Question 1:

We performed the following steps for Preprocessing :

- First step involves noise removal which includes removal of file headers, footers,markup data and extra spaces.
- Second step involves replacing contractions with their expansions, which should be done before tokenization.
- Third step involves tokenization using NLTK's word_tokenize()
- Fourth Step involves removal of non-ascii characters from the list of tokenized words.
- Fifth step involves conversion of each character to lowercase from list of tokenized words.
- Sixth step involves removal of punctuations from the list of tokenized words.
- Seventh step involves removal of stop words from the list of tokenized words.
- The Final step involves lemmatization, we tried using spacy lemmatizer.
- For all the documents we generated a dictionary, where document name is the key and value is the list of tuples of preprocessed words of that document along with word position, which is used for generating constructing positional index data structure.

We Performed the following for positional index data structure construction:

- constructing a dictionary (positional index) containing all the unique terms present inside all the given documents along with a positional posting list for every term. The list corresponding to a particular term holds the term count and the document id of the documents where the term is present along with the term position in that document.
- As part of the process, every document is given a unique document id.
- The text inside every document is read, if a particular term is not present in the positional index, the term is added and the document id of the document in which it is present is also added along with the position. If the term is present then only the document id with position is added to the posting list of that term. Along with this we calculate the term frequency.
- The posting lists are sorted in increasing order of document ids for reducing the number of comparisons.

Query processing:

- Suitable preprocessing systems as explained above for the documents are applied on the input query too.
- The number of **documents matched**, and the **retrieved documents** are displayed immediately after each query.

Phrase queries:

- This step needs positional index dictionary along with document id to name mapping.
- First we need the documents that all query terms appear in and again get the list of documents for each query term and performing intersection

  .

- We should check whether they are in correct order or not, to perform this for each document that contains all query terms, we should get positions of the query terms in the current document, and put each to a separate list and if there are k query terms, there will be k lists where each list contains the positions of the corresponding query term in the current document.
- Now leave the position list of the 1st query term as it is, subtract 1 from each element of the 2nd position list, subtract 2 from each element of the 3rd position list, …, subtract k-1 from each element of kth position list.
- Intersect all the position lists and if the result of the intersection is non-empty, then all query terms appear in the current document in correct order, meaning this is a matching document.

Assumption:

- We used Spacy lemmatizer during preprocessing, sometimes based on context the query terms may lemmatize to different forms because of the POS tag.

Question 2:

In this question there are three parts given. Here we first discuss computing query document similarity using Jaccard coefficient:

1) We perform preprocessing on the given 467 documents. We follow the steps in assignment 1 which includes noise removal (footer, markup etc.), stop word removal, contractions replacement with expansions, lower case conversion and non-ascii character removal and lemmatization.
2) Then an inverted index is formed. This will help us later in reducing the number of documents to compare the query with. This optimization is also made optional and can be toggled by the user. The user may choose to first retrieve the documents that contain at least one term in the given query by applying the OR operator on postings for terms in the query. Then the Jaccard similarity is only computed between the set formed from the given **query** and the sets formed from each **document** retrieved from the inverted index. The user may also choose to compute Jaccard similarity with every document where the disadvantage is that computation is wasted on documents that may not contain even one term in the query. This behaviour is controlled by a toggle variable named **optimization.**
3) We implement the Jaccard similarity function which enables to calculate similarity between documents or query and each document in our case as intersection over union of a set of shingles or n-grams extracted from the query and document. Consider A and B to be two sets consisting of words then

   JACCARD(A,B) = |A intersection B| / |A union B|

4) Then the documents are sorted according to Jaccard similarity scores with the query and top 5 relevant documents are given as output.

Pros:

Jaccard similarity is simple and efficient to compute. It does not require computation of matrix multiplication or arithmetic and only requires set intersection and union operations to be performed.

Cons:
Jaccard similarity or coefficient does not consider the term frequency. Also rare words are more important than frequent words. Jaccard similarity has no mechanism to capture this information. Also when one of the sets are two large it will have little overlap with the smaller set leading to a small value for the Jaccard similarity.

Part 2: TF-IDF Matrix:

The first step for calculating the document scores is to pre-process the data using appropriate techniques. After pre-processing, a posting list is created for every unique term in the vocabulary. There are 467 documents which are mapped to a unique number, similarly all the unique vocabulary terms are mapped to unique integers starting from 0. A term frequency matrix is created with unique terms represented as rows and documents are represented by columns. Each entry (i, j) of the term frequency matrix represents the number of times ith term occurs in jth document. For binary specifications we keep 1, for the jth document that contains the ith term and rest of the cells contains 0. The document scores are computed using term frequency weights (depends on the scheme followed) * inverse document frequency. The top 5 documents with maximum score are retrieved. The terms and documents can be retrieved from their respective mappings.

Pros & Cons:

There are 5 five schemes to be followed for calculating the term frequency. They are mentioned and discussed below

The binary weighting scheme is very basic. Only the terms that are present in the document are marked as 1 for that document. It fails to capture the importance of rare terms, and also it doesn't consider if a term is repeating many times then it may have some significance.

Raw count weighting scheme works on the idea that, if a term is repeated multiple times in a document then it is possible that the document could be more relevant as compared to the other documents. But there is a drawback that, because of the high frequency term the document score will be very high for a document just because it contains one of the query terms.

The term frequency term tries to normalize the terms with high frequency and reduce the bias towards the high frequency terms, but it is impacted by the presence of irrelevant terms in the document, if the document is very large then it will impact the frequency score more than the document that doesn't have a large size.

Log Normalized scheme normalises the frequency terms in much more unbiased manner as compared to the above mentioned schemes. It is not impacted by other terms and their frequencies.

Double Normalization is similar to the term frequency scheme, but it differs on the basis of normalizing on the basis of the highest frequency in that document. It can be biased towards outliers as a document may contain an outlier term with a large number of occurrences, that may not represent the document relevance also, but has direct impact on the term frequency score.

Part 3: Cosine similarity:
The query vector and the document vectors are computed using the tf weighting schemes followed by multiplication with the idf score for the term. The document vector is of size (number_of_docs, vocabulary_size) and the query vector is of size (1,vocabulary_size). The snippet showing computation of cosine similarity is given below.

```
def cosine_similarity(query_vector, document_vector):
  numerator = np.dot(query_vector,document_vector.T)
  denominator = np.linalg.norm(query_vector,axis=1)* np.linalg.norm(document_vector,axis=1)
  cosine_similarity = numerator/denominator
  cosine_similarity = cosine_similarity.squeeze()
  return cosine_similarity
```

Here also we provide two choices to the end user. One can choose to optimize by computing cosine score between the query vector and only the document vectors that contain at least one query term. Or the end user can choose to compute cosine scores between the query and all the documents. Finally top 5 relevant documents along with their cosine scores are given as output.

Question 3:
The dataset is highly tuned towards Information Retrieval tasks. There is a total of 136 features ranging from TD-IDF scores to BM25. It is highly detailed annotation and is applicable for a wide range of tasks. In this question we make use of Distributed Cumulative gain to rank the documents.

1)
Preprocessing:
The dataset is a text file with reach record in a new line and each data point separated by a space. We convert them into an array of arrays, and include only “qid:4”.

2) The first index of each array is the relevance score as mentioned in the readme of dataset. In order to calculate the max DCG, we employ ideal ranking by returning max relevance score from remaining documents to return.
This is the distribution of max relevance score of documents:

Documents with relevance score:
0 = 59 documents
1 = 26 documents
2 = 17 documents
3 = 1 documents

To find number of such arrangements possible with maximum DCG, we apply permutations among documents with same relevance score. Hence, it would be 26!*17!

3) To find nDCG at 50, we take the ideal ranking and calculate the DCG score for the entire document set of “qid:4” which is 103 documents.

Assumptions:
The document order that was present in the raw dataset is assumed to be the given order for ranking, for which the normalized DCG is calculated.

For whole dataset, we simply consider the entire dataset of document order.

4) Simply sorting the set of documents with feature 75 gives the requisite ranking. Plotting the precision-recall curve, given the limited documents, we have a curve that shuttles between 0.4-0.5 and recall reaches 1 as expected.