A project on

Classification

Aurovindhya Srinivasan

The objective of this project is to produce classification predictions and compare them. Ideally, it is to train a machine that can tell which author the book belongs to when asked. The pros and cons of the chosen algorithms were comprehended by visualizing and analyzing the results. The inferred insights have been shared.

This is the process followed:

1. Data Preparation and Pre-processing
2. Feature Engineering
3. Classification
4. Evaluation
5. Error analysis

I) Data Preparation and Pre-processing

Five different books have been selected based on the authors with almost similar genres. The preparation and cleaning of the data took the following steps.

Retrieval and Selection of Samples of Gutenberg Digital Books:

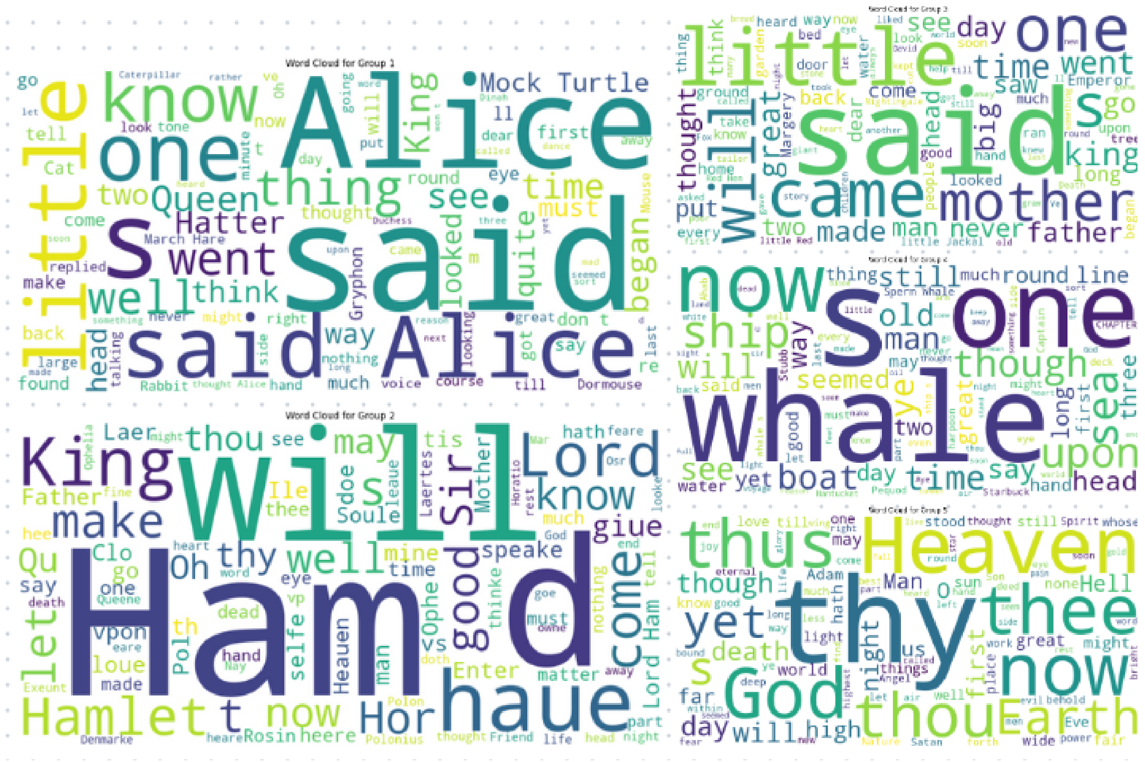
NLTK includes a small selection of texts from the Project Gutenberg electronic text archive, which contains 25,000 free digital books. To begin with, NLTK was imported and the relevant resources from NLTK were downloaded.

```
import re
import pandas as pd
import random
import nltk
import sklearn
nltk.download('gutenberg')
from nltk.corpus import gutenberg
```

The following books were chosen, pertaining to different authors and under the umbrella of "Fiction" genre:

```
# Print the list of available books
print("List of available books: ", gutenberg.fileids())
books = [
        {
            "file": 'carroll-alice.txt',
            "author": "Carroll",
            "genre": "Fiction"
        },
        {
            "file": 'shakespeare-hamlet.txt',
            "author": "Shakespeare",
            "genre": "Fiction"
        },
        {
            "file": 'bryant-stories.txt',
            "author": "Bryant",
            "genre": "Fiction"
        },
        {
            "file": 'melville-moby_dick.txt',
            "author": "Melville",
            "genre": "Fiction"
        },
        {
            "file": 'milton-paradise.txt',
            "author": "Milton",
            "genre": "Fiction"
        }
    ]
```

Here are word clouds for each book depicting the frequently used words:



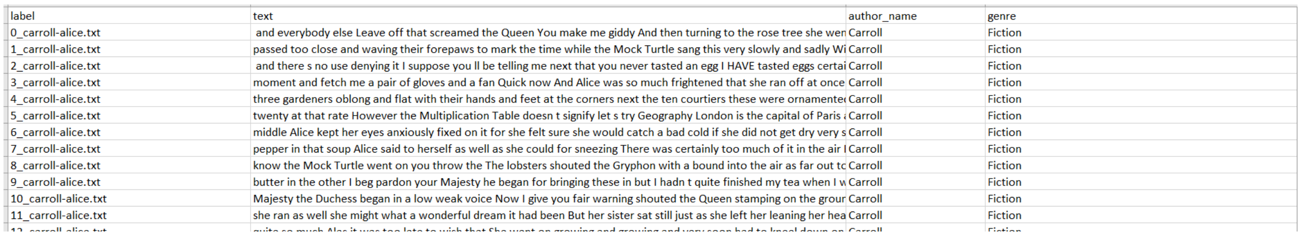
The chosen books were then partitioned into 200 pieces with each partition containing 100 words. Additionally, the data was cleaned by removing garbage characters.

```
def get_book_partitions(book_name, partition_size=100, num_partitions=200):

    # Download the book
    words = gutenberg.words(book_name)

    # Get the total number of words in the book
    total_words = len(words)

    # Randomly select starting indices for the partitions
    partition_indices = random.sample(
        range(0, total_words - partition_size), num_partitions)

    # Create a dictionary to store the partitions and their labels
    partitions = {}

    for i, index in enumerate(partition_indices):
        # Get the partition text
        partition = " ".join(words[index:index+partition_size])

        # Clean the text
        partition = re.sub(r"[^a-zA-Z0-9]+", ' ', partition)

        # Create a label for the partition
        label = "{}_{}".format(i, book_name)  # include book name in the label

        # Add the partition and its label to the dictionary
        partitions[label] = partition

    # Convert the partitions dictionary to a DataFrame
    partitions_df = pd.DataFrame.from_dict(
        partitions, orient='index', columns=['text'])
    partitions_df.reset_index(inplace=True)
    partitions_df.rename(columns={'index': 'label'}, inplace=True)

    return partitions_df
```

The partitioned data was then converted into a csv file for easier consumption.

```
for book in books:
    df = create_dataframe(book["file"], book["author"], book["genre"])
    book["df"] = df

result = pd.concat([x["df"] for x in books])

# Make a csv file of the partitions to make the data easier to consume

result.to_csv("book_partitions.csv", index=False)
```

The resultant csv file, which can then be consumed as a dataframe, looks like this:



label	text	author_name	genre
0_carroll-alice.txt	and everybody else Leave off that screamed the Queen You make me giddy And then turning to the rose tree she wen	Carroll	Fiction
1_carroll-alice.txt	passed too close and waving their forepaws to mark the time while the Mock Turtle sang this very slowly and sadly Wi	Carroll	Fiction
2_carroll-alice.txt	and there s no use denying it I suppose you ll be telling me next that you never tasted an egg I HAVE tasted eggs certai	Carroll	Fiction
3_carroll-alice.txt	moment and fetch me a pair of gloves and a fan Quick now And Alice was so much frightened that she ran off at once	Carroll	Fiction
4_carroll-alice.txt	three gardeners oblong and flat with their hands and feet at the corners next the ten courtiers these were ornamente	Carroll	Fiction
5_carroll-alice.txt	twenty at that rate However the Multiplication Table doesn t signify let s try Geography London is the capital of Paris a	Carroll	Fiction
6_carroll-alice.txt	middle Alice kept her eyes anxiously fixed on it for she felt sure she would catch a bad cold if she did not get dry very s	Carroll	Fiction
7_carroll-alice.txt	pepper in that soup Alice said to herself as well as she could for sneezing There was certainly too much of it in the air I	Carroll	Fiction
8_carroll-alice.txt	know the Mock Turtle went on you throw the The lobsters shouted the Gryphon with a bound into the air as far out tc	Carroll	Fiction
9_carroll-alice.txt	butter in the other I beg pardon your Majesty he began for bringing these in but I hadn t quite finished my tea when I w	Carroll	Fiction
10_carroll-alice.txt	Majesty the Duchess began in a low weak voice Now I give you fair warning shouted the Queen stamping on the grour	Carroll	Fiction
11_carroll-alice.txt	she ran as well she might what a wonderful dream it had been But her sister sat still just as she left her leaning her hea	Carroll	Fiction

Feature Engineering

The text and author names were extracted as input and output.

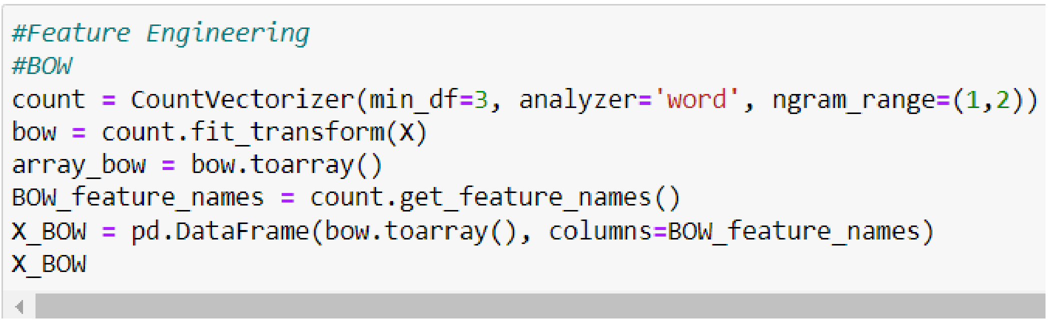
```
#Data Engineering
X = result['text'].values
y = result['author_name'].values
```

The processed data was then transformed for feature engineering using the following:

1. Bag-of-Words (BoW)
2. Term Frequency–Inverse Document Frequency (TF-IDF)
3. N-gram

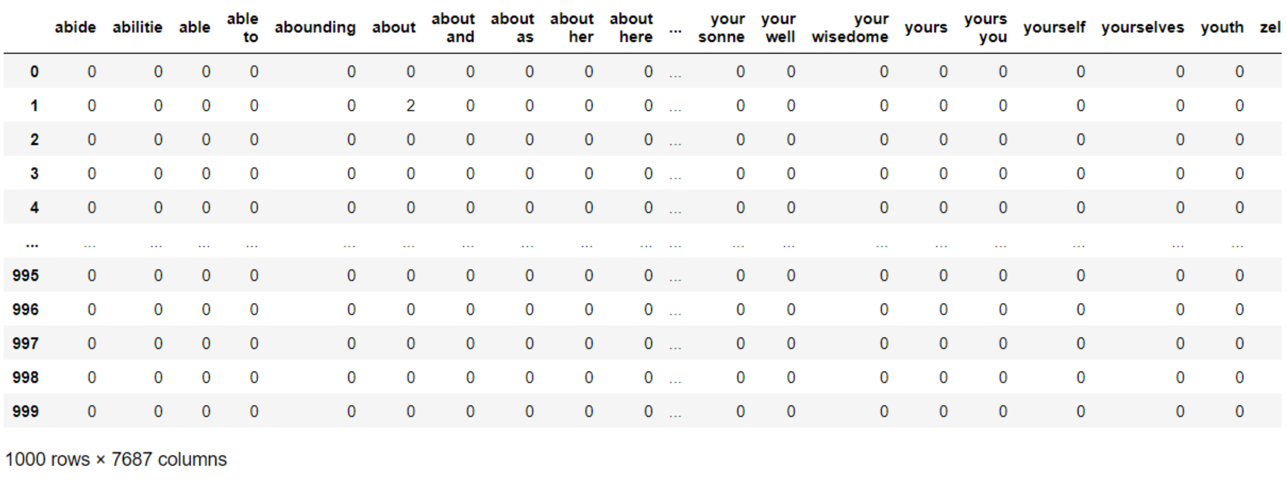
BoW:

The Bag-of-Words model keeps track of the words found in the corpus and uses them as features for the documents. The frequency of each word in a given document is calculated and the word features are then extracted from the textual corpus. This was accomplished using the "CountVectorizer" function, which was set to consider both unigrams and bigrams and had a minimum document frequency of 3, meaning that words appearing in less than 3 documents were disregarded.



```
#Feature Engineering
#BOW
count = CountVectorizer(min_df=3, analyzer='word', ngram_range=(1,2))
bow = count.fit_transform(X)
array_bow = bow.toarray()
BOW_feature_names = count.get_feature_names()
X_BOW = pd.DataFrame(bow.toarray(), columns=BOW_feature_names)
X_BOW
```

The result is that all the words are transformed into a Bag-of-Words representation as displayed below.



	abide	abilitie	able	able to	abounding	about	about and	about as	about her	about here	...	your sonne	your well	your wisedome	yours	yours you	yourself	yourselves	youth	zel
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	2	0	0	0	0	...	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
996	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
997	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
998	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	
999	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	

1000 rows × 7687 columns

TF-IDF:

The most favored method for converting text data into numerical features is the Term Frequency-Inverse Document Frequency (TF-IDF). This method emphasizes words that are important and specific to a given document, but not to all documents, by assigning numerical values to them.

Term Frequency (TF) is calculated as:

TF = (Frequency of the word in the sentence) / (Total number of words in the sentence)

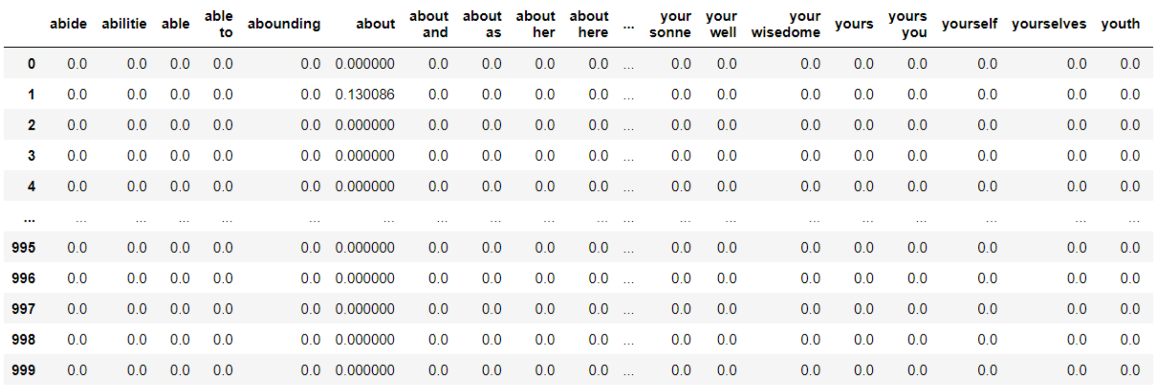
Inverse Document Frequency (IDF) is calculated as:

IDF = (Total number of sentences (documents))/(Number of sentences (documents) containing the word)

Here, it is computed using the TfidVectorizer, considering both unigrams and bigrams with a minimum frequency of 3:

```
#TD_IDF
tf = TfidfVectorizer(analyzer='word',min_df= 3, ngram_range=(1, 2))
Tfid = tf.fit_transform(X)
array_tfid = Tfid.toarray()
tfid_feature_names = tf.get_feature_names()
X_Tfid = pd.DataFrame(Tfid.toarray(), columns=tfid_feature_names)
X_Tfid
```

The result is as follows:



```
     abide  abilitie  able  able  abounding     about  about  about  about  about  ...  your   your       your  yours  yours  yourself  yourselves  youth
                              to                        and     as    her   here       sonne  well  wisedome          you
  0    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
  1    0.0       0.0   0.0   0.0        0.0  0.130086    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
  2    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
  3    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
  4    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
...    ...       ...   ...   ...        ...       ...    ...    ...    ...    ...  ...   ...   ...        ...    ...    ...       ...         ...    ...
995    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
996    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
997    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
998    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
999    0.0       0.0   0.0   0.0        0.0  0.000000    0.0    0.0    0.0    0.0  ...   0.0   0.0        0.0    0.0    0.0       0.0         0.0    0.0
```

N-gram:

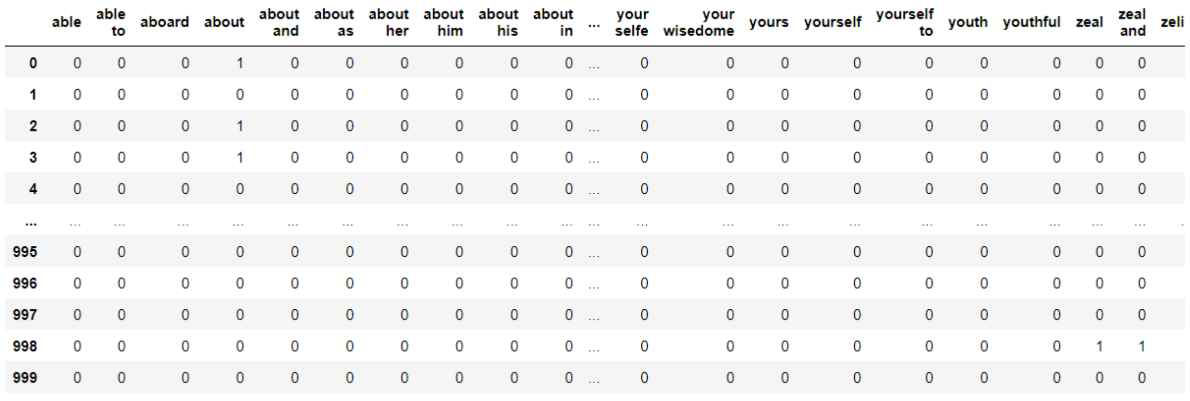
In the fields of computational linguistics and probability, an N-gram is a contiguous sequence of n items from a given sample of text or speech. An N-gram model is built by counting how often word sequences occur in corpus text and then estimating the probabilities.

We've used it because N-gram models are useful in text analytics applications, such as this project, where sequences of words are relevant.

As explained earlier, the "CountVectorizer" is used to manipulate the frequency and word features extracted from the corpus. Then both unigrams and bigrams are considered to build an N-gram model as displayed below.

```
#n-gram
# building a bigram model
# bigrams of words in addition to unigrams (individual words)
bigram_vectorizer = CountVectorizer(analyzer='word',min_df= 3, ngram_range=(1,2))

bigram = bigram_vectorizer.fit_transform(X)
array_bigram = bigram.toarray()
bigram_features = bigram_vectorizer.get_feature_names_out()
X_bigram = pd.DataFrame(array_bigram, columns= bigram_features)
X_bigram
```

The result is a weighted bigram dataframe as follows:



	able	able to	aboard	about	about and	about as	about her	about him	about his	about in	...	your selfe	your wisedome	yours	yourself	yourself to	youth	youthful	zeal	zeal and	zeli
0	0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
2	0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
3	0	0	0	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
996	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
997	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	
998	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	1	1	
999	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	

1000 rows × 7806 columns

Classification

The following classification techniques were used on each of the feature engineering models (BOW, TF-IDF, N-gram):

1. Decision Tree
2. K-Nearest Neighbor (KNN)
3. Support Vector Machine (SVM)

In all of the following classification models, the following steps are followed:

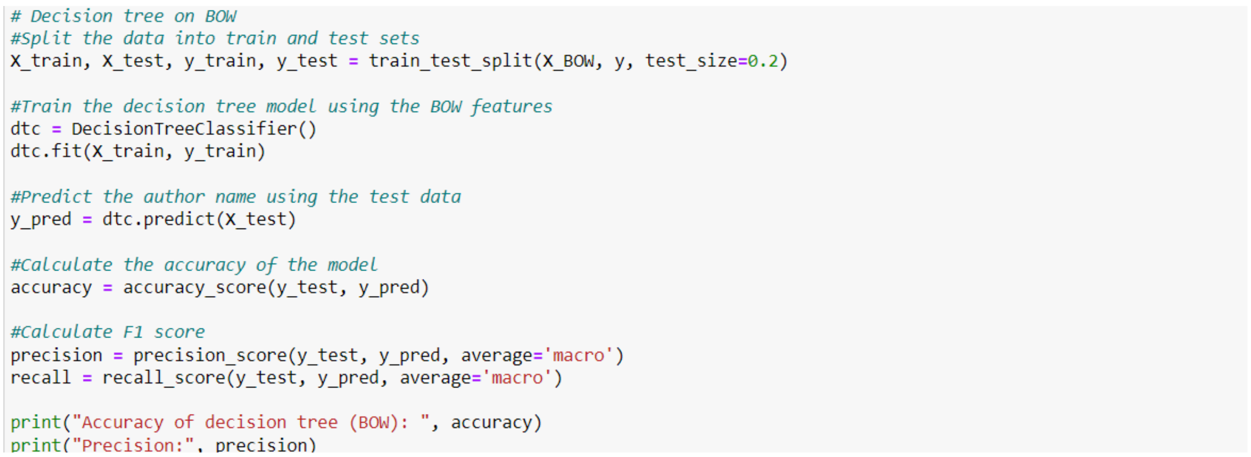
1. The dataset from the respective feature engineering model is divided into training and testing datasets.
2. The model is then trained using appropriate parameters.
3. For validation purposes, the accuracy, precision, recall and F1 score are calculated and analyzed.
4. Finally, the results are visualized.

Decision Tree:

The confusion matrix in the results depicts the misclassified authors as follows.

a) BOW

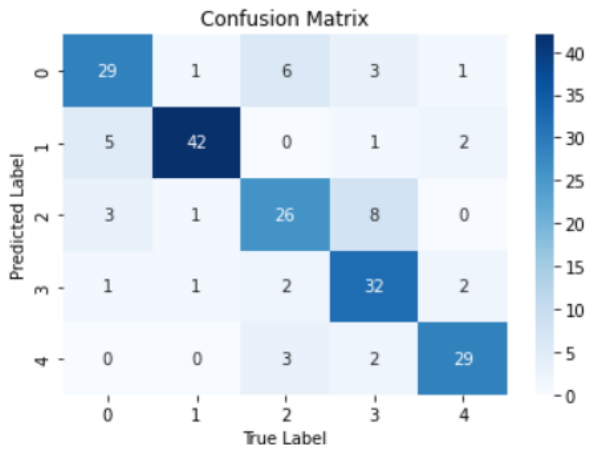
Code:

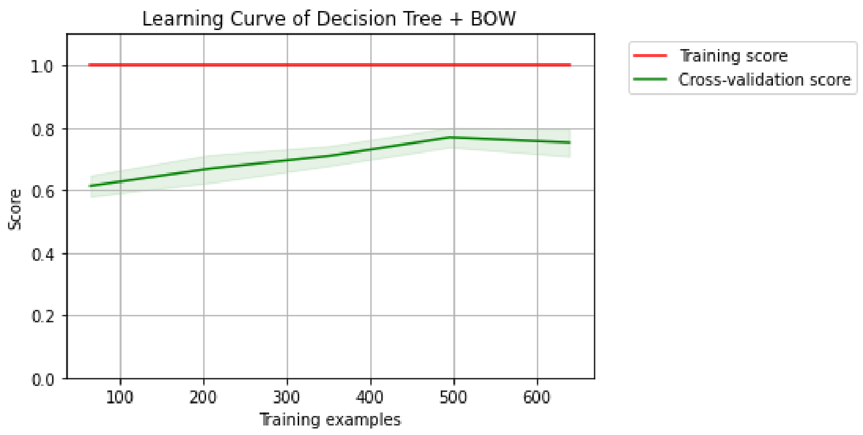


```
# Decision tree on BOW
#Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_BOW, y, test_size=0.2)

#Train the decision tree model using the BOW features
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)

#Predict the author name using the test data
y_pred = dtc.predict(X_test)

#Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

#Calculate F1 score
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of decision tree (BOW): ", accuracy)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)

conf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('True Label')
plt.ylabel('Predicted Label')
plt.title('Confusion Matrix')
plt.show()
```

Results:

```
Accuracy of decision tree (BOW):  0.79
Precision: 0.789557456231302
Recall: 0.7888513931888544
F1 Score: 0.7912658183921342
```





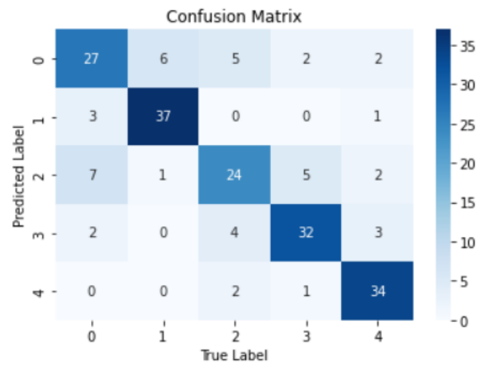
b) TF-IDF

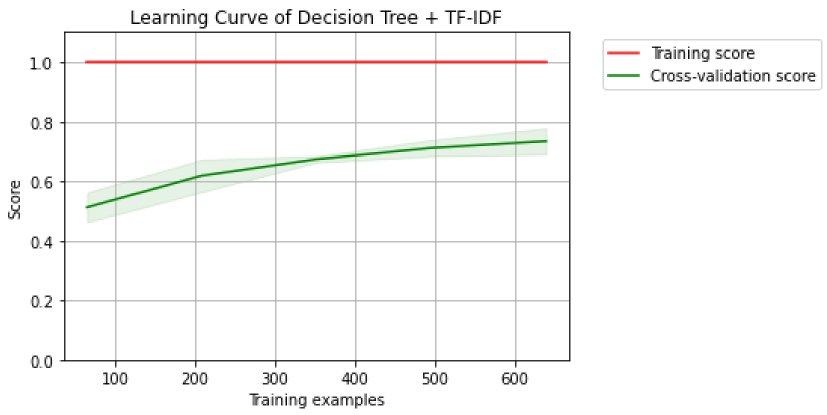
Code:

```
# Decision tree on TF-IDF
#Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_Tfid, y, test_size=0.2)

#Train the decision tree model using the BOW features
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)

#Predict the author name using the test data
y_pred = dtc.predict(X_test)

#Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

#Calculate F1 score
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of decision tree (TF-IDF): ", accuracy)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)

conf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('True Label')
plt.ylabel('Predicted Label')
plt.title('Confusion Matrix')
plt.show()
```

Results:

```
Accuracy of decision tree (TF-IDF):  0.77
Precision: 0.7656909756909757
Recall: 0.7720175012857939
F1 Score: 0.7661728896928699
```





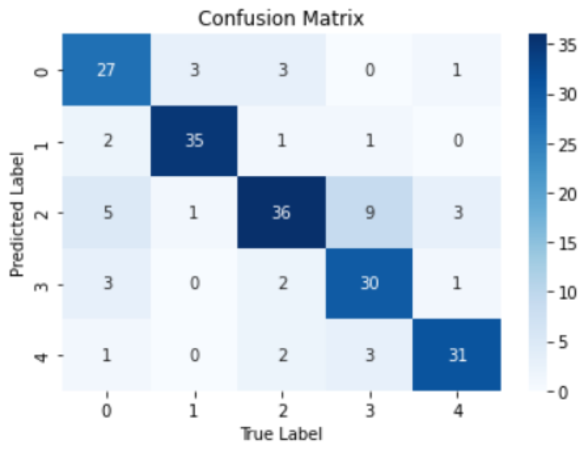
c) N-gram

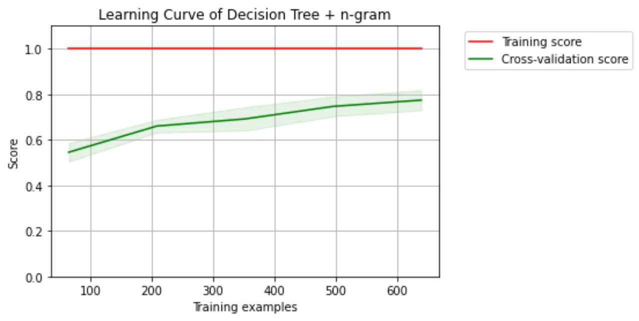
Code:

```
# Decision tree on N-gram
#Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_bigram, y, test_size=0.2)

#Train the decision tree model using the BOW features
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)

#Predict the author name using the test data
y_pred = dtc.predict(X_test)

#Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

#Calculate F1 score
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of decision tree (N-gram): ", accuracy)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)

conf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('True Label')
plt.ylabel('Predicted Label')
plt.title('Confusion Matrix')
plt.show()
```

Results:

```
Accuracy of decision tree (N-gram):  0.795
Precision: 0.7969859122245904
Recall: 0.8058782764665118
F1 Score: 0.7946994953695019
```





KNN

a) BOW

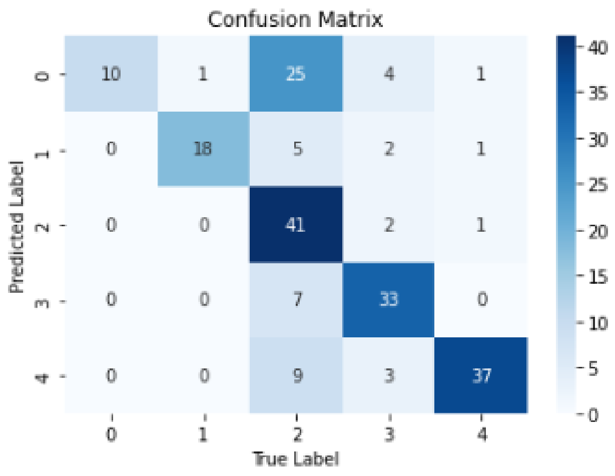
Code:

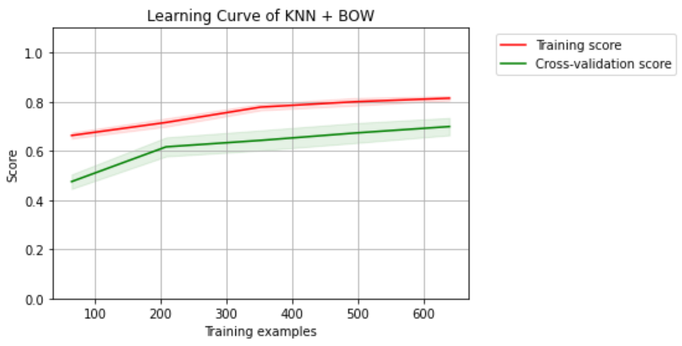
```
#KNN on BOW
#Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_BOW, y, test_size=0.2, random_state=0)

#Train the KNN classifier using BOW features
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

```
#Make predictions on the test set
y_pred = knn.predict(X_test)

#Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of KNN (BOW):", accuracy)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)
```

Results:

```
Accuracy of KNN (BOW): 0.695
Precision: 0.8187265577737447
Recall: 0.6896260707933181
F1 Score: 0.6829528028382638
```





b) TF-IDF

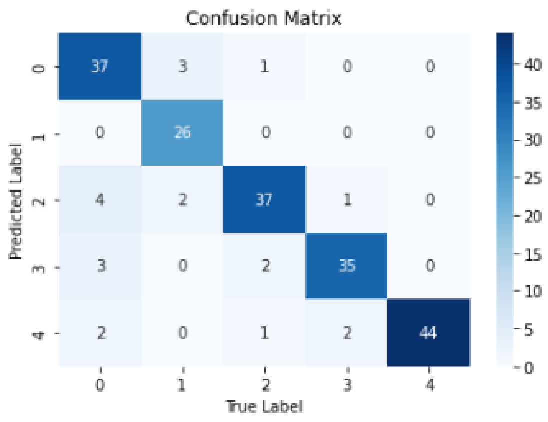
Code:

```
#KNN on TF-IDF
#Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_Tfid, y, test_size=0.2, random_state=0)

#Train the KNN classifier using TF-IDF features
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
```

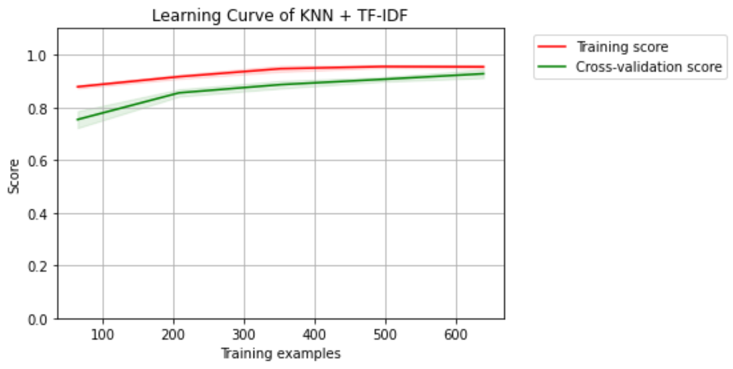
```
#Make predictions on the test set
y_pred = knn.predict(X_test)

#Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of KNN (TF-IDF):", accuracy)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)
```

Results:

```
Accuracy of KNN (TF-IDF): 0.895
Precision: 0.8933098318951005
Recall: 0.9032614597945609
F1 Score: 0.8958088555611571
```



	0	1	2	3	4
0	37	3	1	0	0
1	0	26	0	0	0
2	4	2	37	1	0
3	3	0	2	35	0
4	2	0	1	2	44



c) N-gram

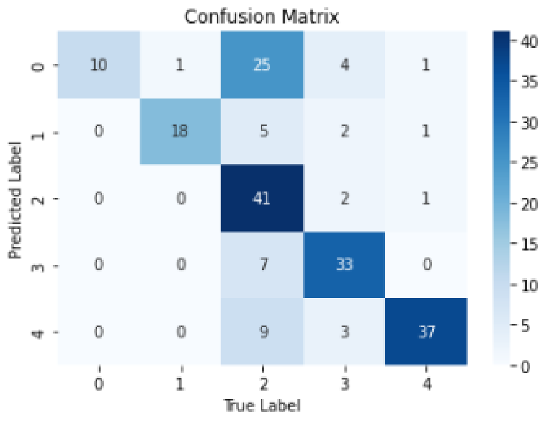
Code:

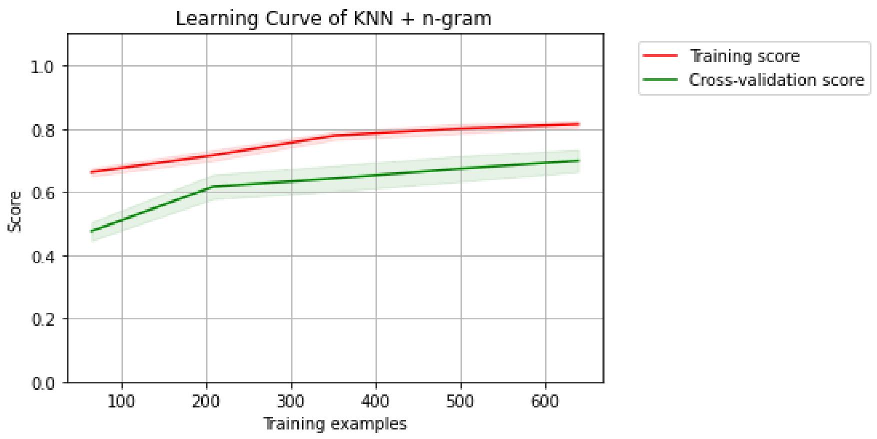
```
#KNN on N-gram
#Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_bigram, y, test_size=0.2, random_state=0)

#Train the KNN classifier using N-gram features
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

#Make predictions on the test set
y_pred = knn.predict(X_test)
```

```python
#Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of KNN (N-gram):", accuracy)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)
```

<u>Results:</u>

```
Accuracy of KNN (N-gram): 0.695
Precision: 0.8187265577737447
Recall: 0.6896260707933181
F1 Score: 0.6829528028382638
```





SVM

a) BOW

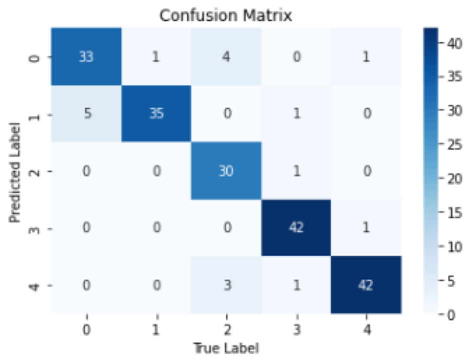
<u>Code:</u>

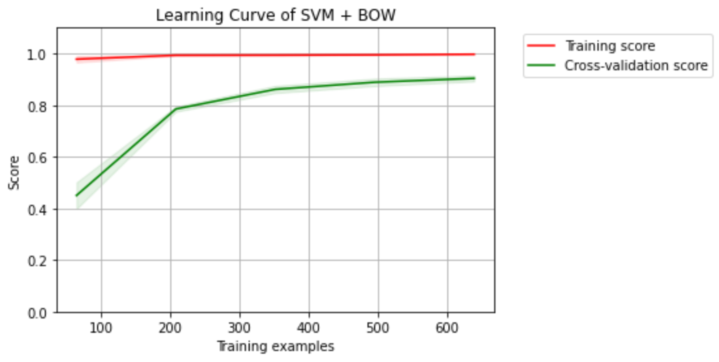
```python
#SVM on BOW

#Split the data into training and testing sets
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X_BOW, y, test_size=0.2)

#Initialize the SVM classifier with a radial basis function kernel
clf = SVC(kernel='rbf', random_state=0)
```

```
#Train the classifier on the training data
clf.fit(X_train, y_train)

#Predict the labels for the test data
y_pred = clf.predict(X_test)

#Evaluate the performance of the classifier
acc = sklearn.metrics.accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of SVM classifier (BOW):", acc)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)
```

Results:

```
Accuracy of SVM classifier (BOW): 0.91
Precision: 0.9078665747086798
Recall: 0.9114683965060928
F1 Score: 0.9101651387827857
```





b) TF-IDF

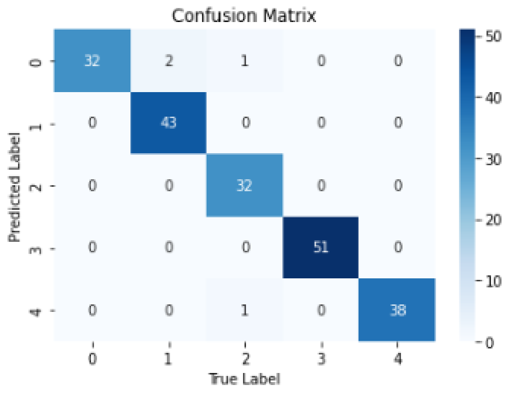
Code:

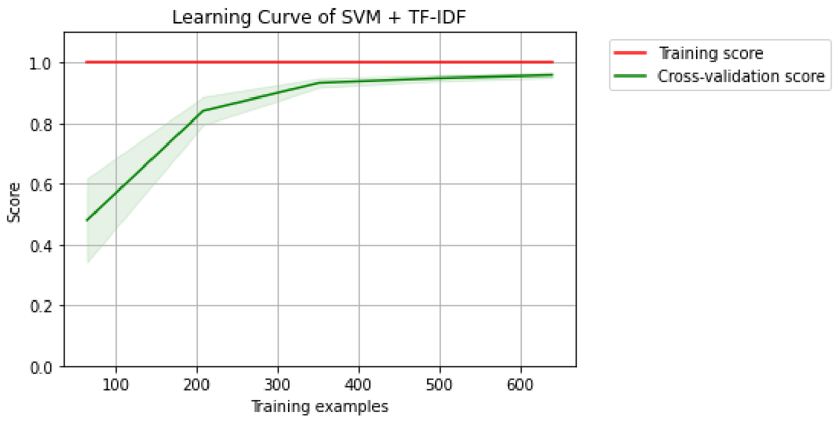
```
#SVM on TF-IDF

#Split the data into training and testing sets
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X_Tfid, y, test_size=0.2)

#Initialize the SVM classifier with a radial basis function kernel
clf = SVC(kernel='rbf', random_state=0)

#Train the classifier on the training data
clf.fit(X_train, y_train)

#Predict the labels for the test data
y_pred = clf.predict(X_test)
```

```
#Evaluate the performance of the classifier
acc = sklearn.metrics.accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("Accuracy of SVM classifier (TF-IDF):", acc)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)
```

<u>Result:</u>

```
Accuracy of SVM classifier (TF-IDF): 0.98
Precision: 0.9793464052287583
Recall: 0.9777289377289377
F1 Score: 0.9798968630871616
```





c) N-gram

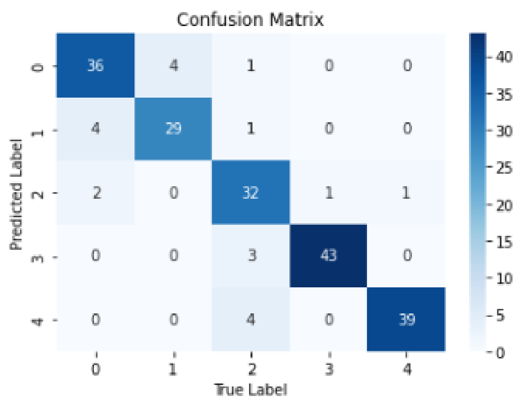
<u>Code:</u>

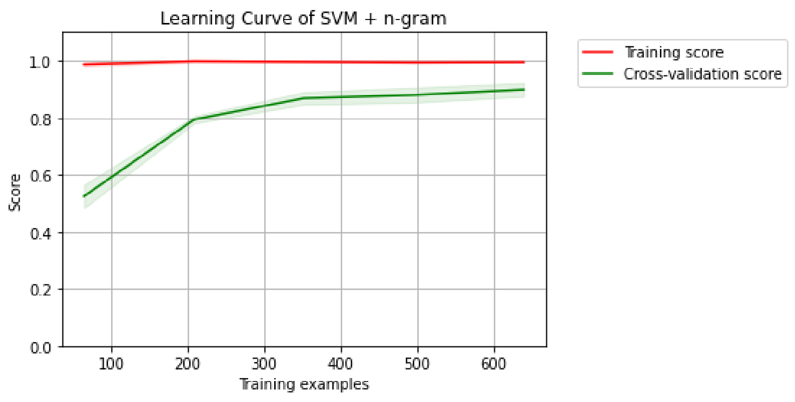
```
#SVM on N-gram

#Split the data into training and testing sets
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X_bigram, y, test_size=0.2)

#Initialize the SVM classifier with a radial basis function kernel
clf = SVC(kernel='rbf', random_state=0)

#Train the classifier on the training data
clf.fit(X_train, y_train)

#Predict the labels for the test data
y_pred = clf.predict(X_test)

#Evaluate the performance of the classifier
acc = sklearn.metrics.accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')
```

```
print("Accuracy of SVM classifier (N-gram):", acc)
print("Precision:", precision)
print("Recall:", recall)

f1 = f1_score(y_test, y_pred, average='weighted')
print("F1 Score:", f1)
```

<u>Result:</u>

```
Accuracy of SVM classifier (N-gram): 0.895
Precision: 0.8937382536163023
Recall: 0.8923276397457961
F1 Score: 0.8964318645649343
```





IV) Evaluation

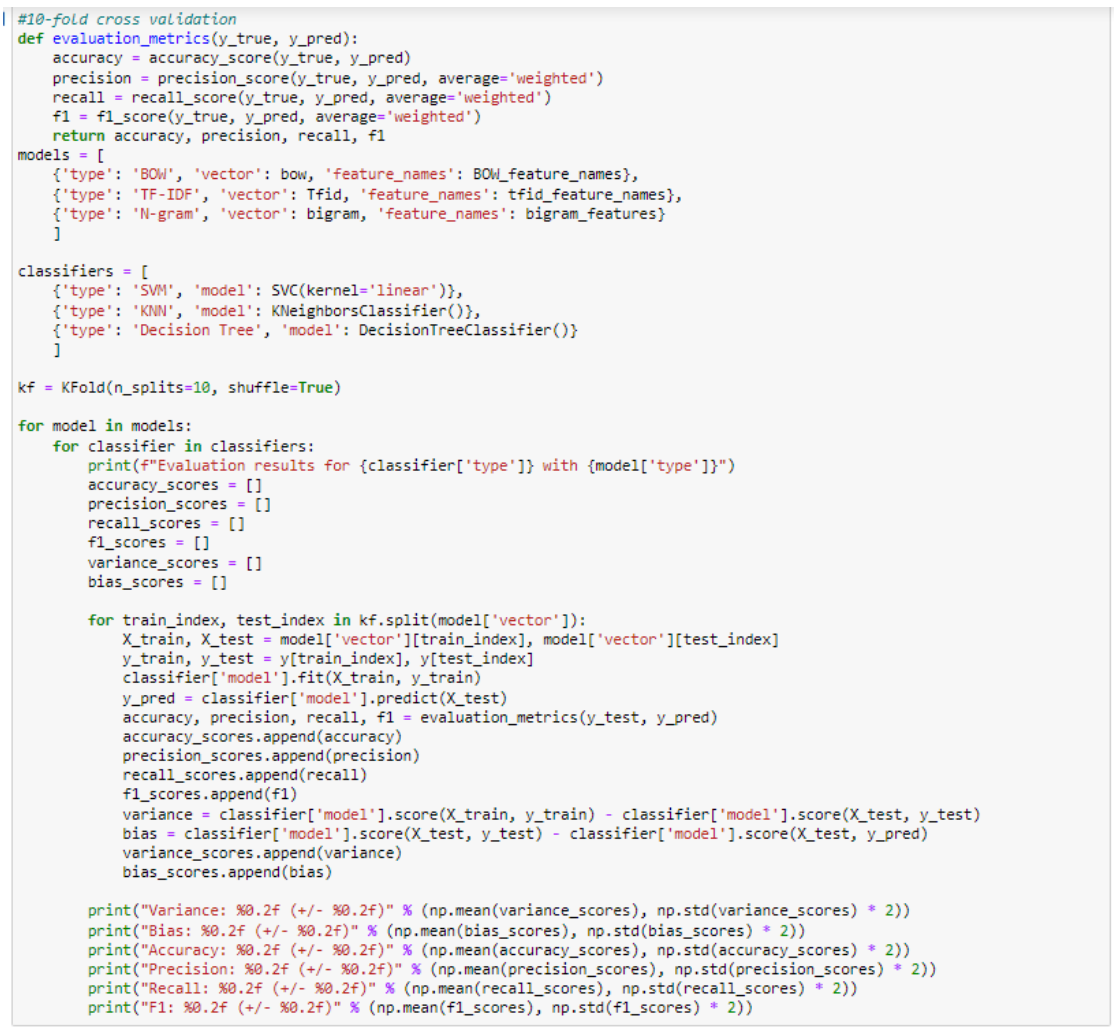
The bias-variance tradeoff is a fundamental concept in machine learning.

Bias refers to the error that results from assuming that the underlying relationship between the features and the target variable is too simple. On the other hand, variance refers to the error that results from the model's excessive sensitivity to small fluctuations in the training data.

There are several ways to analyze the bias and variance of models. Cross-validation is one such method.

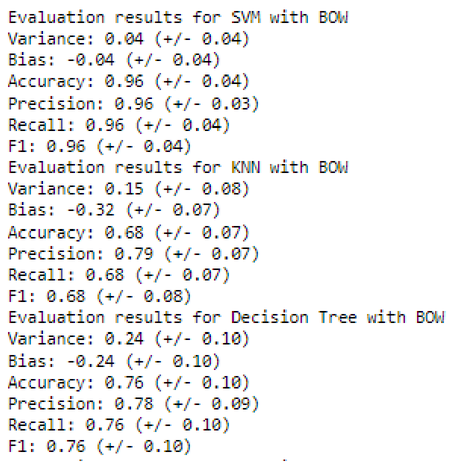
10-fold cross-validation (including variance and bias) was performed for all the models and classifiers, as follows:

Code:

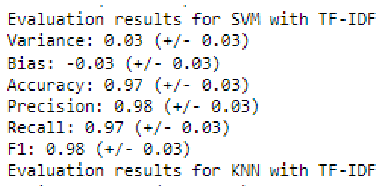


```
#10-fold cross validation
def evaluation_metrics(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return accuracy, precision, recall, f1
models = [
    {'type': 'BOW', 'vector': bow, 'feature_names': BOW_feature_names},
    {'type': 'TF-IDF', 'vector': Tfid, 'feature_names': tfid_feature_names},
    {'type': 'N-gram', 'vector': bigram, 'feature_names': bigram_features}
    ]

classifiers = [
    {'type': 'SVM', 'model': SVC(kernel='linear')},
    {'type': 'KNN', 'model': KNeighborsClassifier()},
    {'type': 'Decision Tree', 'model': DecisionTreeClassifier()}
    ]

kf = KFold(n_splits=10, shuffle=True)

for model in models:
    for classifier in classifiers:
        print(f"Evaluation results for {classifier['type']} with {model['type']}")
        accuracy_scores = []
        precision_scores = []
        recall_scores = []
        f1_scores = []
        variance_scores = []
        bias_scores = []

        for train_index, test_index in kf.split(model['vector']):
            X_train, X_test = model['vector'][train_index], model['vector'][test_index]
            y_train, y_test = y[train_index], y[test_index]
            classifier['model'].fit(X_train, y_train)
            y_pred = classifier['model'].predict(X_test)
            accuracy, precision, recall, f1 = evaluation_metrics(y_test, y_pred)
            accuracy_scores.append(accuracy)
            precision_scores.append(precision)
            recall_scores.append(recall)
            f1_scores.append(f1)
            variance = classifier['model'].score(X_train, y_train) - classifier['model'].score(X_test, y_test)
            bias = classifier['model'].score(X_test, y_test) - classifier['model'].score(X_test, y_pred)
            variance_scores.append(variance)
            bias_scores.append(bias)

        print("Variance: %0.2f (+/- %0.2f)" % (np.mean(variance_scores), np.std(variance_scores) * 2))
        print("Bias: %0.2f (+/- %0.2f)" % (np.mean(bias_scores), np.std(bias_scores) * 2))
        print("Accuracy: %0.2f (+/- %0.2f)" % (np.mean(accuracy_scores), np.std(accuracy_scores) * 2))
        print("Precision: %0.2f (+/- %0.2f)" % (np.mean(precision_scores), np.std(precision_scores) * 2))
        print("Recall: %0.2f (+/- %0.2f)" % (np.mean(recall_scores), np.std(recall_scores) * 2))
        print("F1: %0.2f (+/- %0.2f)" % (np.mean(f1_scores), np.std(f1_scores) * 2))
```

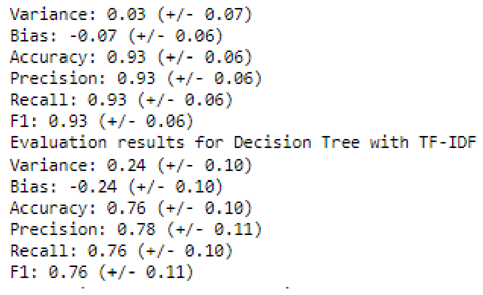
Results:



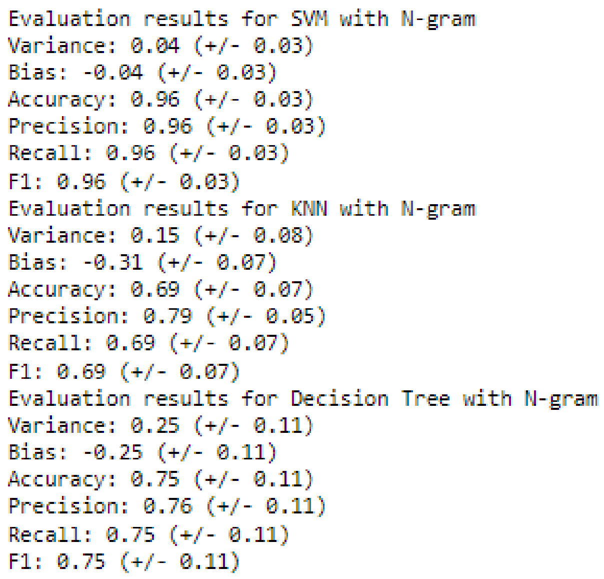
```
Evaluation results for SVM with BOW
Variance: 0.04 (+/- 0.04)
Bias: -0.04 (+/- 0.04)
Accuracy: 0.96 (+/- 0.04)
Precision: 0.96 (+/- 0.03)
Recall: 0.96 (+/- 0.04)
F1: 0.96 (+/- 0.04)
Evaluation results for KNN with BOW
Variance: 0.15 (+/- 0.08)
Bias: -0.32 (+/- 0.07)
Accuracy: 0.68 (+/- 0.07)
Precision: 0.79 (+/- 0.07)
Recall: 0.68 (+/- 0.07)
F1: 0.68 (+/- 0.08)
Evaluation results for Decision Tree with BOW
Variance: 0.24 (+/- 0.10)
Bias: -0.24 (+/- 0.10)
Accuracy: 0.76 (+/- 0.10)
Precision: 0.78 (+/- 0.09)
Recall: 0.76 (+/- 0.10)
F1: 0.76 (+/- 0.10)
```



```
Evaluation results for SVM with TF-IDF
Variance: 0.03 (+/- 0.03)
Bias: -0.03 (+/- 0.03)
Accuracy: 0.97 (+/- 0.03)
Precision: 0.98 (+/- 0.03)
Recall: 0.97 (+/- 0.03)
F1: 0.98 (+/- 0.03)
Evaluation results for KNN with TF-IDF
```



```
Variance: 0.03 (+/- 0.07)
Bias: -0.07 (+/- 0.06)
Accuracy: 0.93 (+/- 0.06)
Precision: 0.93 (+/- 0.06)
Recall: 0.93 (+/- 0.06)
F1: 0.93 (+/- 0.06)
Evaluation results for Decision Tree with TF-IDF
Variance: 0.24 (+/- 0.10)
Bias: -0.24 (+/- 0.10)
Accuracy: 0.76 (+/- 0.10)
Precision: 0.78 (+/- 0.11)
Recall: 0.76 (+/- 0.10)
F1: 0.76 (+/- 0.11)
```



```
Evaluation results for SVM with N-gram
Variance: 0.04 (+/- 0.03)
Bias: -0.04 (+/- 0.03)
Accuracy: 0.96 (+/- 0.03)
Precision: 0.96 (+/- 0.03)
Recall: 0.96 (+/- 0.03)
F1: 0.96 (+/- 0.03)
Evaluation results for KNN with N-gram
Variance: 0.15 (+/- 0.08)
Bias: -0.31 (+/- 0.07)
Accuracy: 0.69 (+/- 0.07)
Precision: 0.79 (+/- 0.05)
Recall: 0.69 (+/- 0.07)
F1: 0.69 (+/- 0.07)
Evaluation results for Decision Tree with N-gram
Variance: 0.25 (+/- 0.11)
Bias: -0.25 (+/- 0.11)
Accuracy: 0.75 (+/- 0.11)
Precision: 0.76 (+/- 0.11)
Recall: 0.75 (+/- 0.11)
F1: 0.75 (+/- 0.11)
```

After analyzing the results, SVM with TF-IDF was found to the most promising model.

Champion Model

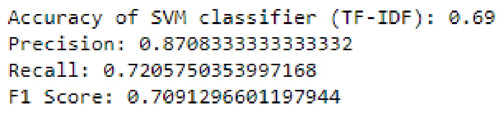
played with the champion model, SVM with TF-IDF, by trying to control the prediction and monitor the machine's behavior, as follows:

a) Modifying the kernel type:

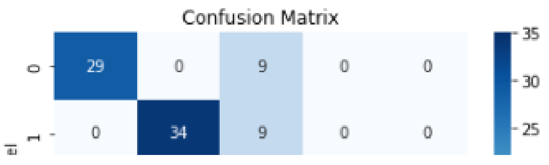
The kernel type was modified from *rbf* (radial basis function) to *poly* (polynomial).

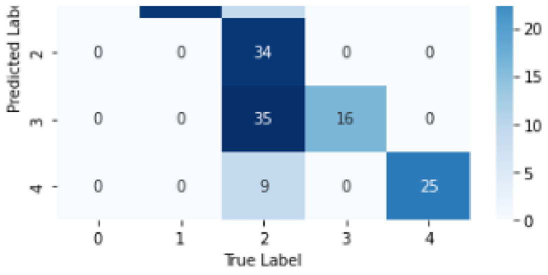
```
#Change the kernel from rbf to poly
clf = SVC(kernel='poly', random_state=0)
```

The results changed negatively such that the model became less accurate by more than 20% as follows:



```
Accuracy of SVM classifier (TF-IDF): 0.69
Precision: 0.8708333333333332
Recall: 0.7205750353997168
F1 Score: 0.7091296601197944
```





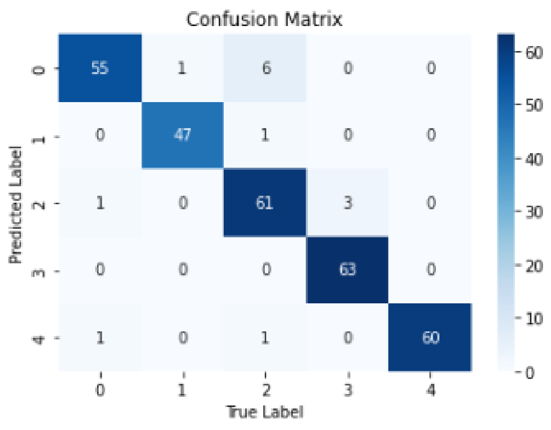
b) Modifying the sample size:

The test sample size was modified from 0.2 to 0.3 as shown below:

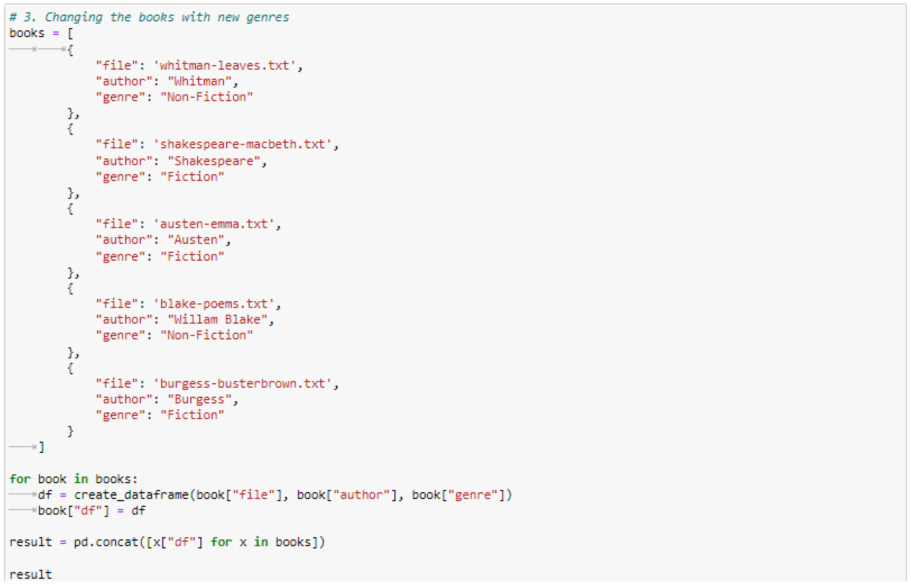
```
#Split the data into training and testing sets
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X_Tfid, y, test_size=0.3, random_state=0)
```

The accuracy increased from 0.69 to 0.953:

```
Accuracy of SVM classifier (TF-IDF): 0.9533333333333334
Precision: 0.9565364745856737
Recall: 0.954493382961125
F1 Score: 0.9533617304188905
```



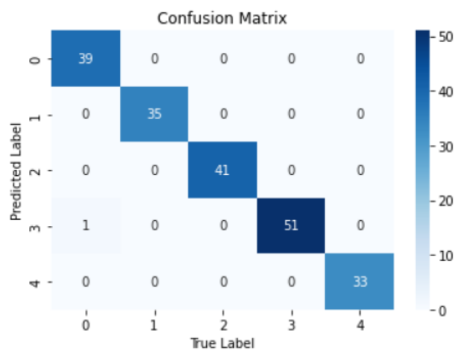
c) Modifying the data by providing new genres:



```
# 3. Changing the books with new genres
books = [
        {
            "file": 'whitman-leaves.txt',
            "author": "Whitman",
            "genre": "Non-Fiction"
        },
        {
            "file": 'shakespeare-macbeth.txt',
            "author": "Shakespeare",
            "genre": "Fiction"
        },
        {
            "file": 'austen-emma.txt',
            "author": "Austen",
            "genre": "Fiction"
        },
        {
            "file": 'blake-poems.txt',
            "author": "Willam Blake",
            "genre": "Non-Fiction"
        },
        {
            "file": 'burgess-busterbrown.txt',
            "author": "Burgess",
            "genre": "Fiction"
        }
    ]

for book in books:
    df = create_dataframe(book["file"], book["author"], book["genre"])
    book["df"] = df

result = pd.concat([x["df"] for x in books])

result
```

The dataset was changed by adding new genre. The champion model was challenged with this dataset, and the following result was obtained:

```
Accuracy of SVM classifier (TF-IDF): 0.995
Precision: 0.9949999999999999
Recall: 0.9961538461538462
F1 Score: 0.99500737372496
```



The accuracy improved from 0.98 to 0.995 (almost 100%). This could be because of the size of the dataset and the number of genres available.

V) Error Analysis

The misclassified data or classification errors are as follows:

```python
def get_misclassified(classifier, X_test, y_test):
    y_pred = classifier.predict(X_test)
    misclassified = []
    for i in range(len(y_test)):
        if y_test[i] != y_pred[i]:
            misclassified.append((i, y_test[i], y_pred[i]))
    return misclassified
```

```python
#classification_error  = misclassified_data(X_train, y_train, X_test, y_test, y_pred)
classification_error  = get_misclassified(clf, X_test, y_test)
print(classification_error)
```

Result for the champion model:

```
[(36, 'Bryant', 'Carroll'), (141, 'Melville', 'Carroll')]
```

The model predicted authors incorrectly, twice (for 2 partitions).

When the partitions for these books were analyzed, it was found that there were commonalities that might have confused the machine.

As literary works, Bryant's stories, Moby Dick, and Alice's Adventures in Wonderland share some common elements, such as:

1. Theme of Adventure: All three works have the theme of adventure and exploration. In Bryant's stories, the adventures may be set in the wilderness or other remote places. In Moby Dick, the journey is a whaling expedition. Alice's Adventures in Wonderland is a

journey through a strange and fantastical world.

2. Use of Imagery: All three works make use of vivid and imaginative imagery to create a sense of place, mood, and character. This can range from the surreal landscapes of Alice's Wonderland to the descriptions of whales in Moby Dick.

3. Characterization: All three works feature memorable and well-defined characters, whether they are the protagonist or supporting characters. From Ishmael in Moby Dick to Alice in her own adventure, these characters are often quirky, whimsical, or quirky in some way.

4. Style: All three works have a distinctive style that sets them apart from other works in their genre. From the poetic language of Bryant's stories to the surreal and fantastical elements of Alice's adventures, each work has its own unique style that contributes to its overall appeal.