Report of K-Nearest Neighbors on Amazon Reviews

Kosei Matsuda
Miner2 ID: koki12340
Mason ID: G01034285
Best Submission: .80

Introduction

For my approach, I first started with importing the packages I needed and familiarizing myself with the packages to use them efficiently. To prevent wasting my time and to be efficient, I made sure to know when and where I would use each package. I imported nltk to aid in manipulating the text data, making use of the stop words list as well as SnowBall stemmer in order to refine my text to create a more efficient and effective dictionary. I also used sklearn to use TfidfVectorizer as the most effective way to measure each document is through term frequency. Cosine similarity was used as well, since it was in my best interest to compare similarity rather than magnitude. Lastly, I utilized Counter in order to identify which words were most common within the document.

Handling of Text Data

When handling text data, it was my goal to manipulate and refine my text data as much as possible to create an effective dictionary. What I first did was open traindata.txt, converted it into a list of list of words, where each list represents a single review. I wanted to remove noise and break down each word to its root. One step was to remove stop-words, which are terms that occur frequently and have little impact on overall sentence meaning. Nltk.corpus already has a set of stop-words that I used in order to identify the stop-words within the data. Next goal was to stem my document, which is the process of altering each word to its root form. This was done by iterating through every word in the data set and converting it to its stem word by utilizing SnowBall stemmer from the nltk package. This greatly reduced the number of unique words found in the data, as similar words such as “lovely” and “loving” will end up grouped together in the form of the word “love”. I then implemented a dictionary called text_dictionary that kept a list of all the different words included in the now altered text data and the number of times each word was mentioned in the data. This was done by adding each word found in the document as a key and the value as one. If the key/word already exists within text_dictionary, I would update text_dictionary by adding 1 to the value, symbolizing the count. This allows me to identify the extremely rare occurrences of a word by identifying which word has a count of only 1 or 2. I then iterated through the dictionary, found which words occurred than less 3 and added it to a dictionary called better_dictionary. Then I made a new dictionary that did not include these words.

Bag of Words vs TF-IDF

My goal was initially to implement Bag of Words methodology to get an idea for this project. This is where we can represent a document as a vector or a string of numbers. The idea is that we can represent every individual document as its own vector with each index representing a word from the dictionary and the number of occurrences of that word. This was done by creating a 2D list of 14999 * 19,000 where each list represents a document and its corresponding indexes represent a word and the value inside the count of the word. I parsed through every document, checked if each word was in our dictionary and then incremented the value within the index by 1, giving us word count. This is expensive for our time complexity if we wish to parse through it and extract the specific values. Additionally, there were many 0s, which results in an incredibly sparse matrix. Also, we do not retain the ordering of the words in the text which may be significant as well. Identifying all these points led me to use TF-IDF. It is a measure calculated by multiplying the number of times the word appears in a document by the the number of documents that contain that word. This model gives each term more information, as it gives each one a measure of weight. For example, if a term occurs very frequently but can also be found in every document, then one can conclude that the term has no real impact in classifying a document. I was able to do this by using TfidfVectorizer. By already defining a dictionary, I simply have to give TfidfVectorizer my training data to retrieve vectors weighted by TF-IDF.

Cosine-Similarity vs Euclidean Distance

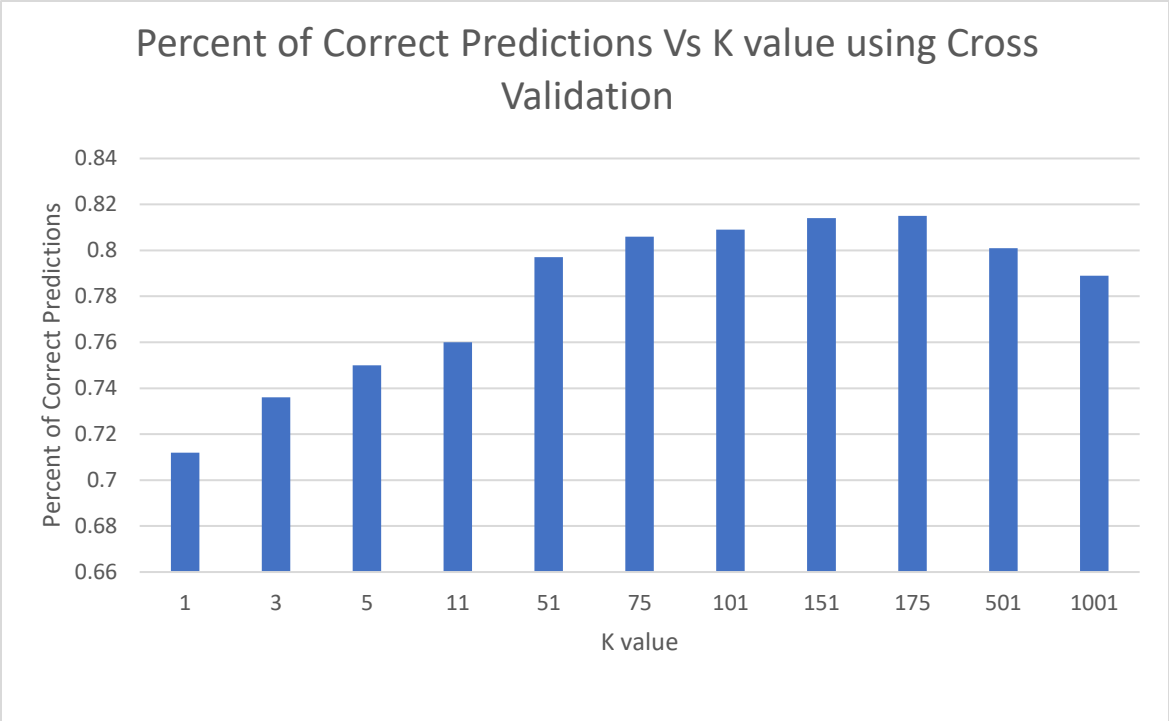
Since I selected TF-IDF, I therefore had to do cosine similarity, which measures the similarity between two vectors within a product space. This is done so by measuring the cosine angle of two vectors and determines if they are both pointing the same direction. This means that it is a measurement of direction/orientation rather than magnitude. By measuring the angle, it makes the size of our document irrelevant as distance is no longer a factor. It will take the dot product of two vectors, divided by the product of both vectors. Given the size of our document and the sparse matrix as its result, it is in our best interest to avoid Euclidean Distance. For example, one document could feature the word ‘chocolate’ 60 times while appearing 10 times in another, making them far apart by the Euclidean distance. However, it is possible for them to have a small angle between them, implying a similarity. I was able to calculate the cosine similarity by plugging in the tf-idf vectors into the imported cosine_similarity.

Using the cosine_similarity function returns a numpy that contains information on each document and its cosine similarity score for each word within that document. Afterwards, I iterate through every index the numpy, where every index represents a single test document. I will then call argsort()[-k] which will order the values from the smallest magnitude to the greatest magnitude. [-k] will pull the greatest magnitudes up to k times to make a list. This list will consist of values that represent the index of the document that has the greatest cosine similarity within the training set. I will then retrieve the score of each document and add them all together. If I end up with a positive number, then I know that majority of the similar documents

had a score of 1, so my test document can be classified as 1. If I have a negative number, then the majority of similar documents had a score of -1, letting me know to classify my test document as -1.

Experimentation

In order to select a value of our k parameter used in our k-nearest neighbor algorithm, I utilized cross validation. For every instance that I tested a k value, I randomized the given data set by using a tool online that shuffles a text by line. ("https://onlinerandomtools.com/shuffle-lines") It uses the Fisher-Yates shuffle algorithm, where it will start from the last element/line in the text data and swap it with another line selected by random number generated from 1 to the number of lines. It will then move on to the next element, but with the random number used being 1 to number of lines – 1, increasing the 1 every time we iterate to the next element. This will be repeated until the first element is reached. I then took the newly shuffled data set and split it into 10 different groups of equal size. Then for each group, I removed it from the data set so that it will act as a test data, and then trained the remaining data set. I then used my k-nearest neighbors algorithm on the test data, retrieving scores for the data. Once I have the scores, I compare it to the scores given the text data and calculate the number of correct scores over the total number of scores. I repeated these steps for each of the ten groups and then calculated the average of the correct result percentage.



I then selected different values of K within cross validation in order to retrieve the best value for our K. I would start with two extreme values of k, such as 1 and 1001 and document the corresponding percent correct of my score. I would then work my k values towards the

middle of both extreme values and try to find the positive trend in order to arrive at my desired k value.

Another way I experimented was by choosing how many “rare” words would be removed. This is a fine line, as the rarer the word, the more effective it can be in weighing a document based on td-idf. For example, if the word “gobbledygook” occurs only 10 times within the total text data but is found within only 2 reviews, then that word will be incredibly useful in comparing those two documents as it holds a great weight. After testing many different values, cross validation showed that it was best to remove words that occur less than 3 times.

Additionally, I created a function that takes in a dictionary and prints the n most common words by using a Counter on my dictionary. I did this so I could see the most common words and make snap judgments on whether a word holds any weight. If I suspect that a word may not be impactful, then I will remove it and cross validate. By doing this, I was able to identify words such as “get” and “day” and add it to my list of stop-words. With this process I was able to add a couple percentage of accuracy to my overall correct percentage.

Conclusion

Upon review, there were a couple choices that could’ve been made to improve my accuracy or efficiency. It was possible to decrease the size of my dictionary by identifying which terms were “neutral” in nature. This could be done by considering the number of times a term was associated with a positive and negative review. If the number of both positive and negative scores for a term were equal, then it can be deducted that the word has minimal weight and therefore could be removed. If a word has a high tf-idf score but can be considered “neutral”, it might give a notion of similarity between documents when in actuality has no real impact, possibly considering neighbors that it should not.