Motivation & Objectives

The objective of our project Review Rater is to create a classifier that can be used to predict the rating of a Yelp review as either “useful” or “not useful”. To do this our team has identified the relevant attributes in the Yelp Dataset necessary for the completion of the project, these being the review text and the number of “useful” votes that this review in particular has received.

Our rationale for pursuing this particular project came from thinking about not just what Yelp is, but also what Yelp would find useful for its business model. Yelp is a site that gathers user reviews for local businesses, these reviews can in addition be voted upon by other users according to various metrics such as “useful”, “cool”, etc. Naturally a visitor who is looking for reviews on a particular business would want to see the most relevant and useful reviews for that business first. This was the idea behind our project, to apply data mining and machine learning techniques to find correlations among the text of user reviews that are deemed “useful”.

If we could indeed find some common characteristics among useful reviews, these could be used to build a model that could predict the usefulness of a given number of reviews. In our initial design we worked towards the goal of classifying reviews as either “useful” or “not useful”. As we continued work on our project we found this binary classification a bit too simple to give meaningful or interesting results. After some more thought we decided to take a slightly different track to our problem.

The end goal of this model was changed to be able to not only predict the usefulness of a review but also to rank these reviews in a descending order of usefulness. This decision was made with the thought of the average Yelp user in mind, when searching for a business review the average user is only interested in the usefulness of a review in relation to the other reviews on the same page, and not necessarily with the “absolute value” of a review’s usefulness. By ranking a business’ reviews in a descending order of usefulness we could ensure that a visitor would see only the most relevant and interesting reviews first.

The majority of our project code will be in Python 3.4 and we will make use of the NLTK library for our text processing and manipulation. Our project website will be hosted on Github pages and will be written in HTML and JavaScript.

Data Tasks

Preparation

This project presented a number of challenges on the road of implementation. Our first task during the initial planning stages was to inspect the data set to identify promising data that could be used to find the results we were looking for. Since our project was focused on rating and ranking user reviews this particular task was accomplished rather easily. We decided to use the JSON file containing all information related to user reviews, including review text, date, ratings, business ID, etc.

However even after deciding to use the user reviews data set we encountered our second challenge, namely the sheer size of the data set which consisted of over 1GB of pure text data. This consisted of billions of JSON encoded data objects. Our next task was to once again narrow our search space in such a way that the smaller domain would not impact our eventual results. We did this by focusing only on the user review text and the total number of times a given review was voted as “useful”. We used the number of “useful” votes to determine if a given review has the quality of being “useful”. This number was used to divide the data set into two subsets consisting of a set of useful reviews and a set of not useful reviews. This number, the “useful” threshold, being of a somewhat arbitrary nature was changed across multiple program executions in order to compare multiple sets of results and to help tune the final program’s performance.

Finally, after separating useful reviews from the rest of the data set we must do one more task before starting our data analysis. We must separate these useful reviews into both training and test sets so that our program can have data to with which to build a model and data against which we may test our model.

Analysis

Now that we had two sets of data to use for our program development we set on the task of designing an algorithm and its associated methods. We brainstormed various methods that seemed promising enough to test, some of these methods in the end worked better than others. Since our major point of analysis was text reviews all of our methods and algorithms are text-centric.

In using text reviews to rank usefulness our team was making the implicit assumption that written language, and its semantic meaning, can be correlated with “useful” reviews. In theory this would mean that certain words, combinations of words, types of words, and semantic categories such as food, location, descriptions, etc. would be found in “useful” reviews vs. non useful reviews.

This alone gave us many promising avenues to pursue during the course of our project. Could a term in a review such as “chicken” be more indicative of a useful review vs. a term such as “table” or “chair”? Would “chicken” appear more often in useful reviews vs. non useful reviews? In an extension of the previous idea, could semantic categories of words appear more often in useful reviews? Maybe word categories such as “food” (chicken, beef, fish, etc.), “price” (cheap, expensive, tips), “location” (downtown, city, local) appear at noticeably different rates in useful reviews. Another aspect we looked at was word classifications such as nouns, adjectives, and verbs; perhaps useful reviews had different ratios of these word classes as well. These and a few other ideas helped to guide us in our implementation.

Methods

Class Review: This is an object for storing a review text, its associated score and its rating against all other document vectors in the training set. This was so it would be

easier to keep all of the review data together so as to keep it together for analysis after computation.

removeNonAscii(string): This method was used to remove non ASCII characters from a review text. This became a necessary when non ASCII characters appeared on occasion in the review text, breaking certain NLTK functions. This works by going through a given string character by character and only copying a char if it has a value less than 128.

divyData(file, testFile, trainFile, corpusRoot): This method is used to divide our chosen reviews into both test and training sets. It starts by opening two files for writing and going line by line through the supplied data set. If a given review had a number of useful votes above a certain threshold then it was chosen to be used in our program. The chosen review is then assigned to either the test or training set on an alternating basis, ensuring that both sets were equal in size. This function was changed a multitude of times so as to generate new sets for testing with our ranking algorithm.

tokenizeDocument(string): This method takes a string and creates a list of tokens from it. It uses the RegexpTokenizer() method from the NLTK to accomplish this, then returns the list of tokens.

removeStopWords(list): This method takes a list of tokens and removes the stopwords from it. It steps through the supplied list, tests if the current token is in a list of stopwords and either adds it or doesn't add it to a new final list of tokens. This final list of tokens is returned.

stemTokens(list): This method stems a given list of tokens using the PorterStemmer from the NLTK. It steps through the supplied list and applies the PorterStemmer to each one and adds them to a new list, this new list is then returned.

stemToken(string token): This method stems a single token using the PorterStemmer before returning this stemmed token.

getIDF(string token, dictionary): This method calculates the inverse document frequency of a given string token with respect to the supplied dictionary of document vectors.

normalizeVector(dictionary): This method length normalizes a given document vector to unit length. It returns a length normalized unit vector.

docDocSim(dictionary, dictionary): This computes the cosine similarity between two supplied document vectors. It returns a similarity score generated by summing all of the cosine similarities between a review to be ranked and all of the reviews in the training set.

convertToTFIDF(dictionary of dictionaries): This takes a dictionary of dictionaries which contain term counts for multiple documents and finds the TFIDF scores for each

respective document and stores them. A dictionary of normalized TFIDF scores are then returned.

ConvertReviewToTFIDF(string, dictionary): This method computes the TFIDF score os a supplied review text string using the TFIDF scores of the training set of documents.

score(dictionary, dictionary of dictionaries): This method takes a TFIDF vector of a given review and computes the cosine similarities of this TFIDF against all other TFIDF vectors in the training set. It then sums all of these similarity scores into a single number.

getTermFrequency(list): This method takes a list of tokens and computes the frequency of each term in the list.

processReviews(): This method computes the TFIDF vectors of all the reviews in the training set of reviews.

Algorithm

As for the actual algorithm and techniques that we used, the first step was to access and read the data set relevant to our program which is the yelp reviews data set. Our main() method opens this file and prompts the user with an option for dividing data into training and test sets. If affirmative then the divyData method is invoked which opens the data set and reads it line by line, if the number of “useful” votes exceeds a certain threshold, defined as the USEFULTHRESHOLD constant, then these reviews are alternately written to either the test or training set text files. The purpose of divyData is to find useful reviews and to assign these reviews to either the test or training set.

Next the processReviews method is called to perform further processing on the training set of data. The processReviews method starts by opening the training set file and then begins to process the file line by line. For each line that is read the method loads it into a dictionary using the json.loads() method, from the dictionary the review text is extracted. Next the text is tokenized using the tokenizeDocument method which implements the tokenize functionality of NLTK. Stopwords are then removed from the token list using removeStopWords method and finally the token list is stemmed using the stemTokens method; the previous methods making use of NLTK functionality. The next step is to call the getTermFrequency method which parses the token list and updates a dictionary of dictionaries (trainingReviewsTF) with the term frequencies for the given review token list.

Finally, after all the lines of the training set have been processed the final method to be called is the convertToTFIDF method, this method steps though trainingReviewsTF and calculates the document vector for each entry. This vector is then length normalized before being stored in trainingReviews and being written to the objectStorage.json file.

To summarize, the core functionality of the script is to put all reviews in the corpus into normalized TFIDF vectors. Some of these are in a training set full of only useful reviews with a yelp usefulness rating exceeding a certain threshold, and some of them are in test

sets. The cosine similarity between a review in the test set and every review in the training set is then calculated and summed together so as to make a sort of "score." The lower this score, the better, as we believe that the more a review has in common with a set of "useful (reviews that exceed the threshold)" reviews, the better it likely is. We then sort these reviews from lowest score to highest – with the lowest score review supposedly being the best.

Testing

To test our implementation we made use of the following algorithms and methods. To begin our program opens the file of test reviews and proceeds to read this file line by line. Next our program follows the same algorithm and uses the same methods to compute the TF-IDF vector of the test review as it did when computing the TF-IDF of the training reviews. Each TF-IDF of a test review is then added to rankedReviewList along with its "score", computed using the score() method. This method gives a review vector a score by comparing its cosine similarity with every other document vector in the training set and summing the results.

Next all of the ranked reviews are ordered in place and the length retrieved. All ranked reviews are divided into quarter sets and the median value of each quarter set is retrieved, this being one of the metrics of review text similarity in each quarter. Also, the usefulness scores provided by yelp are added together to find the overall usefulness of the reviews in a quarter of the testing set. Final results are then written to an output file for later examination. Our hope being that there would be an observable trend in the medians and sums of usefulness in each quarter so as to indicate that the ranking algorithm was effectively ranking the test set.

Evaluation

Our implementation changed a fair deal during our project development from our initial plans. Initially we planned to make use of K Nearest Neighbors as the main technique for classifying reviews as "useful" or not useful. However as we continued development the goal of the project changed from classifying reviews to ranking them in relation to each other. We decided that using K Nearest Neighbor to rank reviews instead of classifying them did not make logical sense due to the fact that we had no real way of determining whether a review was really actually bad as opposed to good. This caused us to change our implementation to using document vectors to rank reviews using the scoring method.

We also ran into a few problems during implementation such as certain methods in the NLTK library that did not work correctly on Python 3.4 as the only stable version of NLTK was designed for 2.7. Occasionally our text parsing methods ran into issues with certain non ASCII characters that showed up every now and then in user reviews, this required us to implement some remove ASCII methods to ensure there were no further problems. Since these non ASCII characters were often used for decorative affect or

emphasis we were confident that they had little real semantic meaning or influence on the review's usefulness votes.

One thing we did find out during our test runs was that shorter reviews had a tendency to be ranked higher by our program than longer reviews. This made some intuitive sense to us since we reasoned that the shorter a review was, the more likely it was that it contained more useful information according to a TFIDF normalized vector comparison - Whether it was discussing the quality of service, prices, food, and other things that visitors would be interested in reading about. One of our early results spit out a ranked review file that contained two very different reviews as number one and 2. The rank 1 review gave a glowing recount of a good time had at an establishment while the next review in the ranking was a terribly negative review of a business, it was interesting to us that the top two ranked reviews in our run was a positive and negative review, this gave hope that we were on the right track due to the fact that they both intuitively seemed quite useful, but contained such different language.

We now realize that this disparity in word use that was a great portion of the reason why we were unable to obtain favorable results. With such different language being considered useful, our method was sure to hit a brick wall without a great deal of breakdown of each review. With negative words like "hate and terrible" as well as "love" and "amazing" all being used in useful reviews, we found it very difficult to get favorable results.

We continued testing our program, but the results were in general more mixed. While there were a few bright spots that seemed to prove our algorithm, most results showed no real distinguishable trend. This was a disappointment to us and led us to believe that with some more tuning and maybe the addition of some other data mining techniques we could create a program that may give us some more promising results.

Deliverables

Our Github repository contains all of our source code and project website, in addition to some text files that stored some of our intermediate data and results.

InterestingOutputExample.txt and Notes.txt contain some of our intermediate program results and some thoughts and interpretations of what we learned. ReviewRankerCore4.1.py contains our initial implementation that we ultimately did not follow through to the end with. ReviewRanker5.0FINAL.py contains our final program that was used to test against test reviews and from which we gained our ultimate findings. Term Classification.py was used to classify terms in both the training and test sets of data according to their grammatical purpose, this program classified terms as nouns, verbs, etc. This information was used to create some infographics and was meant to be used in further improving our program but ended up being unused.

Term Frequency.py was used to find to total term frequency in both training and test sets. Finally we have the last text files, term_classes.txt contain the results of running Term Classification.py. Both termfreq.txt and unstem_freq.txt contain results from

Term Frequency.py, the first contain stemmed terms and the last contains unstemmed words; this was used in creating a word cloud of most common words.

Finally we have both testSet.json and trainSet.json which were our testing and training sets for our program respectively, at any given time.

The project website can be found at this URL: http://l1felock.github.io/ReviewRater/