Stat 547 — Assignment 1

Release Date:	Wednesday March 9, 2011
Due Date:	Wednesday, March 23, 2011 at 11:59 PST

You should submit a written report under my office door (LSK 330) as well as a zipped file by email containing the "answer" and "src" folders (do *not* include the other directory to avoid email quotas problems). The report should contain your work for the written questions as well as a summary of what worked/did not work in your experiments.

1 Getting the code and data

First, make sure as early as possible that you can access the course materials.

`http://www.stat.ubc.ca/~bouchard/pri/stat547-assignment1.zip`

The authentication restrictions are due to licensing terms. The username and password have been announced in class, but if for any reason you did not get it, please let me know by email.
Unzip the downloaded file to your local working directory. It contains both the data that you will need, some evaluation code, and some harness code that will help you do the assignment. The harness code is in Java, but you are not absolutely required to do the assignment in Java (the data and the format for the answers are easy to parse and write). Note however that I will only provide support for Java and I do recommend that you use the harness code. If there is an interest, I could prepare a tutorial to get started with Java (you will only need the basics, which are easy to grasp).

2 Technical stuff

You will need the Java Development Kit (JDK) 1.6 or higher, and I recommend to use an IDE. If you already have your Java environment setup, skip to the next section.
Download the latest JDK from

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

and the latest version of Eclipse from

`http://www.eclipse.org/downloads/`

and follow the installation steps. To launch eclipse, you may have to edit the file eclipse.ini in the eclipse folder and add the line

`-vm [path-to-jdk]/bin`

Next, create a new project (Ctrl-N, then under "general", pick "project"). Give a name to the project, uncheck "use default location" and enter the path to the unzipped folder.
In the folder src, you should be able to see the java files you will be using. At this point, there should not be red crosses on the files (eclipse automatically keeps everything compiled in the background).
To run the main() function of a test, "BasicTest.java", which is in the package "exact", right click on its icon in the project explorer, and select "Run Configurations..." Then click on "Java Application" and on the "New launch configuration" in the top left. Then click "Run". You should see "Basic test PASSED".
For future reference, you can also give command line arguments to your program by clicking on "arguments" in the new launch configuration panel and writing the arguments in "program arguments field". You will need this later. Also, for some experiments, you may want to give more memory to your java code: do that by going into "arguments" again, and writing "-Xmx1g" in the "VM argument" field.
If you want to quickly go over the basics of the Java programming language check out these links:

`http://download.oracle.com/javase/tutorial/java/index.html`
`http://download.oracle.com/javase/tutorial/essential/index.html`
`http://download.oracle.com/javase/tutorial/collections/index.html`

3 First question: Exact inference

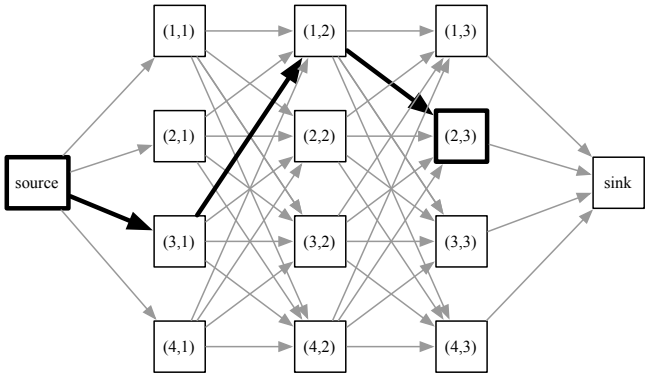
In this question, you will use exact probabilistic inference to estimate sequences of hidden variables from noisy sequences of observations. To give you an idea of what we want to accomplish, run "TestHMM.java." This will launch the worst Part of Speech (POS) tagger in the world. Recall that a part-of-speech is a linguistic category of words such as noun, verb, and adjective. The worst POS tagger tags a sentence by picking a POS uniformly at random for each word. Running "TestHMM.java" will generate a file called "POS.html" in the "answers" folder. Have a look at it to see how bad it is. To understand what the abbreviations mean, see

`http://www.cs.nyu.edu/cs/faculty/grishman/jet/doc/PennPOS.html`

To make it better, we will use the HMM model we have reviewed in class. In this question, we will use a simple maximum likelihood parameters estimate, so the challenge will be to compute the posterior over hidden variables (in this case, POS tags) given observations.

3.1 Theory of exact inference

Before implementing exact inference, let's review the basic theory with the next three exercises.



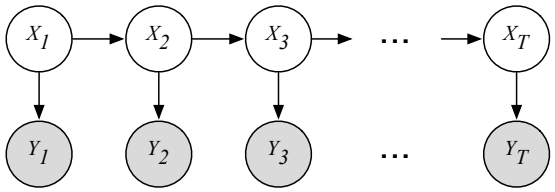
Consider the directed graph above. Vertices in this graph are denoted by $v = (n, t)$ where $t \in \{0, \ldots, T+1\}$ is the horizontal axis, and $n \in \{1, \ldots, N\}$ is the vertical axis on the figure ($N = 4, T = 3$ in the example above). All the vertical layers have N vertices, except for the first (leftmost) and last (rightmost) layers that have only a single element. We call the vertex in the last layer the *sink*, corresponding to $(1, T+1)$, and the vertex in the first layer, the *source*, corresponding to $(1, 0)$. Vertices in adjacent vertical layers are fully connected (i.e. there is an edge $(n, t) \to (m, t+1)$ for all n, m, t). This type of graph is called a *lattice*.

Part A

For a pair of vertices $v = (n, t), v' = (n', t+k)$, $k > 0$, a (directed) *path* is a list of k edges that link v to v'. For example, in the figure, the edges in bold form a path from source to $(1, 2)$. We denote the set of all paths going from vertex v to vertex v' by $\mathcal{S}(v, v')$. The first exercise is to find the number of paths going from the source to the sink, $|\mathcal{S}(\text{source}, \text{sink})|$.

Part B

Suppose now that for each edge e in this lattice, we are given an associated *edge cost* $c(e)$, and similarly, for each vertex v, we are given an associated *vertex cost* $c(v)$. This cost is extended to paths in the following way: for a given path p, the *path cost*, $c(p)$ is defined as the product of the cost of the edges visited in the path, times the cost of the vertices visited, $\left(\prod_{e \in p} c(e)\right)\left(\prod_{v \in p} c(v)\right)$ (by vertices visited, we mean the *set* of all end points of edges, i.e. each vertex is counted only once).
We will now establish the relationship between the lattice and inference in the following directed graphical model:



Assume that we observe a sequence $y_1, y_2, \ldots, y_T$ at the leaves. After introducing notation for the parameters of this model, find an expression for $c(e)$ and $c(v)$ in terms of these parameters such that

$$\sum_{\mathcal{S}(\text{source,sink})} c(p) = \mathbb{P}(Y_1 = y_1, Y_2 = y_2, \ldots, Y_T = y_T).$$

Part C

Next, define

$$B(v) = \sum_{p \in \mathcal{S}(v,\text{sink})} c(p)$$
$$F(v) = \sum_{p \in \mathcal{S}(\text{source},v)} c(p)$$

Convince yourself that the Chapman-Kolmogorov equations can be written with this notation as:

$$B(n,t) = c(n,t) \sum_m B(m,t+1) \cdot c\big((n,t) \to (m,t+1)\big)$$
$$F(n,t) = c(n,t) \sum_m F(m,t-1) \cdot c\big((m,t-1) \to (n,t)\big).$$

Explain how this can be used to efficiently (i.e. in time much smaller than $|\mathcal{S}(\text{source}, \text{sink})|$) compute the moments $\mathbb{P}(X_t = x|\text{obs})$ and $\mathbb{P}(X_t = x, X_{t+1}|\text{obs})$.

3.2 Exact inference in practice

We are now ready to implement the algorithm developed in the previous question to sum over paths in lattices. Open the file "ChainSumProduct.java". It loads some specifications of costs over edges and vertices, and its mission will be to compute the normalization Z (sum of the costs of all paths) and moments (the moment of a vertex v is the sum of the costs of the paths that go through v, divided by Z; and the moment of an edge e is the sum of the costs of the paths that go through e; divided by Z).

Part A

When you run "ChainSumProduct.java", it loads costs for edges and vertices from the folder "data/question1A.gmf" and writes the moments in the same format in "answers/question1A". To check that your code is correct, compare the output to "doc/question1A", they should be equal modulo precision errors. The format used for inputs and outputs (both can be seen as potentials or functions over a chain graphical model) should be easy to understand: for example the file "pair.1-2.mtx" contains a matrix, and entry n, m in this matrix is $c\big((n,1),(m,2)\big)$.

Part B

Once you get it right, change the first line in the function "main()" to true and run it again. It will cause your code to read a larger model and write the results to "answers/question1B". Note that if you haven't done so already, you may have to represent the arrays B and F in logspace to avoid underflows.[1] See the hint in function "computeMoments()" and only exponentiate after subtracting $\log Z$. Double check that you still get "doc/question1A" correctly after migrating your code to logspace.

Part C

You are now ready to use this code to do POS tagging. Open "HMMPosteriorCalculator.java." The responsibility of this part of the code is to transform the HMM model parameters into an undirected graphical model or lattice that ChainSumProduct can work on. This is similar to the graphical model transformation steps we have seen in class.
Using your result in part B of the section on the theory, implement this step. Note that the computation of the MLE from the training data is done for you in the object "data." For example, to access the transition probability, use "SequenceData.getMLETransitionProbability()." Since the code in ChainSumProduct indexes states by integers, and SequenceData.getMLETransitionProbability() is indexed by string, you will have to use "data.getHiddenStateName()" to do the translation.
The last ingredient is to remove the first line of "ExpectedLossMinimizers.java." This line returned a random guess, by removing it, the prediction will be made by calling your code and picking the tag that maximizes the posterior at each node (recall that this is optimal under zero-one loss).
To test your code, run "TestHMM.java." It repeatedly calls your code to infer POS tags. Check "answers/POS.html" to see the improvements. To help you debugging, call TestHMM with the option "-task TOY" This causes TestHMM to use the tiny dataset in "data/TOY", which makes it easier to spot problems by working out the MLE by hand.

[1] There is another, more efficient way to do it called rescaling, but it's more complicated.

Note that many predictions are not attempted, i.e. not approachable by this setup. This is simply because the MLE gives zero probability to some of the test sentences. You can ignore this issue for now, we will address it using Bayesian non-parametric estimators in the next lectures and assignment.

Part D (Optional)

This part explores a different task: secondary structure prediction of proteins. For background, see

`http://en.wikipedia.org/wiki/Protein_secondary_structure`

The observations are sequences of amino acids (again, see the wikipedia article to see the standard encoding used), and the predictions (hidden states) are secondary structures, encoded according to the DSSP format:

`http://swift.cmbi.ru.nl/gv/dssp/DSSP_2.html`

Here we consider a situation where we are only interested in finding out if the current structure, in DSSP format, is either in the set {H, B, E}, or in the complement of the set. We call this loss function alphaBetaLoss.
Start by running TestHMM.java with the option "-task PROTEIN". As in the POS case, the output gets written in "answer/PROTEIN.html", and this is initially a random prediction (Y means that the structure is in {H, B, E}, N otherwise).
Your task is to implement expected loss minimization under alphaBetaLoss. This should be done in ExpectedLossMinimizers.minimizeAlphaBetaLoss().

4 Second question: Approximate inference

4.1 Theory

Part A

Consider the graphical model we used in the previous question, and assume that there is a Dirichlet prior on the parameters. Describe two MCMC moves: one that samples all the sentences at once conditioning on the parameters, and one that samples a single word but collapses the parameters.

Part B

Consider a different prediction problem for part D of the previous question: finding the number of distinct contiguous alpha-beta blocks. For example, in the sequence:

"NNYYNYYYNYYYYYYYNNNN",

the correct answer would be 3. Suppose the loss is the absolute value between the prediction and the truth. How would you approximate the Bayes estimator in this case?

4.2 Implementation

In this question, we will apply approximate inference techniques to a simple Ising model where each variable is binary. We assume that there are location-dependent one-node factors, but that the two-nodes factors are all *agreement potentials*. An agreement potential between two variables with the same value is equal to 2 (a parameter controlled in "TestIsingInference"); and it is equal to one otherwise.

Part A

Open the file "Gibbs.java" in the "approx" package. It contains a function, "computeMoments()", that takes a potential (factor) specification, and should output estimated moments. Note that you only have to estimate the one-node moments in this question. After implementing it, you can test your code by running "TestIsingInference". It reads potentials from the folder "data/question2A.gmf", calls your code, and outputs the moments in the answers folder. The model in "question2A.gmf" is a tiny 2 x 2 model, so you should be able to compute the analytic moments by hand and see if you converge to this value.

Part B

Try now to call "TestIsingInference" with the argument "-task BIG", and trying different values of the argument "-agreementStrength". Use the method of your choice to estimate how many iterations are required to get a good estimate. For the adventurous, show that coupling from the past can be applied to this model, and see if the algorithm terminates in a reasonable time on this problem.

Part C (Optional)

This question is open-ended: describe and try to implement another approximate inference algorithm in "BetterInference.java". This could be for example a block Gibbs sampler using the code you wrote in the first part of the assignment, or a mean-field algorithm. To test your code, change the field "algorithm" to "new BetterInference()" in the file "TestIsingInference.java".