



**Course: DTI5125 - Data Science Applications (Winter 2024)**

**Professor: Dr. Ariya Rahgozar**

**The Classification Project**

**Group 10:**

**Alireza Houshidari (300366752)**

**Brandon Lygo (300030334)**

**Hamed Sarshoghi (300333718)**

**Jonathan Horton (7710257)**

February 2024

# 1. Introduction

In the following study assignment, our academic group selected six classical literature texts to train and evaluate various natural language processing (NLP) models set with the goal of classifying book excerpts to their respective authors. Using Gutenberg text corpus a labeled dataset was obtained for training, evaluation, and testing. Main features being searched in the corpus were the words within them and sequences they were read in. Through a variety of transforms different models were trained to classify the author of a test subset of data and then compared to pick a champion model. Analysis of errors, bias, and variability within the models was also done to better understand how the training or input data may have led to their occurrence with the champion model also receiving changes to better understand why it was the strongest. Through these analyses a better understanding of how different algorithms aid in classification was obtained with initially perceived advanced models such as BERT being challenged by simpler models such as SVM, Naive Bayes, and N-gram.

## 2. Pre-Processing

In this section, we will look specifically at how the data is processed before it is trained on. There are two main stages in the formatting: (1) The data is edited using regular expressions so as to trim off unnecessary text at the top and the bottom of the books, then headings are removed from chapter, part, and book titles; and (2) the data is tokenized into words, lowercased, rid of punctuation, stopwords are removed, stemming is applied, and lemmatization is applied.

### 2.1 Applying Regular Expressions

Each book has its own set of regular expressions in order to remove unnecessary text. At the start of every .txt file that we download from the Gutenberg library, there is some information about the Project Gutenberg License. Training on this text would result in unintended common word sequences between the books, so this data has to be removed. Moreover, training on the table of contents is unnecessary, as the titles are present elsewhere in the book, so it is redundant. Therefore, both of these are removed. This is done by using a regular expression of the following form, in the case of The Brothers Karamazov:

```
# Remove from start of book up until "Alexey Fyodorovitch Karamazov"
text = re.sub(r'^(.*) (Alexey Fyodorovitch Karamazov)', r'\2', text, flags=re.DOTALL)
```

This regular expression searched for the sequence “Alexey Fyodorovitch Karamazov”, which is the first line in the book, and delete all information prior to it. Similarly, for the end of the book, a regular expression is needed to remove unwanted information. In the case of The Brothers Karamazov:

```
# Remove from end of book up
text = re.sub(r'\*\*\* END OF THE PROJECT GUTENBERG EBOOK THE BROTHERS KARAMAZOV \*\*\*$', '', text, flags=re.DOTALL)
```

This line of code removes everything up to and including “\*\*\* END OF THE PROJECT GUTENBERG EBOOK BLEAK HOUSE \*\*\*”. Finally, many of the books also contain chapter

headings, part headings, and sub-book headings. To remove these, a regular expression of the following form is used

```
text = re.sub(r'Chapter [IVXLCDM]+\.', '', text)
```

This line of code removes any text of the form “Chapter \$Roman Numeral\$” where \$Roman Numeral\$ can be any roman numeral. Thus, if the book contained “Chapter III”, it would be removed. Similar regular expressions are used for all 6 books, though there are minor variations from book to book as necessary.

## 2.2 Pre-Processing Text

After the unnecessary text has been removed, the data is tokenized into words, all these words are lowercase, and punctuation is removed. A list of stop words is downloaded from the natural language toolkit (NLP). These are commonly used words like “the”, “and”, and “is” of which our models would likely not benefit from training on. The words are then stemmed, which is to say that endings like “-ing”, “-ly”, “-es”, “-s” are removed from words. For example, the word “running” would transform to “run” after stemming is applied. Lastly, the words are lemmatized. Lemmatizing is the process of converting a word to its base form from a root. For instance, the word “geese” would be converted to “goose”. Both stemming and lemmatization are done using the NLP toolkit.

## 2.3 Bigram Training Data Generation

As a different approach to generating the data, we used bigrams to generate training data for training ensemble methods. The code shown below will be explained by way of example. Say our book produced a list like so: [The quick The dog The quick]. Then a dictionary for the generation of the data would be generated of the form {The: [quick, quick, dog], quick: [The], dog: [The]}. A word at random would then be selected as the starting point of the first partition. For instance, say “The” was chosen as a starting point. A word from its associated list would then be chosen at random as the second word for the first partition. Since quick is present twice in the list, it is more likely to be chosen. Thus, our second word in our first partition could become {“The”, “quick”}, and the third word would be chosen from quick’s list in the dictionary. This process repeats until the partition becomes 100 words long. The advantage of this process is that it is more likely to generate sequences that have word patterns that show up often in each book as opposed to random partitions. This guarantees that models are trained on words with higher frequencies.

```

def partitionText2grams(text):
    section_size = 100
    total_partitions = 200
    #Create the 2 gram dictionary
    ngram = {}
    words = text.split()
    for i in range(len(words) - 1):
        try: #Add the current word if it already exists in the dictionary
            ngram[words[i]].append(words[i + 1])
        except KeyError as _: #If it does not exist in the dictionary, then you create a fresh list
            ngram[words[i]] = []
            ngram[words[i]].append(words[i + 1])

    # Initialize a list to hold all partitions
    all_partitions = []

    # Generate total_partitions number of word lists
    for _ in range(total_partitions):
        words = []
        next_word = random.choice(list(ngram.keys()))
        words.append(next_word)
        for _ in range(section_size - 1): # -1 because we already added the first word
            next_word = random.choice(ngram[next_word])
            words.append(next_word)
        # Add the generated word list to all_partitions
        all_partitions.append(words)

    # Now, all_partitions is a list of 200 word lists, each containing 100 words

    return ngram, all_partitions[:total_partitions]

```

Figure 1 - 2-Gram Code

### 3. Feature Engineering

#### 3.1 Bag of Words and n-grams

The Bag of Words (BoW) Transformer is a technique in natural language processing that helps to represent text data in the form of numerical vectors. This method primarily focuses on the frequency of words used within a document, without taking into consideration their order and structure. In other words, it creates a "bag" of word occurrences by counting the frequency of each unique word in the document. This transformation is particularly useful for various text-based machine learning tasks, such as sentiment analysis, document classification, and information retrieval. Despite its effectiveness in simplifying the representation of textual data, the Bag of Words approach may have limitations in capturing semantic relationships and contextual information. Nevertheless, the Bag of Words Transformer serves as a foundational method in text processing, providing a numerical representation that enables the application of various machine learning algorithms to analyze and classify textual information.

As part of our project, we employed the CountVectorizer module from the sklearn library to convert the textual data into numerical vectors. This conversion is essential as it enables our

machine-learning models to analyze and understand the data. By transforming the text into vectors, we can perform operations such as distance calculation, dimensionality reduction, and clustering, which would not have been possible with raw text data.

When analyzing the ensemble methods, we also employed the `ngram_range = (1,2)` option for `CountVectorizer`. This option enables the option to capture sequences of 2 words in the provided text. For example, in the case of the sentence “I love hockey a lot”, the option would capture both unigrams [“I”, “love” “hockey” “a” “lot”] and bigrams [“I love”, “love hockey”, “hockey a”, “a lot”]. Including this option allowed the ensemble methods to capture more context to word sequences.

### **3.2 TF-IDF**

The TF-IDF Transformer is a fundamental tool in natural language processing, which converts text data into numerical vectors. TF-IDF measures the significance of words in a document relative to their frequency across a collection of documents. It consists of two components: Term Frequency (TF), which measures how often a word appears in a specific document, and Inverse Document Frequency (IDF), which quantifies the rarity of a word across the entire corpus. The TF-IDF Transformer multiplies these two values to assign higher weights to terms that are frequent in a document but rare in the overall corpus. This technique mitigates the influence of common words, emphasizing those that contribute more distinctiveness and meaning to a document. The resulting TF-IDF vectors are used in various natural language processing applications, such as text classification, information retrieval, and document similarity analysis, providing a more nuanced representation of textual data compared to simpler methods like the Bag of Words.

We used the `TfidfVectorizer` module of the `sklearn` library to vectorize the texts in the partitions using the TF-IDF method.

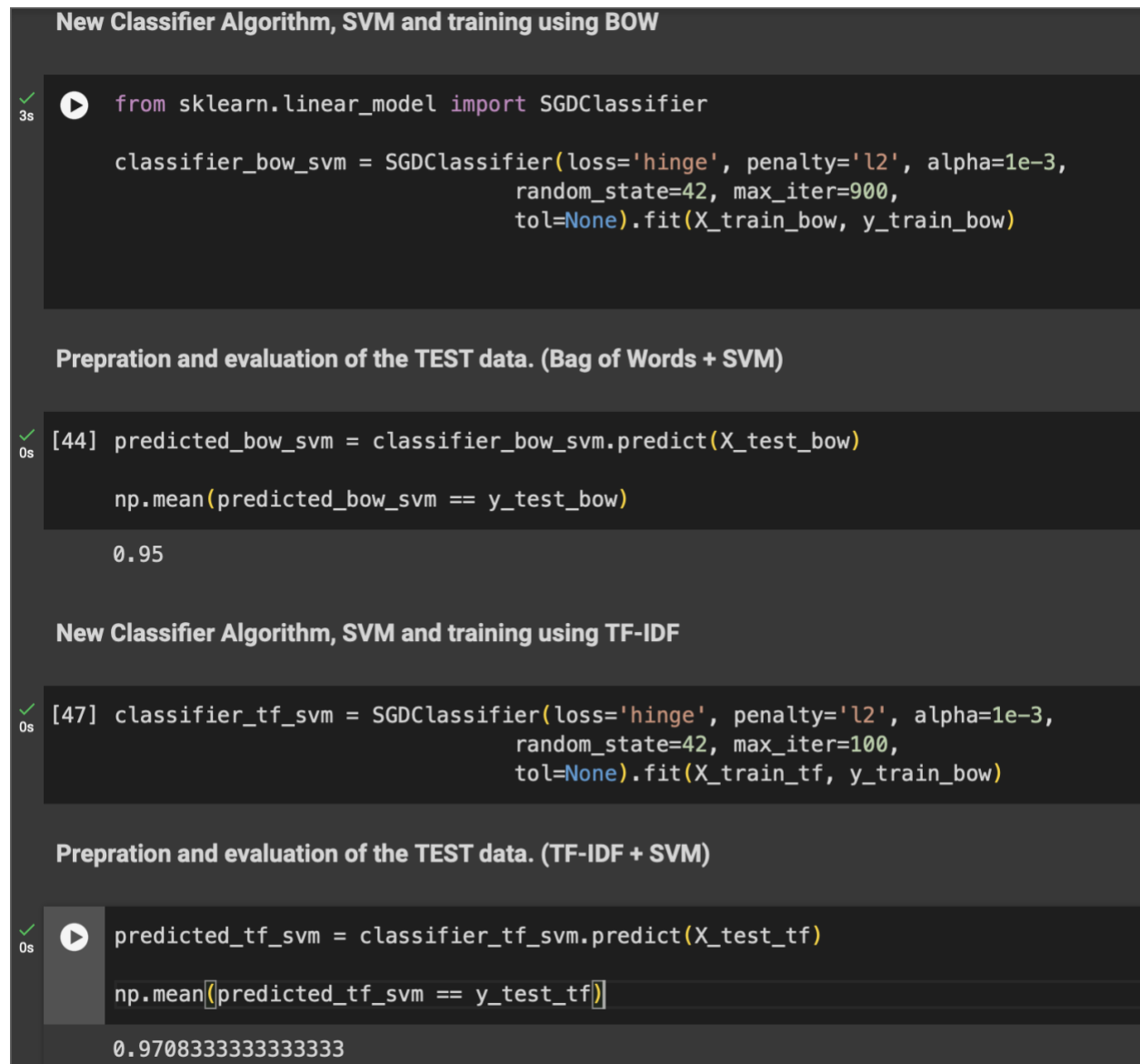
## **4. Classification Methods**

### **4.1. SVM**

The Support Vector Machine (SVM) algorithm is a highly effective and versatile supervised machine learning technique that is used for both classification and regression tasks. Its main goal is to identify the optimal hyperplane that can best separate data points belonging to different classes in a high-dimensional space. One of the main features of SVM is its ability to maximize the margin between classes, which refers to the distance between the hyperplane and the nearest data points from each class, also known as support vectors. SVM can handle not only linearly separable cases but also non-linear relationships through the use of kernel functions that map input data into higher-dimensional spaces. This characteristic makes SVM suitable for various applications such as image classification, text analysis, and bioinformatics as it can capture complex decision

boundaries. SVM is widely adopted in diverse fields of machine learning due to its robustness, versatility, and ability to generalize well to unseen data.

For our project, we utilized the SGDClassifier module of the sklearn library. We set the max\_iter attribute to 900 for BOW and 100 for TF-IDF. We then trained the model using Bag of Words and TF-IDF transformers. We allocated 80% of the data for training in both cases and kept the remaining 20% for testing. After training the data, we used it to predict the results of the test set. We experimented with the attributes and achieved an accuracy of 95% and 97% when using BOW and TF-IDF respectively.



The screenshot displays a Jupyter Notebook interface with three main sections. The first section, titled 'New Classifier Algorithm, SVM and training using BOW', shows the import of SGDClassifier and its training with BOW data, resulting in an accuracy of 0.95. The second section, titled 'New Classifier Algorithm, SVM and training using TF-IDF', shows the training of the model with TF-IDF data, resulting in an accuracy of 0.9708333333333333. Each section includes a code cell with the respective Python code and an output cell showing the accuracy.

```
from sklearn.linear_model import SGDClassifier

classifier_bow_svm = SGDClassifier(loss='hinge', penalty='l2', alpha=1e-3,
                                  random_state=42, max_iter=900,
                                  tol=None).fit(X_train_bow, y_train_bow)
```

```
[44] predicted_bow_svm = classifier_bow_svm.predict(X_test_bow)

np.mean(predicted_bow_svm == y_test_bow)

0.95
```

```
[47] classifier_tf_svm = SGDClassifier(loss='hinge', penalty='l2', alpha=1e-3,
                                       random_state=42, max_iter=100,
                                       tol=None).fit(X_train_tf, y_train_bow)
```

```
predicted_tf_svm = classifier_tf_svm.predict(X_test_tf)

np.mean(predicted_tf_svm == y_test_tf)

0.9708333333333333
```

Figure 2 - SVM Model Output

## 4.2 Naive Bayes

The Naive Bayes algorithm is a machine learning technique used for classification tasks, especially in natural language processing and spam filtering. It is based on Bayes' theorem, which calculates the probability of a hypothesis given observed evidence. The "naive" aspect comes from the assumption of independence between features, meaning that each feature contributes independently to the probability of the class. Despite its simplifying assumption, Naive Bayes often performs remarkably well in practice and is computationally efficient. It works best for high-dimensional datasets and situations where the independence assumption does not significantly impact classification accuracy. Naive Bayes models are easy to implement, require minimal training data, and are robust in the presence of irrelevant features, making them a popular choice for various text classification tasks and applications with limited training data.

We used the MultinomialNB module from the sklearn library to implement the Naive Bayes model. To train the model, we utilized Bag of Words and TF-IDF transformers. We divided the data into two sets, allocating 80% of the data for training and the remaining 20% for testing, in both cases. Once the data was trained, we employed it to predict the results of the test set. During the experimentation phase, we tested various attributes and fine-tuned the model to achieve an accuracy of 96% and 94% when using BOW and TF-IDF, respectively.



#### Classification and training Naive Bayes model on the training dataset transformed by BOW.

```
✓ 0s ▶ from sklearn.naive_bayes import MultinomialNB  
      classifier_bow_nb = MultinomialNB().fit(X_train_bow, y_train_bow)
```

#### Preparation and evaluation of the TEST data. (Bag of Words + Naive Bayes)

```
✓ 0s [23] import numpy as np  
  
      predicted_bow_nb = classifier_bow_nb.predict(X_test_bow)  
      np.mean(predicted_bow_nb == y_test_bow)
```

0.9625

#### Classification and training Naive Bayes model on the training dataset transformed by TF-IDF.

```
✓ 0s [24] classifier_tf_nb = MultinomialNB().fit(X_train_tf, y_train_tf)
```

#### Preparation and evaluation of the TEST data. (TF-IDF + Naive Bayes)

```
✓ 0s [25] predicted_tf_nb = classifier_tf_nb.predict(X_test_tf)  
      np.mean(predicted_tf_nb == y_test_tf)
```

0.9458333333333333

Figure 3 - Naive Bayes Model Output

### 4.3 2-Gram Generations with Various Ensemble Methods

Once the bigram-generated partitions were generated, various ensemble methods were applied in order to determine the optimal model. Note that two data sets were used in this method: (1) the bigram generated dataset which was used for training, and (2) a data set partitioned directly out of the books for testing the trained models. The second data set is necessary in order to fairly compare the generated models to the other classification methods used.

Various ensemble methods were tried, including Random Forest Classifier, Extra Trees Classifier, Gradient Boosting Classifier, AdaBoost Classifier, and the Bagging Classifier combined with the Decision Tree Classifier. Both BOW and TF-IDF were tried in order to engineer the features for



these models, though BOW was generally more successful and therefore became the primary focus for testing classifiers. Of the models used, the results were (1) Random Forest Classifier: 94%, (2) Extra Trees Classifier: 96.5%, (3) Gradient Boosting Classifier: 96.25%, (4) AdaBoost Classifier: 96.25%, and (5) Bagging Classifier combined with the Decision Tree Classifier: 78.8%. Since Extra Trees Classifier had the highest percentage, it was chosen as the model to optimize. After varying its `n_estimators`, `max_depth`, `min_samples_split`, and `min_samples_leaf` and finding the parameters with maximum accuracy, the model was generated randomly 100 times for those parameters in order to obtain the best result.

```

best_score = 0
best_model = None

for _ in range(100):
    # Create a classifier with the best parameters
    classifier = ExtraTreesClassifier(n_estimators=best_params['n_estimators'],
                                     max_depth=best_params['max_depth'],
                                     min_samples_split=best_params['min_samples_split'],
                                     min_samples_leaf=best_params['min_samples_leaf'])

    # Fit the model to training data
    classifier.fit(X_train_bow, y_train_bow)

    # Make predictions on the test data
    predicted = classifier.predict(X_test_bow_2)

    # Calculate the accuracy of the predictions
    score = accuracy_score(y_test_bow_2, predicted)
    print(score)

    # If the score is better than the best found so far, store this classifier as the best
    if score > best_score:
        best_score = score
        best_model = classifier

classifier_bow_et_2_model = best_model
predicted_bow_et_2 = best_model.predict(X_test_bow_2)

print("Best score after 100 iterations: ", best_score)
0.9656666666666666
Best score after 100 iterations: 0.9741666666666666

```

Figure 4 - Optimal Extra Trees Classifier Output

## 4.4 BERT

Introduced by Google AI, Bidirectional Encoder Representations from Transformers (BERT) was a major stepping stone in NLP models. It operates uniquely by taking into account the context of a token by looking at both the preceding and following tokens in a sentence. The library models come from the are pre-trained with the task of Masked Language Models (MLM) for predicting missing words from sentences and Next Sentence Prediction for evaluating if two string sets truly follow / precede each other or are a random fit. These functions boost BERT's evaluation of likewise paragraphs to be better than that found in token tallying models. For our usage, an

established strength in identifying text that may belong together is important for identifying an unseen text to an author's story and writing prose.

When implementing the model for the six books. The Hugging Face 'Transformers' library is implemented to use the BERT pre-trained models (unsupervised) and associated tokenizer. From the base unsupervised model, supervised training with our labeled data partitions builds upon the starting literary processing weights. These weights are then iteratively shifted as the training data is processed through the model and reevaluated N number of times (defined as epochs). The model is trained with pytorch on the CPU requiring the training, validation, and test data be converted into tensors once tokenized:

```
1  ## convert lists to tensors
2
3  train_seq = torch.tensor(tokens_train['input_ids'])
4  train_mask = torch.tensor(tokens_train['attention_mask'])
5  train_y = torch.tensor(train_labels.tolist())
6
7  val_seq = torch.tensor(tokens_val['input_ids'])
8  val_mask = torch.tensor(tokens_val['attention_mask'])
9  val_y = torch.tensor(val_labels.tolist())
10
11 test_seq = torch.tensor(tokens_test['input_ids'])
12 test_mask = torch.tensor(tokens_test['attention_mask'])
13 test_y = torch.tensor(test_labels.tolist())
```

Figure 5: BERT Model Tensors for Training

An important parameter in the training is the batch size, dictating the number of partitions to run while working through the entire training set. Smaller sizes allow for more diverse training due to the variety while larger sets help generalize the model's analysis for all training text at the cost of accuracy when given unseen data.

```
1  from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
2
3  #define a batch size
4  batch_size = 20
5
6  # wrap tensors
7  train_data = TensorDataset(train_seq, train_mask, train_y)
8
9  # sampler for sampling the data during training
10 train_sampler = RandomSampler(train_data)
11
12 # dataloader for train set
13 train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=batch_size)
14
15 # wrap tensors
16 val_data = TensorDataset(val_seq, val_mask, val_y)
17
18 # sampler for sampling the data during training
19 val_sampler = SequentialSampler(val_data)
20
21 # dataloader for validation set
22 val_dataloader = DataLoader(val_data, sampler=val_sampler, batch_size=batch_size)
```

Figure 6: Batch Size Definition and Data Set Handlers for Training

The other main training parameter tested initially were the number of epochs to train the model. An epoch equals a cycle of the data being fed forward in the model for prediction and evaluated against the true classes. In BERT each of these evaluations produces a training and validation dataset loss value to measure the iterative accuracy of the model on train data that affects the

weighting and validation data that gives a metric on performance for text that is not accounted for in the weight adjustment.

```
1 # set initial loss to infinite
2 best_valid_loss = float('inf')
3
4 #defining epochs
5 epochs = 1
6
7 # empty lists to store training and validation loss of each epoch
8 train_losses=[]
9 valid_losses=[]
10
11 #for each epoch
12 for epoch in range(epochs):
13
14     print('\n Epoch {:.} / {:.}'.format(epoch + 1, epochs))
15
16     #train model
17     train_loss, _ = train()
18
19     #evaluate model
20     valid_loss, _ = evaluate()
21
22     #save the best model
23     if valid_loss < best_valid_loss:
24         best_valid_loss = valid_loss
25         torch.save(model.state_dict(), 'saved_weights.pt')
26
27     # append training and validation loss
28     train_losses.append(train_loss)
29     valid_losses.append(valid_loss)
30
31     print(f'\nTraining Loss: {train_loss:.3f}')
32     print(f'\nValidation Loss: {valid_loss:.3f}')
```

Figure: BERT epoch Iterative Code

After all the epochs are ran the best tested weighting (classified through lowest value valid\_loss) is locked in with the file ‘saved\_weights.pt’ and loaded as the models basis for testing:

```
1 #load weights of best model
2 path = 'saved_weights.pt'
3 model.load_state_dict(torch.load(path))
✓ 0.6s
```

Figure 7: Winning epoch Weightings Set as BERT Dictionary

## 5. Evaluation and Analysis

### 5.1. SVM

SVM + BoW Model Accuracy Score: 0.95					
	precision	recall	f1-score	support	
Charles Dickens [f]	0.95	0.92	0.93	38	
Fyodor Dostoyevsky [e]	1.00	1.00	1.00	37	
George Eliot [c]	0.95	0.95	0.95	42	
Herman Melville [d]	0.97	0.94	0.96	35	
Leo Tolstoy [a]	0.95	0.95	0.95	42	
Victor Hugo [b]	0.90	0.93	0.91	46	
accuracy			0.95	240	
macro avg	0.95	0.95	0.95	240	
weighted avg	0.95	0.95	0.95	240	
SVM + TF-IDF Model Accuracy Score: 0.9708333333333333					
	precision	recall	f1-score	support	
Charles Dickens [f]	1.00	0.95	0.97	38	
Fyodor Dostoyevsky [e]	1.00	1.00	1.00	37	
George Eliot [c]	1.00	0.98	0.99	42	
Herman Melville [d]	0.97	0.97	0.97	35	
Leo Tolstoy [a]	0.98	0.95	0.96	42	
Victor Hugo [b]	0.90	0.98	0.94	46	
accuracy			0.97	240	
macro avg	0.97	0.97	0.97	240	
weighted avg	0.97	0.97	0.97	240	

Figure 8: SVM with BoW and TF-IDF

- The TF-IDF model outperforms the BoW model in terms of accuracy, suggesting the effectiveness of TF-IDF in capturing important features. (**SVM + TF-IDF Champion model**)
- SVM + BoW Model: **Fyodor Dostoyevsky [e]** achieves a perfect F1-score, showing excellent precision and recall.
- SVM + BoW Model: The lowest F1-score is observed for **Victor Hugo [b]**, suggesting potential challenges in classifying this author.
- SVM + TF-IDF Model: **Fyodor Dostoyevsky [e]** maintains a perfect F1-score, showcasing robust precision and recall.
- SVM + TF-IDF Model: **Victor Hugo [b]** continues to show challenges in classification, with a lower F1-score compared to other authors.
- SVM + TF-IDF Model: Charles Dickens [f] and George Eliot [c] also exhibit high precision and recall, contributing to the overall model success.

## 5.2.Naive Bayes

NB + BoW Model Accuracy Score: 0.9625					
		precision	recall	f1-score	support
Charles Dickens	[f]	0.95	0.92	0.93	38
Fyodor Dostoyevsky	[e]	1.00	1.00	1.00	37
George Eliot	[c]	0.95	0.95	0.95	42
Herman Melville	[d]	0.97	0.94	0.96	35
Leo Tolstoy	[a]	0.95	0.95	0.95	42
Victor Hugo	[b]	0.90	0.93	0.91	46
accuracy				0.95	240
macro avg		0.95	0.95	0.95	240
weighted avg		0.95	0.95	0.95	240
NB + TF-IDF Model Accuracy Score: 0.9458333333333333					
		precision	recall	f1-score	support
Charles Dickens	[f]	0.90	0.97	0.94	38
Fyodor Dostoyevsky	[e]	0.86	1.00	0.92	37
George Eliot	[c]	0.95	0.98	0.96	42
Herman Melville	[d]	1.00	0.94	0.97	35
Leo Tolstoy	[a]	0.98	0.95	0.96	42
Victor Hugo	[b]	1.00	0.85	0.92	46
accuracy				0.95	240
macro avg		0.95	0.95	0.95	240
weighted avg		0.95	0.95	0.95	240

Figure 9: Naive Bayes with BoW and TF-IDF

- NB+ BOW model slightly outperforms the NB+TF-IDF model in terms of accuracy.
- Fyodor Dostoyevsky has high precision and recall in Bow model.
- **Victor Hugo [b]** has lower precision in the NB + TF-IDF model, suggesting potential challenges in classifying this author.
- Consistent high F1-scores across most authors indicate a well-performing model.
- For authors with slightly lower F1-scores ( **Victor Hugo [b]**), further investigation into specific instances and potential improvements in feature engineering may enhance classification accuracy.

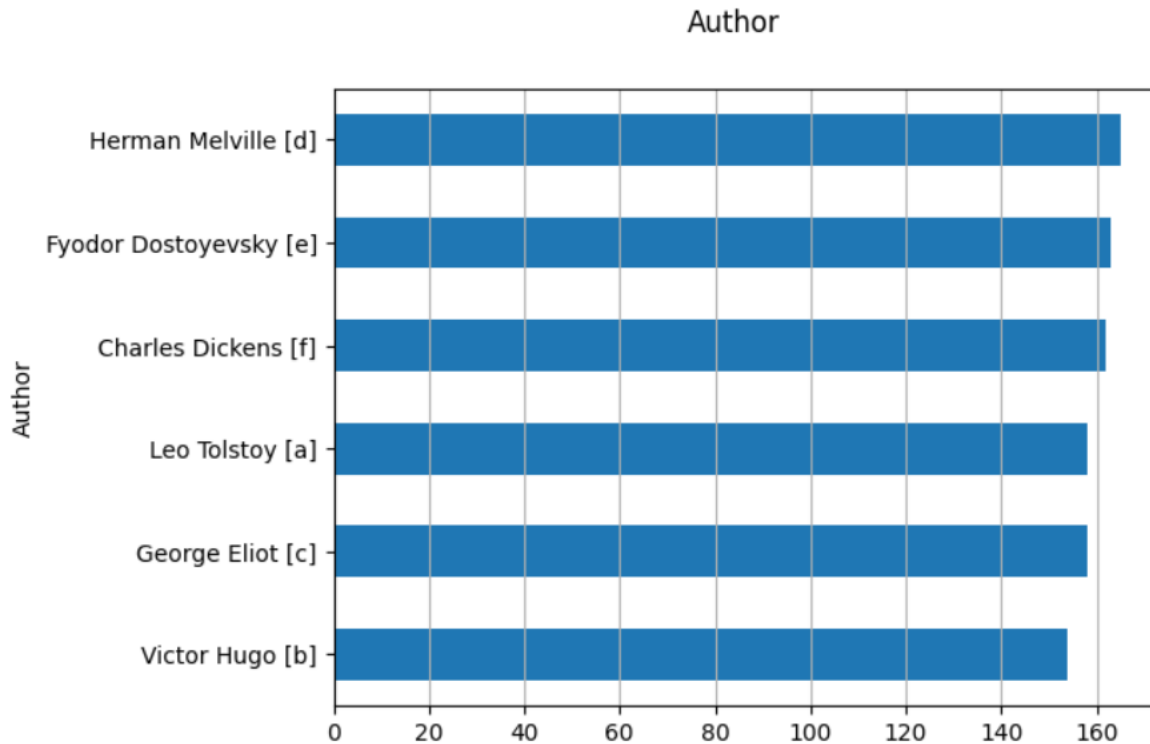


Figure 10: Partition counts by author

In our experiment, we have observed that the uneven distribution of books among different authors introduces class imbalance. Authors with more books can create a biased model favoring the majority class, impacting accuracy. Specifically, for Victor Hugo [b], who has fewer representative samples due to lower book frequency, this imbalance may contribute to a less favorable learning environment, potentially resulting in lower F1-scores. To address this issue, balancing the class distribution and augmenting data for under-represented authors, like Victor Hugo, could enhance overall model performance.

### 5.3 2-Gram Generations with Various Ensemble Methods

As shown in the figure below, the BoW model with bigrams enabled outperformed TF-IDF in terms of accuracy, suggesting that BoW with bigrams enabled captures features better from the bigram generated data. The high F-1 score on George Eliot [c] is indicative of the high precision and recall shown to the left of the score. A lower F1-score for Victor Hugo [b] is also consistent with the previous section. The lower number of partitions to train on in the training set also explains why Victor Hugo [b] was the worst performing author in terms of precision.

ET + BoW Model with Bigrams Enabled Accuracy Score: 0.9741666666666666				
	precision	recall	f1-score	support
Charles Dickens [f]	1.00	0.95	0.97	200
Fyodor Dostoyevsky [e]	0.99	0.97	0.98	200
George Eliot [c]	0.99	0.98	0.99	200
Herman Melville [d]	0.97	0.99	0.98	200
Leo Tolstoy [a]	0.98	0.96	0.97	200
Victor Hugo [b]	0.92	0.98	0.95	200
accuracy			0.97	1200
macro avg	0.98	0.97	0.97	1200
weighted avg	0.98	0.97	0.97	1200
RF + TF-IDF Model Accuracy Score: 0.9466666666666667				
	precision	recall	f1-score	support
Charles Dickens [f]	0.96	0.93	0.94	200
Fyodor Dostoyevsky [e]	0.99	0.90	0.94	200
George Eliot [c]	0.98	0.96	0.97	200
Herman Melville [d]	0.96	0.96	0.96	200
Leo Tolstoy [a]	0.97	0.96	0.96	200
Victor Hugo [b]	0.84	0.97	0.90	200
accuracy			0.95	1200
macro avg	0.95	0.95	0.95	1200
weighted avg	0.95	0.95	0.95	1200

Figure 11: Extra Trees Model Classification Report

## 5.4 BERT

Two sets of BERT training were conducted with the same number of epochs (10) for comparison but differing batch sizes during the model training (20 & 40 partitions). Another run with 1 epoch at batch size = 40 was done to benchmark the model's starting point. It should be noted that each epoch of the training for all cases took about 5-7 minutes to complete making this training method quite time consuming to get to the higher accuracy weights competitive with the other classification options.

For the 1 epoch with batch size equal to 40 case, the model started off quite inaccurate for 4 out of the 6 classes (refer to reference table for Class - Book/Author conversion):



```
#book_names = [
    #'War and Peace' Leo Tolstoy [0],
    #'Les Miserables' Victor Hugo [1],
    #'Middlemarch' George Eliot [2],
    #'Moby Dick' Herman Melville [3],
    #'The Brothers Karamazov' Fyodor Dostoyevsky [4],
    #'Bleak House' Charles Dickens [5]
#]
```

Figure 12: Conversion Table of Model Book.Author to Model Classes

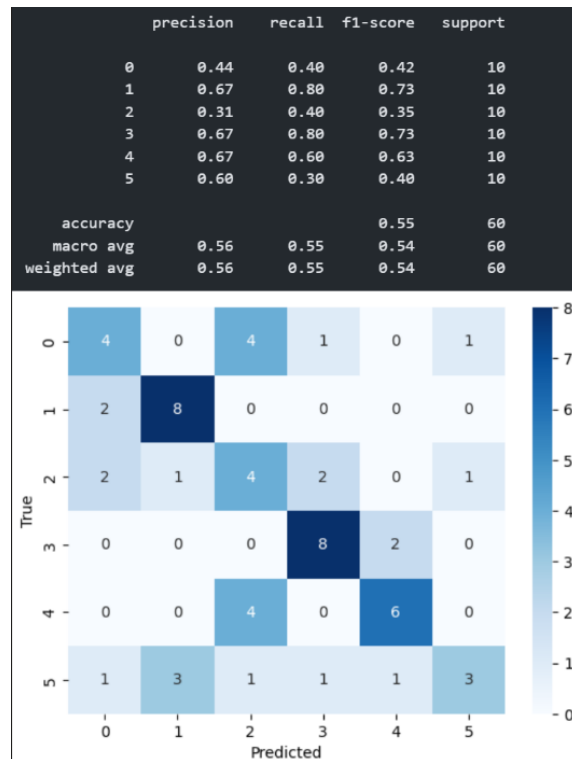


Figure 13: Classification Report and Confusion Matrix (BERT - batch\_size=40, epochs=1)

A wide spread of incorrect predictions is seen except for ‘Les miserables’ and ‘Moby Dick’. Association of this immediate accuracy can be made to the unique sub-genre labels seen for both texts, ‘tragedy’ and ‘nautical adventure’ respectively.

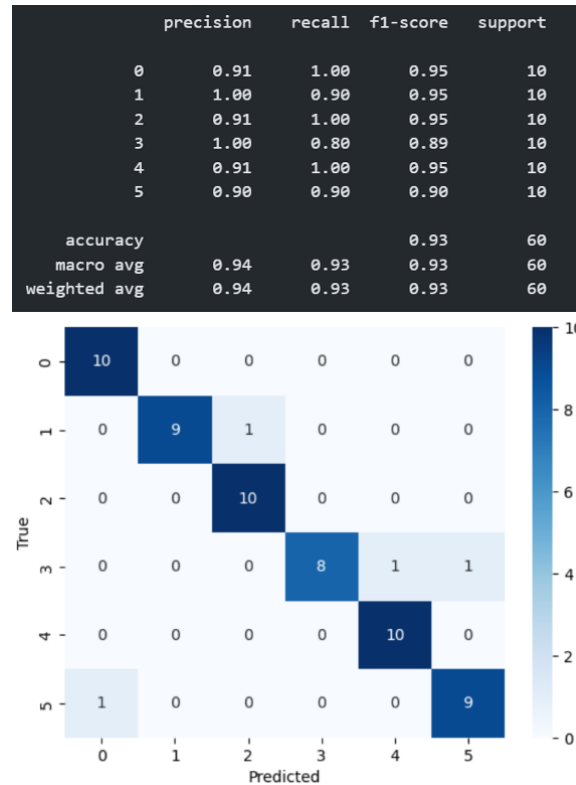


Figure 14: Classification Report and Confusion Matrix (BERT - batch\_size=40, epochs=10)

After 9 more epochs, the BERT model becomes much more accurate in distinguishing the books' texts from one another. With the model focused on shifting weights depending on masked words and Next Sentence Prediction, multiple iterations have improved the model's association of the different partitions to the same author. We can see here that the most confusion lies with Moby Dick now with the weighting shifts now giving more preference to the previously poorly classified authors.

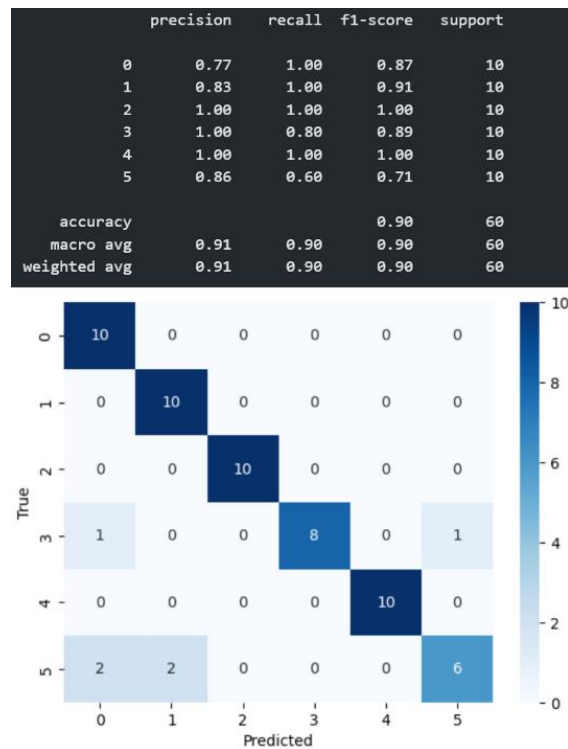


Figure 15: Classification Report and Confusion Matrix (BERT - batch\_size=20, epochs=10)

Testing smaller batch sizes validates the claims of specific classification providing higher precision at the cost of generalized accuracy. Having seen less in each iteration, gradient correction showed higher bias towards specific books in each iteration leading to 4 perfectly precise classes at the cost of the remaining classes being biased toward them.

## 6. Comparison

Observing all the models' metrics it was agreed that the IF-IDF + SVM model was the champion. It provided robust precision and overall accuracy across the 6 classified texts while still only requiring a lower number of epochs that ran through the iterations in a few seconds. The other fundamental transformations showed much higher epoch counts to match performance while the BERT model took excessive time per epoch (5 minutes) to still underperform SVM. Another issue with training BERT comes in the multi-sentence partitions used for training. The 'Next Sentence Prediction' base of the model will struggle to connect related partitions since they are taken at random from the corpus and may not concatenate sentences in sequence accurately.

## 7. Altering Champion Model

With the selected champion of the tested models, testing of parameters controlling the strong prediction algorithm are tested through alteration.

### 7.1 Sample Size

In our analysis, we experimented with various sample sizes and carefully evaluated their impact on the test results. The results were visually presented in a figure that depicted the accuracy scores corresponding to each sample size. After thorough analysis, we found that using a test sample that constituted 15% of the total data yielded the most favorable results with an accuracy score of 97.7%.

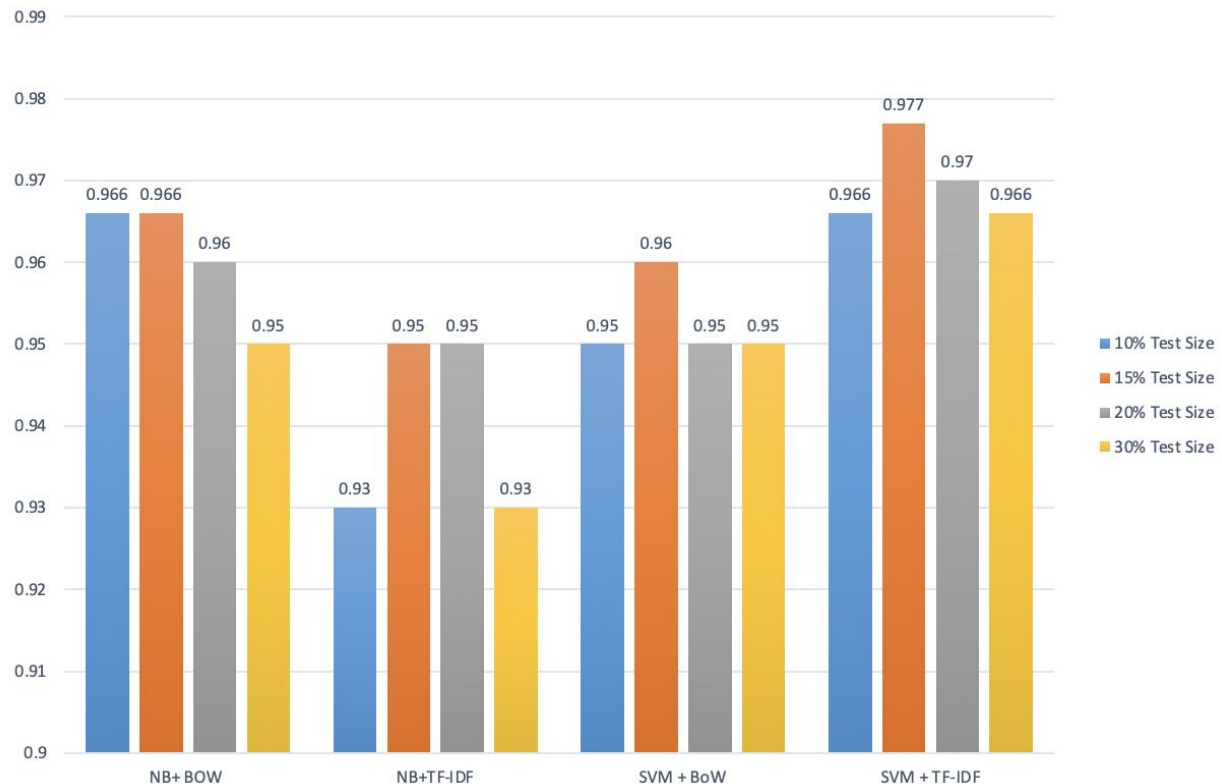


Figure 16 - Testing Different Sampling Size

## 7.2 Max Features

During the process of vectorizing textual data using TF-IDF (term frequency-inverse document frequency) and BoW (bag of words), we experimented with controlling the number of features by using the `max_features()` attribute. However, we observed that limiting the number of features had a negative impact on the accuracy of our model. As a result, we decided to not place any restrictions on the number of features and allowed the model to learn from the full set of features. This approach ultimately led to better performance and improved accuracy.

## 7.3 Max Iterations

The hyperparameter "max\_iter" is used in the SVM classifier of scikit-learn. It is short for "maximum number of iterations" and sets the maximum number of times the algorithm will pass over the training data (epochs). We conducted several tests and found that 900 iterations work best for the SVM+BoW model, while 100 iterations are optimal for the SVM+TF-IDF model.

## 7.4 Stemming and Lemmatization

We experimented with stemming and lemmatization techniques to observe their impact on our model's accuracy. Initially, our model employed stemming as a pre-processing step and it yielded an accuracy of 97.7%. However, upon trying lemmatization, we were able to achieve a better accuracy of 98.3%.

SVM + TF-IDF Model Accuracy Score: 0.9833333333333333					
		precision	recall	f1-score	support
Charles Dickens	[f]	1.00	0.96	0.98	28
Fyodor Dostoyevsky	[e]	1.00	0.96	0.98	28
George Eliot	[c]	1.00	0.97	0.98	32
Herman Melville	[d]	0.96	1.00	0.98	27
Leo Tolstoy	[a]	0.97	1.00	0.98	29
Victor Hugo	[b]	0.97	1.00	0.99	36
accuracy				0.98	180
macro avg		0.98	0.98	0.98	180
weighted avg		0.98	0.98	0.98	180

Figure 17 - SVM+TF-IDF Results Using Lemmatization in the Pre-processing

## 8. Conclusion

In summary, after applying various pre-processing techniques such as regular expressions and stemming, doing some feature engineering with the help of BoW, TF-IDF, and n-grams; several different models were applied. Overall, SVM with TF-IDF was the champion model, scoring 98.3% in terms of accuracy.

In Bow and TF-IDF we realized the unbalanced partitions put Victor Hugo in lower F1 score.

## References

*“Building Language Models: A Step-by-Step BERT Implementation Guide”*, Kajal Kumari, Oct. 26th 2023: <https://www.analyticsvidhya.com/blog/2023/06/step-by-step-bert-implementation-guide/>

*“Hugging Face Transformers Pipeline Functions | Advanced NLP”*, Deepak Moonat, Jan. 5th 2022: <https://www.analyticsvidhya.com/blog/2022/01/hugging-face-transformers-pipeline-functions-advanced-nlp/>

*“Term Frequency - Inverse Document Frequency”*, Fatih Karabiber, Jan, 4th 2023: [https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/#:~:text=Using%20scikit%2Dlearn-.What%20is%20TF%2DIDF%3F,%2C%20relative%20to%20a%20corpus\).](https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/#:~:text=Using%20scikit%2Dlearn-.What%20is%20TF%2DIDF%3F,%2C%20relative%20to%20a%20corpus).)