

Information Retrieval CSE504

Assignment 2

Report

Question 1

TF-IDF Matrix

The inverted index generated in ques 1 has the following structure.

```
{'term': ['frequency', {posting list}]}
```

Using the inverted index (pickle file) generated in Question-1. We first calculated the documented frequency for each term which is further used to calculate the IDF value using the formula $IDF = \log_{10}(DOC_COUNT/(1+doc_freq))$ which is common to all the 5 weighting schemes and stored in a dictionary. So we got a dictionary named **“idf_mapper”** which stores each terms IDF value.

The next we calculated the term frequency of each unique term corresponding to every document. The structure of this **“tf_counter”** is `{'docid': {'term1': 'frequency'}, {'term2': 'frequency'}}`. This will be useful to count to TF values based on different weighting schemes.

1. Binary Scheme

Taking TF as 1 if present in the document and 0 if absent We are making a TF-IDF matrix named **“tf_idf_matrix_binary”** which stores TF-IDF values.

2. Raw Count

Taking TF as frequency of word in that document which can be used from tf_counter.

3. Term Frequency

Taking TF as term frequency of word in that document/ total number of words in the document. We have calculated this using $TF = freq / \text{sum}(\text{terms.values}())$ which is later used to calculate the TF-IDF Matrix.

4. Log Normalization

We will calculate TF using the formula $tf = \text{math.log10}(1 + freq)$ and make TF-IDF matrix named **“tf_idf_matrix_log_norm”**.

5. Double Log Normalization

We will calculate tf using the formula $TF = 0.5 + ((0.5 * freq)/MAX_FREQ)$ and make the TF-IDF Matrix named **“tf_idf_matrix_double_norm”**. The minimum value in this weighting scheme matrix should be 0.5.

We have retrieved top 5 documents by 2 ways:

1. By calculating the **score** with the formula:

Score for a document given a query

$$\text{Score}(q, d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

We first make query vector of the query w.r.t to each of the 5 weighting scheme by finding tf of each word in the query (how many times a word occurs in the query) and multiplying it with the idf of that word and then sum that query vector with the specific document vector and later sum all the values to find a score for a document. We have sorted the scores to find 5 relevant documents for each scheme.

2. By calculating **cosine similarity**

We first make query vector of for the query wrt to each of the 5 weighting scheme by finding tf of each word in the query (how many times a word occurs in the query) and multiplying it with the idf of that word. Final score of relevance of document to a query is cosine similarity between query vector and document vector. Then we have sorted the scores to find the top 5 relevant documents.

Binary Weighting Scheme

Pros: It is simple to implement and it is computationally efficient because it only requires counting the presence or absence of each term in the document, rather than calculating a numerical value for each term.

Cons: It ignores term frequency, which can be a useful indicator of its importance or relevance. By assigning binary values to each term, binary weighting loses information about the relative frequency of different terms in the document.

Raw Count Weighting Scheme

Pros: It is simple to implement and it preserves information about the frequency of each term in the document, which can be a useful indicator of its importance or relevance. It is effective for tasks such as keyword extraction or topic modeling, where the goal is to identify the most frequent or important terms in a document.

Cons: It is sensitive to document length, means longer documents will generally have higher term frequencies and therefore higher weights. When all the documents are of similar size then this scheme will be helpful as we are not considering the normalization and works bad when there are documents of different size

Term Frequency Weighting Scheme

Pros: It captures term frequency, which can be a useful indicator of its importance or relevance. The TF weighting scheme can be normalized by dividing the raw term frequency by the total number of terms in the document, which helps to mitigate the problem of longer documents having higher term frequencies.

Cons: It ignores global term frequency. It treats all terms equally, regardless of how specific or general they are. This can lead to more common terms being overweighted and more specific terms being underweighted.

Log Normalisation Weighting Scheme:

Pros: It reduces the impact of very frequent terms, which can help to mitigate the problem of common words such as "the" or "and" dominating the document representation. It preserves information about the frequency of each term in the document.

Cons: The log normalization weighting scheme treats all terms equally, regardless of how specific or general they are. This can lead to more common terms being overweighted and more specific terms being underweighted.

Double Log Normalisation Weighting Scheme:

Pros: It takes into account global term frequency. The double log normalization weighting scheme takes into account the frequency of each term across all documents in the corpus, which can help to mitigate the problem of terms that are very common or very rare across the corpus.

Cons: It is more complex and it is more difficult to implement or understand. It treats all terms equally, regardless of how specific or general they are. This can lead to more common terms being overweighted and more specific terms being underweighted.

Jaccard Coefficient

JC is a similarity measure which is used to compare the similarity and diversity of two sets of data. We calculate jaccard coefficient for every document wrt to the query using the formula given below:

Formula



$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

J = Jaccard distance

A = set 1

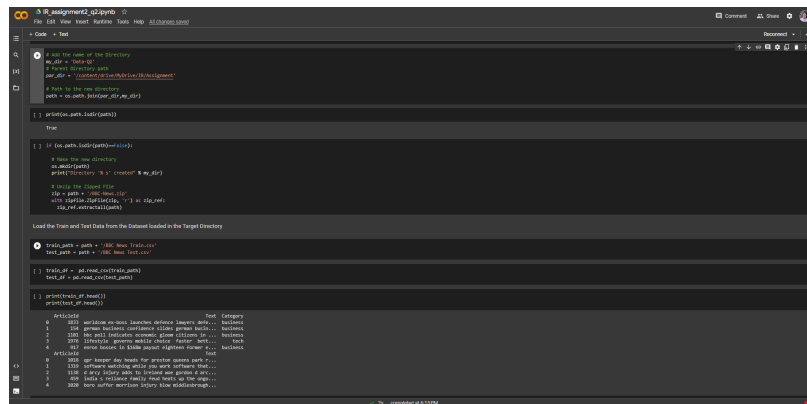
B = set 2

Higher the JC value, Higher the relevance of documents.

Question - 2

The procedure followed for Question-2 is as follows: -

1. Dataset Description - The dataset is downloaded from Kaggle and it consists of two distinct files as the Train.csv and Test.csv. The Text data present in Train.csv consists of three columns as the Article-ID, the Text - which consists of the text content of the article and the Category where each of the article is classified or labeled with one of the five distinct categories as either ;- **Business, Sport, Tech, Entertainment or Politics.** The Test.csv file does not consist of the labels and hence is not used. **The Train-Test splits are executed on the Train.csv file downloaded from Kaggle** only so as to ensure that the test set also has the predicted labels and can be later evaluated for parameters as accuracy, precision, recall and F-1 Score.



```
import os
import pandas as pd

# Path to the directory
root_dir = 'data'

# Path to the train directory
train_dir = os.path.join(root_dir, 'train')

# Path to the test directory
test_dir = os.path.join(root_dir, 'test')

# List the files in the directory
files = os.listdir(train_dir)

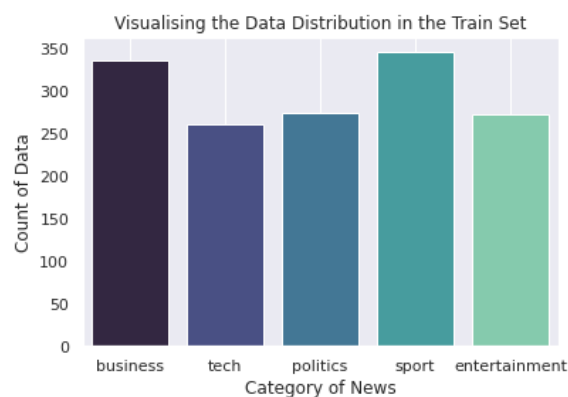
# Read the train data
train_data = pd.read_csv(os.path.join(train_dir, 'train.csv'))

# Read the test data
test_data = pd.read_csv(os.path.join(test_dir, 'test.csv'))

# Preview the data
train_data.head()
```

Article-ID	Text	Category
101	business
102	business
103	business
104	business
105	business

2. Dataset Preprocessing - The data is evaluated to check for any NULL value in any of the columns or any redundant data or duplicated rows are eliminated from the entire dataset.
3. Dataset Visualization - The count of no. of articles across all the categories is checked and the count plot is visualized as below.



4. Text Preprocessing - The function defined as preprocess_text, takes in input as an input text and applies the following techniques for text pre-processing.

- a. The first step is to remove the punctuations and using the set of all available punctuations, it is possible to identify the punctuations and eliminate them by using **string.punctuations**.
 - b. To convert the text to lowercase, use **str.lower()**
 - c. Use NLTK to eliminate the **stopwords** and **tokenize** the given input sentence.
 - d. The next step is to execute **Lemmatization** and we use **WordNetLemmatizer** from NLTK library for the same.
5. The functions for evaluating the TF-ICF are as follows -
- a. **get_tf(X,y)** : Get the Training Data i.e. the text samples in X_train and corresponding labels or category for each row of text in y_train and evaluates the count of each term i.e. the term frequency for each of the term, corresponding to a given category and hence the function returns a dictionary of dictionaries where the outer key consists of the labels or the categories and the inner values correspond to another nested dictionary consisting of the term as the key and the corresponding frequency as the value and hence form a key-value pair for the nested dictionary. NOTE - If a word exists in a document 'n' no of times, then all of its occurrences are evaluated as the term frequency of the term for the label in a specific document. Next if the term appears in another document corresponding to the same label, the frequency is incremented accordingly. The function also evaluates the count of documents in which a term occurs and hence returns the second structure as a dictionary which consists of a unique term as the key and the no. of documents in which the term appears at least once.
 - b. **get_icf (my_tf_dict, all_class)**: The function to evaluate the Inverse Category Frequency. The input to the function is the dictionary of dictionaries for the term frequency and the count for the total number of classes. We first iterate over the keys in the nested dictionary corresponding to the label/category values in the outer dictionary and next, if the given term exists as a key corresponding to one or more labels we increment the category count and evaluate the ICF for the given word using the log of ratio of all categories over the count of categories in which the term exists. The function returns a dictionary with the key as all the unique words in the corpus and the corresponding ICF Value evaluated using the formula as - **$\log(\text{all_class}/\text{cat_count})$** , with base 10, where cat_count for each word is the total number of categories in which the word exists.
 - c. **get_idf(my_vocab,all_doc_train)** - The function is to return the dictionary which contains the Inverse Document Frequency corresponding

to each term and the IDF is evaluated over the entire training corpus where the formula used is **$\log(\text{all_doc_train}/\text{my_vocab}[\text{word}])$** , with the base 10, where my_vocab has a value corresponding to the dictionary as word, as the number of documents in which the word occurs.

- d. **get_tf_icf(my_tf_dict,icf_dict)** - Input to the function is the dictionaries which consists of the term frequency and the inverse category frequencies individually. Used to find the product of the term frequency and the inverse category frequency. Returns a dictionary which consists of the TF-ICF values for the entire training words for a given category.
 - e. **get_tf_idf(my_tf_dict,icf_dict)** - Used to find out the product of values of Term Frequency and Inverse Document Frequency and returns a dictionary for TF-ICF values.
6. Training the Naive Bayes Classifier - The class NaiveBayesClassifier creates the classifier object of Naive Bayes and classifies the test text on the basis of learnt feature and category weights. The function for training in the class initializes the values of dictionaries for the created object from the parameters passed in the function.
- a. The function also evaluated the category count for the total no. of documents i.e. we have to count the number of documents in each category to calculate the category priors.
7. Prediction using the Naive Bayes Algorithm - It has the following steps -
- Calculate the probability of each category based on the frequency of documents in the training set that belong to that category. The prior probability of each category is evaluated in **cat_prob_dict**.
 - Calculate the probability of each feature given each category based on the TF-ICF values of that feature in documents belonging to that category. The **log_prob_dict** is the dictionary which consists of the final evaluated predicted probabilities for each category. The process to evaluate the probability of each feature over each category is summed over all the text or the features in the text currently. All the words are iterated and the feature probabilities are evaluated, conditioned on each label.
 - The label, for which the dictionary has the maximum value is returned as the predicted label for the given text.
8. Evaluation : - The metrics on the basis of which the performance is evaluated are as **Accuracy, Precision, Recall and F-1 Score**. The function defined for the same is as **get_metric_eval(res_category, y_test)** where the res_category are the predicted categories for the input text and y_test are the actual labels for the same category and the original and predicted labels are used to get the final results.

9. The run_main function takes in the input as the mode and split where the mode is either TF-ICF or TF-IDF and the split is defined as the ratio of split for varying train-test split.

10. Improving the Classifier - The model is also evaluated on different splits for both, TF-ICF and TF-IDF weighting schemes and the following train-test split ratios are explored for both as: -

a. 80:20

i. TF-ICF - **'Accuracy': 96.30872483221476, 'Precision': 0.9646332814825966, 'Recall': 0.9623953658282016, 'F1-Score': 0.9627989033516539**

ii. TF-IDF - 'Accuracy': 89.26174496644296, 'Precision': 0.9129645979543755, 'Recall': 0.8835350157089288, 'F1-Score': 0.8890271804644924

b. 70:30

i. TF-ICF - 'Accuracy': 95.74944071588367, 'Precision': 0.9562714617587795, 'Recall': 0.9559947739614486, 'F1-Score': 0.9556439435912824

ii. **TF-IDF - 'Accuracy': 91.2751677852349, 'Precision': 0.9276819293291088, 'Recall': 0.9039195289134916, 'F1-Score': 0.9096198912091318**

c. 60:40

i. TF-ICF - 'Accuracy': 95.46979865771812, 'Precision': 0.9550386275434983, 'Recall': 0.9546563019040082, 'F1-Score': 0.954008947555949

ii. TF-IDF - 'Accuracy': 89.09395973154362, 'Precision': 0.9089094540119549, 'Recall': 0.8806037913836079, 'F1-Score': 0.8820994419277763

d. 50:50

i. TF-ICF - 'Accuracy': 94.8993288590604, 'Precision': 0.9480129743102547, 'Recall': 0.947162776396353, 'F1-Score': 0.9472899291692112

ii. TF-IDF - 'Accuracy': 90.20134228187919, 'Precision': 0.9213582925401134, 'Recall': 0.8922629568650834, 'F1-Score': 0.8994375419513394

e. The model is further evaluated by modifying the combination of Preprocessing techniques as Stemming and Lemmatization. The performance of the classifier is explored using both stemming and lemmatization.

Following are the results using only Lemmatization.

The value of evaluation metric for TF_ICF with split ratios and using both the pre-processing techniques as stemming and Lemmatization. The metrics do not improve significantly. As{'Accuracy': 94.96644295302013, 'Precision': 0.9519709272530781, 'Recall': 0.9478499112827471, 'F1-Score': 0.9483557857819935} Accuracy': 94.96644295302013, 'Precision': 0.9519709272530781, 'Recall': 0.9478499112827471, 'F1-Score': 0.9483557857819935.

11. Conclusion - TF-ICF weighting scheme with the train-Test split as 80:20 and using only Lemmatization as the pre-processing techniques outputs the maximum accuracy for the given test set of text.

Question 3:

In this question, we have given a Microsoft Learning Query-Url pairs ranking dataset, on which we have to perform three tasks.

Dataset Description:

1. **Size:** 264 MB
2. **Format:** .txt
3. **Shape:** 239093 rows and 138 columns

Preprocessing: First, we have loaded the text file into the pandas data frame just to make it easier to perform operations on the dataset. Then we assign column names to each of the columns.

For the first task in which, we have to filter out the dataset for '**qid:4**' only. After filtering the dataset, we get (103, 138) size dataset. Then we sort the dataset based on the relevance score to get the maximum DCG. We then store this data frame in a **filtered_max_dcg.txt** file. Now to find the number of arrangements, we have observed that, to get the maximum DCG, the relevance must be in sorted order.

Taking this observation, we have already sorted the dataset on the basis of relevance score. Now the total number of arrangements will be.

$$\text{factorial}(\text{count}_0) * \text{factorial}(\text{count}_1) * \text{factorial}(\text{count}_2) * \text{factorial}(\text{count}_3)$$

Which results in a very big number: 138683118545689835737939019720389406345902.....

The second task is to find the N-DCG of:

1. **First 50 documents**
2. **Whole dataset**

For calculating the DCG, we use the formula:

$$DCG_p = \sum_{i=1}^p \frac{rel_i}{\log_2(i+1)} = rel_1 + \sum_{i=2}^p \frac{rel_i}{\log_2(i+1)}$$

$$nDCG_p = \frac{DCG_p}{IDCG_p},$$

[1]

N-DCG for first 50 documents: **0.5253808413557646**

N-DCG for the whole dataset: **0.5979226516897831**

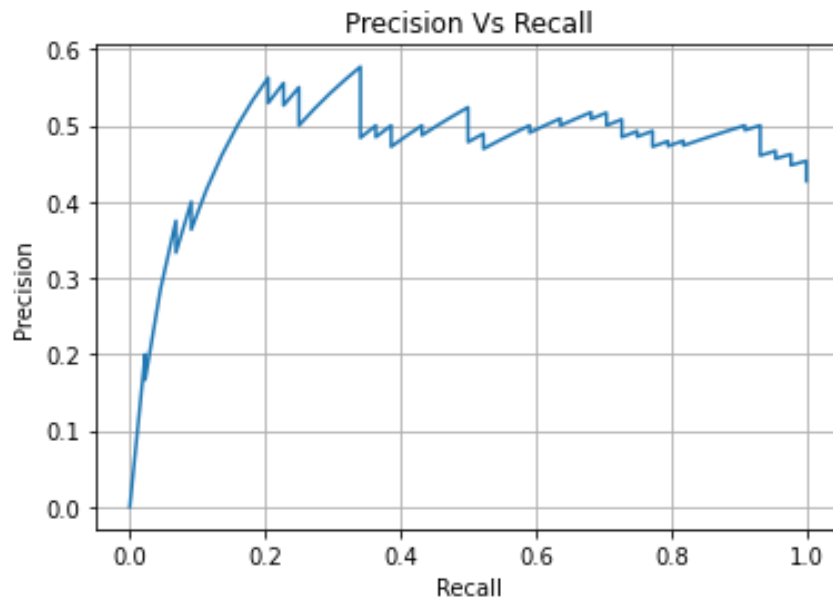
The third task is first to rank the dataset using feature_75, finding precision and recall, then plotting the **precision vs recall curve**.

For that, we first extracted the numerical values from the feature_75, then sort the dataset, then iterating over the whole dataset, and based on the relevance_score, we calculated precision and recall using the standard formula provided.

Precision = relevant_docs / number_of_retreived_docs_found_so_far

Recall = number_of_relevant_documents_retrieved_so_far / total_number_of_relevant_documents

Then after storing the precision and recall of the dataset into a python list and using matplotlib, we plotted the precision/recall curve.



[1]: https://en.wikipedia.org/wiki/Discounted_cumulative_gain