Binary Perceptron Implementation

My peceptron implementation was prototyped on Jupyter Notebooks on OSX and then transfered to a python script. The results were pretty interesting and helped me understand better how this algorithm works. Standard python packages were loaded, I used NLTK to get a stopwords list and scikit's precision and recall score. Each review was loaded into a list before splitting into train and test sets.

1. For the first part, a standard bag of words was created for each review and then the perceptron tried to learn from the ordered train dataset. This was a very bad because our perceptron only learns to predict the last class it sees, so our test result was just 0.5 accuracy, which is basically not better then random guess.

   The explanation is that the weights are not updated enough, the first prediction is positive and the label is negative, which updates the weights to negative, then the next 799 predictions are negative which corresponds to the label, so our weights are not updated. After the first positive review comes, the matrix is updated again but the 799 restant predictions are OK, which leaves our weights almost untouched

2. Shuffling the dataset makes a big improvement, with only one pass we go up to 0.68 precision, which is much better. This is due to the fact that our weight matrix gets much more updates and then it can learn accurate weights for the test predictions.

3. Doing multiple passes over different shuffles of our dataset reinforces the concept of more updates to the weight matrix, which results in much better predictions. In this case we did 10 passes over the dataset which shows a trend of improvement in the accuracy metrics for each iteration, converging to a value around 0.8 precision. Figure 1 shows the learning process.

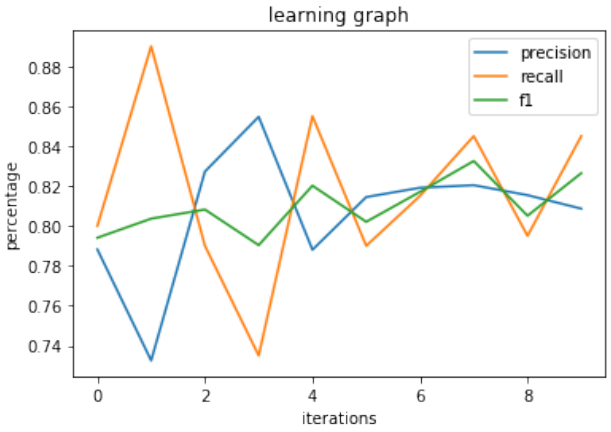

Figure 1: Perceptron Learning process

4. To implement two new features, I added bigrams and trigrams for each review to see if there were some relations between words that could help to get better results. For example the weight of the unigram "very" is not clear for inferring a class, but "very good" or "very bad" can be much more informative and give a better idea to classify the review.

   I first tried using the bag of words plus bigrams, which resulted in a better precision than the part number 2, the precision for this model was 0.73 vs 0.68 from the unigram model.

   Then I tried to predict the classes using the bag of words, bigrams and trigrams, which gave a even better precision of 0.85.

   I think it is important to notice that even though the precision was better in this model, the recall was fairly low (0.52) compared to the 0.8 of the unigram model, so we sacrifice a lot of recall for increasing our precision.

5. The most weighted features for each class are the following:

   - Positive: well, terrific, great, fun, truman, bond, seen, comic, seen, especially.
   - Negative: bad, nbsp, unfortunately, script, worst, supposed, nothing, plot, stupid, reason.

   Most of them make sense but there are some positive weights like "truman" or "especially" which at first glance seem neutral. There is also a mistake in the negative weights like 'nbsp' that needs to be taken out.

   These features will probably not generalize as well for other domains like restaurant reviews, given that the vocabulary used to describe movies is different that food, so words like plot, stupid or comic, won't be very useful as weights. For this domain, features related to taste or flavour (tasty, delicious, homemade) will be much better to predict sentiment.