AI 534 Machine Learning HW2

Submitted by: Apoorv Malik (malikap@oregonstate.edu)

0. Sentiment Classification Task and Dataset

Each of these preprocessing steps in a machine learning pipeline serve specific purposes that contribute to the model's ability to learn from the data effectively.

- **Lowercasing the words** help reduce the overall size of the vocabulary that the model needs to learn, which can improve both performance and speed.
- **Punctuation Splitting** can help ML models differentiate between the end of a sentence, a possessive noun, a contracted verb, etc which could be crucial for understanding the sentiment.
- **Morpheme Splitting** helps in handling negations correctly, which is essential in sentiment analysis since negations can invert the sentiment of a sentence.
- **Quote Normalization** ensures that quoted text is not misinterpreted by the model and can help in identifying emphasis in sentiment analysis.

1. Naive Perceptron Baseline

1. By defining these special methods (__add__, __iadd__, __sub__, __isub__, __mul__, __rmul__, dot, __neg__), the **svector** class enables the use of +, +=, -, -=, *, and unary - operators, respectively, making it act like a mathematical vector. The implementation takes advantage of the sparse nature of the vectors to optimize storage and computation. The dot method implements the dot product of two vectors. It chooses the vector with fewer elements to iterate over for efficiency, and it computes the sum of products of the corresponding elements of the two vectors.

2. The **train** function creates a variable **model** of type **svector** which represents the perceptron. Now, each sentence in the training data is tokenized (each word is a dimension which has magnitude 1) and converted to **svector** which represents a training point. Now, it follows the perceptron algorithm for updating the model weights by performing a **dot product** between the **model vector** and **training point vector**. It then prints the **dev error** after each epoch and also prints the **best dev error** after the last epoch.

   The **test** function runs a given **model** on the development set and returns the error. It does the same operation as train function for converting each sentence to **svector**, and then computing the **dot product** between the **model** and the **dev point vector**.

3. I used <*bias*> keyword because there were some training points where the word *bias* was a dimension. I added the bias dimension to the **model** by instantiating it like this model = svector({'<bias>': 0})
   I then added the bias dimension in the **make_vector** function by instantiating a new vector (for a sentence) like this: v['<bias>'] = 1
   After adding the bias dimension to both model and points, the dev error improved slightly. The new dev error is now **28.9%**.

4. The bias term is not solely for addressing data imbalance. The bias term allows the decision boundary to be offset from the origin. Without a bias, the decision boundary would always pass through the origin, which might not be the optimal split, especially if the data is not symmetrically distributed around the origin. It also effectively adds an extra dimension to the feature space, allowing the learning algorithm to learn a decision boundary that is not constrained to intersect the origin of the feature space.

2. Average Perceptron

1. These are the results after training for 10 epochs. Yes, averaging improved the dev error rate significantly, it is now **26.3%**. The dev error rates are more stable now with less variation at each epoch. It usually hovers around 27% for each epoch (except epoch 1).

```
~/D/O/C/A/h/hw2-data on main × time python3 train.py train.txt dev.txt 2
epoch 1, update 39.0%, dev 31.4%
epoch 2, update 25.5%, dev 27.7%
epoch 3, update 20.8%, dev 27.2%
epoch 4, update 17.2%, dev 27.6%
epoch 5, update 14.1%, dev 27.2%
epoch 6, update 12.2%, dev 26.7%
epoch 7, update 10.5%, dev 26.3%
epoch 8, update 9.7%, dev 26.4%
epoch 9, update 7.8%, dev 26.3%
epoch 10, update 6.9%, dev 26.3%
best dev err 26.3%, |w|=15806, time: 0.8 secs

_____________________________________________________
Executed in    1.41 secs    fish           external
   usr time    2.56 secs   57.00 micros    2.56 secs
   sys time    0.26 secs  509.00 micros    0.26 secs
```

2. The naive perceptron training finished in **2.35 secs**, and the average perceptron training finished in **2.56 secs**. So, yes, the averaging slowed down training by about **9%**, which is not very significant.

(Please move to next page…)

3. These are the top 20 most positive and top 20 most negative features. Yes, these features make sense. The top 20 positive features are the words that are mostly used in context of a positive sentiment. Similarly, the top 20 negative features are the words that are mostly used in context of a negative sentiment.

Top 20 Most Positive Features:

[Feature Name]	[Weight]
engrossing	639282.0
rare	591348.0
french	579413.0
unexpected	559422.0
provides	547909.0
triumph	544577.0
cinema	539322.0
powerful	538348.0
treat	533134.0
wonderful	533020.0
pulls	531673.0
skin	529566.0
open	524179.0
delightful	511862.0
culture	511269.0
refreshingly	511237.0
dots	504579.0
beautiful	503583.0
speaks	486053.0
unique	471715.0

Top 20 Most Negative Features:

[Feature Name]	[Weight]
boring	-798158.0
dull	-716451.0
generic	-614812.0
fails	-583379.0
too	-579673.0
badly	-571385.0
routine	-563549.0
problem	-562938.0
instead	-554150.0
tv	-542904.0
ill	-542089.0
bad	-528848.0
attempts	-523295.0
guy	-513014.0
flat	-511995.0
worst	-506568.0
unless	-502886.0
suffers	-494019.0
neither	-493912.0
clich	-490472.0

4. The model doesn't care about the order in which the words appear in the sentences, which is very important for understanding the sentiment. Without the context information, the model wrongly classifies a positive sentiment sentence as negative sentiment and vice versa. Upon looking at more examples, it can be seen that some highly positive words (features) can be used to make a negative sentiment sentence and vice versa. This is one of the major shortcoming of this approach.

Top 5 Most Negative (Incorrectly) Classified Examples:

[Index]	[Sentence]
366	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
203	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 exactl
915	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
117	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
186	return to never land may be another shameless attempt by disney to rake in dough from baby boomer families , but it 's not half bad

Top 5 Most Positive (Incorrectly) Classified Examples:

[Index]	[Sentence]
237	` in this poor remake of such a well loved classic , parker exposes the limitations of his skill and the basic flaws in his vision '
788	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
923	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
552	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
2	an atonal estrogen opera that demonizes feminism while gifting the most sympathetic male of the piece with a nice vomit bath at his wedding

3. Pruning the Vocabulary

1. Yes after pruning one-count words from the training set, the dev error rate improved to **25.9%**, which is the best seen so far. Below are the training results:

```
» ~/D/O/C/A/h/hw2-data on main × time python3 train.py 1count_filtered_train.txt dev.txt 2
epoch 1, update 39.0%, dev 31.6%
epoch 2, update 26.4%, dev 27.5%
epoch 3, update 22.8%, dev 26.8%
epoch 4, update 18.8%, dev 26.6%
epoch 5, update 17.2%, dev 25.9%
epoch 6, update 14.8%, dev 26.5%
epoch 7, update 13.4%, dev 26.9%
epoch 8, update 12.4%, dev 26.8%
epoch 9, update 12.1%, dev 27.4%
epoch 10, update 10.0%, dev 26.7%
best dev err 25.9%, |w|=8427, time: 0.8 secs

_____________________________________________________
Executed in    1.32 secs    fish          external
   usr time    2.06 secs    0.08 millis    2.06 secs
   sys time    0.61 secs    1.21 millis    0.61 secs
```

2. Yes, the model size shrunk from $|w| = 15806$ to $|w| = 8427$, after removing one-count words from the training set. The new model has 7379 less dimensions, almost 50% the size of the previous one. Yes shrinking definitely help prevent overfitting by making the model resistant to learning idiosyncrasies in the training data.

```
» ~/D/O/C/A/h/hw2-data on main × time python3 train.py 2count_filtered_train.txt dev.txt 2
epoch 1, update 38.9%, dev 31.1%
epoch 2, update 27.9%, dev 29.5%
epoch 3, update 23.2%, dev 29.8%
epoch 4, update 21.2%, dev 27.4%
epoch 5, update 18.9%, dev 26.7%
epoch 6, update 17.0%, dev 27.3%
epoch 7, update 16.3%, dev 27.6%
epoch 8, update 15.3%, dev 27.8%
epoch 9, update 13.7%, dev 27.9%
epoch 10, update 12.9%, dev 28.1%
best dev err 26.7%, |w|=5938, time: 0.7 secs

_____________________________________________________
Executed in    1.24 secs    fish          external
   usr time    1.77 secs    0.08 millis    1.77 secs
   sys time    1.03 secs    1.23 millis    1.03 secs
```

3. Yes the update percentage changed, its doing about the same number of updates per epoch but since the dimensionality of all vectors reduced, the percentage has increased. Yes this change make sense.

4. Yes, the training speed improved to **2.06 secs**, which is fastest among all previous results.

5. After further pruning two-count words, the training speed improved even further to **1.77 secs**. The dev error did not decrease further, rather it increased to **26.7%**. So **no improvement** is observed in **dev error rate** after further pruning two-count words.

4. Try some other learning algorithms with sklearn

1. I tried the Sklearn's **Support Vector Machine.** I used the training dataset with one-count words removed. To make the code work with the algorithm, I first counted the number of unique words in the training data, and then did a process similar to One Hot Encoding where each unique word that appears in the sentence has its corresponding feature's value increased by 1. The dimension of all data point vectors and model is now equal to the number of unique words in the training dataset. I now have to represent the model weights and data vectors as **numpy array of shape (8426,)** — which is a numpy vector instead of **svector** object to make it work with the Sklearn's SVM class.

```
>> ~/D/O/C/A/h/hw2-data on main × time python3 train.py 1count_filtered_train.txt dev.txt 3
[INFO] Train data created
Shape of X_train: (8000, 8426)
Shape of y_train: (8000,)
[INFO] Dev data created
Shape of X_dev: (1000, 8426)
Shape of y_dev: (1000,)
[INFO] Training SVM...
[INFO] SVM trained
[INFO] Evaluating SVM...

[INFO] SVM dev error: 28.10%


_______________________________________________________
Executed in  343.33 secs    fish           external
   usr time  338.71 secs    0.10 millis  338.71 secs
   sys time    1.56 secs    1.22 millis    1.55 secs
```

2. The dev error rate is **28.10%**. The running time for this algorithm is the highest of all previous runtimes (because of high dimensionality of model and data point vectors). The running time is **338.71 secs.**

3. The average perceptron algorithm is faster and more accurate than the SVM algorithm. The faster speed of the average perceptron can be attributed to a more efficient data structure for representing the model and data point vectors. I observed a similar thing with SVM, where removing 2-count words led to a poorer performance.

5. Deployment

I used the average perceptron model trained on the the training dataset with one-count words removed. The best dev error rate I achieved with this model is **25.9%**.

6. Debriefing

1. I have spent about 6 hours on this assignment.
2. I rate this as easy.
3. I worked on it alone.

4. I understand the material 95%.
5. I liked working on this homework.