Mai Vi Vuong
Sept. 26, 2021
CS484 - 001

Identifier: username5
Rank & accuracy score: .80 for k=75

Overview:
In this project I wrote the K nearest neighbor classifier from scratch to predict the sentiment of 15,000 movie reviews. Below is the report that contains implementation details, as well as reasons why I chose certain methods over others.

Preprocessing:
The training data set text file had 14,999 reviews, one review on each line with its corresponding sentiment of -1 or +1. I read in this training text file using the readlines() function, which puts the reviews in a list – each review separated by a comma. I spliced the sentiment of each review and appended it into a separate list for later reference.
Before performing any type of vectorization or KNN, we must preprocess the whole corpus. Preprocessing was a major step and significantly impacted the accuracy of the KNN algorithm. The main goal of preprocessing is to normalize the data and to reduce unnecessary features - for example lemmatizing the corpus so that 'chocolates,' 'chocolatey,' 'choco,' map to its base form 'chocolate.'

- **Punctuation & html tags**: I remove punctuation and HTML tags in the corpus because they don't offer any meaningful features, and will increase computation and speed efficiency. I use regular expressions, which is a special sequence of characters that match a given pattern, to help remove these features in the corpus. The Python module re provides full support for regular expressions in Python, which makes removing punctuation and html tags easy and fast.
- **Lowercase**: I made all of the text in the corpus lowercase using the built in Python function lower(). Lowercasing in the words is important, because the vectorizer recognizes 'CHOCOlate' and 'chocolate' as two completely different words, which we don't want.
- **Stemming vs lemmatization**: Stemming is the process of reducing words to their root form, by chopping off the end of the word. This sometimes results in root words that are not in the English language. Instead of stemming, I decided to lemmatize the words in my corpus. Lemmatization reduces words to their base word in the dictionary, which ensures that the root word belongs to the language. I chose lemmatization over stemming because it is more sophisticated, and I believe to be more accurate. For example, if I were to stem the word saw, it would return just as 's'. Lemmatization would return either 'see' or 'saw' depending on whether the use of the token was a verb or noun.
- **Stop words**: stop words are commonly occurring words in the English language that don't offer any additional value to the document vector. Removing these will increase computation and space efficiency. I used the NLTK library method to download the common stop words in the English language, and iterated over my corpus to remove any stop words.

TF-IDF:
After preprocessing is done, we need to turn each test and training review into its numerical representation - which we need to measure similarity between each review. This process is called text vectorization.
There are many ways to vectorize text data, for this model I chose to use TF-IDF vectorization. I will discuss why later. The TF-IDF matrix makes the observation that some words appear naturally more than others which can have an undesired effect on analysis. For example, in the context of our movie reviews, the word 'review' could come up a lot in all the reviews. on the other hand, the word John Cena may not appear much, but it's highly informative when it does. TF-IDF quantifies words in a set of documents by computing a score for each word to signify its importance in the document and corpus. It works by increasing proportionality to the number of times a word appears in the document, but is offset by the number of documents that contain the word.
TF-IDF stands for term frequency (TF) * inverse document frequency (IDF). TF is how many times a word appears in a review, IDF find how common or rare the word is in the entire corpus. A high TF-IDF calculation is reached when we have a high term frequency in the given review (local parameter) and a low document frequency of the term in the whole corpus (global parameter). When a term's TF-IDF value is high, it means that term is significant in the corpus.

- **TFIDF Vectorizer**: TFIDF Vectorizer creates a set of its own vocabulary from the entire set of documents.
- **Vectorizer fit_transform([]):** this function takes in an iterable (in this case, a list) which contains the corpus to learn the vocabulary and idf, and returns the document term matrix. I use this function to fit my training data set.
- **Vectorizer transform:** this function only transforms a corpus (in an iterable data structure) into a document term matrix.
- **Count vectorizer:** Count Vectorizer uses the bag of words methodology to calculate the importance of each word based on how often they appear in a document. This is ineffective and meaningless because common words such as review which appears many times in all documents, would be considered highly weighted. if you follow this method with TF-IDF Transformer, the result will just be TF-IDF vectorizer.

K Nearest Neighbors (KNN):
The K Nearest Neighbors classification framework requires two major steps. First the inductive step to produce the classification model using our training data (which includes the expected sentiment). To create this model, I normalize the training data and the test data using the preprocessing methods above, then I vectorize each of the training and test reviews using TF-IDF. I calculate the cosine similarity between each of the training reviews to each of the test reviews creating a 14,999 by 15,000 matrix of distance values between each review.

Cosine Similarity:
I chose to calculate the distance between each of the test reviews x training reviews using cosine similarity over Euclidian distance. Cosine similarity measures the cosine of the angle between two vectors (the two reviews) in a multidimensional space. The angle represents how many unique words each review has in common – an orthogonal angle between the two vectors would mean no words in common (distance of 0), whereas

colinear vectors would mean two reviews using identical words (distance of 0). Magnitude between the vectors is mostly irrelevant, some reviews may be significantly longer than others – that doesn't mean they're entirely dissimilar in content - which is why Euclidian distance would have been a poor choice to calculate document similarity.

The second step is our deductive step, where we apply the model onto our test examples, and see if our model produces the correct sentiment for each review. The K nearest neighbors of a given review refers to the K training examples that are closest to the given test instance.
To find the K nearest neighbors for each test review:

- `sort_and_tuplize(cos):`
  First, I transform each test row in the cosine similarity matrix into a list of tuples that contain the (column) index of the corresponding training review and the cosine similarity distance. Afterwards, I sorted each row by distance to find the training reviews that are most similar. The purpose of tuplizing each distance value was to preserve the (column) index value of the corresponding training review after sorting each row.
- `get_knn(sorted_sim, k):`
  Sorting each row by distance allows me to grab the K nearest neighbors very easily and quickly using basic list splicing in python. Looping through each row of the sorted matrix I grab the K nearest neighbors of each test review and place them into a new matrix.
- `get_sentiments_knn(knn_tuples, rating_arr):`
  I loop through the K nearest neighbor matrix (each row index represents each of the 15,000 test reviews, each element in the row is a tuple that represents a k nearest neighbor), using the preserved column index to map to the corresponding training review sentiment (contained in the rating list, which was spliced at the beginning of the file). I append the sum of the k sentiments to the output list.

End Notes:
Overall, this project was the most fun I've had in the last year. Although it was difficult, because everything from Python to vectorization to classification was all new to me, watching the project come together was extraordinarily rewarding. There are some changes that I would like to make had I had more time for this project, such as optimizing my code to have less nested for loops to decrease runtime and potentially take in larger datasets.