Machine Learning Engineering Capstone Project

Thomas Lisankie
April 2018

I. Definition

Project Overview

Poetry and lyrical music are quite old art forms at this point. They aim to tell some sort of story and also have the words and rhythm stick in the reader/listener's mind. One of the elements that poets and writers will often incorporate is rhyme. However, thinking of a word that rhymes with another can be difficult. Rhyming dictionaries attempt to alleviate this problem but often return words that just have a similar spelling and don't actually sound similar (since spelling and pronunciation often don't match well in English). Not only that, but writers sometimes want words that don't directly rhyme (such as *boy* and *toy*) but want words that rhyme in an odd sort of way (*drift* and *ship* where only the vowel sounds rhyme). Thus, we're not specifically looking for a way to measure rhyme, but *sound similarity*.

This project aims to take a first step in developing a rhyming dictionary that finds similar sounding words based on the phonological makeup (the way a word sounds) of the words rather than the spelling. A user will be able to enter any two English words, the program will look up the phonological transcription of the word using data from the CMU Pronouncing Dictionary, feed the two words into a machine learning model, and then a categorization of how similar the two words sound will be output (the categories go from 0.0 to 1.0 in increments of 0.1).

Problem Statement

The problem I am addressing is training a model that will tell a user how well two words the user inputs rhyme based on their pronunciation. This is a classification problem.

The tasks I will need to accomplish include but may not end up being limited to:

1. Creating the dataset. The dataset will be based off of the CMU Pronouncing Dictionary. The dataset will be a sampling of this dictionary found by shuffling all the phonemic transcriptions and then taking only the first 5,000 entries of this newly shuffled list. All of these transcriptions will then be paired with one another. Since a similarity function is symmetric, we can cut out half of these pairs since they are duplicates of one other pair in the dataset. This means our total number of pairs will be 12,500,000. 80% of these will be used for training and the remaining 20% will be used for testing. A scalar representation of how similar the two sequences in each pair sound to one another will then be found using an sequence similarity algorithm I created a couple of years ago called Prhymer. Each of these scalars will then have a floor function applied to them so that we can have 11 discrete categories between 0.0 and 1.0 (inclusive) incremented by 0.1 instead of far more numbers between 0 and 1. These categories serve as the targets. After this, each of the phonemic transcriptions will be translated to a string of

integers where each distinct phoneme is represented by a distinct integer. This is so the sequences can be fed to embedding layers.
2. I will then have to explore this data in order to gain a better understanding of it.
3. A baseline model will then have to be created. This will be a Siamese model where each of the sub-models features densely connected layers.
4. The solution model will be a Siamese model using embedding layers and one-dimensional convolution layers. Since it will be a Siamese model, a similarity measurement will join the two sub-models. The similarity measurement I will use will be cosine similarity.
5. The goal is to see how well the solution model can do compared to the baseline model.

Metrics

The metric I will use to evaluate this model is accuracy. The reason I am choosing accuracy is because the goal of the model should be to correctly categorize pairs of phoneme sequences into categories of how similar they sound, and accuracy is a good measurement of how often the model categorizes correctly.

II. Analysis

Data Exploration

The dataset I will be using is a dataset I crafted myself. It uses the phonemic transcriptions of words from the CMU Pronouncing Dictionary dataset. The phonemic transcriptions were shuffled and the first 5000 entries of this newly shuffled set were taken. The Cartesian product was then found with the dataset on itself yielding 25,000,000 total ordered pairs. All duplicates were then removed (this is reasonable to do since sound similarity is symmetric) leaving a total of 12,500,000 ordered pairs. Prhymer was then run on each ordered pair to get a similarity score between 0.0 and 1.0.

Since the scores obtained are values of a continuous function, it was necessary to apply a floor function to each of the values obtained so that 11 discrete categories could be found. After this transformation was complete, I split my data into a training and a testing set (80-20 split respectively).

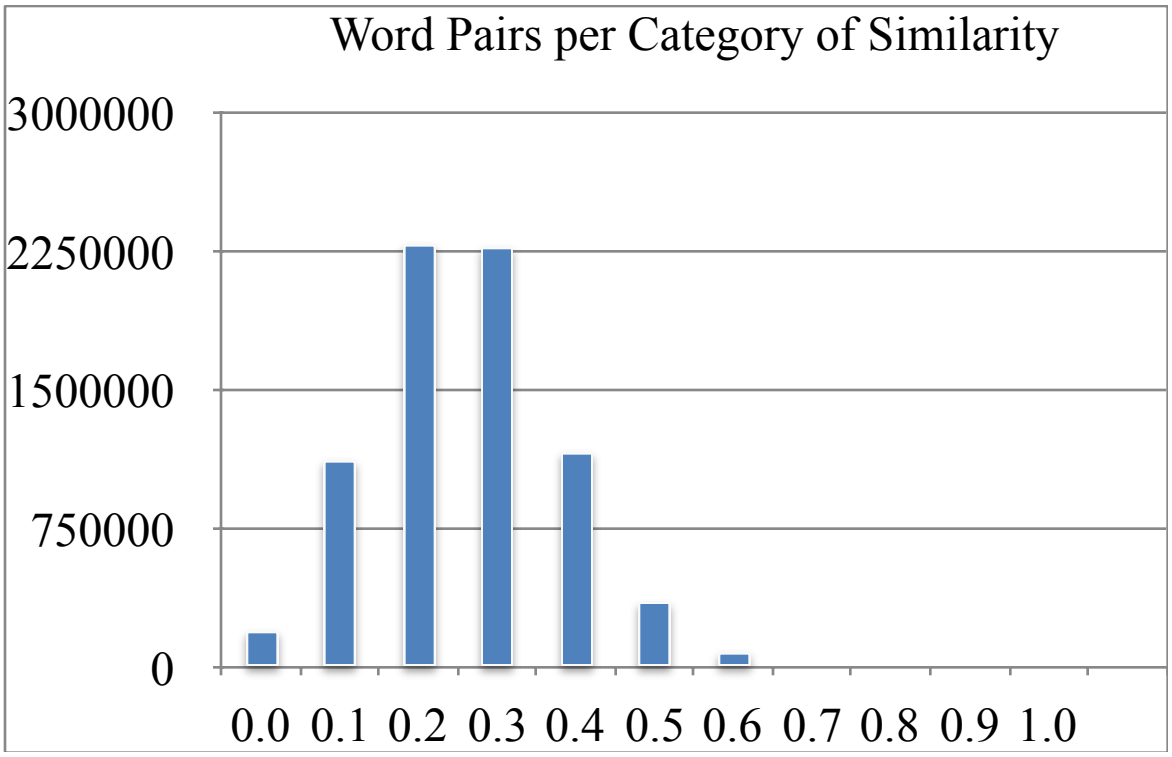
Here are some statistics about the similarity scores of word pairs in the training set:

Mean	0.262
Median	0.3
Mode	0.2

Given that 0.0 corresponds to “these words don’t have any sound similarity at all” and 1.0 corresponds to words being homophones (words that have the exact same pronunciation), these results make sense. Most words don’t sound similar to one another, but many will share a vowel or two and thus won’t be placed at 0.0 or at 1.0, but instead somewhere in between (but closer to 0.0).

Exploratory Visualization

I decided to make a visualization plotting the number of words per category.



As you can see, the number of word pairs per category follows an almost exact normal distribution. This makes sense given that most words don't sound alike at all but may share a common feature or two and thus will be boosted up a couple of categories landing them somewhere between 0.1 and 0.5.

Although it may not appear to be the case, there are actually many word pairs that land in categories 0.7 or higher (22,709 to be exact). The reason they do not seem to even make a dent in the chart is because of the sheer volume of words that don't sound similar at all. The words that compose the categories 0.7 or higher make up only 0.23% of the total word pairs in this training set. And a score of 0.7 is where words start to sound fairly similar and one could see how an artist might use the word pairs as a rhyme in a song.

Of course, the training set only represents a very small sample of the total space if I were to have created word pairs from all the words present in the CMU Pronouncing Dictionary. The total number of word pairs in that case would have been over 17 billion. 0.23% of that is nearly 40 million word pairs that can still be used to make rhymes by artists leaving a tremendous amount of artistic freedom available.

It's also worth noting that since the overwhelming majority of the word pairs are in categories below 0.7, the dataset will again need to be modified. This is because the accuracy metric will not be able to be used properly with such lopsided data. The model will not be able to learn what word pairs that sound very similar actually look like since they will appear to just be anomalies and the model may overfit for those samples or it may just see them as noise and not learn them at all. So, the number of samples in the categories of 0.7 and higher will either have

to be augmented or the number of samples in categories below 0.7 will have to be reduced by some large fraction.

Algorithms and Techniques

I will be using a Siamese neural network architecture for this project. A Siamese architecture is a multi-input model where each of the two sub-networks are identical to each other. The two sub-networks are eventually joined at a layer that applies a distance metric to the two input tensors. This architecture is ideal when you want to find the similarity between two tensors (which is exactly the goal of this project).

Each input tensor will be an integer vector (the phoneme sequences are converted to integers during data preprocessing) of 6 dimensions. The reason I chose 6 was because the average number of phonemes for a word in the dataset is 6.34 so I rounded down. Each input tensor will also be padded with zeroes (for the amount of phonemes it falls short of from 6) and cut off after 6 phonemes.

The input tensor will then be passed to an embedding layer. The reason for using an embedding layer is so that each input vector is assigned to a “pronunciation vector” in the embedding space. This embedding space will map words that have a similar pronunciation closer together with each backpropagation step.

The embedding layer’s output is then passed to two one-dimensional convolution layers with 100 and 50 filters and kernel sizes of 4 and 2, respectively. Each convolution layer used had a rectified linear unit activation function and a stride of 1.

Once the pronunciation vectors are found for each input vector, a similarity metric (I will use cosine similarity) will be applied to find distance between the pronunciation vectors. This new similarity vector will then be passed to a densely connected layer that will categorize the vector using a softmax activation function.

Benchmark

Since there are 11 categories, if you were to randomly pick a category for any given word pair, you would have a 1/11 (or approximately 9%) chance of picking the correct category. So, it would have an accuracy of 9%. My goal is to build a model that surpasses this and achieves an accuracy of at least 10% (preferably much higher). This leads to the model having *statistical power*.

III. Methodology

Data Preprocessing

The steps I took to preprocess the data were as follows:

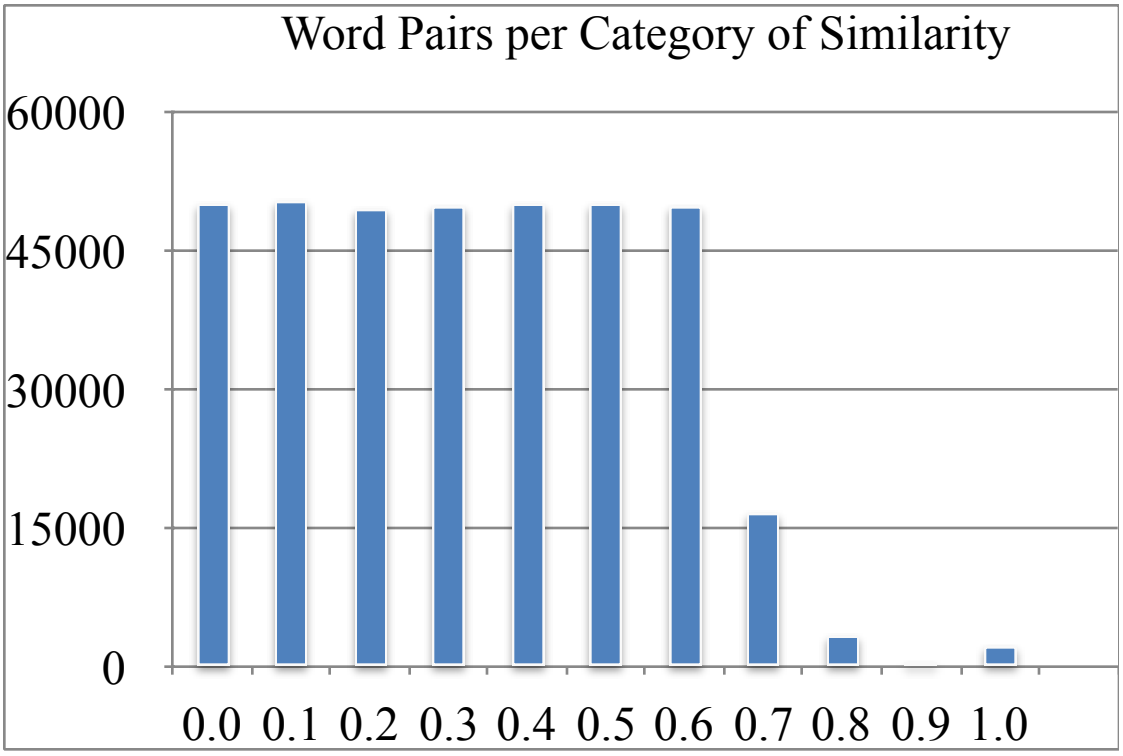
1. Tokenize the phonemes being used. Doing this maps each unique phoneme to a unique integer value (there are 39 unique phonemes).
2. Translate all phonemic sequences to integer sequences using the mapping found in step 1.
3. Pad all sequences with zeroes and cut all sequences that have more than 6 units down so that they all have a length of 6.

4. Since all the labels were floating point values between 0.0 and 1.0 (inclusive) with increments of 0.1, it was necessary to convert these to integer values by multiplying them all by 10.
5. Make labels category vectors.

In addition to all these steps, it was very important to address the problem of an overwhelming majority of the data being in only about half of the possible categories. The choices I had included cutting a large portion of the data out so that the amount of data in each category was far more even or augmenting the lacking data so that it was more even with the plentiful data. I chose the former option.

Even though being able to augment the data would have been an ideal solution (more data is almost always better), doing it may have introduced a whole variety of biases that I would not have been able to predict or handle. Augmenting the words in the latter categories would have involved coming up with some way of generating non-existent words that had phonetic structure that followed the phonological rules of English. It would then involve modifying the generated words so that each had a partner word that was minimally different from it in such a way that would allow it to fall into a specific category.

To cut down on the data, I implemented a function that takes the data and does a random sampling to reduce each category with more than 50,000 items down to approximately 50,000 items. Now, the data amount in the largest categories was at most 100 times the amount of that in the smallest category.



Implementation

I used the Keras functional API with a TensorFlow backend to implement my model. All code is in the included Jupyter notebook.

As discussed earlier, I decided to build a Siamese model as my solution. The model works with batches of size 100. Each input layer takes a 6-dimensional vector.
The embedding layer uses a vocabulary of size 39 (the vocabulary being each unique phoneme) and outputs a 20-dimensional vector (the embedding space that is created is of course 20 dimensions as well). The length of all sequences fed to the embedding layer was 6. This was followed by two one-dimensional convolution layers with 100 and 50 filters and kernel sizes of 4 and 2, respectively. Each convolution layer used had a rectified linear unit activation function and a stride of 1.

The next layer was a merge layer that simply concatenated the two vectors resulting from the last convolution layer of each of the sub-networks.

After this concatenation of vectors was made, the concatenated vector was flattened and then passed to a densely connected layer of 11 units in size (one for each possible category). This layer uses a softmax function for activation.

RMSProp was used as the optimizer and categorical crossentropy was used as the loss function. Accuracy was the only metric I evaluated the model's success on.

Refinement

There were a few different variations of the model I discussed in the Implementation section above that I tried out before finding the model I decided to use as the solution:

- The merging layer I was originally using in the model was a Subtract layer rather than a Concatenate layer. My reasoning for this was that subtracting one vector found in the embedding space from the other should give me a measure of "how far apart" the two words were in pronunciation and then I could just pass the distance vector to the densely connected layer for it to categorize. Using this method was giving me training accuracies of at most 0.23 and a validation accuracy of 0.21. Yes, this technically met my goal of gaining statistical power (any accuracy higher than 0.09 in this case) but I felt there was still a lot of room for improvement. I decided to change this to a Concatenate layer just to see if the densely connected layer would have an easier time classifying the results. It did, and suddenly I was getting a training accuracy of 0.362 and a validation accuracy of 0.359. Looking back, I don't think I had any good reason to believe that simply subtracting two of the vectors from the last convolution layer was going to give me an accurate measure of their distance. After all, the embedding layer I had been training was task-specific and I have no way of knowing what kind of patterns the model was "seeing" when looking at the data.
- After changing the merge layer from Subtract to Concatenate, I wanted to see if removing all the convolution layers would drastically affect the accuracy of the model. I was now getting a training accuracy of 0.33 and a validation accuracy of 0.33. These are still very good compared to the earliest accuracies, but they were not as good as when the convolution layers were in. Of course, it was faster to train than when

convolution layers were in, but I didn't really care about training time in this particular instance.

- I also tried lowering the filter sizes by a factor of 10. So, the first convolution layer now had a filter size of 10 and the second convolution layer now had a filter size of 5. Surprisingly, the accuracies came out pretty close to the best so far. There was a training accuracy of 0.34 and a validation accuracy of 0.343. Still, I didn't want to give up the extra accuracy, so I kept the filter sizes at 100 and 50 for the solution model.

IV. Results

Model Evaluation and Validation

This model was chosen because of how high the training and validation accuracies were. They were both over 0.33 which means that the model is correctly categorizing the word pairs from the training and validation sets over 1/3 of the time.

My testing set had 1,875,251 samples in it. A test accuracy of 0.296 was found when evaluating the model on the entire test set. This is approximately 0.06 less than the validation accuracy. This means that the model is definitely not generalizing perfectly but it's not awful (compared to during training and validation) either. It's still categorizing new data correctly nearly a third of the time.

I would not trust this model in a final product yet. For this to be viable in a real-world setting, the accuracy would need to drastically increase.

Justification

The benchmark established earlier was for the model to have statistical power. Statistical power is simply defined as being more likely to be correct in categorizing a sample than randomly picking a category out of the possible categories. In this case, you would have a 1/11 (or 9%) chance of randomly picking the correct category for any given sample. The accuracy would be 0.09. So, to achieve statistical power, a model must have an accuracy of at least 0.10 (or 10%). The model created here surpassed this benchmark by nearly a factor of 3. The accuracy achieved on the training data was 0.362 and 0.359 for the validation data. The accuracy on testing data (data it had never seen before) was 0.296.

V. Conclusion

Reflection

This project began with wanting to develop a model that would correctly categorize pairs of words based on how similar they were in pronunciation. To be able to start working on the project, I first had to take phonemic transcriptions of words and pair them up with other phonemic transcriptions. Each of these pairs then had to be fed into an algorithm for finding how similar two phonemic transcriptions are (I used an algorithm I created called Prhymer). I applied a floor function to each of the resulting values so that there would only be 11 possible labels.

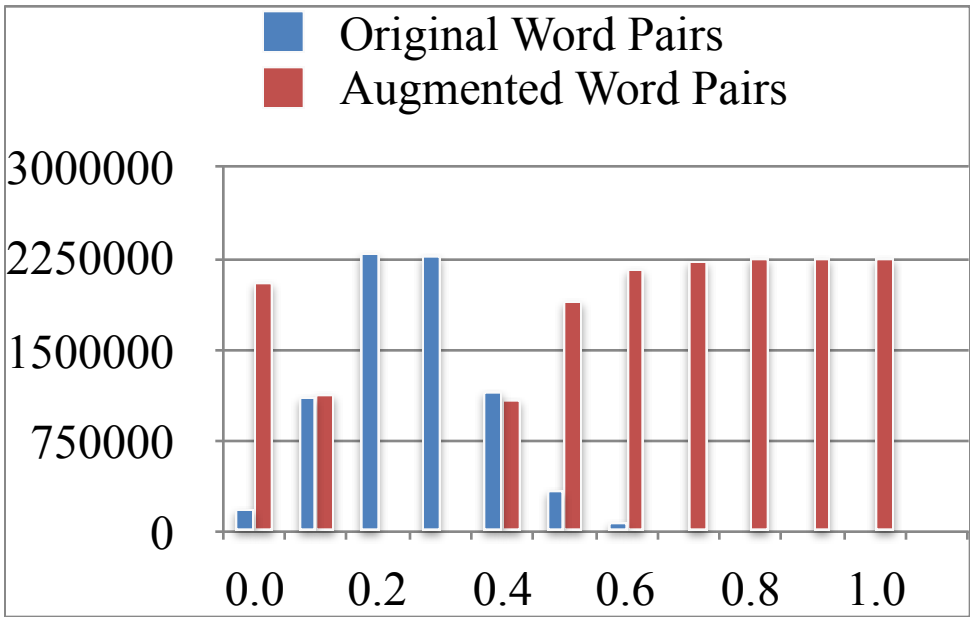
All of this data was then properly preprocessed so that it could be fed into a Keras model. One issue that I knew was going to hurt the effectiveness of my model was how lopsided my data was. I found it very interesting that the number of samples in each category actually ended up falling into a normal curve, but this is not helpful when training a neural network. Because of this, I had to put a ceiling on the number of data points that could be in a category (I chose a maximum of 50,000 per category). In some of the categories, over 98% of the samples were removed. The data was still very lopsided but it was *far* less lopsided than it was previously.

I then had to choose a type of model to use in the project. Since I was seeking to find how similar two tensors were, I chose to use a Siamese neural network model. I chose to use an embedding layer so that the vectors would be mapped into some sort of “pronunciation space” and two convolutional layers for feature extraction and filtering. The resulting tensors from each side of the Siamese model were then merged via a concatenation operation. The resulting concatenated tensor was then flattened and fed into a densely connected layer using a softmax activation function for classification.

I would not use this model in a general product. It may have achieved the benchmark I set for this project of surpassing statistical power, but it still only has an accuracy of slightly less than 30%. If I were to build something like a rhyming dictionary website, I would want the model to have an accuracy of at least 70% or higher.

Improvement

There are many improvements that could be made to this project. One is data augmentation. Since half the categories had far fewer samples than the other half of the categories, it would make sense to augment the data in the categories with less data so that the model would be able to learn what kinds of samples are proper for those categories. This would lead to the model performing better in a real-world setting. This free-form visual shows all the extra samples the augmentation technique would have to create in relation to the original data:



I also would try to improve this model by using a dropout technique. I have read a little bit about it, but I do not currently know how to use it. If I were to use my current solution as a benchmark for another version of this project, I believe that there is a way to surpass it.