Advanced Methods in Natural Language Processing – HW 3

# **2. Most frequent tag baseline**

1. The accuracy of the most frequent tagger on the dev set is: 92.6%.

# **3. HMM tagger**

1. The pruning policy used relies on the fact that for many words, only a small subset of tags is possible according to the language grammar. Therefore, observing the *emissions* (denoted ) for some word , we get that for the set of all possible tags: , more often than not, .

Viterbi algorithm relies on dynamic programming, and dictates that for word and sets of tags , the trajectory for triplet (k, u, v) is:

When , the current trajectory in the calculation is zero, and the same applies for all future trajectories stemming from this current trajectory. Consequentially there is no point in calculating trajectories for pairs of words and tags where the frequency of pair for word k and some tag v is 0.

The pruning scheme initializes the set of possible tags for each word to:

effectively eliminating many “zero” probability trajectories.

*Note: For practical precision purposes, all actual calculations in the code are done in log space.*

1. The best accuracy obtained on the dev-set with HMM & Viterbi is 95.55%.

The hyperparameters chosen are:

1. Assume the following setting (the greedy algorithm choices are bolded):

The greedy algorithm opts to choose the sequence: ‘NN’, ‘JJ’.

The probability of this sequence is:

Viterbi opts to choose the higher probability sequence in this case: ‘VB’, ‘JJ’

The highest probability is:

*Note: The example excludes the ‘STOP’ token for brevity.*

1. Assume the following setting:

*;*

*;*

*;*

The highest probability sequence for third-order transition parameters is:

While the correct labeling sequence gets the following lower probability for third-order transitions:

The highest probability sequence for second-order transition parameters is:

Which is also the correct labeling sequence.

# **4. HMM tagger**

(c) We incorporated a few optimizations in our implementation:

* When extracting features for position i, we first extracted features with “mock” tags of ‘#’ (for extracting the features related to the input words like prev\_word, prefixes etc.). Then, for each prev\_tag and prevprev\_tag we just updated the features dictionary with the correct tags and saved if separately.
* We predicted scores for all t, u, v options in specific position i with one call to logreg.predict\_log\_proba(). It seems that it does its own optimizations such that predicting score for one example and for multiple examples together takes pretty much the same time. Then, when calculating the score for each t, u, v triplet we took the proper score from the predicted scores.
* for each k, u, v we took the maximum score (and back pointer) in-place, without going over all possibilities of t after all evaluations.

(d) Our accuracy on the dev set is 95.89% with the Viterbi algorithm and 95.87% with the greedy algorithm.

(e) We sampled errors from the MEMM model. We noticed some failure cases:

* Tagging rare words replaced with a category. For example:
  + “…,and **lowerCase** from it…”: True tag: VBN, model tag: NN
  + ” …Shearson's **'UNK'**, UNK…”: True tag: JJ, model tag: NN
  + “…“Where We **initCap**” commercials…”: True tag: VBP, model tag: NNPS

A reasonable explanation to this can be that the categories are too general so there is a lot of variance in the true tags in the data.

* Confusion between VBN and JJ. Sometimes it seems that the true label was the one that is incorrect:
  + “Fidelity, for example, **prepared** ads several…” True tag: VBN, model tag: JJ
  + “…,when **frightened** investors flooded…”: True tag: JJ, model tag: VBN
  + “'The **complicated** language in…”: True tag: VBN, model tag: JJ (here it seems that actually the model was right, and the ground truth is wrong)