CSE 5525 Assignment 1 - Sentiment Analysis

Jake Imyak

September 8, 2021

1 Question 1

For the UnigramFeatureExtractor, I was tasked with creating feature extraction for the unigram bag-of-words. The first decision I made was to utilize the Indexer class to create and index to store the words. I looped through each word in the word list to add to the Indexer. Within this loop, I choose to make the each word lowercase and removed the stop words to reduce the amount of words in the vector. The logic behind making everything lowercase was the sentiment change between lowercase and uppercase isn't significant; hence, I thought it was worthwhile to speeding up run time. Then, I used the NLTK stop words corpus to get rid of filler words such as "the" and "a" since they won't sway sentiment when we are only looking at individual words. The feature vector is representing a vector that is initially 0 and adding one if the unigram with the corresponding index is in the sentence. To do this, the Counter from the python collection class is used to store the sparse vector and the indexer provided in utils.py is used for keeping track of objects and indices. This feature vector is created in the extract_features() function.

2 Question 2

To implement the Bigram feature extactor, I used a similar process as the Unigram feature extractor. I utilized a indexer to keep track of the index of each pair - the bigram is represented by each adjacent pair of words. Then, adding features of the words by looping through n-1 words and adding the lower case of both the i-th and i+1-th position of said words. I tried removing bigrams with stops words here, however, it made for worse performance so I omitted it. The feature vector is representing a vector that is initially 0 and adding one if the bigram or adjacent words with the corresponding index is in the sentence. To do this, the Counter from the python collection class is used to store the sparse vector and the indexer provided in utils.py is used for keeping track of objects and indices. This feature vector is created in the extract_features() function.

3 Question 3

I used Figure 5.5 as a guide in JM for this implementation. To implement Logistic Regression, first I had to implement a function to get the feature set from a List of words. To do this, I took the list and converted it into a string then using the feature extractor initialized with the Logistic Regression class, get all the features then store and return it. In order to optimize our result, we need to take the gradient of our cross entropy loss function or $L_{CE}(y, \hat{y}) = -[ylog\sigma(w \cdot x + b) + (1 - y)log(1 - \sigma(w \cdot x + b))]$. To optimize this, we can use gradient descent to find the optimal weight; hence, minimizing the loss function. Therefore, we can find the weighted feature value by looping through all the values in our feature vector and multiplying that by the weights. Then using our sigmoid function, get the probability for a prediction. Next using the following equation: $\frac{\partial L_{CE}(y,\hat{y})}{\partial w_j} = [\sigma(w \cdot x + b) - y]x_j$, loop through and update the weights accordingly by subtracting alpha times that gradient from the current weight to update. We will predict positive for p greater than 0.5 and negative otherwise. I used 20 epochs and a learning rate of 0.1. Within training I shuffled the training data then for each example, get the label, use the model for the prediction, then update the model using gradient descent as stated before.

The results are as follows for the unigram:

Train Accuracy:

Accuracy: 6851 / 6920 = 0.990029

Precision (fraction of predicted positives that are correct): 3571 / 3601 = 0.991669; Recall (fraction of true positives predicted correctly): 3571 / 3610 = 0.989197; F1 (harmonic mean of precision and recall): 0.990431

Dev Accuracy:

Accuracy: 677 / 872 = 0.776376

Precision (fraction of predicted positives that are correct): 353 / 457 = 0.772429; Recall (fraction of true positives predicted correctly): 353 / 444 = 0.795045; F1 (harmonic mean of precision and recall): 0.783574

Time for training and evaluation: 3.84 seconds

4 Question 4

The results for the bigrams are as follows:

Train Accuracy

Accuracy: 6920 / 6920 = 1.000000

Precision (fraction of predicted positives that are correct): 3610 / 3610 = 1.000000; Recall (fraction of true positives predicted correctly): 3610 / 3610 = 1.000000; F1 (harmonic mean of precision and recall): 1.000000

Dev Accuracy

Accuracy: 643 / 872 = 0.737385

Precision (fraction of predicted positives that are correct): 349 / 483 = 0.722567; Recall (fraction of true positives predicted correctly): 349 / 444 = 0.786036; F1 (harmonic mean of precision and recall): 0.752967

Time for training and evaluation: 5.41 seconds

5 Question 5

For the BetterFeatureExtractor, I combined unigrams and bigrams to make a feature extractor that took both of these into account. To do this, I stored both bigrams and unigrams in an indexer. By doing this, we can use the data between both the bigrams and bag-of-words approach to hopefully make a more informed decision. The feature vector is representing a vector that is initially 0 and adding one if the unigram or bigram (adjacent words) with the corresponding index is in the sentence. To do this, the Counter from the python collection class is used to store the sparse vector and the indexer provided in utils.py is used for keeping track of objects and indices.

I thought this FeatureExtractor would work well because increasing the amount of data we are using to make our prediction will make it slower, however, there will more records to use to make a decision which is good for supervised learning such as this case. The performance is as followed:

Train Accuracy

Accuracy: 6920 / 6920 = 1.000000

Precision (fraction of predicted positives that are correct): 3610 / 3610 = 1.000000; Recall (fraction of true positives predicted correctly): 3610 / 3610 = 1.000000; F1 (harmonic mean of precision and recall): 1.000000

Dev Accuracy

Accuracy: 679 / 872 = 0.778670

Precision (fraction of predicted positives that are correct): 355 / 459 = 0.773420; Recall (fraction of true positives predicted correctly): 355 / 444 = 0.799550; F1 (harmonic mean of precision and recall): 0.786268

Time for training and evaluation: 9.74 seconds

6 Question 6

Between the BetterFeatureExtractor and the Unigram, the performance was approximately the same (around 76-78%) and the Unigram ran significantly faster; hence, I think the unigram performed best of the dev set.

The both Figures 1 and 2 can be seen on the next page containing the graphs for accuracy and loss. In Figure 1, we can see that as the number of epochs increases, the loss decreasing until be reach a minima at which point the curve begins to increase. This can be attributed to over fitting our data which would cause for an increase in error. The minima for 0.10 happens around 19 epochs so around then would be the optimal amount to train our model. Moreover, the amount of error was lowest step size = 0.10 compared to step size 0.99 and 0.02. For learning rate at 0.02, the decrease in error from start is much slower which is why it's higher. For learning rate at 0.99, why are getting an unpredictable amount of error every time. Therefore, 0.10 is a good number

to pick. In Figure 2, it can see that as the model continues to be trained the accuracy increases. After around 19 epochs, this curve begins to flatten which is consistent with our minima in Figure 1. The accuracy on 0.02 was the worst because the slow learning rate did not allow for the weights to be optimized. The accuracy on 0.99 was decent especially in the beginning because it is upddating weights quickly, however, it began to flatten faster. The accuracy for 0.10 was the highest because it allowed for weights to be updated according not too fast nor too slow.

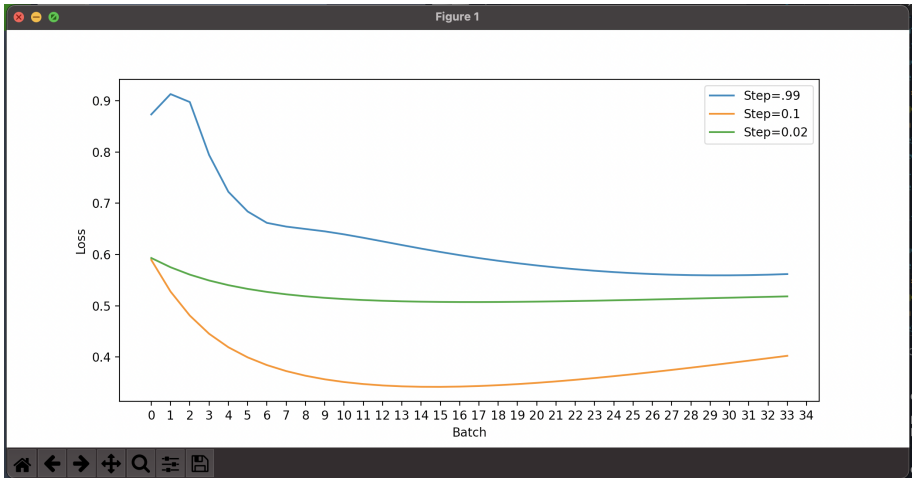

Figure 1: Number of Epochs vs. Loss

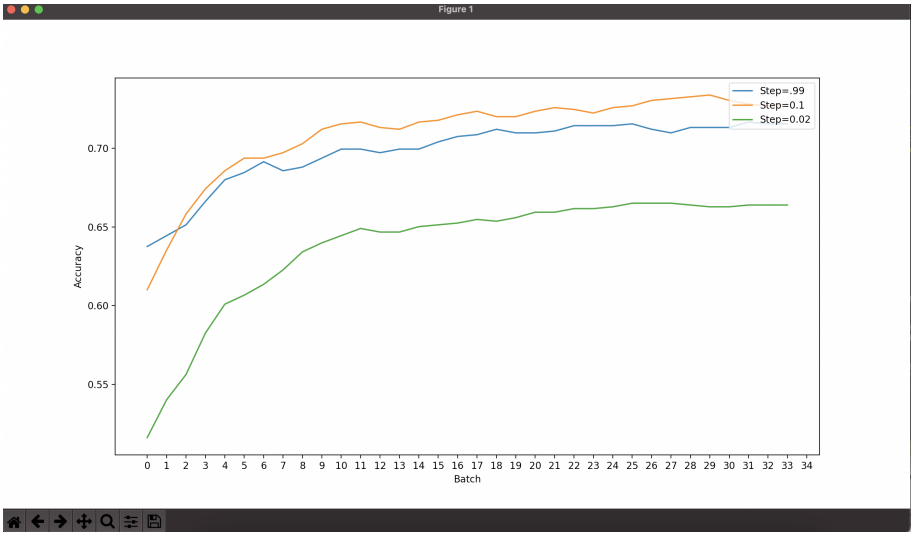

Figure 2: Number of Epochs vs. Accuracy