ASSIGNMENT 1: TEXT CLASSIFICATION

GNG5125 Data Science Applications

(Spring-Summer 2022)

Abstract

Classification refers to a predictive modeling problem wherein a set of input data is categorized into classes. The focus of this assignment is to implement a supervised learning task capable of classifying the author, given a set of texts belonging to the same genre.

Yinruo Jiang (300274815)
Rasheeq Mohammad (6849734)
Shahin Mahmud (300274789)
Group 12

Contents

1. **Objectives** 2
2. **Preparing the IDE** 2
3. **Sample Selection and Preprocessing** 3
4. **Data Preparation** 9
5. **Feature Engineering** 12

Bag-of-Words (BoW) 13

TF-IDF 13

6. **Evaluation** 14

K-Fold Cross Validation 14

7. **Implementation of Different ML Models for The Classification Task** 14

Support Vector Machine Classifier 14

Decision Tree Classifier 15

K-Nearest Neighbours Classifier 16

Random Forest Classifier 17

Naive Bayes Classifier 18

Scoring Different ML Models 20

8. **Display Results of Evaluation** 21

Champion Model 21

9. **Testing Documents without Labels** 21
10. **Test and Train** 22
11. **Error Analysis** 22

Naive Bayes 22

Support Vector Machine 24

12. **Conclusion** 26

1. Objectives

Selection of Samples: Take five different samples of Gutenberg digital books, which are of five different authors and are of the same genres as well as are semantically the same.

Preprocessing: Read the texts and remove stop-words and garbage characters if needed.

Preparation of Data:

- create random samples of 200 documents of each book, representative of the source input.
- Prepare the records of 100 words records for each document,
- label them as a, b and c etc., as per the book they belong to.

Feature Engineering: Transform the preprocessed data to BOW, and TF-IDF, n-gram, (LDA, word-embedding, optional) etc.

Train: Train a machine that can tell which author (or genre) when asked.

Evaluation: K-fold Cross-validation is for the evaluation step to compare and select the champion model. This includes various types of transformation and the classification algorithms plus other parameters that will lead us to the final result.

Perform Error-Analysis: Identification of what were the characteristics of the instance records that threw the machine off. We will use the champion predictions against test data to do error analysis.

2. Preparing the IDE

In this step, all the necessary libraries and packages were imported and downloaded to prepare our python environment.

```
from collections import Counter

# pip install contractions
# Used for expanding contractions
import contractions

import matplotlib.pyplot as plt

import nltk
from nltk import word_tokenize
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.probability import FreqDist
from nltk.stem.wordnet import WordNetLemmatizer

import pandas as pd

import random

import re
```

```
import seaborn as sns

from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransfor
mer
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import cross_val_score, ShuffleSplit, train_t
est_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier

# Used for removing accented characters
import unicodedata

# pip install wordcloud
from wordcloud import WordCloud

nltk.download("punkt")
nltk.download("stopwords")
nltk.download("wordnet")
```

3. Sample Selection and Preprocessing

Five different books were selected based on the authors with the aim of having distinct targets for the model. NLTK includes a small selection of texts from the Project Gutenberg electronic text archive, which contains 25,000 free digital books.

Our five digital books, selected from Gutenberg's project are given below:

	Title	Author	Year of Publication	URL
1	Star Born	Andre Norton	1957	https://www.gutenberg.org/files/18458/18458.txt
2	The Goddess of Atvatabar	William R. Bradshaw	1892	https://www.gutenberg.org/files/32825/32825.txt
3	Twenty Thousand Leagues Under the Sea (slightly abridged)	Jules Verne	1872	https://www.gutenberg.org/files/164/164.txt

4	A Princess of Mars	Edgar Rice Burroughs	1912	https://www.gutenberg.org/files/62/62.txt
5	The Lost World	Arthur Conan Doyle	1912	https://www.gutenberg.org/files/139/139.txt

In this stage, the five digital books were accessed from their respective URLs. The raw texts were taken from the books as well as the title and author information were extracted. Afterwards, text preprocessing, text tokenization and text cleaning were performed on the raw texts from the books. Various functions were also declared to find the start and end of the book, to normalize accented characters and to expand contractions.

The text cleaning process includes:

- Removing unwanted characters and stop-words had been done as these would corrupt the training model.
- The sentences were further tokenized to words and lemmatization/ stemming took place which is the process by which all the words were reduced to the root words by using the morphological analysis of words. It helps in reducing the size of the data and also makes it convenient to keep track of the specific words instead of the same words used in different forms.
- Since there was a requirement for only words, there was a need to remove all data that do not count as words e.g. numbers, punctuations etc. It reduces the size of the data and makes the process faster.
- All the words are also transformed into the lower case so that the program doesn't recognise same words in different cases as unlike.

```
def read_utf8_file(file):

    bom = "\ufeff"
    text = ""
    with open(file, mode="r", encoding="utf-8") as f:
        text = f.read()
        if text.startswith(bom):
            text = text[1:]
    return text

def extract_title(text):
    pattern = "Title: (.*?)\n"
    match = re.search(pattern, text)
    if match is None:
        return ""
    else:
        return match.group(1).strip()

def extract_author(text):
    pattern = "Author: (.*?)\n"
```

```
    match = re.search(pattern, text)
    if match is None:
        return ""
    else:
        return match.group(1).strip()

def find_start_of_book(text):
    pattern = "\*\*\* START OF (.*?) \*\*\*\n"
    match = re.search(pattern, text)
    if match is None:
        return -1
    else:
        return text.find(match.group(0))

def find_end_of_book(text):
    pattern = "\*\*\* END OF (.*?) \*\*\*\n"
    match = re.search(pattern, text)
    if match is None:
        return -1
    else:
        return match.end()

def normalize_accented_chars(text):
    text = unicodedata.normalize("NFKD", text).encode("ascii", "ignore").d
ecode("utf-8", "ignore")
    return text

def expand_contractions(text):
    text = contractions.fix(text)
    return text

def preprocess_text(text, nor_acc:True, exp_con=True):
    if nor_acc:
        text = normalize_accented_chars(text)
    if exp_con:
        text = expand_contractions(text)
    return text

def tokenize_into_words(text):
    text = word_tokenize(text)
    return text

def remove_punctuation(text):
    text = [w for w in text if w.isalpha()]
    return text
```

```
def convert_to_lower_case(text):
    text = [w.lower() for w in text]
    return text

def remove_stop_words(text):
    stop_words = stopwords.words("english")
    text = [w for w in text if not w in stop_words]
    return text

def perform_stemming(text):
    stemmer = PorterStemmer()
    text = [stemmer.stem(w) for w in text]
    return text

def perform_lemmatization(text):
    lemmatizer = WordNetLemmatizer()
    text = [lemmatizer.lemmatize(w) for w in text]
    return text

def clean_text(text, rem_pun:True, con_low=True, rem_sto=True, per_ste=Fal
se, per_lem=True):
    if rem_pun:
        text = remove_punctuation(text)
    if con_low:
        text = convert_to_lower_case(text)
    if rem_sto:
        text = remove_stop_words(text)
    if per_ste:
        text = perform_stemming(text)
    if per_lem:
        text = perform_lemmatization(text)
    return text

def find_most_common_words(text_df, authors):
    author_to_most_common_words = {}
    index = 0
    for r in text_df.groupby(["Author"])["Cleaned Samples"].apply(lambda x
: " ".join(x)):
        words_to_count = (w for w in r.split(" "))
        words_to_count = [w for w in words_to_count if w]
        c = Counter(words_to_count).most_common(20)
        c = list(zip(*c))
        c = c[0]
        author_to_most_common_words[authors[index]] = c
```

```
        index += 1
    return author_to_most_common_words

def draw_word_cloud(words, titles):
    plt.rcParams["figure.figsize"] = [24, 6]
    index = 1
    i = 0
    for key, value in words.items():
        wc = WordCloud().generate((" ").join(value))
        plt.subplot(2, 3, index)
        index += 1
        plt.imshow(wc, interpolation="bilinear")
        plt.axis("off")
        plt.title(titles[i] + " by " + key)
        i += 1

def remove_most_common_words(row ,words):
    new_text_df = [word for word in row["Cleaned Samples"].split() if word
 not in words[row["Author"]]]
    text = " ".join(new_text_df)
    return text

def clean_text_advanced(text_df, authors, titles, rem_com=True):
    if rem_com is True:
        new_stop_words = find_most_common_words(text_df, authors)
        draw_word_cloud(new_stop_words, titles)
        text_df["Advanced Cleaned Samples"] = text_df.apply(lambda x: remo
ve_most_common_words(x, new_stop_words), axis=1)
```

Applying the defined functions and performing respective tasks accordingly:

```
books = ["pg18458.txt", "pg32825.txt", "pg164.txt", "pg62.txt", "pg139.txt
"]

authors = []
titles = []

raw_texts = [] # Read text from file
simplified_texts = [] # Remove copyright and license information
preprocessed_texts = [] # Normalize accented characters and expand contrac
tions
tokenized_texts = [] # Tokenize text into words
cleaned_texts = [] # Clean text (e.g. remove punctuation, convert to lower
 case, remove stop words, peform stemming, peform lemmatization, etc.)
sampled_texts = [] # Sample text
```

```
labeled_texts = []

text_info = []

for b in books:
    raw_text = read_utf8_file(b)
    raw_texts.append(raw_text)
    title = extract_title(raw_text)
    titles.append(title)
    author = extract_author(raw_text)
    authors.append(author)

    start = find_start_of_book(raw_text)
    end = find_end_of_book(raw_text)
    simplified_text = raw_text[start:end]
    simplified_texts.append(simplified_text)

    preprocessed_text = preprocess_text(simplified_text, nor_acc=False, ex
p_con=True)
    preprocessed_texts.append(preprocessed_text)

    tokenized_text = tokenize_into_words(preprocessed_text)
    tokenized_texts.append(tokenized_text)

    cleaned_text = clean_text(tokenized_text, rem_pun=True, con_low=True,
rem_sto=True, per_ste=False, per_lem=True)
    cleaned_texts.append(cleaned_text)

    sampled_text = performing_sampling(cleaned_text)
    sampled_texts.append(sampled_text)

    print(f"Collected, prepared, and cleaned {title} written by {author} (
{b})")

for a, t, rt, sit, pt, tt, ct, sat in zip(authors, titles, raw_texts, simp
lified_texts, tokenized_texts, preprocessed_texts, cleaned_texts, sampled_
texts):
    text_info.append((a, t, len(rt), len(sit), len(tt), len(pt), len(ct)))
    labeled_texts += ([(txt, a) for txt in sat])
```

4. Data Preparation

Another function (performing_sampling) was defined which is responsible for breaking down a book into partitions or documents of 100 words each, then selecting 200 random partitions or documents. This function was applied to the preprocessed texts.

```
def performing_sampling(text, num_samples=200, num_words_per_sample=100):
    data = []
    string = ""
    count = 0
    for w in text:
        if count < num_words_per_sample:
            string += w
            if count == (num_words_per_sample - 1):
                pass
            else:
                string += " "
            count += 1
        else:
            data.append(string)
            string = ""
            count = 0
    random.seed(0)
    return random.sample(data, num_samples)
```

Frequency distribution of words in tokenized texts and cleaned texts had been shown and also visualized using the 'plot' function.

```
# Frequency distribution of words in cleaned texts
plt.rcParams["figure.figsize"] = [25, 8]
index = 1
for t, a, ct in zip(titles, authors, cleaned_texts):
    plt.subplot(2, 3, index)
    index += 1
    freq_dist = FreqDist(ct)
    title = f"{t} by {a}"
    freq_dist.plot(30, cumulative=False, title=title)
    plt.show()
```

A data frame had been shown with labeled columns with samples and authors.

```
labeled_texts_df = pd.DataFrame(labeled_texts)

labeled_texts_df = labeled_texts_df.rename(columns={0: "Cleaned Samples",
                                                    1: "Author"})
labeled_texts_df
```

	Cleaned Samples	Author
0	delicately ready flee first hint suspected bel...	Andre Norton
1	explain could one make plain feeling sensible ...	Andre Norton
2	stubbornly gray murmur wonstead went drone ach...	Andre Norton
3	seeming unconcern sssuri first intimation hunt...	Andre Norton
4	raf first reaction must still merman young str...	Andre Norton
...	...	...
995	must difficult one otherwise creature would co...	Arthur Conan Doyle
996	face flashed back went south america solitary ...	Arthur Conan Doyle
997	page disappointing however contained nothing p...	Arthur Conan Doyle
998	one indian group dragged forward edge cliff ki...	Arthur Conan Doyle
999	day sat late mcardle news editor explaining wh...	Arthur Conan Doyle

1000 rows × 2 columns

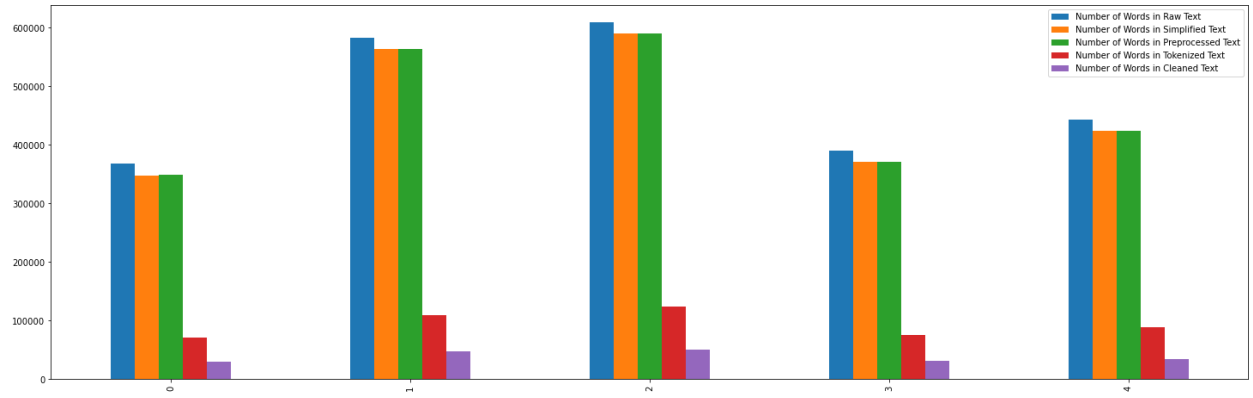
Another modified data frame was implemented with labelled columns as well as the information of titles, authors, number of words in raw text, simplified text, preprocessed text, tokenized text and cleaned text.

```
text_info_df = pd.DataFrame(text_info)

text_info_df = text_info_df.rename(columns={0: "Authors",
                                            1: "Titles",
                                            2: "Number of Words in Raw Tex
t",
                                            3: "Number of Words in Simplif
ied Text",
                                            4: "Number of Words in Preproc
essed Text",
                                            5: "Number of Words in Tokeniz
ed Text",
                                            6: "Number of Words in Cleaned
 Text"})
text_info_df
```

	Authors	Titles	Number of Words in Raw Text	Number of Words in Simplified Text	Number of Words in Preprocessed Text	Number of Words in Tokenized Text	Number of Words in Cleaned Text
0	Andre Norton	Star Born	367551	348402	348660	71828	29333
1	William R. Bradshaw	The Goddess of Atvatabar	582554	563280	563659	109510	47696
2	Jules Verne	Twenty Thousand Leagues under the Sea (slightl...	609064	589919	590361	123803	50111
3	Edgar Rice Burroughs	A Princess of Mars	390249	371188	371227	75137	31870
4	Arthur Conan Doyle	The Lost World	443413	424311	424852	89354	34862

We visualized the number of words in all kinds of texts (from raw text to cleaned text) using the 'plot' function.

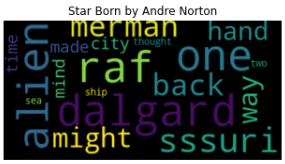
```
text_info_df.plot.bar()
```

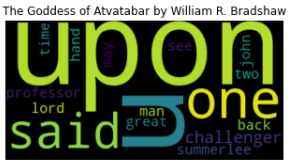


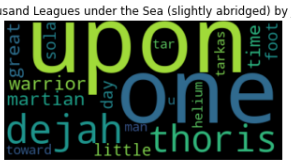
Moreover, advanced cleaning had been performed. At first, we generated word clouds of the most common words and then removed the said words.

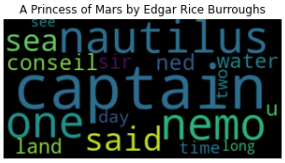
```
#  Will evaluate impact of advanced cleaning in later sections
```

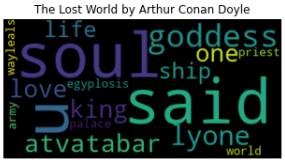
```
clean_text_advanced(labeled_texts_df, authors, titles, rem_com=True)
```











5. Feature Engineering

Creating n-gram models for both cleaned and advanced clean samples to conduct feature engineering:

```
types = ["Cleaned Samples", "Advanced Cleaned Samples"]

for ty in types:
    fig, ax = plt.subplots(nrows=3, ncols=2, figsize=(25, 15))
    fig.suptitle("Most Frequent Bigrams in " + ty, fontsize=15)
    row = 0
    column = 0
    for a, t in zip(authors, titles):
        corpus = labeled_texts_df[labeled_texts_df["Author"] == a][ty]
        tokens = word_tokenize(corpus.str.cat(sep=" "))
        freq_dist = FreqDist(nltk.ngrams(tokens, 2))
        dtf_bi = pd.DataFrame(freq_dist.most_common(5), columns=["Bigrams"
, "Frequency"])
        dtf_bi["Bigrams"] = dtf_bi["Bigrams"].apply(lambda x: " ".join(str
ing for string in x) )
        dtf_bi.set_index("Bigrams").iloc[:].sort_values(by="Frequency").pl
ot(kind="barh", ax=ax[row][column], width=0.5, alpha=0.5)
        title = t + " by " + a
        ax[row][column].set(title=title)
        if column < 1:
            column += 1
        else:
            column = 0
            row += 1
    ax[2][1].set_visible(False)
```

Feature engineering is an approach where all the textual data are converted into numeric form as the Machine Learning Algorithms work only on numeric data. Since we only have textual data available, the numeric features are extracted by using a number of different techniques, which are Bag-of-Words (BOW) and Term Frequency-Inverse Document Frequency (TF-IDF). These are discussed below.

Bag-of-Words (BoW)

The Bag-of-Words preserves the words present in the corpus. The words are referred to as a feature for the documents. The frequency of each word is calculated in the current document; in this way, the word features are engineered or extracted from the textual corpus.

This was done using the 'CountVectorizer' function for both cleaned and advanced cleaned samples.

TF-IDF

The Term Frequency-Inverse Document Frequency is the most popular method to convert the textual data into numerical features. This method helps us to highlight words which are interesting and unique in the particular document but not across all the documents. It also assigns a particular integer number to each of these words.

The Term Frequency is:

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

The Inverse Document Frequency is:

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

This was done using the 'TfidfVectorizer' function for both cleaned and advanced cleaned samples.

```
bow_cln_tr = CountVectorizer().fit(labeled_texts_df["Cleaned Samples"])

bow_cln_docs = bow_cln_tr.transform(labeled_texts_df["Cleaned Samples"])

bow_adv_cln_tr = CountVectorizer().fit(labeled_texts_df["Advanced Cleaned
Samples"])
adv_cln_docs = bow_adv_cln_tr.transform(labeled_texts_df["Advanced Cleaned
 Samples"])

tfidf_cln_tr = TfidfTransformer().fit(bow_cln_docs)
tfidf_adv_cln_docs = tfidf_cln_tr.transform(bow_cln_docs)

tfidf_adv_cln_tr = TfidfTransformer().fit(adv_cln_docs)
documents_tfidf_2 = tfidf_adv_cln_tr.transform(adv_cln_docs)

clf_and_samp_type_to_avg_acc = {}

n_splits = 10
test_size = 0.1
random_state = 0
```

6. Evaluation

K-Fold Cross-Validation

The cross-validation is one of the most important parts as it is essential to validate the stability of the machine learning models, and it generalizes the new data. The split of the data needs to be done carefully. If the amount of train data is less in percentage than the amount of test data, the important patterns in the data set may be at risk, which in turn increases the error in the process. Hence, a large portion of data is required for the train data set where the K fold cross-validation technique can serve the purpose.

In this process, the data is divided into k subsets, and hence the same process is repeated k times. Every time one of the k subsets is used as the test set, the other k-1 subsets are combined to form a train data set.

The algorithm for the cross-validation is given below

- Randomly split the whole data set into k fold.
- For each k fold in the data set, we build a model on k-1 fold of the dataset. Then it tests the model to check the effectiveness for kth fold.
- Check the errors for the predictions
- Repeat the whole process until each of the k folds has served as the test set.
- The average of the k recorded errors is called cross-validation error, and it will serve as the performance metric for the model

In our code, the K-Fold is a function which consists of parameters for splitting the data number of times, shuffling the data, and performing the 'random_state'. As there's a need to perform ten-fold cross-validation, we need to run the code ten times in a 'for loop', by specifying the input words from the Bag-of-Words transformation or the TF-IDF transformation. After that, we simultaneously implement the ten-fold using various ML models for the train data set and test data set.

7. Implementation of Different ML Models for The Classification Task

Support Vector Machine Classifier

The Support Vector Machine is a linear model which can be used for the classification of data. It is also used to solve the linear and non-linear problems. The algorithm of SVM creates a line that separates and distinctly classifies the data points. Support vector machine is highly preferred by many as it produces significant accuracy with less computation power. This algorithm is memory efficient, and It is s also used when the number of dimensions is greater than the number of samples.

The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points. To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

```
#
# Support Vector Machine (SVM)
#

# SVM + Cleaned Samples
```

```
pipeline = Pipeline([("bow", bow_cln_tr),
                     ("tfidf", tfidf_cln_tr),
                     ("clf", SGDClassifier(loss="hinge", penalty="l2", alp
ha=1e-3, random_state=random_state, max_iter=5, tol=None))])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=0)
scores = cross_val_score(pipeline, labeled_texts_df["Cleaned Samples"], la
beled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["SVM + Cleaned Samples"] = scores.mean()

# SVM + Advanced Cleaned Samples
pipeline = Pipeline([("bow", bow_adv_cln_tr),
                     ("tfidf", tfidf_adv_cln_tr),
                     ("clf", SGDClassifier(loss="hinge", penalty="l2", alp
ha=1e-3, random_state=random_state, max_iter=5, tol=None))])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=0)
scores = cross_val_score(pipeline, labeled_texts_df["Advanced Cleaned Samp
les"], labeled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["SVM + Advanced Cleaned Samples"] = scores.me
an()
```

Decision Tree Classifier

Decision Tree is a Supervised Machine Learning Algorithm that uses a set of rules to make decisions, similar to how humans make decisions. The intuition behind Decision Trees is that we use the dataset features to create yes/no questions and continually split the dataset until we isolate all data points belonging to each class. With this process, you're organizing the data in a tree structure. Every time you ask a question, you're adding a node to the tree. And the first node is called the root node.

The result of asking a question splits the dataset based on the value of a feature and creates new nodes.

This algorithm is used for non-parametric effective machine learning modelling techniques for classification and regression problems. To get a solution, the decision tree makes a hierarchical and sequential decision about the variables based on the predictor data. It is resistant to outliers, so it requires less data processing.

```
#
# Decision Tree (DT)
#

# DT + Cleaned Samples
pipeline = Pipeline([("bow", bow_cln_tr),
                     ("tfidf", tfidf_cln_tr),
                     ("clf", DecisionTreeClassifier(random_state=0))])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
```

```
scores = cross_val_score(pipeline, labeled_texts_df["Cleaned Samples"], la
beled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["DT + Cleaned Samples"] = scores.mean()

# DT + Advanced Cleaned Samples
pipeline = Pipeline([("bow", bow_adv_cln_tr),
                     ("tfidf", tfidf_adv_cln_tr),
                     ("clf", DecisionTreeClassifier(random_state=0))])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
scores = cross_val_score(pipeline, labeled_texts_df["Advanced Cleaned Samp
les"], labeled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["DT + Advanced Cleaned Samples"] = scores.mea
n()
```

K-Nearest Neighbours Classifier

The k-nearest neighbours (KNN) algorithm is a simple, supervised machine learning algorithm. The algorithm finds the distances between the query specified and the examples In the data. It selects the specified number of examples(K) closest to the query and then votes for the most frequent label.

The KNN Algorithm:

- Load the data
- Initialize K to our chosen number of neighbours
- For each example in the data:
  - ✓ Calculate the distance between the query example and the current example from the data.
  - ✓ Add the distance and the index of the example to an ordered collection
- Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances
- Pick the first K entries from the sorted collection
- Get the labels of the selected K entries
- If regression, return the mean of the K labels
- If classification, return the mode of the K labels

```
#
# KNeighbors (KN)
#

# KN + Cleaned Samples
pipeline = Pipeline([("bow", bow_cln_tr),
                     ("tfidf", tfidf_cln_tr),
                     ("clf", KNeighborsClassifier(n_neighbors=3))])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
```

```
scores = cross_val_score(pipeline, labeled_texts_df["Cleaned Samples"], la
beled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["KN + Cleaned Samples"] = scores.mean()

# KN + Advanced Cleaned Samples
pipeline = Pipeline([("bow", bow_adv_cln_tr),
                     ("tfidf", tfidf_adv_cln_tr),
                     ("clf", KNeighborsClassifier(n_neighbors=3))])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
scores = cross_val_score(pipeline, labeled_texts_df["Advanced Cleaned Samp
les"], labeled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["KN + Advanced Cleaned Samples"] = scores.mea
n()
```

Random Forest Classifier

Random forest, as its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction, and the class with the most votes becomes our model's prediction.

The fundamental concept behind random forest is a simple but powerful one — the wisdom of crowds. In data science-speak, the reason that the random forest model works so well is:

A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.

This algorithm was used in our program because since it includes the randomness feature and bagging, which is beneficial while creating an individual tree to make an uncorrelated forest of tree. This in turn, increases the accuracy compared to an individual.

```
#
# Random Forest (RF)
#

# RF + Cleaned Samples
pipeline = Pipeline([("bow", bow_cln_tr),
                     ("tfidf", tfidf_cln_tr),
                     ("clf", RandomForestClassifier())])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
scores = cross_val_score(pipeline, labeled_texts_df["Cleaned Samples"], la
beled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["RF + Cleaned Samples"] = scores.mean()

# RF + Advanced Cleaned Samples
pipeline = Pipeline([("bow", bow_adv_cln_tr),
```

```
                    ("tfidf", tfidf_adv_cln_tr),
                    ("clf", RandomForestClassifier())])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
scores = cross_val_score(pipeline, labeled_texts_df["Advanced Cleaned Samp
les"], labeled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["RF + Advanced Cleaned Samples"] = scores.mea
n()
```

Naive Bayes Classifier

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem. This section will focus on an intuitive explanation of how naive Bayes classifiers work, followed by a couple of examples of them in action on some datasets.

Naive Bayes classifiers are built on Bayesian classification methods. These rely on Bayes's theorem, which is an equation describing the relationship of conditional probabilities of statistical quantities. In Bayesian classification, we're interested in finding the probability of a label given some observed features, which we can write as P(L | features). Bayes's theorem tells us how to express this in terms of quantities we can compute more directly:

$$P(L \mid \text{features}) = P(\text{features} \mid L) P(L) / P(\text{features})$$

If we are trying to decide between two labels—let's call them L1L1 and L2L2—then one way to make this decision is to compute the ratio of the posterior probabilities for each label:

$$(P(L1 \mid \text{features})) / (P(L2 \mid \text{features})) = (P(\text{features} \mid L1)\, P(L1)) / (P(\text{features} \mid L2)\, P(L2))$$

All we need now is some model by which we can compute P(features | Li)P(features | Li) for each label. Such a model is called a *generative model* because it specifies the hypothetical random process that generates the data. Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier.

```
#
# Naive Bayes (NB)
#

# NB + Cleaned Samples
pipeline = Pipeline([("bow", bow_cln_tr),
                    ("tfidf", tfidf_cln_tr),
                    ("clf", MultinomialNB())])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
scores = cross_val_score(pipeline, labeled_texts_df["Cleaned Samples"], la
beled_texts_df["Author"], cv=cv)
```

```
clf_and_samp_type_to_avg_acc["NB + Cleaned Samples"] = scores.mean()

# NB + Advanced Cleaned Samples
pipeline = Pipeline([("bow", bow_adv_cln_tr),
                     ("tfidf", tfidf_adv_cln_tr),
                     ("clf", MultinomialNB())])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
scores = cross_val_score(pipeline, labeled_texts_df["Advanced Cleaned Samp
les"], labeled_texts_df["Author"], cv=cv)
clf_and_samp_type_to_avg_acc["NB + Advanced Cleaned Samples"] = scores.mea
n()
```

The average accuracy of all classifier models with both cleaned and advanced cleaned samples:

```
pd.DataFrame(clf_and_samp_type_to_avg_acc.items(), columns=["Classifier +
Sample Type", "Average Accuracy"])
```

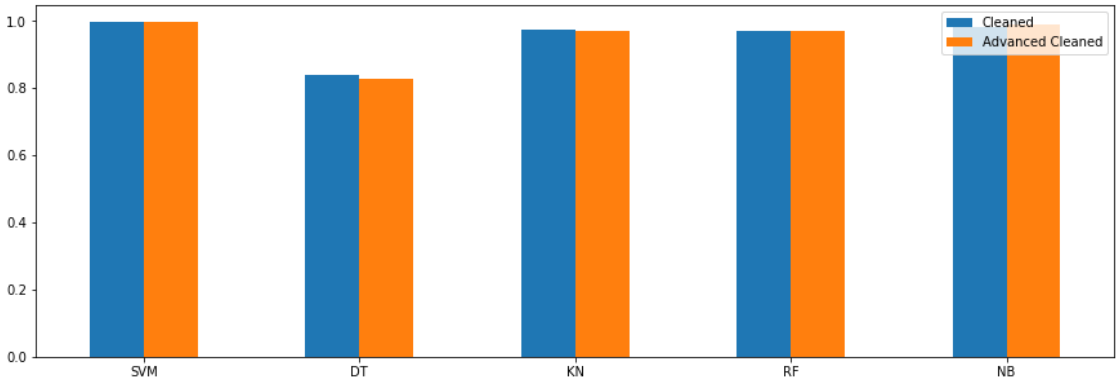
	Classifier + Sample Type	Average Accuracy
0	SVM + Cleaned Samples	0.997
1	SVM + Advanced Cleaned Samples	0.998
2	DT + Cleaned Samples	0.838
3	DT + Advanced Cleaned Samples	0.828
4	KN + Cleaned Samples	0.974
5	KN + Advanced Cleaned Samples	0.971
6	RF + Cleaned Samples	0.971
7	RF + Advanced Cleaned Samples	0.969
8	NB + Cleaned Samples	0.983
9	NB + Advanced Cleaned Samples	0.989

Scoring Different ML Models

```
cln_acc = [clf_and_samp_type_to_avg_acc["SVM + Cleaned Samples"],
           clf_and_samp_type_to_avg_acc["DT + Cleaned Samples"],
           clf_and_samp_type_to_avg_acc["KN + Cleaned Samples"],
           clf_and_samp_type_to_avg_acc["RF + Cleaned Samples"],
           clf_and_samp_type_to_avg_acc["NB + Cleaned Samples"]]
adv_cln_acc = [clf_and_samp_type_to_avg_acc["SVM + Advanced Cleaned Sample
s"],
               clf_and_samp_type_to_avg_acc["DT + Advanced Cleaned Samples
"],
               clf_and_samp_type_to_avg_acc["KN + Advanced Cleaned Samples
"],
               clf_and_samp_type_to_avg_acc["RF + Advanced Cleaned Samples
"],
               clf_and_samp_type_to_avg_acc["NB + Advanced Cleaned Samples
"]]
index = ["SVM", "DT", "KN", "RF", "NB"]
cross_validation_scores_df = pd.DataFrame({"Cleaned": cln_acc, "Advanced C
leaned": adv_cln_acc}, index = index)
cross_validation_scores_df.plot.bar(figsize=[15, 5], rot=0)
cross_validation_scores_df
```

The below table shows the accuracies for the SVM, DT, KNN, RF and NB Classifiers based on the data, transformed by Bag-of-Words and TF-IDF techniques.

	Cleaned	Advanced Cleaned
SVM	0.997	0.998
DT	0.838	0.828
KN	0.974	0.971
RF	0.971	0.969
NB	0.983	0.989

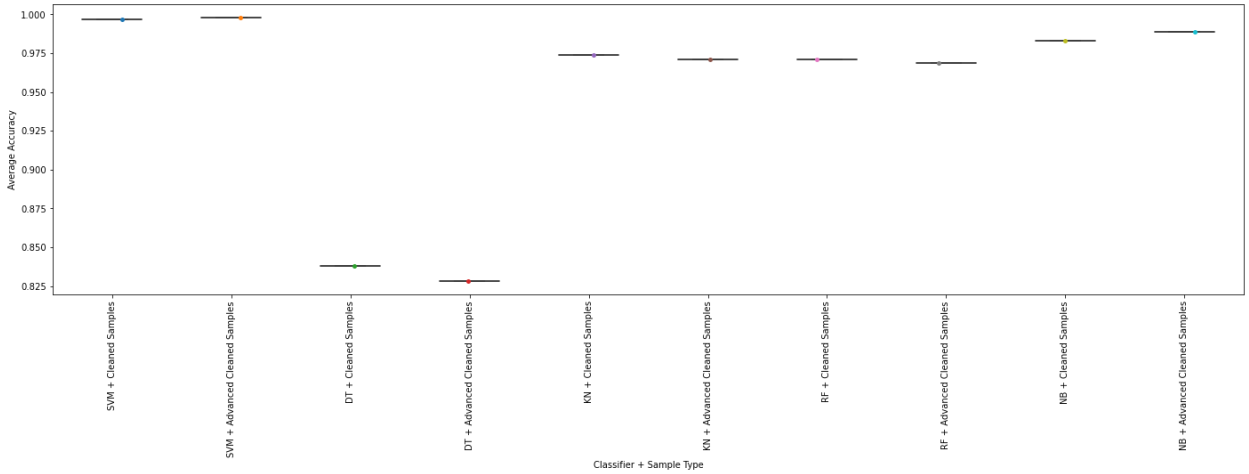


8. Display Results of Evaluation

```
bow_scores_df = pd.DataFrame(clf_and_samp_type_to_avg_acc.items(), columns
=["Classifier + Sample Type", "Average Accuracy"])
sns.boxplot(x="Classifier + Sample Type",
            y="Average Accuracy",
            data=bow_scores_df,
            width=0.5,
            fliersize=10)
sns.stripplot(x="Classifier + Sample Type",
              y="Average Accuracy",
              data=bow_scores_df,
              size=5,
              jitter=True)
plt.xticks(rotation=90)
plt.show()
```

Champion Model

The above plot represents the accuracies of different models with both cleaned and advanced cleaned samples.



As we have seen from the above graph, the accuracy for SVM and NB is high compared to the other models. Therefore, we have chosen these two ML models as our champion models.

9. Testing Documents without Labels

We tested both the cleaned and advanced cleaned samples that do not have any labels using our champion models (SVM and NB).

```
predict_text = sampled_text
for a, sat in zip(authors, predict_text):
    labeled_texts += ([(txt, a) for txt in sat])

predict_document = pd.DataFrame(labeled_texts)
predict_document = labeled_texts_df.rename(columns={0: "Cleaned Samples",
```

```
                                                    1: "Author"})
clean_text_advanced(predict_document, authors, titles, rem_com=True)
predict_document = predict_document.sample(frac=1).reset_index(drop=True)
```

```
bow_cln_tr = CountVectorizer().fit(predict_document["Cleaned Samples"])
bow_cln_docs = bow_cln_tr.transform(predict_document["Cleaned Samples"])

bow_adv_cln_tr = CountVectorizer().fit(predict_document["Advanced Cleaned
Samples"])
adv_cln_docs = bow_adv_cln_tr.transform(predict_document["Advanced Cleaned
 Samples"])

tfidf_cln_tr = TfidfTransformer().fit(bow_cln_docs)
tfidf_adv_cln_docs = tfidf_cln_tr.transform(bow_cln_docs)

tfidf_adv_cln_tr = TfidfTransformer().fit(adv_cln_docs)
documents_tfidf_2 = tfidf_adv_cln_tr.transform(adv_cln_docs)
```

10. Test and Train

We need to split the data into two portions because if we create a model based on all of the available data, we will not have a metric on how the model performs on new data. For this very reason, we split the data into two parts which are the train data and test data. The following code shows how to split the data in 90% and 10%.

```
x_train_cln, x_test_cln, y_train_cln, y_test_cln = train_test_split(labele
d_texts_df["Cleaned Samples"], labeled_texts_df["Author"], test_size=0.1)
x_train_adv_cln, x_test_adv_cln, y_train_adv_cln, y_test_adv_cln = train_t
est_split(labeled_texts_df["Advanced Cleaned Samples"], labeled_texts_df["
Author"], test_size=0.1)
```

11. Error Analysis

With regards to our champion models (Naive Bayes and SVM Classifiers), we are going to look at the confusion matrix and show the discrepancies between predicted and actual labels.

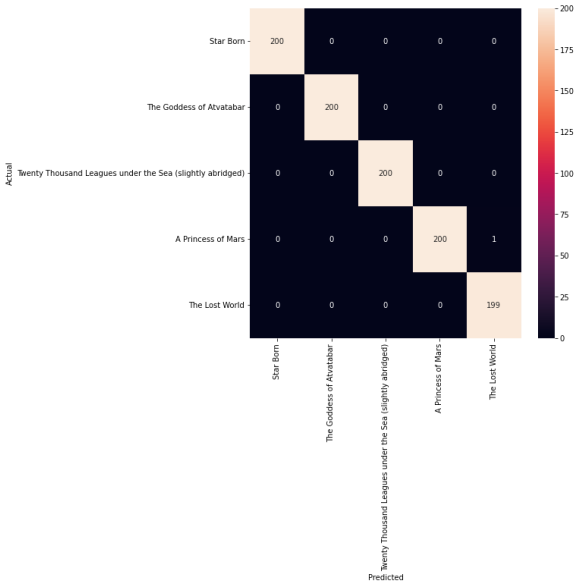
In the following section, we implemented the confusion matrix to show the differences between predicted and actual labels.

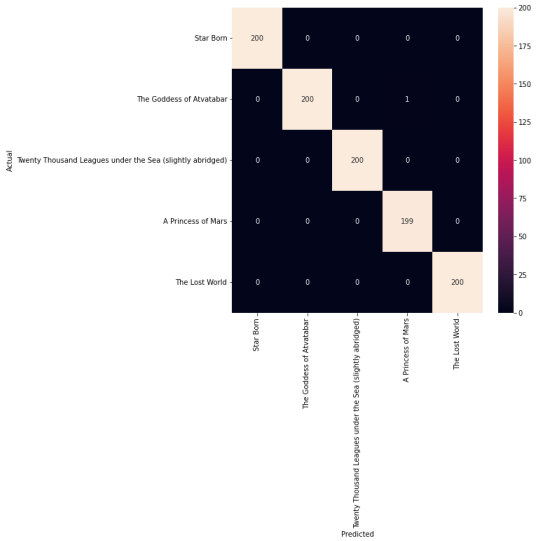
Naive Bayes

```
#
# Champion Classifier: NB
#

# NB + Cleaned Samples
pipeline = Pipeline([("bow", bow_cln_tr),
```

```
                    ("tfidf", tfidf_cln_tr),
                    ("clf", MultinomialNB())])
pipeline.fit(x_train_cln, y_train_cln)
predictions_cln = pipeline.predict(predict_document["Cleaned Samples"])

# NB + Advanced Cleaned Samples
pipeline = Pipeline([("bow", bow_adv_cln_tr),
                    ("tfidf", tfidf_adv_cln_tr),
                    ("clf", MultinomialNB())])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
pipeline.fit(x_train_adv_cln, y_train_adv_cln)
predictions_adv_cln = pipeline.predict(predict_document["Advanced Cleaned
Samples"])

# Extract wrong predictions
print("NB + Cleaned Samples")
df = pd.DataFrame({"Prediction": predictions_cln, "Actual": predict_docume
nt["Author"]})
df["Predicted Wrong"] = df.apply(lambda x: x["Prediction"] == x["Actual"],
 axis=1)
rslt_df = df[df["Predicted Wrong"] == False]
print(rslt_df)

# Extract wrong predictions
print("NB + Advanced Cleaned Samples")
df = pd.DataFrame({"Prediction": predictions_adv_cln, "Actual": predict_do
cument["Author"]})
df["Predicted Wrong"] = df.apply(lambda x: x["Prediction"] == x["Actual"],
 axis=1)
rslt_df = df[df["Predicted Wrong"] == False]
print(rslt_df)

print("NB + Cleaned Samples")
report = classification_report(predictions_cln,predict_document["Author"])
print(report)
conf_mat = confusion_matrix(predictions_cln, predict_document["Author"])
fig, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(conf_mat, annot=True, fmt="d", xticklabels=titles, yticklabels
=titles)
plt.ylabel("Actual")
plt.xlabel("Predicted")

print("NB + Advanced Cleaned Samples")
```

```
report = classification_report(predictions_adv_cln,predict_document["Autho
r"])
print(report)
conf_mat = confusion_matrix(predictions_adv_cln, predict_document["Author"
])
fig, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(conf_mat, annot=True, fmt="d", xticklabels=titles, yticklabels
=titles)
plt.ylabel("Actual")
plt.xlabel("Predicted")
```





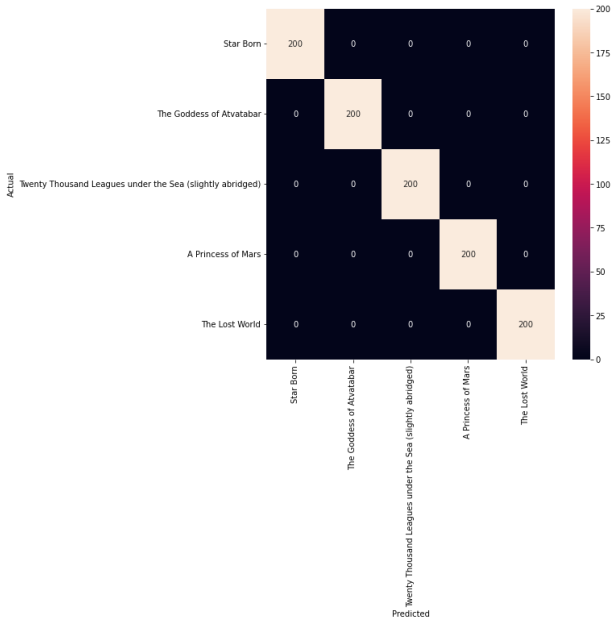
As seen from the confusion matrix above, the vast majority of the predictions are represented at the diagonal.

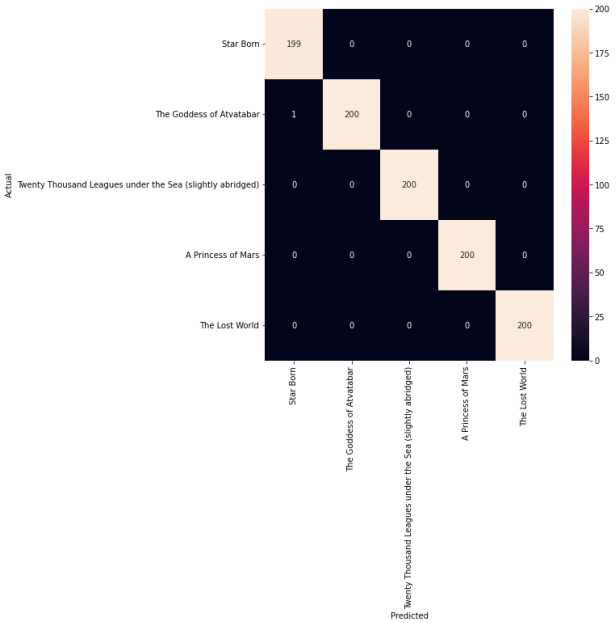
Support Vector Machine

```
#
# Champion Classifier: SVM
#

# SVM + Cleaned Samples
pipeline = Pipeline([("bow", bow_cln_tr),
                     ("tfidf", tfidf_cln_tr),
                     ("clf", SGDClassifier(loss="hinge", penalty="l2", alp
ha=1e-3, random_state=random_state, max_iter=5, tol=None))])
pipeline.fit(x_train_cln, y_train_cln)
predictions_cln = pipeline.predict(predict_document["Cleaned Samples"])

# SVM + Advanced Cleaned Samples
pipeline = Pipeline([("bow", bow_adv_cln_tr),
```

```
                    ("tfidf", tfidf_adv_cln_tr),
                    ("clf", SGDClassifier(loss="hinge", penalty="l2", alp
ha=1e-3, random_state=random_state, max_iter=5, tol=None))])
cv = ShuffleSplit(n_splits=n_splits, test_size=test_size, random_state=ran
dom_state)
pipeline.fit(x_train_cln, y_train_cln)
predictions_adv_cln = pipeline.predict(predict_document["Advanced Cleaned
Samples"])

# Extract wrong predictions
print("SVM + Cleaned Samples")
df = pd.DataFrame({"Prediction": predictions_cln, "Actual": predict_docume
nt["Author"]})
df["Predicted Wrong"] = df.apply(lambda x: x["Prediction"] == x["Actual"],
 axis=1)
rslt_df = df[df["Predicted Wrong"] == False]
print(rslt_df)

# Extract wrong predictions
print("SVM + Advanced Cleaned Samples")
df = pd.DataFrame({"Prediction": predictions_adv_cln, "Actual": predict_do
cument["Author"]})
df["Predicted Wrong"] = df.apply(lambda x: x["Prediction"] == x["Actual"],
 axis=1)
rslt_df = df[df["Predicted Wrong"] == False]
print(rslt_df)

print("SVM + Cleaned Samples")
report = classification_report(predictions_cln,predict_document["Author"])
print(report)
conf_mat = confusion_matrix(predictions_cln, predict_document["Author"])
fig, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(conf_mat, annot=True, fmt="d", xticklabels=titles, yticklabels
=titles)
plt.ylabel("Actual")
plt.xlabel("Predicted")

print("SVM + Advanced Cleaned Samples")
report = classification_report(predictions_adv_cln,predict_document["Autho
r"])
print(report)
conf_mat = confusion_matrix(predictions_adv_cln, predict_document["Author"
])
fig, ax = plt.subplots(figsize=(8, 8))
```

```
sns.heatmap(conf_mat, annot=True, fmt="d", xticklabels=titles, yticklabels
=titles)
plt.ylabel("Actual")
plt.xlabel("Predicted")
```





As seen from the confusion matrix above, the vast majority of the predictions are represented at the diagonal.

12. Conclusion

After using the SVM, Decision Tree, KNN, Random Forest, SVM and Naive Bayes for the classification of text for a data set from 5 different books, we have decided that Naive Bayes and SVM would be our champion models based on the cross-validation accuracy of each model. The Decision Tree classifier performed badly potentially due to a high-dimensional feature space. The Naive Bayes classifier performed the best and was simple to use as well.