Movie Review Classification Report

Venkat Sai Charan Bandi
G01219773

Miner UserName: Cherry_SLYR

Rank	User	Submission Time	Public Score
405	Cherry_SLYR	Sept. 15, 2020, 4:17 a.m.	0.71

Instructions to the TA

I attached my PyCharm Project folder as src, it contains three main files:

1. preprocess.py -> has the code for import, data cleaning and pickle files
2. algorithm.py -> has the code for running the large vector and cosine calculations
3. Value_tune -> contains the final code for knn attributes and features and then pushing it into the final format.dat file

Project Walkthrough

During writing of this project,
I underwent and implemented multiple choices and combinations of algorithms, some i discarded and some i kept for the final code. I will explain all the thoughts i had while writing this code and i divided my code into three blocks of execution:

1. Preprocess Data Clean
2. Data Point collection
3. KNN Calculation

I will explain my code in the same way, during this document.

1. **Preprocess Data Clean**

    - Importing and converting the data to a suitable format in this case, using the Pandas library to store versatile large arrays with multiple columns. And using Python lists to iterate through individual movie reviews to make string manipulation easier
    - From here i'm running both Train Data and Test Data through a Data Clean Function that i wrote that does the following:
        - I observed there are HTML Tags that were in the Data accidentally so i started by removing all the
 tags in the data
        - Then converting all the text into lowercase
        - Removing words containing numbers, punctuation, brackets, and nonsensical text

- STEMMING OR LEMMATIZING
    - Reading about normalization Techniques, i read up about type of text normalization, the most famous being the ones mentioned above
    - I Ran the code with both and analysed that STEMMING showed better results. Since it cuts down the tail end.
- Data Storing
    - I then used external libraries, such as “PICKLE” that serializes custom objects and saved the two for the next chunk of code

2. Data Point collection

- I noticed this part of the code is the one with the most amount of unskippable runtime as the current dataset takes in an O(n^n) times loop. So this one is solely for the purpose of this data looping.
- To save continuous runtimes, i created a list of array that will store the similarity values, so i can pick and store the sentiments and features in the later chunk
- While doing so i explored multiple similarity functions:
    - Fuzzy Wuzzy package that implements Levenshtein distance
    - Cosine similarity
    - Jaccard index
    - Word Mover’s Distance (WMD)
- I picked cosine similarity as it seemed most appropriate given we weren't considering or optimizing computing.
- The cosine similarity function, i could be using a library, but to try and understand the algorithm better i used a stackoverflow answer that implemented the algorithm from start (https://stackoverflow.com/a/15174569)
- The final steck is to store this N-array and pickle them into later use. The Runtime is very bad and i could’ve made it better with sparse matrix or inbuilt libraries but this is a from scratch code and hence didn't.

```
14997  [[0.350314365610564, 1], [0.33094680639450197, 1], [0.31435546275338544, 1], [0.30903849271230344, 1], [0.3023382558493013, -1]]
14998  [[0.6064282497741627, 1], [0.5445024598988236, 1], [0.5392800025540302, 1], [0.5344545363391198, -1], [0.4707919090691997, 1]]
14999  [[0.36757352205735255, -1], [0.3332135132551981, -1], [0.33266962253515925, -1], [0.32033479218618655, -1], [0.3164915089868014, 1]]
--- 22129.646807432175 seconds ---

Process finished with exit code 0
```

3. **KNN Calculation**
    - The final step was to iterate through all of this 15000 calculated cosine similarity values. This gave me the opportunity to run the code with changed attributes and features and varying ‘N’ in less than 10 secs.
    - The question to answer here is picking the right size of ‘n’, i found 4 or 5 to be a better fit after running it on MINER and used the same in the final code.
    - What sentiment to select finally:
        - I experimented with weighted selection and divided the weights with (0.5 for the most similar, 0.3 for the next similar, 0.2 for the next) and so on.
        -
```
val0 = float((i[0][0]) * int(i[0][1])) * (0.5)
val1 = float((i[1][0]) * int(i[1][1])) * (0.25)
val2 = float((i[2][0]) * int(i[2][1])) * (0.1)
val3 = float((i[3][0]) * int(i[3][1])) * (0.1)
val4 = float((i[4][0]) * int(i[4][1])) * (0.1)
```
    - The final step was to pick the sentiment from the weighted values and push the code into a .dat file to upload on miner.