CSE 390 Assignment 2: Part-of-Speech Tagging

Due: Mar 23, 2016 by 11:59 PM EST

You will implement a frequency-based tagger, and a Hidden Markov Model based POS tagger in this assignment. You will implement three applications.

1. **Training** – This application will take a training file and generate the necessary transition and emission probabilities.
2. **Tagger** – This application will take as input the test file, the transition and emission probabilities files, and output the most likely tag sequence. You will implement two versions of this – a frequency-based tagger, and the HMM tagger.
3. **Evaluator** – This will evaluate the output from the taggers and print evaluation measures such as Accuracy, Precision, Recall, and F1.

1 Training (10 points)

The goal is to learn the probability matrices $\lambda = (A, B)$. The input file will consist of one tagged sentence per line as shown below.

Example: The/DT dog/NN chased/VBD the/DT cat/DT ./.

The input file will be on blackboard.

You will implement two estimation methos, MLE and Laplace. You will generate two sets of files mle-transitions.txt, mle-emissions.txt and laplace-transitions.txt, and laplace-emissions.txt, and laplace-tag-unigrams.txt.

Matrix A is the transition probability matrix, whose entries a_{ij} correspond to $Pr(t_j|t_i)$, and B is the emission probability matrix, whose entries b_{ij} correspond to $Pr(w_j|t_i)$.

The laplace-tag-unigrams.txt should contain the $Pr_L(t_i)$ for all tags. This will be used for the Frequency-based Tagger but not for the HMM Tagger. You are free to write these files in any format you like.

2 Baseline Tagger (35 points)

2.1 Frequency-based Tagger (20 points)

Inputs: test.txt, laplace-emissions.txt, laplace-tag-unigrams.txt **Outputs:** Frequency-based POS tag sequence.

For each input sentence (S) in the test file, this tagger will return a tag using the following equation:

$$\forall w \in S, \text{predicted-tag}(w) = \arg\max_{t \in T} Pr_L(t|w) = \arg\max_{t \in T} Pr_L(w|t) Pr_L(t)$$

You are making independent predictions for each word in the sentence. In other words, the tag of a word only depends on the word and not on the previous word's tag (unlike the HMM).

In the case of ties, it should pick the more frequent tag overall i.e., across all tokens.

2.2 Unknown Word Heuristics (15 points)

As with the language model, you will encounter new words in the test data. You need to design **three heuristics** that will let you assign a POS tag for previously unseen word. Your goal is to improve performance on previously unseen words.

- Recall that issue with unseen words is that $Pr_{mle}(w_u|t_j)$ will be zero for all POS tags t_j. With Laplace $Pr_L(w_u|t_j)$ won't be zero but will give the same probabilities for all words for a given tag. For instance, if you did not see *John* and *race* in training, $Pr_L(John|NNP) = Pr_L(race|NNP)$, which is undesirable.

  As we saw in class there are a few things about the morphology (the structure of words) that we can use to get a better of $Pr(w_u|t_j)$.

- Identify words in training that appear less than two times across each POS category. Use these words to identify some subset of features that you can use to devise features.

  Features could be any aspect of the word. For example, a feature $f_{caps}(word)$ could indicate whether the word is capitalized or not. If the word w_u is capitalized, then you can use the following $Pr(f_{caps} = 1|t_j)$, i.e., the fraction of times that words with tag t_j were observed with capitalized first letter in training.

  Example Unknown Word Feature Probability Estimation: Let $count(t_j)$ be the number tokens that were tagged t_j in training. Let $count(f_{caps} = 1, t_j)$ denote the number of tokens that are tagged t_j and are capitalized. Then, the you can

compute the following probability as an approximation to the probability of an unknown capitalized word.

$$Pr(f_{caps} = 1|t_j) = \frac{count(f_{caps} = 1, t_j)}{count(t_j)}$$

For an unknown uncapitalized word, you can use the following probability.

$$Pr(f_{caps} = 0|t_j) = \frac{count(f_{caps} = 0, t_j)}{count(t_j)}$$

- Once you have identified three such features, you can use these features to approximate the unknown word probabilities. Suppose w_u has f_1 and f_3 (e.g., caps and ends with -ed), then one possibility is to do:

  $Pr_{L'}(w_u|t_j) = \beta_1 Pr(f_1 = 1|t_j) + \beta_2 Pr(f_2 = 0|t_j) + \beta_3 Pr(f_3 = 1|t_j) + \beta_4 Pr(unk|t_j)$

  where, $\beta_1 + \beta_2 + \beta_3 + \beta_4 = 1$

Note: You don't have to ensure that $Pr_{L'}(w_u|t_j)$ remains a valid probability distribution here, although that would be desirable because then you can readily incorporate it into the HMM Tagger or any other method that relies on proper probability distributions. For this assignment, any hack is fine. Welcome to empirical NLP!

3 HMMTagger (30 points)

This tagger will use the Viterbi Decoding algorithm to find the most likely tag.

Inputs: test.txt, x-transitions.txt, x-emissions.txt, where x is either mle or laplace. **Outputs:** Most likely tag sequences for each sentence along with their probabilities.

For ease of evaluation later on, for each sentence, output each word along with its true pos tag (which is in the test.txt), and the predicted pos tag (output by the HMM). You can use whatever format you like but here is an example. The number in the beginning is the probability of the tag sequence according to viterbi.

```
...
0.000323    John/NNP/NNP ate/VBD/VBN egg/NN/NNP
0.000234    The/DT/DT girl/NN/NN kicked/VBD/NN
...
```

Main Steps:

1. Iterate through each sentence in text.txt
2. For each sentence extract the words in the sentence and put into an array S.
3. Run the Viterbi algorithm to find the best tag-sequence for the array of words. See pseudo-code (in Algorithm 1) for an outline of the algorithm.

Some details:

- A and B are data structures that contain the transition and emission probabilities. You need to populate these from the transition and emissions file you wrote out during training.
- I am using t_i in the pseudo-code as a short for the i^{th} tag. One way to implement this is to use a tag array T, and replace it with $T[i]$. Similarly, I am using S_i to denote the i^{th} word in the sentence. Again use an array.
- Some look-ups for B would fail. You may not have seen all word-tag combinations during training, and some words in S maybe new words not seen in training. Whenever you access B if the particular tag-word combination is not in B you need to handle it properly. An easy trick is to add $B[t][unk]$ entries for all values of t. When a look up fails, use the corresponding unk entry. For instance, if $B[NNP][Niranjan]$ fails, then use $B[NNP][unk]$ instead.
- Instead of the product of probabilities in the forward pass loops, use the sum of log-probabilities (i.e, instead of $\delta[i-1][k] \times A[t_k][t_j] \times B[t_j][S_i]$ use $\log(\delta[i-1][k]) + \log(A[t_k][t_j]) + \log(B[t_j][S_i])$). The product can lead to underflow. It is common practice to replace the product with sum of log-probabilities.
- The procedure outlined only returns the best tag sequence. Please modify it to also output the probability of the best tag sequence.

4 Evaluator (15 points)

Compute the overall accuracy of the predictions you made on the test set. For each POS tag, compute the precision, recall and F1 measures. Here is a brief specification of the measures:

- Accuracy: # correct predictions / # predictions (i.e., number of words in the test set).
- Precision for tag k (P): # system correctly predicts tag k / # system predicts tag k
- Recall for tag k (R): # system correctly predicts tag k / # tag k in the test set.
- F1 = $\frac{2PR}{P+R}$

5 Error Analysis (10 points)

Error analysis is a critical part of building and using NLP tools.

- Analyze the outputs from each of the three methods (i) Baseline tagger, (ii) HMM with MLE and (iii) HMM with Laplace. Identify one type of error that is specific to each method, give two examples, and explain what caused that error.

Algorithm 1 Viterbi Algorithm

procedure VITERBIDECODER(S, A, B)
S is the input sentence, stored as an array. $S[i]$ is the i^{th} word in the sentence.
A contains transition probabilities: $A[t_i][t_j] = Pr(t_j|t_i)$
e.g., $A[NNP][NN] = Pr(NN|NNP)$
B contains emission probabilities $B[t][w] = Pr(w|t)$
e.g., $B[NNP][John] = Pr(John|NNP)$.
For A and B, you can use a map or a two dimensional array. Either way you need to handle cases where the look-ups can fail.

    //Initialization.
    $\delta \leftarrow$ A 2-dimensional array of size $n \times T$ to store the scores of the best paths.
    $\psi \leftarrow$ A 2-dimensional array of size $n \times T$ to keep backpointers.
    $n \leftarrow len(S)$, the number of words in S

    //Initialize δ and ψ for the first word in the sentence S_1.
    for $i \leftarrow 1$ to T **do**
        $\delta[1][i] \leftarrow A[\langle s \rangle][t_i] \times B[t_i][S_1]$
        $\psi[1][i] \leftarrow 0$, index of $\langle s \rangle$

    //Forward pass
    for $i \leftarrow 2$ to n **do**
        for $j \leftarrow 1$ to T **do**
            $\delta[i][j] \leftarrow \max_{1 \le k \le T} \delta[i-1][k] \times A[t_k][t_j] \times B[t_j][S_i]$

            $\psi[i][j] \leftarrow \arg\max_{1 \le k \le T} \delta[i-1][k] \times A[t_k][t_j] \times B[t_j][S_i]$

    //Backtrace
    $bestTags[n] \leftarrow \arg\max_{1 \le k \le T} \delta[n][k]$
    for $k \leftarrow n-1$ to 1 **do**
        $bestTags[k] \leftarrow \psi[k+1][bestTags[k+1]]$
    return bestTags

- Compute a confusion matrix – a matrix whose rows and columns are POS tags. An entry c_{ij} is the count for the number of times that the tagger output t_i for a word whose actual tag is t_j. See below for an example.

  Plot confusion matrix for the top 10 most frequent tags in the test set. You will produce a 10 x 10 matrix.

	Actual POS Tag			
	NN	NNS	PRP	VBG
NN	129	74	12	54
NNS	12	154	2	2
PRP	3	0	1023	0
VBG	23	17	0	98

Table 1: Example Confusion Matrix.

6 Submission

You are free to implement in any language.

- You should submit your code to blackboard. Please document your code and use a README that tells us how to run your code if we need to.
- You need to demo your work to the grader.
- Submit a report including any assumptions you've made, any other relevant implementation detail (e.g., How did you handle unknown words etc.), and the analysis.