Homework 10: Hidden Markov Models

TAs in charge: Ming, David

Due Date: Wednesday, April 15th 2015, 11:59pm EST

Office Hours:

	David Klaper GHC 5508	Ming Sun GHC 6223
Thu 4/9	4.00-5.00pm	2.30-3.30pm
Fri 4/10	5.00-6.00pm	1.30-2.30pm
Sat 4/11	-	-
Sun 4/12	6.00-7.00pm	-
Mon 4/13	5.00-6.00pm	4.00-5.00
Tue 4/14	8.00-9.00pm	1.00-2.00
Wed 4/15	5.00-6.00pm	4.00-5.00

1 Collaboration Policy

Some of the homework assignments used in this class may have been used in prior versions of this class, or in classes at other institutions. Avoiding the use of heavily tested assignments will detract from the main purpose of these assignments, which is to reinforce the material and stimulate thinking. Because some of these assignments may have been used before, solutions to them may be (or may have been) available online, or from other people. **It is explicitly forbidden to use any such sources, or to consult people who have solved these problems before. It is explicitly forbidden to search for these problems or their solutions on the internet.** You must solve the homework assignments completely on your own. I will be actively monitoring your compliance, and any violation will be dealt with harshly. Collaboration with other students who are currently taking the class is allowed, but only under the conditions stated below.

The purpose of student collaboration is to facilitate learning, not to circumvent it. Studying the material in groups is strongly encouraged. It is also allowed to seek help from other students in understanding the material needed to solve a particular homework problem, provided no written notes are shared, or are taken at that time, and provided learning is facilitated, not circumvented.

The actual solution must be done by each student alone, and the student should be ready to reproduce their solution upon request. In the case of programming assignments, **all code must be written by each student alone.** We will strictly enforce this policy.

Policy Regarding "Found Code":

You are encouraged to read books and other instructional materials, both online and offline, to help you understand the concepts and algorithms taught in class. These materials may contain example code or pseudo code, which may help you better understand an algorithm or an implementation detail. However, when you implement your own solution to an assignment, you must do so *completely on your own, starting "from scratch"*. Specifically, you may not use any code you found or came across.

If you find or come across code that implements any part of your assignment, you must disclose this fact in your collaboration statement.

The presence or absence of any form of help or collaboration, whether given or received, must be explicitly stated and disclosed in full by all involved. Specifically, **each assignment must contain a file named collaboration.txt where you will answer the following questions:**

- Did you receive any help whatsoever from anyone in solving this assignment? Yes / No. If you answered 'yes', give full details? (e.g."Jane explained to me what is asked in Question 3.4").
- Did you give any help whatsoever to anyone in solving this assignment? Yes / No. If you answered 'yes', give full details? (e.g. "I pointed Joe to section 2.3 to help him with Question 2").
- Did you find or come across code that implements any part of this assignment? Yes / No. If you answered 'Yes', please provide full detail (book & page, URL & location within the page, etc.).

If you gave help after turning in your own assignment and/or after answering the questions above, you must update your answers before the assignment?s deadline, if necessary by emailing the TA in charge of the assignment.

Collaboration without full disclosure will be handled severely, in compliance with CMU's Policy on Cheating and Plagiarism.

Duty to Protect One's Work

Students are responsible for pro-actively protecting their work from copying and misuse by other students. If a student's work is copied by another student, the original author is also considered to be at fault and in gross violation of the course policies. It does not matter whether the author allowed the work to be copied or was merely negligent in preventing it from being copied. When overlapping work is submitted by different students, **both students will be punished.**

Severe Punishment of Violations of Course Policies

All violations (even the first one) of course policies will always be reported to the university authorities, will carry severe penalties, usually failure in the course, and can even lead to dismissal from the university. This is not an idle threat - it is my standard practice. You have been warned!

2 Overview

Hidden Markov Models (HMMs) are powerful statistical models for modeling sequential or time series data. The purpose of this assignment is to familiarize you with the underlying statistical model by implementing an HMM and its associated algorithms for the task of Part-of-Speech (PoS) tagging.

2.1 PoS Tagging

Part-of-Speech (PoS) tagging is a central task in Natural Language Processing that is useful for a wide variety of applications. A PoS tag is a label such as "Noun" or "Verb" that is assigned to words based on their function in a context. For example, the word "ate" in the sentence "I ate a pizza" is a verb. What makes the problem of PoS tagging interesting is that words are often ambiguous and can be assigned different labels based on their context. For example the same word "bank" is a noun in the sentence "I went to the bank", while it is a verb in the sentence "I bank on you".

If that was complete gibberish, don't worry! You do not need to be familiar with PoS tags or proficient in linguistics for the purpose of this homework. You can think of words as sequences of observed symbols and PoS tags as unobserved underlying states associated with these symbols. HMMs are then a natural modeling solution to this phenomenon.

2.2 Numerical Underflow

Because we compute a long sequence of probability multiplications. Our numbers will become extremely small. **So small in fact that they won't be represented properly anymore by our standard floating point representations and they will just become 0**. That is a problem.

One solution to this problem is doing the calculations in **logarithmic space**. That means we calculate with log-probabilities instead of probabilities. The only downside is that there is no exact way to perform what was addition in normal space. Thus, we provide you with a formula that will calculate a close approximation of log(a+b) which we call logsum. logsum(log(a),log(b)) will result in something that is close to log(a+b). Also remember that $log(a * b) = log(a) + log(b)$, $log(\frac{a}{b}) = log(a) - log(b)$ and $log(a^b) = log(a) * b$. This should allow you to reformulate the algorithms in terms of log probabilities, such that underflow will not be a problem.

2.3 Note about Formulas

HMMs have been hugely popular in the research literature, leading to several variants that use slightly different terminology or notation, and even interpretation of model parameters. This can lead to marginally differing implementations, that produce potentially varying results. Since this is an autograded assignment, we strongly recommend you exactly follow the algorithms (and their associated notation) used in this hand-out. Failure to do so may lead to (possibly valid) implementations that will be judged as incorrect.

3 Hidden Markov Models

The online slides on the course web site provide an excellent introduction to the HMM and the related algorithms. Here we will summarize the HMM algorithms that are relevant to this assignment.

The main difference between the formulation given here and that in the course slides is the initial state distribution. In the course slides, it is assumed that we have both a fixed starting state and a fixed ending state. Here, we will allow any state to be a starting state and any state to be an ending state2. Thus we will introduce another set of parameters $\Pi = \{\pi_i\}$ for the probability of being in each state at the beginning.

Formally, we define an HMM as a 5-tuple (S,V,Π, A,B), where S $= \{s_0, s_1, ..., s_{N-1}\}$ is a finite set of N states, V $= \{v_0, v_1, ..., v_{M-1}\}$ is a set of M possible symbols in a vocabulary, $\Pi = \{\pi_i\}$ are the initial state probabilities, A $= \{a_{ij}\}$ are the state transition probabilities, B $= \{b_i(v_k)\}$ are the emission probabilities. We use $\lambda = (\Pi, A, B)$ to denote all the parameters. The meaning of each parameter is as follows:

- π_i - the probability that the system starts in state i at the beginning
- a_{ij} - the probability of going from state i to state j
- $b_i(v_k)$ - the probability of "generating" symbol v_k from state i

Clearly, we have the following constraints:

$$\sum_{i=0}^{N-1} \pi_i = 1 \tag{1}$$

$$\sum_{j=0}^{N-1} a_{ij} = 1 \text{ for } i = 0, 1, ..., N-1 \tag{2}$$

$$\sum_{k=0}^{M-1} b_i(v_k) = 1 \text{ for } i = 0, 1, ..., N-1 \tag{3}$$

The three problems associated with an HMM are:

1. **Evaluation**: Evaluating the probability of an observed sequence of symbols $O = o_1, o_2, ..., o_T$ where $o_t \in V$, given a particular HMM, i.e., calculate $P(O|\lambda)$. You will implement this.
2. **Decoding**: Finding the most likely state transition path associated with an observed sequence. Let $Q = q_1, q_2, ..., q_T$ be a sequence of states. We want to find $Q^* = argmax_Q P(Q|O, \lambda)$. You will implement this.
3. **Learning**: Adjusting all the parameters¸ to maximize the probability of generating an observed sequence, i.e., to find $\lambda^* = argmax_\lambda p(O|\lambda)$.

The first problem can be solved by using either the Forward or the Backward iterative algorithms. The second problem is solved by using the Viterbi algorithm, also an iterative algorithm to "grow" the most likely state sequence given the observable sequence O and the model λ. The last problem is solved by the Baum-Welch algorithm (also known as the Forward-Backward algorithm), which uses the forward and backward probabilities to update the parameters iteratively. Instead of implementing it you will answer questions about the quantities involved in the Forward-Backward Algorithm.

4 Data and Files

The archive file hw10-data.tar.gz can be downloaded at `https://autolab.cs.cmu.edu/10601-s15/homework10/handout` and contains several files that you will use in this homework. The contents and formatting of each of these files is explained below.

- **hmm-trans.txt, hmm-emit.txt and hmm-prior.txt** These files contain pre-trained model parameters of an HMM that you will use in testing your implementation of the **Evaluation** and **Decoding** problems. The format of the first two files are analogous and is as follows. Every line in these files consists of a conditional probability distribution. In the case of transition probabilities, this distribution corresponds to the probability of transitioning into another state, given a current state. Similarly, in the case of emission probabilities, this distribution corresponds to the probability of emitting a particular symbol, given a current state. For example, every line in **hmm-trans.txt** has the following format:

  `<Curr-State> <Nxt-State0>:<Prob-Val0> ... <Nxt-StateN>:<Prob-ValN>`

  The format of **hmm-prior.txt** is slightly different and only contains a single probability distribution over starting states. Each line contains the name of a state and its associated starting probability value, like so: `<State0> <Prob-Val0>`.

- **train.txt and dev.txt** These files contains plain text data that you will use in testing your implementation of the **Evaluation** and **Decoding** problems. Specifically the text contains one sentence per line that has already been pre-processed, cleaned and tokenized. You do not need to perform any processing of any kind. You should treat every line as a separate sequence and assume that it has the following format:

`<Word0> <Word1> ... <WordN>`

where every `<WordK>` unit token is whitespace separated. Note that **dev.txt** is the plain text version of dev-tag.txt.

- **dev-tag.txt** This file contains labelled data that you will use to debug your implementation of the **Decoding** problem. The labels are not gold standard but are generated by running our decoder on **dev.txt**. Specifically the text contains one sentence per line that has already been pre-processed, cleaned and tokenized. You should treat every line as a separate sequence and assume that it has the following format:

  `<Word0>_<Tag0> <Word1>_<Tag1> ... <WordN>_<TagN>`

  where every `<WordK>_<TagK>` unit token is whitespace separated.

- **dev-probs.txt** This file contains log-probability values that you will use to debug your implementation of the **Evaluation** problem. Every line in this file is a log-probability value assigned to a sentence in **dev.txt**, under the HMM parameters in **hmm-trans.txt, hmm-emit.txt and hmm-prior.txt**. Specifically, the Kth line in **dev-probs.txt** corresponds to the log-probability of the Kth sentence in **dev.txt** under the model parameters.

- **eval.py** This is an evaluation script that you can use to test your output from decoding against reference POS tagged text. It will be useful in debugging and testing your implementation of the Decoding problem. The command line signature for calling the script is as follows: `python eval.py <ref-file> <sys-file>`, where `<ref-file>` is some POS-tagged reference file, and `<sys-file>` is your system generated hypothesis. Both files need to be in the format of **dev-tag.txt**. An evaluation score is assigned to `<sys-file>` based on the number of sentences tagged correctly with respect to `<ref-file>`.

- **logsum.py** This script contains a function that allows summing of exponentiated logarithmic numbers to a very high degree of precision. It will be useful in your implementation, with regards to the problem of numerical underflow you will likely face. While the sample function is in Python, equivalent Java code should be fairly easy to implement. For example, check out the Java.lang.Math.log1p() method, which is a necessary building block for this function.

Note that the data provided to you is to help in developing your implementation of the HMM algorithms. Your code will be tested on Autolab using different data with different HMM parameters, likely coming from a different domain — although the format will be identical.

5 Evaluation: Forward and Backward Algorithms (40 points)

Your task is to implement both the forward and the backward algorithms. Define $\alpha_t(i) = P(o_1, ..., o_t, q_t = s_i|\lambda)$ as the probability that all the symbols up to time point t have been generated and the system is in state s_i at time t. The α's can be computed using the following recursive procedure:

1. $\alpha_1(i) = \pi_i b_i(o_1)$ (Initially in state s_i and generating o_1)
2. For $1 \leq$ t $<$ T , $\alpha_{t+1}(i) = b_i(o_{t+1}) \sum_{j=0}^{N-1} \alpha_t(j)a_{ji}$ (Generate o_{t+1}, and we can arrive at state s_i from any of the previous states s_j with probability a_{ji})

Note that $\alpha_1(i)$, ..., $\alpha_T(i)$ correspond to the T observed symbols. It is not hard to see that $P(O|\lambda) = \sum_{i=0}^{N-1} \alpha_T(i)$, since we may end at any of the N states. This is slightly different from the course slides, where $P(O|\lambda) = \alpha_T(s_N)$ as s_N is assumed to be the only ending state. Also note that there are N+1 states in the course slides, whereas we have N states.

Another way to solve the evaluation problem is the backward algorithm. Both of them are needed to train an HMM and you will also implement the backward algorithm. Just as it says you will compute the probabilities backwards.

The backward probabilities are called β. Define $\beta_t(i) = P(o_{t+1}, ..., o_T|q_t = s_i, \lambda)$ as the probability of generating all the symbols after time t, given that the system is in state s_i at time t. Just like the α's, the β's can also be computed using the following backward recursive procedure:

1. $\beta_T(i) = 1$ (All states could be ending states)
2. For $1 \leq$ t $<$ T, $\beta_t(i) = \sum_{j=0}^{N-1} \beta_{t+1}(j)a_{ij}b_j(o_{t+1})$ (Generate o_{t+1} from any state)

For getting the total probability of the backwards algorithm you calculate $\sum_{i=0}^{N-1} \pi_i * b_i(o_1) * \beta_1(i)$.

You should now implement the Forward and Backward Algorithms in two different files named `alpha.py` and `beta.py` respectively. Both scripts should expect the same command-line arguments and print their outputs to std-out in the same format. The command-line signature of `alpha.py` is given as an example: `python alpha.py <dev> <hmm-trans> <hmm-emit> <hmm-prior>` Here, the four arguments should be in the format of **dev.txt, hmm-trans.txt, hmm-emit.txt and hmm-prior.txt** respectively. Your implementation should treat each sentence (i.e. every line in the input file) as a separate observed sequence and compute its forward or backward table separately. The output of both scripts should be a list of log-probability values to std-out in the format of **devprobs.txt**. The Kth element of this list should correspond to the log-probability of the Kth sentence in `<dev>` under the model parameters in

`<hmm-trans>`, `<hmm-emit>` and `<hmm-prior>` using the Forward and Backward algorithms.

An indication that your implementation is correct is that the outputs from `alpha.py` and `beta.py` are always identical. Also when you run both scripts on **dev.txt** with the model parameters **hmm-trans.txt, hmm-emit.txt and hmm-prior.txt**, they should output a list of values that are the same as the contents of **dev-probs.txt** (a 0.1% relative error is acceptable as round-off difference). A correct implementation of the Forward and Backward algorithms is worth full points on this part of the homework. Correctness of only one of the two will result in partial credit.

6 Decoding: Viterbi Algorithm (40 points)

Your task is to implement the Viterbi algorithm to extract the most likely state given the observable sequence O and the model λ. The Viterbi algorithm is a dynamic programming algorithm that computes the most likely state transition path given an observed sequence of symbols. It is very similar to the forward algorithm, except that we will be taking a "max", rather than a "$\sum$", over all the possible ways to arrive at the current state under consideration.

Let $Q = q_1, q_2, ..., q_T$ be a sequence of states. We want to find $Q^* = argmax_Q P(q|O, \lambda)$, which is the same as finding $Q^* = argmax_Q P(Q, O|\lambda)$, since $P(Q, O|\lambda) = P(Q|O, \lambda)P(O|\lambda)$ and $P(O|\lambda)$ does not affect our choice of q.

The Viterbi algorithm "grows" the most likely path Q^* gradually while scanning each of the observed symbols. At time t, it will keep track of all the most likely paths ending at each of the N different states. At time t + 1, it will then update these N most likely paths.

Let Q^*_t be the most likely path for the subsequence of symbols $O(t) = o_1, ..., o_t$ up to time t, and $Q^*_t(i)$ be the most likely path ending at state s_i given the subsequence O(t), Let $VP_t(i) = P(O(t), Q^*_t(i)|\lambda)$ be the probability of following path $Q^*_t(i)$ and generating O(t). Thus, $Q^*_t = Q^*_t(k)$ where $k = argmax_i VP_t(i)$ and $Q^* = Q^*_T$. The Viterbi algorithm is as follows:

1. $VP_1(i) = \pi_i b_i(o_1)$ and $Q^*_1(i) = (i)$
2. For $1 \leq t < T$, $VP_{t+1}(i) = max_{0 \leq j < N} VP_t(j) a_{ji} b_i(o_{t+1})$ and $Q^*_{t+1}(i) = Q^*_t(k).(i)$, where $k = argmax_{0 \leq j < N} VP_t(j) a_{ji} b_i(o_{t+1})$ and "." is a concatenation operator of states to form a path.

And, of course, $Q^* = Q^*_T = Q^*_T(k)$ where $k = argmax_{0 \leq i < N} VP_T(i)$

You should now implement the Viterbi algorithm in a file named `viterbi.py`. The command-line signature of this script should be as follows:

`python viterbi.py <dev> <hmm-trans> <hmm-emit> <hmm-prior>` Here, the four arguments should be in the format of **dev.txt, hmm-trans.txt, hmm-emit.txt and hmm-prior.txt** respectively. Again, your implementation should treat each sentence as a separate observed sequence and compute its

Viterbi path separately. The output of the script should be printed to std-out and should be in the format of **dev-tags.txt**. An indication that your implementation is correct is that you obtain 0.0% error when you evaluate the output of a run of Viterbi on **dev.txt** against **dev-tags.txt** using the evaluation script **eval.py**. A correct implementation of Viterbi (or a 0.0% error on our held-out test set) is worth full points on this part of the homework. Partial credit will be assigned based on how close your error percentage is to 0.0 on the held-out dataset.

7 Learning: Baum-Welch Algorithm (Theory) (20 points)

In this section we will look at the Baum-Welch algorithm for learning the parameters. Instead of implementing it, we will test your understanding of the quantities involved with several questions.

You will hand in a file called `answers.txt`. Each line will contain 1-4 letters corresponding to your answers. The line number aligns with the question number.

Below is an example of answers.txt for 4 questions. Your answers.txt **must have 6 non-blank lines**

```
a
bd
c
b
```

Your questions start here. The task is to choose all correct answers. Wrong answers will be penalized. However, **you must give at least one answer to every question**, but you cannot get negative points on a question (so you are never worse off giving an answer than giving no answer). The score of this part of your assignment will not be released until after the deadline.

1. The (first-order) Markov assumption states that:

   a) The states are independent
   [✓ b)] The current state is generated based only on the previous state (and not e.g. on the state preceding the previous state)
   c) The observations are independent
   d) The current observation only depends on the previous observation

2. Which of the following independence assumptions hold in an HMM:

   a) The current state i given the previous state h, is conditionally independent of all states that follow i
   [✓ b)] The current state i given the previous state h, is conditionally independent of all states before h

[✓ c)] The current observation o_j given the current state i, is conditionally independent of all other observations
d) The current observation o_j is unconditionally independent of all states except the current state i

3. Which of the following is true about the Baum-Welch algorithm?

a) The Baum-Welch algorithm looks at every training example only once.
[✓ b)] The Baum-Welch algorithm guarantees that the likelihood never decreases.
c) The Baum-Welch algorithm uses the forward algorithm to check that the results of the backward algorithm are correct.
[✓ d)] The Baum-Welch algorithm only finds a local optimum. Thus, rerunning Baum-Welch with different initializations for your parameters might give different results.

4. In the remaining questions you will always see two quantities and decide what is the strongest relation between them. (? means it's not possible to assign any true relation). As such there is **only one correct answer**. What is the relation between $\sum_{i=0}^{N-1}(\alpha_5(i) * \beta_5(i))$ and $P(O|lambda)$? Select only the **strongest** relation that necessarily holds.

a) ?
b) $>=$
c) $<=$
d) $>$
e) $<$
[✓ f)] $=$

Alpha and Beta defined to match!
αt(i) * βt(i) = P(qt=i, O I λ)

Remember αt(i)=P(o1,o2,...,ot,qt=i|λ) and βt(i)=P(ot+1,ot+2,...,oT|qt=i,λ)
Thus αt(i)∗βt(i)=P(o1,o2,...,oT,qt=i|λ)=P(qt=i,O|λ)
If you now sum over all states i you get P(O|λ)

5. What is the relation between $\alpha_5(i)$ and $\beta_5(i)$? Select only the **strongest** relation that necessarily holds.

[✓ a)] ?
b) $>=$
c) $<=$
d) $>$
e) $<$
f) $=$

6. What is the relation between $P(q_4 = s1, q_5 = s2, O|\lambda)$ and $\alpha_4(s1) * \beta_5(s2)$? Select only the **strongest** relation that necessarily holds.

a) ?
b) $>=$
[✓ c)] $<=$
d) $>$

α5(i)∗β6(j) is basically P(q5=i,q6=j,O|λ) without aij∗bj(o6)
Since both those terms must be probabilities they are <= 1
so if you multiply them you can only get something thats less or equal than the alpha beta version.

e) $<$
f) $=$

8 Autolab Submission

Submit a .tar archive containing all your source code, including alpha.py/java, beta.py/java, and viterbi.py/java as well as your answers.txt and collaboration.txt. You can create this file by running `tar -cvf hw10.tar *.py *.txt`. DO NOT put your source files in a folder and then tar the folder, nor name your files anything but as instructed. You must submit the file hw10.tar to the "homework10" link on Autolab.