Natural Language Processing – Prediction of POS Tags using HMM Model (Viterbi Algorithm)

CSE 398/498-013

1. Problem Statement

Given a POS-tagged (labeled/training) corpus and un-tagged (unlabeled/test) corpus, train an HMM model to predict POS tags on the test corpus.

1.1. Three problems and their solutions in HMM

- Likelihood calculation: Given a sentence O and the HMM parameter $\lambda = \{A, B, \pi\}$, find $p(O|\lambda)$. This problem can be solved using the forward algorithm. See Chap 6.3 of SLP2 for more details.
- Decoding: Given a sentence O and the HMM parameter $\lambda = \{A, B, \pi\}$, find a Q*, a sequence of POS tags, to maximize $p(Q|O, \lambda)$. This problem is solved using the Viterbi algorithm. See Chap 6.4 of SLP2 for more details.
- Model training: Given an unlabeled corpus of D sentences $D = \{O_1, \ldots\ldots, O_D\}$, find the HMM parameter $\lambda = \{A, B, \pi\}$ that maximizes $p(D|\lambda)$. This problem is solved using the Baum-Welch algorithm, an example of Expectation-Maximization (EM) algorithm.

1.2. Numerical Issues

During the Viterbi algorithm, we are taking products of many probabilities, leading to infinitesimal numbers that underflows and our computers do not have sufficient precision to represent the numbers. Note that only the relative quantities of $v_t(j)$ matter when maximizing and we can work in the log scale:

$$\log v_t(j) = \max_i[\log v_{t-1}(i) + \log a_{ij} + \log b_{jot}]$$

The maximizing index i* will be the same whether log is taken or not so that the backtracking pointers are not selected. The reconstructed prediction Q* will be the same.

2. Experiments

2.1. Data exploration

Download the small POS-tagged training and test corpora from https://www.clips.uantwerpen.be/conll2000/chunking/. Refer to the website regarding the data format. Report the number of POS-tags, tokens (unigrams), sentences, maximal length of sentences.

2.2. System design and implementations

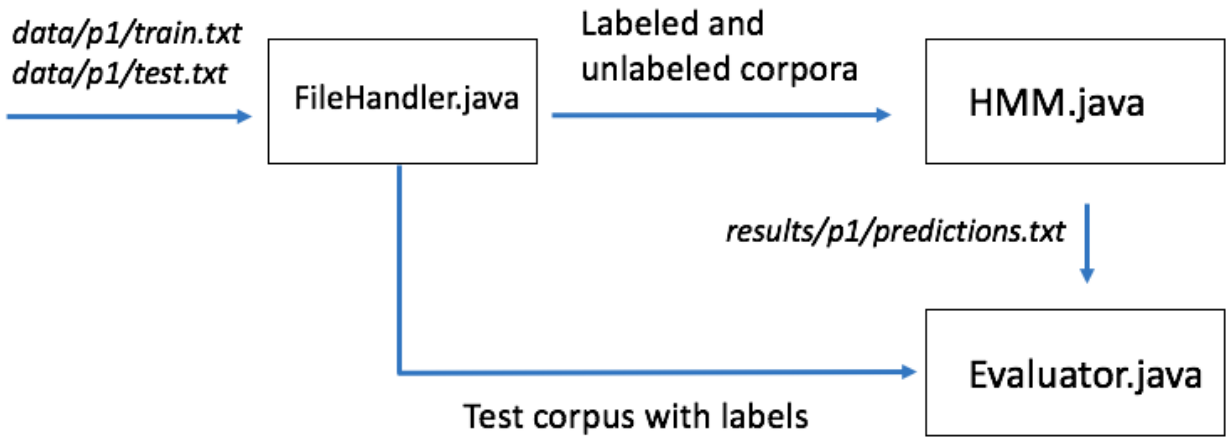

Figure 1: Overall system design.

Figure 1 shows the design. Boxes indicate java classes, arrows indicate flow of data, with names of data over arrows being the input/output (*Italic* names are files on disk, while normal names are Java objects within your program). See the attached bash script and the functions/APIs to understand the flow. You may create other helper files, but I will enforce the formats of those files shown in Figure 1.

The list of Java classes in the project:

- **FileHandler.java** It is provided to you to read and write tagged and untagged sentences. data/p1/train.txt, data/p1/test.txt and results/p1/test predictions.txt will be handled by this class. Note that the prediction files are not that different from the train.txt and test.txt files: the only two differences are: 1) the POS tags in the predictions files are generated by your HMM model instead of given, and 2) there are only two columns (unigram tokens and POS tags) in the prediction files.
- **HMM.java** Implement the Viterbi algorithm. Use FileHandler to read D and L. Initialize $\lambda = \{A, B, \pi\}$ using MLE on L. Matrix class Jama is recommended to represent all matrices and improve the readability of your codes. Numerical issues need to be taken care of as indicated in the problem statement. Produce and output predictions of POS tags for sentences in D to results/p1/predictions.txt.
- **Evaluator.java** This class calculates performance metrics using your predictions (written to a file) and ground truth POS tags (provided in the data/p1/test.txt file). This class will be provided to you.
- **Word.java** This class encapsulates word information including the token, POS tag (in a particular position, if any) and other features of a word. Public methods provide ways to access and modify word information.

- **Sentence.java** The class represents a single sentence, which is a list of Word objects. Note that a corpus can be represented by an array of Sentence objects. Public methods provide ways to access and modify a sentence.

Bash script files (build java.sh and run java.sh) are provided to compile and run the programs. Use ./build_java.sh and ./run_java.sh to compile and run the program respectively. The necessary third party packages are in the folder “jars" in the zipped file (jama.jar in this project).