CARNEGIE MELLON UNIVERSITY - LANGUAGE TECHNOLOGIES INSTITUTE

10601A-Fall 16: Homework #10 - “Hidden Markov Model” [*]

Instructor:
Prof. Roni Rosenfeld

TA-in-charge:
Shaojie (Jerry) Bai (shaojieb@andrew.cmu.edu)

Office Hours:

Thurs. 11/17	5-6:30pm	GHC 5 Commons, Carrell 2
Fri. 11/18	10:30-11:30am	GHC 5 Commons, Table 5
Fri. 11/18	2:30-3:30pm	GHC 5 Commons, Table 5
Sat. 11/19	12:30-2pm	GHC 5 Commons, Table 5
Mon. 11/21	9:30-11am	GHC 5 Commons, Table 6
Tue. 11/22	5-6:30pm	GHC 5 Commons, Table 5

Assigned: Wednesday, 16 November 2016.
Due: 11:59:59pm on Tuesday, 22 November 2016.
Late Penalty: 20% per day.

[*] This assignment is very challenging in terms of the requirements on the understanding algorithms and the implementations tasks. Start early so that you can enjoy your Thanksgiving!

Contents

0 Policy on Collaboration among Students

0.1 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. We 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.

0.2 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.

0.3 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.** If you have additional question regarding code protection, please post your concerns/doubts on Piazza or email your instructors.

0.4 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. By submitting this homework to the Autolab, you automatically signify and agree to the rules outlined above. **You have been warned**!

1 Overview & Preparation

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.

You are strongly recommended to read the following sections before you work on the assignment.

1.1 PoS Tagging Introduction

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.

1.2 Numerical Underflow

Because we compute a long sequence of probability multiplications. Our numbers will become extremely small— in fact, **so small that they won't be represented properly anymore by our standard floating point representations**. That is a problem (for more information on how floating-point is managed by computers, see IEEE 754 standard).

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. In this assignment, you may safely assume that

$$\text{logsum}\big(\log(a), \log(b)\big) \approx \log(a+b)$$

We have provided the logsum implementation in Python (`logsum.py`) and C++ (`logsum.hpp`). Regarding how to use logsum (or even implement it yourself), see section 3 for more details.

Meanwhile, recall that

- $\log(a \cdot b) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^p) = p \cdot \log(a) \quad \text{for } p \in \mathbb{R} \text{ and } a \in \mathbb{R}_{++}$

This should allow you to reformulate the algorithms in terms of log probabilities, such that the precision of floating-point will not be a problem. When computing logarithm yourself, use base e (i.e.

function $\ln(\cdot)$). In most of the programming languages, the default log function in math packages should have base e. For instance, in Python, `math.log(x[, base])` is in fact a natural logarithm.

1.3 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 auto-graded 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. This will also make your debug attempts easier in general.

There is also an FAQ section at the end of this handout which collects some of the popular questions/confusions. Good luck on your assignment, and happy Thanksgiving (in advance)!

2 Hidden Markov Models

In this section, we will summarize the HMM algorithms that are relevant to this assignment. However, for more details about how HMM works, please refer to Roni's video and the slides that have been posted on the course website: HMM slides.

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 state. Thus we will introduce another set of parameters $\Pi = \{\pi_i\}$ for the probability distribution 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 = $\{o_0, o_1, ..., o_{M-1}\}$ is a set of M possible symbols in a vocabulary, Π = $\{\pi_i\}$ are the initial state probabilities, A = $\{a_{ij}\}$ are the state transition probabilities, B = $\{b_i(o_k)\}$ 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
- $b_i(o_k)$ - the probability of generating symbol o_k from state i

Clearly, we have the following constraints:

$$\sum_{i=0}^{N-1} \pi_i = 1 \tag{2.1}$$

$$\sum_{j=0}^{N-1} a_{ij} = 1 \text{ for } i = 0, 1, ..., N-1 \tag{2.2}$$

$$\sum_{k=0}^{M-1} b_i(v_k) = 1 \text{ for } i = 0, 1, ..., N-1 \tag{2.3}$$

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 $Q^* = \arg\max_Q P(Q|O, \lambda)$. You will implement this.
3. **Learning**: Adjusting all the parameters to maximize the probability of generating an observed sequence, i.e., to find $\lambda^* = \arg\max_\lambda p(O|\lambda)$.

In the sections below, you will need to implement the Forward/Backward iterative algorithm as well as the Viterbi algorithm. Then in the section 6, you will answer a few questions regarding certain properties of Baum-Welch algorithm (a.k.a. Forward-Backward algorithm). In general, familiarity with dynamic programming (DP) is not necessary at all for this assignment. However, it does help you better understand the logic behind the algorithm if you can tell the resemblance.

3 Data and Files

The archive file hw10-data.tar.gz can be downloaded at

`https://autolab.andrew.cmu.edu/courses/10601a-f16/assessments/homework10`

under "Download handout". The tarball 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 k^{th} line in **dev-probs.txt** corresponds to the log-probability of the k^{th} 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** and **logsum.hpp** Both files contain 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 and C++, equivalent Java or C code should be fairly easy to implement. For example, check out the `Java.lang.Math.log1p()`, which is a necessary building block for this function.

  To use the `logsum` file, you can import it. In python this can be accomplished by

  ```
  from logsum import log_sum
  ```

  while in C++ you can do

  ```
  #include "logsum.hpp"
  ```

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.

4 Evaluation: Forward and Backward Algorithms (40 points)

Your task is to implement both **the Forward** and **the Backward** algorithms.

4.1 Forward Algorithm

In order to find out the parameters of the model, one way is to use **the Forward algorithm**. In particular, for computation convenience, let's 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 $1 \leq t < T$,

$$\alpha_{t+1}(i) = \underbrace{b_i(o_{t+1})}_{\text{Generate } o_{t+1} \text{ at } s_i} \cdot \underbrace{\sum_{j=0}^{N-1} \left[\alpha_t(j)a_{ji}\right]}_{\text{... come from } s_j \text{ with prob. } a_{ji}}$$

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.

4.2 Backward Algorithm

Another way to solve the evaluation problem is **the Backward algorithm**, which you will also implement. 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 $1 \leq t < T$,

$$\beta_t(i) = \sum_{j=0}^{N-1} \left[\underbrace{\beta_{t+1}(j)a_{ij}}_{\text{Moving on to state } j} \cdot \underbrace{b_j(o_{t+1})}_{\text{... and generate } o_{t+1}} \right]$$

For getting the total probability of the backwards algorithm you calculate

$$P(O|\lambda) = \sum_{i=0}^{N-1} \pi_i \cdot b_i(o_1) \cdot \beta_1(i)$$

because we can start from any of the N states.

4.3 Implementation

You should now implement the Forward and Backward Algorithms in two different files with the programming language of your choice: `forward.{py|java|c|cpp}`, and `backward.{py|java|c|cpp}`, respectively. Both scripts should expect the same command-line arguments and print their outputs to stdout in the same format. The command-line signature of `forward.{py|java|c|cpp}` is given as an example:

```
$ python forward.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 stdout in the format of **devprobs.txt**. The k^{th} element of this list should correspond to the log-probability of the k^{th} sentence in `<dev>` under the model parameters in `<hmm-trans>`, `<hmm-emit>` and `<hmm-prior>` using the Forward and Backward algorithms.

Wondering whether your implementation is correct? An indication that your implementation is correct is that the outputs from `forward.{py|java|c|cpp}` and `backward.{py|java|c|cpp}` 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). Since $\log(1) = 0$, you should expect that all of the answers that get printed out are **negative**. 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.

5 Decoding: Viterbi Algorithm (40 points)

5.1 Viterbi Algorithm

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 $\max(\cdot)$, rather than a $\sum \cdot$, over all possible ways to arrive at the current state.

Let $Q = q_1, q_2, ..., q_T$ be a sequence of states. We want to find

$$Q^* = \arg\max_Q P(Q|O,\lambda) = \arg\max_Q P(Q,O|\lambda)$$

$P(Q,O|\lambda) = P(Q|O,\lambda)P(O|\lambda)$ 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 given the subsequence $O(t)$ **if we require the path to end at state** s_i. Meanwhile, denote $VP_t(i) = P(O(t), Q_t^*(i)|\lambda)$ as the probability of following path $Q_t^*(i)$ and generating $O(t)$. Thus, $Q_t^* = Q_t^*(k)$ if and only if

$$k = \arg\max_i VP_t(i)$$

and $Q^* = Q_T^*$. The Viterbi algorithm is as follows:

1. $VP_1(i) = \pi_i \cdot b_i(o_1)$ and $Q_1^*(i) = (i)$
2. For $1 \le t < T$, compute $VP_{t+1}(i) = \max_{0\le j\le N-1} VP_t(j)\cdot a_{ji}\cdot b_i(o_{t+1})$. Now we should have $Q_{t+1}^*(i) = (\underbrace{\dots}_{Q_t^*(k)}, i)$, where $k = \arg\max_{0\le j<N-1} VP_t(j)a_{ji}b_i(o_{t+1})$

And, of course, $Q^* = Q_T^* = Q_T^*(k)$ where $k = \arg\max_{0\le i\le N-1} VP_T(i)$

5.2 Implementation

You should now implement the Viterbi algorithm to extract the most likely state sequence, in a file named `viterbi.{py|java|c|cpp}`. 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-tag.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-tag.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.

6 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.

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

```
a
bd
c
b
```

Choose all correct answers. Wrong answers will be penalized. In addition, **you must give at least one answer to every question** (i.e. don't leave a blank), 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. Which of the following are true under the (first-order) Markov assumption in an HMM:
   a) The states are independent
   b) The observations are independent
   c) $q_t \perp q_{t-1} | q_{t-2}$
   d) $q_t \perp q_{t-2} | q_{t-1}$
2. Which of the following independence assumptions hold in an HMM:
   a) The current observation o_t is conditionally independent of all other observations given the current state q_t
   b) The current observation o_t is conditionally independent of all other states given the current state q_t
   c) The current state q_t is conditionally independent of all states given the previous state q_{t-1}
   d) The current observation o_t is conditionally independent of o_{t-2} given the previous observation o_{t-1}.
3. Which of the following is true about the Baum-Welch algorithm?
   a) The Baum-Welch algorithm guarantees that the likelihood never decreases.
   b) The Baum-Welch algorithm looks at every training example only once.
   c) The Baum-Welch algorithm only finds a local optimum. Thus, rerunning Baum-Welch with different initializations for your parameters might give different results.
   d) The Baum-Welch algorithm uses the forward algorithm to check that the results of the backward algorithm are correct.

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**.

4. 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) $\leq$

   e) $\geq$

   f) ?

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

   a) $=$

   b) $>$

   c) $<$

   d) $\leq$

   e) $\geq$

   f) ?

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

   a) $=$

   b) $>$

   c) $<$

   d) $\leq$

   e) $\geq$

   f) ?

7 FAQ

This section contains some common questions that students usually have. Since this is a large assignment and there is only a limited # of OH's, the following resources can be extremely helpful:

- Roni's video in Spring 2016 (available on course webpage)
- Course slides on HMM
- Section 4, 5 and 6 of this handout
- This FAQ section

1. **Q: Do we need to print the result in the exact format as In "dev-probs.txt" ? Is there any rounding rule or should we keep the same digits after the decimal point as dev-probs.txt?**

   A: Please do not round your answer. We will handle the precision of your answer in our autograder.

2. **Q: I don't quite understand the Forward/Backward, Viterbi and BW algorithm. Can you summarize and explain them to me briefly?**

   A: First, it's hard to really "briefly" summarize three such important algorithms. Second, there are many students waiting to be helped on Piazza and in OH— so I strongly suggest you first go to Roni's lecture video and the course slides for more information. There are also lots of information online on HMM that can familiarize you with the algorithms quickly. After that, if you are still confused, read this handout and go to the OH.

3. **Q: What are the `dev-*.txt` files?**

   A: They are designed to help you debug. But you don't have to use them if everything goes well— following the algorithms outlined in the slides and this handout should be sufficient to ensure the correctness of your algorithm. Since this assignment is recursively implemented (dynamic programming), debugging can be (a bit) annoying. Be careful!

   Note: Though `dev-*.txt` files are not necessarily used, `dev.txt` should be used. See section 4 "Implementation" part.

4. **Q: The Autolab imposed a huge penalty on my submission! What's wrong?**

   A: Check the following things before you submit:

   - Did you name your solution files correctly? Check the spelling carefully. The three programming solution files should be `[forward, backward, viterbi].{py|java|c|cpp}` and the theory part solution file should be `answers.txt`.
   - Did you submit your `collaboration.txt`, and it is **non-empty**?
   - Is the programming language that you use in this assignment consistent (i.e. you only got to choose one of the four that we support)?

   If you have met all of these but there is still a problem, contact your TA.

5. **Q: How many submissions do we have?**

   A: Read the next page.

8 Autolab Submission

Submit a `.tar` archive containing all your source code, including

- `forward.{py|java|c|cpp}` (the Forward Algorithm)
- `backward.{py|java|c|cpp}` (the Backward Algorithm)
- `viterbi.{py|java|c|cpp}` (the Viterbi Algorithm)
- `answers.txt`
- `collaboration.txt`

Please make sure that the programming language that you choose is consistent across different tasks of this assignment! You can create the tar file by running

```
$ tar -cvf hw10.tar *.{py|java|c|cpp} *.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.

You have up to 25 submissions. Submit *wisely*!