Homework 1: Movie Review Classification

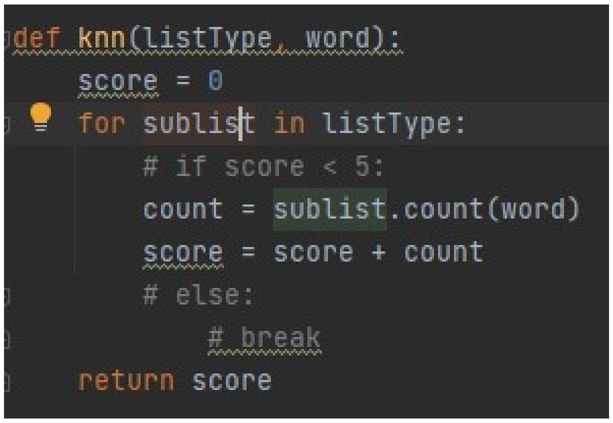
Siddharth Patel (minerID - spatel53)

I. Rank & accuracy score for my submission

As of writing this report, my current rank is 195. Moreover, my submission score is 77%.

II. Initial KNN Approach

Before I discuss how I advanced to a brighter implementation, I want to discuss my biggest hurdle when I first implemented this project. Without much knowledge of data mining, I started my project with pre-processing which I was able to accomplish fairly quickly. As I proceed to the next step of implementation, I was unsure of how to vectorize my data from words into weighted integers. Moreover, I did not have any bright ideas regarding the approach to KNN. With the fear of wasting too much time on the thought process, I tried to improvise by skipping vectorization and distance calculation, steps which are considered a must to generate an optimized KNN algorithm (steps are to be discussed below).



```python
def knn(listType, word):
    score = 0
    for sublist in listType:
        # if score < 5:
        count = sublist.count(word)
        score = score + count
        # else:
            # break
    return score
```

Fig. 1. Initial KNN Implementation

As you can see above, this was my first KNN implementation. All I did to calculate my scores was traverse through each of 14999 words and compare it to each individual word in the testing file after pre-processing.

The problem with this implementation was that it was extremely slow. Each run took 5-7 hours to compile successfully. If there are any errors in the code then you won't know until you are half-way through the run time defining this code as a waste of time. Finally, I admit that my initial implementation was hard-coded which clearly led to poor accuracy fluctuating from 50% to 60%.

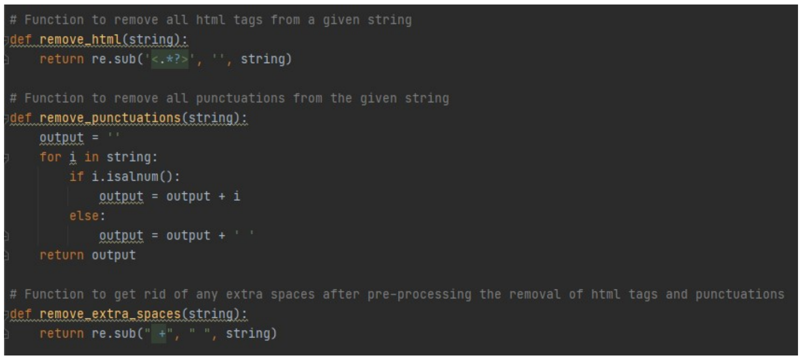
To add on to my poor decision making above, I used stemming rather than lemmatization in initial implementation. Let's say that your document contains two words after pre-processing, "car" and "car". The former word defines a vehicle while the latter is the stem of the word "care". Reverting a word to its stem caused issue. I later fix this issue in my redesigned implementation.

Now that I shared my biggest blunder, lets talk about the improvements I made with my newly refined implementation which will be the focus of this article moving forward.

III. Final KNN Approach

Moving on from my biggest failure, this article will share how I implemented a a refined KNN implementation with steps discussed in class to better improve the performance and accuracy of the given data.

A. Text Pre-Processing

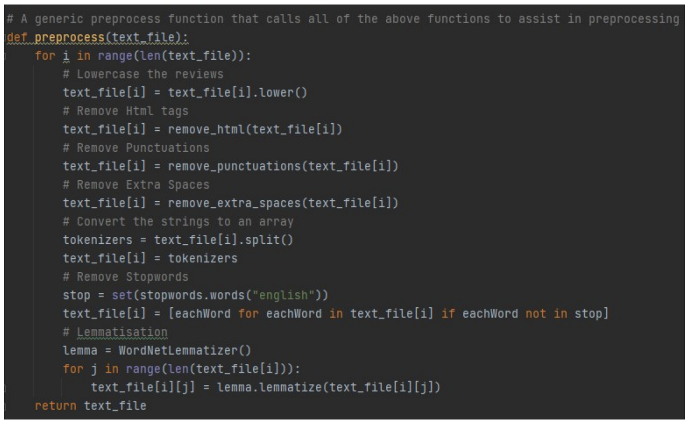


```python
# Function to remove all html tags from a given string
def remove_html(string):
    return re.sub('<.*?>', '', string)

# Function to remove all punctuations from the given string
def remove_punctuations(string):
    output = ''
    for i in string:
        if i.isalnum():
            output = output + i
        else:
            output = output + ' '
    return output

# Function to get rid of any extra spaces after pre-processing the removal of html tags and punctuations
def remove_extra_spaces(string):
    return re.sub(" +", " ", string)
```

Fig. 2. HTML Tags and Punctuation Removal

My first approach to implementing this algorithm was to remove HTML tags followed by punctuation. I then went on to remove any extra spaces that had been left behind during the removal of tags and punctuation. I chose this approach to ensure that when I convert the same words into weight integers, the TF-IDF algorithm (will be discussed later) will consider them the same preventing the increase in number of features. For example, if I have "tiger" and " tiger", I want these words to be referred as similar even though one has an additional space. Hence, I decided to remove any extra spaces. This would also assist me in improving the KNN's performance since there will be less features generated. For this section I mainly used the regex import in python.

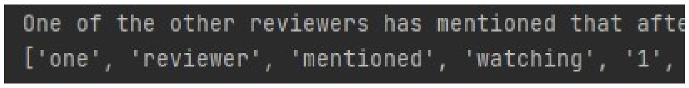
My next step was to remove stopwords. One can think of stopwords as a set of commonly used words in any language. In English, some common stopwords are this, there, he, she, and I. Such words hold no weight when comparing your data. Therefore, it is a common practice in Data Mining to remove them. To remove all the stop words, I used the stopwords library from nltk.corpus.



```
# A generic preprocess function that calls all of the above functions to assist in preprocessing
def preprocess(text_file):
    for i in range(len(text_file)):
        # Lowercase the reviews
        text_file[i] = text_file[i].lower()
        # Remove Html tags
        text_file[i] = remove_html(text_file[i])
        # Remove Punctuations
        text_file[i] = remove_punctuations(text_file[i])
        # Remove Extra Spaces
        text_file[i] = remove_extra_spaces(text_file[i])
        # Convert the strings to an array
        tokenizers = text_file[i].split()
        text_file[i] = tokenizers
        # Remove Stopwords
        stop = set(stopwords.words("english"))
        text_file[i] = [eachWord for eachWord in text_file[i] if eachWord not in stop]
        # Lemmatisation
        lemma = WordNetLemmatizer()
        for j in range(len(text_file[i])):
            text_file[i][j] = lemma.lemmatize(text_file[i][j])
    return text_file
```

Fig. 3. PreProcess Function

Thereafter, I initially stemmed my data, but I immediately saw that the problem with stemming is that your words are cut back a whole lot more in comparison to lemmatization. Here is an example: lets say you have a word, "caring"; when stemmed, that words turns into "car" since stemming uses the stem of the word. On the other hand, lemmatization converts the word into "care" because it uses the context in which a word is used. As a result, I immediately switched to utilizing lemmatization rather than stemming. This was without a doubt a leading factor to an increase in accuracy by 15%.



```
One of the other reviewers has mentioned that afte
['one', 'reviewer', 'mentioned', 'watching', '1',
```

Fig. 4. Pre-processing Output

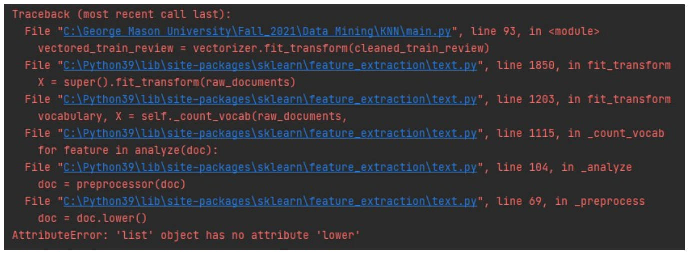
Overall, I packed my text pre-processing algorithm into a generic function which can be called for processing, both, by the test and train data (shown in fig. 3). This function would take each string in the list, pre-process it, and set it to that particular index it was extracted from in the form of an array. Notice that before pre-processing, our data is a list of strings, but after processing, my data was a list of lists.

Figure 4 depicts a glimpse of how the data looks like after pre-processing. You can see that the HTML tags, punctuation, spaces, and stopwords have been removed from the data. Moreover, the data has been lemmatized.

B. *Vectorizing*

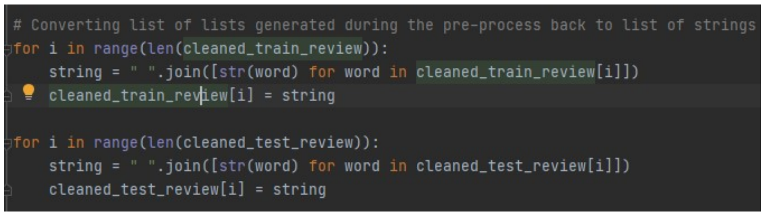
Once I had cleaned up my data, I decided to use TF-IDF vectorization over Count Vectorization since the count vectorizer only returns the number of occurrences a word appears in the document. On the other hand, TF-IDF considers overall document weight per word giving you floating numbers. This means that TF-IDF is more precise. Because of this I chose TF-IDF over count vectorization.

When I was implementing TF-IDF, I ran across an attribute error (shown in fig. 5) which required me to turn my pre-processed data into a list of strings (shown in fig. 6). From there, implementing TF-IDF on test and train data



```
Traceback (most recent call last):
  File "C:\George Mason University\Fall_2021\Data Mining\KNN\main.py", line 93, in <module>
    vectored_train_review = vectorizer.fit_transform(cleaned_train_review)
  File "C:\Python39\lib\site-packages\sklearn\feature_extraction\text.py", line 1850, in fit_transform
    X = super().fit_transform(raw_documents)
  File "C:\Python39\lib\site-packages\sklearn\feature_extraction\text.py", line 1203, in fit_transform
    vocabulary, X = self._count_vocab(raw_documents,
  File "C:\Python39\lib\site-packages\sklearn\feature_extraction\text.py", line 1115, in _count_vocab
    for feature in analyze(doc):
  File "C:\Python39\lib\site-packages\sklearn\feature_extraction\text.py", line 104, in _analyze
    doc = preprocessor(doc)
  File "C:\Python39\lib\site-packages\sklearn\feature_extraction\text.py", line 69, in _preprocess
    doc = doc.lower()
AttributeError: 'list' object has no attribute 'lower'
```

Fig. 5. Vectorization Error



```
# Converting list of lists generated during the pre-process back to list of strings
for i in range(len(cleaned_train_review)):
    string = " ".join([str(word) for word in cleaned_train_review[i]])
    cleaned_train_review[i] = string

for i in range(len(cleaned_test_review)):
    string = " ".join([str(word) for word in cleaned_test_review[i]])
    cleaned_test_review[i] = string
```

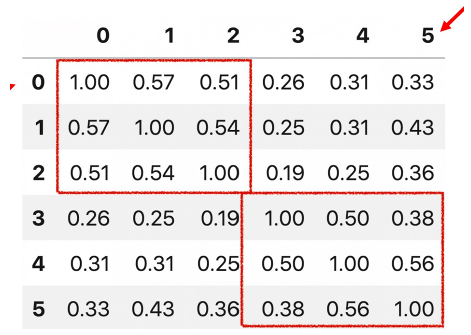
Fig. 6. Solution to Above Code

was simple. I used fit_transform function on train data and transform function on test data. The reason for doing this is because when you call the fit method, it is actually calculating the mean and variance of each feature present in our data. The transform method is transforming all these features using the respective mean and variance. If you call fit_transform again on test data, you will reset the respective mean and variance.

To accommodate the next step, all data generated during Vectorization was converted and stored into a data frame.

C. *Finding Distance*

To find the distance between the test and train data, I used the cosine similarity rather than euclidean distance because you are normalizing your data in a way rather than just calculating the distance between two vectors, leading to better results. This process generated a 15000 x 14999 matrix consisting of distance from each word in the test reviews to the train review. The output would look similar to what in shown below.

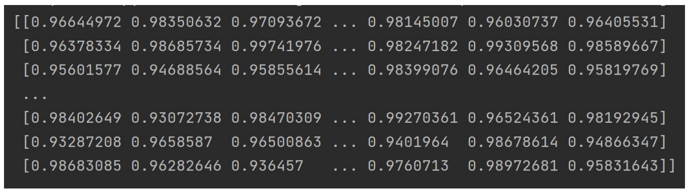


	0	1	2	3	4	5
0	1.00	0.57	0.51	0.26	0.31	0.33
1	0.57	1.00	0.54	0.25	0.31	0.43
2	0.51	0.54	1.00	0.19	0.25	0.36
3	0.26	0.25	0.19	1.00	0.50	0.38
4	0.31	0.31	0.25	0.50	1.00	0.56
5	0.33	0.43	0.36	0.38	0.56	1.00

Fig. 7. Distance

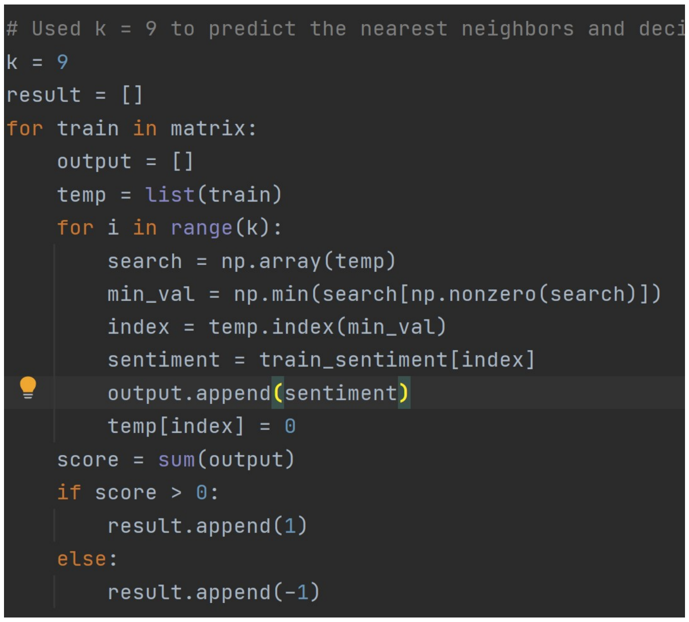
D. *KNN Implementation*

Finally, my last step involved calculating the nearest neighbors. To accomplish this task, I simply traversed through the



```
[[0.96644972 0.98350632 0.97093672 ... 0.98145007 0.96030737 0.96405531]
 [0.96378334 0.98685734 0.99741976 ... 0.98247182 0.99309568 0.98589667]
 [0.95601577 0.94688564 0.95855614 ... 0.98399076 0.96464205 0.95819769]
 ...
 [0.98402649 0.93072738 0.98470309 ... 0.99270361 0.96524361 0.98192945]
 [0.93287208 0.9658587  0.96500863 ... 0.9401964  0.98678614 0.94866347]
 [0.98683085 0.96282646 0.936457   ... 0.9760713  0.98972681 0.95831643]]
```

Fig. 8. Cosine Similarity Output



```
# Used k = 9 to predict the nearest neighbors and deci
k = 9
result = []
for train in matrix:
    output = []
    temp = list(train)
    for i in range(k):
        search = np.array(temp)
        min_val = np.min(search[np.nonzero(search)])
        index = temp.index(min_val)
        sentiment = train_sentiment[index]
        output.append(sentiment)
        temp[index] = 0
    score = sum(output)
    if score > 0:
        result.append(1)
    else:
        result.append(-1)
```

Fig. 9. KNN Algorithm

generated matrix searching for the minimum value in the array. I would then use the index of the minimum value to search for the correlating sentiment in the training data and append it to a newly built array. This process would continue for k amount of time. Once I had finished looping through k, I would sum up the newly built array returning either +1 or -1. To avoid getting 0 I purposefully chose K = odd_value. The calculated score was appended into an array called result (shown in fig. 9). If the sum was positive then the test review was considered positive. If the sum was negative, the test review was considered negative.

E. Returning KNN

To wrap this code up, I traversed through the result array and appended each score per line into a file. I later uploaded this file to miner to verify my accuracy.

IV. Conclusion

Since this was my first machine learning project implementation, I spent more than 40+ hours writing code and 30+ hours understanding the whys and hows behind the implementation. To conclude this project, I would like to share that this project may have been the hardest. Throughout this journey, I faced many decisive challenges along with a huge learning curve that hindered my performance. With time and patience, I was able to successfully generate a working code with an accuracy of 77%. A practice that I had picked up from Object-Oriented programming that I commonly used throughout this project was converting all my data into either an array or a string. Moreover, to overcome the lack of python syntax, I used various resources throughout this project strengthening my knowledge of python.