Logistic Regression for Text Classification

Here are the steps I took during the development of this Logistic Regression classifier. I am going to skip the part where I followed the assignment guidelines to get my code running and talk about my hyperparamter tuning and feature engineering steps. Finally, I am going to reflect on my evaluation results on the development set.

I began with a straightforward implementation, using simple word counts (unigrams) for feature representation. Each document was transformed into a feature vector where each dimension corresponded to the frequency of a specific word within the document. In line with the assignment's requirements, a bias term was included as the last element of each vector.

When I initially got my code up and running, the performance metrics look like the following:

```
Epoch 1 out of 1
Average Train Loss: 4.651314353120697
Results for class 'neg':
Precision: 0.5714285714285714
Recall: 0.9504950495049505
F1 Score: 0.7137546468401488
Results for class 'pos':
Precision: 0.84375
Recall: 0.2727272727272727
F1 Score: 0.41221374045801523
[[96.  5.]
 [72. 27.]]
Overall Accuracy: 0.615
```

After getting this initial output, I strived to improve precision and recal for the positive class and get a more balanced performance across both classes.

Hyperparameter Tuning

After setting up the initial model, I decided to experiment with increasing the number of training epochs from 1 to 10. This decision was based on the intuition that allowing the model more iterations over the training data could potentially lead to better learning and generalization. Here is the result I got with 10 training epochs:

```
Epoch 1 out of 10
Average Train Loss: 4.828262189461036
Epoch 2 out of 10
Average Train Loss: 3.8470607189408486
Epoch 3 out of 10
```

```
Average Train Loss: 3.3329956843600494
Epoch 4 out of 10
Average Train Loss: 3.121450397673057
Epoch 5 out of 10
Average Train Loss: 2.779418143549321
Epoch 6 out of 10
Average Train Loss: 2.236966814424457
Epoch 7 out of 10
Average Train Loss: 2.203352647370467
Epoch 8 out of 10
Average Train Loss: 1.9641581420623342
Epoch 9 out of 10
Average Train Loss: 1.518939468130532
Epoch 10 out of 10
Average Train Loss: 1.5758587306013871
Results for class 'neg':
Precision: 0.9714285714285714
Recall: 0.33663366336633666
F1 Score: 0.5000000000000001
Results for class 'pos':
Precision: 0.593939393939394
Recall: 0.98989898989899
F1 Score: 0.7424242424242425
[[34. 67.]
 [ 1. 98.]]
Overall Accuracy: 0.66
```

As we can see the overall accuracy went from 61.5% to 66%, which showed me I was moving in the right direction. To more systematically determine the optimal number of training epochs, I developed a function dedicated to this task. This function trained the model across a range of epoch values and evaluated its performance on the development set after each training cycle. Through this methodical approach, the function identified that the model achieved its peak performance at 13 epochs.

Following the optimization of epochs, I experimented with different learning rates to further refine performance. I tested learning rates of 0.001, 0.01, 0.1, and 0.05, evaluating the model's accuracy on the development set after training with each rate. This experimentation yielded the following results: a learning rate of 0.001 achieved 73.5% accuracy, 0.01 reached 83%, 0.1 resulted in 82%, and notably, 0.05 led to the highest accuracy of 83.5%. Through these findings I decided to submit my assignment with a learning rate of 0.05.
Now that I had this accuracy I wanted to improve my featurize function from simple counts to including bigram features.

I transformed my featurize function from this:

```
def featurize(self, document: Sequence[str]) -> np.array:
    '''
    Given a document (as a list of words), returns a feature vector.
    Note that the last element of the vector, corresponding to the bias, is a
    "dummy feature" with value 1.
    '''
    vector = np.zeros(self.n_features + 1)   # + 1 for bias
    # your code here

    for word in document:
        if word in self.feature_dict:  # Check if the word is a recognized feature
            vector[self.feature_dict[word]] += 1  # Increment feature count

    vector[-1] = 1   # bias
    return vector
```

To this:

```
def featurize(self, document: Sequence[str], n=2) -> np.array:
    '''
    Given a document (as a list of words), returns a feature vector.
    Note that the last element of the vector, corresponding to the bias, is a
    "dummy feature" with value 1.
    '''
    vector = np.zeros(self.n_features + 1)   # + 1 for bias
    # your code here

    for word in document:
        if word in self.feature_dict:  # Check if the word is a recognized feature
            vector[self.feature_dict[word]] += 1  # Increment feature count

    # Count bigrams
    if n >= 2:
        for i in range(len(document)-1):
            bigram = ' '.join(document[i:i+2])
            if bigram in self.feature_dict:
                vector[self.feature_dict[bigram]] += 1

    vector[-1] = 1   # bias
    return vector
```

Additionally, I made some changes on the make_dicts function to include n-grams, specifically unigrams and bigrams.

While this approach significantly enriched the model's understanding of the text, it also introduced a considerable increase in computational complexity, leading to a longer runtime. Despite the slower performance, the incorporation of bigrams proved improved the overall accuracy to 84%.

Afterwards, to enhance the consistency of my model, I decided to convert all words to lowercase. This adjustment was implemented in both the make_dicts and featurize functions. Although this modification did not result in an improvement in the overall accuracy, which remained at 84%, I believe it was a beneficial step. Lowercasing all words helped standardize the input data and ensure that the model treats words with different capitalizations as identical features.

Moving forward, I would like to experiment with more effective featurization strategies to improve my model. If I had more time, I would've like to try is to use stemming, which simplifies words to their base forms, potentially making the model more efficient by grouping similar words together. Additionally, I'm thinking about using a list of positive and negative words to give the model more direct clues about sentiment.