# Algorithms for AI and NLP (Fall 2017): Obligatory Exercise (3a)

## High-Level Goals

- Understand fully the computation of probabilities in Hidden Markov Models (HMMs);
- Design a data structure and associated functions to train an HMM from tagged training data;
- Implement the Viterbi decoding algorithm; train, test, and evaluate a near-realistic PoS tagger.

## Background

This is part one (of two) of the *third* (and final) obligatory exercise in INF4820. You can obtain up to ten points for this problem set (3a), and you need a minimum of twelve points (or 60,% of the total available) for (3a) and (3b) in total. Please make sure you read through the entire problem set before you start coding (four pages). If you have any questions, please post them on our Piazza discussion board or email `inf4820-help@ifi.uio.no`, and make sure to take advantage of the laboratory sessions.

Answers must be submitted via Devilry by the end of the day (i.e. before 23:59 h) on Thursday, November 2, 2017. Please provide your code and answers (in the form of Lisp comments) in a single '`.lisp`' file.

## Necessary Files

We will be using labeled data from the Penn Treebank (PTB) to train and evaluate our tagger. Furthermore, we provide a 'toy' data file comprising the Eisner ice cream example from Jurafsky & Martin (2008), as well as an auxiliary Lisp function to help you evaluate tagger outputs.

To obtain our data files and code for this problem set, please obtain updates from the SVN repository for the course, e.g. (assuming you opted for the recommended directory location previously):

```
cd ~/inf4820
svn update
```

Inside the new sub-directory '`3a/`' you should see four new files—'`eisner.tt`', '`wsj.tt`', '`test.tt`', and '`evaluate.lisp`—which contain two sets of training data (one small, one large), a smaller amount of test data, and the Lisp evaluation code. Note that '`wsj.tt`' and '`test.tt`' are parts of the PTB, for which UiO holds a license for unlimited research use at the university. Please do not re-distribute these files.

All files ending in '`.tt`' contain part-of-speech tagged 'sentences' (or sequences of observations of ice cream consumption in the case of '`eisner.tt`'). Each sentence consists of a sequence of lines, where each line provides one $\langle o_i, q_i \rangle$ pair, i.e. an observation and its label. On each line, the observation and label are separated by one tab character (which can be referenced by name in Common Lisp as `#\tab`). Sentences are separated by one blank line.

# 1 Theory: Hidden Markov Models

(a) Assume the following part-of-speech tagged training 'corpus' of just one sentence:

| still | , | time | s | move | is | being | received | well | , | once | again | . |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RB | , | NNP | POS | NN | VBZ | VBG | VBN | RB | , | RB | RB | . |

Ignoring smoothing and making the standard simplifying assumptions for a naïve bi-gram HMM (including the assumption that the training corpus provides the full inventory of distinct tags and complete vocabulary), calculate the following:

(i) For each tag $t$, the probability of $t$ following the tag RB, i.e. $P(t|\text{RB})$

(ii) The emission probabilities $P(\textit{move}|\text{NNP})$, $P(\textit{move}|\text{NN})$, and $P(\textit{well}|\text{RB})$.

(b) In a few sentences, discuss the concept of smoothing and explain why it is important. How do the above transition probabilities change when you apply Laplace smoothing? What is the probability of the tag bi-gram $\langle \text{NN}, \text{POS} \rangle$, i.e. a common noun followed by the possessive marker?

(c) In a few sentences, summarize the key points of the Viterbi algorithm. What is the interpretation of each cell in the trellis? What is the complexity of the algorithm, i.e. the number of computations performed in relation to (i) the length of the input sequence and (b) the size of the tag set?

# 2 Representing our Tagger

(a) To represent HMMs in our code, we will again implement an abstract data type using `defstruct`. Define a structure *hmm* that has at least:

- `states`: a structure containing legitimate states (tags) for our model;
- `n`: the number of legitimate tags (i.e. the number of elements in `states`);
- `transitions`: a structure to hold transition probabilities; and
- `emissions`: a structure to hold emission probabilities.

Choosing suitable data structures for `transitions` and `emissions` should be your main focus here. The `transitions` component will be indexed by a pairs of states; we will assign numeric identifiers to states for efficient storage and manipulation. Include a commentary sentence motivating your choice of data structure(s) for the transition matrix.

The `emissions` component is indexed by a state and an observation (word). Why might we not want to assign numeric identifiers to words? In this light, comment on your choice of data structure(s) for storing emission probabilities.

(b) To hide the internals of our HMM implementation, write access functions `transition-probability` and `emission-probability`. Both take an *hmm* object as their first argument, plus either two state ids (`transition-probability`) or a state id and a word string (`emission-probability`) as additional arguments, and return the value stored in `transitions` or `emissions`, as appropriate. Also write a `state2id()` function that takes an *hmm* and a state label (i.e. string) as arguments, and returns an integer id. For states not yet known to the model, this function should allocate a new id, adding the state to the `states` structure and incrementing `n`.

# 3 Reading the Training Data

(a) Write a function `read-corpus` that takes two arguments, a file name corresponding to a labelled training corpus, and an integer, specifying the size of the state set (not counting the initial and final states of the HMM). The function should read the corpus, one line at a time, break each line into its observation and label parts (both represented as Lisp strings), and count state bi-gram and labelled observation frequencies. At this stage, store the counts in the `transition` and `emission` structures. In the next step, we will turn these counts into probabilities.

Since we wish to model start and terminate transitions, your `read-corpus` function needs to pay attention to the empty lines that indicate sentence breaks, and include counts of $(\langle s \rangle, t)$ and $(t, \langle /s \rangle)$ bi-grams to model these.

```
? (setf eisner (read-corpus "~/inf4820/3a/eisner.tt" 2))
→ #S(HMM :STATES ("<s>" "H" "C" "</s>") :N 4 :TRANSITIONS ??? :EMISSIONS ???)
```

It will be important to have the correct counts, hence test your implementation of `read-corpus` and access functions on our sample file '`eisner.tt`' carefully. Here is what we would expect at this point:

```
? (transition-probability eisner (state2id eisner "<s>") (state2id eisner "H"))
→ 77
? (transition-probability eisner (state2id eisner "C") (state2id eisner "H"))
→ 26
? (transition-probability eisner (state2id eisner "C") (state2id eisner "</s>"))
→ 44
? (emission-probability eisner (state2id eisner "C") "3")
→ 20
```

(b) Once you have all the (correct) counts, what remains to be done is to alter the values in the `transitions` and `emissions` components to be relative frequencies, which are estimates of the probabilities $P(s_i|s_{i-1})$ and $P(o_i|s_i)$. Write a function `train-hmm` that, given an HMM recording frequency counts, destructively modifies the transitions and emissions components. For each transition and emission probability, `train-hmm` should retrieve the corresponding counts, compute the probability, and change the relevant entry in the `transitions` and `emissions` data structures. Test to make sure your probabilities look correct, for example:

```
? (setf eisner (train-hmm (read-corpus "~/inf4820/3a/eisner.tt" 2)))
? (transition-probability eisner (state2id eisner "C") (state2id eisner "H"))
→ 13/68
? (emission-probability eisner (state2id eisner "C") "3")
→ 5/34
```

# 4 Tagging with Viterbi

(a) Implement the Viterbi algorithm as a function `viterbi` that takes two arguments, an HMM and an input sequence, and returns the most probable sequence of state labels. You may find Chapters 5 and 6 in Jurafsky & Martin (2008) a useful reference, in addition to your lecture notes.

Internally, the function should make use of two two-dimensional matrices: the so-called Viterbi *trellis* to record, for each state $s$ and each time point (i.e. input position) $t$, the maximum probability of being in state $s$ at time $t$; and the *backpointer* matrix that records the best path to each state. Ensure your implementation can handle unseen data. Since we are not implementing a dedicated smoothing technique, a simple option is to return a small default value for unseen data (e.g. `1/1000000`). Again using the Eisner corpus, test to see that your function does the right thing:

```
? (viterbi eisner '("1" "1" "3" "3" "3" "3" "1" "1" "1" "1"))
→ ("H" "H" "H" "H" "H" "H" "C" "C" "C" "C")
```

(b) Now that it looks like your code works, we are ready to test on a real data set used commonly for PoS tagging research for English. As we move to much larger data volumes, make sure to compile all your code. Create an *hmm* using the 'wsj.tt' file and test your viterbi() function on a single sentence:

```
? (setf wsj (train-hmm (read-corpus "~/inf4820/3a/wsj.tt" 45)))
? (viterbi wsj '("No" "," "it" "was" "n't" "Black" "Monday" "."))
→ ("UH" "," "PRP" "VBD" "RB" "NNP" "NNP" ".")
```

The file 'evaluate.lisp' contains a function evaluate-hmm() that takes an HMM and the name of a file containing tagged test data, and calls the viterbi() function on the observation sequences in the test data. Use it to test your implementation:

```
? (evaluate-hmm wsj "~/inf4820/3a/test.tt")
→ 0.95744324
```

What accuracy does it return? If your accuracy is below 95%, this may indicate remaining problems in your implementation. In our examples this far, we have used regular probabilities here to make debugging easier, but any practical system would use log-probabilities, i.e. perform numeric computations in logarithmic space. In a couple of sentences explain why log-probabilities are preferable, and implement the few changes required to your implementation to take advantage of logarithmic space (in case you had not done so all along).

**Happy coding!**