Architectural Decisions Document (ADD_V2)

1.0 Introduction

1.0.0 Assessment Guidance for Peers

For peers assessing my project, I have made a roadmap of the relevant sections so that you can grade more easily:

1) Why have I chosen a specific method for data quality assessment?
   - Please see subsections 1.2.4, 1.3.1

2) Why have I chosen a specific method for feature engineering?
   - Please see all subsections in section 1.4

3) Why have I chosen a specific algorithm?
   - Please see all subsections in section 1.5

4) Why have I chosen a specific framework?
   - Please see subsection 1.5.1

5) Why have I chosen a specific model performance indicator?
   - Please see subsection 1.7.1

1.0.1 Contents

This document is organized as following:

1.0.2 Why does this look different to the ADD template?

After consulting the discussion forums on the difference between the ADD template provided by the course and the actual tasks we were set, I decided that creating sections based on the tasks I performed would make for better documentation of my work. In the original template, each subsection is further delineated into:

- 1.x.1 Technology Choice
- 1.x.2 Justification

However, I feel that this is overly mechanical and that it would be unwieldy to separate technology choice from justification. Therefore, I will instead break down my sections based on the smaller tasks that comprise them.

1.0.3 Technology Use

I decided to complete this project in a single Jupyter notebook for a few reasons. My ETL, initial data exploration, and feature engineering steps (i..e steps 1.2, 1.3, and 1.4) were generally integrated and relied on many of the same steps and visualizations. It thus made sense to not only perform these all in the same notebook, but also to intersperse aspects of them, rather than doing them sequentially.

I then decided to perform my model definition, training, and evaluation steps (i.e. 1.5, 1.6, and 1.7) in this notebook too as otherwise I would have had to save my DataFrames and featurization details to IBM's Cloud Object Storage. This seemed like unnecessary amounts of work, especially given that to save files to COS I would need to first make a connection. I would then need to download all of this data again into my new notebook, which I felt was a waste both of time, and of the limited CPU capacity I had for the month.

1.0.4 Why is this the second iteration of the project?

My first iteration (details available in ADD_V1) contained decent quality visualizations, good n-gram analysis and feature vector formation, and resulted in an extremely high accuracy of 99.9%. However, the model reached approx. 90% accuracy on the train set within a couple of mini-batches and was upwards of 95% accurate on the test set after the first epoch. I took this as a clear signal that there was something about the data set which made it extremely easy to distinguish real from fake news articles. Upon further inspection of the data set, I realized that this was indeed the case, as the fake news articles were very clearly distinguishable. As such, although my model worked extremely well on the data set, this would mean its real world performance is yet unknown. Whils this would have been sufficient for the purposes of the course, I wanted this project to have some real world value. Therefore, I decided to search for a new data set and execute the project using that; this is what I have documented in this ADD.

1.1 Data Set and Use Case

1.1.1 Data Set

For this project, I used data from three separate CSV files.[1]These were:

- train.csv = real and fake articles with labels
- test.csv = real and fake articles without labels
- submit.csv = labels for articles in test.csv

I tried to find data contained in CSV files as they in general, they are clearly organized and therefore easy to read data from. As I needed multiple articles with labels for my supervised learning problem, this data set was better than others (e.g. those in .txt format) as it had well-structured data with different feature columns and crucially was well labelled.

1.1.2 Use Case

I used the data to build a disinformation classifier which takes a news article's text as input and outputs a prediction on whether it is real or not.

[1] Data accessed at https://www.kaggle.com/c/fake-news/data, 08/31/20. This data was originally the given data set for a Kaggle competition run by the UTK Machine Learning Club.

1.2 Extract, Transform, Load

1.2.1 Extraction

To extract the data, I downloaded the CSV files directly to my local file system as CSV files openable with Excel. I attempted to use !wget commands to download the data into my Jupyter notebook directly, but this was unsuccessful.

1.2.2 Loading

To load the data, I used IBM Watson Studio's data integration functionality. Once I had saved the data to my local file system, I integrated it into the Jupyter notebook and then used the "insert to code" function to automatically generate code that would read the CSV files into my notebook as Pandas DataFrames. As Pandas DataFrames have lots of functionality for data manipulation, this was a very useful format in which to have the data. For example, the DataFrames format made it very easy to merge and concatenate the DataFrames I got from the CSV files, as well as extract my own DataFrames from them.

1.2.3 Data Cleaning – Part I

First, I located and removed all empty rows as these were not useful inputs. It was important to do this before attempting descriptive stastictis as these would otherwise interfere. I will explain the other data cleaning steps and the rationale behind them in 1.3.4 Data Cleaning – Part II.

1.2.4 Other Transformation Steps

I added string labels to the DataFrames as these would be important for my visualization steps, and also cast all values within the text columns to strings for better manipulation. The other transformation steps (e.g. further manipulation of the DataFrames) only made sense to do at the same time as data visualization. I therefore integrated these steps into 1.3 initial Data Exploration.

1.3 Initial Data Exploration

1.3.1 Statistics

I defined a function "stats" which used lambda functions to calculate word count, sentence count, average word length and average sentence length for an input article. It would then add each of these new values into its own new column in the row of the input article. As such, when applied to an entire DataFrame, the function created these four new values for each row in the DataFrame. I then called this function on both the real and fake news datasets. One issue I had to contend with was that of NaN and infinity values. I read through Pandas' documentation and found a way to convert all infinity values to NaN, and then drop all NaN values.

I also used the interquartile range to calculate the outlier bounds, and defined functions that would automatically bound inputs to my visualization function.

1.3.2 Data Visualization – Part I

Visualizations used:

1) Bar graphs:
- Article count

2) Histogram with best fit line and corresponding boxplot
- Word count
- Sentence count
- Average word length

- Average sentence length

1.3.3 Justifications and Technology Choices

For my visualizations, I imported the seaborn library. Initially, I used matplotlib but found that the code ran extremely slow if I also sued seaborn, so I decided to be consistent and only use the latter. First, I plotted a bar graph comparing the number of real and fake articles, which was useful in checking that the data set was relatively balanced despite removing many more fake articles (due to empty rows). I then defined a graphing function which produced histograms and boxplots for each of word count, sentence count, average word length, and average sentence length. These visualizations were much more effective in this second version of my capstone project as I correctly cropped outliers, allowing for better data spread.

1.3.4 Data Cleaning – Part II

In NLP, it is often very important to clean data before attempting certain forms of visualization. This is because punctuation and stopwords (a set of common words like “a”, “the”, etc.) can interfere with analysis of the text. For example, if you did not remove stopwords and tried to find the most frequent words in an article, you would likely find some stopwords as among the most frequent words, which does not tell us anything about the textual content. However, these cleaning techniques can interfere with attempts to calculate descriptive statistics of textual input, as they can artificially alter word count.

Therefore, before my next visualization steps, I created a function to denoise the text, which removed punctuation, stopwords, and HTML code. These required imports of multiple libraries, as well as the Natural Language Toolkit package.

1.3.5 Data Visualization – Part II

1) Bar graphs:

- Top non-stopwords vs frequency
- Top bigrams vs frequency

2) Word clouds

For my second stage of data visualization, I plotted bar graphs of top non-stowords vs frequency and top bigrams vs frequency for real and fake articles respectively. This was intended to serve as topic modeling. In practice, only the latter worked well in this regard. It is worth noting that both of these were simply forms of n-gram analysis, with “top non-stopwords” really being synonymous with unigrams. I also plotted word clouds for each set of articles as an easy visualization of the most common words. Again, this was less useful as a form of topic modeling than bigram analysis.

1.4 Feature Engineering

1.4.1 Tokenization

Tokenization is an incredibly important step in NLP. I used NLTK’s built-in function “tokenize” to create a one-hot encoding for every word in the combined data set.

1.4.2 Word2Vec Model

First, I installed genism, which is an open-source library for topic modelling in NLP. I needed to import the genism library to create a Word2Vec model, which would allow me to create 100-dimensional feature vectors for every word in the vocabulary (i.e. the list of all words in the articles).

1.2.3 Further Tokenization and Input Bounding

I then used Tokenizer() from the Keras preprocessing library to develop a consistent mapping from the word feature vectors to unique integers. I then bounded the input to a max length (as determined by my outlier analysis), to avoid the skew of the distribution adversely affecting predictive performance.

1.2.4 TF-IDF Featurization

I used the sklearn library to implement TF-IDF featurization, which gives each word in a corpus a score based on the number of times it appears in a single document and the inverse of the total number of appearances across documents. This sort of featurization is very useful for traditional machine learning algorithms.

1.5 Model Definition

1.5.1 Keras Framework Choice

I decided to use tf.keras as my framework as it is very high level and made it easy to implement my convolutional neural network, especially with the embedding layer.

1.5.2 Shallow 1-D Convolutional Neural Network

I considered using an RNN with LSTMs (since the input is a sequence), but ultimately decided on a CNN as this is a many-to-one classification task. As such, I hoped that a convolutional layer would be useful in extracting features early. I was consistent in using the tensorflow.keras library for my layers. It was important to use the Keras framework as this greatly accelerated creating a convolutional model.

I ended up testing a few different models including a 1-D CNN, a simple LSTM, and a 1-D CNN with LSTM. Both networks containing LSTMs trained extremely slowly, and due to my computing power limitations, I decided to stick with the CNN.

Please see 1.7.1 CNN Training for further details about how I arrived at my final model definition. The final architecture was a !-D shallow convolutional network with max pooling and dropout layers (please see notebook for specific architectural details).

1.5.3 Support Vector Machine

To create the SVM model, I used the sklearn library.to create a support vector machine classifier, which I then fit on the training data. I chose an SVM as I had read online that they were very effective at text classification tasks and they also seemed relatively simple to implement.

1.6 Model Training

1.6.1 CNN Training

I used the standard model.fit() and initially trained for just two epochs to check everything was working. Training took a surprisingly long time, which I assume was due to my notebook environment and the long sequence input. However, as discussed before, my convolutional implementation was much faster than using an LSTM.

1.6.2 SVM Training

The SVM ran quite fast.

1.7 Model Evaluation

1.7.1 Model Performance Indicators

To evaluate my model, I used sklearn's "classification_report" which gave accuracy, loss, precision, recall, and F1 score. I chose to use F1 score as my main evaluator since my model is performing classification, so it is important to have a sense of both precision and recall in classification.

1.7.2 CNN Evaluation

Iteration notes on 1-D CNN performance:

On my first training run, I achieved 93.3% train accuracy, with 85.9% validation accuracy, and 85% test accuracy. Therefore, it appears I have a variance problem. I will now consider possible regularization methods:

First, I attempted to use a Dropout layer before the Flatten layer, with a rate of 0.5. Dropout is a standard regularization technique. On my first iteration with dropout, I achieved 91.3% train accuracy, with 86.8% validation accuracy, and 88.4% test accuracy. It appears that the dropout layer had some effect in reducing variance. My F1 scores also improved.

Second, I realized that one way to address overfitting would be to simpligy my architecture. Therefore, I reduced the number of filters from 128 to 32.[2] I also increased dropout rate to p=0.8. I reduced overfitting but test accuracy did not change.

1.7.3 SVM Evaluation

The SVM model worked well, as it had a relatively short run time and yet achieved a high accuracy of around 89%. I did not do any additional iterations as I had already done TF-IDF additional feature creation just for this step.

1.8 Model Deployment

1.8.1 PDF Deployment

I chose to deploy my model as a simple PDF document.

1.8.2 Github Deployment

I will also be uploading my notebook to Github.

[2] I used http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/ to help me understand exactly how a CNN for text classification tasks works.