Assignment - 2

Aayush Kumar : MT22001 Abhinav Joshi MT22092

Question 1

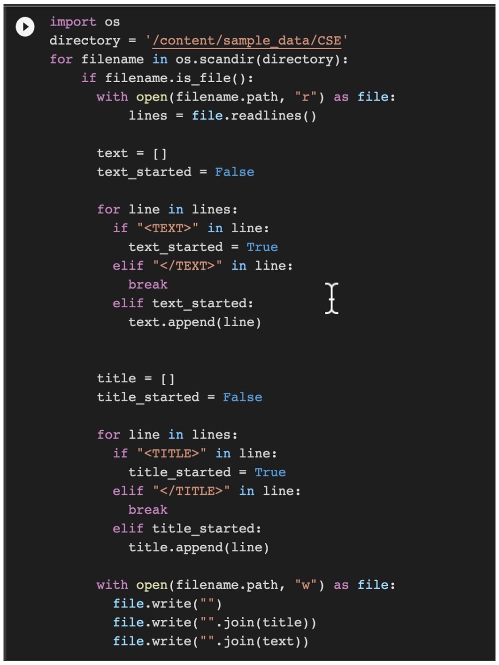
1. Preprocessing the dataset:

1.1-Relevant Text Extraction: For each file, extracting the contents between the <TITLE>...</TITLE> and <TEXT>...</TEXT>

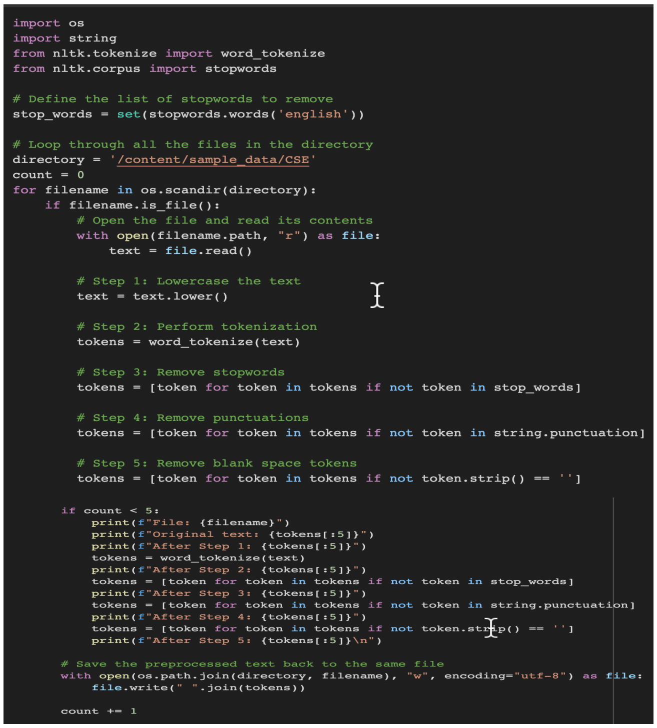
1.2-Preprocessing Text:

1.2.1. Lowercase the text
1.2.2. Perform tokenization
1.2.3. Remove stopwords
1.2.4. Remove punctuations
1.2.5. Remove blank space tokens

It is implemented using the following code.



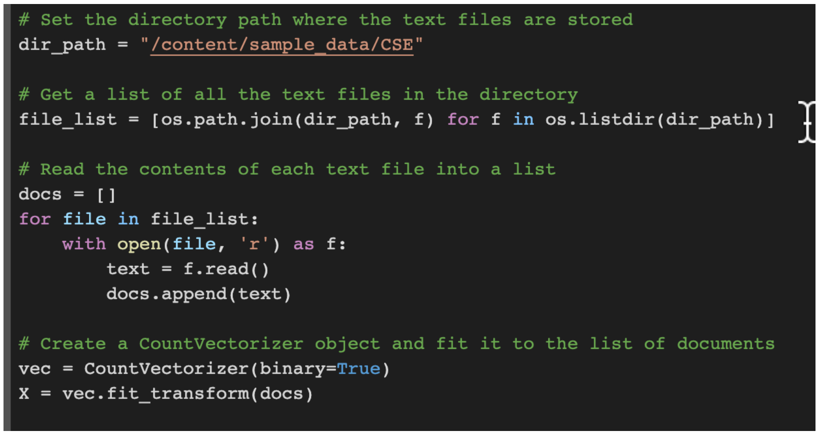
```python
import os
directory = '/content/sample_data/CSE'
for filename in os.scandir(directory):
    if filename.is_file():
      with open(filename.path, "r") as file:
          lines = file.readlines()

      text = []
      text_started = False

      for line in lines:
        if "<TEXT>" in line:
          text_started = True
        elif "</TEXT>" in line:
          break
        elif text_started:
          text.append(line)


      title = []
      title_started = False

      for line in lines:
        if "<TITLE>" in line:
          title_started = True
        elif "</TITLE>" in line:
          break
        elif title_started:
          title.append(line)

      with open(filename.path, "w") as file:
        file.write("")
        file.write("".join(title))
        file.write("".join(text))
```



```python
import os
import string
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

# Define the list of stopwords to remove
stop_words = set(stopwords.words('english'))

# Loop through all the files in the directory
directory = '/content/sample_data/CSE'
count = 0
for filename in os.scandir(directory):
    if filename.is_file():
        # Open the file and read its contents
        with open(filename.path, "r") as file:
            text = file.read()

        # Step 1: Lowercase the text
        text = text.lower()

        # Step 2: Perform tokenization
        tokens = word_tokenize(text)

        # Step 3: Remove stopwords
        tokens = [token for token in tokens if not token in stop_words]

        # Step 4: Remove punctuations
        tokens = [token for token in tokens if not token in string.punctuation]

        # Step 5: Remove blank space tokens
        tokens = [token for token in tokens if not token.strip() == '']

        if count < 5:
            print(f"File: {filename}")
            print(f"Original text: {tokens[:5]}")
            print(f"After Step 1: {tokens[:5]}")
            tokens = word_tokenize(text)
            print(f"After Step 2: {tokens[:5]}")
            tokens = [token for token in tokens if not token in stop_words]
            print(f"After Step 3: {tokens[:5]}")
            tokens = [token for token in tokens if not token in string.punctuation]
            print(f"After Step 4: {tokens[:5]}")
            tokens = [token for token in tokens if not token.strip() == '']
            print(f"After Step 5: {tokens[:5]}\n")

        # Save the preprocessed text back to the same file
        with open(os.path.join(directory, filename), "w", encoding="utf-8") as file:
            file.write(" ".join(tokens))

        count += 1
```

2.TF-IDF Binary : This code reads text files in a directory and uses the CountVectorizer function from scikit-learn to create a binary count vector for each file. The binary count vector represents the occurrence of each word in the document as either 0 or 1. The IDF (inverse document frequency) values for each term in the documents are then calculated. The TF (term frequency) values are calculated using a binary scheme, where each term is represented as either 0 or 1, depending on whether it occurs in the document or not. The TF-IDF scores are calculated by multiplying the TF and IDF values for each term in each document. The resulting scores are stored in a Pandas DataFrame for further analysis.

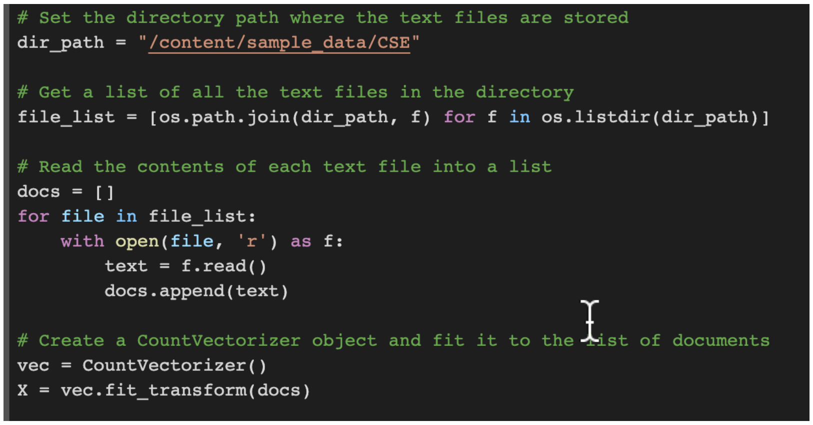
Tf binary is implemented using the following code.



```
# Set the directory path where the text files are stored
dir_path = "/content/sample_data/CSE"

# Get a list of all the text files in the directory
file_list = [os.path.join(dir_path, f) for f in os.listdir(dir_path)]

# Read the contents of each text file into a list
docs = []
for file in file_list:
    with open(file, 'r') as f:
        text = f.read()
        docs.append(text)

# Create a CountVectorizer object and fit it to the list of documents
vec = CountVectorizer(binary=True)
X = vec.fit_transform(docs)
```

3.TF-IDF Raw Count : The code uses the raw count scheme to calculate the term frequency (TF) values for each term in a collection of text documents. The CountVectorizer object from the Scikit-learn library is used to create a bag-of-words representation of the documents, where each term is assigned a count value based on the number of times it occurs in the document. The TF values are then multiplied by the inverse document frequency (IDF) values to calculate the TF-IDF scores, which are a measure of the importance of a term in a document relative to its importance in the collection as a whole. The resulting scores are stored in a Pandas DataFrame for further analysis.

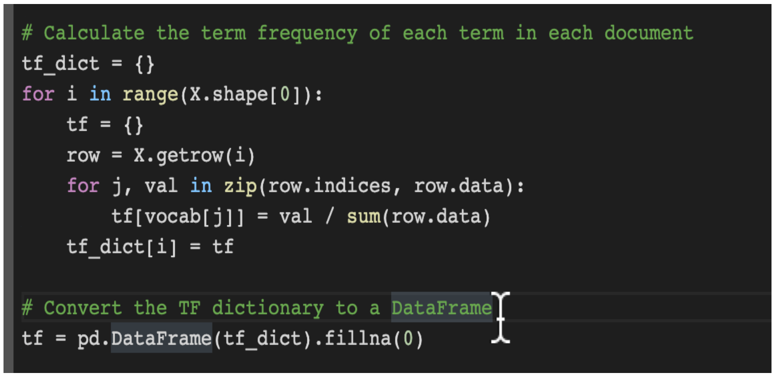
Tf raw count is implemented using the following code.



```
# Set the directory path where the text files are stored
dir_path = "/content/sample_data/CSE"

# Get a list of all the text files in the directory
file_list = [os.path.join(dir_path, f) for f in os.listdir(dir_path)]

# Read the contents of each text file into a list
docs = []
for file in file_list:
    with open(file, 'r') as f:
        text = f.read()
        docs.append(text)

# Create a CountVectorizer object and fit it to the list of documents
vec = CountVectorizer()
X = vec.fit_transform(docs)
```

4.TF-IDF Term Frequency : This code calculates the term frequency of each term in each document, where the term frequency is the number of times a term appears in a document divided by the total number of terms in the document. It creates a dictionary to store the term frequency of each term in each document and converts it to a Pandas DataFrame. The resulting DataFrame has the term frequency of each term in each document, with the columns representing the documents and the rows representing the terms in the vocabulary.

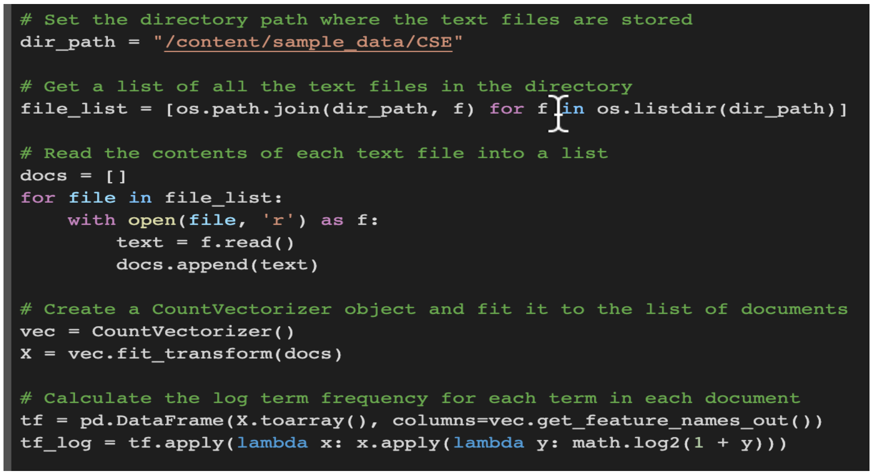
Tf Term frequency is implemented using the following code.



```
# Calculate the term frequency of each term in each document
tf_dict = {}
for i in range(X.shape[0]):
    tf = {}
    row = X.getrow(i)
    for j, val in zip(row.indices, row.data):
        tf[vocab[j]] = val / sum(row.data)
    tf_dict[i] = tf

# Convert the TF dictionary to a DataFrame
tf = pd.DataFrame(tf_dict).fillna(0)
```

5.TF-IDF Log Normalization : This code performs TF-IDF weighting on a set of text documents stored in a directory. The code uses the natural logarithm of the term frequency of each term in each document. The document frequency of each term is calculated and used to compute the inverse document frequency. The TF-IDF score for each term in each document is then calculated by multiplying the log term frequency with the inverse document frequency. The output is a pandas dataframe with the TF-IDF score of each term in each document. The code uses the CountVectorizer function from scikit-learn to tokenize the text documents and calculate the term frequency.

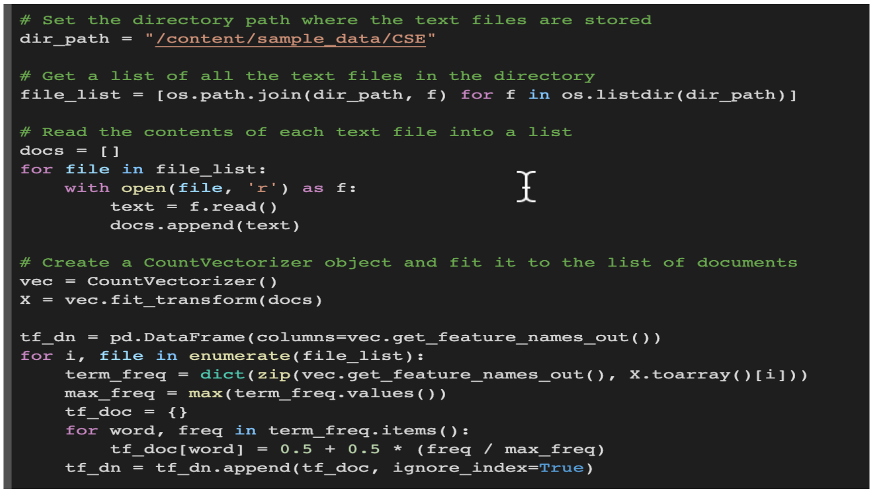
Tf log normalization is implemented using the following code.



```
# Set the directory path where the text files are stored
dir_path = "/content/sample_data/CSE"

# Get a list of all the text files in the directory
file_list = [os.path.join(dir_path, f) for f in os.listdir(dir_path)]

# Read the contents of each text file into a list
docs = []
for file in file_list:
    with open(file, 'r') as f:
        text = f.read()
        docs.append(text)

# Create a CountVectorizer object and fit it to the list of documents
vec = CountVectorizer()
X = vec.fit_transform(docs)

# Calculate the log term frequency for each term in each document
tf = pd.DataFrame(X.toarray(), columns=vec.get_feature_names_out())
tf_log = tf.apply(lambda x: x.apply(lambda y: math.log2(1 + y)))
```

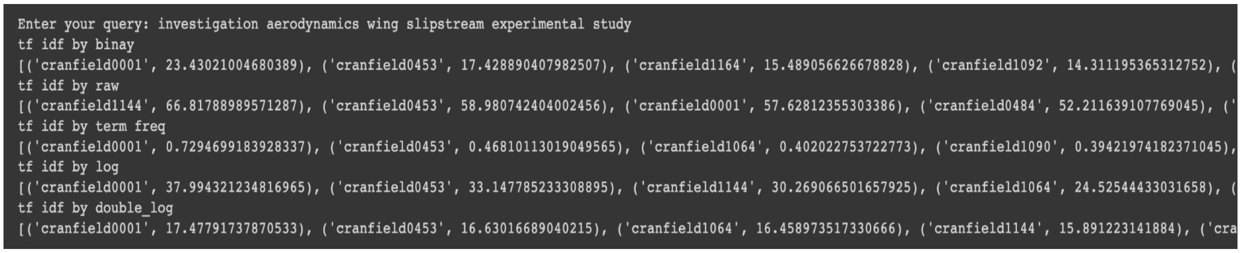
6.TF-IDF Double Log Normalization : The code calculates the term frequency (TF) of each word in each document using the maximum frequency of any word in that document. For each document, the maximum frequency is determined and the TF value for each word is calculated as 0.5 plus 0.5 times the frequency of the word divided by the maximum frequency.The TF values are then multiplied by the inverse document frequency (IDF) to calculate the TF-IDF score of each word in each document. The output is a pandas dataframe containing the TF-IDF score of each word in each document.

Tf double log normalization is implemented using the following code.



```
# Set the directory path where the text files are stored
dir_path = "/content/sample_data/CSE"

# Get a list of all the text files in the directory
file_list = [os.path.join(dir_path, f) for f in os.listdir(dir_path)]

# Read the contents of each text file into a list
docs = []
for file in file_list:
    with open(file, 'r') as f:
        text = f.read()
        docs.append(text)

# Create a CountVectorizer object and fit it to the list of documents
vec = CountVectorizer()
X = vec.fit_transform(docs)

tf_dn = pd.DataFrame(columns=vec.get_feature_names_out())
for i, file in enumerate(file_list):
    term_freq = dict(zip(vec.get_feature_names_out(), X.toarray()[i]))
    max_freq = max(term_freq.values())
    tf_doc = {}
    for word, freq in term_freq.items():
        tf_doc[word] = 0.5 + 0.5 * (freq / max_freq)
    tf_dn = tf_dn.append(tf_doc, ignore_index=True)
```

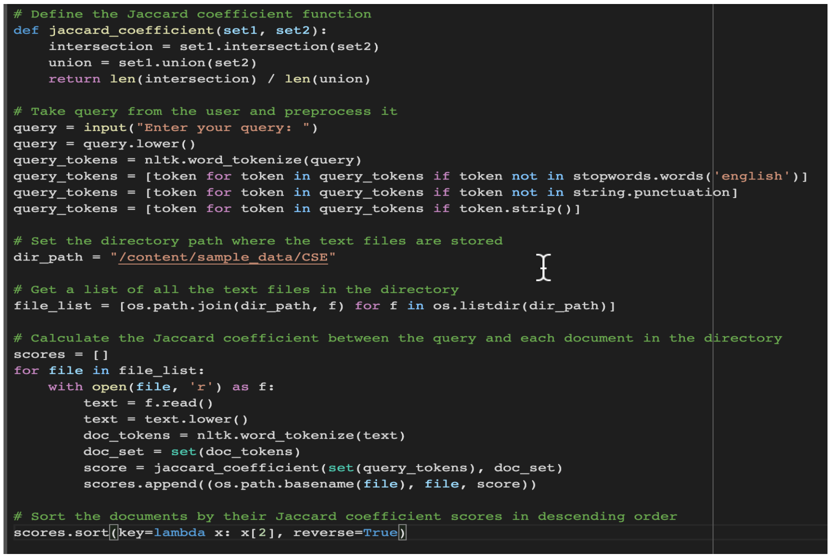
7.Query Output :



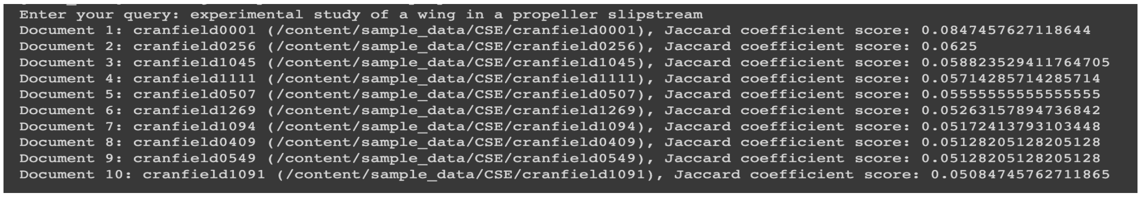
```
Enter your query: investigation aerodynamics wing slipstream experimental study
tf idf by binay
[('cranfield0001', 23.43021004680389), ('cranfield0453', 17.428890407982507), ('cranfield1164', 15.489056626678828), ('cranfield1092', 14.311195365312752), (
tf idf by raw
[('cranfield1144', 66.81788989571287), ('cranfield0453', 58.980742404002456), ('cranfield0001', 57.62812355303386), ('cranfield0484', 52.211639107769045), ('
tf idf by term freq
[('cranfield0001', 0.7294699183928337), ('cranfield0453', 0.46810113019049565), ('cranfield1064', 0.402022753722773), ('cranfield1090', 0.39421974182371045),
tf idf by log
[('cranfield0001', 37.994321234816965), ('cranfield0453', 33.147785233308895), ('cranfield1144', 30.269066501657925), ('cranfield1064', 24.52544433031658), (
tf idf by double_log
[('cranfield0001', 17.47791737870533), ('cranfield0453', 16.63016689040215), ('cranfield1064', 16.458973517330666), ('cranfield1144', 15.891223141884), ('cra
```

8.Jaccard Coefficient : This code allows users to enter a query and retrieves the top 5 most relevant documents based on the Jaccard coefficient score. First, it preprocesses the query by removing stopwords and punctuation, and then calculates the Jaccard coefficient between the query and each document in a given directory. The Jaccard coefficient measures the similarity between two sets of tokens, where a higher score indicates greater similarity. Finally, the documents are sorted in descending order based on their Jaccard coefficient scores, and the top 5 most relevant documents are printed out, along with their respective scores and file paths. The nltk library is used to tokenize the text and remove stopwords.

Jaccard Coefficient is implemented using the following code.



```
# Define the Jaccard coefficient function
def jaccard_coefficient(set1, set2):
    intersection = set1.intersection(set2)
    union = set1.union(set2)
    return len(intersection) / len(union)

# Take query from the user and preprocess it
query = input("Enter your query: ")
query = query.lower()
query_tokens = nltk.word_tokenize(query)
query_tokens = [token for token in query_tokens if token not in stopwords.words('english')]
query_tokens = [token for token in query_tokens if token not in string.punctuation]
query_tokens = [token for token in query_tokens if token.strip()]

# Set the directory path where the text files are stored
dir_path = "/content/sample_data/CSE"

# Get a list of all the text files in the directory
file_list = [os.path.join(dir_path, f) for f in os.listdir(dir_path)]

# Calculate the Jaccard coefficient between the query and each document in the directory
scores = []
for file in file_list:
    with open(file, 'r') as f:
        text = f.read()
        text = text.lower()
        doc_tokens = nltk.word_tokenize(text)
        doc_set = set(doc_tokens)
        score = jaccard_coefficient(set(query_tokens), doc_set)
        scores.append((os.path.basename(file), file, score))

# Sort the documents by their Jaccard coefficient scores in descending order
scores.sort(key=lambda x: x[2], reverse=True)
```



```
Enter your query: experimental study of a wing in a propeller slipstream
Document 1: cranfield0001 (/content/sample_data/CSE/cranfield0001), Jaccard coefficient score: 0.0847457627118644
Document 2: cranfield0256 (/content/sample_data/CSE/cranfield0256), Jaccard coefficient score: 0.0625
Document 3: cranfield1045 (/content/sample_data/CSE/cranfield1045), Jaccard coefficient score: 0.058823529411764705
Document 4: cranfield1111 (/content/sample_data/CSE/cranfield1111), Jaccard coefficient score: 0.05714285714285714
Document 5: cranfield0507 (/content/sample_data/CSE/cranfield0507), Jaccard coefficient score: 0.05555555555555555
Document 6: cranfield1269 (/content/sample_data/CSE/cranfield1269), Jaccard coefficient score: 0.05263157894736842
Document 7: cranfield1094 (/content/sample_data/CSE/cranfield1094), Jaccard coefficient score: 0.05172413793103448
Document 8: cranfield0409 (/content/sample_data/CSE/cranfield0409), Jaccard coefficient score: 0.05128205128205128
Document 9: cranfield0549 (/content/sample_data/CSE/cranfield0549), Jaccard coefficient score: 0.05128205128205128
Document 10: cranfield1091 (/content/sample_data/CSE/cranfield1091), Jaccard coefficient score: 0.05084745762711865
```

9. Advantages and disadvantages of different scoring schemes:

Pros of Jaccard Coefficient:

- The size of the two sets being compared can be different.
- The output value is always between 0 and 1.
- It can be used for categorical and continuous values.

Cons of Jaccard Coefficient:

- It does not consider term frequency, i.e., how many times a term occurs in a document.
- Jaccard Coefficient does not distinguish that terms appearing rarely provide more information than frequently appearing terms.

Pros of TF-IDF:

- It is easy to calculate the similarity between two documents.
- It helps to extract the most relevant terms from a document.

Cons of TF-IDF:

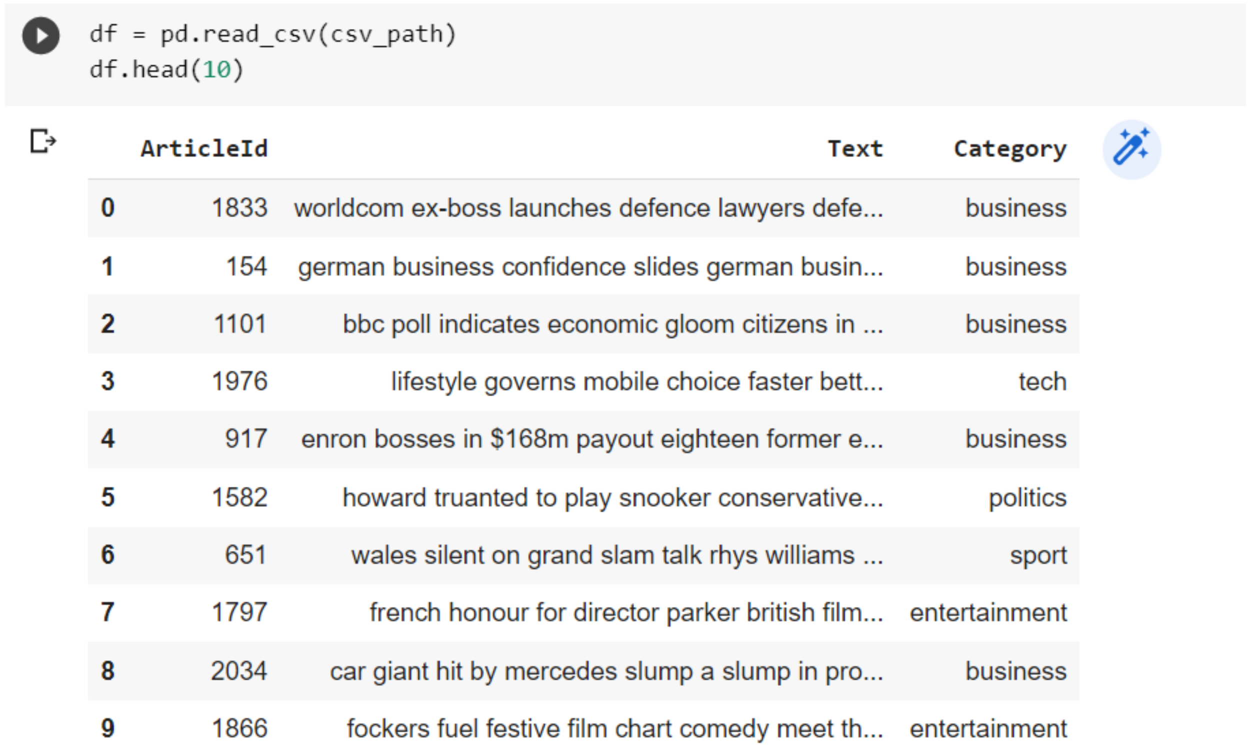
- It is based on the Bag of Words model, which does not consider the word order in the document.
- It does not account for the semantics and position of the terms in the document.

Question 2

1. Preprocessing the dataset:

● The dataset is in CSV format with the following columns: ‘ArticleId’, 'Text', and 'Category'.

The dataset is loaded using *pd.read_csv(filename)*



```
df = pd.read_csv(csv_path)
df.head(10)
```

	ArticleId	Text	Category
0	1833	worldcom ex-boss launches defence lawyers defe...	business
1	154	german business confidence slides german busin...	business
2	1101	bbc poll indicates economic gloom citizens in ...	business
3	1976	lifestyle governs mobile choice faster bett...	tech
4	917	enron bosses in $168m payout eighteen former e...	business
5	1582	howard truanted to play snooker conservative...	politics
6	651	wales silent on grand slam talk rhys williams ...	sport
7	1797	french honour for director parker british film...	entertainment
8	2034	car giant hit by mercedes slump a slump in pro...	business
9	1866	fockers fuel festive film chart comedy meet th...	entertainment

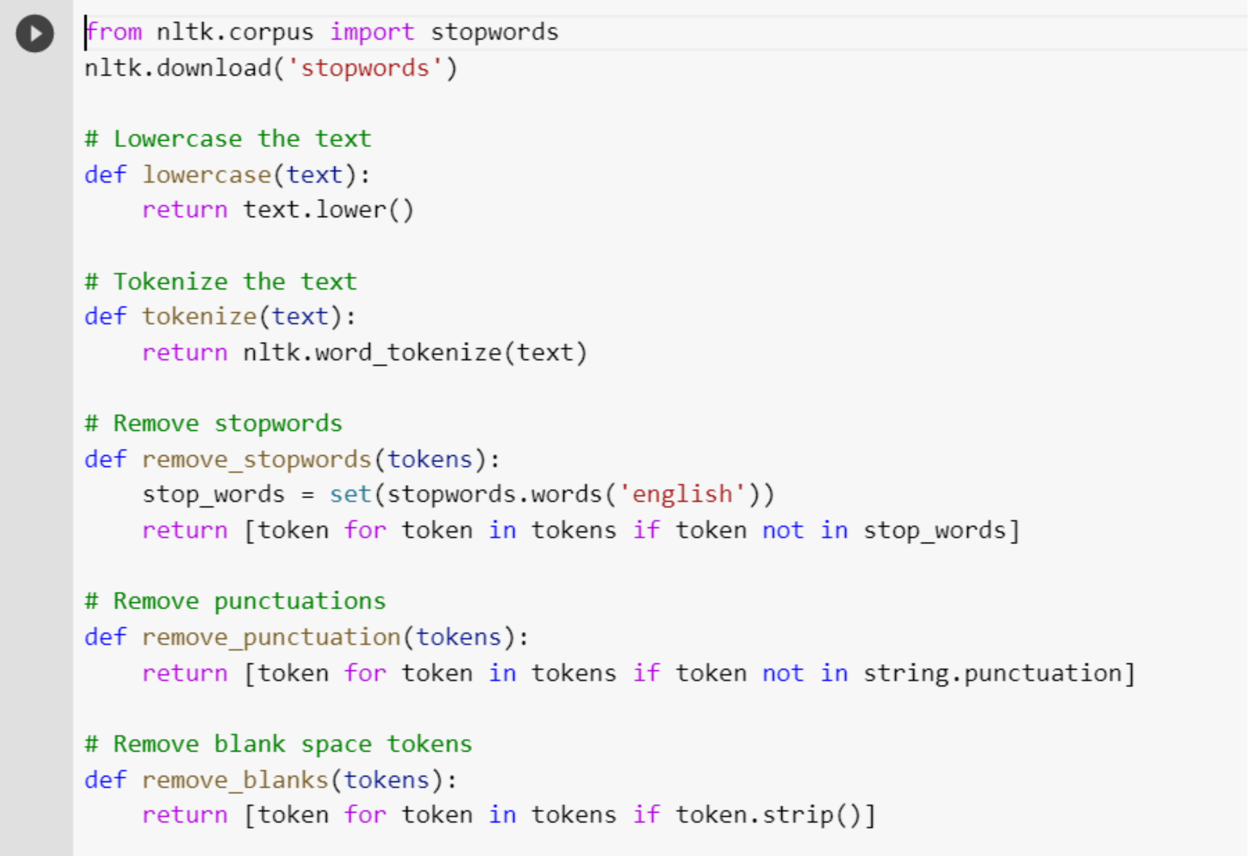
● Remove any unnecessary columns.

We drop the ArticleID column since it is not required.

df_new = df.drop(['ArticleId'], axis=1)

● Clean the text by removing punctuation, stop words, and converting all text to Lowercase.

We apply the pre-processing functions that were used in Assignment 1 for removing punctuation, stop words etc.



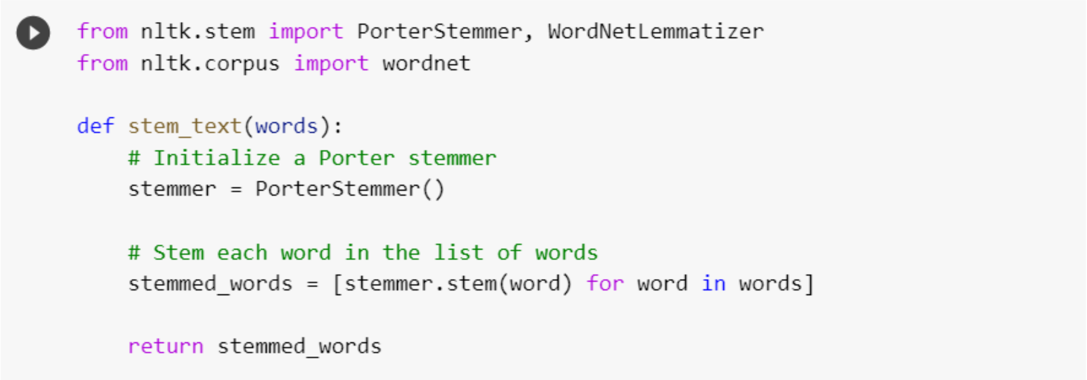
```python
from nltk.corpus import stopwords
nltk.download('stopwords')

# Lowercase the text
def lowercase(text):
    return text.lower()

# Tokenize the text
def tokenize(text):
    return nltk.word_tokenize(text)

# Remove stopwords
def remove_stopwords(tokens):
    stop_words = set(stopwords.words('english'))
    return [token for token in tokens if token not in stop_words]

# Remove punctuations
def remove_punctuation(tokens):
    return [token for token in tokens if token not in string.punctuation]

# Remove blank space tokens
def remove_blanks(tokens):
    return [token for token in tokens if token.strip()]
```

- Tokenize the text by splitting it into words.

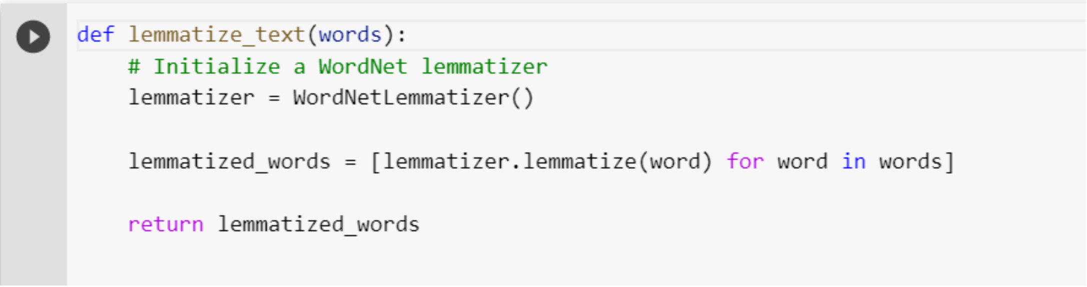
We tokenize the text using: nltk.word_tokenize

- Perform stemming or lemmatization to reduce words to their root form.

We use PorterStemmer and WordNetLemmatizer



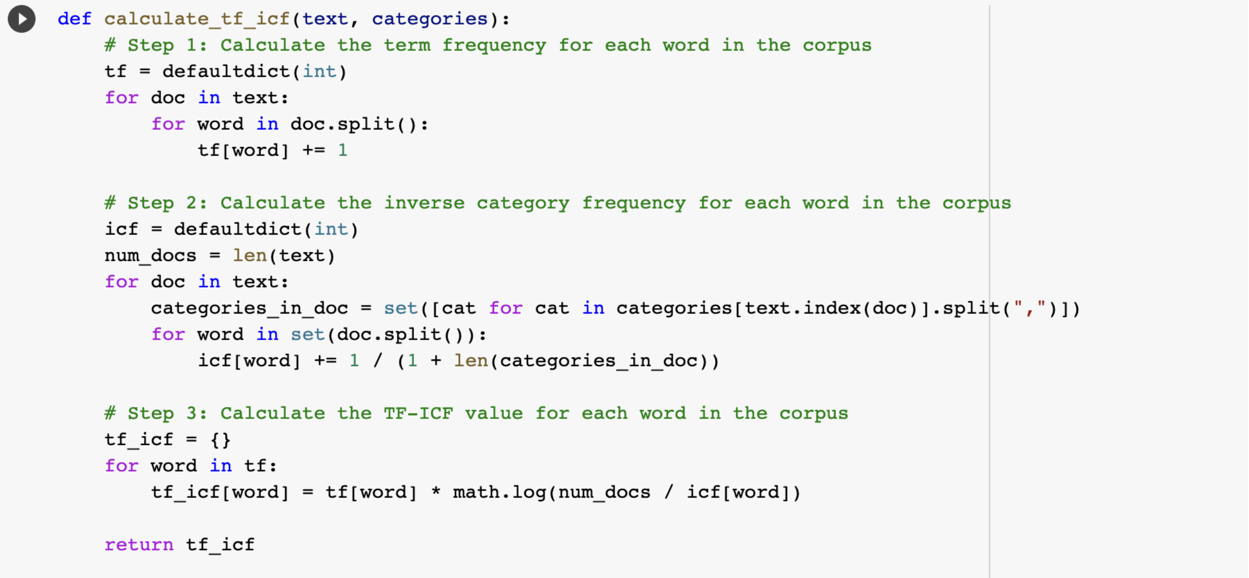
```python
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.corpus import wordnet

def stem_text(words):
    # Initialize a Porter stemmer
    stemmer = PorterStemmer()

    # Stem each word in the list of words
    stemmed_words = [stemmer.stem(word) for word in words]

    return stemmed_words
```



```python
def lemmatize_text(words):
    # Initialize a WordNet lemmatizer
    lemmatizer = WordNetLemmatizer()

    lemmatized_words = [lemmatizer.lemmatize(word) for word in words]

    return lemmatized_words
```

- Implement the TF-ICF weighting scheme.

It is implemented using the following code.



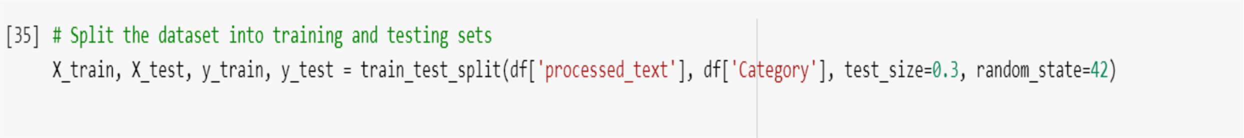
```
def calculate_tf_icf(text, categories):
    # Step 1: Calculate the term frequency for each word in the corpus
    tf = defaultdict(int)
    for doc in text:
        for word in doc.split():
            tf[word] += 1

    # Step 2: Calculate the inverse category frequency for each word in the corpus
    icf = defaultdict(int)
    num_docs = len(text)
    for doc in text:
        categories_in_doc = set([cat for cat in categories[text.index(doc)].split(",")])
        for word in set(doc.split()):
            icf[word] += 1 / (1 + len(categories_in_doc))

    # Step 3: Calculate the TF-ICF value for each word in the corpus
    tf_icf = {}
    for word in tf:
        tf_icf[word] = tf[word] * math.log(num_docs / icf[word])

    return tf_icf
```

2. The dataset:

- Split the BBC train dataset into training and testing sets.

It is implemented using *train_test_split* from *sklearn* library.

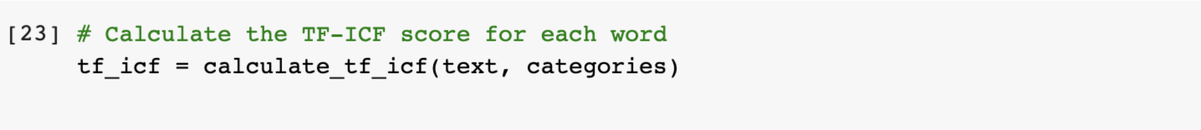
- Use a 70:30 split for the training and testing sets, respectively.



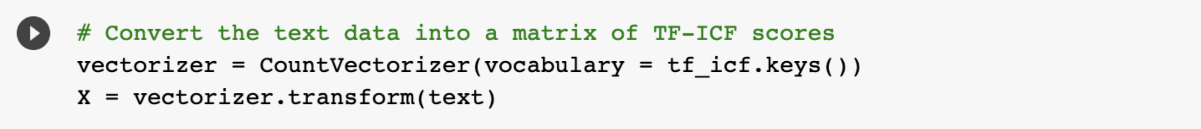
```
[35] # Split the dataset into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(df['processed_text'], df['Category'], test_size=0.3, random_state=42)
```

3. Training the Naive Bayes classifier with TF-ICF:

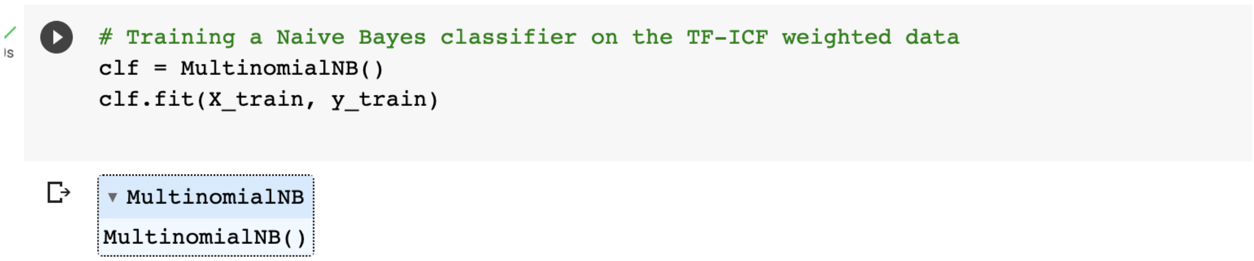
- Implement the Naive Bayes classifier with the TF-ICF weighting scheme.



```
[23] # Calculate the TF-ICF score for each word
     tf_icf = calculate_tf_icf(text, categories)
```



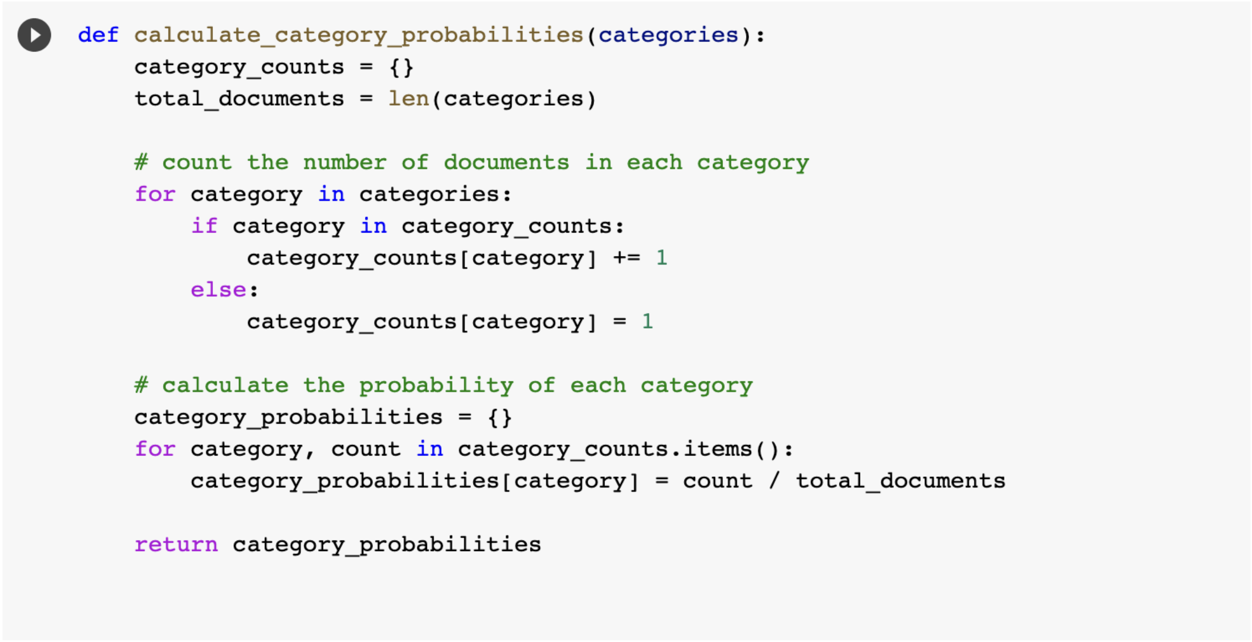
```
# Convert the text data into a matrix of TF-ICF scores
vectorizer = CountVectorizer(vocabulary = tf_icf.keys())
X = vectorizer.transform(text)
```



```
# Training a Naive Bayes classifier on the TF-ICF weighted data
clf = MultinomialNB()
clf.fit(X_train, y_train)
```

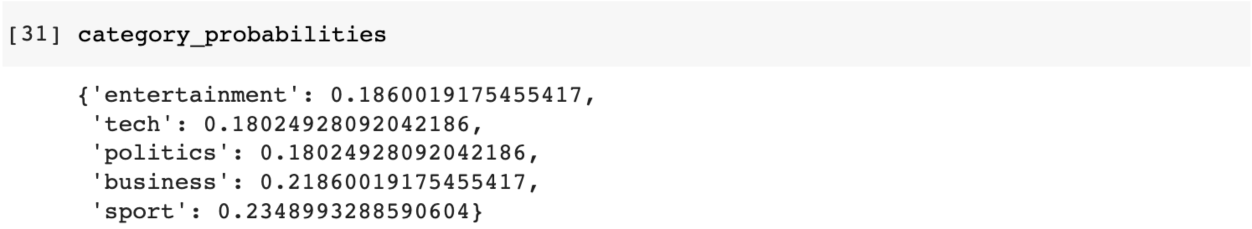
```
▾ MultinomialNB
MultinomialNB()
```

- Calculate the probability of each category based on the frequency of documents in the training set that belong to that category.



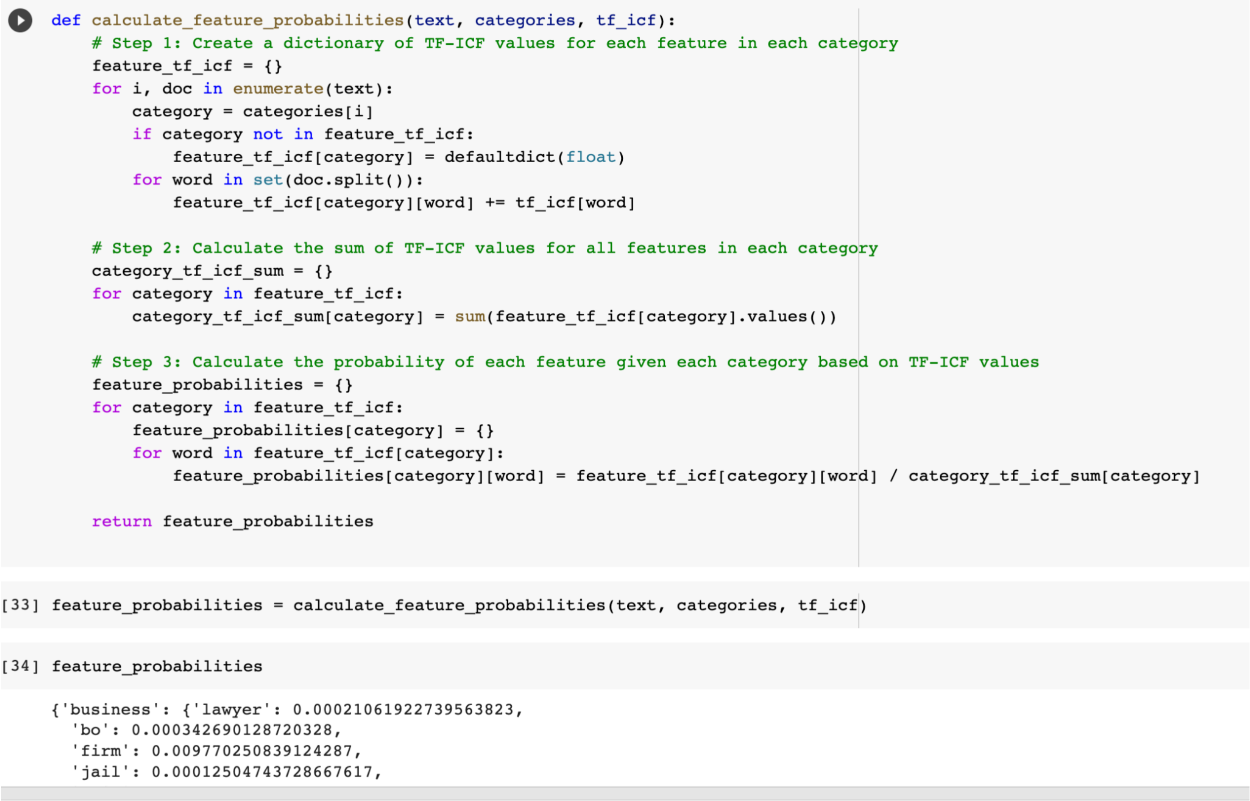
```
def calculate_category_probabilities(categories):
    category_counts = {}
    total_documents = len(categories)

    # count the number of documents in each category
    for category in categories:
        if category in category_counts:
            category_counts[category] += 1
        else:
            category_counts[category] = 1

    # calculate the probability of each category
    category_probabilities = {}
    for category, count in category_counts.items():
        category_probabilities[category] = count / total_documents

    return category_probabilities
```

```
[29] category_probabilities = calculate_category_probabilities(y_train)
```



```
[31] category_probabilities

     {'entertainment': 0.1860019175455417,
      'tech': 0.18024928092042186,
      'politics': 0.18024928092042186,
      'business': 0.21860019175455417,
      'sport': 0.2348993288590604}
```

- Calculate the probability of each feature given each category based on the TF-ICF values of that feature in documents belonging to that category.



```
def calculate_feature_probabilities(text, categories, tf_icf):
    # Step 1: Create a dictionary of TF-ICF values for each feature in each category
    feature_tf_icf = {}
    for i, doc in enumerate(text):
        category = categories[i]
        if category not in feature_tf_icf:
            feature_tf_icf[category] = defaultdict(float)
        for word in set(doc.split()):
            feature_tf_icf[category][word] += tf_icf[word]

    # Step 2: Calculate the sum of TF-ICF values for all features in each category
    category_tf_icf_sum = {}
    for category in feature_tf_icf:
        category_tf_icf_sum[category] = sum(feature_tf_icf[category].values())

    # Step 3: Calculate the probability of each feature given each category based on TF-ICF values
    feature_probabilities = {}
    for category in feature_tf_icf:
        feature_probabilities[category] = {}
        for word in feature_tf_icf[category]:
            feature_probabilities[category][word] = feature_tf_icf[category][word] / category_tf_icf_sum[category]

    return feature_probabilities
```

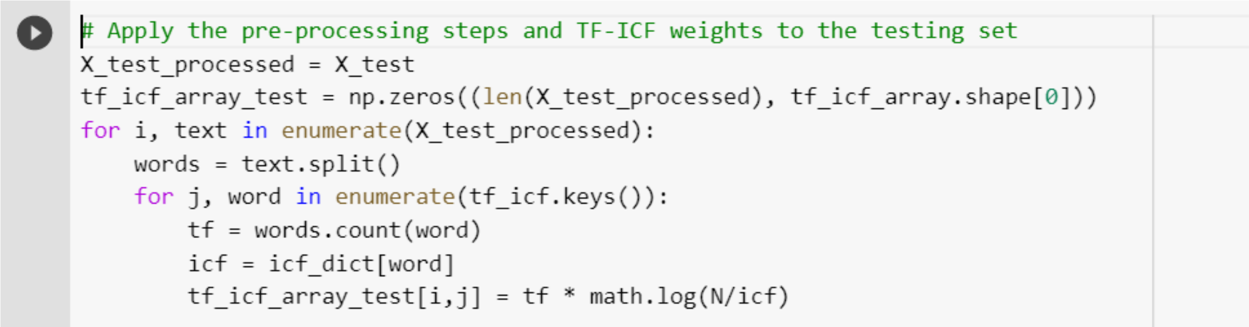
```
[33] feature_probabilities = calculate_feature_probabilities(text, categories, tf_icf)
```

```
[34] feature_probabilities
```

```
{'business': {'lawyer': 0.00021061922739563823,
  'bo': 0.000342690128720328,
  'firm': 0.009770250839124287,
  'jail': 0.00012504743728667617,
```

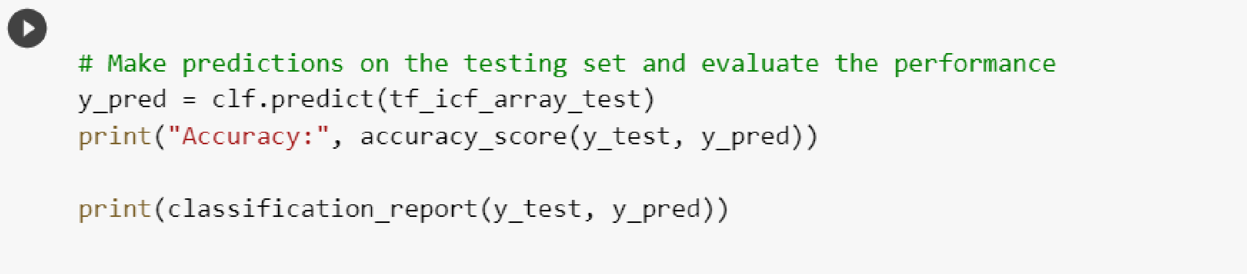
4. Testing the Naive Bayes classifier with TF-ICF:

- Use the testing set to evaluate the performance of the classifier.



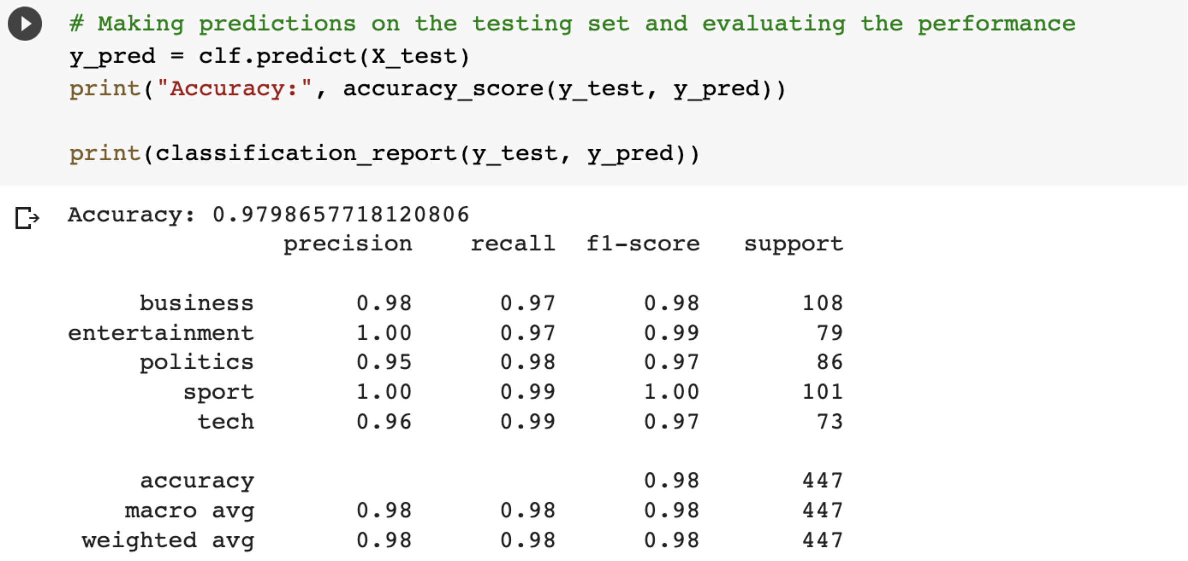
```
# Apply the pre-processing steps and TF-ICF weights to the testing set
X_test_processed = X_test
tf_icf_array_test = np.zeros((len(X_test_processed), tf_icf_array.shape[0]))
for i, text in enumerate(X_test_processed):
    words = text.split()
    for j, word in enumerate(tf_icf.keys()):
        tf = words.count(word)
        icf = icf_dict[word]
        tf_icf_array_test[i,j] = tf * math.log(N/icf)
```

- Classify each document in the testing set and compare the predicted category with the actual category.



```
# Make predictions on the testing set and evaluate the performance
y_pred = clf.predict(tf_icf_array_test)
print("Accuracy:", accuracy_score(y_test, y_pred))

print(classification_report(y_test, y_pred))
```

● Calculate the accuracy, precision, recall, and F1 score of the classifier.



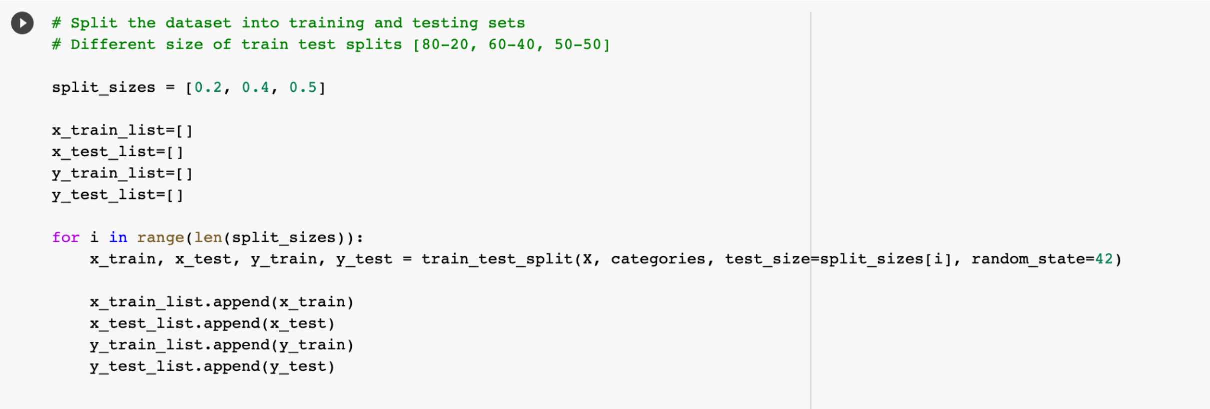
```
# Making predictions on the testing set and evaluating the performance
y_pred = clf.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))

print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.9798657718120806
               precision    recall  f1-score   support

     business       0.98      0.97      0.98       108
entertainment       1.00      0.97      0.99        79
     politics       0.95      0.98      0.97        86
        sport       1.00      0.99      1.00       101
         tech       0.96      0.99      0.97        73

     accuracy                           0.98       447
    macro avg       0.98      0.98      0.98       447
 weighted avg       0.98      0.98      0.98       447
```

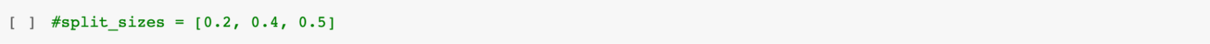
5.

Improving the classifier:

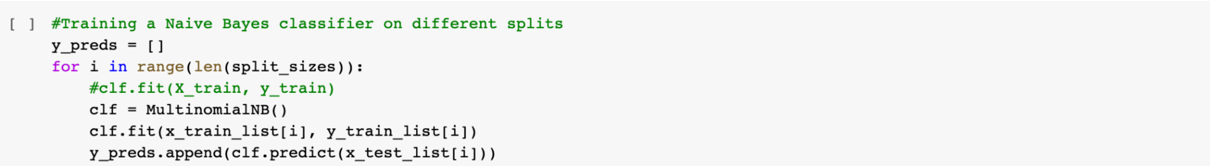
● Experiment with different preprocessing techniques and parameters to improve the performance of the classifier(including different splits like 60-40,80-20, 50-50).



```
# Split the dataset into training and testing sets
# Different size of train test splits [80-20, 60-40, 50-50]

split_sizes = [0.2, 0.4, 0.5]

x_train_list=[]
x_test_list=[]
y_train_list=[]
y_test_list=[]

for i in range(len(split_sizes)):
    x_train, x_test, y_train, y_test = train_test_split(X, categories, test_size=split_sizes[i], random_state=42)

    x_train_list.append(x_train)
    x_test_list.append(x_test)
    y_train_list.append(y_train)
    y_test_list.append(y_test)
```

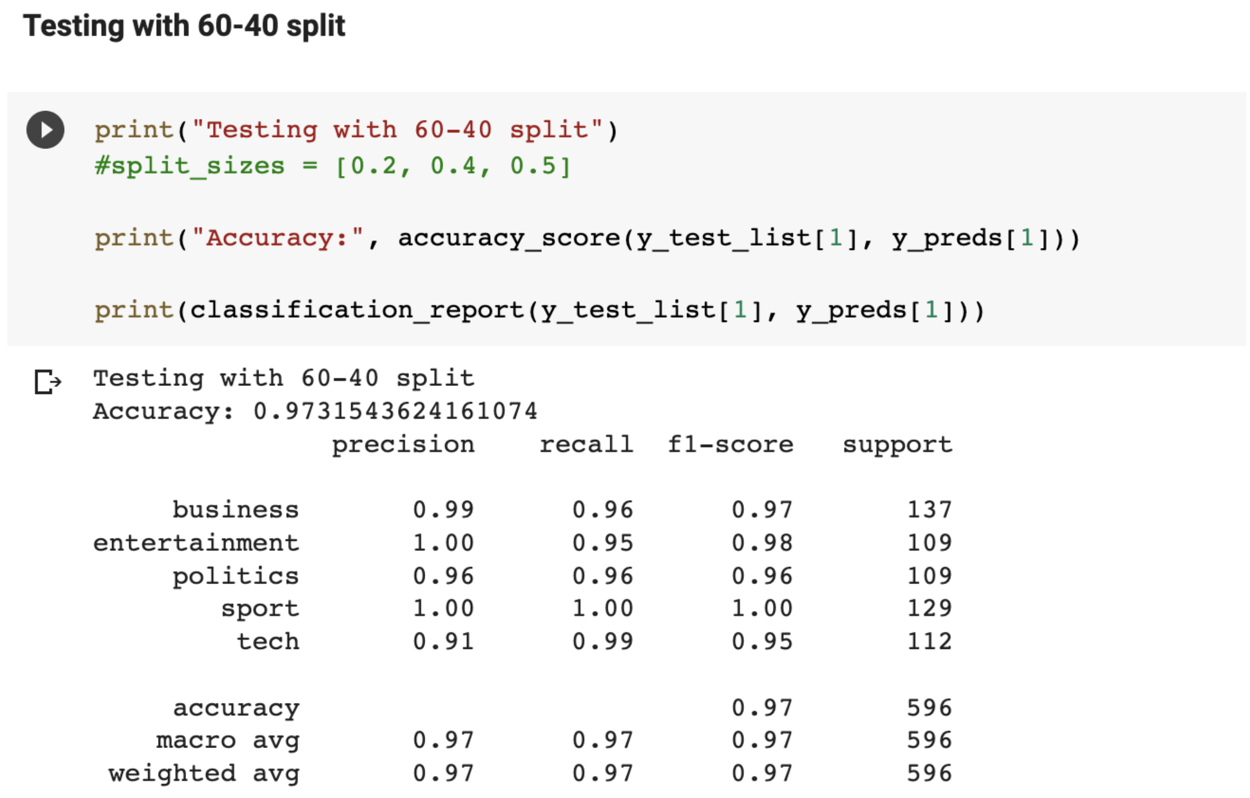


```
#split_sizes = [0.2, 0.4, 0.5]
```



```
#Training a Naive Bayes classifier on different splits
y_preds = []
for i in range(len(split_sizes)):
    #clf.fit(X_train, y_train)
    clf = MultinomialNB()
    clf.fit(x_train_list[i], y_train_list[i])
    y_preds.append(clf.predict(x_test_list[i]))
```

60-40 split

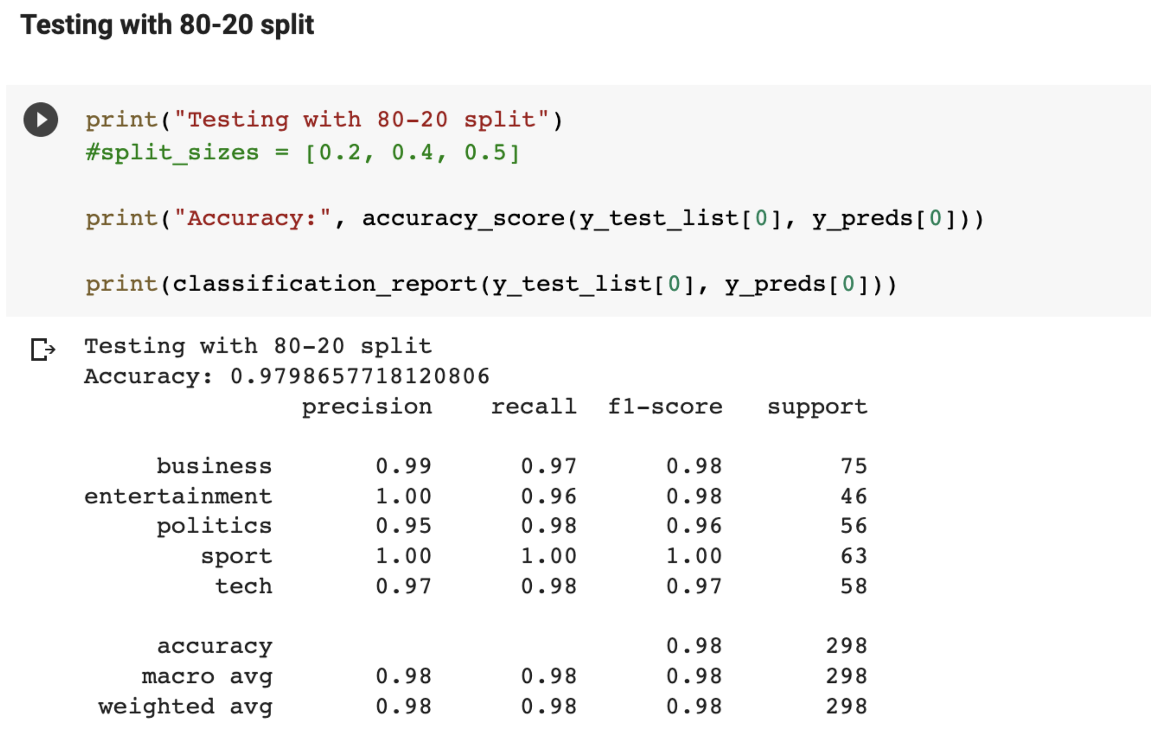


Testing with 60-40 split

```
print("Testing with 60-40 split")
#split_sizes = [0.2, 0.4, 0.5]

print("Accuracy:", accuracy_score(y_test_list[1], y_preds[1]))

print(classification_report(y_test_list[1], y_preds[1]))
```

```
Testing with 60-40 split
Accuracy: 0.9731543624161074
               precision    recall  f1-score   support

     business       0.99      0.96      0.97       137
entertainment       1.00      0.95      0.98       109
     politics       0.96      0.96      0.96       109
        sport       1.00      1.00      1.00       129
         tech       0.91      0.99      0.95       112

     accuracy                           0.97       596
    macro avg       0.97      0.97      0.97       596
 weighted avg       0.97      0.97      0.97       596
```

80-20 split

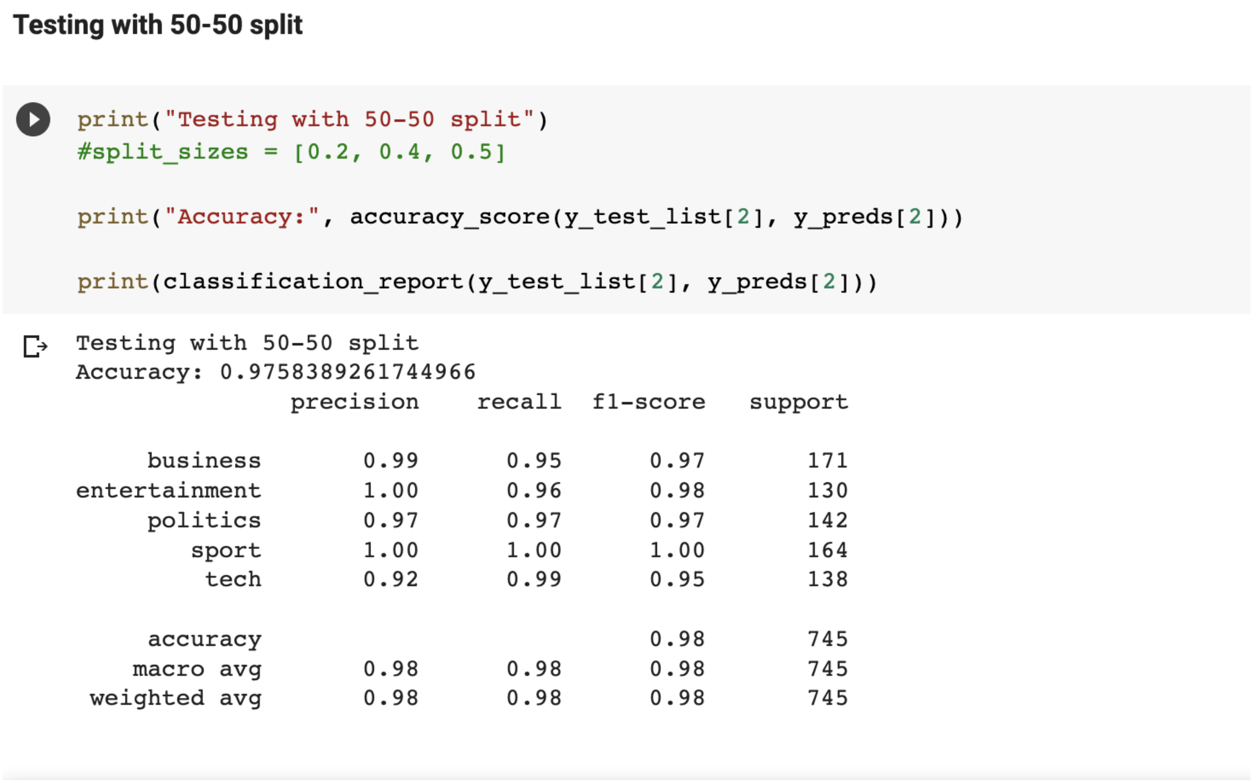


Testing with 80-20 split

```
print("Testing with 80-20 split")
#split_sizes = [0.2, 0.4, 0.5]

print("Accuracy:", accuracy_score(y_test_list[0], y_preds[0]))

print(classification_report(y_test_list[0], y_preds[0]))
```

```
Testing with 80-20 split
Accuracy: 0.9798657718120806
               precision    recall  f1-score   support

     business       0.99      0.97      0.98        75
entertainment       1.00      0.96      0.98        46
     politics       0.95      0.98      0.96        56
        sport       1.00      1.00      1.00        63
         tech       0.97      0.98      0.97        58

     accuracy                           0.98       298
    macro avg       0.98      0.98      0.98       298
 weighted avg       0.98      0.98      0.98       298
```

50-50 split



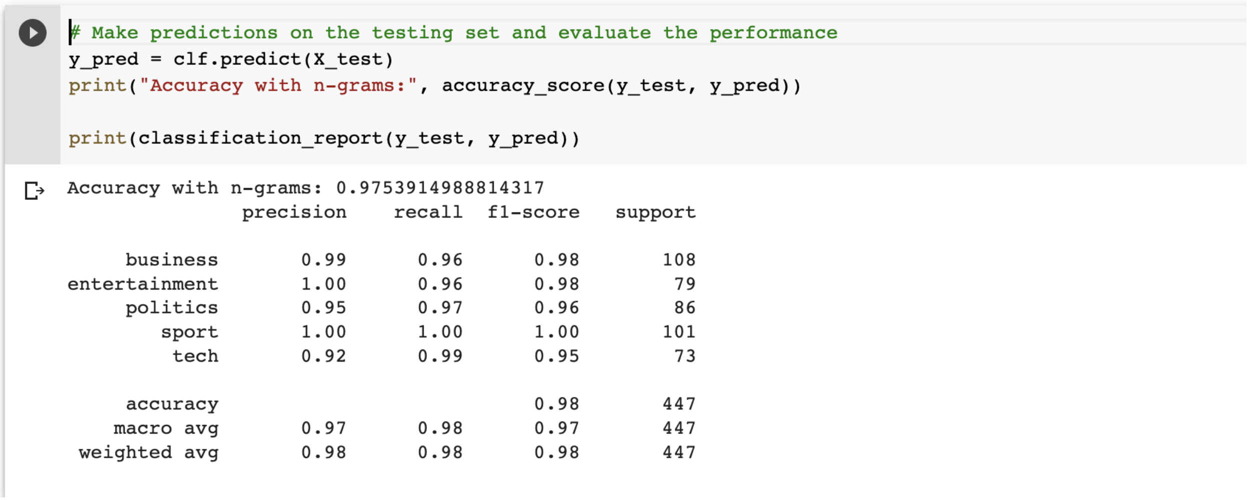
Testing with 50-50 split

```
print("Testing with 50-50 split")
#split_sizes = [0.2, 0.4, 0.5]

print("Accuracy:", accuracy_score(y_test_list[2], y_preds[2]))

print(classification_report(y_test_list[2], y_preds[2]))
```

```
Testing with 50-50 split
Accuracy: 0.9758389261744966
               precision    recall  f1-score   support

     business       0.99      0.95      0.97       171
entertainment       1.00      0.96      0.98       130
     politics       0.97      0.97      0.97       142
        sport       1.00      1.00      1.00       164
         tech       0.92      0.99      0.95       138

     accuracy                           0.98       745
    macro avg       0.98      0.98      0.98       745
 weighted avg       0.98      0.98      0.98       745
```

- Try using different types of features such as n-grams or TF-IDF weights.

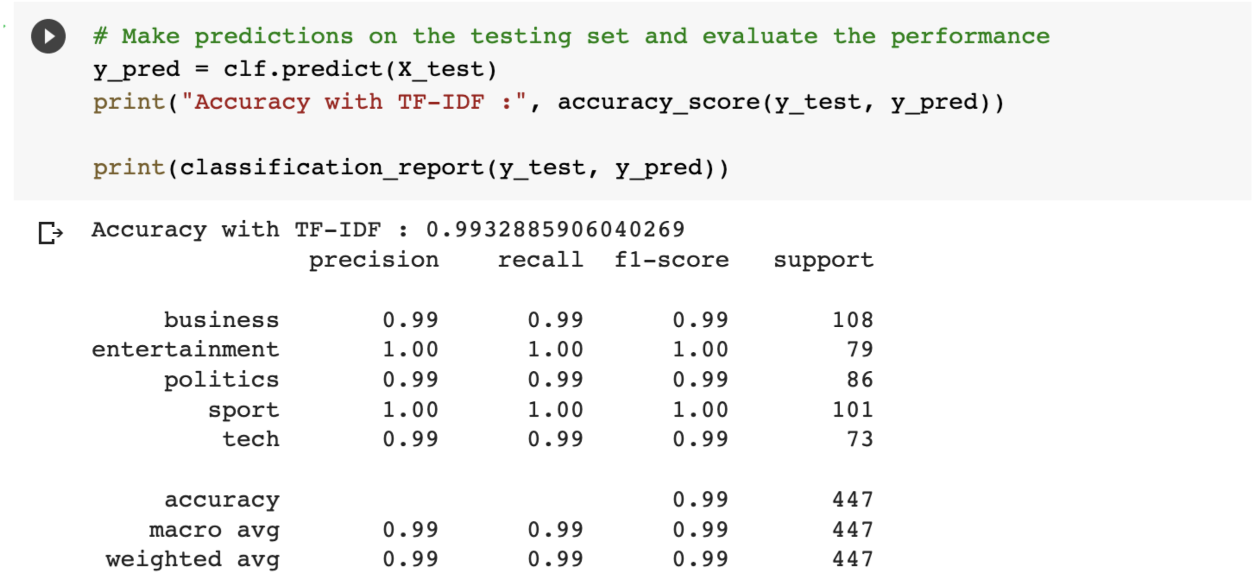
n-grams



```
# Make predictions on the testing set and evaluate the performance
y_pred = clf.predict(X_test)
print("Accuracy with n-grams:", accuracy_score(y_test, y_pred))

print(classification_report(y_test, y_pred))
```

```
Accuracy with n-grams: 0.9753914988814317
               precision    recall  f1-score   support

     business       0.99      0.96      0.98       108
entertainment       1.00      0.96      0.98        79
     politics       0.95      0.97      0.96        86
        sport       1.00      1.00      1.00       101
         tech       0.92      0.99      0.95        73

     accuracy                           0.98       447
    macro avg       0.97      0.98      0.97       447
 weighted avg       0.98      0.98      0.98       447
```

TF-idf weights



```
# Make predictions on the testing set and evaluate the performance
y_pred = clf.predict(X_test)
print("Accuracy with TF-IDF :", accuracy_score(y_test, y_pred))

print(classification_report(y_test, y_pred))
```

```
Accuracy with TF-IDF : 0.9932885906040269
               precision    recall  f1-score   support

     business       0.99      0.99      0.99       108
entertainment       1.00      1.00      1.00        79
     politics       0.99      0.99      0.99        86
        sport       1.00      1.00      1.00       101
         tech       0.99      0.99      0.99        73

     accuracy                           0.99       447
    macro avg       0.99      0.99      0.99       447
 weighted avg       0.99      0.99      0.99       447
```

● Evaluate the performance of the classifier after each experiment and compare it with the previous results.

The performance of the classifier is compared after each experiment and the test results are shown after each sub-heading.

All the classifier give almost the same results, with the tf-idf having the maximum accuracy.

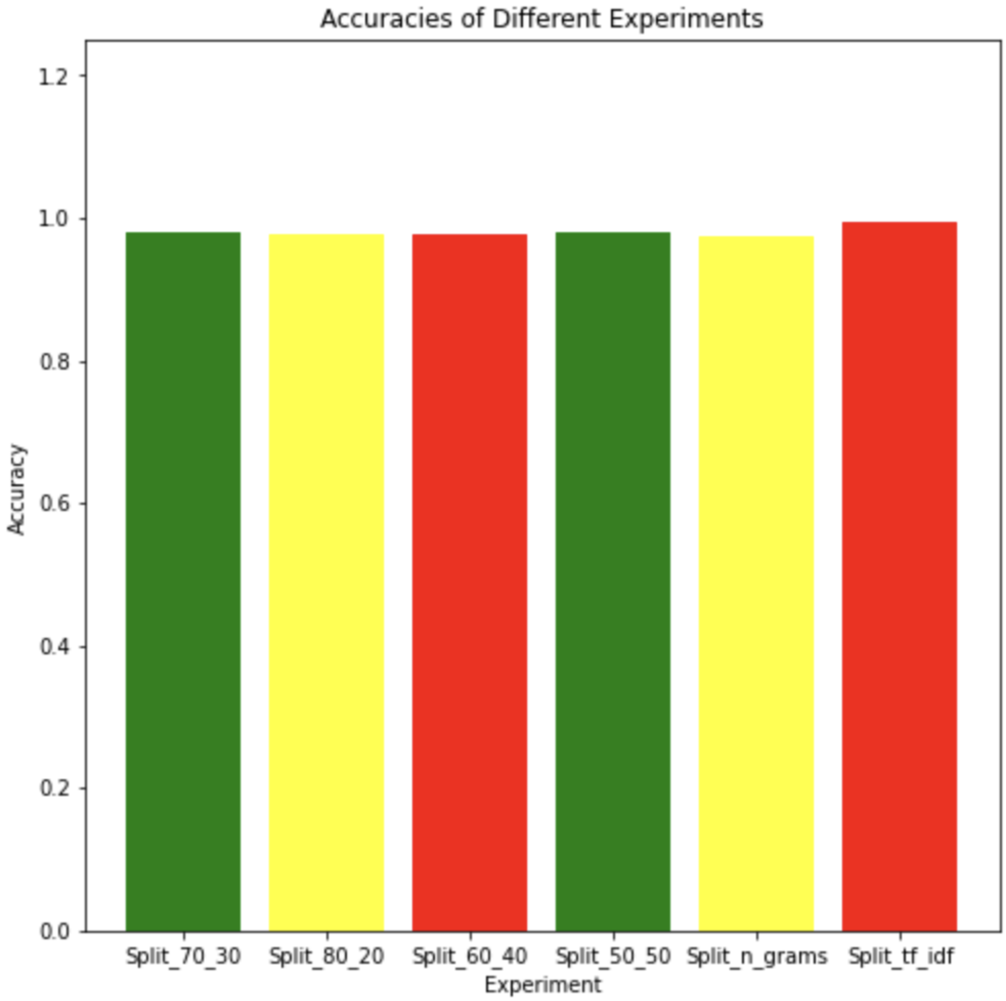
6. Conclusion:

● Write a brief report summarizing your findings.

The report is written under each subheading above corresponding to all the sub questions asked in the assignment.

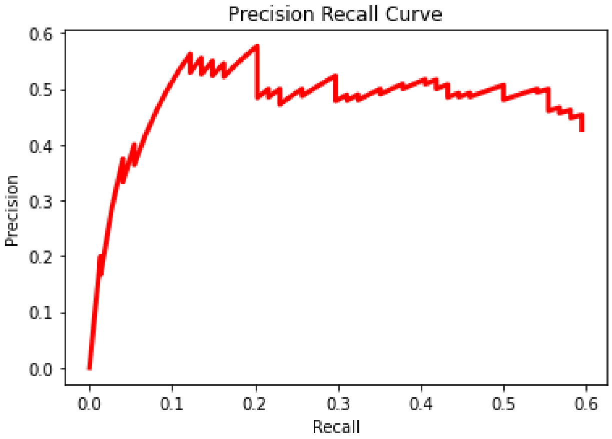
● Discuss the performance of the classifier and the impact of different preprocessing techniques, features, and weighting schemes on the results.

All the classifier give almost the same results, with the tf-idf having the maximum accuracy. There is no such drastic difference with any of the methods since all of them give an accuracy above 95% during testing.


Accuracies of Different Experiments
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Accuracy
Split_70_30
Split_80_20
Split_60_40
Split_50_50
Split_n_grams
Split_tf_idf
Experiment

Question 3

- Our code analyzes a dataset and extracts information related to file types and scores.
- The program creates a dictionary of files that have the second column equal to "qid:4" and extracts information for these files.
- The program creates a new dataframe with only the rows that correspond to the files stored in the dictionary and saves the contents of the new dataframe to a file.
- The program converts the file dictionary into a list of tuples and sorts the list of tuples in descending order based on the values of the second item in each tuple.
- The program calculates the factorials of the file type counts and multiplies them together to get the final answer.
- The program calculates the nDCG (normalized Discounted Cumulative Gain) score for all files in the dataset and for the first 50 files.
- The program computes the nDCG scores for the two sets of files and prints the results.
- The program extracts data from the dataset and stores the score and doc_id as a tuple in the filedatabase dictionary.
- The program sorts the filedatabase dictionary and calculates the precision and recall for each tuple.
- The program plots a precision-recall curve using the calculated values.


Precision Recall Curve
Precision
Recall