NLP Problem Set 4 – Programming Section
Juliana Louback - jl4354@columbia.edu

Question 4.

Comments: If a bigram is not seen in the tag.model file, or if the word-tag combo is also unseen, the history receives weight 0. The tagging result is saved to the file q4_output. The file q4_histories contains all possible histories for the development data (from running tagger_history_generatory.py ENUM); it is saved as it is used in quesiton5.py.

Run: `python question4.py`

Output: `q4_output, q4_histories`

Expected Runtime: ~3 seconds

By an oversight, I initially wrote a tagger that only considered features of words and their respective POS tags, resulting in 81% accuracy. The evaluation of that tagger is displayed below:

```
Julianas-MacBook-Air:python julianalouback$ python eval_tagger.py tag_dev.key
q4_output
2014 2459 0.819032126881
```

Finding the accuracy levels lower than expected, I quickly found the cause and modified the tagger to also include bigram features. This is, of course, the final version of question4.py, as such, the output file q4_output has the best tagging results achieved, that of 90.5% accuracy. The evaluation of the tagger for question 4 is displayed below:

```
Julianas-MacBook-Air:python julianalouback$ python eval_tagger.py tag_dev.key
q4_output
2226 2459 0.905246034974
```

Question 5.

Comments: The update method I chose is to penalize inaccurate feature weights by -0.0001 and reward accurate features by +0.0001. This penalty value was established after several experiments. All weights are initialized as 0.

Run: `python question5.py`

Output: `suffix_tagger.model, q5_output`

Expected Runtime: ~ 2 minutes

I experimented running the perceptron 5 times, testing on the development data with the feature vector returned after each run (combined with the tag and bigram features from tag.model). The results are displayed below for 3-5 perceptron iterations. As can be seen in the results, the perceptron converged after 4 iterations, increasing accuracy from 91.9% to 92.06%. Comparing this model to that of question 4, which used only features from tag.model, there is an increase in performance from 90.5% accuracy to 92.06%. The suffix model features and weights are saved to suffix_tagger.model.

```
k = 3:      2261 2459 0.919479463196
k = 4:      2264 2459 0.92069947133
k = 5:      2264 2459 0.92069947133
```

Question 6.

Comments: All the new features added to the model in this section were selected based on the errors made in the suffix+POS+bigram model built in question5.py

Run: `python question6.py`

Output: `q6_output_combo1, q6_output_combo2, q6_output_combo3, q6_output_combo4`

Expected Runtime: ~10 seconds

Combination 1

Looking over the tagging results of the model in question5.py, I noticed several frequent mistakes. Words with the suffix "ed" were constantly being tagged as nouns instead of verbs; words with suffix "ly" were tagged as adjectives instead of adverbs; words with suffix "ing" were tagged as nouns instead of verbs. I decided to add 'bonus' weight of +5 to the *[SUFFIX:ed:VERB]* feature, +5 to the *[SUFFIX:ly:ADV]* feature and +0.05 to the *[SUFFIX:ing:VERB]*. Less weight given to the third feature as there are various occurrences of nouns ending with "ing" as well. The predicted tags are saved to the file q6_output_combo1. This model resulted in an increase in accuracy compared to the model in question 5, from 92.06% to 92.59%. Later I noted that removing the third feature *[SUFFIX:ing:VERB]* did not alter the performance; I ruled it inconsequential.

```
Julianas-MacBook-Air:python julianalouback$ python eval_tagger.py tag_dev.key
q6_output_combo1
2277 2459 0.925986173241
```

Combination 2

With regard to the bigram features, I noticed that the sequence VERB VERB was frequently marked as incorrect. I set the weight for this feature *[BIGRAM:VERB:VERB]* to -0.5, modifying the model composed of tag.model and suffix_tagger.model; this alone increased the accuracy from 92.06% to 92.27%, not as great an improvement as before but this is a single feature. See q6_output_combo2.

```
Julianas-MacBook-Air:python julianalouback$ python eval_tagger.py tag_dev.key
q6_output_combo2
2269 2459 0.922732818219
```

Combination 3

For some reason, many times numbers with decimal points were tagged as nouns. The same would happen for time (i.e. 3:00). This led to the creation of a new feature, *[CONTAINS:DIGIT:NUM]* with a relatively high weight, +5. Another common mistake was tagging hyphenated words as a noun instead of an adjective. I also added the feature *[CONTAINS:HYPHEN:ADJ]* with weight +5. This resulted in the most dramatic improvement by far, from 92.06% to 94.02%. See output_combo_3.

```
Julianas-MacBook-Air:python julianalouback$ python eval_tagger.py tag_dev.key
q6_output_combo3
2312 2459 0.940219601464
```

Combination 4

Putting all the features together now (with the exception of the third feature of Combo 1) resulted in a slight increase in performance, from 94.02% accuracy to 94.06%. See output_combo_4.

```
Julianas-MacBook-Air:python julianalouback$ python eval_tagger.py tag_dev.key
q6_output_combo4
2313 2459 0.940626270842
```