Ex3 NLP

Question 2

replaced: 0.0352368561769

replaced: 0.0494304160331

dev: most frequent acc: 0.922501682578

Question 3 (b)

Optimal Lambda Values (based on Viterbi accuracy):

Apply gid search with step = 0.1, start value 0.0 and end value 1.0

Best lambdas: lambda1 = 0.7, lambda2 = 0.3, lambda3 = 0.0

Viterbi Pruning Policy:

In the outer loop, we iterate on the current tag. Assume we wish to compute *PI[k, u,v]* where you *u* is the previous tag and *v* is the current tag. Let x_k be the current word. Check if $e(x_k/v) > 0$: if yes, we calculate *max PI[k-1,w,u]*, otherwise – and here is the pruning part – we set *PI[k,u,v] = 0* and continue to the next possible current tag. In fact, the pruning saves us *n-square* calculations where *n* is the number of possible tags. In terms of run time, each run on the test data and with a fixed set of lambda values took us around 70 seconds. Total run time 5,036 seconds. Here is a small portion of the trace:

lambda1=0 lambda2=0 score=0.915970785453 time in Sec=69.9900000095

lambda1=0 lambda2=0.1 score=0.923149786873 time in Sec=69.6319999695

.

.

lambda1=0.8 lambda2=0.2 score=0.939601665129 time in Sec=70.0780000687

lambda1=0.9 lambda2=0 score=0.9394521026 time in Sec=70.3169999123

lambda1=0.9 lambda2=0.1 score=0.939277612982 time in Sec=71.2699999809

lambda1=1.0 lambda2=0 score=0.938978487923 time in Sec=70.6990001202

Question 3 (c)

Viterbi accuracy: 0.94

Question 4 (c)

Pruning Policy in Viterbi – we defined a beam width parameter which chooses best N pairs of tag_{k-2} and tag_{k-1} of previous stage k-1 and prunes the rest, that is, the N-top *PI[k-1,w,u]* where *w,u* are tags. Due to the aggressive number of 10 as the beam width and calling logreg.predict_proba once per sentence position (with multiple examples), we were able to achieve a reasonable run time of 2 hours and 20 minutes for the evaluation of the dev. set part.

Question 4 (d)

Accuracy: greedy – 0.9579, viterbi – 0.9583

Question 4(e)

- Making wrong judgement (prediction) on **initCap** words

  Replacing NN (Noun) with NNP (Proper Noun)
  "..at the Coliseum.. "
  Since Coliseum is a rare word, it was replaced to initCap. It is reasonable that the model *always* treats initCap as a proper noun however in reality (or in truth) initCap can represent words of different syntactic functions (though the most common tag would be probably a proper noun).

  Replacing JJ (adjective) with NNP (proper noun)
  "..New-York Yankees-Mets series.."

Yankees-Mets is a rare word. Since it is replaced by initCap, it is predicted as NNP, however, as it is cleared from the expression, it functions as an Adjective.

- Wrong prediction on firstWord words

  Replacing RB (Adverb) with NNP (proper noun)
  "Anyway, the A's gave you.."
  Same case as initCap. *Anyway* (in upper case) is a rare word. Since the model sees the category word and is not aware of the particular word, it is hard to predict the correct tag. Noun, as the first word in a sentence, would be most likely.

- Predicting JJ (adjective) instead of VB (verb) and NN instead of VBG
  "..to separate opposing fans.."
  Here we have, I believe, an error propagation example – it is not clear, at first glance at least, why the model treats *separate* as adjective since it would be expected to see verb after *to*. However, *separate* can be an adjective as well, thus this kind of an error can be related to the ambiguity of that word. Since it is common that a noun follows an adjective, it is not surprising that *opposing* was treated by the model as a noun.

- Replacing JJ (adjective) with RB (adverb)
  "..when he hit his first career home run last season, .."
  The model predicted the word *last* as an adverb though clearly it is an adjective followed by the noun *season* (*home run* is an expression). Since *last* can be an adverb as well, we may relate the error to its ambiguity or perhaps even to the ambiguity of *run*. But since *run* was correctly treated as noun, the question why the model predicted adverb after a noun is still opened.