CS506 Midterm

Runsheng Wang

Upon obtaining the dataset, I performed some initial EDA to identify patterns for each of the columns. There exist a wide range of product ID and user ID, and there’s no particular pattern to the assignment of these IDs in relation to review score. The helpfulness denominator column is 0 for a lot of the reviews, indicating that it is a quite sparse feature. A simple groupby() also shows that there is no correlations between the percentage helpfulness of the review and the score, which is to be expected considering that individual experience of a product can vary quite a lot among different people. The score column is heavily skewed to the right, and there are very little lower ratings compared to 4/5 stars. The distribution of the scores indicates the potential need for a resampling technique, which will be discussed later. By converting UNIX time to datetime format and splitting it into hours, days, months, and years, the different timeframes doesn’t seem to exhibit an apparent correlation with the review scores. The only two remaining columns are summary of the review and text of the review. As I’m planning on using tf-idf to generate feature spaces, and that all other features seem to be a lot less relevant in predicting scores, I dropped the other columns.

Before proceeding with anything else, I realized that the dataset is very large. To test for small implementations and functions efficiently, I decided to generate a random sample whose size is 1% of the original training set. Using this model as validation might not be very informative, but it allows me to quickly check if I’ve implemented something correctly.

Before passing the text into tfidf vectorizer to extract features based on the frequency function, there are several things that need to be done, and the order in which these are performed is crucial to the success of the algorithm. First, NA values in training is removed, because there’s no way to correctly impute reviews, and that this portion of the data is rather small. One way that could potentially be implemented is to use synonyms based on star rating (e.g., if the review has a score of 5, generate synonyms based on another review with score 5). However, this process requires tokenization and cleaning first, and it contradicts with our goal on preprocessing. Thus, the small number of NAs are simply ignored.

Given that the dataset is skewed, I performed a custom under-sampling technique, where I kept all data from star rating 1 and 2 (their sizes are very similar), as well as star rating 3 (whose size is roughly twice of one star/two star). I reduced the sample size in 4 stars and 5 stars to two times the number of samples from one star/two star in order to match the sample size of star rating 3. After this resampling, three, four, and five stars all have a sample size that’s roughly twice the size of the one star sample. This dataset is still skewed, but it does reduce the original skewness while maintaining some properties of the “nature” of the review distributions. Since there would naturally be more data in the 3, 4, and 5 star category from any random sample, it is worth keeping some degree of the original characteristic of the dataset.

Pandas stores strings as Objects, but nltk and other text processing languages works on strings. Before passing into the text processors, I converted the string Objects into strings. My goal is to perform lemmatization before I pass the strings into the tfidf vectorizer. In order to lemmatize, I need to tokenize my strings. There are some built in tokenizers in the packages, but they all fail to accurately address noise in the input string. Thus, I built my own pipeline for string preprocessing.

First, I decided to remove punctuations, special characters, and white spaces. Punctuations could be useful in analyzing sentiments/emotions, but implementing it correctly is very complicated, and processing could take very long. Thus, I skipped this step and removed all punctuations except apostrophes. Why did I keep apostrophes? I wanted to identify stop words correctly, and to do that I need to keep the apostrophes for future processing. If I removed the apostrophe from “don’t”, then the word becomes “dont” and it is no longer in the stopwords dictionary. To address this, I first removed all punctuations except apostrophes, then I replaced apostrophes with a single space character. Then, I performed tokenization/split based on the whitespace character. This way, I correctly split stopwords into their sub-formats.

Since I want to do lemmatization, I need to tokenize and maintain my tokenized text as close as the original text as possible. Since lemmatization benefits from POS (part-of-speech), I can’t just remove stopwords while tokenizing. Instead, I should do them after in order to improve the performance of the lemmatizer.

I experimented with spacy lemmatizer, but it is very slow compared to the NLTK WordNet Lemmatizer (I was able to optimize the execution time of the wordnet lemmatizer manually through dictionary and function calls, but those approaches are not possible on spacy). By default, the lemmatizer takes in an input string and tries to lemmatize it, so if you pass in a word, it will lemmatize it treating it as a noun by default. It does take the POS tag into account, but it doesn’t magically determine it. To make the lemmatization better and context dependent, I would need to find out the POS tag for each word using NLTK and pass it on to the lemmatizer.

After tokenizing and lemmatizing, I removed stopwords and any words with a length below 2 (no meaning). I optimized this step by caching the stopwords prior to checking and using a function on each iteration of checking so that memory is freed as soon as a stepwise computation is complete.

After doing all the steps above, I’m ready to input my word into the tfidf vectorizer. I created two separate vectorizer for summary and text, because the amount of information contained in the two features are different. Review text is considerably longer and contains more words and combination of words, so there would be more features. On the other hand, summary is shorter and there would be less words and combination of words. A brief google search reveals that a person uses 5000-6000 words on a daily basis, which can be thought of as common words. I’m performing unigram and bigrams on the datasets. Since the text data contains some noise, using the daily commonly used word threshold of 5000 would be appropriate, whereas for the shorter summary text, I only need a few important word from it, so I will use 20% of the commonly used amounts.

I’m also using two other parameters: max\_df and min\_df. Min\_df is set at constant 5 to filter out text data that are misspelled or occurs too infrequently but are not filtered out by stopwords. Max\_df is set at 0.9, or 90%, to filter out words that appears in more than 90% of the documents. This is to double check that stopwords are indeed removed correctly, and it also removes certain words that are not stopwords but are irrelevant in uniquely identifying/help identifying review scores.

I’ve generated a lot of features, or a quite sparse text matrix. Too many feature is bad for model fitting because it almost always results in overfits. To counter this, I used LSA on the text data, which is essentially PCA on the sparse matrix. I want to extract words that are representative of a specific score. For example, if “terrible” only appears in one star reviews, then the basis of the SVD transformation would contain a pivot column corresponding to the word “terrible” and/or its close synonyms. My truncatedSVD uses 3% of the actually generated feature space.

After generating feature space, I performed train-test split on the dataset, and applied a random forest to the training set. I used random forest because other methods are either too time consuming/memory consuming to run, or that they are incredibly difficult (slow) to tune via grid search methods. For instance, SVM typically consumes a lot of memory space. Naïve Bayes relies on the conditional independence assumption. XGBoost is powerful but extremely time consuming to tune because of the number of parameters associated with it. I used random forest with 5 parameters. The n-estimators is the number of trees in the forest. The more this number, the more the ensemble. Ideally, we want this number to be as high as possible, but actual practices depends on the hardware specs. I found that it runs in a reasonable time with n =1000, and performs much better than the n =100 default. The n\_jobs = -1 speeds up the process by allowing the random forest to access all cores of the CPU, reducing its runtime. The oob\_score use out-of-bag samples to improve generalization accuracy, and the class\_weight parameter further balances out our imbalanced training data. The random state variable makes the process reproducible.

After training the model, I applied it to predict the validation set and measured its RMSE. I tuned the input data characteristics as well as the random forest tree according to the output of the RMSE, without touching the submission data. After consulting with the confusion matrix multiple times, I finalized my parameters and generated the submission file for output. I found the most time consuming step to be lemmatization, which was to my surprise. I imagined that model training would take a long time, but turns out finding word bases is a much more difficult job