Natural Language Processing

Assignment 1

Sam Selvaraj (NUID: 002597877)

Q1) Implement unigram perceptron. To receive full credit on this part, you must get at least **74% accuracy** on the development set, and the training and evaluation (the printed time) should take less than **20 seconds** on the autograder. Note that it's fine to use your learning rate schedules from Q2 to achieve this performance. Please list the performance you observe in your writeup, but otherwise you do not need to write anything else to answer this question.

Answer: The unigram perceptron was trained for 10 epochs with a scheduler that reduced the learning rate by 10% after each iteration. The initial learning rate was 0.01. Reported metrics are averaged over 3 runs.

Metrics	Training	Development
Accuracy	97.4%	76.0%
Precision	97.6%	75.7%
Recall	97.6%	77.8%
F1	97.6%	76.7%
Time	2.73	2.73

Q2) Try at least two different "schedules" for the step size for perceptron (having one be the constant schedule is fine). A common one is to decrease the step size by some factor every epoch or few; another is to decrease it like 1/epoch. How do the results change?

Answer: Three different scheduling methods were used:

1) Constant LR: Learning rate was constant throughout the training process.
2) Exponential Decay: The LR is reduced by 10% at every epoch.
3) Inverse Decay: The LR is updated at every iteration by dividing the existing LR by the no. of epochs.

Constant LR and Exponential Decay produced the best results, with Exponential Decay performing slightly better.

	UNIGRAM (LR = 0.01 | 20 epochs)			
	Perceptron (dev)		Logistic Regression (dev)	
	Scheduler		Scheduler	
Metrics	**No**	**Yes**	**No**	**Yes**
Accuracy	75.6%	77.2%	77.6%	77.2%
Precision	77.2%	78.0%	79.9%	75.8%
Recall	73.9%	76.8%	75.0%	81.1%
F1	75.5%	77.4%	77.4%	78.3%
Time (s)	5.10	5.16	4.83	4.80

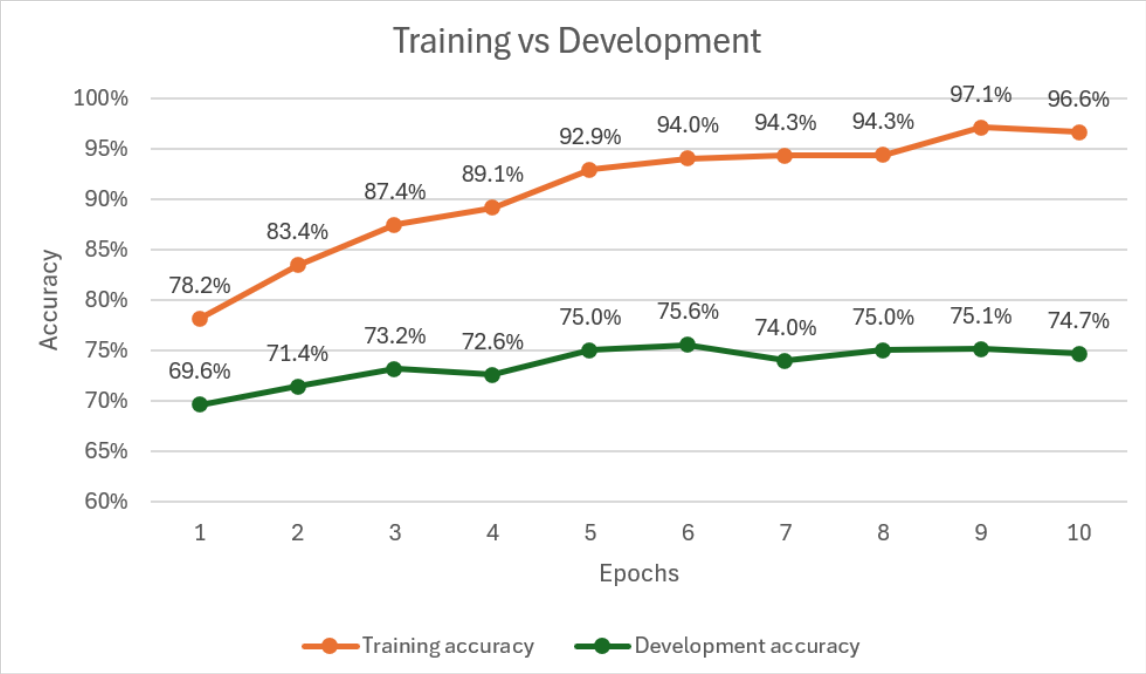
Q3) List the 10 words that have the highest positive weight under your model and the 10 words with the lowest negative weight. What trends do you see?

Answer: We can clearly observe that the weights of positive words are positive, and the weights of negative words are negative. It is fascinating to observe that the model learned this association on its own.

Top 10 positive words		Top 10 negative words	
Word	**Weights**	**Word**	**Weights**
powerful	0.1251	stupid	-0.1463049
spirit	0.119907521	tv	-0.109681105
manages	0.11060441	suffers	-0.106535205
remarkable	0.10629	mess	-0.1051
rare	0.104821441	unfortunately	-0.1051
hilarious	0.101912846	flat	-0.10070441
appealing	0.101339957	worst	-0.098943254
solid	0.10060441	none	-0.098815777
intriguing	0.100201136	lazy	-0.097861
refreshing	0.0975659	unfunny	-0.0951

Q4) Compare the training accuracy and development accuracy of the model. What do you see? Explain in 1-3 sentences what is happening here.

Answer: Using the data below, we can easily infer that the model tends to overfit the training data quite quickly as the no. of epochs increase. The development accuracy of the model stabilizes around 75% in comparison to the training accuracy which seems to be increasing and stabilizing at around 97%.



PERCEPTRON UNIGRAM		
Epochs	**Training accuracy**	**Development accuracy**
1	78.2%	69.6%
2	83.4%	71.4%
3	87.4%	73.2%
4	89.1%	72.6%
5	92.9%	75.0%
6	94.0%	75.6%
7	94.3%	74.0%
8	94.3%	75.0%
9	97.1%	75.1%
10	96.6%	74.7%

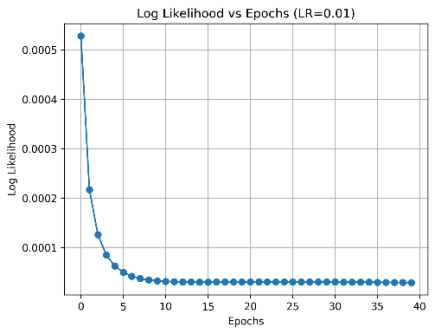
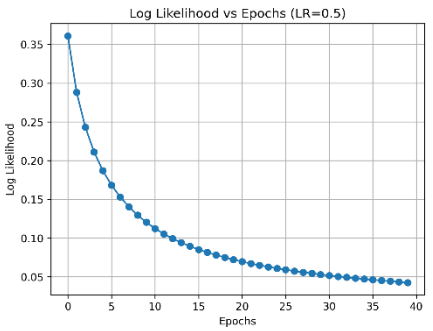
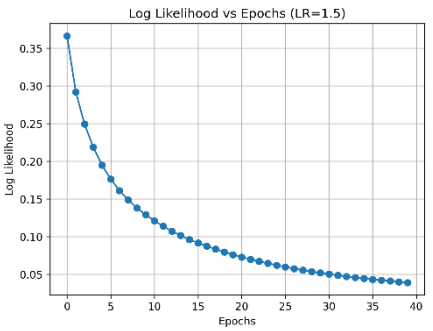
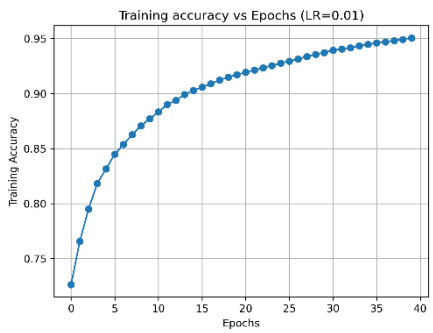
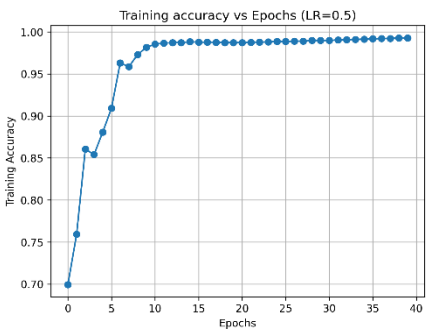
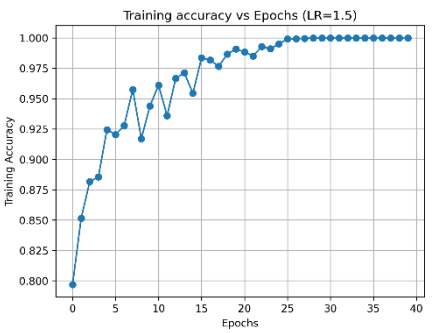
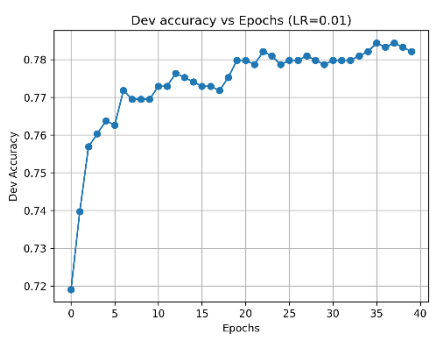
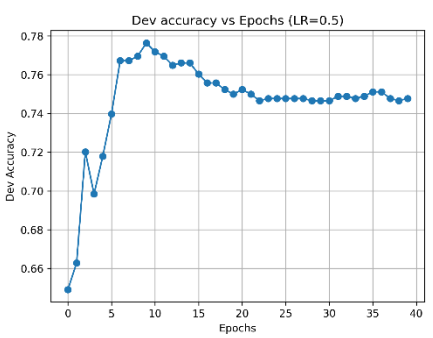
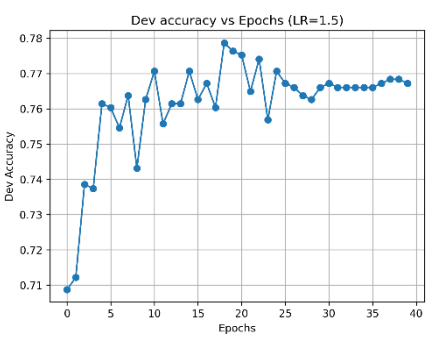
Q5) Implement logistic regression. Report your model's performance on the dataset. You must get at least **77% accuracy** on the development set and it must run in less than **20 seconds** on the autograder.

Answer: The unigram LR was trained on 10 epochs and an exponential scheduler was used which reduced learning rate by 10% at each iteration. The initial learning rate was set at 0.01. An average of 3 iterations was taken for calculating the metrics.

Metrics	Training	Development
Accuracy	87.6%	78.0%
Precision	89.0%	79.0%
Recall	87.0%	77.5%
F1	88.0%	78.2%
Time	2.67	2.67

Q6) Plot (using matplotlib or another tool) the training objective (dataset log likelihood) and development accuracy of logistic regression vs. number of training iterations for a couple of different step sizes. What do you observe?

Answer: The table shows that the log likelihood reaches its minimum fastest when the learning rate is set to 0.01. With higher learning rates, the model takes longer to converge. In terms of training and development accuracy, 0.01 provides the best results, as the graphs become increasingly erratic at larger learning rates. At 1.5, both training and development accuracy fluctuate heavily across epochs.

Learning Rate = 0.01	Learning rate = 0.5	Learning rate = 1.5
		
		
		

Q7) Implement and experiment with BigramFeatureExtractor. Bigram features should be indicators on adjacent pairs of words in the text. Report the performance of your perceptron classifier and of logistic regression with this feature set.

Answer: Bigram features were created by assigning indices to adjacent word pairs. After 10 epochs, bigram features led to overfitting: training accuracy approached 100%, while development accuracy was lower than with unigrams.

	BIGRAM (LR = 0.01 w Scheduler)			
	Perceptron		Logistic Regression	
Metrics	**Training**	**Dev**	**Training**	**Dev**
Accuracy	100.0%	71.2%	96.9%	71.9%
Precision	100.0%	71.8%	97.0%	69.2%
Recall	99.9%	71.6%	97.1%	80.6%
F1	100.0%	71.7%	97.0%	74.5%
Time	2.92		2.42	

Q8) Experiment with at least one feature modification in BetterFeatureExtractor. Try it out with either algorithm (though it should work with both). Report the performance it gives. Things you might try: other types of *n*-grams, *tf-idf* weighting, clipping your word frequencies, discarding rare words, discarding stopwords, etc. **Your final code here should be whatever works best (even if that's one of your other feature extractors)**. When used with logistic regression, this model should train and evaluate in at most 60 seconds. **This feature modification should not just consist of combining unigrams and bigrams.**

Answer: Explored multiple approaches for feature extraction given below:

1) Stopwords - Removed stopwords from sentences.
2) Special characters – Eg. don't -> dont
3) Lemmatizing – Converting words to their base form. Eg. running -> run
4) TF-IDF – Weighing a word's importance in a document by combining how often it appears (TF) with how unique it is across all documents (IDF).

After careful consideration and examination, it was observed that TF-IDF and lemmatization didn't provide any benefits [Implementation in UnperformingBetterFeatureExtractor()]. In fact, the performance was lower than the simple unigram extractor. Hence, the final BetterFeatureExtractor() has only stopword removal and the unigram implementation. This extractor increased the accuracy slightly.

	BETTER (LR = 0.01 | 10 epochs)	
Metrics	**Perceptron**	**Logistic Regression**
Accuracy	76.0%	77.8%
Precision	78.9%	78.3%
Recall	72.3%	77.9%
F1	75.4%	78.1%
Time	4.36	3.12

Q9) Describe your feature modification and give one reason you might be seeing the performance delta (or lack of delta) from this modification.

Answer:

The feature modification used stopword removal, special character removal, **lemmatization** and **TF-IDF weighting** together. Lemmatization converted words to their base forms (*running* → *run*) to reduce feature redundancy, while TF-IDF weighted words based on their importance within documents versus across the entire corpus.

There can be multiple reasons for lemmatization and TF-IDF not working.

- It is possible that reducing the words to their base form could cause the loss of sentiment from the words. Eg. “The match was better than I expected” vs “The match was good”. The word “better” is most likely converted to “good”. This causes loss of the comparative aspect of better.
- Perceptron and logistic regression are very simple linear models; they sometimes work better with raw counts than with normalized TF-IDF.
- By down-weighting frequent but sentiment-rich words such as *not* or *never*, TF-IDF can weaken important signals. In addition, the transformation makes the features sparse, which can make it harder for simple models to learn effectively without stronger regularization.