Project with Ktmine

Jonas Szum

CS 398

Professor Evan McCarty

1 May 2020

Abstract

This paper is a review on Jonas's research project with professor Evan McCarty and the data science department of Ktmine. The main project is a NLP multi-class text classification problem with upwards of seventy possible classes. The corpus used consisted of various contracts, agreements, and other legal documents. Each document could be classified as one or multiple of the given classes. All of the code for this portion was written in Python. My initial approach was to use a basic text classification scheme, TF-IDF, and a neural network provided by Python's Sklearn library. This model was able to achieve an average of 85% accuracy using ten-fold cross validation. After completing the basic model, I created a convolutional neural network using Python's Keras library. This model was able to achieve roughly 95% accuracy after five epochs of training. Next, we decided to pipeline the data from the Keras model into Ktmine's database. We did this using Python's Pyodbc library.

There was a small side project in which we aimed to classify paragraphs as Payment documents, RR documents, both, or neither. This project was done using a Corpus of paragraphs from legal documents, provided by Ktmine.

Explanation: Initial Program

For this project, I was allowed to help Ktmine update their text classification strategy. Ktmine is an intellectual property data & analytics platform company, founded by current CEO Michael Taylor. To update Ktmine's text classification strategy, I was in contact with Michael Taylor and Ktmine's Data Operations Manager Megan Rourke. The specific task at hand was to convert Ktmine's document classification system, with over seventy different classifications, from regular expressions to a machine learning model.

As it stands, Ktmine is using a text-classification system based on regular expressions. The system is able to achieve upwards of 80% accuracy with a false-negative rate reportedly being 0%. For my first attempt at matching this, I started with a basic combination of Python's TfidfVectorizer and sklearn.neural_network libraries. TF-IDF stands for term frequency-inverse document frequency, and works by counting the frequency of a term in one document with respect to the frequency of that term on average, over all documents. For this project, I decided that n-grams of up to a length of three, or three word combinations, would be sufficient to classify these documents. I also only allowed a maximum of 5,000 different features to be extracted through the use of the TfidfVectorizer. After extracting the features from the corpus, I decided to use binary classification for each possible class rather than making the model predict for all seventy-plus classes at once. To ensure accuracy, I ignored classifications that occurred less than fifty times in the corpus. As a consequence, a

vast majority of classifications were avoided because of how sparse they were in the 1,000-document corpus. For the classification of the documents, I used a four-layer, one-hundred-node neural network from the previously mentioned sklearn.neural_network library. This technique resulted in an average accuracy of 85% with a non-zero false-negative rate (about 50% of misclassifications). If the classifications which had less than fifty occurrences were permitted, the neural network might have been trained to predict zero for every document and achieve an accuracy of over 90%. This led me to my next model.

Current model

While 85% was a decent accuracy, I needed to create a stronger model. If I created a more accurate model, I would be able to tune the predictions to lessen (or even prevent) false negatives while still maintaining high accuracy. To do this, I got completely rid of the TfidfVectorizer library as well as the sklearn.neural_network library. To replace the previous model, I used the keras library to find a more complex solution. My new solution would include an embedding layer for the text, an LSTM layer, a dropout layer to prevent overfitting, and a dense layer for the output. Due to the embedding layer, I did not need to vectorize my words. As a consequence, the documents were allowed to maintain their original ordering. This new model trained very slowly, and ended up being more inaccurate than the simple MLPClassifier. I tried to increase dropout rate, and then I completely removed it; but none of these strategies significantly increased accuracy. I decided to change my model from the traditional recurrent neural network (my LSTM layer) to a convolutional neural network; a network usually used for image classification.

My new model had many of the same layers as my LSTM model, with the exception of the LSTM layer itself. As is necessary for any deep-learning, text-classification problem, I kept the embedding layer. Rather than feed the output to an LSTM, however, I fed it to a one-dimensional convolution layer (Conv1D in python). Due to the nature of convolutional neural networks, I had to decide on a pooling layer to take in the Conv1D's output. Having used global max pooling in my CS 512 class, I was

already familiar with it and comfortable with the results it gave. After those two new layers, the final two still remained as a dropout layer and a dense output layer. The dense layer used a binary cross-entropy loss function along with a sigmoid function. I created it this way because I wanted my model to be able to run through a document once and be able to predict all of the labels. The way I had done it with my MLP classifier would have required me to train a new convolutional neural-network for every different possible label. I also used binary cross-entropy because I wanted each label to be classified completely independently from other labels. This was a naive approach because many labels are codependent on each other. I figured the naive approach would simplify my model and lessen the room for error, so I was happy to let each document be classified independently. The convolutional neural-network ended up training a lot faster and being notably more accurate than its predecessor. The accuracy after five epochs of training ended up leveling anywhere from 93% to 95%. Left as-is, the false-positive to false-negative ratio was still about 1:1.

After achieving an accuracy of 95%, I was able to tune the model's output based on its confidence in each label for each document. In the case of these documents, I was told that there should be little to no false negatives. With that in mind, I made a simple variable that allowed the model's label prediction to be adjustable. If the variable was set at 0.1, that would mean that, if the model was even only 10% confident in a certain classification, it would output a 1 instead of a 0. I leave the adjustment of that variable to the professionals at Ktmine. Adjusting the variable to anything other than 0.5 will decrease the accuracy, but also decrease the rate of false negatives (if the variable

is below 0.5) or the rate of false positives (if the variable is above 0.5) while increasing the other's respective rate. The final task for this model is to pipeline it into Ktmine's database.

Side Project with Ktmine

Another small project that will be added to my resume was a simple paragraph classification problem. In short, I was given a dataset that included documents that were separated into paragraphs (each document had a unique ID, and each paragraph was labeled in sequential order), and was tasked with determining whether or not they were payment agreements or record of right (RR) agreements. To do this, I used the same strategy as with my initial model for the main project. This project was given with no particular restrictions on false positives or negatives, so I didn't need to worry about tuning the neural network to accomodate for that. With the simple TfidfVectorizer plus MLPClassifier combination, I was able to achieve roughly a 92% accuracy on classifying which paragraphs were payment or RR agreements. It is also worth noting that while an extremely large portion of these documents were negative for both types, the MLPClassifier did not simply predict zero across the board. The false-negative rate was strictly equal-to or less-than the true-positive rate during my testing runs.

Conclusion and Restrictions

Overall, I'm happy that I got the chance to work with an actual company and apply some of my machine learning knowledge in a productive manner. I would have increased the number of epochs in my keras model, but my laptop takes a couple minutes on each epoch. This is due to the fact that my laptop does not have a GPU, so I'm not able to take advantage of the processing strength that keras is able to provide. While I feel that the model could have increased accuracy if I used more complex parameters, I believe that Ktmine is happy with the 95% accuracy rate as it stands. I appreciate that professor McCarty vouched for me and introduced me to Michael Taylor, and helped me debug my code when I was at the end of my wits.