1. Unigram Perceptron Performance

```
=====Train Accuracy=====
Accuracy: 6906 / 6920 = 0.997977
Precision (fraction of predicted positives that are
correct): 3600 / 3604 = 0.998890;
Recall (fraction of true positives predicted correctly):
3600 / 3610 = 0.997230;
F1 (harmonic mean of precision and recall): 0.998059
=====Dev Accuracy=====
Accuracy: 659 / 872 = 0.755734
Precision (fraction of predicted positives that are
correct): 340 / 449 = 0.757238;
Recall (fraction of true positives predicted correctly):
340 / 444 = 0.765766;
F1 (harmonic mean of precision and recall): 0.761478
Time for training and evaluation: 19.20 seconds
```

2. Try 2 different learning rate (lr) "schedules"

The first one I tried was a constant schedule with the lr set at 1e-3.
The performance for ~50 epochs is listed below.

- Constant Schedule:
  - lr = 1e-3

Train Acc: ~0.99
Dev Acc: ~0.74

Next I tried a cosine learning rate scheduler where the initial learning rate you set is the peak of the cosine function and within N iterations the lr changes and completes half of one period through the cosine function. So after N iteration if it starts at say 1, it will reach ~0, and after another N iterations it'll be back at 1, and so on.
The performance for ~50 epochs is listed below.

  - Cosine Schedule:
    - initial lr = 1e-3
    - iterations to complete one cos period: 6920 (one full iteration through the train set)

Train Acc: ~0.99

Dev Acc: ~0.76

- How do the results compare?

  There is definitely an improvement using the cosine scheduler, it allows for more precision and relaxes the weights in a smoother manner compared to a simple constant scheduler. With the constant scheduler you can't get any smaller precision than what you set your lr to

3. List the 10 words that have the highest positive/lowest negative weight under your model.

   Highest Positive:

   Index of 10 words that have the highest positive weights:

   ```
   [5151  705 1035  880  646  878 2245  945 4289 1427]
   ```

   Associated Weights:
   ```
   [0.01910951 0.01826015 0.01796501 0.01770236
   0.01756759 0.01633033 0.01625786 0.01588733 0.01564778 0.01534047]
   ```

   Words:

   ['half-bad', 'solid', 'enjoyable', 'rare', 'sharp', 'powerful', 'hilarious', 'remarkable', 'assured', 'manages']

   Lowest Negative:

   Index of 10 words that have the lowest negative weights:

   ```
   [360 9790 6406 6720 4543 8736 2325 9280   32  198]
   ```

   Associated Weights:
   ```
   [-0.03054444 -0.02844495 -0.02577993 -0.02038321
   -0.02010306 -0.01921671 -0.01883665 -0.01827004 -0.01820852 -0.01789801]
   ```

   Words: ['?', 'stupid', 'mess', 'suffers', 'mediocre', 'flat', 'pretentious', '\\/', '.', '!']

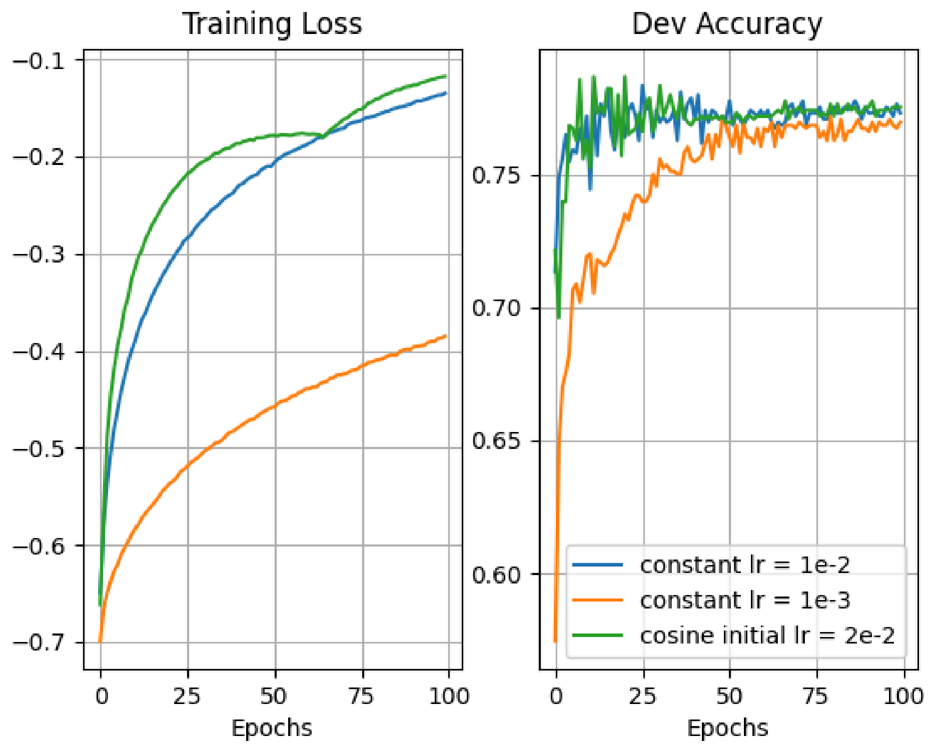
- What trends do you see?

It somewhat performs a word-by-word sentient analysis, the most negative weights are associated with more negative words (stupid, suffers, medoicre, pretentious) that tend to be in reviews that are negative and vice versa. Even though we performed sentient analysis on a more macro level via classifying entire reviews it somewhat learned a simpler and smaller version of the problem which was to count the number of negative/positive words in the review and use the sum to classify the review. Also in particular it seems that the biggest indicator of negative reviews are the punctuation involved. (i.e. “?”, “!”)

4. Compare the training accuracy and development accuracy of the model. What do you see?
   Using results from Q1, train acc = 0.997977 and dev acc = 0.755734
   The accuracy on the dev set was significantly lower than the accuracy on the training set. This implies our model did not generalize to new data very well and it most likely just memorized the training dataset rather than learning deeper underlying intricacies and features of the reviews that could reveal it's overall sentiment. Even though it did learn somewhat to classify sentiment word-by-word, it doesn't actually understand the language on a deeper level. That is how the phrase "That movie was not terrible" is actually a positive sentiment but it has the words "not" and "terrible" which are negative words typically and our classifier would most likely label it as a negative review.

5. Unigram Logistic Regression Performance
```
=====Train Accuracy=====
Accuracy: 6833 / 6920 = 0.987428
Precision (fraction of predicted positives that are
correct): 3561 / 3599 = 0.989442;
Recall (fraction of true positives predicted correctly):
3561 / 3610 = 0.986427;
F1 (harmonic mean of precision and recall): 0.987932
=====Dev Accuracy=====
Accuracy: 677 / 872 = 0.776376
Precision (fraction of predicted positives that are
correct): 360 / 471 = 0.764331;
Recall (fraction of true positives predicted correctly):
360 / 444 = 0.810811;
F1 (harmonic mean of precision and recall): 0.786885
Time for training and evaluation: 16.76 seconds
```

6. Plot loss and dev accuracy for different lrs
   (made loss negative for ease of viewing relationship)



I observe that as the training loss goes down to 0 the dev accuracy tends to go up and that increasing the learning rate greatly increases the slope between epochs to a greater degree (either really positive/really negative). That is, it changes more from one step when lr is higher but it doesn't ALWAYS result in a positive change to the dev acc, it stochasticly navigates the loss function to the lowest point, which has a direct relationship to the TRAINING accuracy but not so strictly to the dev accuracy.

7. Bigram Perceptron Performance

```
=====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: 617 / 872 = 0.707569
Precision (fraction of predicted positives that are
correct): 352 / 515 = 0.683495; Recall (fraction of true
positives predicted correctly): 352 / 444 = 0.792793; F1
(harmonic mean of precision and recall): 0.734098
Time for training and evaluation: 19.16 seconds
```

Bigram Logistic Regression Performance

```
=====Train Accuracy=====
Accuracy: 6911 / 6920 = 0.998699
Precision (fraction of predicted positives that are
correct): 3605 / 3609 = 0.998892; Recall (fraction of true
positives predicted correctly): 3605 / 3610 = 0.998615; F1
(harmonic mean of precision and recall): 0.998753
=====Dev Accuracy=====
Accuracy: 638 / 872 = 0.731651
```

```
Precision (fraction of predicted positives that are
correct): 352 / 494 = 0.712551; Recall (fraction of true
positives predicted correctly): 352 / 444 = 0.792793; F1
(harmonic mean of precision and recall): 0.750533
Time for training and evaluation: 59.10 seconds
```

8. BetterFeatureExtractor Logistic Regression Performance

```
=====Train Accuracy=====
Accuracy: 6733 / 6920 = 0.972977
Precision (fraction of predicted positives that are
correct): 3495 / 3567 = 0.979815; Recall (fraction of true
positives predicted correctly): 3495 / 3610 = 0.968144; F1
(harmonic mean of precision and recall): 0.973945
=====Dev Accuracy=====
Accuracy: 686 / 872 = 0.786697
Precision (fraction of predicted positives that are
correct): 361 / 464 = 0.778017; Recall (fraction of true
positives predicted correctly): 361 / 444 = 0.813063; F1
(harmonic mean of precision and recall): 0.795154
Time for training and evaluation: 15.33 seconds
```

9. My Modification consisted of two things:

The first was to disregard stop words. I experimented with a few different stop words, at first I used the most N common words according to the textbook and used N=10 and N=50, both of which resulted in hardly any difference in the performance. Then I used a hard-coded stop word list from nltk (NLP library) which again resulted in hardly any change. After this I examined the words and chose a distinct list that resulted in the greatest improvement which was this list of stop words:
['the', 'is', 'a', 'it', 'to', 'at', 'of', '.', ',']
I noticed that these appear VERY frequently in many of the examples, non-uniformly compared to other words, and don’t provide much information as to the sentiment of the review,

and thus eliminating them would prevent the algorithm from picking up on coincidental or very niche patterns that appear in the training data.

The second thing I did was simply use the appearance of the word (binary 0/1) instead of the frequency of the word as the value of the input vector. This was because most of the reviews in our data set were very small and not the paragraph-long ones that you would imagine on a typical IMDB website and so I thought there would be again a lot of niche patterns the algorithm could pick up on here that could bias it.

Overall this improved my logistic regression performance to ~78.8% on the dev set, improving it by about ~1% in total.