1 Part 1: Parameter Estimation

In the following, tagToTag is a map. For each tag it stores a map containing each other tag mapped to the number of times it occurs following the first tag. Similarly, tagToWord contains a map for each tag which maps each word to the number of times it is associated with the given tag. The "count" key stores the number of times that a tag appears as a predecessor to another tag in the case of tagToTag or the number of times it appears in total in the case of tagToWord. As part of the preprocessing I simply ignored words which had tags that occurred fewer than roughly 20 times throughout the entire corpus.

1.1 MLE Implementation

```
public Double Transition(String given, String next) {
    double count = tagToTag.get(given).containsKey(next) ? tagToTag.get(given).get(next)
        : 0.0;
    double numerator = count + delta;
    double denominator = tagToTag.get(given).get("count") + delta *
        tagToTag.get(given).get("count");
    double result = numerator / denominator;
    return result;
}

public Double Emission(String word, String tag) {
    double count = tagToWord.get(tag).containsKey(word) ? tagToWord.get(tag).get(word)
        : 0.0;
    double numerator = count + sigma;
    double denominator = tagToWord.get(tag).get("count") + sigma *
        tagToWord.get(tag).get("count");
    double result = numerator / denominator;
    return result;
}
```

1.1.1 Top Ten Words

The following are the top ten words under the tag "NN."

```
company
year
market
trading
stock
program
president
share
government
business
```

These are the top ten most probable tags after the tag "VB."

```
DT
IN
VBN
JJ
NN
NNS
RB
PRP$
NNP
PRP
```

2 Part 2: Viterbi Algorithm for Posterior Inference

2.1 Implementation

In the following a TrellisItem is a simple object that contains a Double value and an int pointer to a cell in the previous row in order to store the best sequence of tags discovered so far. "possibleTags" contains all of the tags that we can predict, i.e. all tags that were seen in the training documents except for "START" which cannot be predicted. The function takes a list of words and returns a list of corresponding tags.

```
public List<String> Viterbi(List<String> wordSequence) {
    TrellisItem [][] trellis = new TrellisItem[wordSequence.size()][possibleTags.size()];
    ArrayList tagSequence = new ArrayList();

    // Initialize first column
    for (int j = 0; j < trellis[0].length; j++) {
        trellis[0][j] = new TrellisItem(Math.log(Emission(wordSequence.get(0),
            possibleTags.get(j)) * Transition("START", possibleTags.get(j))));
    }

    //Fill out remaining columns
    for (int i = 1; i < trellis.length; i++) {
        for (int j = 0; j < trellis[0].length; j++) {
            double maxProb = trellis[i - 1][0].value + Math.log(Transition(possibleTags.get(0),
                possibleTags.get(j)) * Emission(wordSequence.get(i), possibleTags.get(j)));
            int maxTag = 0;
            for (int y0=0; y0 < possibleTags.size(); y0++) {
                double prob = trellis[i - 1][y0].value + Math.log(Transition(possibleTags.get(y0),
                    possibleTags.get(j)) * Emission(wordSequence.get(i), possibleTags.get(j)));
                if (prob > 0) {
                    prob = trellis[i - 1][y0].value + Math.log(Transition(possibleTags.get(y0),
                        possibleTags.get(j)) * Emission(wordSequence.get(i), possibleTags.get(j)));
                }
                if (prob > maxProb) {
```

```
                maxProb = prob;
                maxTag = y0;
            }
        }
        trellis[i][j] = new TrellisItem(maxProb, maxTag);
        }
    }

    //Trace back path
    int maxj = maxCol(trellis, trellis.length-1);
    tagSequence.add(possibleTags.get(maxj));
    for (int i = trellis.length - 1; i > 0; i--) {
        maxj = trellis[i][maxj].pointer;
        tagSequence.add(0, possibleTags.get(maxj));
    }

    return tagSequence;
}

int maxCol(TrellisItem[][] trellis, int row) {
    int maxj = 0;
    double max = trellis[row][0].value;
    for (int i = 0; i < trellis[row].length; i++) {
        if (trellis[row][i].value > max){
          max = trellis[row][i].value;
          maxj = i;
        }
    }
    return maxj;
}
```

2.2 Cross-Validation

2.2.1 Overall

- Accuracy: 0.9889580042
- Precision: 0.8751228193
- Recall: 0.8404957789

2.2.2 NN

- Precision: 0.8968463556
- Recall: 0.8081405155

2.2.3 VB

- Precision: 0.7176518168
- Recall: 0.9060525288

2.2.4 JJ

- Precision: 0.8389728527
- Recall: 0.7213624687

2.2.5 NNP

- Precision: 0.89475758
- Recall: 0.6053340037

2.3 Parameter Configuration

I ran the cross-validation with every combination of values for δ and σ from the set {.01, .1, .5, 2.0, 5.0, 10.0}. I used the overall F1 score as an indicator of the tagger's performance. The standard parameter settings produced a score of 0.8404918618. The best parameter combination that I identified was setting both δ and σ equal to 0.01. This combination produced an F1 score of 0.8553507274. In general it appeared that performance increased slightly as δ decreased but it was very important for σ to be small or performance quickly degraded. For example, leaving δ at 0.5 and changing σ to 10 decreased average F1 score to 0.5541105268. In comparison, leaving σ at 0.1 and changing δ to 10 only decreased the average F1 score to 0.7757359441. I used a paired t-test to compare the performance of the tagger at the default parameter setting and the setting which the experiments suggested to be the best. An online statistics package calculated the p-value to be less than .0001, indicating that there is significant evidence that the average F1 score at the new parameter setting is higher than at the default setting. This does not imply that the new setting is better for any particular class of tags. Further evaluation should be done depending on the context in which the tagger is going to be used.