Q1- Part A (Jaccard Coefficient)

```
def preprocessing(d):
    no_punc = []
    d_lower=d.lower()

    nltk_tokens = nltk.word_tokenize(d_lower)

    stop_words_removed = []
    for w in nltk_tokens:
        if w not in stop_words:
            stop_words_removed.append(w)


    new_words = []
    for x in stop_words_removed:
        if(x.isalnum() and x!=" "):
            new_words.append(x)


    return new_words
```

Preprocessing is being performed by removing stopwords , punctuations , converting to lower case as well as tokenization of sentences.

```
jaccard_score = []
for file_data in preprocessed_data:
    intersection = list(set(file_data) & set(preprocessed_query))
    union = list(set().union(file_data, preprocessed_query))
    js = len(intersection)/len(union)
    jaccard_score.append(js)
```

Jaccard score for each document is being calculated with respect to query entered by the user and is being stored in a list called jaccard_score list and in accordance with this list 5 documents with highest scores are printed as shown in below image.

```
i=0
for i in range(5):
    print(Z[i])
```

```
justify
footfun.hum
naivewiz.hum
lobquad.hum
roach.asc
```

Q1- Part A (TF-IDF Score)

```
def preprocessing(d):
    no_punc = []
    d_lower=d.lower()

    nltk_tokens = nltk.word_tokenize(d_lower)

    stop_words_removed = []
    for w in nltk_tokens:
        if w not in stop_words:
            stop_words_removed.append(w)


    new_words = []
    for x in stop_words_removed:
        if(x.isalnum() and x!=" "):
            new_words.append(x)


    return new_words
```

Preprocessing is being performed by removing stopwords , punctuations , converting to lower case as well as tokenization of sentences.

```
vocab_list = []
count = 0
for p in preprocessed_data :
    i=0
    for i in range(len(p)):
        if not p[i] in vocab_list:
            count=count+1
            vocab_list.append(p[i])
print(count)
```

Generating a general vocabulary for all the words in the documents.

```
doc_freq_list = []
for v in vocab_list:
    counter = 0
    for p in preprocessed_data:
        if v in p:
            counter=counter+1

    doc_freq_list.append(counter)
```

Creating a list which is taking into account number of docs in which a term is present

```
total_docs = len(file_names)
#i=0
for i in range(len(v_list)):
    idf = math.log10(total_docs/(new_doc_freq_list[i]+1))
    #j=0
    for j in range(len(file_names)):
        tf_idf_matrix[j][i] = idf
```

Generation of IDF matrix which will be common for all weighing schemes

Method 1 (Binary Weighing Scheme) :

```
tfid matrix with term weighing scheme (Binary)
for i in range(len(v_list)):
    word = v_list[i]
    for j in range(len(file_names)):
        if word not in preprocessed_data[j]:
            tf_idf_matrix1[j][i] = 0
```

Pros :

1)Fast Method

2)Reduces Feature vector

Cons:

1) **Does'nt weigh terms according to its importance.**
2) **Frequency of a term is not given importance.**

Method 2(Raw Count)

```
# tfid matrix with term weighing scheme (Raw count)
for i in range(len(v_list)):
    word = v_list[i]
    for j in range(len(file_names)):
        if word not in preprocessed_data[j]:
            tf_idf_matrix2[j][i] = 0
        else:
            tf_idf_matrix2[j][i] = tf_idf_matrix2[j][i] * preprocessed_data[j].count(word)
```

Pros :

1)It takes into account frequency of each term hence weighs it according to importance

Cons:

1) **But sometimes it gives too much of importance to to a term which is part of query as its frequency is very high in the document.**

Method 3(Term frequency)

```
# tfid matrix with term weighing scheme (Term frequency)
tf_idf_3 = tf_idf_2
for i in range(len(file_names)):
    doc_length = len(preprocessed_data[i])
    for j in range(len(v_list)):
        tf_idf_3[i][j] = tf_idf_3[i][j]/doc_length
```

Pros:

1) **This scheme gives importance to length of document , documents which are small and contains query are favoured in order to tackle more frequency problem.**

Cons :

2) **Sometimes it gives undue advantage to those docs which are small but do not contain complete query compared to those docs which are very big but contain particular query.**

Method 4(Log Normalization)

```
for i in range(len(v_list)):
    word = v_list[i]
    for j in range(len(file_names)):
        if word not in preprocessed_data[j]:
            tf_idf_matrix4[j][i] = 0
        else:
            tf_idf_matrix4[j][i] = tf_idf_matrix4[j][i] * math.log10(1+preprocessed_data[j].count(word))
```

Pros:

1)It takes into account frequency of each term hence weighs it according to importance and normalizes it to small values.

2) Cons:

1) **But sometimes it gives too much of importance to to a term which is part of query as its frequency is very high in the document.**

Method 5(Double Normalization)

```
# tfid matrix with term weighing scheme (Double Normalization)
for i in range(len(v_list)):
    word = v_list[i]
    for j in range(len(file_names)):
        if word not in preprocessed_data[j]:
            tf_idf_matrix5[j][i] = tf_idf_matrix5[j][i] * 0.5
        else:
            most_freq_word = mode(preprocessed_data[j])
            most_freq_count = preprocessed_data[j].count(most_freq_word)
            tf_idf_matrix5[j][i] = tf_idf_matrix5[j][i] * (0.5 + 0.5*(preprocessed_data[j].count(word)/most_freq_count))
```

Pros:

1) **Here it penalizes the word which is having very high frequency , which helps not to give unnecessary importance to those words which are not required.**

Cons:

1) **But sometimes there comes an outlier when score of doc gets dependent on most frequently occurring term.**

Ques2-

1. **Considering only the queries with qid=4.**

We have split query-url pairs and considered the one having qid==4.

```
for line in f:
    line_words = line.split()
    if line_words[1] == 'qid:4':
        doc_del.append(line_words)
    else:
        break
```

2. **Finding the DCG of the whole Dataset**

DCG of the whole dataset is calculated and the respective count is calculated.

```
dcgWD=[]
for i in range(len(doc_del)):
    dcgWD.append((float(doc_del[i][0]))/math.log(i+2,2))
```

```
dic
```

```
{0.9463946303571861: 1,
 0.4627564263195183: 1,
 0.455340497393906: 1,
 0.4421294589150075: 1,
 0.43620858397106305: 1,
 0.4206198357143049: 1,
 0.38685280723454163: 2,
 0.3811028248535468: 1,
 0.37840071903374006: 1,
 0.3708980468307378: 1,
 0.34753068574288: 1,
 0.33858761519756286: 1,
```

This dictionary displayed the sorted values of dcg and their corresponding count.
To calculate the number of files possible with file rearranging the query-URL pairs in order of max DCG.

We have calculated the factorial of the count and got the below result:

```
ans
```

```
138683118545689835737939019720389406345902876772687432540821294940160000000000004
```

The answer is too huge because of the presence of a huge no. of qid=='0'.

3) i) <u>Computing ndcg at 50.</u>

```python
for i in range(50):
    dcg50.append((float(doc_del2[i][0]))/math.log(i+2,2))
    #calculate idcg
    idcg50.append((float(doc_del[i][0]))/math.log(i+2,2))
    # summision of dcg
    totaldcg50 = totaldcg50 + dcg50[i]
    # summision of idcg
    totalidcg50 = totalidcg50 + idcg50[i]
    #calculate ndcg
    ndcg50=float(totaldcg50/(totalidcg50+1))
```

Result Obtained:

Ndcg at 50= 0.35

```
sum of dcg is 7.19450227631398

sum of idcg is 19.407247618668023

ndcg at 50 is 0.3525464291290627
```

3) ii) <u>Computing ndcg of the Whole DataSet:</u>

```
for i in range(len(doc_del)):
    dcgWD.append((float(doc_del2[i][0]))/math.log(i+2,2))
    #calculate idcg
    idcgWD.append((float(doc_del[i][0]))/math.log(i+2,2))
    # summision of dcg
    totaldcgWD = totaldcgWD + dcgWD[i]
    # summision of idcg
    totalidcgWD = totalidcgWD + idcgWD[i]
    #calculate ndcg
    ndcgWD=float(totaldcgWD/(totalidcgWD+1))

print("dcg is",dcgWD)
```

The following Result is Obtained-

Ndcg of the whole Dataset- 0.60

```
sum of dcg is 12.337484420604602

sum of idcg is 19.407247618668023

ndcg of whole dataset is 0.6045638613861181
```

4). Finding Total relevant documents

Total relevant Documents

```
t_relevant=0
for i in dual:
    if(i[0]!=0):
        t_relevant=t_relevant+1
```

```
t_relevant
```

```
44
```

```
recall=[]
rel_docs=[]
for line in f:
    line_words = line.split()
    if (line_words[0]!='0'):
        rel_docs.append(line_words[76].split(":")[1])
        recall.append(len(pr)/len(rel_docs))
```

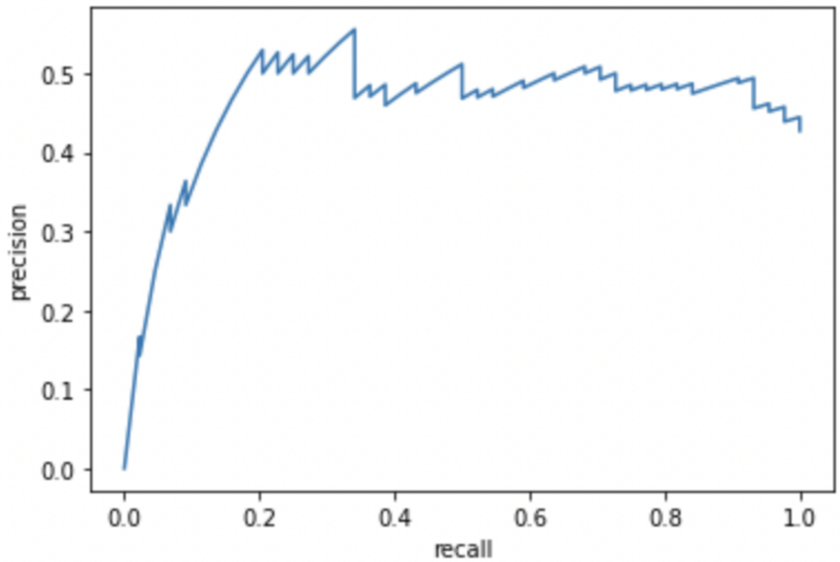
Finding precision and recall of all the documents

```
rel_ret=0
ret_docs=1
precision=[]
recall=[]

for i in dual:
    recall.append(rel_ret/t_relevant)
    precision.append(rel_ret/ret_docs)
    if(i[0]==1):
        rel_ret=rel_ret+1
    ret_docs=ret_docs+1

```

```
plt.plot(recall,precision)
plt.xlabel('recall')
plt.ylabel('precision')
plt.show()
```



precision
0.0
0.1
0.2
0.3
0.4
0.5
0.0
0.2
0.4
0.6
0.8
1.0
recall

Q3. Naive Bayes Classifier

Preprocessing :

```
def preprocessing(d):
    no_punc = []
    d_lower=d.lower()

    nltk_tokens = nltk.word_tokenize(d_lower)

    nltk_tokens=remove_num(nltk_tokens)

    nltk_tokens = remove_punc(nltk_tokens)

    stop_words_removed = []
    for w in nltk_tokens:
        if w not in stop_words:
            stop_words_removed.append(w)


    new_words = []
    for x in stop_words_removed:
        if(x.isalnum() and x!=" "):
            new_words.append(x)


    return new_words
```

Removing stopwords , lowering tokens , removing punctuations ,removing numbers

```python
def get_data_from_paths(paths):
    data=[]
    for i in range(len(paths)):
        try:
            f = open(str(paths[i]) , encoding='utf8')
            var=f.read().replace('\n'," ")
            data.append(var)

        except Exception as e:
            f = open(str(paths[i]) , encoding='unicode_escape')
            var=f.read().replace('\n'," ")
            data.append(var)
    return data
```

```python
data_from_file1 = get_data_from_paths(paths_for_file1)
data_from_file2 = get_data_from_paths(paths_for_file2)
data_from_file3 = get_data_from_paths(paths_for_file3)
data_from_file4 = get_data_from_paths(paths_for_file4)
data_from_file5 = get_data_from_paths(paths_for_file5)
```

Extraction of data is being done from paths of all files inside 5 folders

```python
vocabulary = []
for data in all_data:
    for word in data:
        if word not in vocabulary:
            vocabulary.append(word)
```

Vocabulary generation is done here

```
class_freq_list=[]
for term in v_list:
    count = 0
    for file1 in data_from_file1_preprocessed:
        if term in file1:
            count=count+1
            break
    for file2 in data_from_file2_preprocessed:
        if term in file2:
            count=count+1
            break
    for file3 in data_from_file3_preprocessed:
        if term in file3:
            count=count+1
            break
    for file4 in data_from_file4_preprocessed:
        if term in file4:
            count=count+1
            break
    for file5 in data_from_file5_preprocessed:
        if term in file5:
            count=count+1
            break
    class_freq_list.append(count)
```

Calculation of number of classes in which a particular term occurs

```
#only entered ICF values of each term
for j in range(len(v_list)):
    for i in range(5):
        tf_icf_matrix[i][j] = math.log10(5/class_f_list[j])
```

Calculation of ICF value being done in this section and entered in a matrix called tc_icf_matrix

```
for j in range(len(v_list)):
    count1 = 0
    for file1 in data_from_file1_preprocessed:
        count1 = count1 + file1.count(v_list[j])
    tf_matrix[0][j] = count1

    count2=0
    for file2 in data_from_file2_preprocessed:
        count2 = count2 + file2.count(v_list[j])
    tf_matrix[1][j] = count2

    count3=0
    for file3 in data_from_file3_preprocessed:
        count3 = count3 + file3.count(v_list[j])
    tf_matrix[2][j] = count3

    count4=0
    for file4 in data_from_file4_preprocessed:
        count4 = count4 + file4.count(v_list[j])
    tf_matrix[3][j] = count4

    count5=0
    for file5 in data_from_file5_preprocessed:
        count5 = count5 + file5.count(v_list[j])
    tf_matrix[4][j] = count5
```

Term frequency for each term in each class is calculated and stored in tf_matrix which will be multiplied with tc_icf_matrix matrix to get final tc_icf_matrix which will be used further

```python
v_list1 = v_list
tc_icf_scores_for_class1=[]
for j in range(len(v_list)):
    tc_icf_scores_for_class1.append(tc_icf_mat[0][j])

v_list2 = v_list
tc_icf_scores_for_class2=[]
for j in range(len(v_list)):
    tc_icf_scores_for_class2.append(tc_icf_mat[1][j])

v_list3 = v_list
tc_icf_scores_for_class3=[]
for j in range(len(v_list)):
    tc_icf_scores_for_class3.append(tc_icf_mat[2][j])

v_list4 = v_list
tc_icf_scores_for_class4=[]
for j in range(len(v_list)):
    tc_icf_scores_for_class4.append(tc_icf_mat[3][j])

v_list5 = v_list
tc_icf_scores_for_class5=[]
for j in range(len(v_list)):
    tc_icf_scores_for_class5.append(tc_icf_mat[4][j])
```

```python
#sorting vocab list wrt tc_icf_score
v_list1_new = [v_list1 for _,v_list1 in sorted(zip(tc_icf_scores_for_class1,v_list1))]
v_list2_new = [v_list2 for _,v_list2 in sorted(zip(tc_icf_scores_for_class2,v_list2))]
v_list3_new = [v_list3 for _,v_list3 in sorted(zip(tc_icf_scores_for_class3,v_list3))]
v_list4_new = [v_list4 for _,v_list4 in sorted(zip(tc_icf_scores_for_class4,v_list4))]
v_list5_new = [v_list5 for _,v_list5 in sorted(zip(tc_icf_scores_for_class5,v_list5))]
```

Tc_icf_scores for each class is extracted and on the basis of that vocab list is sorted for each class according to scores and union of all 5 vocabularies is done to get final vocabulary which will form features of our final feature_matrix

```python
for i in range(1000):
    for j in range(len(final_vocab_list)):
        if final_vocab_list[j] in all_data[i]:
            ind = v_list.index(final_vocab_list[j])
            feature_matrix[i][j] = tc_icf_mat[0][ind]

for i in range(1000,2000):
    for j in range(len(final_vocab_list)):
        if final_vocab_list[j] in all_data[i]:
            ind = v_list.index(final_vocab_list[j])
            feature_matrix[i][j] = tc_icf_mat[1][ind]

for i in range(2000,3000):
    for j in range(len(final_vocab_list)):
        if final_vocab_list[j] in all_data[i]:
            ind = v_list.index(final_vocab_list[j])
            feature_matrix[i][j] = tc_icf_mat[2][ind]

for i in range(3000,4000):
    for j in range(len(final_vocab_list)):
        if final_vocab_list[j] in all_data[i]:
            ind = v_list.index(final_vocab_list[j])
            feature_matrix[i][j] = tc_icf_mat[3][ind]

for i in range(4000,5000):
    for j in range(len(final_vocab_list)):
        if final_vocab_list[j] in all_data[i]:
            ind = v_list.index(final_vocab_list[j])
            feature_matrix[i][j] = tc_icf_mat[4][ind]
```

Creation of feature matrix is being done here which will be finally used in classification using Naïve bayes

```python
class NBC():

    def prior_probability_calculation(self, features, target):
        self.prior = (features.groupby(target).apply(lambda x: len(x)) / self.rows).to_numpy()
        return self.prior

    def summary_calculation(self, features, target):
        self.mean = features.groupby(target).apply(np.mean).to_numpy()
        self.var = features.groupby(target).apply(np.var).to_numpy()
        return self.mean, self.var

    def prob_density_distri(self, index_of_class, x):
        mean = self.mean[index_of_class]
        var = self.var[index_of_class]
        var=var+2
        numerator = np.exp((-1/2)*((x-mean)**2) / (2 * var))
        denominator = np.sqrt(2 * np.pi * var)
        prob = numerator / denominator
        return prob

    def posterior_probability_calculation(self, x):
        posteriors = []
        for i in range(self.count):
            prior = np.log(self.prior[i])
            conditional = np.sum(np.log(self.prob_density_distri(i, x)))
            posterior = prior + conditional
            posteriors.append(posterior)
        return self.classes[np.argmax(posteriors)]

    def fit(self, features, target):
        self.classes = np.unique(target)
        self.count = len(self.classes)
        self.feature_nums = features.shape[1]
        self.rows = features.shape[0]
        self.summary_calculation(features, target)
        self.prior_probability_calculation(features, target)

    def predict(self, features):
        preds = [self.posterior_probability_calculation(f) for f in features.to_numpy()]
        return preds
```

Naïve Bayes implementation to classify text into 5 different classes