Shrutika Shrutika USC ID: 7908890187

BONUS QUESTIONS

Detail information is provided in Readme.md here:
https://github.com/ShrutikaSingh/csci444_assignment1/blob/main/_bonus/Readme.md

Question 1) Logistic Regression

Main Code file: _bonus_1_logistic_regression_model.py, **Note:** Code takes around 2 mins to run

Logistic Regression Code Logic:

```
        linear_model = np.dot(X, self.weights) + self.bias # Linear model: y_hat = X * w + b
        y_hat = self.sigmoid(linear_model) # Apply sigmoid function
        # Gradient computation
        dw = (1 / n_samples) * np.dot(X.T, (y_hat - y))
        db = (1 / n_samples) * np.sum(y_hat - y)
    def predict(self, X):
     linear_model = np.dot(X, self.weights) + self.bias
     y_hat = self.sigmoid(linear_model)
```

Logistic regression Evaluation Metrics:

```
(venv) 206819985@OTS-FVFHK1DWQ05Q-MAC hw1-imdb % python3 _bonus_1_logistic_regression_model.py --data_src /Users/206819985/Documents/csci_444_assign/hw1-imdb
vocab length for Logistic Regression = 10000
Validation Accuracy: 0.8264
Validation Precision: 0.8199
Validation Recall: 0.8363
Validation F1: 0.8281
```

Command for running the code:

``` python _bonus_1_logistic_regression_model.py –data_src <path_to_data_folder>

```python3 _bonus_1_logistic_regression_model.py --data_src /Users/206819985/Documents/csci_444_assign/hw1-imdb```

### Result Files Generated

**Test_predictions:** **test_predictions.csv**
**Val Predictions:** **val_predictions.csv**

**Results Comparison:** Comparison of Naive Bayes and Logistics Regression Evaluation Metrics for Imdb data set with Vocab Size =10000

| | Naive Bayes | Logistic Regression | My Reasoning |
|---|---|---|---|
| Accuracy | 0.8570 | 0.8264 | Naive Bayes is effective in handling high-dimensional sparse data so that better overall accuracy can be achieved by assuming that features are conditionally independent given the class label. |
| Precision | 0.8653 | 0.8199 | Naive Bayes better captures relevant features and reduces false positives, hence achieving higher precision. |
| Recall | 0.8455 | 0.8363 | Naive Bayes performs well in identifying relevant instances due to its probabilistic nature, leading to higher recall. |
| F1 Score | 0.8553 | 0.8281 | Naive Bayes balances precision and recall effectively, resulting in a higher F1 score by maintaining a good trade-off between precision and recall. |

I believe naive bayes performed better because of following reasons

**Feature Assumptions**: Naive Bayes assumes that features are independent, thus going well with the bag-of-words approach. Therefore, this model can also result in better performance in those cases where the features are sparse, as in textual data.

**Simplicity with High Dimensionality**: Naive Bayes is really efficient for high-dimensional feature space and less prone to overfitting, while Logistic Regression may be problematic in the case of high-dimensional data if it is not regularized properly.

**Robustness to Noise**: Naive Bayes does not care about noisy features and/or irrelevant words. Because this is a probabilistic algorithm, it is pretty robust in cases of text data, where noise is quite common.

**Effective with Sparse Data**: Naive Bayes should be more effective in the case of the binary features that are sparse in nature, like the presence or absence of words, because it does not need to normalize or scale. This will most probably work better than Logistic Regression in this case, where scaling of features is not quite appropriate.

## Q2) Naive Bayes with Continuous Features (TF-IDF):

**Command for Running the code**

> python3 _bonus_2.py –data_src <path_to_data_folder>

> python3 _bonus_2.py --data_src /Users/206819985/Documents/csci_444_assign/hw1-imdb

**Tokenizer Logic with TF-iDF**

```
    def text_to_tfidf_vector(self, text, tfidf_vectorizer):
        # Convert text to TF-IDF feature vector
        return tfidf_vectorizer.transform([text]).toarray()[0]
```

```
    # Tokenize and convert texts to TF-IDF feature vectors
    tokenizer = Tokenizer()
    tokenizer.build_vocab(train_texts)
    #  to generate continuous features
    tfidf_vectorizer = TfidfVectorizer(max_features=10000, vocabulary=tokenizer.vocab,
stop_words=list(tokenizer.stop_words))
    tfidf_vectorizer.fit(train_texts)
```

Result Comparison (Vocab Size=10000):

	Naive Bayes with Binary Features	Naive Bayes TF-IDF (Continuous Features)	**My Reasoning**
Accuracy	0.8570	0.8616	TF-IDF's representation improves overall accuracy by better capturing term importance.
Precision	0.8653	0.8636	Very slightly lower for TF-IDF, since it can pick up noise from less common terms, despite weighting the terms better overall.
Recall	0.8455	0.8587	Since TF-IDF can weigh terms, it improves on recall because it focuses more on the relevant terms.
F1 Score	0.8553	0.8612	A higher F1 score with TF-IDF reflects a better balance between precision and recall.

Result Files Generated

Test_predictions: **test_predictions.csv**

Val Predictions: **val_predictions.csv**

PROBLEM 3: POS TAGGING

IMPORTANT NOTE: Since the code was taking more than 7 hrs to run, I created a dataset with a total of just **100 samples,** 70 text files were in the train folder, 15 were in test and 15 was in the validation.

And using the data set provided by the professor, I manually update the train_labels.csv, val_labels.csv, test_paths.csv corresponding to the same labels and data provided by the professor.

Command to run

> python -m spacy download en_core_web_sm

> python3 _bonus_3.py --data_src <path_to_your_data_folder>

> python3 _bonus_3.py --data_src /Users/206819985/Documents/csci_444_assign/hw1-imdb/_bonus/data

Code logic

```
# Load spaCy model
nlp = spacy.load("en_core_web_sm")
```

```
        most_common_words = sorted(word_counts.items(), key=lambda x: x[1], reverse=True)[:vocab_size]
        self.vocab = {word: idx for idx, (word, _) in enumerate(most_common_words)}
        self.pos_vocab = {pos: idx + len(self.vocab) for idx, (pos, _) in enumerate(pos_counts.items())}
        self.vocab_size = len(self.vocab) + len(self.pos_vocab)
```

For 100 samples,
True Positives (TP): 30
False Positives (FP): 15
True Negatives (TN): 27
False Negatives (FN): 28

For 100 samples,

Validation Accuracy: 0.570
Validation Precision: 0.667
Validation Recall: 0.517
Validation F1: 0.583

	Simple Naive Bayes **(Samples = 40k)**	**Naive Bayes With POS tagging (samples = 100)**
Accuracy	0.8570	0.570
Precision	0.8653	0.667
Recall	0.8455	0.517
F1 Score	0.8553	0.583

My Observation

For the smaller dataset (100 samples), the model with POS tagging performs worse, which aligns with the observation that more data might be needed to see if POS tagging could offer any improvements.

But if the simple naive bayes is trained with just 100 samples without POS tagging, the performance is even worse. That means that with POS tagging , accuracy is improved if we have a large data set.

Result Files Generated

Test_predictions: **test_predictions.csv**
Val Predictions: **val_predictions.csv**