Assignment 2 Report

Mohammad Aflah Khan, 2020082
Neemesh Yadav, 2020529

Q1. All the preprocessing results have been shown in the notebook. The following sections show the results and our methodology for the Jaccard Scoring and TFIDF parts.

Queries Used for our experiments:-
Query 1: `"turbulent incompressible laminar peripheral jets proximity"`
Query 2: `"reynolds number and potential shear"`

Methodology for TF-IDF:

The first section of the code defines a class named `TFIDF` that calculates the Term Frequency - Inverse Document Frequency (TF-IDF) score for a list of documents. The constructor takes in a list of documents and a weighting scheme for the term frequency, and initializes the object with the following properties:

- `documents`: the list of documents passed to the constructor.
- `weighting`: the weighting scheme for the term frequency.
- `vocab`: a dictionary that maps terms to their indices in the TF-IDF matrix.
- `N`: the total number of documents in the corpus.
- `tf`: a dictionary that stores the term frequency for each document and term, where the keys are the document indices and the values are dictionaries that map terms to their frequency.
- `idf`: a dictionary that stores the inverse document frequency for each term, where the keys are the terms and the values are their IDF score.
- `tf_idf`: a 2D numpy array that stores the TF-IDF score for each document and term.

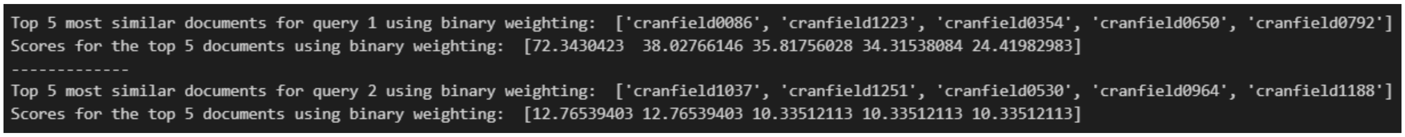
The class has several methods that can be used to query the TF-IDF matrix:

- `compute_tf()`: a method that computes the term frequency for each document using the specified weighting scheme.
- `compute_idf()`: a method that computes the inverse document frequency for each term.
- `compute_tf_idf()`: a method that computes the TF-IDF score for each document and term.
- `query_processing(query)`: a method that processes a query by applying the same weighting scheme used for the documents.
- `get_score(query)`: a method that returns the TF-IDF score for a given query.
- `get_top_k(query, k)`: a method that returns the indices of the top k documents that match the query.

- `get_tf()`: a method that returns the term frequency dictionary.
- `get_idf()`: a method that returns the inverse document frequency dictionary.
- `get_tf_idf()`: a method that returns the TF-IDF matrix.
- `get_vocab()`: a method that returns the vocabulary dictionary that maps terms to their indices in the TF-IDF matrix.

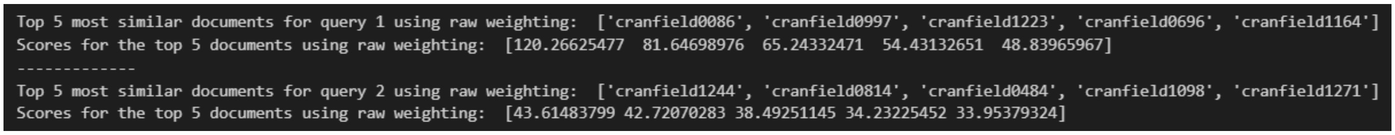
Results for each query:-

Binary Weighting:



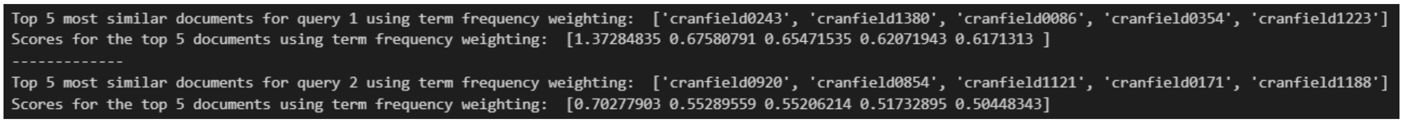
```
Top 5 most similar documents for query 1 using binary weighting:  ['cranfield0086', 'cranfield1223', 'cranfield0354', 'cranfield0650', 'cranfield0792']
Scores for the top 5 documents using binary weighting:  [72.3430423  38.02766146 35.81756028 34.31538084 24.41982983]
-------------
Top 5 most similar documents for query 2 using binary weighting:  ['cranfield1037', 'cranfield1251', 'cranfield0530', 'cranfield0964', 'cranfield1188']
Scores for the top 5 documents using binary weighting:  [12.76539403 12.76539403 10.33512113 10.33512113 10.33512113]
```

Raw Weighting:



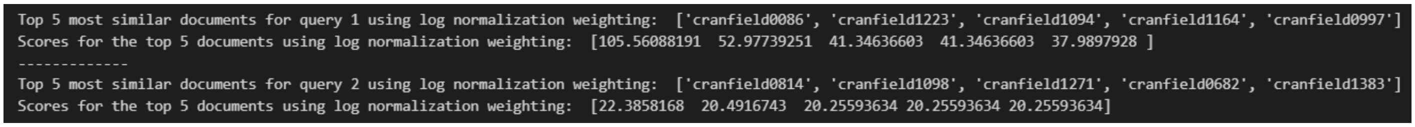
```
Top 5 most similar documents for query 1 using raw weighting:  ['cranfield0086', 'cranfield0997', 'cranfield1223', 'cranfield0696', 'cranfield1164']
Scores for the top 5 documents using raw weighting:  [120.26625477  81.64698976  65.24332471  54.43132651  48.83965967]
-------------
Top 5 most similar documents for query 2 using raw weighting:  ['cranfield1244', 'cranfield0814', 'cranfield0484', 'cranfield1098', 'cranfield1271']
Scores for the top 5 documents using raw weighting:  [43.61483799 42.72070283 38.49251145 34.23225452 33.95379324]
```

Term Frequency Weighting:



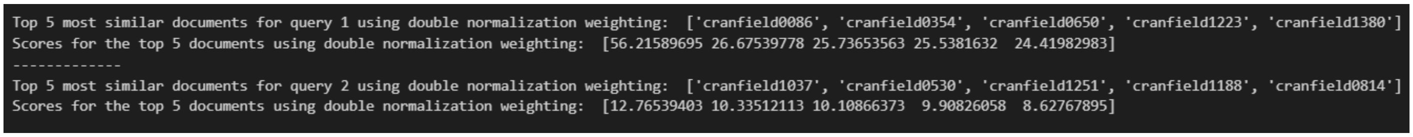
```
Top 5 most similar documents for query 1 using term frequency weighting:  ['cranfield0243', 'cranfield1380', 'cranfield0086', 'cranfield0354', 'cranfield1223']
Scores for the top 5 documents using term frequency weighting:  [1.37284835 0.67580791 0.65471535 0.62071943 0.6171313 ]
-------------
Top 5 most similar documents for query 2 using term frequency weighting:  ['cranfield0920', 'cranfield0854', 'cranfield1121', 'cranfield0171', 'cranfield1188']
Scores for the top 5 documents using term frequency weighting:  [0.70277903 0.55289559 0.55206214 0.51732895 0.50448343]
```

Log Normalization Weighting:



```
Top 5 most similar documents for query 1 using log normalization weighting:  ['cranfield0086', 'cranfield1223', 'cranfield1094', 'cranfield1164', 'cranfield0997']
Scores for the top 5 documents using log normalization weighting:  [105.56088191  52.97739251  41.34636603  41.34636603  37.9897928 ]
-------------
Top 5 most similar documents for query 2 using log normalization weighting:  ['cranfield0814', 'cranfield1098', 'cranfield1271', 'cranfield0682', 'cranfield1383']
Scores for the top 5 documents using log normalization weighting:  [22.3858168  20.4916743  20.25593634 20.25593634 20.25593634]
```

Double Normalization Weighting:



```
Top 5 most similar documents for query 1 using double normalization weighting:  ['cranfield0086', 'cranfield0354', 'cranfield0650', 'cranfield1223', 'cranfield1380']
Scores for the top 5 documents using double normalization weighting:  [56.21589695 26.67539778 25.73653563 25.5381632  24.41982983]
-------------
Top 5 most similar documents for query 2 using double normalization weighting:  ['cranfield1037', 'cranfield0530', 'cranfield1251', 'cranfield1188', 'cranfield0814']
Scores for the top 5 documents using double normalization weighting:  [12.76539403 10.33512113 10.10866373  9.90826058  8.62767895]
```

TF-IDF (Term Frequency-Inverse Document Frequency) weighting schemes are used to assign weights to the terms in a document based on their importance in the corpus. Each weighting scheme has its own pros and cons, which are outlined below:

1. Binary:
    a. Pros: This is the simplest scheme, where the weight of a term is either 1 or 0 depending on whether it is present or absent in the document. It is useful in text

classification problems where presence/absence of a term is the only thing that matters.

   b. Cons: This scheme does not take into account the frequency of the term in the document or the corpus.

2. Raw:
   a. Pros: This scheme assigns the raw frequency of a term in the document as its weight, which is useful when frequency of a term in a document is indicative of its importance.
   b. Cons: This scheme does not account for the frequency of the term in the corpus, which can lead to the over-representation of common words.
3. Term Frequency (TF):
   a. Pros: This scheme normalizes the raw frequency of a term by the total number of terms in the document, which makes it less sensitive to document length. It is widely used in text retrieval applications.
   b. Cons: This scheme does not account for the frequency of the term in the corpus, which can lead to the over-representation of common words.
4. Log normalization:
   a. Pros: This scheme takes the log of the term frequency, which reduces the impact of high frequency terms and smooths out the distribution. It is particularly useful for long documents where some terms may occur many times.
   b. Cons: This scheme can underestimate the importance of low frequency terms.
5. Double normalization:
   a. Pros: This scheme normalizes the term frequency using the maximum term frequency in the document, which makes it less sensitive to document length and frequency of the term in the corpus. It can improve the accuracy of the retrieval results.
   b. Cons: This scheme can overestimate the importance of terms that occur frequently in the document.

Methodology for Jaccard Similarity:

The second section of the code defines a class named `Jaccard`, which has a constructor method `__init__` that takes in two arguments, `documents` and `vocab`.

`documents` is a list of documents, where each document is represented as a string of words. `vocab` is a dictionary that contains the vocabulary of the documents, i.e., the set of unique words that occur in the documents.

The class has a method named `compute_jaccard`, which takes in a `query` argument, which is also a string of words. The method computes the Jaccard similarity coefficient between the query and each document in the `documents` list. The Jaccard coefficient measures the similarity between two sets, which in this case are the set of words in a document and the set of words in the query.

The method computes the Jaccard coefficient for each document by first splitting the document and the query into sets of words using Python's `set` data type. It then computes the intersection and union of the sets using the `&` and `|` operators, respectively. Finally, it divides the size of the intersection by the size of the union to obtain the Jaccard coefficient.

The method returns an array `jaccard_coeff`, which contains the Jaccard coefficient for each document in the `documents` list.

Results for each query:-

Query 1 —>

```
Top 10 most similar documents for query 1 according to jaccard
similarity:  ['cranfield0382', 'cranfield0376', 'cranfield0243',
'cranfield0254', 'cranfield1141', 'cranfield0387', 'cranfield0242',
'cranfield0258', 'cranfield0418', 'cranfield0664']

Scores of top 10 most similar documents for query 1 according to
jaccard similarity:  [0.08695652 0.08108108 0.07894737 0.06666667
0.05882353 0.05882353

 0.05714286 0.05714286 0.05714286 0.05405405]
```

Query 2 –>

```
Top 10 most similar documents for query 2 according to jaccard
similarity:  ['cranfield0389', 'cranfield0670', 'cranfield0254',
'cranfield1085', 'cranfield0491', 'cranfield0530', 'cranfield0669',
'cranfield0003', 'cranfield0361', 'cranfield0965']

Scores of top 10 most similar documents for query 2 according to
jaccard similarity:  [0.07692308 0.07692308 0.06896552 0.06060606
0.06       0.05555556

 0.05405405 0.05263158 0.05263158 0.05263158]
```

Q2.

Methodology:

The code provided defines a Python class named TF_ICF which is used to calculate Term Frequency-Inverse Class Frequency (TF-ICF) for a given dataset.

TF-ICF is a statistical measure used in information retrieval and text mining. It is used to evaluate the importance of a term in a document. The measure combines two factors: term frequency (TF) and inverse class frequency (ICF).

The class takes in four parameters: classes, ls_text, ls_classes and initializes tf, cf, icf, and tf_icf.

classes is a list of all the classes in the dataset, ls_text is a list of all the documents, and ls_classes is a list of class labels for each document.

The build method is called upon initialization of the class which then calls four other methods: build_tf, build_cf, build_icf, and build_tf_icf.

The build_tf method takes a class as input and builds the Term Frequency (TF) for each word in that class. It does so by filtering the text for a given class, calculating the frequency of each word, and storing the frequency in a dictionary named self.tf.

The build_cf method calculates the Class Frequency (CF) for each word in the entire dataset. It does so by calculating the number of classes in which each word occurs and stores the frequency in a dictionary named self.cf.

The build_icf method calculates the Inverse Class Frequency (ICF) for each word in the entire dataset. It does so by taking the log10 of the ratio of the total number of classes to the CF of each word, and storing the frequency in a dictionary named self.icf.

Finally, the build_tf_icf method calculates the TF-ICF score for each word in each class. It does so by multiplying the TF score from self.tf and the ICF score from self.icf and storing the result in a dictionary named self.tf_icf.

This code defines a function called featurize that takes in several parameters:

- ls_text_train: a list of training text
- ls_classes_train: a list of training classes
- ls_text_test: a list of test text

- ls_classes_test: a list of test classes
- tf_icf: a TF_ICF object
- unique_classes: a list of unique classes
- feature_set: a list of features to be used

The purpose of the function is to featurize the dataset using TF_ICF. It returns four lists:

- train_X: a list of training features for each text
- train_Y: a list of training classes for each text
- test_X: a list of test features for each text
- test_Y: a list of test classes for each text

The function first loops through the unique_classes list to filter the dataset for each unique class. It then featurizes the dataset for each class using the feature_set list. For each text in the training and test sets, it creates a list of features based on the frequency of the words in the feature_set list in that text. It appends these feature lists to the appropriate train_X, train_Y, test_X, and test_Y lists. The function then returns these four lists.

The dimensions of the returned lists will depend on the number of unique classes and the number of features in the feature set.

train_X and test_X will have dimensions of (number of training or test texts, number of features), where the number of training or test texts is the sum of the number of texts in each unique class.

train_Y and test_Y will have dimensions of (number of training or test texts,), where the number of training or test texts is the sum of the number of texts in each unique class.

For example, if there are 5 unique classes and each class has 100 training texts and 50 test texts, and the feature set has 5000 features, then train_X and test_X will have dimensions of (500, 5000) and train_Y and test_Y will have dimensions of (500,).

This code also has a class called NaiveBayes which implements a Naive Bayes classifier with Laplace smoothing. The class has two main methods: train() and predict().

The train method in the NaiveBayes class is used to train a Naive Bayes classifier with Laplace smoothing. The method takes two parameters: train_x and train_y, which are lists of training features and training classes, respectively. The train_x parameter is a list of lists, where each inner list represents the feature vector for a single training example. The train_y parameter is a list of corresponding class labels for each training example.

The method starts by initializing some variables, such as total_class_samples, which represents the total number of training examples, self.num_features, which represents the number of features in each training example, and self.class_labels, which is a list of unique class labels present in the training data. It then initializes some dictionaries to store the prior and conditional probabilities, as well as some statistics related to the training data.

Next, it calculates the prior probability of each class by counting the number of training examples in each class and dividing by the total number of training examples. It also calculates the cumulative frequency statistics for each feature in each class, which is the sum of the frequencies of that feature across all training examples in that class. This is used to calculate the conditional probabilities later.

The method then calculates the conditional probability of each feature given each class. This is done by iterating over each class and feature and calculating the ratio of the number of occurrences of that feature in that particular class, plus a smoothing factor self.alpha, to the sum of the frequencies of all features in that class plus self.alpha times the number of features. This is the Laplace smoothing technique, which is used to avoid zero probabilities when a feature is not present in a particular class.

Finally, the method returns the prior and conditional probabilities, which are stored in the self.prior_prob and self.conditional_prob dictionaries, respectively. These probabilities are used to make predictions on new data using the predict method.

The predict() method takes a single argument test_x, which is a list of features for the test data. The method uses the prior and conditional probabilities calculated in the train() method to calculate the posterior probabilities for each class, and then returns the label with the highest probability as the predicted label for each sample in test_x.

Additionally, the code also defines a helper function called calculate_metrics() which takes two arguments: true_y and pred_y, which are lists of true and predicted labels respectively. The function then calculates the confusion matrix, accuracy, precision, recall and f1-score for each class, and returns them as a dictionary.

Findings:
We tried 2 different values of alpha for smoothing and found them to give the same results

Accuracy: 0.9753914988814317
Precision: 0.9739492301290055
Recall: 0.974875845074872
F1: 0.9738436157123648

We then tried not removing stopwords and our accuracy went down to 0.9642058165548099. We also swapped Lemmatization for Stemming and our accuracy went up to 0.9798657718120806. These observations show that stopwords bring more noise and hence our IR system does not benefit from their presence. Also Stemming seems to be a better choice when compared to Lemmatization which maybe because it abruptly cuts tokens reducing vocab size which may act as some sort of an implicit regularizer to curb curse of dimensionality. Complete results for the 2 experiments are -

Stop Words Not Removed:

Accuracy: 0.9642058165548099
Precision: 0.9635892377885131
Recall: 0.9630680380868373
F1: 0.9627551179176397

Stemming over Lemmatization:

Accuracy: 0.9798657718120806
Precision: 0.9794836364685349
Recall: 0.9801349260786811
F1: 0.9796371215328188

For different splits the accuracy results are:

Split	Accuracy	Precision	Recall	F1
50-50	0.9624161073825503	0.9610456250200092	0.9617318570853218	0.960955571549917
60-40	0.947986577181208	0.9456123815822272	0.9482966126984025	0.9456552331512895
80-20	0.9630872483221476	0.962337363564326	0.9635230643023917	0.9618617454450613
90-10	0.959731543624161	0.953781512605042	0.9608067226890757	0.9558216932257853

Hence the 80-20 Split seems to be the best among all splits. The reason for this might be that 80-20 split allocates more training data and also has more representative testing data as compared to 90-10 split where the testing split might be skewed to some vector subspace which is sort of an outlier for the training data. The difference between the two is still very negligible.

When we tried countvectorizer to get ngram vectors and tfidfvectorizer to get tfidf vectors we get the following accuracy results -

Vectorization Method	Accuracy	Precision	Recall	F1
Unigram Vectors	0.9753914988814317	0.9739512472638593	0.9744118200639941	0.9738093374713547
Unigram and Bigram Vectors	0.9753914988814317	0.9739512472638593	0.9744118200639941	0.9738093374713547
Bigram Vectors	0.9753914988814317	0.9739512472638593	0.9744118200639941	0.9738093374713547
TFIDF Vectors	0.9642058165548099	0.9664366083986338	0.9625157041769073	0.963786951933636

Surprisingly all 3 ngram vector configurations give the same result, hence we can simply use the Unigram version due to its smaller compute requirements. The NGram vectorizer performs onpar with the TFICF method while TFIDF performs below par.

How to Run Code:

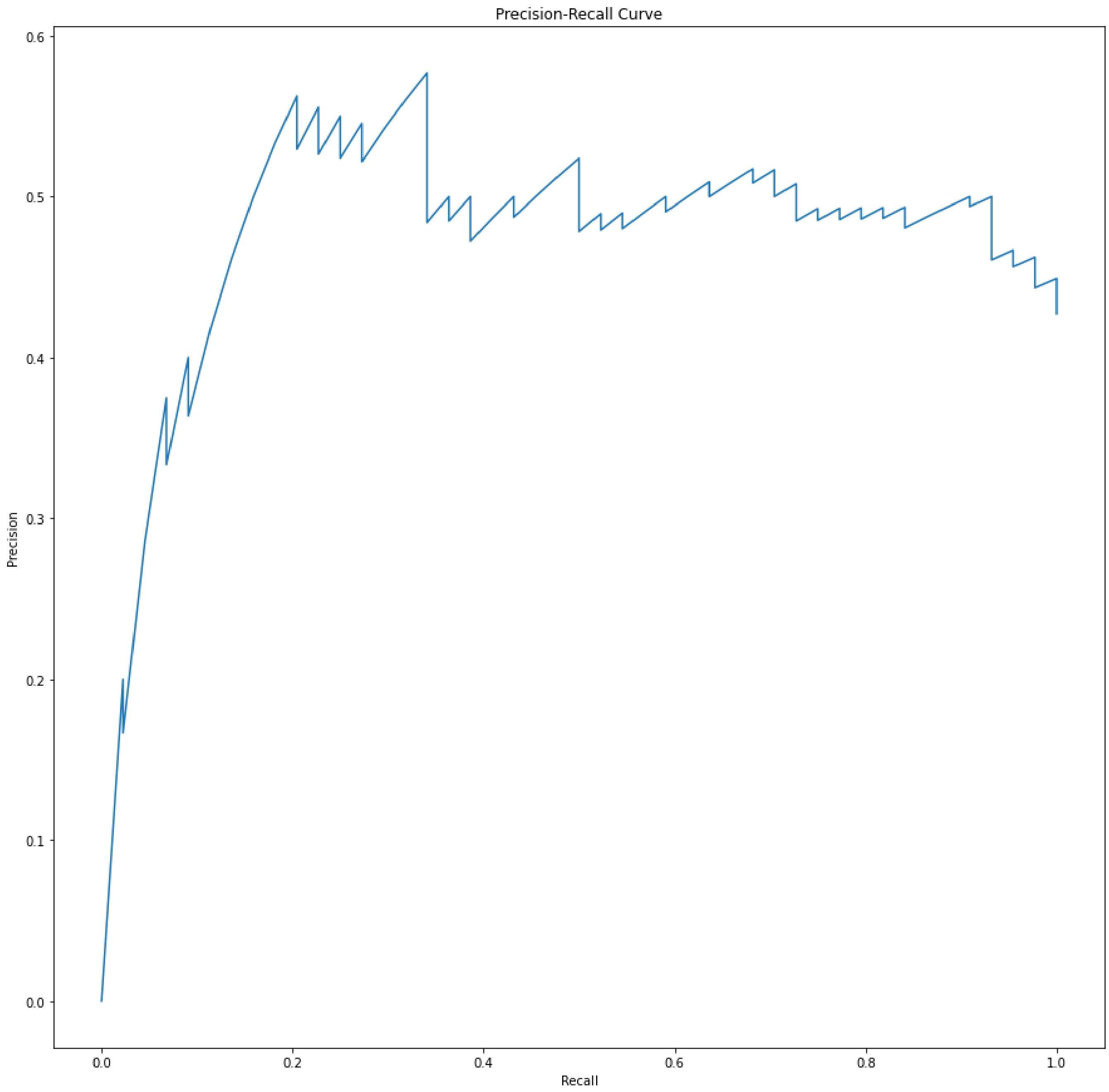
Simply open main.ipynb and run all cells!

Q3.

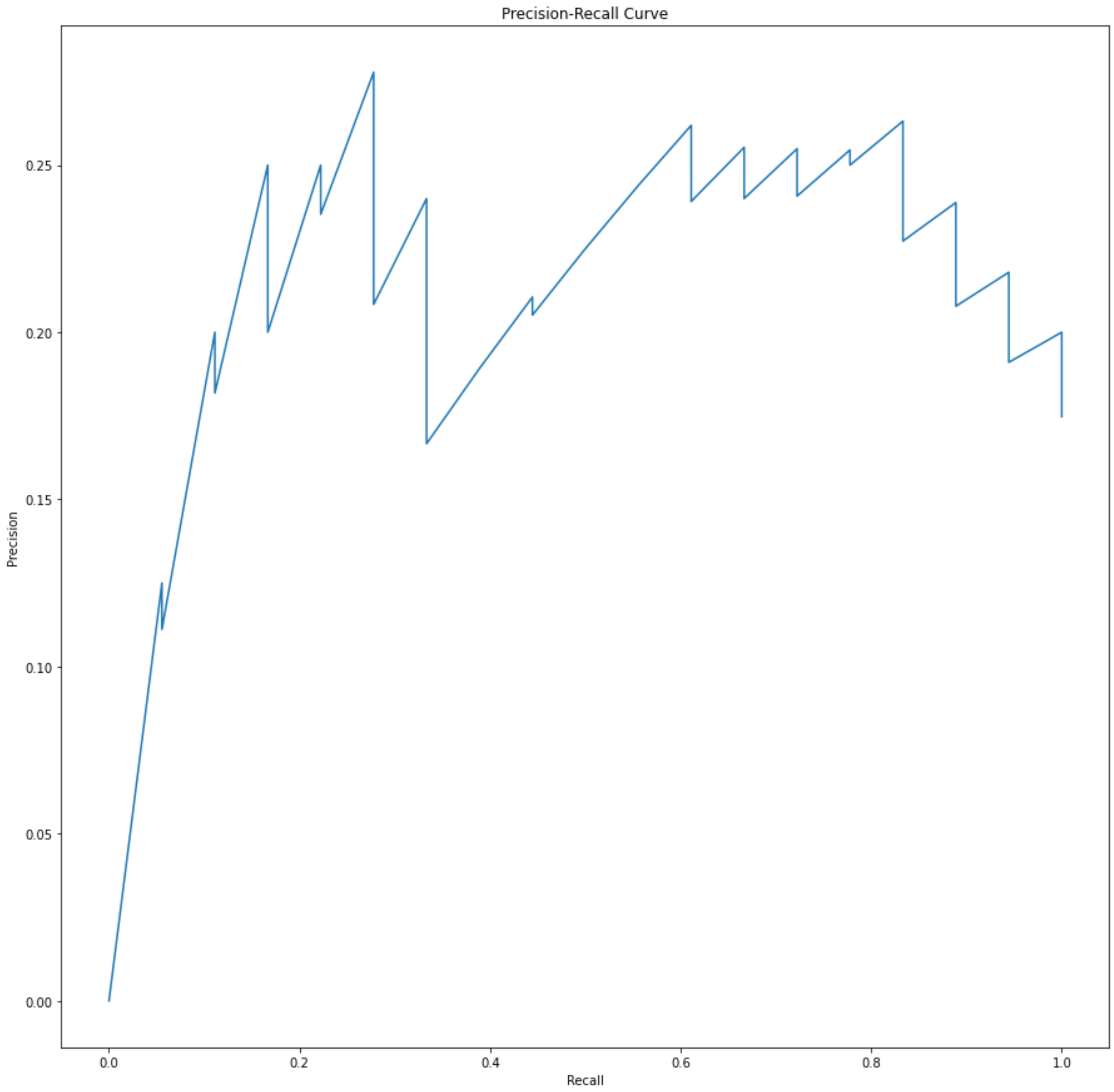
Number of different possible files:
198934973759383705998260476149053298969368401705665705882051803127048579926951934824126865654310502400000000000000000000000

nDCG@50: 0.3521042740324887
nDCG@entire: 0.5979226516897831

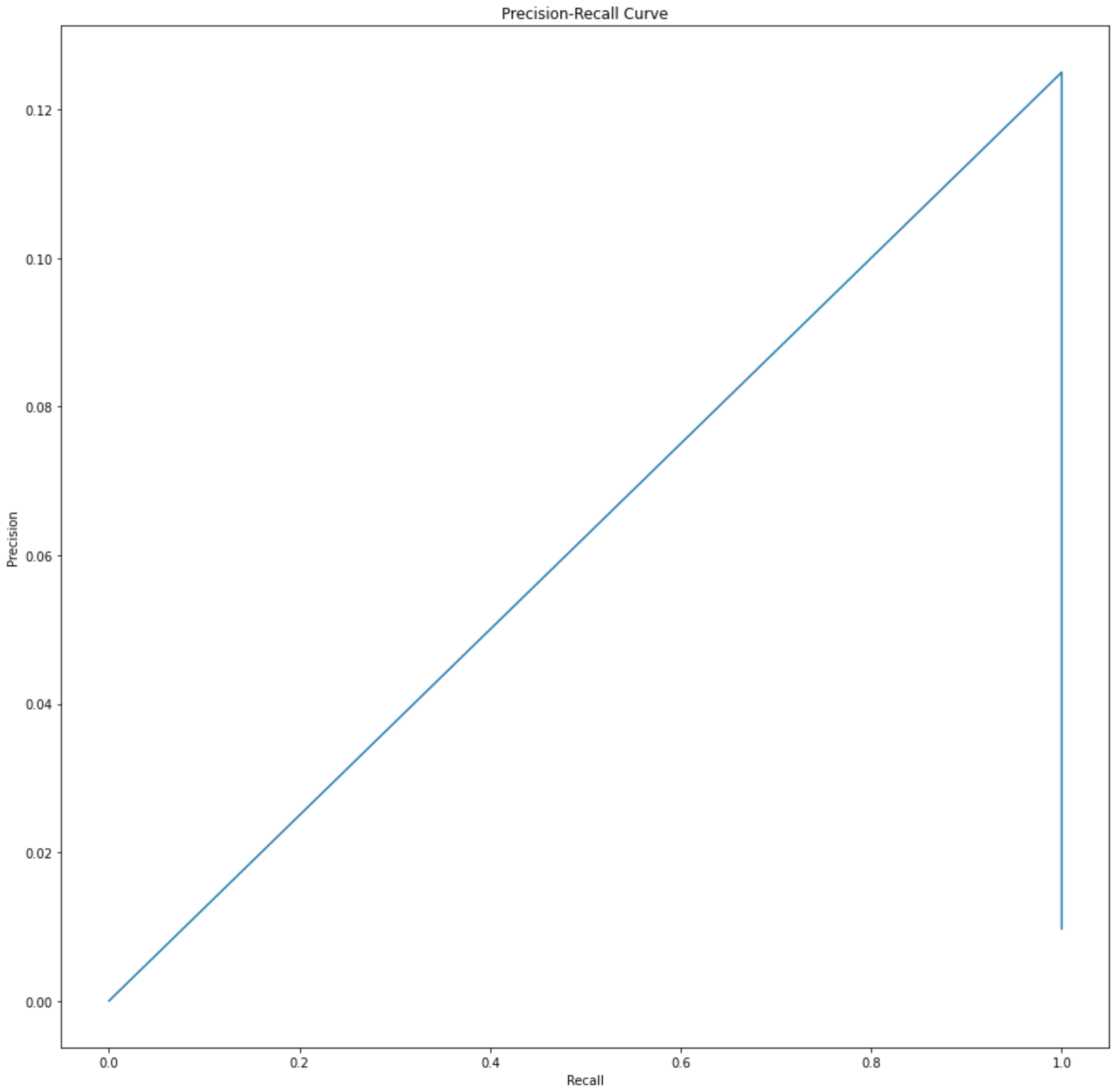
Plot for Relevance Threshold 0:



Plot for Relevance Threshold 1:



Plot for Relevance Threshold 2:



In information retrieval, the precision-recall (PR) curve is used to evaluate the effectiveness of a search engine or information retrieval system. In this context, precision is the fraction of retrieved documents that are relevant to the user's query, while recall is the fraction of relevant documents that are retrieved by the system.

The PR curve in information retrieval is similar to that in binary classification, where precision is plotted against recall at different decision thresholds. However, in this case, the goal is to maximize both precision and recall simultaneously, rather than finding the optimal trade-off between them.

Hence with lower thresholds we can see the curve is less spiky while with higher thresholds unless we find a highly relevant document the recall and precision do not increase making these spiky curves. Infact as we see more irrelevant documents both our precision decreases while recall stays the same giving such as a curve.

```
nDCG@50:  0.3521042740324887
nDCG@entire:  0.5979226516897831
```