University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

uOttawa

2021 Summer DTI5125 Data Science Applications

Assignment Three

Classification Assignment (Group) Report

Group Members:
Salma Mousa, Tokka Hassan, Reyad Melies, Shahenda Youssef

Group ID.: DSA_202101_ 10

Under Supervision: Dr. Arya Rahgozar

Submission Date: 30/05/2021

Table of Contents

LIST OF FIGURES

INTRODUCTION

NLP is one of the most interesting topics in the data science field, which is in continuous advancement and progression. Natural language processing and text analysis aims to extract and identify various information from a piece of writing such as a social media post, books, and newspaper articles, and this serves as very useful information for numerous areas of research and business. Information extracted can be emotions, tone, writing style and many more. This project focuses on text Classification that works on identifying different authors writing styles and predict to which author or genre the piece of writing belongs.

1 REPORT BODY

1.1 Data Preparation

1.1.1 Clean Data

The first step in data preparation is data cleaning from any unwanted data that can affect the model results.

We removed all non-alphabetic characters, stop words and then we used the stemming function to reduce the word to its stem.

```
text = re.sub(r'[^\w\s]', '', str(text).lower().strip())
text = re.sub(r'[\d_]', '', text)

print("--- removing stop words ---")           #stop words
word_tokens = word_tokenize(text)
filtered_words = []
for w in word_tokens:
    if w not in stop_words:
        filtered_words.append(w)

print("--- stemming ---")                       #stemming
new_text = stemming_text(filtered_words)
```

Figure 1 – Removing non-alphabetic characters code

Now the data consists of only words in lower case separated by white spaces and reduced to its stem.

1.1.2 Partition Data

After cleaning the data, it is ready for partitioning. We started by splitting the text to words, then we made a list of partitions each of 100 words. Finally, we took random 200 partitions and repeated the steps for five books of the same genre.

All partitions and their book labels are added to a pandas dataframe.

```
for word in filtered_sentence:
  temp.append(word)
  if len(temp) == 100:
   list_of_lists.append(temp)
   temp=[]

for i in range(200):
 ran = random.randint(0, len(list_of_lists)-1)
 list_of_lists1.append([listToString(list_of_lists[ran]), x])

dataFrame = pd.DataFrame(list_of_lists1, columns=["partition", "book"])
global dataFrameT
dataFrameT = dataFrameT.append(dataFrame, ignore_index = True)
```

Figure 2 - Text partitioning code

And the resulted dataframe is as follows.

	partition	book
0	bear run hard ever could sammi jay fli behind ...	burgess-busterbrown.txt
1	fish thought pretti soon two three swam right ...	burgess-busterbrown.txt
2	friend fear worri put end littl joe otter foun...	burgess-busterbrown.txt
3	hide place reach green forest kept right deepe...	burgess-busterbrown.txt
4	heedless bad said though didnt let grandfath f...	burgess-busterbrown.txt

Figure 3 - Dataframe after text partitioning

1.1.3 Label Data

Now the dataframe is ready with the partitions and their book names, we can start labeling the data. This is done simply by separating the authors name from the book label, then adding it to the dataframe as a new column.

	partition	book	Author
0	bear run hard ever could sammi jay fli behind ...	burgess-busterbrown.txt	burgess
1	fish thought pretti soon two three swam right ...	burgess-busterbrown.txt	burgess
2	friend fear worri put end littl joe otter foun...	burgess-busterbrown.txt	burgess
3	hide place reach green forest kept right deepe...	burgess-busterbrown.txt	burgess
4	heedless bad said though didnt let grandfath f...	burgess-busterbrown.txt	burgess

Figure 4 - Dataframe after labeling partitions

1.2 Feature Extraction

1.2.1 Sentiment Analysis

The first feature we wanted to explore is the sentiment of the book partitions. We used the Sentiment analysis function from the TextBlob library to obtain the sentiment scores.

```
from textblob import TextBlob
dataFrameT["Sentiment"] = dataFrameT["partition"].apply(lambda x:
                    TextBlob(x).sentiment.polarity)
```

Figure 5- Sentiment analysis code

	partition	book	Author	Sentiment
0	wrap lilli leaf ah weep littl voic thou canst ...	blake-poems.txt	blake	0.167273
1	mild meekin mouth contagi taint thi wine doth ...	blake-poems.txt	blake	0.083333
2	pray distress pray human form divin love merci...	blake-poems.txt	blake	0.171282
3	imag truth newborn doubt fled cloud reason dar...	blake-poems.txt	blake	0.014286
4	night sun descend west even star shine bird si...	blake-poems.txt	blake	0.257245

Figure 6- Sentiment analysis scores

1.2.2 Data Visualization

We generated some plots that might give us an insight about the data presented.

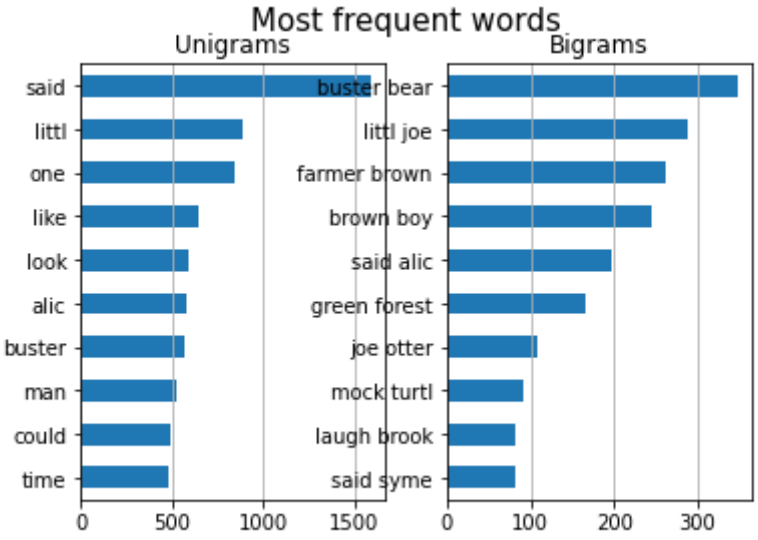

Figure 7- Top Frequent words

Figure 8 - Wordcloud

By examining both plots, we can see that the words that are frequent in unigrams are common words while when using the bigrams it gives detailed words and more names, which are repeated up to 300 times and more, so we can deduce that using unigram will probably give a better accuracy since the words it presents are common and related to the writing style more.

1.2.3 Bag of Words

For implementing the bag of words method, we used the count vectorizer approach, where the resultant represents a sparse matrix of words that are most frequent in the partitions, with a maximum number of 2500 words.

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer(max_features = 2500)
X_train_BoW = count_vect.fit_transform(dataFrameT["partition"])
X_train_BoW
```

```
<600x2500 sparse matrix of type '<class 'numpy.int64'>'
        with 42615 stored elements in Compressed Sparse Row format>
```

Figure 9 - Bag of words code

yonder	young	younger	youngest	youth	youthtim	zeal	Sentiment
0	0	0	0	0	0	0	0.014286
0	0	0	0	0	0	0	0.152500
0	0	0	0	0	0	0	0.083333
0	0	0	0	0	0	0	0.217647
0	0	0	0	1	0	0	-0.041544

Figure 10- Bag of words dataframe

1.2.4 N-Grams

To implement the n-grams, we used the same approach as in the case of bag of words, where we used the count vectorizer with the parameter "ngram-range", and specified the range to be (1,2).

```
from sklearn.feature_extraction.text import CountVectorizer          # we
count_vect = CountVectorizer(ngram_range = (1, 2), max_features = 2500)
X_train_N_grams = count_vect.fit_transform(dataFrameT["partition"])
X_train_N_grams
```

```
<600x2500 sparse matrix of type '<class 'numpy.int64'>'
        with 46468 stored elements in Compressed Sparse Row format>
```

Figure 11- N-grams implementation code

yet tongu	yet visit	yield	yonder	young	young father	youth	zeal	Sentiment
0	0	0	0	0	0	0	0	0.014286
0	0	0	0	0	0	0	0	0.152500
0	0	0	0	0	0	0	0	0.083333
0	0	0	0	0	0	0	0	0.217647
0	0	0	0	0	0	1	0	-0.041544

Figure 12 - N-grmas resultant dataframe

1.2.5 TFIDF Vectorizer with N-Grams

Another test we prepared is using the TFIDF vectorizer which transforms text to feature vectors that can be used as input to estimator. We set the parameter "max-features" to 2500.

account	acquaint	across	act	actual	ad	addit	address	admir	admit	a
0.0	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	
0.0	0.000000	0.000000	0.0	0.080341	0.0	0.0	0.0	0.0	0.0	
0.0	0.000000	0.061685	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	
0.0	0.092412	0.000000	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	
0.0	0.000000	0.000000	0.0	0.000000	0.0	0.0	0.0	0.0	0.0	

Figure 13 - Dataframe result from TFIDF vectorizer with n-grams

1.2.6 TFIDF Vectorizer without N-Grams

Same steps as the TFIDF vectorizer but without using n-grams.

acquaint	across	act	activ	actor	actual	ad	addit	a
0.000000	0.000000	0.0	0.0	0.0	0.000000	0.0	0.0	
0.000000	0.000000	0.0	0.0	0.0	0.093011	0.0	0.0	
0.000000	0.074634	0.0	0.0	0.0	0.000000	0.0	0.0	
0.103577	0.000000	0.0	0.0	0.0	0.000000	0.0	0.0	
0.000000	0.000000	0.0	0.0	0.0	0.000000	0.0	0.0	

Figure 14- Dataframe for TFIDF vectorizer without n-grams

1.3 Classification

1.3.1 Decision Tree Model

After the feature extraction step, we can now start implementing our models, but first we shuffled the data to get the best results and to make sure the model does not train on a class more than the others do.

```
# first we join the author column
author = dataFrameT["Author"]
dataset_countV = dataset_countV.join(author, rsuffix = "_")

# then we shuffle the data
dataset_countV = dataset_countV.sample(frac = 1)

# then we split the input and the target
dataset_countV_target = dataset_countV['Author']
dataset_countV_input = dataset_countV.drop(columns = ['Author'])
```

Figure 15 - Shuffling data code

Now the Data is ready to be split into train and test datasets.

We choose the split ration to be 0.4.

First, we will see the results of using the decision tree model with the dataset from BoW.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import ShuffleSplit, cross_val_score, cross_val_predict

tree_model = DecisionTreeClassifier(random_state=0)
tree_model.fit(X_train_countV, y_train_countV)

scores = cross_val_score(tree_model, X_train_countV, y_train_countV, cv= 10 )
print('Accuracy of cross-validation: ' , scores.mean())

y_pred = tree_model.predict(X_test_countV)

print('Accuracy of test', tree_model.score(X_test_countV, y_test_countV))
```

Figure 16 - Decision tree code for BoW dataset

```
Accuracy of cross-validation:  0.9599999999999997
Accuracy of test 0.9525
```

Figure 17- Decision tree model accuracy with BoW dataset

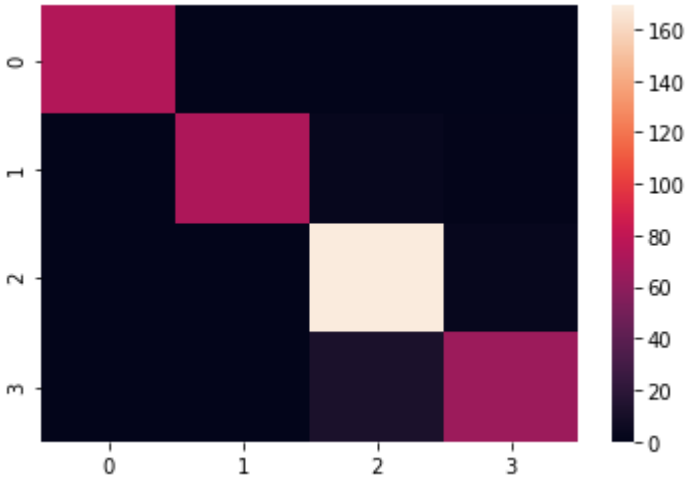

Figure 18 - Decision tree model with BoW dataset

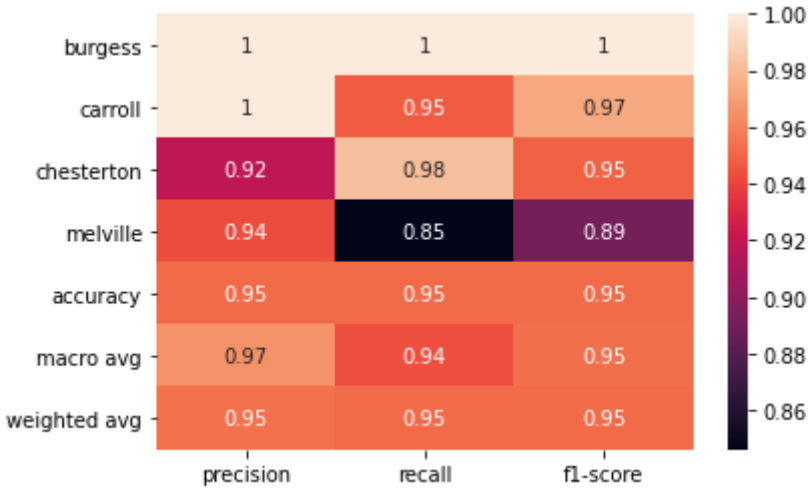

Figure 19 - Decision tree model precision, recall and fscore with BoW dataset

Second, we will see the results of using the decision tree model with the n-grams.

```
Accuracy of cross-validation:  0.9633333333333332
Accuracy of test 0.975
```

Figure 20 - Decision tree model accuracy using n-grams dataset

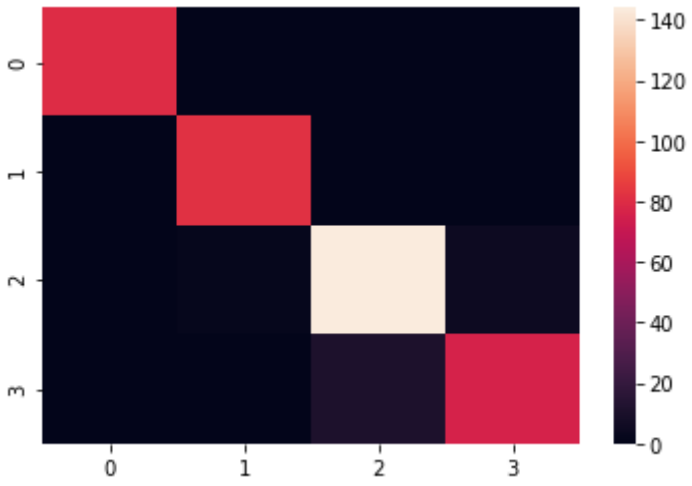

Figure 21 - Decision tree model heat map using n-grams dataset

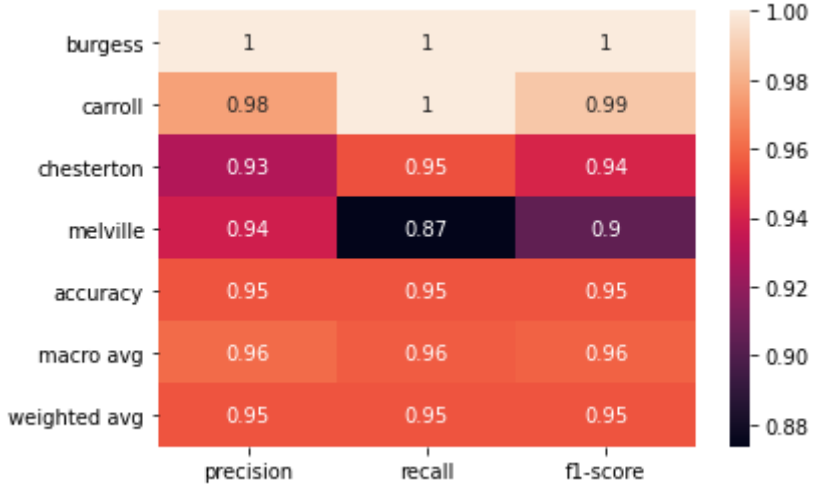

Figure 22 - Decision tree model percision, recall and fscore using n-grams dataset

Third, we will see the results of using the decision tree model but with the features we obtained from using the TFIDF vectorizer with n-grams.

```
Accuracy of cross validation:  0.9633333333333333
Accuracy of test 0.9525
```

Figure 23 - Deicison tree model accuracy using TFIDF with n-grams dataset

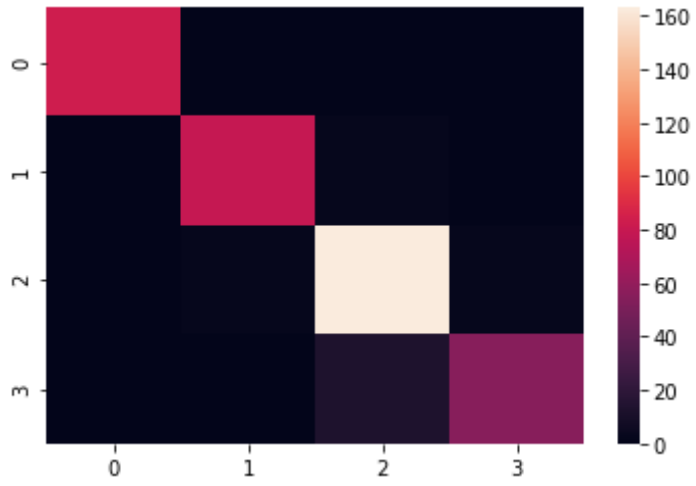

Figure 24 - Decision tree model heat map using TFIDF with n-grams dataset

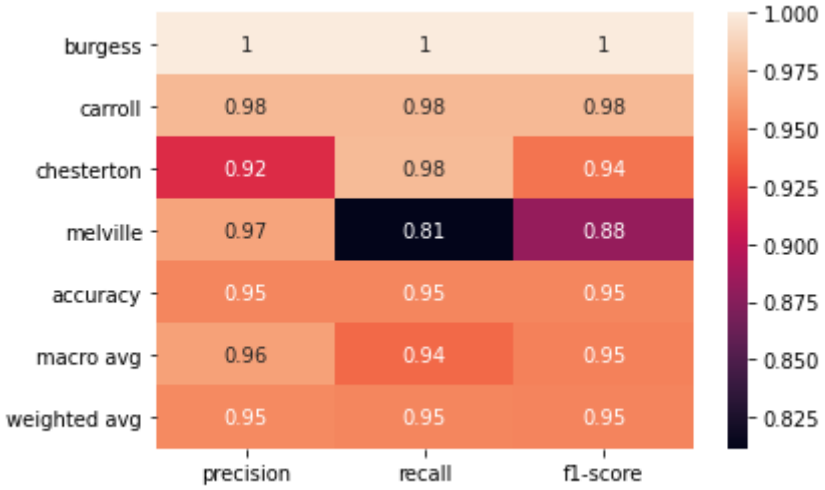

Figure 25 - Decision tree model percision, relcall and fscore using TFIDF with n-grams dataset

Finally, we will see the results of using the decision tree model using the features we obtained from using the TFIDF vectorizer but without n-grams.

```
Accuracy of cross validation:  0.9399999999999998
Accuracy of test 0.9675
```

Figure 26 - Decision tree model accuracy using TFIDF dataset

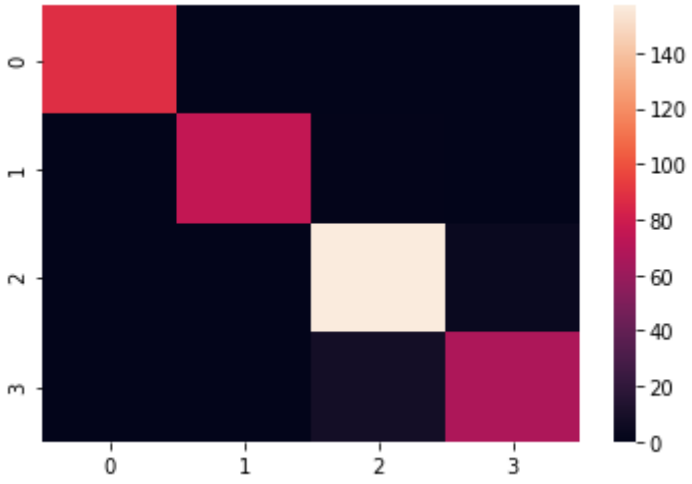

Figure 27 - Decision tree model heat map using TFIDF dataset

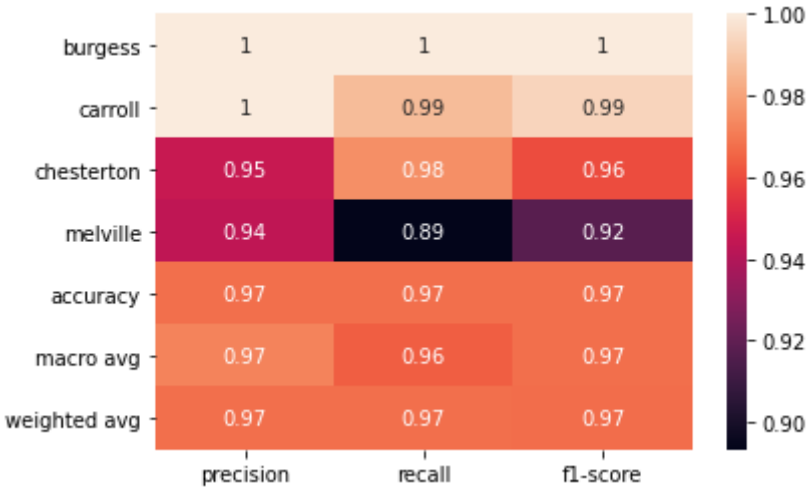

Figure 28 - Decision tree model percision, recall and f1score using TFIDF dataset

1.3.2 SVM Model

We tested twice with the SVM model, once with the parameter "kernel" as RBF which gave a very bad accuracy, then we changed it to Linear, and that gave us a very high accuracy.

First, we will see the results of using the SVM model with the n-grams dataset.

```
# cross_validation
clf = SVC(kernel= 'linear').fit(X_train_countV, y_train_countV)
print('accuracy of cross validation: ', cross_val_score(clf, X_train_countV, y_train_countV, cv=10).mean())
dataSVCCV=clf.predict(X_test_countV)
#y_test_countV
print ('accuracy of test: {}'.format(clf.score(X_test_countV, y_test_countV)))
```

Figure 29 - SVM model code for n-grams dataset

```
accuracy of cross validation:  0.9833333333333332
accuracy of test: 0.9925
```

Figure 30 - SVM model accuracy using n-grams dataset

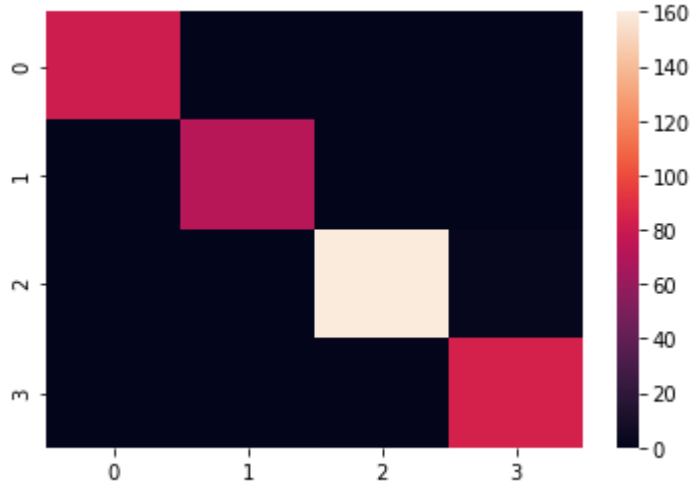

Figure 31 - SVM model heat map using n-grams dataset

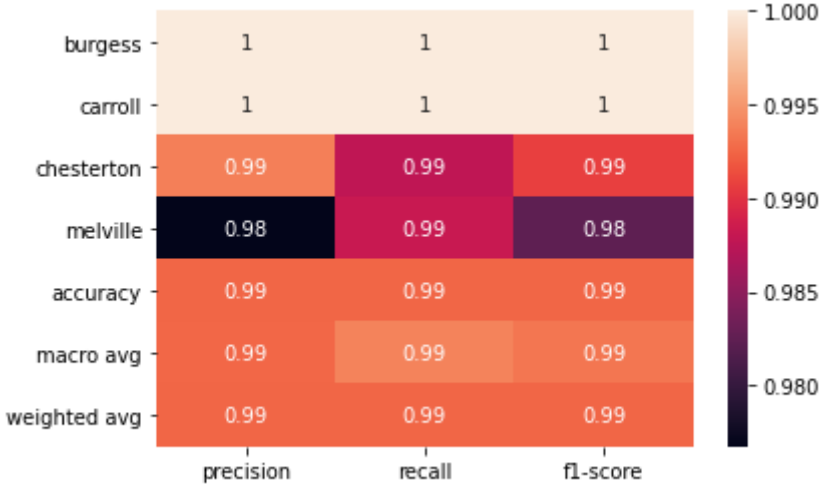

Figure 32 - SVM model percision, recall and f1score using n-grams dataset

And here are the results for the SVM model using BoW dataset.

```
accuracy of cross validation:  0.9799999999999999
accuracy of test: 0.9925
```

Figure 33 - SVM model accuracy using BoW dataset

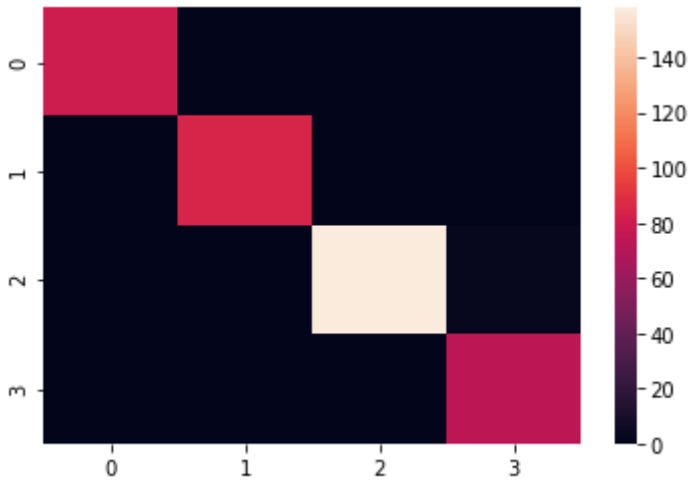

Figure 34 - SVM model heat map using BoW dataset

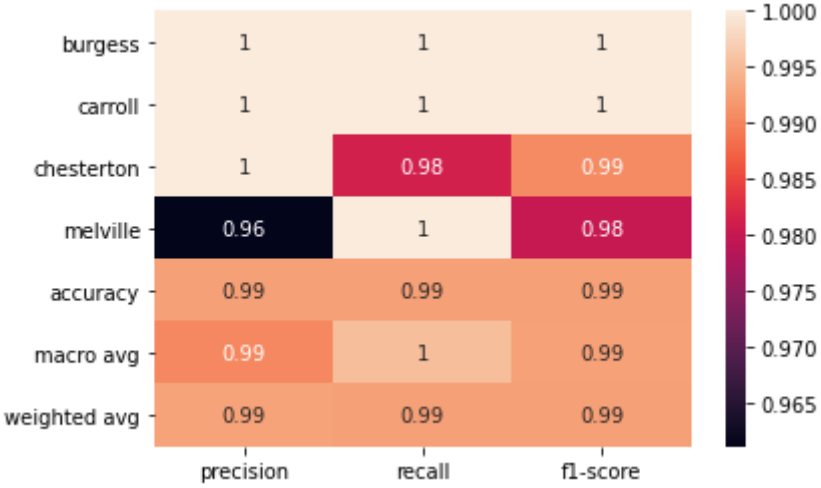

Figure 35 - SVM model percision, recall and f1score BoW dataset

Here we will see the results of using the SVM model with the TFIDF with n-grams dataset.

```
accuracy of cross validation:  0.9883333333333333
accuracy of test: 0.99
```

Figure 36 - SVM model accuracy using TFIDF with n-grams dataset

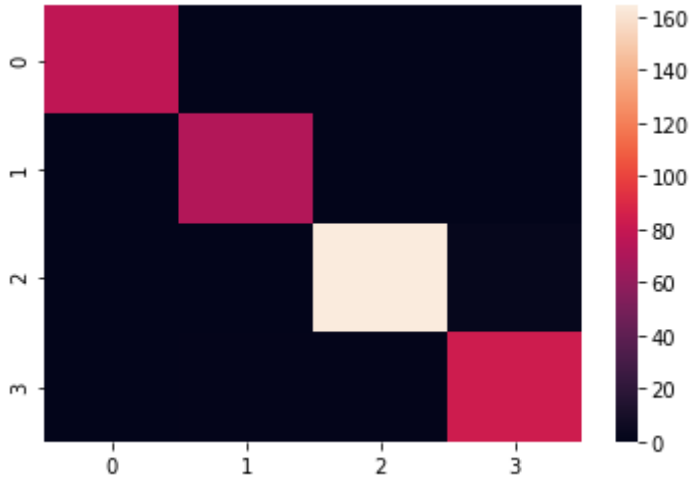

Figure 37 - SVM model heat map using TFIDF with n-grams dataset

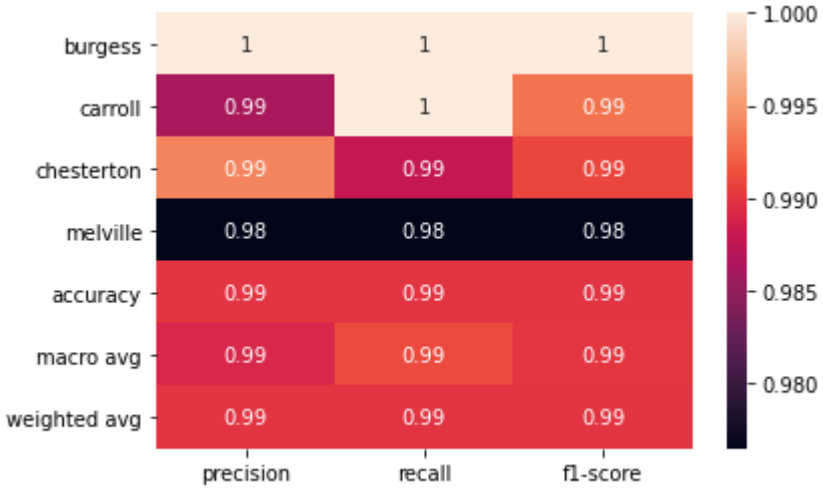

Figure 38 - SVM model percision, recall and f1score using TFIDF with n-grams dataset

And Here we see the results of using the SVM model with TFIDF without n-grams dataset.

```
accuracy of cross validation:  0.985
accuracy of test: 0.99
```

Figure 39 - SVM model accuracy using TFIDF dataset

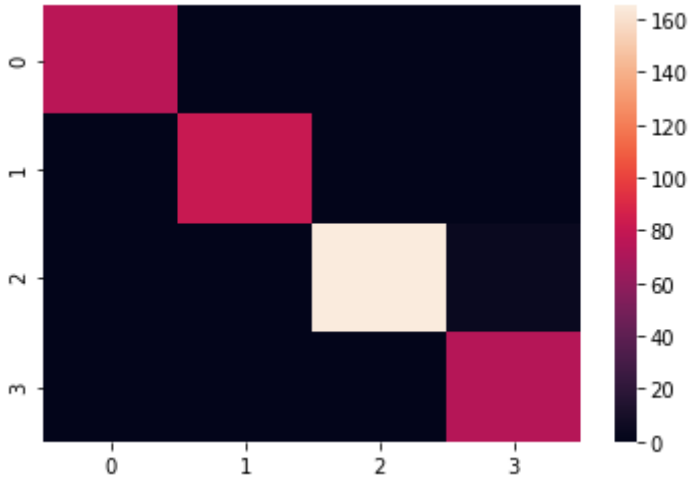

Figure 40 - SVM model heat map using TFIDF dataset

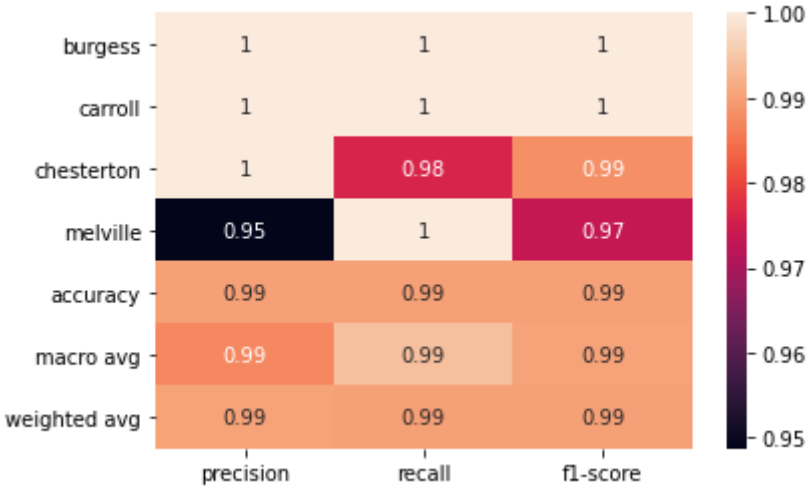

Figure 41 - SVM model percision, recall and f1score using TFIDF dataset

1.3.3 KNN Model

The last model we tested was the K-Near neighbors model.

```
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors= 3, weights='distance')
neigh.fit(X_train_countV, y_train_countV)
print('cross validation accuracy: ', cross_val_score(neigh, X_train_countV, y_train_countV, cv=10).mean())
predicted = neigh.predict(X_test_countV)
print ('accuracy of test: {}'.format(np.mean(predicted == y_test_countV)))
```

Figure 42 - KNN model code

First, we will see the results of using the KNN model with BoW dataset.

```
cross validation accuracy:  0.7833333333333332
accuracy of test: 0.84
```

Figure 43 - KNN model accuracy using BoW dataset

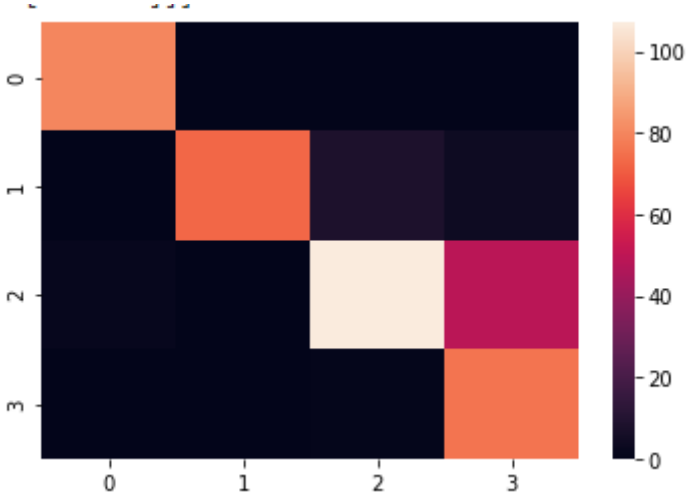

Figure 44 - KNN model heat map using BoW dataset

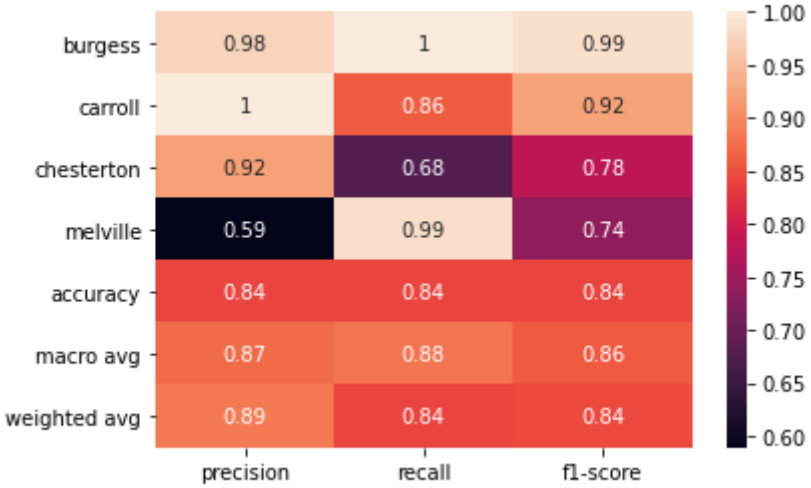

Figure 45 - KNN model percision, recall and f1score using BoW dataset

Second, we will check the results of using the KNN model with the n-grams dataset.

```
cross validation accuracy:  0.6283333333333333
accuracy of test: 0.6425
```

Figure 46 - KNN model accuracy using n-grams dataset

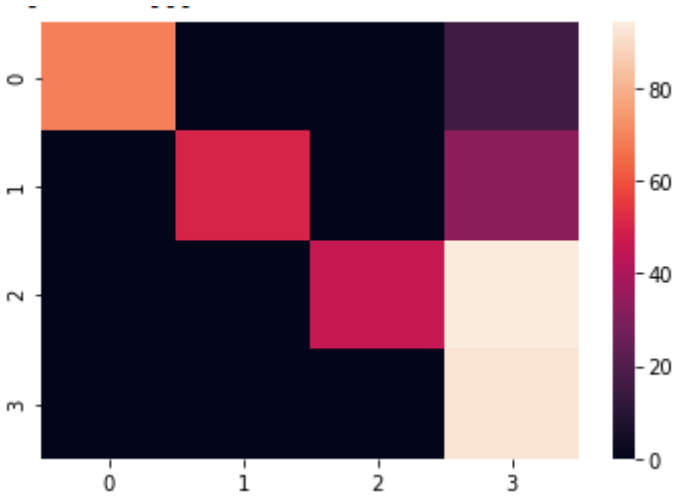

Figure 47 - KNN model heat map using n-grams dataset

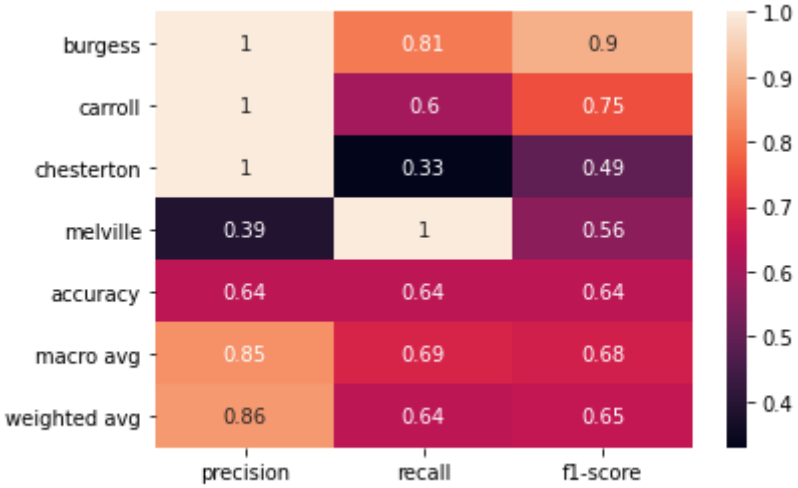

Figure 48 - KNN model percision, recall and f1score using n-grams dataset

Third, we will check the results of using the KNN model with the TFIDF with n-grams dataset.

```
Cross validation accuracy:  0.6833333333333333
accuracy of test: 0.6925
```

Figure 49 - KNN model accuracy usinf the TFIDF with n-grams dataset

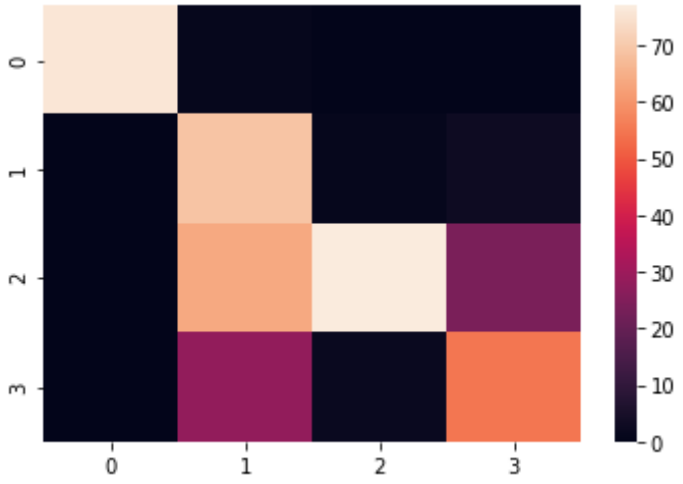

Figure 50 - KNN model heat map using TFIDF with n-grams dataset

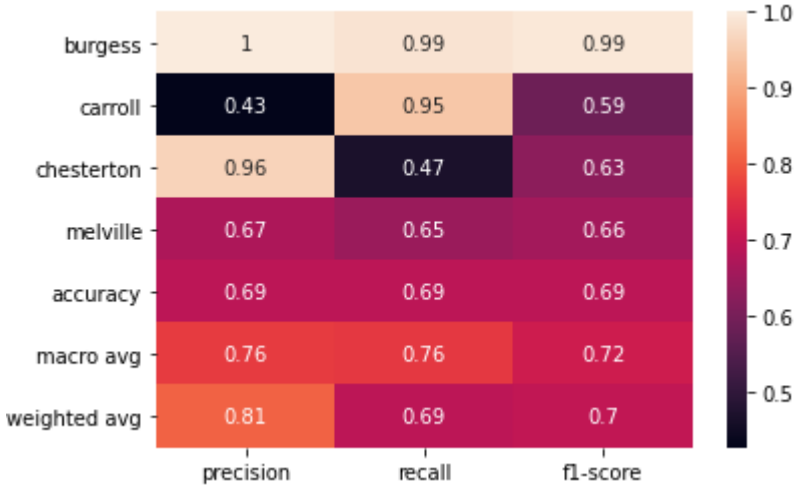

Figure 51 - KNN model percision, recall and f1score using TFIDF with n-grams dataset

Finally, we will see the results of using the KNN model with the TFIDF without n-grams dataset.

```
cross validation accuracy:  0.6133333333333335
accuracy of test: 0.585
```

Figure 52 - KNN model accuracy for TFIDF dataset

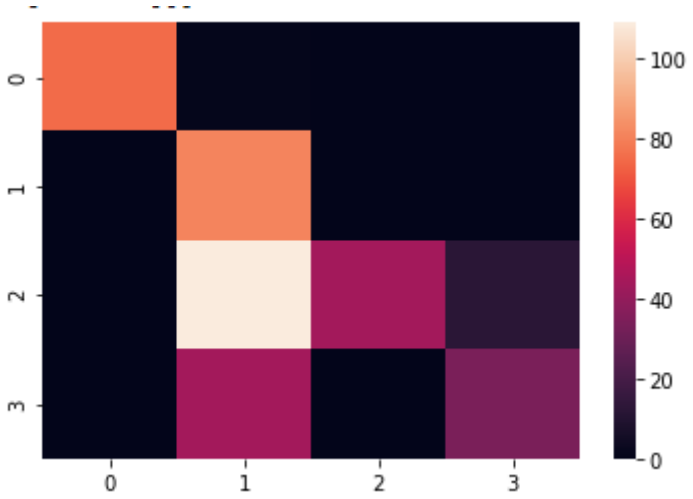

Figure 53 - KNN model heat map using the TFIDF dataset

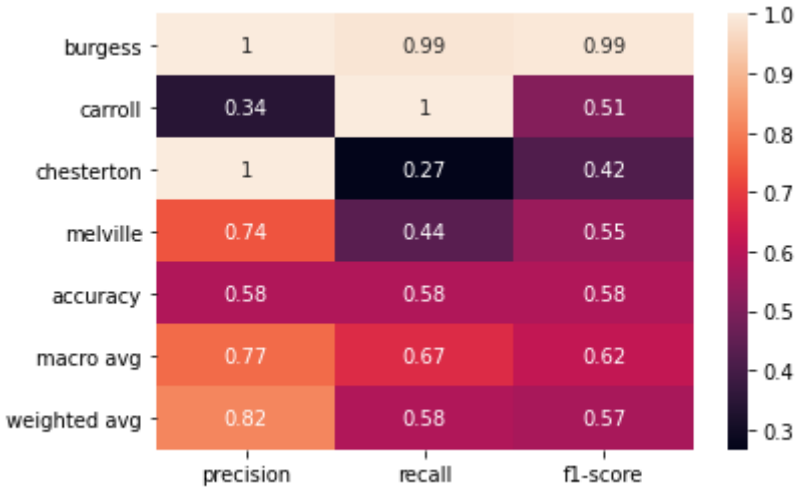

Figure 54 - KNN model percision, recall and f1score using TFIDF dataset

1.3.4 Error analysis and Observations

After testing with four different datasets on three different models, we have observed the following:

First, The KNNs model has the lowest accuracies between all models, and that's given that the model itself is rather weak in performance with complex dependent and independent data.

The model was tested with different parameters, choosing the number of neighbors to be "3" instead of "5", and choosing the weights to be distance instead of uniform, this enhanced the performance of the KNNs greatly.

Second, The Decision tree model has the second best performance out of the three models, which is due to multiple factors.

The model deals with complex features very well as it is a non-linear model and it can model very well the grammatical structure of sentences/phrases.

The models were tested twice with the depth parameter once as "None" and once as "5", when the depth was set to "None" it gave a much higher accuracy than when it was set to "5".

The last model we have is the SVM model, and this model gave the best accuracy for all four test cases we made.

We first tested with the "RBF" or Gaussian kernel which gave an accuracy of 94%, then testing with the "linear" kernel we obtained an accuracy of 98.7%. So there is a noticeable enhancement in the accuracy, also the linear kernel provides faster performance.

The reason why SVM performs so well is that it mainly depends on creating decision boundaries between vectors that belong to a certain group or class, and vectors are just a list of numbers that represents the coordinates in some space, and since we transform texts into large vectors with many features, the SVM works on creating these decision boundaries between these vectors and obtains very good results.

1.4 Champion Model

1.4.1 Choosing the champion model

We have collected all the accuracy data we obtained from each model and it is shown that the SVM model exceeds all the other models in performance and hence we choose it as the champion model.

Note that these results can differ from the ones in the notebook by a slight difference, since each run generates a different dataset.

Feature	Accuracy	Precision	Recall	F-score
KNN Count Vectorizer	77.6%	0.87	0.76	0.76
KNN TF-IDF	59.3%	0.84	0.61	0.62
KNN TF-IDF/N-Grams	63.3%	0.82	0.68	0.67
KNN Count Vectorizer/N-Grams	70.6%	0.84	0.76	0.77
SVM Count Vectorizer	98.1%	0.99	0.99	0.99
SVM TF-IDF	98.7%	0.99	0.99	0.99
SVM TF-IDF/N-Grams	99.3%	0.99	0.99	0.99
SVM Count vectorizer/N-Grams	99.1%	0.99	0.99	0.99
Decision Tree Count vectorizer/N-Grams	94.3%	0.93	0.93	0.93
Decision Tree TF-IDF	94.9%	0.95	0.95	0.95
Decision Tree TF-IDF/N-Grams	94.9%	0.95	0.95	0.95
Decision Tree Count Vectorizer	95.3%	0.95	0.95	0.95

Figure 55 - All Accuracies Table

1.4.2 Drop Accuracy by 20%

First, we dropped number of partitions from 100 to 30 partitions this caused to drop the accuracy by 9%.
To drop the accuracy of the champion model by 20%, we decreased the number of words per partition to 20 words instead of 100, this made it much harder for the model to get meaningful information from the given partitions since the word count is too low.
Here we can see the results of reducing the word count.

```
cross validation accuracy:  0.7533333333333334
accuracy of test: 0.7275
```

Figure 56 - SVM model accuracy using 20 words per partition

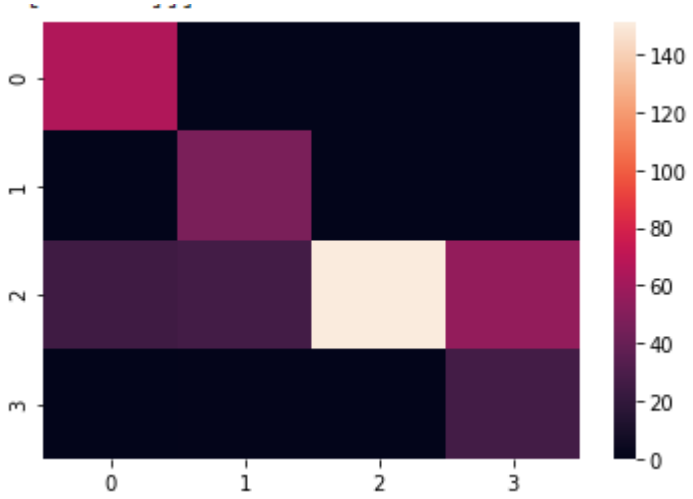

Figure 57 - SVM model heat map using 20 words per partition

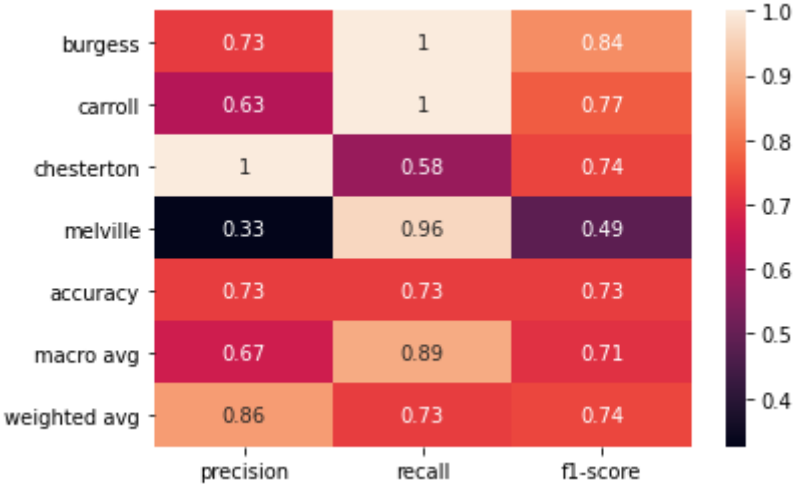

Figure 58 - SVM model percision, recall and f1score using 20 words per partition

We can notice from the heat map and the precision, recall and f1score plot, we can notice that the Melville class is incorrectly classified more than the other classes.

For more investigation, we created a dataframe containing all the wrong predictions.

	Predicted Label	Correct Label
count	108	108
unique	2	3
top	chesterton	melville
freq	107	55

Figure 59 - Dataframe of wrongly predicted labels

We can clearly see that there are two main issues here. First the Melville class is incorrectly classified more frequently than any other class, and second, the incorrect prediction is almost always the Chesterton class. This can be due to the fact that the Chesterton class has double the data of other classes, since we are using two books for the same author Chesterton.

This indicates that data balancing is important for the model to achieve the best performance.

1.4.3 Bias and Variance Calculation

We calculated the bias and variance of the model, before and after we adjusted the training data.

```
from mlxtend.evaluate import bias_variance_decomp
# estimate bias and variance
# clf = SVC(kernel= 'linear')
mse, bias, var = bias_variance_decomp(clf, X_train_tfidfV_ngram,
                                      np.array(string_to_integer(y_train_tfidfV_ngram)),
                                      X_test_tfidfV_ngram, np.array(string_to_integer(y_test_tfidfV_ngram)),
                                      loss='mse', num_rounds=200, random_seed=1)
# summarize results
print('MSE: %.3f' % mse)
print('Bias: %.3f' % bias)
print('Variance: %.3f' % var)
```

Figure 60 - Code for Calculating bias and variance

Here we can see the result before dropping the accuracy by 20%

```
MSE: 1.787
Bias: 1.496
Variance: 0.291
```

Figure 61 - Bias and variance

```
MSE: 1.081
Bias: 0.772
Variance: 0.309
```

Figure 62 - Bias and Variance After

REFERENCES

https://machinelearningmastery.com/calculate-the-bias-variance-trade-off/

https://monkeylearn.com/text-classification-support-vector-machines-svm/

Proceedings of the Annual Symposium of the Institute of Solid Mechanics and Session of the Commission of Acoustics, SISOM 2015