README

1. Scoring and Term-Weighting

Preprocessing steps

- Removed escape sequences from the text and replaced them with space
- Converted to lower case: converted the text to lower case using the python lower() function
- Removed punctuations: remove the punctuations by removing the characters that are in python string.punctuation and replaced with space except for apostrophe which was replaced by an empty string
- Tokenization: tokenize the text using word_tokenize() from nltk
- Removed words with special characters
- Removed stopwords: remove the stopwords in nltk's stopword corpus for English
- Removed purely numeric tokens: remove the tokens that are numeric

Methodology

- Preprocessing as mentioned above
- Jaccard Coefficient
  - Creating document-token index: for each document, storing the set of tokens
  - Creating a set of tokens appearing in the query
  - Calculating the Jaccard coefficient between the query-token set and every document-token set as the size of intersection set by union set
  - Choosing the top 5 documents with the highest Jaccard coefficient as the most relevant
- TF-IDF Matrix
  - Creating document-token index: for each document, storing the set of tokens
  - Creating documentName to documentId index to map each document
  - Creating unigram inverted index for using to find idf values
  - Creating a an idf index that stores idf values for all the words
  - Creating the document-MostFrequentWordCount index for use in the double normalization scheme
  - creating a 5*vocab_size*totalDocs sized index which stores tf-idf (all 5 schemes) value for every word for every document
  - Using tokens in the preprocessed query, finding and retrieving the top 5 documents based on all 5 schemes of tf-idf
  - **Pros and Cons**
    - Binary: In this, the frequency with which a word occurs in a document is not important as long as the word is present in the document. Thus, this can be helpful when we want to receive any possible relevant documents and don't want to lose any potential candidates. Thus, all documents that contain the terms of the query (same terms) will have the same score. However, this is a con as well since it doesn't give importance to the

occurrence. The more frequent existence of query tokens may imply a more relevant document but this can’t be handled by binary count.

- Raw Count: This helps us to get a doc that talks more about the terms present in the document. This means that, since tf value is dependent on the count of the word, docs which mention more about the query terms will have a high score which is required when we want to see which doc has the maximum mention of query terms. However, longer documents may have more occurrence of a token as compared to shorter documents but it may happen that the ratio with which the term occurs in shorter document is high making it more relevant. Eg: 10 occurrences in a doc of length 100 as compared to 15 occurrences in a doc of length 1000. First doc seems more relevant but will be given lesser score.
- Term Frequency: This is a normalized frequency. This means that instead of raw count it presents tf as a ratio of frequency/totalLenOfDoc. Thus, this takes care of the cons of the Raw count scheme and helps us find relevant documents when we want to find documents that talk more about our query terms than they talk about other features. It takes into consideration the total length of the document when evaluating the significance of the doc based on the frequency of the terms. However, the change in the size of the document can change the weighting by a huge factor that in general would not reflect the change in the relevance of the document. For eg: the increase in the size of a document by a factor of thousand would result in the decrease of the weighting by a factor of a thousand.
- Log Normalization: If we want to see which documents have more mention of query terms irrespective of the ratio they are present in the doc, then we can use raw count. However, the raw count can lead to a large scaling of the tf-idf value. Eg: occurrence as 2 instead of 4 makes the tf-idf score 2x larger which might be not true. Thus using log normalization we can bring down this scalar factor and get a better representation of tf-idf values. But again, this doesn’t take into account normalization with respect to the document length and will end up assigning the same score to a word that occurs 4/10000 times and 4/10 times in different docs.
- Double Normalization: The increase in the frequency of the max occurring word by a factor of a thousand should not result in the decrease of the weighting by a factor of a thousand. By including the smoothing factor, double normalization dampens the effect of the change in the size of the document or frequency of the max occurring word. It is possible that in some documents the max frequency for a term is significantly higher but that it is a commonly occurring word and is not a true representative of the feature set of the document. But in this case, it will get a higher tf value with respect to some other word of the document that occurs in

lesser frequency but was a better representative of the document (maybe because its occurrence in other documents was significantly lesser).

Assumptions

- Numbers removed from text and query
- Stemming not done, can uncomment to add stemming
- Escape sequences removed from the text
- Punctuations replaced with space except apostrophe replaced with empty string before tokenization
- Words with special characters removed from the text
- Preprocessed data was stored and loaded to avoid spending time in preprocessing every time

2. Ranked-Information Retrieval and Evaluation

Preprocessing steps

- Selecting only lines containing qid 4
- Creating a dataframe by parsing the data and naming the features

Methodology

- Preprocessing as mentioned above
- To make a file rearranging the query-url pairs in order of max DCG, sorting the data by relevance score and storing data in a csv file
- Calculating the number of possible such files as the product of factorials of the number of files of each relevance score type
- Finding DCG of data sorted by feature 75 and normalising with data sorted by max relevance score upto 50 and for the full dataset
- Calculating precision and recall at each position and plotting the results

Assumptions

- nDCG has been calculated for the sorting of data based on feature 75

3. Naive Bayes Classifier

Preprocessing steps

- Removed escape sequences from the text and replaced them with space
- Converted to lower case: converted the text to lower case using the python lower() function
- Removed punctuations: remove the punctuations by removing the characters that are in the python string.punctuation and replaced with space except apostrophe which was replaced by an empty string
- Tokenization: tokenize the text using whitspacetokenizer from nltk

- Removed words with special characters
- Removed stopwords: remove the stopwords in nltk's stopword corpus for English
- Removed purely numeric tokens: remove the tokens that are numeric
- Lemmatization performed

Methodology

- Extracted documents in either utf-8 or ISO-8859-1 (those that didn't follow utf-8) encoding
- Preprocessing as mentioned above
- Created a mapping of documents to their class and a single data structure to store all documents, their tokens and corresponding class
- Shuffled the dataset
- Created an index to store class level tokens for all the classes
- Found TF-ICF values for the tokens for each of the class
- Sorted the tokens based on decreasing TF-ICF values
- Extracted top k (input) features and take its union to make the feature set for Naive Bayes
- Multinomial Naive Bayes
    - Prepared a dataset matrix to store the count of each feature in the documents. This serves as input data to the model
    - Found Class prior probabilities
    - Found posterior probabilities for every word-class combination.
    - Used Laplace smoothing to avoid zero-frequency problem
    - For test data, calculated probabilities using log addition instead of the product as otherwise, probabilities become small which isn't handled properly by the interpreter
- Reported accuracy for both train and test data
- Reported Confusion Matrix and precision, recall, f1 for every class

Analysis of Results

- As we increase the value of k, we can see that the accuracy also increases. This is because of very less k (like 2-3) will lead to a lot of information loss and might misclassify samples. However, a larger k value like 50 to 75 is better able to capture the essence of the document due to mapping each doc with many features.
- Analysis of performance across different train:test ratios

  Results for 50:50 Split
    - Train Accuracy = 0.94
    - Test Accuracy = 0.9336

  Results for 70:30 Split
    - Train Accuracy = 0.9608571428571429
    - Test Accuracy = 0.9613333333333334

  Results for 80:20 Split
    - Train Accuracy = 0.95075
    - Test Accuracy = 0.941

70:30 split performs the best and also gives the least difference in performance between train and test data which can be explained as it allows for the best split of data - the training data is sufficient to learn the best model without overfitting, and test data is sufficient to evaluate the performance of the model.
- However, we should note that results can vary depending on the shuffling of data and which documents are in the train set. Naive Bayes is a probabilistic model which treats all features as independent so training data can have a decent impact on the models as some posteriors may get different probabilities based on the train data.

Assumptions
- Numbers removed from text and query
- Escape sequences removed from text
- Punctuations replaced with space except apostrophe replaced with empty string before tokenization
- Words with special characters removed from the text
- Preprocessed data was stored and loaded to avoid spending time in preprocessing every time
- Implemented Multinomial Naive Bayes
- Class Name ID Mapping
  - Politics:1
  - Graphics:2
  - Sports:3
  - Space:4
  - Med:5