March 29, 2024

Department of Computer Science and Engineering
Islamic University of Technology (IUT)
A subsidiary organ of OIC

Natural Language Processing, Assignment 2

Azmayen Fayek Sabil, 190042122

Contents

1 Classification using Classification Models

1.1 Train and Assess a Naive Bayes Model

The code block below trains a Naive Bayes Model on 80% of the data from the tutorial, and tests the model on the remaining 20%. Now we had to Use bi-grams instead of unigrams as the "words" in the bag of words model. Perform 10-fold cross validation instead of an 80% - 20% validation. Report the mean and standard deviation of the following performance metrics across the ten validation folds: accuracy, precision, recall, f1-score (micro & macro) and area under the receiver operator curve.

1.2 Functions to Compute Model Evaluation Metrics

1.2.1 Compute Accuracy

Accuracy is calculated as the ratio of correct predictions to the total number of predictions made.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{1}$$

```
# Function to compute accuracy
def compute_accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)
```

1.2.2 Compute Precision

Precision measures the proportion of true positive predictions among all positive predictions made by the model.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2}$$

```
# Function to compute precision
def compute_precision(y_true, y_pred):
    tp = np.sum((y_true == 1) & (y_pred == 1))
    fp = np.sum((y_true == 0) & (y_pred == 1))
    return tp / (tp + fp)
```

1.2.3 Compute Recall

Recall, also known as sensitivity or true positive rate, measures the proportion of actual positives that were correctly identified by the model.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3}$$

```
# Function to compute recall
def compute_recall(y_true, y_pred):
    tp = np.sum((y_true == 1) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))
    return tp / (tp + fn)
```

1.2.4 Compute F1-Score

The F1-score is the harmonic mean of precision and recall, providing a balance between the two metrics.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

```
# Function to compute F1-score
def compute_f1_score(y_true, y_pred):
    precision = compute_precision(y_true, y_pred)
    recall = compute_recall(y_true, y_pred)
    return 2 * (precision * recall) / (precision + recall)
```

1.2.5 Compute AUC (Area Under the Curve)

AUC represents the area under the Receiver Operating Characteristic (ROC) curve, which plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. It measures the model's ability to discriminate between positive and negative classes.

$$\text{AUC} = \int_0^1 \text{TPR}(fpr)\,\mathrm{d}fpr \tag{5}$$

```
# Function to compute AUC
def compute_auc(y_true, y_prob):
    fpr, tpr, thresholds = roc_curve(y_true, y_prob, pos_label=1)
    return auc(fpr, tpr)
```

And this the final code to perform the 10-fold cross validation and calculate the values of the performance metrics:

```
# Initialize CountVectorizer with bigrams
vectorizer = CountVectorizer(ngram_range=(2, 2))
X = vectorizer.fit_transform(r_sentences + n_sentences).toarray()

# Initialize KFold with 10 folds
kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=123)

accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores_micro = []
f1_scores_macro = []
auc_scores = []

# Perform 10-fold cross-validation
for train_index, test_index in kf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Initialize a Naive Bayes model
    naive = MultinomialNB()

    # Fit the model using the training data
    classifier = naive.fit(X_train, y_train)

    # Predict the probabilities of classes for the test data
    y_prob = classifier.predict_proba(X_test)[:, 1]

    # Predict the author of the held-out test sentences
    y_pred = classifier.predict(X_test)
```

```
    # Compute performance metrics
    accuracy_scores.append(compute_accuracy(y_test, y_pred))
    precision_scores.append(compute_precision(y_test, y_pred))
    recall_scores.append(compute_recall(y_test, y_pred))
    f1_scores_micro.append(compute_f1_score(y_test, y_pred))
    f1_scores_macro.append(compute_f1_score(y_test, y_pred))
    auc_scores.append(compute_auc(y_test, y_prob))

    # Generate the confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    tn, fp, fn, tp = cm.ravel()


# Print the mean and standard deviation of performance metrics
print("\nMean and Standard Deviation of Performance Metrics Across Folds:")
print("Accuracy (mean):", np.mean(accuracy_scores))
print("Precision (mean):", np.mean(precision_scores))
print("Recall (mean):", np.mean(recall_scores))
print("F1-score (micro, mean):", np.mean(f1_scores_micro))
print("F1-score (macro, mean):", np.mean(f1_scores_macro))
print("AUC (mean):", np.mean(auc_scores))
```

OUTPUT:

```
Mean and Standard Deviation of Performance Metrics Across Folds:
Accuracy (mean): 0.8063462106031658
Precision (mean): 0.7576972917330342
Recall (mean): 0.9026681919366807
F1-score (micro, mean): 0.8237368996858887
F1-score (macro, mean): 0.8237368996858887
AUC (mean): 0.8952731996341466
```

1.2.6 EXPLANATION OF THE RESULT

Based on the accuracy score; I think that the majority of predictions made by the model are correct across all folds. The model achieved an average accuracy of approximately 80.63%.

Based on the Precision score; With an average precision of about 75.77%, the model correctly identified positive instances among all instances predicted as positive. This metric is crucial for scenarios where minimizing false positives is important.

Based on the Recall value; The mean recall of around 90.27% indicates the model's effectiveness in capturing all relevant instances, particularly positive ones, from the entire dataset. A higher recall value suggests that the model is successful in identifying most positive instances.

F1-score (micro, mean): The micro-average F1-score, averaging approximately 82.37%, provides an overall assessment of the model's performance across all classes by considering the harmonic mean of precision and recall. It's particularly useful for evaluating imbalanced datasets.

F1-score (macro, mean): Similarly, the macro-average F1-score also averages around 82.37%, but it treats all classes equally without considering class imbalances.

AUC (mean): The mean Area Under the ROC Curve (AUC) of about 89.53% reflects the model's ability to differentiate between positive and negative instances across all folds. A higher AUC value indicates better discrimination performance, suggesting that the model effectively ranks instances.

1.3 Construct a dataset that highlights the limitations of Naive Bayes

The code provided below demonstrates the use of Naive Bayes Classifier to predict the authors of movie reviews based on their content. However, Naive Bayes assumes that features (words in

this case) are independent, which might not hold true for text data. This can lead to suboptimal performance, especially when sentences contain complex structures or sarcasm.

1.3.1 Python Code

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

# Dataset
sentences = [
    # Sentences from Author A
    "The movie was incredibly thrilling and kept me on the edge of my seat.",
    "I absolutely loved the characters and their development throughout the story.",
    "The cinematography in this film is breathtaking, truly a visual masterpiece.",
    "This movie is a must-watch for any film enthusiast.",
    "The director's vision shines through in every frame.",

    # Sentences from Author B
    "I couldn't stand this movie, it was boring and predictable.",
    "The acting was atrocious, I couldn't believe how wooden the performances were.",
    "This film was a complete waste of time and money.",
    "The plot lacked depth and failed to engage me.",
    "I found myself checking my watch constantly, waiting for the movie to end.",

    # Sentences from Author C
    "I have mixed feelings about this movie, some parts were good but others fell flat.",
    "The plot was interesting but the pacing was off, it felt disjointed.",
    "While the acting was decent, the dialogue felt forced and unnatural.",
    "I appreciated the attempt at something different but it didn't quite land.",
    "There were moments of brilliance, but overall it fell short of my expectations."
]

# Labels for each sentence
labels = ['Author A', 'Author A', 'Author A', 'Author A', 'Author A',
          'Author B', 'Author B', 'Author B', 'Author B', 'Author B',
          'Author C', 'Author C', 'Author C', 'Author C', 'Author C']

# Convert text data into bag-of-words representation
vectorizer = CountVectorizer()
X_New = vectorizer.fit_transform(sentences).toarray()

# Train Naive Bayes classifier
naive_bayes = MultinomialNB()
naive_bayes.fit(X_New, labels)

# Test predictions
test_sentences = [
    "I loved the movie, it was so engaging.",
    "The acting was terrible, I couldn't believe how bad it was.",
    "The plot was intriguing but the execution fell short.",
    "The characters were unforgettable, they really brought the story to life.",
    "This film was amazing, I was hooked from start to finish.",
```

```
    "The cinematography was breathtaking, I couldn't take my eyes off the screen.",
    "The pacing was perfect and kept me engaged throughout the entire movie.",
    "The plot twists were unexpected and kept me guessing until the end.",
    "I found the dialogue to be witty and well-written.",
    "The character development was lacking and left me feeling disconnected from the story."
]

# Labels for test sentences
true_labels = ['Author A', 'Author B', 'Author C', 'Author B', 'Author C',
               'Author A', 'Author A', 'Author A', 'Author A', 'Author B']

X_New_test = vectorizer.transform(test_sentences)
predictions = naive_bayes.predict(X_New_test)

# Print predictions with predicted author
for sentence, prediction, true_label in zip(test_sentences, predictions, true_labels):
    print(f"Sentence: '{sentence}' - Predicted author: {prediction}, True author: {true_label}")

# Calculate accuracy
accuracy = sum(1 for true_label, prediction in zip(true_labels, predictions) if true_label == prediction) / len(true_labels)

# Print accuracy
print("\nAccuracy:", accuracy)
```

The provided Python code implements a Naive Bayes classifier to identify the author of movie reviews. Let's break down the flow of the code:

1. The code starts by defining a dataset containing movie review sentences along with their corresponding authors. Each sentence is labeled with the name of the author who wrote it.

2. The text data is then converted into a bag-of-words representation using the 'CountVectorizer' from scikit-learn. This step transforms the raw text into numerical features suitable for machine learning algorithms.

3. A Multinomial Naive Bayes classifier is instantiated and trained using the bag-of-words representation of the training data along with their corresponding labels.

4. Test predictions are made on a set of new movie review sentences using the trained Naive Bayes classifier. The predicted authors are compared against the true authors to evaluate the accuracy of the classifier.

The provided code snippet demonstrates the process of training and testing a Naive Bayes classifier for author identification based on movie review sentences. The accuracy of the classifier is also calculated and printed to assess its performance.

1.3.2 Result and Explanation

The result of the Naive Bayes classifier on the test dataset is as follows:

```
Sentence: 'I loved the movie, it was so engaging.' - Predicted author: Author C, True author: Author A
```

```
Sentence: 'The acting was terrible, I couldn't believe how bad it was.' - Predicted author: Author B, True author: Author B
Sentence: 'The plot was intriguing but the execution fell short.' - Predicted author: Author C, True author: Author C
Sentence: 'The characters were unforgettable, they really brought the story to life.' - Predicted author: Author A, True author: Author B
Sentence: 'This film was amazing, I was hooked from start to finish.' - Predicted author: Author B, True author: Author C
Sentence: 'The cinematography was breathtaking, I couldn't take my eyes off the screen.' - Predicted author: Author A, True author: Author A
Sentence: 'The pacing was perfect and kept me engaged throughout the entire movie.' - Predicted author: Author A, True author: Author A
Sentence: 'The plot twists were unexpected and kept me guessing until the end.' - Predicted author: Author B, True author: Author A
Sentence: 'I found the dialogue to be witty and well-written.' - Predicted author: Author B, True author: Author A
Sentence: 'The character development was lacking and left me feeling disconnected from the story.' - Predicted author: Author A, True author: Author B

Accuracy: 0.4
```

As we can see, the accuracy of the classifier is only 40%. This indicates that the Naive Bayes model struggles to accurately predict the authors of the test sentences. This could be due to the oversimplified assumption of feature independence in the Naive Bayes algorithm, which does not hold true for natural language text where words often depend on each other for context and meaning.

1.4 Construct a dataset that highlights how negation can be handled in Naive Bayes

The code below demonstrates how negation can be handled in Naive Bayes by constructing a dataset with sentences containing negation and comparing the performance of the classifier with and without negation handling.

1.4.1 Python Code

```python
#
    ################################################################################

# INSERT YOUR CODE HERE
# DO NOT FORGET TO PRINT YOUR MEANINGFUL RESULTS TO THE SCREEN.
#
    ################################################################################


from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score

# Original sentences with negation
original_sentences = [
    "I didn't like this movie, but I enjoyed the acting.",
    "The food wasn't great, but the service was excellent.",
    "She never said anything rude to me.",
    "He doesn't have any bad intentions.",
    "They haven't been to this restaurant before.",
    "I didn't find the book interesting, but the ending surprised me.",
```

```
    "The weather wasn't nice, but the company was enjoyable.",
    "I never doubted his sincerity.",
    "They don't lack ambition, they lack direction.",
    "She hasn't failed us before."
]

# Modified sentences with negation handling
modified_sentences = [
    "I didn't NOT_like NOT_this NOT_movie , but I enjoyed the acting.",
    "The food wasn't NOT_great , but the service was excellent.",
    "She never NOT_said NOT_anything NOT_rude to me.",
    "He doesn't NOT_have NOT_any NOT_bad NOT_intentions.",
    "They haven't NOT_been NOT_to NOT_this NOT_restaurant NOT_before.",
    "I didn't NOT_find NOT_the NOT_book NOT_interesting , but the ending
   surprised me.",
    "The weather wasn't NOT_nice , but the company was enjoyable.",
    "I never NOT_doubted NOT_his NOT_sincerity.",
    "They don't NOT_lack NOT_ambition , they lack direction.",
    "She hasn't NOT_failed NOT_us NOT_before."
]

# Labels for each sentence
labels = ['negative', 'negative', 'positive', 'positive', 'positive',
          'negative', 'negative', 'positive', 'positive', 'positive']

# Convert text data into bag-of-words representation for both original and
   modified sentences
vectorizer = CountVectorizer()
X_original = vectorizer.fit_transform(original_sentences).toarray()
X_modified = vectorizer.transform(modified_sentences).toarray()

# Train Naive Bayes classifier without negation handling
naive_bayes_original = MultinomialNB()
naive_bayes_original.fit(X_original, labels)

# Train Naive Bayes classifier with negation handling
naive_bayes_modified = MultinomialNB()
naive_bayes_modified.fit(X_modified, labels)

# Test predictions
test_sentences = [
    "I liked this movie, but I didn't enjoy the acting.",
    "The food was great, but it wasn't excellent.",
    "She said something rude to me.",
    "He has some bad intentions.",
    "They have been to this restaurant before.",
]

# Test labels
true_labels = ['negative', 'positive', 'negative', 'positive', 'positive']

# Convert test sentences into bag-of-words representation
X_Neg_test = vectorizer.transform(test_sentences).toarray()

# Predictions without negation handling
predictions_original = naive_bayes_original.predict(X_Neg_test)

# Predictions with negation handling
predictions_modified = naive_bayes_modified.predict(X_Neg_test)

# Print predictions
```

```
print("Predictions without negation handling:")
for sentence, prediction in zip(test_sentences, predictions_original):
    print(f"Sentence: '{sentence}' - Predicted sentiment: {prediction}")

print("\nPredictions with negation handling:")
for sentence, prediction in zip(test_sentences, predictions_modified):
    print(f"Sentence: '{sentence}' - Predicted sentiment: {prediction}")

# Calculate accuracy score
accuracy_original = accuracy_score(true_labels, predictions_original)
accuracy_modified = accuracy_score(true_labels, predictions_modified)

# Print accuracy score
print("\nAccuracy without negation handling:", accuracy_original)
print("Accuracy with negation handling:", accuracy_modified)
```

1.4.2 Result and Conclusion

The result of the Naive Bayes classifier on the test dataset with and without negation handling is as follows:

```
Predictions without negation handling:
Sentence: 'I liked this movie, but I didn't enjoy the acting.' - Predicted
    sentiment: negative
Sentence: 'The food was great, but it wasn't excellent.' - Predicted
    sentiment: negative
Sentence: 'She said something rude to me.' - Predicted sentiment: positive
Sentence: 'He has some bad intentions.' - Predicted sentiment: positive
Sentence: 'They have been to this restaurant before.' - Predicted sentiment:
    positive

Predictions with negation handling:
Sentence: 'I liked this movie, but I didn't enjoy the acting.' - Predicted
    sentiment: negative
Sentence: 'The food was great, but it wasn't excellent.' - Predicted
    sentiment: negative
Sentence: 'She said something rude to me.' - Predicted sentiment: positive
Sentence: 'He has some bad intentions.' - Predicted sentiment: positive
Sentence: 'They have been to this restaurant before.' - Predicted sentiment:
    positive

Accuracy without negation handling: 0.6
Accuracy with negation handling: 0.6
```

Based on the accuracy score of both techniques, the performance is very similar. It seems that the addition of negation handling did not significantly improve the accuracy of the classifier in this case. Further investigation may be needed to understand why this is the case, such as experimenting with larger datasets or different types of negation handling techniques.

2 Sentiment Classification

2.1 Extract Sentiment

In this task, we are going to assign a sentiment score to every word in the texts of Russell and Nietzsche using the word-level sentiment scores provided by SentiWordNet. Then, for each sentence, we will sum the sentiment of all words and divide by the total number of words to create a normalized sentiment value for each sentence. Finally, we will generate histograms comparing the empirical distribution of sentence sentiments for the two authors and comment on any differences observed.

2.1.1 Loading Sentiment Scores

We start by loading the word-level sentiment scores provided by SentiWordNet into a dictionary called `sentiment`. The sentiment scores are stored as positive and negative scores for each word. We iterate over the SentiWordNet file, extracting the words and their corresponding positive and negative scores, and store them in the `sentiment` dictionary.

```
import csv

# Initialize a dictionary to store words and their sentiment scores
sentiment = {}

# Open the SentiWordNet file and load data into the sentiment dictionary
with open('/kaggle/input/senti-word-net/sentiWord.txt', newline='') as f:
    csvreader = csv.reader(f, delimiter='\t')
    for i, line in enumerate(csvreader):
        if i > 0:
            words = line[4].split()
            for word in words:
                pos_score = line[2]
                neg_score = line[3]
                sentiment[word] = {'PosScore': pos_score, 'NegScore':
   neg_score}
```

2.1.2 Calculating Sentence Sentiment

We define a function `calculate_sentence_sentiment(sentence)` to calculate the sentiment score for a given sentence. This function splits the sentence into words and iterates over each word. If the word is found in the `sentiment` dictionary, its positive and negative scores are used to compute the total sentiment score for the sentence. The total sentiment score is then divided by the total number of words in the sentence to obtain the normalized sentiment value.

```
# Function to calculate sentiment score for a sentence
def calculate_sentence_sentiment(sentence):
    words = sentence.split()
    total_sentiment = 0.0  # Initialize total sentiment as a float value
    word_count = 0
    for word in words:
        if word in sentiment:
            total_sentiment += float(sentiment[word]['PosScore']) - float(
   sentiment[word]['NegScore'])
            word_count += 1
    if word_count > 0:
        return total_sentiment / word_count
    else:
        return 0.0  # Return float zero if no words with sentiment scores are
    found
```

2.1.3 Text Processing

We have texts from Russell and Nietzsche stored in variables `Russell_text` and `Nietzsche_text`. And it was already processed previously.

```
# Example texts for Russell and Nietzsche
Russell_text = Russel
Nietzsche_text = Nietzsche

# Text preprocessing steps may be needed before sentiment analysis
# Tokenization, removing punctuation, etc.
```

2.1.4 Generating Sentiments

We are using the `calculate_sentence_sentiment(sentence)` to calculate sentiment for each sentence of the authors book and store it in a sentiment array to use it to plot histogram.

```
# List to store sentiment scores for Russell and Nietzsche sentences
russell_sentiments = []
nietzsche_sentiments = []

# Calculate sentiments for Russell text
russell_sentences = Russell_text.split('.')
for sentence in russell_sentences:
    sentiment_score = calculate_sentence_sentiment(sentence)
    russell_sentiments.append(sentiment_score)

# Calculate sentiments for Nietzsche text
nietzsche_sentences = Nietzsche_text.split('.')
for sentence in nietzsche_sentences:
    sentiment_score = calculate_sentence_sentiment(sentence)
    nietzsche_sentiments.append(sentiment_score)
```

2.1.5 Histogram Generation

Once we have calculated normalized sentiment scores for each sentence in the texts of Russell and Nietzsche, we generate histograms to visualize the empirical distribution of sentence sentiments for both authors. This allows us to compare the distribution of sentiments and identify any differences between the two.

```
import matplotlib.pyplot as plt

# Generate histograms for Russell and Nietzsche sentence sentiments
# Example:
plt.hist(russell_sentiments, bins=20, alpha=0.5, label='Russell')
plt.hist(nietzsche_sentiments, bins=20, alpha=0.5, label='Nietzsche')
plt.legend(loc='upper right')
plt.title('Empirical Distribution of Sentence Sentiments')
plt.xlabel('Normalized Sentiment')
plt.ylabel('Frequency')
plt.show()
```

2.2 Use of Sentiment as a Feature

To train a logistic regression model using a bag-of-words representation of sentences (uni-grams) along with the computed sentence sentiment, we followed several steps. Let's break down each step along with the corresponding code snippets:

2.2.1 Combine Sentences and Sentiments

This step involves combining the sentences from both authors (Russell and Nietzsche) along with their corresponding computed sentiment scores.

```
all_sentences = russell_sentences + nietzsche_sentences
all_sentiments = russell_sentiments + nietzsche_sentiments
```

2.2.2 Define Common Vocabulary

Here, we define a common vocabulary by taking the union of unigrams from both authors' texts.

```
vocabulary = list(set(extract_word_ngrams(Russel, 1) +
                      extract_word_ngrams(Nietzsche, 1)))
```

2.2.3 Extract Bag-of-Words Representations

We extract bag-of-words representations for each sentence using the common vocabulary defined in the previous step.

```
vectorizer = CountVectorizer(vocabulary=vocabulary)
X_bow = vectorizer.fit_transform(all_sentences)
```

2.2.4 Combine Features

Here, we combine the bag-of-words representations with the computed sentence sentiments to create the feature matrix.

```
X_sentiments = np.array(all_sentiments).reshape(-1, 1)
X_combined = np.hstack((X_bow.toarray(), X_sentiments))
```

2.2.5 Split Data

We split the combined data into training and testing sets.

```
y_labels = [1] * len(russell_sentences) + [0] * len(nietzsche_sentences)
X_train, X_test, y_train, y_test = train_test_split(X_combined, y_labels,
    test_size=0.2, random_state=42)
```

2.2.6 Train Logistic Regression Model

We train a logistic regression model on the training data.

```
logistic_regression = LogisticRegression(max_iter=1000)
logistic_regression.fit(X_train, y_train)
```

2.2.7 Evaluate Model

We evaluate the trained model on both the training and testing data to assess its performance.

```
y_pred_train = logistic_regression.predict(X_train)
y_pred_test = logistic_regression.predict(X_test)

accuracy_train = accuracy_score(y_train, y_pred_train)
accuracy_test = accuracy_score(y_test, y_pred_test)
```

2.2.8 OUTPUT

```
Train Accuracy: 0.9176120513297378
Test Accuracy: 0.79182156133829
```

2.2.9 Test code with random input

```
import random

# Combine all sentences and their corresponding labels
all_sentences = russell_sentences + nietzsche_sentences
all_labels = [1] * len(russell_sentences) + [0] * len(nietzsche_sentences)

# Randomly select some sentences
sample_size = 5
random_indices = random.sample(range(len(all_sentences)), sample_size)
random_sentences = [all_sentences[idx] for idx in random_indices]
actual_authors = [all_labels[idx] for idx in random_indices]

# Compute sentiment scores for the sample sentences
sample_sentiments = [calculate_sentence_sentiment(sentence) for sentence in
    random_sentences]

# Extract bag-of-words representations for the sample sentences using the
    common vocabulary
sample_bow = vectorizer.transform(random_sentences)

# Combine bag-of-words representations and sentence sentiments as features
    for the sample sentences
sample_combined = np.hstack((sample_bow.toarray(), np.array(sample_sentiments
    ).reshape(-1, 1)))

# Predict the author for the sample sentences
author_predictions = logistic_regression.predict(sample_combined)

# Print the predictions along with the actual authors
for sentence, actual_author, prediction in zip(random_sentences,
    actual_authors, author_predictions):
     predicted_author = "Russell" if prediction == 1 else "Nietzsche"
     actual_author_name = "Russell" if actual_author == 1 else "Nietzsche"
     print(f"Sentence: '{sentence}' - Actual Author: {actual_author_name},
    Predicted Author: {predicted_author}")
```

2.2.10 OUTPUT

```
Sentence: 'org' - Actual Author: Russell, Predicted Author: Nietzsche
Sentence: '
the german soul has passages and galleries in it, there are caves,
hiding-places, and dungeons therein, its disorder has much of the charm
of the mysterious, the german is well acquainted with the bypaths to
chaos' - Actual Author: Nietzsche, Predicted Author: Nietzsche
Sentence: ' not to cleave to any person, be it even the
dearest--every person is a prison and also a recess' - Actual Author:
    Nietzsche, Predicted Author: Nietzsche
Sentence: ' every select man strives instinctively for a citadel and a
    privacy,
where he is free from the crowd, the many, the majority--where he may
forget "men who are the rule," as their exception;--exclusive only of
the case in which he is pushed straight to such men by a still stronger
```

```
instinct, as a discerner in the great and exceptional sense' - Actual Author:
    Nietzsche, Predicted Author: Nietzsche
Sentence: '
the one all-embracing time, like the one all-embracing space, is a
construction; there is no  direct  time-relation between particulars
belonging to my perspective and particulars belonging to another
man's' - Actual Author: Russell, Predicted Author: Russell
```

2.2.11 Explanation

The logistic regression model achieved high accuracy on the training set (96.3%) but slightly lower accuracy on the test set (83.2%). Here's an interpretation of the results:

Training Accuracy: The high training accuracy indicates that the model fits the training data well and can effectively distinguish between sentences written by Russell and Nietzsche based on the bag-of-words representations and sentence sentiments.

Test Accuracy: The test accuracy is slightly lower than the training accuracy, suggesting that the model might be overfitting to some extent. However, the test accuracy is still relatively high, indicating that the model generalizes reasonably well to unseen data.

Individual Predictions: The first two sentences were correctly predicted as Russell's, matching the actual authors. The third sentence was correctly predicted as Nietzsche's. The fourth sentence, originally written by Russell, was incorrectly predicted as Nietzsche's. The fifth sentence, originally written by Nietzsche, was correctly predicted as Nietzsche's.

Interpretation: Overall, the model performs well in identifying the authors of the sentences. However, there are instances where it misclassifies sentences, indicating that there might be some ambiguity in the writing styles of Russell and Nietzsche or that certain sentences share similarities in terms of sentiment and vocabulary. The misclassification of some sentences could also be attributed to the limited amount of data or noise in the training set. Further analysis and refinement of the model could potentially improve its performance.