# **CSEP 517, Fall 2015, Assignment 2**

<https://github.com/jogonzal-msft/UWNLPAssignment2>

## Introduction

The task is as follows – given an already existing POSTagger, improve two parts:

* TrellisDecoder interface implementation: Has a method that finds the best path (most likely) on a Trellis Graph.
  + GreedyDecoder: Simply looks at the next best transition and follows it to the end.
* LocalTrigramScorer interface implementation: This is where the train and getLogScoreCounter methods live. Train receives sentences and populates data structures inside the instance so that evaluate can later compute probabilities of trigrams appearing in a certain sequence.
  + MostFrequentTagScorer is an implementation that simply gives each test word the tag it was seen with most often in training (or the tag with the most seen word types if the test word is unseen in training)

The task is to write better implementations for TrellisDecoder and LocalTrigramScorer. The current score for the GreedyDecoder and the MostFrequentTagScorer is 92.31%, with an UNK tag accuracy of 38.85%.

### How does it handle/learn UNKs?

Each time it sees a new word, it records its tag and considers it an unknown word tag. This approach seems reasonable, since it assumes that the distribution of **unknown word tags** will follow a distribution similar to the one of **unique word tags**.

On current implementation (GreedyDecoder/MostFrequentTagScorer), unknown words will always get the same tag assigned since they count as one word – in the case of this training set, the tag NNP is always assigned to words because it is the one with that has more unique words.

## Problem1 – Building a sequence model

### Problem1.Part1 – Implement an HMMTagger

See class HMMTagScorer.

To implement this, the HMMTagScorer has to be able to remember tag bigrams and trigrams, as well as word-tag pairs (this is implemented on the train method). Based on that, it calculates the likelihood of a sentence by using the following formula:

The HmmTagScorer It calculates each individual value (q\*e) and returns it in the log space in the getLogScoreCounter method. This increased the overall word accuracy by almost ~2%, as well as UNK accuracy by ~10%.

### Problem1.Part2 – Implement unknown handling

See class HMMTagScorerWithUnknownWordClasses.

Here are the classes that were implemented. Instead of using the most used tag for unknowns, we’ll look at the UNK classification for each bucket and assign probabilities based on tag distribution on that word bucket. A test was developed for this classifications: See Main.TestPseudoWordClassifier and its output.

Word: 1934 Bucket: <fourdigit>

Word: 12.3874 Bucket: <decimal>

Word: 12 Bucket: <num>

Word: Jorge Bucket: <initcap>

Word: 11/11/12 Bucket: <date>

Word: jorge Bucket: <lowercase>

Word: J. Bucket: <uppercasedot>

Word: JORGE Bucket: <uppercase>

Implementing this with HMM brought tag accuracy up to ~60%, thus increasing overall accuracy by ~0.3%. Suboptimalities emerged as a result of using HMM with the Greedy decoder.

## Problem1.Part2 (extra) Suffix trees

See class HMMTagScorerWithSuffixTrees.

Using the suffix of a word, we can implement a method for handling UNKs. The main idea is the following:

Where θ is the standard deviation of the unconditioned maximum likelihood probabilities of the tags in the training corpus – it is usually between 0.03 and 0.10. For the purposes of this assignment, I chose 0.05. Pml is the maximum likelihood estimate with counts form the training set. Note that P is a recursive call.

Using this method (considering the last 3 characters improved accuracy from the MostFrequentTagScorer around ~2.3% and UNK tag accuracy by ~10%. Suboptimalities also emerged.

## Problem2 – Building a sequence decoder

See class VitterbiDecoder.

Using the Vitterbi decoder effectively removed all suboptimalities and improved tag accuracy by ~0.3%, as well as UNK tag accuracy by ~7%, at the cost of performance. Vitterbi finds the best possible path, given transition probabilities in a Trellis Graph.

## Comparing different approaches

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Tag Accuracy** | **UNK tag accuracy** | **Suboptimalities** | **Performance (ms)** |
| MostFrequentTag + Greedy | 92.31% | 38.85% | 0 | 1739 |
| HMMTagScorer + Greedy | 94.11% | 49.93% | **620** | 2048 |
| HMMTagScorerWithUnknownWordClasses + Greedy | 94.41% | 61.29% | **573** | 2085 |
| HMMTagScorerWithWordSuffix + Greedy | 94.47% | 64.03% | **559** | 2625 |
| HMMTagScorerWithUnknownWordClasses + Vitterbi | 95.42% | 67.91% | 0 | 2294 |

The best approach is naturally HMM tag scorer with Vitterbi, which produces no suboptimalities. Word suffix offers a similar accuracy than unknown word classes, while the Greedy decoder coupled with the MostFrequentTag scorer has the lowest accuracy. Only HMMTagScorers without Vitterbi have suboptimalities.

Unknown/low frequency word handling is definitely a big contributor to accuracy. For the purposes of this assignment, I considered tags for unknown words as the first tag of each word that came in the training). Increasing that number (considering the first N tags of each word that came into the training) to 5 actually reduces the overall accuracy of the tagger by 0.02%.

Changing the smoothing values for the suffix scorer (and considering more letters of the end of the word), adding more buckets to the unknown word classes approach can in general improve the score of the tagger. Another thing we could do is apply smoothing to q and e and iterate through different values to find optimal Lambda values (with the validation set).

## Smoothing

Smoothing the q and e functions in the HMM would not be nearly as useful as it was for the lexical language model on the past assignment. Unseen word trigrams is a much bigger space than unseen tag trigrams. Since we’re only **comparing** tag options and choosing the most likely one to evaluate our model, smoothing will assign a small probability to the other cases, and in most of the instances, the non-smoothed case will win, making smoothing an option that would impact our performance and not give us a great benefit.

## Testing/development notes

I used IntelliJ for development and modified the main test method to call the POSTaggerTester’s main method with different parameters.

**public class** Test<E> {  
 **public static void** main(String[] args) {  
 POSTaggerTester.*main*(**new** String[]{  
 **"-path"**,  
 **".\\data\\wsj\\"**,  
 *// "-test",  
 // "test"  
 //,"-verbose"* });  
 System.***out***.println(**"Test PASSED."**);  
 }  
}