Group Members: Branden Diniz, Xuanyu Li, Dennis Silva, Chuxin Wei

**DS 501 Case Study 3: Textual Analysis of Movie Reviews**

**Section 0 – Introduction and Motivations**

Business Intelligence is a technology driven process that creates efficient and rational decision-making processes through the gathering, exploring, interpreting and analyzing of data. One takes information from the business itself, its users or consumers, in additional to other internal systems and external sources to generate economic growth. This case study about analyzing text data of movie reviews reflects this type of process. Automated sentiment of text documents can be quite useful to a company in a variety of forms. In particular, two forms stand out. First, a company gains the ability break down or drill into reviews specific to topics of interest. For movie reviews, one could look for whether people would rate a movie high or low in overall performance. One could also look specifically, for example, about peoples opinion on the level of CGI used or whether the actors fit their role well. Secondly, automation of text sentiment would be less costly to a company who has to manually analyze individual sentiments. Of course a good company would use some level of manual analysis, but an automated analysis of the data gives a close to real-time representation of what is being talked about. Trends in movie reviews can be spotted earlier and actions can be taken quicker to attempt to sway public opinion about a specific movie. Sentiment analysis is not perfect, but it is a useful tool for a company to gain insight into opinions of their products or services by changing a subjective review into an objective stance.

For this case study our group used movie review data commonly used in sentiment analysis experiments. Particularly we used the ‘polarity\_dataset\_v2.0’ which includes one-thousand positive and one-thousand negative pre-processed movie reviews released in June of 2014. The overall goal was to train a system coded in Python to distinguish a movie review as either a positive or negative one only given the textual data from the review. Information such as individual review history or user/movie demographic data was not used in this system. We conducted our analysis of this data specifically in order of the problem outlines described in the Jupyter case study 3 notebook. As such, each of the following sections reflects an individual problem number of the notebook.

**Section 1 – Completion of Exercise 2: Sentiment Analysis on Movie Reviews**

In order to begin familiarizing ourselves with sentiment analysis and programming with scikit learn, we were first to utilize the scikit learn *Working with Text Data* tutorial for analyzing a collection of text documents. The tutorial itself provided us with a baseline for understanding how to download a data set of textual documents, create training and testing data sets, extract features, train a classifier, build a pipeline, and tune our parameters to better the evaluation of system performance. From the scikit-learn source code, we utilized the skeletons of solutions to exercise 2 of the tutorial to solve the exercise within the Jupyter notebook. At times we also used the code provided in the solutions section of the source code when we became confused about how to proceed. We loaded into python the ‘polarity\_dataset\_v2.0’ and created training and testing subsets of it. We decided to do a 75%/25% split of the entire dataset to create these training/testing subsets. In addition, we wanted to set a random state or seed when generating our subsets so that each individual group member could work with the exact same randomly divided data. In total we had two-thousand total reviews, fifteen-hundred training reviews, and five-hundred testing reviews. Within the training reviews there were 745 negative and 755 positive reviews. Within the testing reviews there were 255 negative and 245 positive reviews. Although we did some initial testing of the classification system over a few different machine learning algorithms to make sure our notebooks were in order, we did most of our parameter and algorithm testing in the later problems.

**Section 2 – Explore the scikit-learn TfidVectorizer class**

In order to transform the textual information stored in a set of documents into numerical data, one can use the *term frequency-inverse document frequency* (tf-idf) of a vocabulary to create feature vectors for each document. The term *term frequency* (tf) of a specific term in a document is the number of times that token occurs within that document. Although tf can reflect importance of some words, it poorly reflects that of others. The word ‘the’ may appear several times in a text document, but it also appears several times in most other text documents. The descriptive nature of the word ‘the’ with respect to a single document is therefore lackluster. Thus, one also incorporates *inverse document frequency* (idf) into the mix. Idf is a factor that, when incorporated, diminishes the weight of a term that occurs very frequently within a document and increases the weight of terms that occur rarely. In other words, it is inverse function of the number of documents in which the term occurs. Tf-idf is the product of tf and idf, and is, within this case study, a better reflection of the relevance of individual terms within the reviews. Within scikit-learn, there exists a class called ‘TfidVectorizer.’ In short this class is meant to help convert a collection of raw documents into a matrix of tf-idf features. Since many of the tf-idf values within any given document will be zero, the matrix returned is sparse. This means rather than storing the data in a 2d-array like object, Python stores it more like a list of coordinate pairs and values. This is done to avoid extensive memory and storage waste.

When TfidVectorizer is used a series of raw documents, it creates features from the words of the documents specified by parameters passed in. Two of these parameters are ‘min\_df’ and ‘max\_df.’ When building a vocabulary, all terms with a tf strictly less than the cutoff provided by min\_df are ignored. If min\_df is an integer, all terms that occur in count less than min\_df are ignored. If min\_df is a float between 0.0 and 1.0, all terms that occur less than min\_df percent of all documents will be ignored. This is same for max\_df, but rather all terms strictly greater than max\_df will be ignored. To explore the effects of changing these two parameters on the features obtained, we fixed on parameter to its default and varied the other. These results are depicted in the tables below.

|  |  |  |  |
| --- | --- | --- | --- |
| min\_df | Number of  Features | Number of  non-zero elements | Sparsity (%) |
| 1 | 35,542 | 500,167 | 0.938 |
| 3 | 16,047 | 475,692 | 1.976 |
| 4 | 13,172 | 467,067 | 2.364 |
| 5 | 11,233 | 459,311 | 2.726 |
| 10 | 6,715 | 429,698 | 4.266 |
| 25 | 3,137 | 375,598 | 7.982 |
| 50 | 1,694 | 325,931 | 12.83 |
| 100 | 844 | 267,074 | 21.10 |

Table 1: Feature table with fixed max\_df = 1.0 and 1,500 total documents

|  |  |  |  |
| --- | --- | --- | --- |
| max\_df | Number of  Features | Number of  non-zero elements | Sparsity (%) |
| 1.0 | 35,542 | 500,167 | 0.938 |
| 0.9 | 35,528 | 479,637 | 0.900 |
| 0.5 | 35,465 | 415,039 | 0.780 |
| 0.1 | 35,000 | 270,344 | 0.515 |
| 0.05 | 34,419 | 208,865 | 0.004 |
| 0.01 | 30,922 | 95,990 | 0.002 |
| 0.005 | 27,606 | 60,153 | 0.001 |
| 0.001 | 14,515 | 14,515 | 6.666e-4 |

Table 2: Feature table with fixed min\_df = 1 and 1,500 total documents

In addition recording the number of features and non-zero elements, we also recorded the sparsity. *Sparsity,* sometimes known as matrix density, is the number of non-zero elements divided by the total number of elements capable of being stored in a non-sparse version of the matrix. Each matrix has 1,500 rows, reflecting each document. The number of columns is the number of features, and varies from matrix to matrix. One can see from the tables above that by fixing max\_df and increasing min\_df, overall sparsity increases. Although the number of features decreases, the amount of non-zero elements also significantly decreases. The remaining features are those which more documents have a non-zero tf-idf value, i.e. they are significant to more documents. One can also see that by fixing min\_df and decreasing max\_df, the sparsity decreases. The number of features decreases at a much slower rate than the number of non-zero elements. Features that occur in many of the documents are removed, and thus more unique features to each document are kept. Sparsity in of itself is not a direct indicator for future performance over different machine learning algorithms. Sparsity can help find the correct balance in desired features to non-zero elements, and therefore help with parameter tuning and estimation.

Another parameter of TfidVectorizer is ‘ngram\_range.’ This parameter is tuple indicating the range of n-values for different n-grams to be extracted from the documents. These n-grams, or terms, are collections of one or more words within the document. For example, a unigram would be a single word, and a bigram would be a pair of words. All values of n such that it is within the closed interval [min(n), max(n)] of ngram\_range will be considered. Thus, by increasing this range one can increase the number of features taken from the documents and, hopefully, find additional meaning or relevance in them. We also tried different ngram\_range values, as seen in the tables below, but due to the computational limitations of our computers we only tested a few pairs of values.

|  |  |  |  |
| --- | --- | --- | --- |
| ngram\_range | Number of  Features | Number of  non-zero elements | Sparsity (%) |
| (1, 1) | 35,542 | 500,167 | 0.938 |
| (1, 2) | 436,474 | 1,371,924 | 0.209 |
| (1, 3) | 1,200,436 | 2,300,740 | 0.128 |
| (1, 4) | 2,096,905 | 3,236,768 | 0.103 |
| (2, 2) | 400,932 | 871,757 | 0.145 |
| (2, 3) | 1,164,894 | 1,800,573 | 0.103 |

Table 3: Feature table with fixed min\_df = 1 and max\_df = 1.0, and 1,500 documents.

From Table 3, one can see that as we increase the range of the ngrams the overall sparsity decreases. As more word combinations are added as features more and more documents fail to have those combinations. Although the number of features increases, many of them are relevant to a very few documents. It is interesting to see that having ngrams ranging from (1, 4) actually creates a matrix with the same sparsity of when using a range of (2, 3).

**Section 3 – Machine Learning Algorithms**

In this section we test out different supervised classification machine learning algorithms. We were to pick parameters we believed to be optimal based from problem 2, however we found it difficult to gauge the performance of the different algorithms from the feature matrix alone. We decided to first do a grid search over a select number of parameters pertaining to TfidVectorizer through LinearSVC. However, we found that these parameters were not near-optimal for other algorithms when they were tested. Thus to give each algorithm a chance in deciding which one classified our reviews the best, we included various parameters of TfidVectorizer when performing individual grid searches for the parameters of each machine learning algorithm. Besides the two classifiers described in problem 3, LinearSVC and KNeighborsClassifier, we also tested a DecisionTreeClassifier and Lasso. The best results over the various tested parameters are given in the table below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Testing Set:  neg = 255  pos = 245 | **TfidfVectorizer**  **Parameter Set** | | | **Unique**  **Parameter Set** | | **Best Score** | **Confusion Matrix of**  **Testing Set** |
| ngram\_range | max\_df | min\_df |
| **Linear SVC** | (1, 2) | 0.7 | 4 | C | loss | 0.850 | [[215 40] [ 25 220]] |
| 15 | 'squared\_hinge' |
| **KNeighbors Classifier** | (1, 1) | 0.8 | 4 | n\_neighbors | weights | 0.693 | [[166 89] [ 77 168]] |
| 9 | ‘distance’ |
| **Decision Tree** | (1, 1) | 0.9 | 4 | max\_  depth | max\_  features | 0.623 | [[152 103] [ 87 158]] |
| 14 | ‘sqrt’ |
| **Lasso** | (1, 1) | 0.6 | 30 | alpha | | 0.274 | [[154 101] [ 48 197]] |
| 0.001 | |

Table 4: Results with greatest score for each machine learning algorithm tested

Out of the four classifiers tested, Lasso did the worst with a score of 0.274. Since Lasso provides coefficients with classification, we needed to round these coefficients to either 0 or 1, negative or positive. To achieve even this score, we needed to increase min\_df substantially from the commonly used 4 all the way to 30. The DecisionTree classifier did the next best with a score of 0.623. Compared to Lasso, the DecisionTree performed better at predicting positive reviews but poorer at predicting negative ones. We needed to limit the maximum number of features, max\_features, used in each decision of the tree. We tested the log, square root, and the number of total features as this limiting factor. Log limited the tree too severely and using all total features slightly lowered overall performance. Since these were the least performing algorithms, we focused most of our attention on the best two.

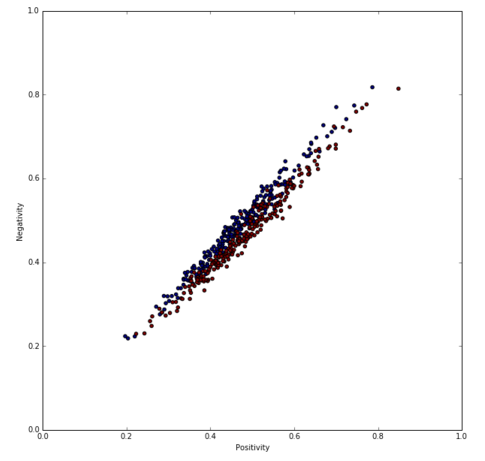
The top two scoring algorithms were LinearSVC and KNeighborsClassifier. For KNeighborsClassifer, as we increased the number of neighbors, the overall score increased. Due to limitations in computing power, we did not test neighbors above 9. This trend may continue as we increase the number of neighbors, but would likely reach a horizontal asymptote in performance fairly quickly. In addition, we tested two different form of weighting functions, uniform and distance. Uniform gives equal weights to all neighbors while distance gives weights inversely proportional to the distance between voters and the original point. It turned out that using distance to weight the points lead to better performance.

LinearSVC did better with an ngram\_range of (1, 2) while KNeighborsClassifer used (1, 1). LinearSVC performed better with some of the additional features of the bigrams. Max\_df under LinearSVC seemed to decrease since not all of these additional bigram features were relevant and hampered performance when compiled on one another. In the solution provided by scikit-learn, LinearSVC had a parameter C set to 1000. C is the penalty parameter of the error term, and should be fairly small for this type of data. We thus tested very small C values, the best being 0.001. We also tested different loss functions, hinge or squared hinge. Hinge is the default loss function for the SVC class, and squared hinge is the square of the hinge loss value. Squared hinge provided better performance.

LinearSVC is a common choice for text classification as well as for other support vector machines. SVC is able to handle these high dimensional feature spaces generated by the tf-idf conversion of the terms within the documents. Because of this, we need not remove features that are deemed ‘to irrelevant’ due to computational complexity. Although LinearSVC was the best performing algorithm, it did make a few mistakes. For example, document ‘cv915\_8841’ is a positive review, but was classified as negative. It becomes apparent when reading the review that it is filled with sarcasm and seems to have more negative word choice despite being positive. For example, the first sentence of the review reads, “what do you get when you combine clueless and dumb and dumber?” In another example, the algorithm predicted the document ‘cv524\_24885’ to be positive, but was actually negative. The movie itself is said to be filmed very well by the reviewer. However, the plot was not the review author’s cup of tea and was therefore originally classified as a negative review.

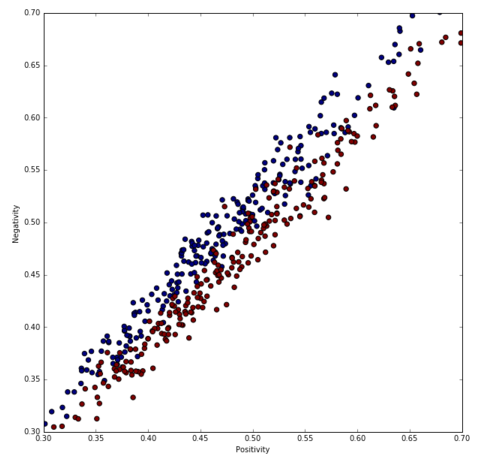
**Section 4 – Open Ended Question: Finding the Right Plot**

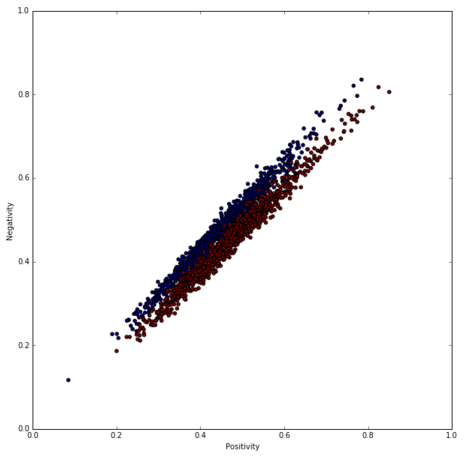
Being an open ended question, our group had many different ideas of how to go about answering problem 4. At a very basic level, our original concept was to think of words or phrases that occur in more positive reviews and those that occur in more negative reviews. For example, a term such as “great acting” or “awesome” may appear in more positive reviews. Originally we thought of counting the number of times these sets of words occurred in the document, but then we had a better idea. Instead of trying to think of the words, or features ourselves, why not use the machine learning techniques in combination with the classifications we already generated. We first divided all documents into a new 75%/25% split of training/testing data. Using the TfidfVectorizer class, with our optimal parameters found for LinearSVC, we created a vocabulary using all features as terms. In total this created a vocabulary of 45,139 terms or features. We then split the training data into two subsets, positive and negative reviews, based on the known classifications. We then used the TfidVectorizer class again to convert the documents into two tf-idf weighted document-term matrices using the vocabulary found prior. We used the overall vocabulary since some terms may only occur, or be relevant, in one type of review. From there we took the mean of each feature over all the documents for both matrices. This created two arrays which we used as the coefficients of the tf-idf weighted feature values produced from the TfidfVectorizer, using the same vocabulary, over the testing set. We then summed each of these new feature values within each document to create two new coordinate arrays we titled ‘Positivity’ and ‘Negativity.’ In essence, we converted testing documents into tf-idf weighted features and used those to understand what makes up positive and negative reviews individually. Rather than words like “awesome” being counted and made into an axis, some other collection of the 45,139 features closely resembling other positive or negative reviews are being combined. Some example graphs are given below. The division line is the line where positivity = negativity. If a document has more positivity then negativity, most seem to actually be a positive rating and vice-versa.



Plot 1: Plot of 500 testing documents separated by positivity and negativity values we   
defined above. Blue indicates a negative document, and red indicates a positive document.

Plot 2: Plot of 500 testing documents (zoomed in for clarity) separated by positivity and negativity   
values we defined above. Blue indicates a negative document, and red indicates a positive document.





Plot 3: Plot of 1,500 testing and training documents separated by positivity and negativity   
values we defined above. Blue indicates a negative document, and red indicates a positive document.