University of Wisconsin-Madison	Spring 2018
BMI/CS 776: Advanced Bioinformatics	Homework #2
Prof. Anthony Gitter	Due: Thu, Mar 8, 2018 11:59 PM

Assignment goals

- Use mutual information to reconstruct gene expression networks
- Evaluate classifier predictions
- Examine Gibbs sampling for a Markov random field
- Control for multiple hypothesis testing with *q*-values

Instructions

- To submit your assignment, log in to the biostat server **`mi1.biostat.wisc.edu`** or **`mi2.biostat.wisc.edu`** using your biostat username and password.
- Copy all relevant files to the directory **`/u/medinfo/handin/bmi776/hw2/<USERNAME>`** where **`<USERNAME>`** is your biostat username. Submit all of your Python source code and *test that it runs* on **`mi1.biostat.wisc.edu`** or **`mi2.biostat.wisc.edu`** without error. Do not test your code on **`adhara.biostat.wisc.edu`**.
- For the rest of the assignment, compile all of your answers in a single file and submit as **`solution.pdf`**.
- Write the number of late days you used at the top of **`solution.pdf`**.
- For the written portions of the assignment, show your work for partial credit.

Part 1: Mutual information in regulatory networks

In class, we saw how FIRE uses mutual information to detect relationships between sequence motifs and gene expression levels. Mutual information is also a popular technique for reconstructing transcriptional regulatory networks from gene expression datasets[1]. After measuring gene expression levels for all genes in a sufficient number of biological conditions, mutual information can detect some types of pairwise dependencies that may suggest one gene is a regulator (transcription factor) and another is its target. Fluctuations in the regulator's expression can influence the expression levels of the target gene. We can create an undirected gene-gene network by computing mutual information for all pairs of genes. Thresholding the mutual information produces a set of gene-gene edges.

[1] http://link.springer.com/article/10.1186%2F1471-2105-7-S1-S7

For this assignment, we will use the DREAM3[2] network inference challenge dataset. The file **`data.txt`** has a 21 time point simulation of the gene expression levels for ten genes over four simulated replicates. The first line of the file provides the column labels. The first column is the time point, which you will not need. The other columns are the expression levels of the gene named in the first line, where genes are represented with a numeric index. The file is in a tab-delimited format.

***1A**: Mutual information via discrete binning*
Complete the program **`CalcMI.py`** using the provided template that takes as input the expression data for a set of genes over a number of conditions and reconstructs pairwise gene-gene dependencies using mutual information. The program will output the list of gene-gene dependencies and their mutual information. It should only consider the dependencies between pairs of unique genes, not the mutual information of a gene and itself (the entropy of that gene's expression).

Compute mutual information by discretizing the gene expression values, mapping continuous gene expression into discrete bins, as in HW0. For a pair of genes, for example G1 and G2, construct a count matrix that tracks the number of times G1's expression is in some bin a and G2's expression is in some bin b. Add a pseudocount of 0.1 to all entries in the G1:G2 count matrix. From this count matrix, you can compute the terms $P(G1 = a)$, $P(G2 = b)$, and $P(G1 = a, G2 = b)$ needed to calculate mutual information.

You can run your program from the command line as follows:

```
CalcMI.py --dataset=<dataset> --bin_num=<bins> --out=<out>
```

where

- **`<dataset>`** is the name of the text file that contains the gene expression data formatted according to the description above.
- **`<bin_num>`** is an integer that represents the number of bins that should be used to discretize the continuous gene expression data when calculating the mutual information. In this part, you should use a

[2] http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0009202

uniform binning of the gene expression range. For example, if a gene has expression values in the range [1, 11] and **`bin_num = 4`**, the bins would be [1, 3.5), [3.5, 6), [6, 8.5), and [8.5, 11]. Later, you will consider an alternative strategy so make your code flexible.

- **`<out>`** is the name of the text file into which the program will print *all* unique gene pairs and their mutual information values in decreasing order. After rounding mutual information to three decimal places, break ties based on the index of the first gene and then the index of the second gene if needed, sorting genes indexes in ascending order[3]. Each line in the file should contain the undirected gene-gene edge and its mutual information separated by a tab '\t'. The file should be formatted as follows:

```
(6,9)     0.817
(3,10)    0.633

...       ...

(5,8)     0.480
(3,7)     0.463

...       ...

(8,9)     0.427
```

***1B**: Implement a different binning strategy*
In this part, you will try a different binning strategy for the gene expression data. Add a new argument **`str`** to implement the equal density binning strategy. This uses a percentile-based assignment to assign expression values to bins, as in HW0. For example, with **`bin_num = 2`** and **`str = density`**, the lowest 50% of a gene's expression values would be mapped to bin 0 and the highest 50% would be mapped to bin 1.

You can run your program from the command line as follows:

```
CalcMI.py --dataset=<dataset> --bin_num=<bins> --str=density
 --out=<out>
```

[3] See https://docs.python.org/3/howto/sorting.html for sorting tips

***1C**: Mutual information via kernel density estimation*

The definition of mutual information naturally extends to continuous random variables. In this part, you will calculate mutual information of gene pairs on the same data using a kernel density estimator with a Gaussian kernel. Kernel density estimation is a nonparametric method for learning a smooth probability density function (pdf) from observed data. A kernel density estimator for n training points $x_1, \cdots, x_n$ is defined to be

$$\widehat{f_n}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where the *kernel* K enables the smoothing effect and the *bandwidth* h determines how influential a point is in its neighborhood.

To abstract away the technical details of the kernel, you are *required* to use the **SciPy** function **gaussian_kde** for estimating the pairwise joint probability distribution of gene expression levels[4]. Continuous mutual information is defined with a double integral over both variables. To avoid the integration, you will sample the estimated pdf on a 100×100 grid of points uniformly distributed in the region [-0.1, 1.1]×[-0.1,1.1] and use the sampled densities to approximate the estimated pdf[5]. Add 0.001 to the density at each sampling location in the grid to avoid precision error. The marginal probability of a single gene expression level can then be estimated via approximate integration over the other dimension (that is, a finite sum over 100 intervals defined by the grid). Similarly, mutual information of a gene pair can be computed via approximate integration over the region [-0.1, 1.1]×[-0.1,1.1] (that is, a finite sum over 100×100 squares defined by the grid).

You can run your program from the command line as follows:

```
CalcMI.py --dataset=<dataset> --str=kernel --out=<out>
```

[4,5] See https://docs.scipy.org/doc/scipy-0.19.0/reference/generated/scipy.stats.gaussian_kde.html for examples, including **np.mgrid** syntax. The integration will also be discussed on Piazza.

***1D**: Plot the receiver operating characteristic (ROC) curve*

Complete another program `plot.py` to plot the ROC curve[6] for the output of `CalcMI.py`. The gold standard edges, that is, the edges in the true gene-gene network, are tabulated in a file called `network.txt` as two columns. Each line represents an undirected edge between two different genes. The genes in an edge are always listed with the smaller index first. Code for generating the plot and computing the area under the ROC curve (AUROC) is provided. You task is to add code for calculating the points on the curve.

You can run your program from the command line as follows:

```
plot.py --MI=<MI> --gold=<gold_network> --name=<name>
```

where

- `<MI>` is the path of the output file from `CalcMI.py`, formatted and sorted as in ***1A***
- `<gold_network>` is the gold standard network file for calculating TPR and FPR.
- `<name>` is the name for your plot image. `plot.py` will automatically append `.png` so only provide a name without the file type extension.

***1E**: Evaluate your predictions*

After you compute the mutual information, plot the ROC curves and calculate the AUROC scores for the three following scenarios:

```
bin_num = 7            str = uniform
bin_num = 9            str = density
                       str = kernel
```

Include the mutual information output files and plots in your handin directory and the AUROC scores in your solution PDF. Which combination gave the best AUROC?[7]

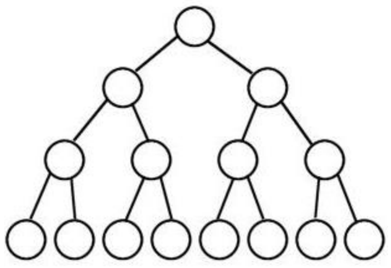
[6] The ROC curve is defined in the assigned reading Lever et al. (2016). More detailed instructions for computing it are on slide 24 of the CS 760 slides http://pages.cs.wisc.edu/~dpage/cs760/evaluating.pdf

[7] Note that in a real research problem it would be improper to select the hyperparameters (the number of bins and binning strategy) based on the AUROC on the gold standard network. Also note that AUROC is not actually an appropriate metric for this task because the gold standard has few positive edges.

Input files, example output files, and template Python files can be downloaded from **`https://www.biostat.wisc.edu/bmi776/hw/hw2_files.zip`**

Part 2: Gibbs sampling

Consider a fictitious cellular signaling cascade in the form of a binary tree with 15 nodes.



Each node $\boldsymbol{i}$ represents a gene $\boldsymbol{X_i}$ with two possible values, 1 for on and -1 for off. Let $\boldsymbol{R}$ be the root node (that is, gene $\boldsymbol{X_1}$) and $\boldsymbol{S}$ be the right-most leaf node (that is, gene $\boldsymbol{X_{15}}$). The network is represented by the following probabilistic model:

$$P(X_1, X_2, \dots, X_{14}, X_{15}) = \frac{1}{Z} \prod_{(i,j)} \psi(X_i, X_j)$$

where (i, j)'s are pairs of *directly* connected nodes, $\psi(X_i, X_j)$=$e^{X_i X_j}$ is called a potential function, and Z is the normalization factor given by

$$Z = \sum_{X_1, X_2, \dots, X_{14}, X_{15}} \prod_{(i,j)} \psi(X_i, X_j)$$

where the sum is over all possible configurations. This is an instance of a Markov random field (a.k.a. undirected graphical model).

***2A**: Probabilities in the Markov random field*

Answer the following questions in your solution PDF:

- How many possible configurations are there in this network?
- What are the most probable and the least probable configurations?
- Is it true that $P(R = 1|S = 1)$= $P(S = 1|R = 1)$? Show your work.

***2B**: Gibbs sampling in the Markov random field*

Next, write a Gibbs sampler, `gibbs.py`, to estimate $P(R=1|S=1)$. This Python script does not take any input file and outputs the estimated probability $P(R=1|S=1)$ to the screen.

Start sampling from the configuration where all $\boldsymbol{X_i}$ except S are set to -1, and S is set to 1. Iteratively update nodes with *one* of the following strategies: (a) traverse the tree by stepping through the levels in top-down order, and scan through the nodes within each level from left to right, or (b) randomly pick a node at each iteration.

In sampling-based inference, we typically discard the initial samples during a burn-in period. Discard the first 10^4 samples your Gibbs sampler generates. Estimate $P(R=1|S=1)$ using the next 10^5 samples. Run your sampler three times and report your three estimated $P(R=1|S=1)$ values in your solution PDF.

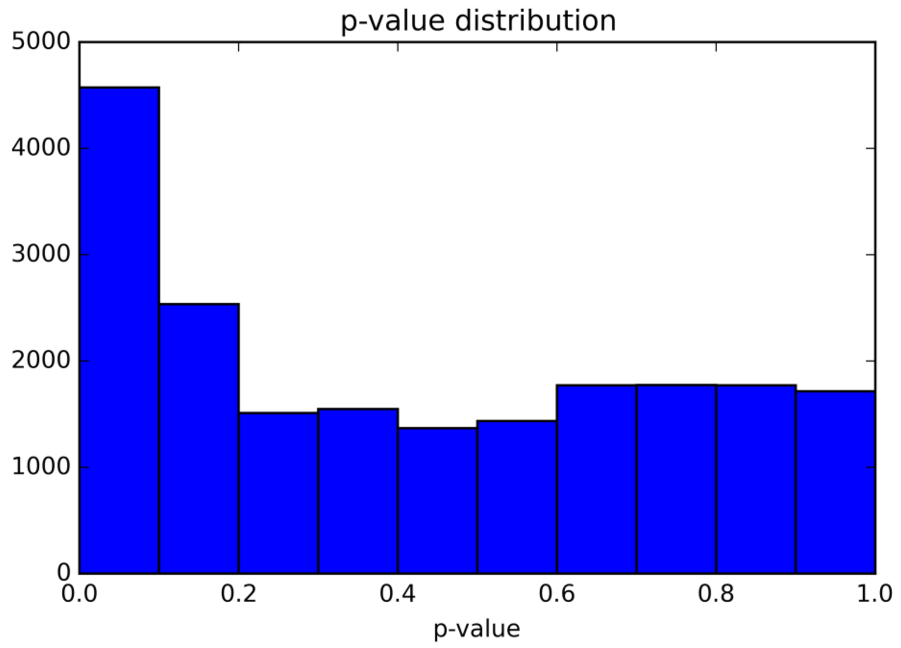
If you want to check whether your Gibbs sampler is correct, you can also write code that computes the *exact* probability $P(R=1|S=1)$ by enumerating all possible configurations. No code needs to be submitted for this optional sanity check.

Part 3: Calculating *q*-values

You will manually calculate *q*-values from a *p*-value distribution.

***3A**: Estimating λ*

First, use the histogram of the *p*-value distribution below to estimate λ visually as in Storey and Tibshirani 2003. The distribution contains *p*-values for 20000 features. Estimate λ to the nearest 0.1 and report the value you estimated.



***3B**: Estimating* $\hat{\pi}_0(\lambda)$

Use the table below to estimate $\hat{\pi}_0(\lambda)$ for the value of λ that you selected. Report the $\hat{\pi}_0$ you estimated rounded to two decimal places.

λ	$\#\{p_i > \lambda; i = 1 \ldots m\}$
0.0	**20000**
0.1	**15427**
0.2	**12893**
0.3	**11382**
0.4	**9834**
0.5	**8466**
0.6	**7030**
0.7	**5259**
0.8	**3484**
0.9	**1714**

***3C**: Calculating* q-*values*

Although 20000 features have been tested, only the top 10 features ranked by *p*-value are listed below:

Rank	p-value
1	**0.000003**
2	**0.000007**
3	**0.000013**
4	**0.000024**
5	**0.000028**
6	**0.000033**
7	**0.000046**
8	**0.000055**
9	**0.000096**
10	**0.000099**

Calculate the *q*-value for these 10 features rounded to three decimal places. You may assume that the *q*-values for the remaining features not shown in the list above do not affect the *q*-values of these 10 features.