Algorithms for AI and NLP (Fall 2011, Assignment 1c)

Goals

1. Understand fully the computation of probabilities in Hidden Markov Models (HMMs);
2. design a data structure and associated functions to train an HMM from tagged training data;
3. implement the Viterbi decoding algorithm; investigate smoothing; train and test a PoS tagger.

1 Bring up the Editor and the Allegro Common Lisp Environment

- As always, start our integrated Lisp development environment, by typing (at the shell prompt):

```
acl &
```

- For this session, we provide three files—'`eisner.tt`', '`wsj.tt`', and '`test.tt`'—which contain two sets of training data (one small, one large) and a smaller amount of test data. Our course setup should have copied these files into your IFI home directory. Note that '`wsj.tt`' and '`test.tt`' are parts of the Penn Treebank, 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 observed 'word' combined with its class label (or 'tag'). For simplicity, we will use the (PoS tagging) notation $\langle w_i, t_i \rangle$ in the following, thus also referring to observations of ice cream consumption as 'words', and to temperature indicators for the days in Baltimore, MD, as 'tags'. On each line, the word and tag are separated by one tabulator character. Individual characters can be referenced by name in Common Lisp, where the name for the tabulator character is `#\tab`. Sentences are separated from each other by one empty line, and because we will be tagging one sentence at a time, it will be important to pay attention to the sentence structure. Some tags will be more likely to start a sentence, others may be more likely to end one. Take a quick look at '`eisner.tt`' to make sure you understand the file format.

  Much like for the previous problem set, we ask that you create a new file '`exercise1c.lsp`'. Make sure to use emacs commands like 'M-C-x' or 'C-c C-b' to interactively load your code from '`exercise1c.lsp`' into the Lisp system; remember that, after each change you make in the file, you need to re-evaluate the code before your changes will take effect.

2 Theory: Hidden Markov Models

Assume the following part-of-speech tagged training 'corpus'

hvis	jeg	hadde	hatt	min	gamle	hatt	,	s	hadde	jeg	hatt	hatt	.
CONJ	PRN	AUX	VB	POSS	ADJ	NN	DL	CONJ	AUX	PRN	VB	NN	DL

(a) In a few sentences, discuss the concept of smoothing and explain why it is important. Next, ignoring smoothing and making the standard simplifying assumptions for a bigram HMM, calculate the following:

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

  (ii) For each word w, the probability of w given tags CONJ or VB, i.e. $P(w|\text{CONJ})$ and $P(w|\text{VB})$.

(b) Assuming further that our training corpus demonstrates our complete set of PoS tags, and further assuming that for each tag t $P(t|\texttt{<s>}) = P(t)$, construct part of the Viterbi trellis for tagging the sentence '*jeg hadde hatt .*' Rather than calculating all values, indicate the total size of the trellis and the computations for filling in the first two columns.

(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? Discuss the naïve method of computing the most probable tag sequence t_1^n, given an input string w_1^n very briefly; state how the Viterbi algorithm improves over this approach.

3 Realizing and Reading HMMs

(a) To represent HMMs in our code, we implement an abstract data type. Define a structure *hmm*, with at least the following components *tags*, *n*, *transitions*, and *emissions.* We will use the *tags* component to record—as a set of symbols (implemented as a list)—the legitimate tags (i.e. state labels) for our machine; The *n* component will record the total number of states, i.e. the size of *tags* (including the initial and final states). The *transitions* and *emissions* components in *hmm* will hold the transition and emission probability.

Finding suitable data structures for these components will be essential to our implementation. The transition matrix will be used to look up $P(t_i|t_{i-1})$, the probability of transitioning from state t_{i-1} to t_i. Even though the state labels are tags (which we will represent as strings), it will be convenient to introduce numeric state *identifiers*, i.e. consecutively numbered integers between 0 and $n-1$. What will be a suitable data structure to internally represent the transition matrix, both in terms of storage size and efficiency of access?

The *emissions* component, on the other hand, will be used to look up $P(w_i|t_i)$. Hence, we will need to index it by a state and a word. Much like the tags, we will represent words as strings. Would it be feasible to introduce numeric word identifiers, and if so, what would be the expected numeric range. If not, how else can we provide an efficient way of indexing data, using strings as the keys? Again, reflect on both the use of storage size and efficiency of access. Choose a suitable data structure or combination of data structures to internally represent the *emissions* component.

(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 identifiers (`transition-probability()`) or a state identifier and a word string (`emission-probability()`) as additional arguments. Write these functions to retrieve the appropriate values, based on your choices for the *transitions* and *emissions* data structures above. Further, write a function `tag-to-code()` that takes two arguments: an HMM and a tag (or state label, i.e. a string); `tag-to-code()` returns the integer identifying the state corresponding to the tag. For tags not yet known in the HMM, `tag-to-code()` should further 'allocate' a new state identifier, i.e. add the tag to the *tags* component, increment *n* in the HMM, and return the resulting new state identifier.

(c) Write a function `read-corpus()` that takes two arguments, a file name—corresponding to a tagged training corpus—and an integer, specifying the size of the tag 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 its the w_i and t_i parts (both represented as Lisp strings) and count bi-gram and tagged word frequencies. In other words, we will eventually estimate transition and emission probabilities as follows:

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$$

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$

Thus, while reading the training data, we will need to keep track of these counts. It turns out that the indexing facilities of our *transitions* and *emissions* components are exactly what we need to efficiently count $C(t_{i-1}, t_i)$ (bi-gram frequencies) and $C(t_i, w_i)$ (tagged word frequencies). For the purpose of reading and training our HMM, we will allow ourselves to temporarily use the *transitions* and *emissions* data structures to initially keep track of the actual counts. In a later step, we will then actually 'train' and estimate probabilies from these counts.

Our function `read-corpus()` should update these counts for each line. However, it also needs to pay attention to the initial and final states of the HMM, which we will model as (pseudo-)tags `<s>` and `</s>`.

For each ‘sentence’ read from the file, `read-corpus()` should record all transitions seen among the tags of that sentence, plus one additional transition from the initial state to the first tag, and another one from the last tag of the sentence to the final state.

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

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

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

The training data in ‘`eisner.tt`’ is a sample of 100 weather observations for sequences of days in Baltimore, MD, throughout the summers of 1993 and 2006. But weather records for those years are incomplete, and so is the diary of our colleague Jason Eisner. Hence, the amount of training data—sequences of days for which both the temperature and ice cream consumption by Jason are known—is quite limited. Each sequence is only between three and six days long, which might make our estimates of transition probabilities somewhat unreliable, specifically for the ‘start’ and ‘end’ probabilities, i.e. the likelihood of beginning or completing a sequence of days on a hold or cold day, respectively.

Nevertheless, as true empiricists we will work with the data we have available. Compare our actual counts to the probability estimates provided by Jurafsky & Martin (2008). Given our training corpus, what is the likelihood of starting a sequence of days on a hot day? Conversely, what is the chance of a hot day following a cold day? Finally, note that our model is different from that of Jurafsky & Martin (2008) in another respect. The counts recorded by our `read-corpus()` include transition frequencies for going from either hot (`H`) or cold (`C`) into the final state (noted as `</s>`). In other words, we model the likelihood of the last day in each sequence being hot or cold. For the task of climate research based on the diary of Jason Eisner, one might argue that the probability of a given class occuring first or last in a sequence may be hard to grasp intuitively. For PoS tagging of words in sentences, on the other hand, we have strong intuitions that both sentence-initial and -final position will impact the likelihood of various parts of speech occuring in these positions.

(d) Once you have all the (correct) counts, what remains to be done is to re-compute the values in the *transitions* and *emissions* components according to the fomulae for estimating probabilities $P(t_i|t_{i-1})$ and $P(w_i|t_i)$. 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 actual probability, and change the relevant entry in the *transitions* and *emissions* data structures. Once we have invoked `train-hmm()` on our Eisner HMM, we would expect values as follows (remember that `1/3` is a valid numeric data type in Lisp, a rational number):

```
? (transition-probability eisner (tag-to-code eisner "C") (tag-to-code eisner "H"))
→ 13/68
? (emission-probability eisner (tag-to-code eisner "C") "3")
→ 5/34
```

Note that making `train-hmm()` *destructive* means that it permanently alters the data recorded in the HMM. Thus, for repeated testing, make sure to re-construct a fresh, unmodified HMM containing the original corpus counts, e.g.

```
? (setf eisner (train-hmm (read-corpus "~/eisner.tt" 2)))
```

4 Implementing the Viterbi Algorithm

(a) Based on your extensive reading of (parts of) Chapters 5 and 6 in Jurafsky & Martin (2008), implement a function `viterbi()`. This function takes two arguments—an HMM and an input sequence—and returns the (labels of the) most probable state sequence corresponding to the input. For example:

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

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. You will need to initialize the first column of the trellis, then fill in all remaining columns based on the recursive definition for $v_t(j)$ given by Jurafsky & Martin (2008), and finally find the most probable path from the 'rightmost' column of the trellis into the final state. Out of the n possible values in computing each trellis entry (with n being the size of the tag set), we want each cell to record the largest such value, i.e. in each step (in the innermost iteration) you will need to compare the existing cell value (if any) to the new value, updating the cell whenever the new value is larger than the existing value. There are many good ways of implementing the core of the trellis computation, but three nested usages of `loop()` may well yield a workable skeleton. What should be the start and end values for each iteration?

Alongside the computation of trellis entries, we also need to keep track of the actual path through the trellis that resulted in the maximal probability for the sequence as a whole. Another two-dimensional matrix, indexed parallel to the trellis, can serve to record what Jurafsky & Martin (2008) call *back pointers*, i.e. for each combination of a state s and time point t, we record the state identifier (at time $t-1$) that led to the maximum trellis probability for s and t. Once `viterbi()` has computed all trellis values, including final transitions into the final state `</s>`, it can use the back pointers to retrieve the sequence of state identifiers corresponding to the most probable path through the trellis, and it can then 'reverse map' each state identifier into the actual state label, i.e. PoS tag.

5 A More Realistic Part of Speech Tagger

(a) Once you have confirmed that your HMM implementation, training, and Viterbi algorithm all do the right thing, we want to use the machinery to train and test an actual part of speech tagger. The file '`wsj.tt`' contains the results of hand-annotating a little more than one million words of English newspaper text (taken from Wall Street Journal articles of the late 1980s) with parts of speech drawn from an inventory of 45 distinct syntactic categories; this data is part of the so-called Penn Treebank, or PTB (Marcus et al., 1993). In '`wsj.tt`' we provide the standard choice of training data from the PTB (Sections 2 – 21), and '`test.tt`' provides the first 500 sentences of the standard test data (Section 23). Read the corpus counts from '`wsj.tt`' and estimate transition and emission probabilities. This computation should maybe take a few seconds, but not much more than that. If your implementation appears to take substantially more time on reading the corpus and computing probability estimates, go back to the design of data structures for the *transitions* and *emissions* components in your *hmm* structure.

```
? (setf wsj (train-hmm (read-corpus "~/wsj.tt" 45)))
```

To evaluate our PoS tagger, we will invoke it on previously unseen sentences, compute the Viterbi (most probable) tag sequence, and compare the actual tagger output to what is called the 'gold standard', i.e. the hand-annotated PoS tags in the Peen Treebank. Try a few single sentences, e.g.

```
? (viterbi wsj '("No" "," "it" "was" "n't" "Black" "Monday" "."))
```

In case your `viterbi()` implementation encounters problems with unseen inputs, identify and correct the problem(s)? Make sure that all transition and emission probabilities, even for previously unseen combinations, are properly defined, but avoid probabilities of 0. Instead, consider giving a tiny 'default' value (e.g. $1/1,000,000$) to probabilities that would otherwise be zero. Compare your results to the gold-standard PoS tags in '`test.tt`'; what percentage of actual tag assignments (produced by our tagger) are correct according to the gold standard?

(b) The percentage of correct tags in the tagger output is called the tagging accuracy. Write a function `evaluate-hmm()` that takes an HMM and the name of a file containing tagged test data. We want the function to read all sentences from the file, tag the sequence of forms for each sentence, compare the `viterbi()` results to the gold-standard tags, and count the total number of tag assignments and the overall number of correct tag assignments. Note that our current implementation of Viterbi decoding still is comparatively expensive in terms of time and memory usage; this is owed to our excessive use of full-precision rational numbers, which can be expensive to multiply. Therefore, testing on the full '`test.tt`' may take a little while, though should complete in a matter of a couple of minutes. What tagging accuracy do you obtain on '`test.tt`'? If you land below 95 %, this may indicate that there are remaining problems in your implementation.

6 Submitting Your Results

Please submit your results in email to Johan ('`johanbev@ifi.uio.no`'), Jonathon ('`jread@ifi.uio.no`'), and Stephan ('`oe@ifi.uio.no`') before the end of the day on Monday, October 24. Please provide all files that you created as part of this exercise (e.g. minimally '`exercise1c.lsp`'), including all code and answers to the questions above. Note that it is good practice to generously document your code with comments (using the '`;`' character).