Oregon State University

AI534 - MACHINE LEARNING

HW2: Perceptron for Sentiment Classification

AUTHORS

Melek Derman - 934382167

November 5, 2024

Contents

1 Part 0: Sentiment Classification Task and Dataset

0.1: Why is each of these steps necessary or helpful for machine learning?

The preprocessing steps are essential to ensure uniformity and eliminate redundant data, enhancing the algorithm's efficiency. Converting all words to lowercase ensures consistency across terms that might appear in different cases, reducing redundancy in the dataset. Separating punctuation prevents it from being processed as part of words, allowing for more accurate interpretation of its syntactic and semantic roles within sentences. For instance, separating the possessive 's in children's allows children to be recognized as a distinct word, reducing vocabulary size and facilitating correct interpretation of the possessive form. Standardizing single (' ') and double (" ") quotation marks, which often vary across data sources, ensures uniformity and prevents misinterpretation or compatibility issues when using different Python libraries. Moreover, excluding Spanish reviews, which are few in number, avoids unnecessary data volume increase and potential processing slowdowns.

2 Part 1: Naive Perceptron Baseline

1.1 Take a look at svector.py, and briefly explain why it can support addition, subtraction, scalar product, dot product, and negation (0.5 pts)

The `svector.py` file defines a dictionary-like class, named `svector`, which incorporates various vector operations through specific magic methods. These methods enable the `svector` class to perform mathematical operations commonly required for vector manipulation, including addition, subtraction, scalar multiplication, dot product calculation, and negation.

Implemented Magic Methods in the `svector` *Class*

__add__ function returns a new vector that is the sum of two vectors, self and other.

__sub__ function returns the difference between two vectors (self and other).

__mul__ function facilitates the multiplication of the self vector by a scalar c.

__dot__ function computes the dot product. It first identifies the shorter vector and processes it accordingly, reducing iteration number.

__neg__ function converts all elements in the vector to their negative counterparts (by multiplying each element by -1) and returns a new `svector()`.

1.2: Take a look at train.py, and briefly explain train() and test() functions (0.5 pts).

`test()`*function:*

The `test` function calculates the error rate on a development dataset. The labels and words are read from `devfile` and enumerated, starting the index at 1. During the error calculation, each label (which is $+1$ for positive labels and -1 for negative labels, as set in the `read_from` function) is multiplied by the prediction value calculated with `model.dot(make_vector(words))` for the given words. If the result is less than or equal to 0, it is considered an error, and the error count (`err`) is incremented. Finally, the function returns the error rate (`err/i`), which is the ratio of the total error count to the number of examples.

`train()`*function:*

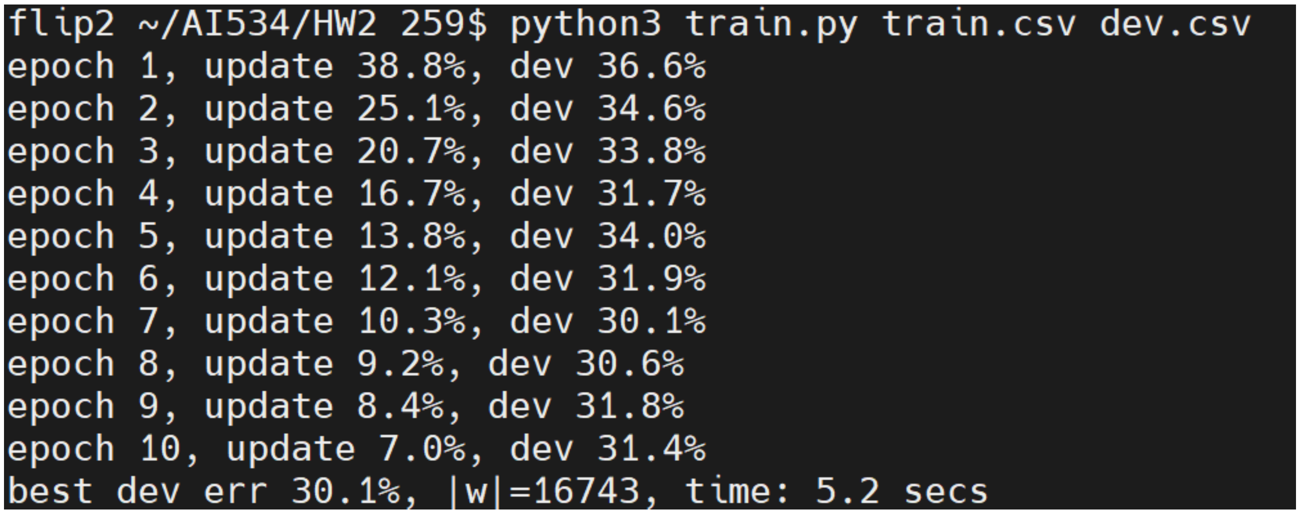
The `train()` function is used to train for the Perceptron model. A timer is started, and the model is initialized as an `svector()`. The function iterates over the specified number of epochs. The label and words read from `trainfile` are enumerated, allowing the function to loop over each entry. After the word list is converted into a vector, if the model makes an incorrect prediction (i.e., the product of the label and the prediction is less than or equal to 0), the update count is incremented by 1, and the model is updated by adding `label * sent`. At the end of each epoch, the model is tested on the development set, and the error rate and update percentage for that epoch are printed. After all epochs are completed, the best error (`best_err`), model length, and elapsed time are displayed.

1.3: There is one thing missing in my train.py: the bias dimension! Try add it. How did you do it? (Hint: by adding bias or <bias>?) Did it improve error on dev? (0.5 pts)

It can be done by adding `v['<bias>'] = 1` to the `make_vector` function. `<bias>` is used so that the model can recognize and fit the bias term. Adding the bias dimension reduced the best dev error from 30.1 to 28.9.

without `<bias>` *dimension:*

As can be seen from the Figure 1, `|w|` = 16743 and `best_dev_err` = 30.1%. The calculation took 5.2 seconds.

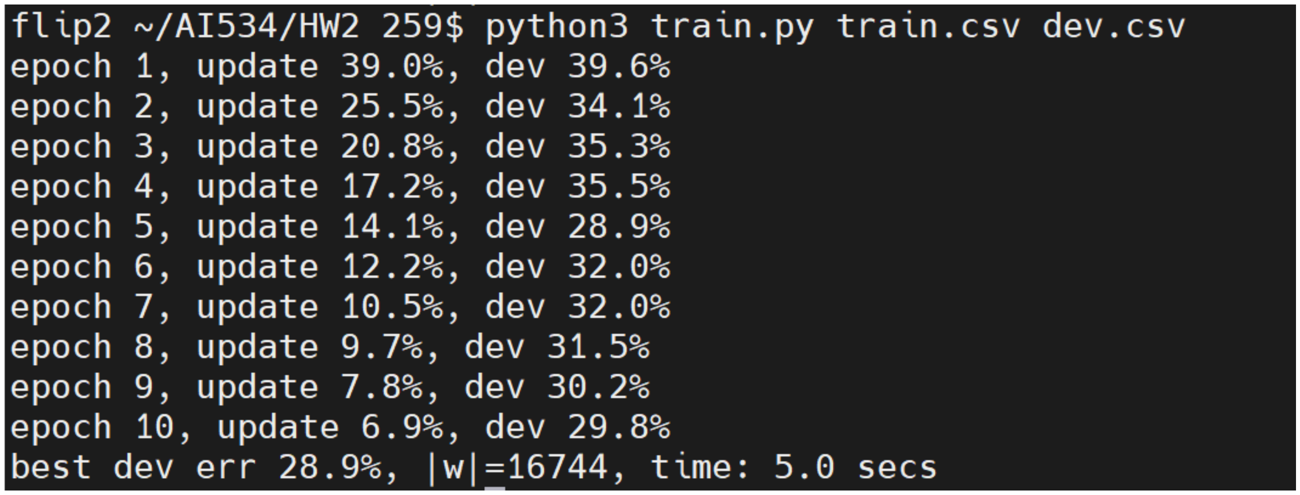


```
flip2 ~/AI534/HW2 259$ python3 train.py train.csv dev.csv
epoch 1, update 38.8%, dev 36.6%
epoch 2, update 25.1%, dev 34.6%
epoch 3, update 20.7%, dev 33.8%
epoch 4, update 16.7%, dev 31.7%
epoch 5, update 13.8%, dev 34.0%
epoch 6, update 12.1%, dev 31.9%
epoch 7, update 10.3%, dev 30.1%
epoch 8, update 9.2%, dev 30.6%
epoch 9, update 8.4%, dev 31.8%
epoch 10, update 7.0%, dev 31.4%
best dev err 30.1%, |w|=16743, time: 5.2 secs
```

Figure 1: Naive Perceptron Algorithm Results without a Bias Dimension

with `<bias>` *dimension:*

As shown in Figure 2, with a bias dimension, `|w|` = 16744 and `best_dev_err` = 28.9%. The calculation took 5.0 seconds. While there is no significant change in terms of time, the dev error rate decreased by 1.2 %.

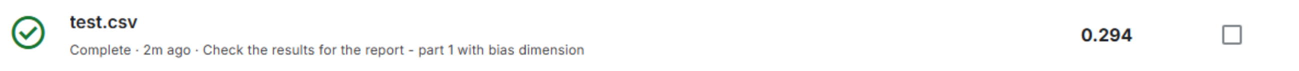


```
flip2 ~/AI534/HW2 259$ python3 train.py train.csv dev.csv
epoch 1, update 39.0%, dev 39.6%
epoch 2, update 25.5%, dev 34.1%
epoch 3, update 20.8%, dev 35.3%
epoch 4, update 17.2%, dev 35.5%
epoch 5, update 14.1%, dev 28.9%
epoch 6, update 12.2%, dev 32.0%
epoch 7, update 10.5%, dev 32.0%
epoch 8, update 9.7%, dev 31.5%
epoch 9, update 7.8%, dev 30.2%
epoch 10, update 6.9%, dev 29.8%
best dev err 28.9%, |w|=16744, time: 5.0 secs
```

Figure 2: Naive Perceptron Algorithm Results with a Bias Dimension

1.4: Using your best model (in terms of dev error rate), predict the semi-blind test data, and submit it to Kaggle (follow instructions from Part 5). What are your error rate and ranking on the public leaderboard? Take a screenshot. Hint: your public error rate should be $\sim$ 31%. (0.5 pts)

My error rate was 29.4% (Figure 3), but I do not know my leaderboard score for this entry because my previous entries had lower error rates than this one. (For an unknown reason, all of my initial entries had an error rate of 27.0%.)



test.csv
Complete · 2m ago · Check the results for the report - part 1 with bias dimension
0.294

Figure 3: Kaggle Score of the Predictions for Naive Perceptron Algorithm with Bias Dimension

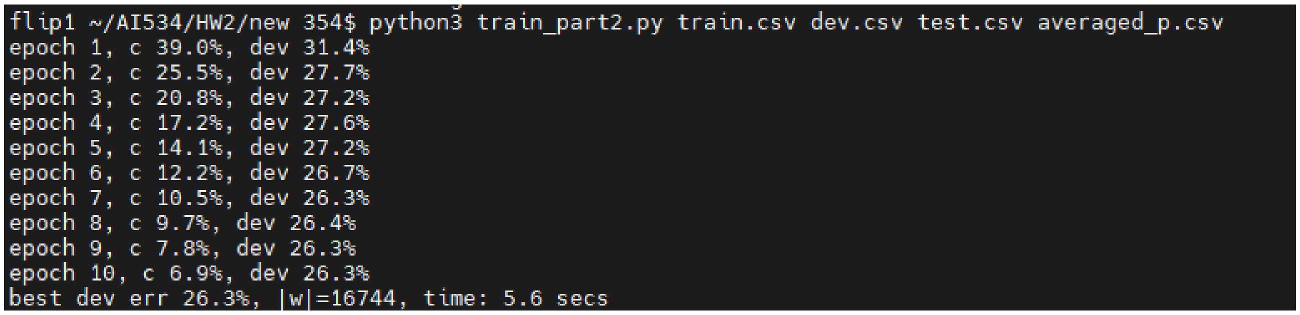
1.5 Wait a second, I thought the data set is already balanced (50% positive, 50% negative). I remember the bias being important in highly unbalanced data sets. Why do I still need to add the bias dimension here?? (0.5 pts)

Although the data set is balanced at a ratio of 50% positive and 50% negative, the data may not be symmetric through origin. Without the bias term, the decision boundary would separate the positive and negative points through the origin; however, real data may not be this symmetric. The bias term allows the decision boundary to be adjusted more accurately for cases where the data is not separable through the origin.

3 Part 2: Average Perceptron

2.1: Train for 10 epochs and report the results. Did averaging improve the dev error rate? (Hint: should be around 26%). Did it also make dev error rates more stable? (1.5 pts)

The dev error rate for the average perceptron was 26.3% (Figure 4). While the dev error rate showed large increases and decreases through epochs in the naive perceptron, these differences were smaller in the average perceptron. There was no significant change in computation time.



```
flip1 ~/AI534/HW2/new 354$ python3 train_part2.py train.csv dev.csv test.csv averaged_p.csv
epoch 1, c 39.0%, dev 31.4%
epoch 2, c 25.5%, dev 27.7%
epoch 3, c 20.8%, dev 27.2%
epoch 4, c 17.2%, dev 27.6%
epoch 5, c 14.1%, dev 27.2%
epoch 6, c 12.2%, dev 26.7%
epoch 7, c 10.5%, dev 26.3%
epoch 8, c 9.7%, dev 26.4%
epoch 9, c 7.8%, dev 26.3%
epoch 10, c 6.9%, dev 26.3%
best dev err 26.3%, |w|=16744, time: 5.6 secs
```

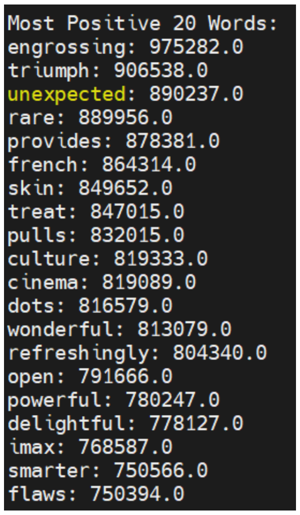
Figure 4: Average perceptron algorithm results.

2.2: Did smart averaging slow down training? (1 pt)

There was no significant change in computation time (0.6 seconds slower than the naive perceptron calculations).

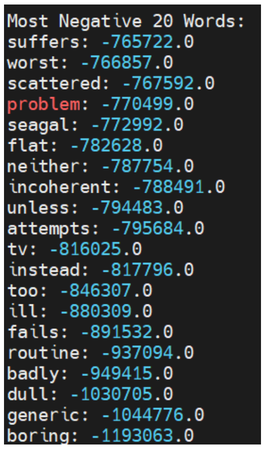
2.3: What are the top 20 most positive and top 20 most negative features? Do they make sense? (1 pt)

The positive and negative words listed generally appear to make sense. Among the positive words are clearly favorable terms such as "engrossing," "triumph," and "delightful," although there are exceptions like "flaws." In the negative features, words like "suffers," "worst," "fails," and "boring" convey distinctly negative meanings, while terms such as "tv" and "too" are more neutral. Overall, both lists are consistent, though exceptions are present in each.



```
Most Positive 20 Words:
engrossing: 975282.0
triumph: 906538.0
unexpected: 890237.0
rare: 889956.0
provides: 878381.0
french: 864314.0
skin: 849652.0
treat: 847015.0
pulls: 832015.0
culture: 819333.0
cinema: 819089.0
dots: 816579.0
wonderful: 813079.0
refreshingly: 804340.0
open: 791666.0
powerful: 780247.0
delightful: 778127.0
imax: 768587.0
smarter: 750566.0
flaws: 750394.0
```

Figure 5: The Top 20 Most Positive Feature



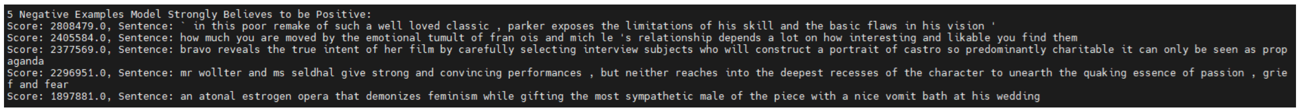
```
Most Negative 20 Words:
suffers: -765722.0
worst: -766857.0
scattered: -767592.0
problem: -770499.0
seagal: -772992.0
flat: -782628.0
neither: -787754.0
incoherent: -788491.0
unless: -794483.0
attempts: -795684.0
tv: -816025.0
instead: -817796.0
too: -846307.0
ill: -880309.0
fails: -891532.0
routine: -937094.0
badly: -949415.0
dull: -1030705.0
generic: -1044776.0
boring: -1193063.0
```

Figure 6: The Top 20 Most Negative Feature

Figure 7: Most positive and most negative features

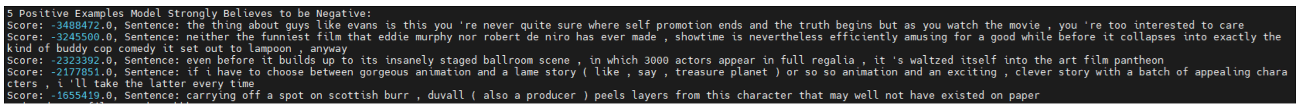
2.4: Show 5 negative examples in dev where your model most strongly believes to be positive. Show 5 positive examples in dev where your model most strongly believes to be negative. What observations do you get? (2 pts)

The model seems to make errors in complex and long sentences, in texts with irony, in sentences with nuanced meanings, or those referencing other films.



```
5 Negative Examples Model Strongly Believes to be Positive:
Score: 2808479.0, Sentence: ` in this poor remake of such a well loved classic , parker exposes the limitations of his skill and the basic flaws in his vision '
Score: 2405584.0, Sentence: how much you are moved by the emotional tumult of fran ois and mich le 's relationship depends a lot on how interesting and likable you find them
Score: 2377569.0, Sentence: bravo reveals the true intent of her film by carefully selecting interview subjects who will construct a portrait of castro so predominantly charitable it can only be seen as prop
aganda
Score: 2296951.0, Sentence: mr wollter and ms seldhal give strong and convincing performances , but neither reaches into the deepest recesses of the character to unearth the quaking essence of passion , grie
f and fear
Score: 1897881.0, Sentence: an atonal estrogen opera that demonizes feminism while gifting the most sympathetic male of the piece with a nice vomit bath at his wedding
```

Figure 8: Five Negative examples in dev set where your model most strongly believes to be positive.

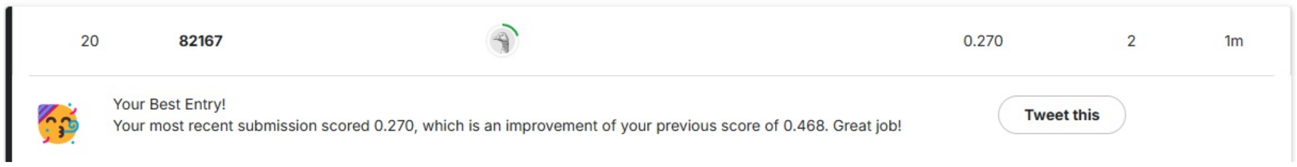


```
5 Positive Examples Model Strongly Believes to be Negative:
Score: -3488472.0, Sentence: the thing about guys like evans is this you 're never quite sure where self promotion ends and the truth begins but as you watch the movie , you 're too interested to care
Score: -3245500.0, Sentence: neither the funniest film that eddie murphy nor robert de niro has ever made , showtime is nevertheless efficiently amusing for a good while before it collapses into exactly the
kind of buddy cop comedy it set out to lampoon , anyway
Score: -2323392.0, Sentence: even before it builds up to its insanely staged ballroom scene , in which 3000 actors appear in full regalia , it 's waltzed itself into the art film pantheon
Score: -2177851.0, Sentence: if i have to choose between gorgeous animation and a lame story ( like , say , treasure planet ) or so so animation and an exciting , clever story with a batch of appealing chara
cters , i 'll take the latter every time
Score: -1655419.0, Sentence: carrying off a spot on scottish burr , duvall ( also a producer ) peels layers from this character that may well not have existed on paper
```

Figure 9: Five positive examples in dev where your model most strongly believes to be positive.

2.5: Again, using your new best model (in terms of dev error rate), predict the semi-blind test data, and submit it to Kaggle. What are your new error rate and ranking on the public leaderboard? Take a screenshot. Hint: your public error rate should improve to $\sim 27\%$. (0.5 pts)

My error rate was 27%, and my public leaderboard ranking was 20.


20 82167 0.270 2 1m

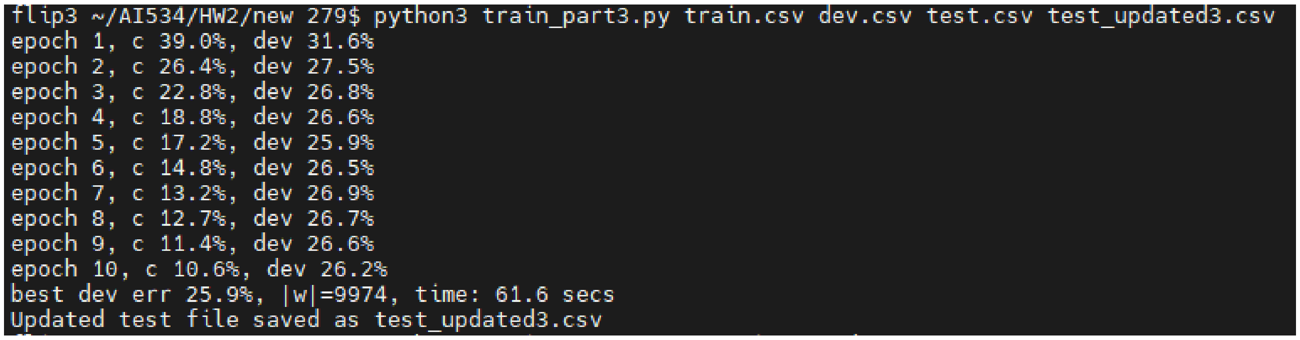
Your Best Entry!
Your most recent submission scored 0.270, which is an improvement of your previous score of 0.468. Great job!

Figure 10: Ranking on the public leaderboard and public error of my results.

4 Part 3: Pruning the Vocabulary

3.1: Try neglecting one-count words in the training set during training. Did it improve the dev error rate? (Hint: should help a little bit, error rate lower than 26%). (1 pt)

Neglecting one-count words from the training set decreased the dev error rate from 26.3% to 25.9%.



```
flip3 ~/AI534/HW2/new 279$ python3 train_part3.py train.csv dev.csv test.csv test_updated3.csv
epoch 1, c 39.0%, dev 31.6%
epoch 2, c 26.4%, dev 27.5%
epoch 3, c 22.8%, dev 26.8%
epoch 4, c 18.8%, dev 26.6%
epoch 5, c 17.2%, dev 25.9%
epoch 6, c 14.8%, dev 26.5%
epoch 7, c 13.2%, dev 26.9%
epoch 8, c 12.7%, dev 26.7%
epoch 9, c 11.4%, dev 26.6%
epoch 10, c 10.6%, dev 26.2%
best dev err 25.9%, |w|=9974, time: 61.6 secs
Updated test file saved as test_updated3.csv
```

Figure 11: Average perceptron algorithm results after neglecting one-count words.

3.2: Did your model size shrink, and by how much? (Hint: should almost halve). Does this shrinking help prevent overfitting? (0.25 pts)

This reduced the model size from 16,744 to 9,974. Yes, the model size did shrink. The reduction was close to half, dropping from 16,744 to 9,974 features. This shrinking can help prevent overfitting, as it may reduce overfitting by removing noise and allowing the model to focus on more informative data.

3.3: Did update % change? Does the change make sense? (0.25 pts)

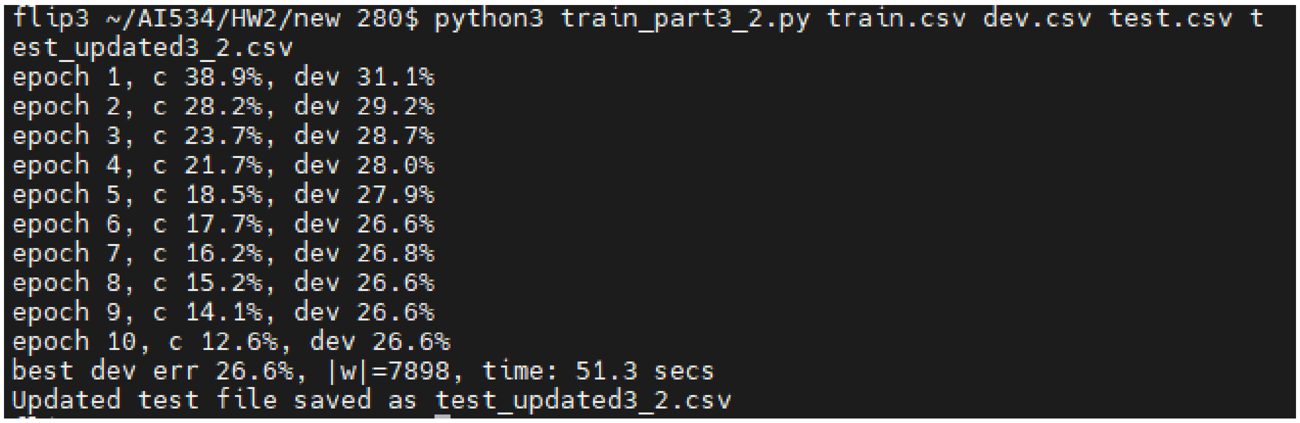
Yes, the update % changed. There could be several reasons for this. Some possible reasons include a reduction in the model's flexibility due to the decreased model size or an increase in the weights of commonly used words, such as "the" or "and," as a result of information loss.

3.4: Did the training speed change? (0.25 pts)

Yes, the training speed increased significantly, rising from 5.3 seconds to 61.6 seconds â approximately 12 times.

3.5: What about further pruning two-count words (words that appear twice in the training set)? Did it further improve dev error rate? (0.75 pts)

Neglecting two-count words along with one-count words from the training set increased the dev error rate by 0.7%.

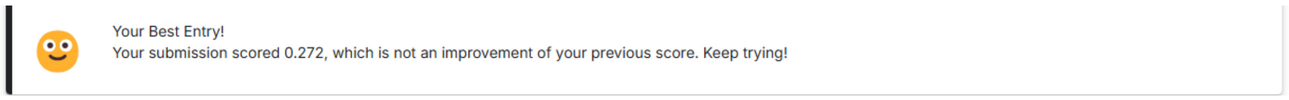


```
flip3 ~/AI534/HW2/new 280$ python3 train_part3_2.py train.csv dev.csv test.csv t
est_updated3_2.csv
epoch 1, c 38.9%, dev 31.1%
epoch 2, c 28.2%, dev 29.2%
epoch 3, c 23.7%, dev 28.7%
epoch 4, c 21.7%, dev 28.0%
epoch 5, c 18.5%, dev 27.9%
epoch 6, c 17.7%, dev 26.6%
epoch 7, c 16.2%, dev 26.8%
epoch 8, c 15.2%, dev 26.6%
epoch 9, c 14.1%, dev 26.6%
epoch 10, c 12.6%, dev 26.6%
best dev err 26.6%, |w|=7898, time: 51.3 secs
Updated test file saved as test_updated3_2.csv
```

Figure 12: Average perceptron algorithm results after neglecting one-count and two-count words.

3.6: Using your current best model (in terms of dev error rate) from this part, predict the semi-blind test data, and submit it to Kaggle. What are your new error rate and ranking on the public leaderboard? Take a screenshot. Hint: your public error rate should still be $\sim 27\%$ and it is not necessarily lower than Part 2. (0.5 pts)

My error rate was 27.2%, and my public leaderboard ranking did not change due to the previous 0.270 score.



Your Best Entry!
Your submission scored 0.272, which is not an improvement of your previous score. Keep trying!

Figure 13: Average perceptron algorithm results after neglecting one-count and two-count words.

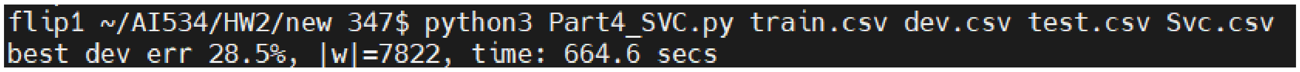
5 Part 4: Try some other learning algorithms with sklearn

4.1: Which algorithm did you try? What adaptations did you make to your code to make it work with that algorithm? What specific setting (e.g., vocabulary pruning) did you use? (0.5 pts)

- Without removing any words:
  - Logistic Regression Error Rate: 30.00%
  - SVC Error Rate: 27.80%
  - Naive Bayes Error Rate: 27.50%
  - Bernoulli Naive Bayes Error Rate: 27.00%
  - Complement Naive Bayes Error Rate: 27.50%
  - Stochastic Gradient Descent Classifier Error Rate: 29.60%
  - Linear SVM Error Rate: 30.70%
  - Random Forest Error Rate: 33.70%
  - Multi-layer Perceptron Error Rate: 30.00%
- After removing the 10 most frequent words: “the", “,", “a", “and", “of", “to", “’s", “is", “it", “that"
  - Logistic Regression Error Rate: 30.80%
  - SVC Error Rate: 28.50%
  - Naive Bayes Error Rate: 27.50%
  - Bernoulli Naive Bayes Error Rate: 27.80%
  - Complement Naive Bayes Error Rate: 27.50%
  - Stochastic Gradient Descent Classifier Error Rate: 29.60%
  - Linear SVM Error Rate: 31.20%
  - Random Forest Error Rate: 32.30%
  - Multi-layer Perceptron Error Rate: 29.70%

4.2: What‘s the dev error rate(s) and running time? (0.5 pts)

The best model I tried in terms of Kaggle public score was SVC, with a dev error rate of 28.5% and a running time of 664.6 seconds.

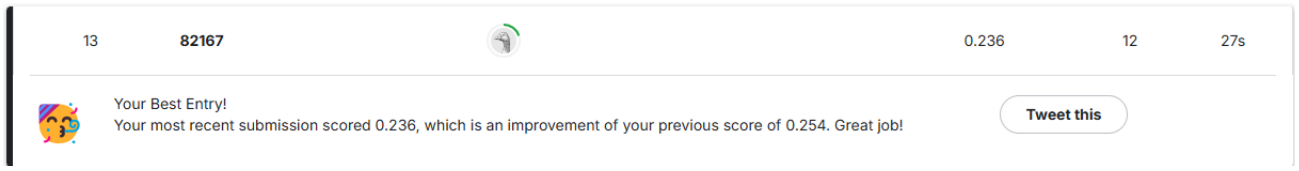


```
flip1 ~/AI534/HW2/new 347$ python3 Part4_SVC.py train.csv dev.csv test.csv Svc.csv
best dev err 28.5%, |w|=7822, time: 664.6 secs
```

Figure 14: SVC results.

4.3: Submit your best prediction to Kaggle. What was your public score and rank? For example, our TA Zetian got ∼ 24% (quite a bit better than 27%). (0.5 pts)

My public score was 0.236, and my ranking was 13.



```
13    82167    0.236    12    27s
```

Your Best Entry!
Your most recent submission scored 0.236, which is an improvement of your previous score of 0.254. Great job!

Figure 15: Ranking on the public leaderboard and public error of my SVC results.

4.4: What did you learn in terms of the comparison between averaged perceptron and these other (presumably more popular and well-known) learning algorithms? (0.5 pts)

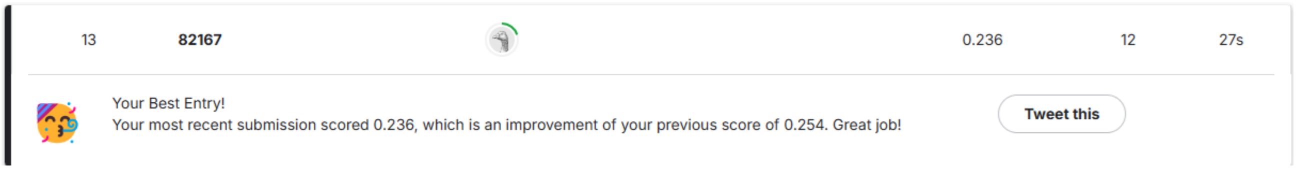
In comparing the averaged perceptron algorithm with other popular learning algorithms, I observed that the averaged perceptron performed quite well, achieving an dev error rate of 26.3%, the lowest among all tested models. The algorithms that came closer to the performance of the averaged perceptron included SVC (27.80%), Naive Bayes (27.50%), and Bernoulli Naive Bayes (27.00%). On the other hand, while the average perceptron achieved a 0.27 rank in the Kaggle competition, the best-ranking model was SVC.

6 Part 5: Deployment

5.1: What's the dev error rate(s) and how did you achieve it? (0.5 pts)

For my best model in terms of Kaggle public score, dev error rate was 27.80%.

5.2: Submit your best prediction to Kaggle. What was your overall best public score and final rank? (0.5 pts)



13 82167 0.236 12 27s

Your Best Entry!
Your most recent submission scored 0.236, which is an improvement of your previous score of 0.254. Great job!

Figure 16: Best ranking on the public leaderboard and public error of my SVC results.

7 Part 6: Debriefing

1. Approximately how many hours did you spend on this assignment?

I spent approximately 20 hours on this assignment.

2. Would you rate it as easy, moderate, or difficult?

I would rate this assignment as difficulty.

3. Did you work on it mostly alone, or mostly with other people?

I worked on it independently.

4. How deeply do you feel you understand the material it covers (0%-100%)?

My rate of deeply understanding the material is 70%.

5. Any other comments?

Thank you!