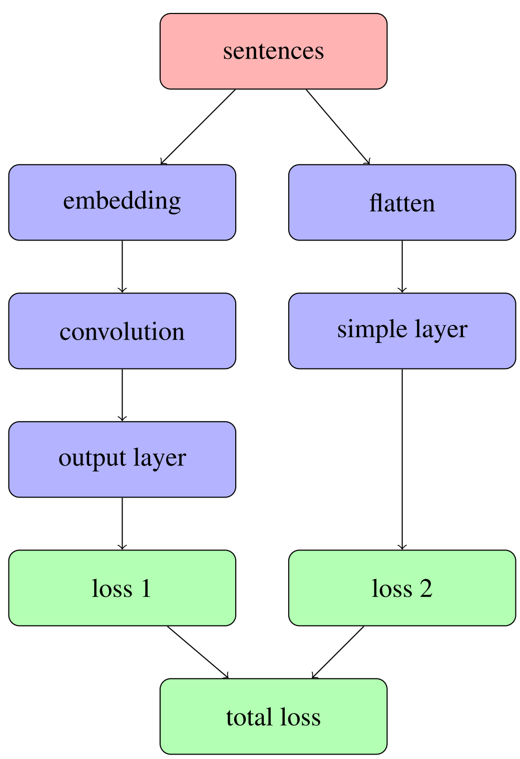
We have no theoretical reason to believe this computation should be meaningful when applied to pooled word vectors, but it's an unusual result and perhaps we will see unexpected success.

But although successful approaches are often unusual, most unusual approaches don't work; this attempt fell into the latter category. The CNN with this additional multiplicative layer performed a few percentage points worse than it did without.

5.2 Two-column network

Kim (2014) implemented multichannel convolutional neural networks, which run the same filters across two sets of word vectors, one static and one non-static. Given that pre-trained embeddings did not perform better than embeddings trained on the model, I did not believe this would provide much benefit to my model; but it did inspire a similar idea. Rather than using multiple sets of word vectors and apply the same filters, we can apply multiple filters to a single set of inputs. A convolutional neural network typically does something like this by computing convolutions for multiple filters. We can further expand this by performing computations other than convolutions.

I previously described two different models: a primary convolutional network that achieves 84% accuracy, and a secondary simple model over collapsed sentences that achieves 81% accuracy. A two-column neural network combines these by applying each separately on inputs and taking a weighted sum of the losses for each (for some fixed weighting).



This performed as well as a convolutional neural network on its own. This suggests that the weaker second column does not contribute additional information to the classifier beyond what the CNN already has, so it does not improve the classifier's accuracy.

A brief literature search did not find any previous attempts at this sort of solution, although it is a sufficiently obvious idea that I suspect it has been attempted before. An extension of this project could survey the literature more carefully for solutions similar to this one to determine if there exist ways to improve the approach and possibly perform better than a simple convolutional neural network.

6 Testing on fake Trump text

If our objective is to recognize when people are writing in a style meant to imitate Trump, we should test our classifier not just on Trump speech but on fake Trump speech. Such text is hard to come by, but I collected about a dozen sources containing a total of a few hundred sentences written in a Trump-like style.

I found that the classifier actually performed slightly better on this test set than it did on the development set. This result was fairly consistent–a logistic regression algorithm showed the same pattern.

Why might this be? I suspect that fake Trump speech is actually *more* Trump-like than most things Trump says, because they are designed to be recognizable. For example, this sentence is clearly meant to imitate Trump's style:

> Believe me when I say, those worms don't want anything to do with me. Trust me. I'm going to bring in all the best people, incredible people, people you wouldn't believe, believe me. (Karnofsky 2016)

This is a direct quote from Donald Trump, but it sounds much less "Trump-like" than the first quote:

> It was originally supposed to be that way. And certainly sounds better that way. But it has all been taken over now by the bureaucrats in Washington, and they are not interested in what's happening in Miami or in Florida, in many cases. (American Presidency Project 2016)

These texts are not meant to be representative–most of Trump's speech sounds more like the former quote than the latter. Rather, the relevant point is that Trump talks in a lot of different ways, and doesn't always follow his same iconic style. But people imitating Trump as a joke must always sound like Trump–they want their speech to be recognizable as Trump-like, or else the joke won't work.

7 Discussion

The neural network models tested were unable to achieve validation accuracy greater than 84%. They did outperform a logistic regression model, but not by a wide margin; and they performed worse than they have on some other text classification tasks. Kim (2014) surveyed the performance of convolutional neural networks on several text classification tasks; on binary sentiment analysis, subjectivity analysis, and opinion polarity detection, CNN's achieved accuracy of 88% or greater. For classifying movie reviews and customer reviews, CNN's performed as well or worse on these compared to classifying speakers.

We have reason to believe that classifying speakers should be more difficult than, e.g., sentiment analysis. A human reader can fairly easily classify the sentiment of a sentence, but cannot easily identify the speaker. Here we attempted to classify Donald Trump's speech because he has a particularly iconic style of speaking–this makes the problem more interesting, but perhaps more importantly, it makes his speech easier to identify.

This project may have suffered somewhat from relatively small data samples. It is difficult to acquire a suitably large corpus of text spoken by a single person, which imposes some limits on how well we can train a neural network. This could potentially lead to overfitting, although I found that the CNN classifier did not have greater validation error after running many epochs than after few, and L2 regularization did not improve performance (as we would expect if we were overfitting our data).

Perhaps one could improve upon these results by applying a deeper network. Generally speaking, deeper networks require more training, which suggests that this may not be effective unless we can acquire a larger data set; so I am not particularly optimistic about this strategy.

8 Concluding Remarks

This model works well. Some people say it should be better. They say other classifiers do better. But the other classifiers, they work on different problems. I'm not classifying movie reviews here. This problem is harder. I'm working on the biggest problem. The biggest. This model, it's not good enough. I need a small loan of a million dollars. And then I'm going to build the best model. I'm going to build a classifier and make Trump pay for it.[1]

Notes

[1]The classifier successfully classifies 10 of these 13 sentences as being Trump-like.

References

1. The American Presidency Project (2016). Presidential Debates. Retrieved from `http://www.presidency.ucsb.edu/debates.php`
2. Karnofsky, H. (2016). Beat the Worms. Retrieved from `http://www.givewell.org/node/2654`
3. Deeplearning4j (2016). Word2vec: Neural Word Embeddings in Java. Retrieved from `http://deeplearning4j.org/word2vec`
4. Kalchbrenner, N., Grefenstette, E., & Blunsom, P. (2014). A Convolutional Neural Network for Modelling Sentences. Acl, 655665.
5. Kim, Y. (2014). Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882.
6. Le, Q. V., & Mikolov, T. (2014). Distributed representations of sentences and documents. arXiv preprint arXiv:1405.4053.
7. Santos, C. N. dos, & Gatti, M. (2014). Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts. In COLING-2014 (pp. 6978).
8. Silva, J., Coheur, L., Mendes, A. C., & Wichert, A. (2011). From symbolic to sub-symbolic information in question classification. Artificial Intelligence Review, 35(2), 137-154.
9. Shen, Y., He, X., Gao, J., Deng, L., & Mesnil, G. (2014). A Latent Semantic Model with Convolutional-Pooling Structure for Information Retrieval. Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management CIKM 14, 101110.
10. Socher, R., Huval, B., Manning, C. D., & Ng, A. Y. (2012). Semantic compositionality through recursive matrix-vector spaces. In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (pp. 1201-1211). Association for Computational Linguistics.
11. Wang, S., & Manning, C. D. (2012, July). Baselines and bigrams: Simple, good sentiment and topic classification. In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers-Volume 2 (pp. 90-94). Association for Computational Linguistics.
12. Wermter, S. (2000). Neural network agents for learning semantic text classification. Information Retrieval, 3(2), 87-103.