Joseph Houghton
AI 534: HW2

Part 0: Sentiment Classification Task and Dataset

1. We want to standardize the data as much as we can so that we make input as easy for our algorithms to understand as we can. So these are ways that we can normalize our data across all examples so that we get better performance out of our learning algorithms and also run into fewer bugs.

Part 1: Naive Perceptron Baseline

1. svector.py can perform these operations because the Python dunder/magic methods have been rewritten to accommodate vector/default_dict operations instead of, say, integer operations.
2. train() and test() in train.py
    a. train():
        i. this method takes in the training and dev info, it then instantiates the weights vector which it calls "model"
        ii. then for each run through the data, or for each "epoch," the function will:
            1. read from the training file and perform the $y * (w . x) <= 0$ check which the Perceptron algorithm uses to check for a mistake or a "mis-classification"
                a. here the Perceptron algorithm is checking to see if the current weights/model *w* is producing outputs which match up with the given label *y* when fed with the inputs *x*
            2. if there is a mistake, the function will perform an "update" via this equation: $w = w + yx$
                a. here the Perceptron algorithm is adjusting its model so that the weights vector more closely aligns with the "oracle vector" which will eventually correctly separate the data
            3. then the function checks the dev error for the current epoch by reading in from the dev file and using the same $y * (w . x) <= 0$ equation to check the dev data against the current model
        iii. the function concludes by reporting which of the dev error rates was the best amongst the epochs
    b. test ():
        i. this function takes in dev data, and a Perceptron weights vector as a "model" and it uses Perceptron's $y * (w . x) <= 0$ equation to check the dev data against the current model

        ii. it then returns the number of mistakes there were compared to the total number of data points in the dev data: $dev\ mistakes\ /\ total\ dev$

3. I opted to add my bias dimension implicitly by adding a 1 to the features vector, and a 0 to the weights vector. This improved my dev error from 30.1% to 28.9%
4. Even though all of the data sets are balanced, adding the bias dimension still allows our model to be more flexible when fitting to the data since it now has the extra dimension. Now our model can shift its separator to provide a better delta buffer zone, and this can give us more accuracy with unseen data. So our model receives an extra degree of freedom with which it can capture a broader range of patterns in the data.

Part 2: Averaged Perceptron

1. Results after changing vanilla Perceptron to averaged Perceptron:
    a. epoch 1, update 39.0%, dev 31.4%
    b. epoch 2, update 25.5%, dev 27.7%
    c. epoch 3, update 20.8%, dev 27.2%
    d. epoch 4, update 17.2%, dev 27.6%
    e. epoch 5, update 14.1%, dev 27.2%
    f. epoch 6, update 12.2%, dev 26.7%
    g. epoch 7, update 10.5%, dev 26.3%
    h. epoch 8, update 9.7%, dev 26.4%
    i. epoch 9, update 7.8%, dev 26.3%
    j. epoch 10, update 6.9%, dev 26.3%
    k. best dev err 26.3%, |w|=15806, time: 0.6 secs

    l. This improved the dev error rate from 28.9% to 26.3%
    m. As can be seen above, this also made the dev error rates more stable. The vanilla Perceptron was jumping back and forth with its error rates in each epoch.

2. The vanilla Perceptron ran the 10 epochs at 0.5s and the averaged Perceptron ran them at 0.6s according to the results in the given print statements. So the timing in this practical case is almost the same and the time complexity should be the same $O(Ud_{dw} + d)$ as indicated by Daumé's thesis.

3. Top 20 most positive/negative features (note that this part has been commented out in the code to more easily view Part 4 of this assignment):
    a. Pos Word: triumph, Pos Weight: 16.0
    b. Pos Word: smarter, Pos Weight: 16.0
    c. Pos Word: culture, Pos Weight: 15.0
    d. Pos Word: flaws, Pos Weight: 14.0
    e. Pos Word: rare, Pos Weight: 14.0
    f. Pos Word: skin, Pos Weight: 14.0
    g. Pos Word: engrossing, Pos Weight: 14.0

h. Pos Word: unexpected,   Pos Weight: 14.0
i. Pos Word: provides,   Pos Weight: 14.0
j. Pos Word: 1920,   Pos Weight: 14.0
k. Pos Word: winning,   Pos Weight: 13.0
l. Pos Word: happy,   Pos Weight: 13.0
m. Pos Word: imax,   Pos Weight: 13.0
n. Pos Word: refreshingly,   Pos Weight: 13.0
o. Pos Word: loved,   Pos Weight: 13.0
p. Pos Word: everyday,   Pos Weight: 13.0
q. Pos Word: treat,   Pos Weight: 13.0
r. Pos Word: dots,   Pos Weight: 13.0
s. Pos Word: am,   Pos Weight: 13.0
t. Pos Word: wedding,   Pos Weight: 12.0

u. Neg Word: generic,   Neg Weight: -18.0
v. Neg Word: badly,   Neg Weight: -17.0
w. Neg Word: boring,   Neg Weight: -17.0
x. Neg Word: routine,   Neg Weight: -16.0
y. Neg Word: incoherent,   Neg Weight: -16.0
z. Neg Word: waste,   Neg Weight: -15.0
aa. Neg Word: ill,   Neg Weight: -15.0
bb. Neg Word: god,   Neg Weight: -15.0
cc. Neg Word: bore,   Neg Weight: -14.0
dd. Neg Word: deserve,   Neg Weight: -14.0
ee. Neg Word: results,   Neg Weight: -14.0
ff. Neg Word: mindless,   Neg Weight: -13.0
gg. Neg Word: dull,   Neg Weight: -13.0
hh. Neg Word: disguise,   Neg Weight: -13.0
ii. Neg Word: coherent,   Neg Weight: -13.0
jj. Neg Word: neither,   Neg Weight: -13.0
kk. Neg Word: fails,   Neg Weight: -13.0
ll. Neg Word: pie,   Neg Weight: -13.0
mm. Neg Word: scattered,   Neg Weight: -13.0
nn. Neg Word: inane,   Neg Weight: -13.0

oo. Some of these words make sense, like "triumph, loved, happy" in the most positive features and "boring, dull, fails" in the most negative features. But there are some other words that do not really make sense when it comes to classifying positive/negative reviews. For example, having the features "1920, skin" in the positive features does not make sense.

4. Note that this part has been commented out in the code to more easily view Part 4 of this assignment. I took my 5 largest positive/negative dev errors from epoch 10 since that had the best-performing model on dev, here are the results:

i. 5 positive examples my model most strongly predicted to be negative:

ii. Error of 3488472.0 on Pos Dev Label:
    1. 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

iii. Error of 3245500.0 on Pos Dev Label:
    1. 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

iv. Error of 2323392.0 on Pos Dev Label:
    1. 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

v. Error of 2177851.0 on Pos Dev Label:
    1. 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 characters , i 'll take the latter every time

vi. Error of 1655419.0 on Pos Dev Label:
    1. carrying off a spot on scottish burr , duvall (also a producer) peels layers from this character that may well not have existed on paper

vii. 5 negative examples my model most strongly predicted to be positive:

1. Error of 2808479.0 on Neg Dev Label:
    a. ` in this poor remake of such a well loved classic , parker exposes the limitations of his skill and the basic flaws in his vision '

2. Error of 2405584.0 on Neg Dev Label:
    a. 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

3. Error of 2377569.0 on Neg Dev Label:

        a. 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 propaganda

    4. Error of 2296951.0 on Neg Dev Label:
        a. 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 , grief and fear

    5. Error of 1897881.0 on Neg Dev Label:
        a. an atonal estrogen opera that demonizes feminism while gifting the most sympathetic male of the piece with a nice vomit bath at his wedding

viii. Observations:

    1. I would say that these examples mostly contain ambiguous language and it makes sense that the model would mis-classify these types of examples most strongly. For example, the model's worst mistake on the negative examples has the phrase "a well loved classic" and most of these other examples contain words that would be considered both positive and negative. So Perceptron has trouble distinguishing these examples most likely due to the simplicity with which it classifies.

Part 3: Pruning the Vocabulary

1. Yes, my dev error improves a little after pruning one-count words from the training set: 26.3% → 26.0%
2. Yes my model size also shrank by almost half: 15806 words → 8425 words
    a. this shrinking can certainly help prevent overfitting because one-count words could be considered "noise" since they are uncommon across all examples in the training set and thus may be considered "too specific" to be useful in recognizing the general patterns in our data. So cutting them out of training could help our model cut out more noise in our data and allow it to focus on tuning the weights of more important words.
3. The update % for the epochs showed a general slight increase after removing the one-count words.
    a. This may make sense if it allows Perceptron to focus more on essential features and thus it may perform more updates as it does this. It may be the case, that when no words are pruned, our model "learns" more noise in the data, and thus it makes less mistakes during training, but then it performs worse during dev. So the model may have an "easier" time learning noise, but when words are pruned,

the model may have a more difficult time fine-tuning less-noisy data and thus it may make more mistakes and more updates during training after pruning.

4. Results from linux's time command on my local:
    a. no words pruned: user time 0.623 secs
    b. one-wount words pruned: user time 0.653 secs
    c. so the pruned data set actually took a little bit longer, and this may be because the update % also increased and thus Perceptron is making more mistakes and more updates on the pruned data set during training.
5. After pruning one-count and two-count words from the training set, my dev error rate actually became worse: 26.0% → 27.2%
    a. This may mean that we cut out words that were not just noise, but useful in the construction of the model. So this may be an example of over-pruning such that our algorithm's ability to correctly classify data becomes worse.

Part 4: Other Learning Algorithms with skLearn

1. I opted to try out different algorithms side by side to compare performance. I used TfidfVectorizer() to convert the words in the input examples into TF-IDF features which I then ran through sklearn's SVM, Multinomial NB, and Multilayer Perceptron algorithms. I also converted the input into an xgboost DMatrix() and tried it out on the XGBoost algorithm. I did also add some hyperparameters to the XGBoost algorithm but I couldn't get it to work in the end. I then ran these algorithms through a script which tests all of them with different levels of pruning.
2. My results can be seen by running the tasks.sh bash script in my code, note that the MLP can take a long time and that something was wrong with my XGBoost. Below are the results for no pruning, one-count pruning, and two-count pruning. The best dev error rate and runtime came from the Multinomial NB when no words were pruned: dev error: 23.0% , runtime <0.1 secs

    i. Training data with no words pruned:
    ii. Avg Perceptron Dev Error: 26.3%, |w|: 15806, time: 0.6 secs
    iii. Avg Perceptron % Pos on Test data: 40.2%
    iv.
    v. SVC Classifier Dev Error: 26.8%, |w|: 15747, time: 4.5 secs
    vi. SVC % Pos on Test data: 47.8%
    vii.
    viii. MNB Classifier Dev Error: 23.0%, |w|: 15747, time: 0.0 secs
    ix. MNB % Pos on Test data: 48.6%
    x.
    xi. XGB Classifier Dev Error: 37.4%, |w|: 15747, time: 0.4 secs
    xii. XGB % Pos on Test data: 0.0%
    xiii.
    xiv. MLP Classifier Dev Error: 27.5%, |w|: 15747, time: 38.7 secs

xv. MLP % Pos on Test data: 49.6%

xvi.

xvii.

xviii.

xix. Training data with 1-count words pruned:

xx. Avg Perceptron Dev Error: 26.0%, |w|: 8425, time: 0.6 secs

xxi. Avg Perceptron % Pos on Test data: 49.4%

xxii.

xxiii. SVC Classifier Dev Error: 26.5%, |w|: 8376, time: 4.1 secs

xxiv. SVC % Pos on Test data: 47.8%

xxv.

xxvi. MNB Classifier Dev Error: 23.9%, |w|: 8376, time: 0.0 secs

xxvii. MNB % Pos on Test data: 46.9%

xxviii.

xxix. XGB Classifier Dev Error: 37.4%, |w|: 8376, time: 0.3 secs

xxx. XGB % Pos on Test data: 0.0%

xxxi.

xxxii. MLP Classifier Dev Error: 27.4%, |w|: 8376, time: 36.8 secs

xxxiii. MLP % Pos on Test data: 50.6%

xxxiv.

xxxv.

xxxvi.

xxxvii. Training data with 1 and 2-count words pruned:

xxxviii. Avg Perceptron Dev Error: 27.2%, |w|: 5934, time: 0.6 secs

xxxix. Avg Perceptron % Pos on Test data: 36.4%

xl.

xli. SVC Classifier Dev Error: 26.8%, |w|: 5888, time: 3.9 secs

xlii. SVC % Pos on Test data: 48.0%

xliii.

xliv. MNB Classifier Dev Error: 24.9%, |w|: 5888, time: 0.0 secs

xlv. MNB % Pos on Test data: 46.8%

xlvi.

xlvii. XGB Classifier Dev Error: 37.6%, |w|: 5888, time: 0.3 secs

xlviii. XGB % Pos on Test data: 0.0%

xlix.

l. MLP Classifier Dev Error: 28.9%, |w|: 5888, time: 21.5 secs

li. MLP % Pos on Test data: 50.0%

3. My results showed that Averaged Perceptron performed its best when 1-count words were pruned:
    a. dev error: 26.0%
    b. time: 0.6 secs
    c. percent positive predictions on test.txt: 49.4%

   b. However, the Averaged Perceptron was much more sensitive to changes in the input data. Its performance declined as more words were pruned. On the other hand, the SVM, MNB, and MLP all continued to return fairly consistent results with respect to the positive percentage predicted on test.txt even when up to 20-count words had been pruned. This may signal a lack of depth in the averaged perceptron algorithm's ability to learn. Since it is a simpler model than these others, it may be more susceptible to things like overfitting. The SVM and the MLP algorithms in sklearn are able to introduce nonlinearity, among many other additions like hyperparameters, into their learning and thus may be able to recognize patterns in the data in more complex ways. And while the MNB is also a linear algorithm, it is known to handle text classification very well. These differences may give these other algorithms an advantage over the older perceptron algorithms.
   c. Also worth mentioning is the fact that these more complex algorithms utilized the data in TF-IDF form whereas the Averaged Perceptron pre-processing was only keeping track of the frequency of a word within its given data example. The TF-IDF vectorizer however seems to also take into account how frequent that word is across all examples and it is able to then generate a more useful value for how that word should be weighted at testing time. This may also explain why these other algorithms are able to remain more resilient even as many words are pruned from the training data.

Part 5: Deployment

1. The best dev error rate was from the Multinomial Naive Bayes algorithm when no pruning was performed: dev error: 23.0%
   a. note that the training data in this case was converted into a matrix of TF-IDF features

Part 6: Debriefing

1. I don't actually count the hours I spend on assignments but maybe around 25 hours
2. easy
3. alone
4. Maybe 90%. I think the only parts that were a little fuzzy for me were the details of the averaged perceptron implementation. It was intuitive on a higher level what the sum vectors and auxiliary vectors were for and how they were used to balance out the perceptron updates but the details of how exactly this was working was a little less clear for me this first time around.
5. I don't really have further comments, it's a fun assignment and it's nice being able to choose any sklearn learning algorithm that we wish to use