Nicole Buckley, Sierra Rodriguez

Language Classifier Final Report

I. Project & Purpose

For our project, we explored language classification based on poetry text corpuses. We initially planned on gathering a few datasets from across the internet, but we ultimately decided to use Kaggle's datasets that integrated well with Colab, which downloads them each time we run through the notebook since colab does not store files between sessions. From a general sense, we aimed to have an algorithm that trains on poems and provides classification to a certain piece of text based on the training data, classifying it into one of the provided languages.

The work is interesting because a lot of text corpuses that exist use language that is either quite repetitive or oftentimes does not capture a lot of the nuances of poetry. For example, poetry would likely use a lot more specific words that do not come up as much in say, a basic literature text corpus or a Twitter dataset that likely has repetitive hashtags or similar statements that come up relatively quickly.

II. Data

We decided on using three languages: Spanish, English, and Portuguese. We wanted to include English for a baseline, as it makes it easier for us to debug and interpret results in English since we both have a relatively strong grasp on the language, so we do not need to deal with learning a new language to handle much of the input or understand what it means. For Spanish and Portuguese, since Buckley is fluent in Spanish and Sierra can read it better than other languages, and Portuguese being syntactically similar seemed like a good test for the model to differentiate between words. For example, there are many words that are spelled the same, so on a pure unigram model, it would be difficult to differentiate these two.

The data ended up coming from Kaggle's integration with Python/Colab, so it is imported every time that the notebook is run and downloaded[1]. The data for each language is . English had by far the most data, having just over 900,000 lines, then Portuguese with 317,296, then Spanish with the lowest at just over 150,000. To avoid English bias in training by sheer size, we scaled all of these down to match with the Spanish set, then further down to about 75,000 lines per language for faster train times. The data was not encoded with labels as we retrieved it, but we added a label of 0, 1, or 2 to correlate with Portuguese, Spanish, and English. The data was relatively clear, but there was some variability; there's a bit more on this in the Naive Bayes section where we discuss finding data anomalies, but there were small amounts of German and Hungarian in the Portuguese set, so the data may be flawed, though by a glance over and testing metrics, it seems like Portuguese is still consistent as the dominant language in the set.

[1] Note: In order to run the code, a Kaggle.json file with API authentication must be present in the workspace. There is a comment in the notebook at the relevant location to explain this.

The data comes in csv files with text for each poem that is extracted out, and we disregard metadata such as the title of each of the poems. For the text itself, we separate it into lines and add tags, <s> and </s> to the ends of each line, as poetry is typically written in lines that have relatively distinct meanings and differences from line to line. We also separate based on punctuation and consider the words by themselves instead.

The scaling of our data took out a lot of the samples in the Portuguese and English sets, as fewer lines in Spanish meant that we scaled everything down to have an equal number of input lines to avoid any major issues. Finally, our data was cleaned and separated by language by going into the algorithms that we used.

III. Models

Naive Bayes

First, we used Naive Bayes as some baseline to see how our neural models would do. We found that our baseline was actually doing quite well, with accuracy of 96.9 percent and very few general errors on the testing data, and it also trained nearly instantly. This was just done on a very baseline bag of words model that took each word and determined how likely it was to be from a certain language, and then output accordingly. Even if there were some crossover words, this still functioned relatively well because it was able to pick out which language had then most often, so it generally had the best success out of any of the models. Naive Bayes performed by far the best across all languages, as we had a relatively hard time finding significant results or better than guessing algorithms through neural networks. That said, the confusion that did come most in the Naive Bayes algorithm was between Portuguese and Spanish, which was expected, since these two are far more common in syntax than English and Spanish.

I was very curious to see how making the model specifically would influence uncertainty between languages. Thus we implemented NB twice with the language classes being represented differently each time, for example pt: 0 en: 2 and pt: 2 and en: 0. This did actually affect the accuracy, with the latter (with english represented as zero) increasing the accuracy. However, even though it increased the overall accuracy, it decreased the accuracy on stop words between Portuguese and Spanish. This was our most interesting finding on the whole project, showing that the way the model is built influenced performance between languages and that optimizing accuracy is not always the most important aspect of a model.

After cursory exploration using the Naive Bayes model, we noticed that these were mostly coming from the English dataset (which makes sense since the dataset was labeled “mostly English”), but we could not think of a good way to remove only these characters.

Feed-Forward Net

When we moved onto the neural network, we had two things to consider; first, we decided to implement a feed-forward neural network as a base net, then we also wanted to implement an RNN as well. As a note on the timing of the project, we ran into many errors relatively near the due date regarding the output of the neural networks, and we were unclear how to get an accurate prediction out of it; the day of, we had a breakthrough by making the

output layer size 3 (one for each language, which was accomplished through flattening). This enabled us to improve our accuracy from 33 percent (random guessing) to mid-90s percentages, though we did not have a lot of time to improve these in particular. Some modifications that made significant differences were changing the loss function from binary cross entropy to categorical cross entropy, exploring with learning rates, and changing the batch size. We determined after looking at the data enough that although the accuracy more or less leveled out over time, there was relatively little utility in increasing the number of epochs to see its effect, as there was a general clear trend of accuracy going up over time. Even if it were to approach extremely high percentages after a day of training, it would still likely be overtraining the data to extremely large extents. For unknown words, there is no prior towards any particular language, so the feedforward net would predict randomly. Decoding was not necessary except for specific examples, in which we just needed to translate ints to their Language class (0, 1, 2 to to Portuguese, Spanish, English).

For our model inputs, we used a doc2vec (as opposed to a word2vec for previous implementations in class) for the feed-forward net, and did some numpy conversion to both the doc2vec vectors and the gold labels. We used doc2vec to make up for variable length documents; word2vec and padding could have been used to the same effect, but there was relatively similar performance. There was a lot of iteration and now deleted code in this section in attempts to figure out correct input types, sizes, etc., but the final iterations show numpy transformations being done for both training input and labels.

RNN

Overview

We followed an example of binary sentiment analysis using an RNN to model our project off of. However, there had to be changes since our RNN was meant to analyze the order of letters in a word/sentence. Also, we were not completing a binary classification task, but rather one with three outputs.

Preprocessing

The RNN, rather than using words, separates each sentence in a language and separates them by character (represented in unicode). We took the entire training corpus and found all of the unique characters to get the vocabulary for the model. After exploring the vocabulary for a while, we found that there were a good amount of Arabic and Chinese characters, among characters of which we could not identify the language of. We could have removed some of them during the encoding/decoding process (as in the encoding to unicode from utf-8, not index encoding), but this also removed characters we *did* want represented such as letters with tilde.

We took the vocabulary of characters and made an encoding layer that turned all the characters into numerical encoding. There were around 340 characters, plus a space and unknown character that the encoding layer generated. Then we created the dataset of the list of characters in a sentence (per sentence) corresponding to the one-hot vector of its respective language. We did the same but for words totalling in We shuffled and then batched the data into a batch of 64.

Model

For the model we used a SimpleRNN layer with a couple dense layers. The last dense layer had an output of 3, corresponding to the number of languages we were trying to classify. The output layer represents the probability matrix over the three language classes.

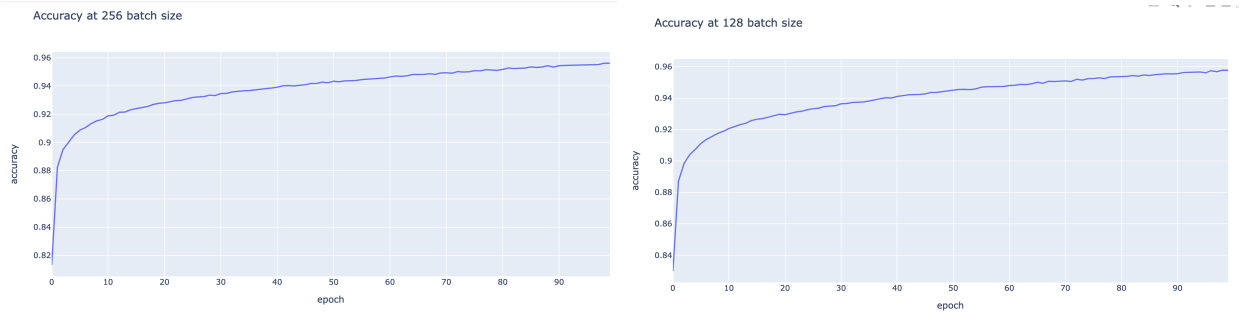
Findings

The model was slow: really slow. It was slow enough compared to the example that we were following to note that we did something wrong, as it was taking 5 minutes per epoch, compared to 30 seconds on the same device. There was not much difference at a code level, and surprisingly, the average length of a sentence for sentiment analysis was much longer than the average length of characters per sentence for language classification.

Although accuracy was very high (including val_accuracy), when doing some small tests like the ones on Naive Bayes, all of the small tests were very wrong. I flipped the order of the test data from Portuguese->English to English->Portuguese, hoping this would help the issue. It did not; the results were still all english.

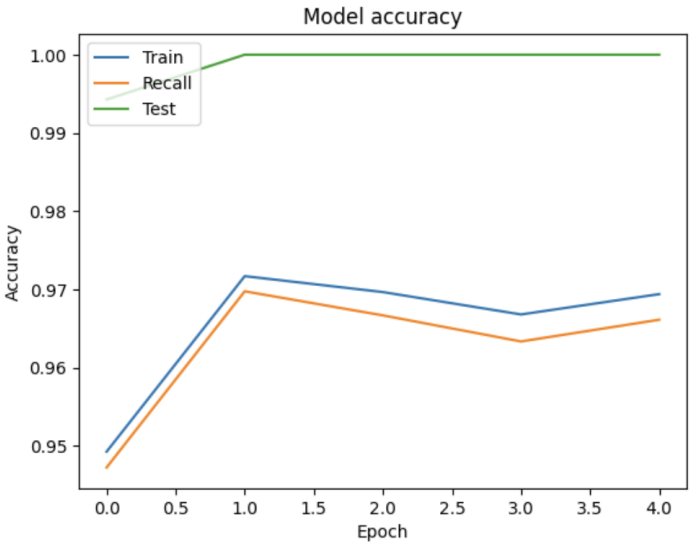
IV. Results & Conclusion

Our results are more or less what we expected. One surprising thing was how well Naive Bayes worked as a baseline model, accomplishing 96.9 percent accuracy as a bag of words model, better than our neural nets. We ended up with results of 95.6 and 94 percent overall accuracy with our feed forward and RNN models respectively.



As can be seen throughout the plots of the accuracy above, regardless of batch size, the accuracy of the feedforward net more or less flattened in the mid 90s. The initial rise happens faster with a batch size of 256, but the algorithm ran about 60 percent faster when using 128 as the batch size instead.

As can be seen in the graphs below on the next page, the RNN shows similar numbers. There is one longer run not graphed that achieved 94 percent, but to show the curve below, we ran it for 10 epochs, which took longer than the feedforward network, and we saw rapid growth in accuracy and decrease in loss at the beginning as expected.



V. Future experiments + explorations

In the future, given more time, we would do a lot more optimization with our models, specifically the neural ones. We're curious about how much changing the hidden layers' sizes and frequencies would affect accuracy and loss, but we did not have time to explore it far in depth.

Additionally, it would be insightful to see how other languages perform in the set and if using more than three languages decreases the likelihood of getting any single one right by a particular margin. Most of our current code is hardcoded to 3 languages, though it could be relatively easily adapted for future explanation.

For the RNN models there are a couple things that we had primed to do, but were not able to execute. First, adding more RNN layers to both the forward and bidirectional RNN's would have been exciting (as well as not having the bidirectional model take 20 minutes per epoch then crash). Also, playing around with the inputs to both models e.g. the average length of characters needed to accurately predict the language, would be as well.

VI. Works Cited

Some code is adapted from a Medium article for technical Python advice. It's linked in the notebook, but it is here as well for convenience: Medium TensorFlow.

RNN:
Example on Binary Sentiment Classification using RNN
Example Character-RNN for Generating the Next Letter
Keras Documentation
How to Make a Dataset using Tensors for Keras Models