1. For the HMM Viterbi implementation, a 75 F1 is required. This program reaches 76.89 F1 in less than 1 second. The outputs are shown below:

```
Viterbi decoding took 0.766158 seconds
Labeled F1: 76.89, precision: 4154/4862 = 85.44, recall: 4154/5943 = 69.90
```

   Because all of the log probabilities were represented as numpy arrays, I was able to avoid too many nested loops and was able to rely on matrix addition to calculate the probabilities and store backpointers. Also, by preprocessing words into their word index, I only had to use numpy array indexing to get the proper emission probabilities.

   However, this program did have two error cases on the data where an invalid I sequence was predicted, but I saw on the Teams that this was acceptable. This system could be improved by taking into account more relationships and features, which was the goal of the CRF approach.

2. The required performance for the CRF training and decoding was 85 F1. My implementation reaches 88.05 F1. The training of the CRF took about 1150 seconds, while the Viterbi decoding took around 33.3 seconds. The outputs are shown below:

```
Data reading and training took 1149.515621 seconds
Viterbi decoding took 33.342589 seconds
Labeled F1: 88.05, precision: 5178/5818 = 89.00, recall: 5178/5943 = 87.13
Running on test
Test Viterbi decoding took 30.231422 seconds
Wrote predictions on 3684 labeled sentences to eng.testb.out
```

   For implementation, the probabilities were no longer stored in matrices so I had to use the FeatureBasedSequenceScorer to obtain initial, transmission, and emission probabilities. Also, the gradients were calculated and stored in a dictionary, which was able to efficiently encode the sparse features. The CRF Viterbi decoding was also modified, since I had to get initial, transition, and emission probabilities from the FeatureBasedSequenceScorer.

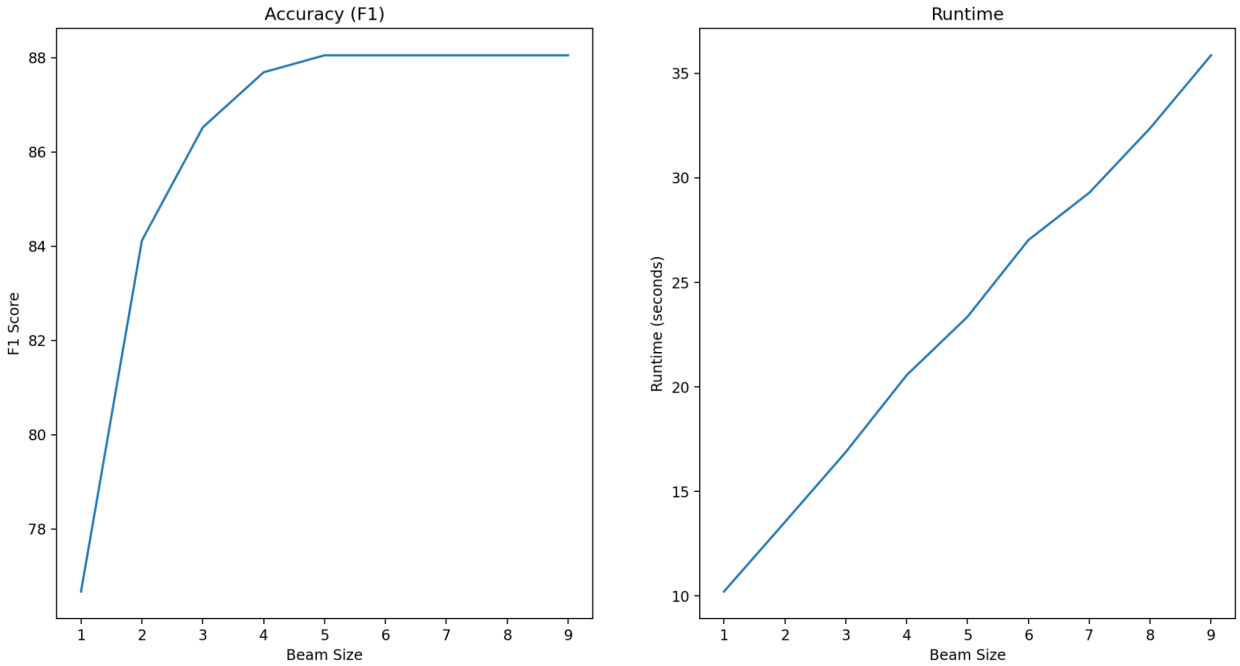
   This system was more stable, and did not predict any invalid tag sequences. To improve this system, I could have encoded more features or tried using a different optimizer.

3. The inference using beam search cannot lose more than 5 F1 compared to the Viterbi algorithm. My implementation with a beam size of 2 lost 3.2 F1, from the 88.04 (Viterbi) to 84.84 (Beam Search) while taking half the time (35 down to 14 seconds). The outputs are shown below:

```
Data reading and training took 1.174507 seconds
Viterbi decoding took 35.235212 seconds
Labeled F1: 88.04, precision: 5177/5818 = 88.98, recall: 5177/5943 = 87.11
Beam decoding took 13.532603 seconds
Labeled F1: 84.12, precision: 4957/5843 = 84.84, recall: 4957/5943 = 83.41
Running on test
Test Viterbi decoding took 31.455971 seconds
Wrote predictions on 3684 labeled sentences to eng.testb.out
```

The implementation of beam search is extremely similar to the Viterbi implementation, so I just made some slight modifications to it. First, instead of storing every exhaustive probability in a KxN matrix, I instead initialized an array of K beam objects to store the scores. Then, instead of running a for loop through each possible previous state, I only had to look at the K largest ones stored in the previous beam data structure.

After trying multiple beam sizes for the CRF model, its impacts on the accuracy and speed were analyzed and plotted below:



From this, it seems like decreasing beam size leads to a logarithmic decrease in F1 score, while leading to a linear decrease in runtime. For example, halving the beam size had almost no impact on accuracy, but led to a near 50% decrease in runtime. This is as expected; with the beam search, we are making a speed-accuracy tradeoff.

Beam search would be more effective relative to Viterbi when the amount of possible states gets much larger. In this model, words belonged to one of 9 states, which isn't particularly large. However in other models where there may be hundreds or thousands of states, this would result in a significant time improvement without losing out on a lot of accuracy. This is because instead of a runtime of $O(TN^2)$, we are decreasing this quadratic N^2 factor to K^2 (K = beam size), which can be a significant runtime improvement for smaller K values.