# CS688: Graphical Models - Spring 2016

## Assignment 4

Assigned: April 14. Due: Friday, April 29, 4pm

**General Instructions:** Submit a report with the answers to each question at the start of class on the date the assignment is due. You are encouraged to typeset your solutions. To help you get started, the full $\LaTeX$source of the assignment is included with the assignment materials. For your assignment to be considered "on time", you must upload a zip file containing all of your code to Moodle by the due date. Make sure the code is sufficiently well documented that it's easy to tell what it's doing. You may use any programming language you like. If you think you've found a bug with the data or an error in any of the assignment materials, please post a question to the Moodle discussion forum. Make sure to list in your report any outside references you consulted (books, articles, web pages, etc.) and any students you collaborated with.

For this assignment, you may only use libraries for linear algebra and sampling from multivariate normals.

For this assignment, on the written parts (the math/algorithm derivations, understanding the paper, etc.) **you may work with one partner.** But you have to do your own implementations of the code (it is very educational...) and do the analysis of your own code's behavior. Each person must turn in their own final writeup and say who they worked with (if any).

We're giving slightly more time with final due date on Friday, April 27. If you're turning in a paper copy of the writeup then, ask at the CS main office front desk to turn it in. Or you can finish early by the last class lecture on Wednesday. Or turn it in electronically.

**Academic Honesty Statement:** Copying solutions from external sources (books, web pages, etc.) or other students is considered cheating. Sharing your code solutions with other students is also considered cheating. Any detected cheating will result in a grade of 0 on the assignment for all students involved, and potentially a grade of F in the course.

To be clear, it is OK to read related materials to better understand the problem. Cite your sources. You may find it useful to refer to the original Martin and Quinn C++ code (or not so useful...), though you must write your own new implementation.

For each problem, in your writeup specify the name of the code function that implemented it.

## 1. (*35 points*) **Bayesian linear regression.**

In this problem you'll implement and test one piece of the puzzle: Bayesian linear regression. In your writeup please specify the names of the functions that correspond to the questions. We provided optional starter code in *linreg.py*.

**(a)** (5 points) Implement the linear regression MLE,

$$w^{(MLE)} = (X'X)^{-1}X'Y$$

Check it by comparing against a standard implementation of linear regression on some toy examples. Note that for the intercept term, you'll have to specify a column of $X$ to be all 1s, but most linreg implementations expect an $X$ without it and add it in for you then remove it again when returning the result.

**(b)** (10 points) Implement posterior inference for Bayesian linear regression with the closed form from Murphy section 7.6.1 (equations 7.52–7.58). You should have a function that returns the posterior mean and covariance of the weights, given the data ($X$ and $Y$) and priors ($w_N, V_0$ in Murphy's notation). Feel free to fix the noise variance $\sigma^2 = 1$.
Check this function's correctness informally in two ways. First use a prior of $w \sim N(0, big)$, which should yield a posterior mean that is nearly the same as the MLE. Second, vary the prior and make sure the posterior changes in a way you expect. Report how you designed this check, what your variation was, and describe the results and why they behave as expected.

**(c)** (20 points) Analyze your implementation's correctness with the Cook test. Don't worry about the chi-square hypothesis tests or z-value dotplots; just do the decile histogram visualization.

- Note you don't need to simulate $X$ since linear regression is conditional on it; just make a small toy dataset.
- The Cook et al. paper focuses on MCMC methods. However this is not relevant here. Your Bayesian linear regression method calculates the *exact* posterior. You can directly calculate the posterior CDF for any point (i.e. for the "true" simulated value) by using the univariate normal CDF function from *norm1d.py*.
- It's fine to analyze only a single variable; just choose, say, the first or last weight parameter.
- Concretely, you will want to do something like the following. Fix a weights prior $\mathcal{N}(w_0, V_0)$. For a single simulation, sample $w$ from the prior, and sample each $y_i$ from $\mathcal{N}(w'x_i, \sigma^2)$, then infer the posterior $P(w \mid X, Y, \sigma^2)$ and the CDF of the "true" simulated $w$.

Describe how you designed this test. Use at least 10,000 simulations. If you found a bug from this test, please describe it and show both the histogram under the bug and after it was fixed. If your initial implementation didn't have a bug, introduce one to make sure you get a screwed up histogram (that is, test the test!).
Show the decile histogram plots for both the buggy and correct version. Describe the bug.

Implementation notes:

- In Python, *numpy.random.multivariate_normal()* knows how to simulate from multivariate normals. Try it with some toy examples to make sure you know how it works. Make sure you're aware of whether you're working with a variance or a standard deviation.
- *norm1d.py* in the starter code contains a univariate normal CDF function. It works the same as *scipy.stats.norm(...).cdf()*.

- Warning for future: for this assignment you can just use the normal equations as specified. But these have numerical stability issues; for a more robust implementation, you should never invert a matrix and instead use linear system solvers and tricks with QR and/or LU decompositions, etc. Gelman et al.'s *Bayesian Data Analysis* book has some details on this for the Bayesian linear regression case.

## 2. (*65 points*) **Supreme Court latent space model.**

In this problem, you will replicate the Martin and Quinn (2002) model, with constant positions for each justice. That is, instead of a $\theta_{j,t}$ parameter for each time period for each justice, only use a $\theta_j$ parameter for each justice. Use fixed priors for $\alpha$, $\beta$, and $\theta$. (This model is much simpler than the full M&Q model, though it still has the same key property with the probit link and truncated normal $z$ Gibbs update.)
This will be done with the same data the paper uses; a copy is included here (it's just a reformatted version from the authors' website). You'll see the data is just a big matrix of cases against justices' votes. The first two columns are metadata which you can throw away. Values of -9 indicate missing values; values of 0 and 1 indicate votes to reject or affirm the lower court's decision. Using this data, you should be able to get results very similar to Table 1 of their paper.
The first parts of this problem (a-c) are to carefully describe the model and the algorithm before implementing it. The Martin and Quinn paper is pretty good at this, but writing it yourself, and all in the same place, makes it conceptually clearer. We provide an example writeup of (a-c) for LDA later in this document.

**(a)** (5 points) Write in a table, or as bullet points, every variable and piece of notation for the data/model. For each entry, give a brief one sentence description of that variable or index. This all should fit within half a page so it's a convenient "cheat sheet" for understanding the model and your implementation.

This is good thing because (1) if you build this up while reading a modeling paper (even informally scribbling it on a sheet of paper), it may help you understand better. (2) If you are clear yourself on all the pieces of the model, you are more likely to code it up without bugs. (3) You are more likely to explain it in a clear way to someone else. There are lots of bad papers out there where it seems the authors never though through something like this.

**(b)** (5 points) Write down the model's assumed generative process that created the data.

**(c)** (10 points) Derive the Gibbs sampling update steps you will use and write them out mathematically with enough details such that it is clear how to compute them. For some steps you may want to call out to an external function, such as your Bayesian linear regression implementation from the previous problem; if so, precisely specify what quantities are going to be passed to the function.

**(d)** (20 points) Implement the model. We provide starter code that might help but feel free to write your own. If you use the starter code you should play around with the data first to make sure the data structure makes sense to you.
We suggest writing an update function for each of the Gibbs sampling steps in the previous part. Once you are feeling secure about them, then try running a Gibbs sampler for multiple iterations.

**(e)** (10 points) Run the sampler for at least 1000 iterations. Plot the $\theta_j$ justice parameters over sampling iterations. (Make a second plot just for the earlier iterations, if they look a lot different.) What does its convergence behavior look like?

**(f)** (10 points) Print or plot a summary of the $\theta_j$ parameters: for each justice, their posterior mean and standard deviations (or another summary if you think another is more appropriate; please justify), from samples that you think occur after burn-in seems to be complete.

**(g)** (5 points) Who are the most liberal and most conservative justices? See *justicenames.txt* which contains their last names (same order as columns of the data matrix). Look them or others up on Wikipedia. How consistent are the model's inferences with our (or, perhaps, Wikipedia authors') understanding of history?

Miscelleaneous notes:

- For the $\alpha$, $\beta$, and $\theta$ updates, a good approach is to figure out how to formulate them in terms of a Bayesian linear regression.
- For the $z$ updates with truncated normals, if you're working in Python you may want to use the functions in the supplied *norm1d.py*. It also includes code for normal CDFs and normal inverse CDFs; these work the same as *scipy.stats.norm(...).cdf()* and *.ppf()* but are faster. (Turns out *scipy.stats* is way too slow for MCMC. See Brendan's angry comments in the code.)
- For the $z$ updates, you don't need the third clause of equation 9. The M&Q paper needs it because of their dynamic model, but we aren't doing that part. You never need to ever infer anything about missing values. Missing values can be completely ignored.

**3.** (*20 points*) **Extra credit (up to 20 points).** Change the model to allow justices' preferences to change over time. An easy way to do this is with binning by, say, each 5-year period; a better way is to do binning with a shared prior for the justice across time; a more satisfying but trickier way is to use the dynamic model in the original paper with Gibbs or an FFBS sampler; the best but way too hard way is to use a Gaussian process or other nonlinear temporal prior. Describe your change, implement it, and show the results and analyze how it is different than before.

**4.** (*10 points*) **Extra credit (up to 10 points).** Derive and implement a MAP estimation algorithm for this model, perhaps with gradient descent or coordinate descent. Describe your approach, how it behaves, and how its inferences compare to the Gibbs sampler approach.
And/or implement a variational inference approach. This may be quite difficult to figure out.

**5.** (*10 points*) **Extra credit (up to 10 points).** Implement the model in a probabilistic programming language—such as Stan or Anglican or one of the others. (We suspect Stan is most likely to be successful?) Show the implementation, describe how it behaves and compare to your implementation. Discuss the advantages or challenges of this approach.

## Example model writeup

For latent Dirichlet allocation with a standard (non-collapsed) Gibbs sampler.

(a) Table of all notation and variables.

| Name | Description |
|---|---|
| $D$ | Number of documents in the corpus. |
| $V$ | Number of unique word types (the vocabulary size). |
| $K$ | Number of latent topics. (Fixed hyperparamter.) |
| $d$ | Document index ($d \in 1..D$) |
| $i$ | Token index within a document |
| $n_d$ | Number of tokens in document $d$. |
| $\alpha$ | Latent vector of doc-topic pseudocount priors. Non-negative and sums to one. |
| $\theta_d$ | Latent doc-topic distribution for document $d$. |
| $z_{d,i}$ | Discrete latent topic for the $i$th token in document $d$. $z_{d,i} \in 1..K$. |
| $w_{d,i}$ | Word for the $i$th token in document $d$. $w_{d,i} \in 1..V$. |
| $\phi_k$ | Latent topic-word probability distribution (basically, a weighted dictionary) for topic $k$. Length $V$, non-negative and sums to one. |
| $\beta$ | Topic-word distributions' pseudocount prior. |

**(b)** Generative process assumed by the model.

- For each topic $k = 1..K$, generate $\phi_k \sim Dir(\beta_1..\beta_K)$.
- For each document $d = 1..D$,
  - Generate $\theta_d \sim Dir(\alpha)$
  - For each token $i = 1..n_d$,
    - Choose topic $z_{d,i} \sim Mult(\theta_d)$
    - Choose word $w_{d,i} \sim Mult(\phi_k)$

**(c)** Update equations. We execute these steps in order for one Gibbs sweep.

For each $d$, resample $\theta_d$ from

$$P(\theta_d \mid \alpha, z_{d,1}..z_{d,n_d}) \propto Dir(\theta_d \mid \alpha)\, Mult(z_{d,1}..z_{d,n_d} \mid \theta_d) \tag{1}$$

$$= Dir\left(\left[\alpha_k + \sum_i 1\{z_{d,i} = k\}\right]_{k=1..K}\right) \tag{2}$$

For each token in the corpus, resample $z_{d,i}$ from

$$P(z_{d,i} = k \mid w_{d,i}, \theta_d, \phi) \propto \theta_{d,k}\phi_{k,w_{d,i}} \tag{3}$$

For each topic $k$, where a latent topic word count $n_{k,w} = \sum_d \sum_i 1\{z_{d,i} = k\}1\{w_{d,i} = w\}$ is the number of times word $w$ was seen generated by topic $k$, resample the $\phi_k$ distribution from

$$P(\phi_k \mid w, z, \beta) = Dir\left([\beta_k + n_{k,w}]_{w=1..V}\right) \tag{4}$$

Finally, update $\alpha$ by calling a slice sampler with the unnormalized log-likelihood as the unnormalized posterior; that is, under an improper uniform prior on $\alpha$.

$$P(\alpha \mid \theta) \propto \prod_d P(\theta_d \mid \alpha) \tag{5}$$

$$\ell(\alpha) = \sum_d \sum_k (\alpha_k - 1) \log \theta_{d,k} \tag{6}$$

Don't update $\beta$. Just let it be a fixed hyperparameter.