

Assignment Goals

- i. Use mutual information to reconstruct gene expression networks.
- ii. Gain a deep understanding of convolutional neural networks (CNN) for regulatory genomics.
- iii. Develop working knowledge of genome-wide association studies (GWAS).
- iv. Understand and experiment with multiple testing correction procedures.

Submission Instructions

- To turn in your assignment, log in to the server **mi1.biostat.wisc.edu** or **mi2.biostat.wisc.edu** using your BMI (biostat) username and password.
- Copy all relevant files to the directory

/u/medinfo/handin/bmi776/hw2/<USERNAME>

where **<USERNAME>** is your BMI (biostat) username. Submit all of your Python source code and test that it runs on the biostat server.

- 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 (50 points)

In class, we saw how FIRE uses mutual information to detect relationships between sequence motifs and gene expression levels. Information-theoretic algorithms are also popular for reconstructing transcriptional regulatory networks from gene expression profiles. Given the expression levels for a set of genes measured in a sufficient number of biological conditions, mutual information (MI) can detect certain types of pairwise dependencies that may suggest one gene is a regulator (e.g., a transcription factor) and another is its target. For this assignment, we will create simple undirected gene-gene networks by ranking and thresholding the MI of gene pairs.

Write a program, **CalcMI.py**, that takes as input the expression data for a set of genes across a number of biological conditions and outputs the list of gene-gene dependencies and their MI. It should only consider the dependencies between unique genes, not the MI of a gene and itself (i.e., the entropy of that gene's expression). Compute MI by discretizing the gene expression levels, mapping continuous values into discrete bins. For a pair of genes $G1$ 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 count matrix. From this count matrix, estimate $P(G1 = a)$, $P(G2 = b)$ and $P(G1 = a, G2 = b)$ needed for calculating the MI between $G1$ and $G2$.

Implement the following two binning strategies:

- *Equal size binning.* This results in equal-sized bins. For example, if a gene's expression values lie in the range [1, 11] and we assign them into four bins, the bins would be [1, 3.5), [3.5, 6), [6, 8.5), and [8.5, 11].
- *Equal density binning.* This uses a percentile-based assignment to discretize expression values. With two bins, 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.

Your program should be callable from the command line as follows:

```
python CalcMI.py \  
    --bin_num=<bins> \  
    --bin_str=<strategy> \  
    --out=<out> \  
    <dataset>
```

where

- **<dataset>** is a text file containing gene expression values. 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 contain the temporal expression profile of genes, each labeled with a numeric index. The file is in a tab-delimited format.
- **<bins>** is the number of bins into which you should assign the continuous gene expression values when calculating the MI.
- **<strategy>** is the binning strategy ("*size*" for equal size binning and "*density*" for equal density binning).
- **<out>** is the name of the text file into which the program will print *all* unique gene pairs and their MI values line by line. Round MI to three decimal places, and print the lines in descending order of the rounded MI values. Break ties based on the index of the first gene and then the index of the second gene if needed, sorting gene indexes in ascending order.

Example input files **example1.txt** and **example2.txt**, their corresponding output files, and the template **CalcMI.py** with argument parsing code can be found in the **hw2_files** directory. Your program will be evaluated on the example inputs and additional datasets that will be kept private.

Part 2: Deep ReguLatory GenOmic Neural Networks (DragoNN) (15 points)

We will use the DragoNN Python package to explore convolutional neural networks (CNN) for regulatory genomics. DragoNN can create DeepSEA-like networks but is more user-friendly, which makes it easier to simulate training sequence data for user-

specified *cis*-regulatory modules, experiment with different network architectures, and visualize the filters learned by a CNN.

To answer the questions below, you are *required* to perform all experiments on the biostat servers. To access the DragonNN package, first confirm that you are using the BMI776 Python environment (see HW0 for how to set up the environment). Next, type

```
source activate dragonn
dragonn -h
```

in the command line to activate the conda environment where DragonNN is installed and test that DragonNN is available. Finally, copy all **.fa** files and **interpret.py** to your handin directory. Run everything in your handin directory and leave the output files there.

- (A) (*CNN Training*) You will first train a CNN on data from a simulated ChIP-Seq experiment. You are provided with a FASTA-formatted file of 5,000 DNA sequences bounded by some regulatory proteins, **positive_train.fa**, and a negative set of 5,000 unbounded sequences, **negative_train.fa**. Use the following command to train a one-layer CNN with five hidden channels (or filters) and a convolutional kernel of width 15:

```
dragonn train \
    --pos-sequences positive_train.fa \
    --neg-sequences negative_train.fa \
    --prefix one_layer \
    --num-filters 5 \
    --conv-width 15
```

This trains the one-layer CNN and saves the model architecture and learned weights to **one_layer.arch.json** and **one_layer.weights.h5**. DragonNN splits the input data into training and validation sets, and reports several performance metrics after each epoch of training.

- **What are the training and validation auPRC (area under the precision-recall curve) after the last epoch? (1 point)**

The one-layer CNN is an extremely simple network. We can train a more complex network by adding more layers and filters. Use the following command to train a two-layer CNN with 15 filters per layer and a convolutional kernel of width 15:

```
dragonn train \
    --pos-sequences positive_train.fa \
```

```
--neg-sequences negative_train.fa \  
--prefix two_layer \  
--num-filters 15 15 \  
--conv-width 15 15
```

This trains the two-layer CNN and saves the network architecture and learned weights to **two_layer.arch.json** and **two_layer.weights.h5**.

- What are the training and validation auPRC after the last epoch? Why is the two-layer CNN's performance better than the one-layer CNN? (3 points)

(B) (*CNN Inference and visualization*) You will inspect and visualize the two-layer CNN you trained in (A) using the following command:

```
python interpret.py \  
--pos-sequences positive_test.fa \  
--neg-sequences negative_test.fa \  
--arch-file two_layer.arch.json \  
--weights-file two_layer.weights.h5 \  

```

This will load the trained two-layer CNN from **two_layer.arch.json** and **two_layer.weights.h5**, load positive and negative test sequences from **positive_test.fa** and **negative_test.fa**, predict the probabilities that the test sequences are bounded, and visualize the filters learned by the network.

Examine the output file **two_layer_architecture.png**, which shows a graphical view of the CNN layers and their sizes.

- What do the input and output dimensions of the first Convolution2D layer correspond to (ignore the Ones and 1's)? (2 points)
- What do the input and output dimensions of the Dense (i.e., fully connected) layer correspond to? (2 points)

Suppose we predict that all sequences with probability ≥ 0.5 are bounded (i.e., *positive*) and all others are unbounded (i.e., *negative*).

- How many true positives, false positives, true negatives and false negatives are predicted? (2 points)

The output files **motif1.png** and **motif2.png** visualize the true motifs used for generating the positive training and test data. The output file **two_layer_convolutional_filters.png** visualizes the filters learned in the first layer, i.e., the weights for the hidden channels in that layer.

- **Discuss whether or not any of the learned filters resemble the true motifs, and what concepts the filters in the first layer may have learned in general. (3 points)**

DeepLIFT provides an improved way to interpret CNNs by computing a score for each input feature. Examine the DeepLIFT plots for each positive test sequence in the subdirectory `two_layer_deeplift_positive`. The top panel shows the summarized score at each position in the input sequence. The gray region is zoomed and shown in the bottom panel with nucleotide-specific scores.

- **Do the DeepLIFT scores look more or less similar to the true motifs than the convolutional filter visualizations? Do they represent both true motifs equally well? (2 points)**

Part 3: GWAS (15 points)

Suppose that we perform a GWAS for a disease of interest (e.g., Type I diabetes) and obtain the summary tables below for one particular SNP in the genome. The subjects in the study come from one of two distinct populations, A and B.

<i>Population A</i>			
	CC	CG	GG
<i>Disease</i>	28	42	56
<i>Control</i>	95	31	17

<i>Population B</i>			
	CC	CG	GG
<i>Disease</i>	448	482	466
<i>Control</i>	435	511	497

- (A) Use a Pearson's χ^2 -test and an Armitage test for linear trend to determine if there is an association between the genotype at this SNP and disease status within population A. For each test, state the null and alternative hypotheses, calculate the test statistic by hand, report the p -value, and state your conclusion.
- (B) Repeat the same tests for population B using the code in `gwas_tests.py`.
- (C) Suppose that we did not record which population each subject came from. Repeat the same tests for the entire set of subjects using the provided code.

Discuss your results in (C) in light of your results from (A) and (B).

Part 4: Multiple testing correction (20 points)

In a study that produces thousands of p -values, such as GWAS and RNA-Seq differential expression analysis, we need to apply a multiple testing correction procedure to adjust the p -values based on the number of tests performed. Here, you will investigate the statistical power of several such procedures for the task of identifying differentially expressed genes from a microarray dataset.

The dataset, **log2counts.csv**, consists of the \log_2 -counts of 3,170 genes measured from seven *BRCA1*- and eight *BRCA2*-mutation-positive tumor samples. The genes with outlier counts have been removed. The nominal p -values are calculated using a permutation test as detailed in Remark C in the Appendix of Storey & Tibshirani (2003).

The base code in **MTC.py** guides you through data loading, permutation testing and plotting of p -values. Your task is to implement Bonferroni, Benjamini-Hochberg, and Storey-Tibshirani multiple testing correction procedures. All procedures take as input a list of *sorted* p -values and a significance threshold α , and return the indices of genes that are considered differentially expressed (i.e., significant). The Storey-Tibshirani procedure takes an additional parameter, λ , estimated visually from a *density* histogram of the p -values.

Your program should be callable from the command line as follows:

```
python MTC.py \
    --procedure=<procedure> \
    --alpha=<alpha> \
    --lamb=<lambda> \
    --fig=<fig> \
    <counts>
```

where

- **<counts>** is a text file containing the counts for the samples, one gene per row.
- **<procedure>** is the correction procedure ("*bf*" for Bonferroni, "*bh*" for Benjamini-Hochberg and "*st*" for Storey-Tibshirani) for processing the p -values.
- **<alpha>** is the significance threshold, default to 0.05.
- **<lambda>** is the estimated λ used by the Storey-Tibshirani procedure. It is not used by the other two procedures.
- **<fig>** is the path where the generated p -value plot is saved.

The program outputs a plot of sorted p -values in which the significant genes are colored in red. Run the program with all three procedures. Discuss the plots.