An Analysis Pathway for Methylation Data

Pavel Komarov

August 12, 2016

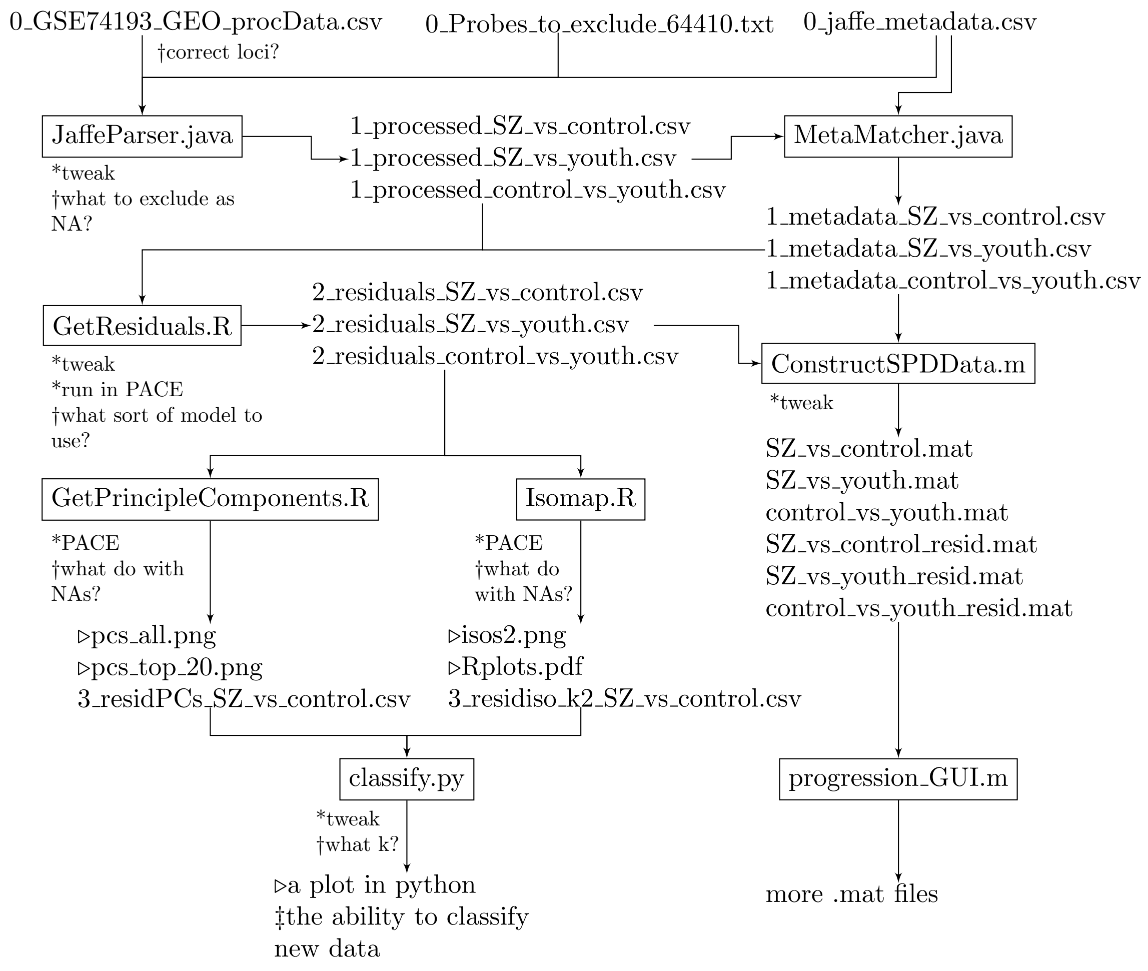
1 Introduction

In the end, we hope to gain an understanding of how genomic methylation may deviate from normal in Schizophrenia patients. We take a heavily data-driven and algorithmic approach, attempting to elucidate the data in a way its generators [1] did not. We hope the methods and expertise we develop in this process are reusable on different, better data in the future and might enable us to answer important questions like "Can we diagnose a disease with epigenetic data?" or "Can we identify a complex combination of factors that might be causing or correlated with a disease?", questions we might otherwise find intractible.

2 Data Flow

Immediately after the questions "What are we doing?" and "Why are we doing it?" comes "How?". In the case of this project, how is with a datapath. In order to understand our investigation precisely and be able to replicate or expand the results, it is essential to understand the data transformations shown below.

2.1 A Diagram of the process


0_GSE74193_GEO_procData.csv
†correct loci?
0_Probes_to_exclude_64410.txt
0_jaffe_metadata.csv
JaffeParser.java
*tweak
†what to exclude as NA?
1_processed_SZ_vs_control.csv
1_processed_SZ_vs_youth.csv
1_processed_control_vs_youth.csv
MetaMatcher.java
1_metadata_SZ_vs_control.csv
1_metadata_SZ_vs_youth.csv
1_metadata_control_vs_youth.csv
GetResiduals.R
*tweak
*run in PACE
†what sort of model to use?
2_residuals_SZ_vs_control.csv
2_residuals_SZ_vs_youth.csv
2_residuals_control_vs_youth.csv
ConstructSPDData.m
*tweak
SZ_vs_control.mat
SZ_vs_youth.mat
control_vs_youth.mat
SZ_vs_control_resid.mat
SZ_vs_youth_resid.mat
control_vs_youth_resid.mat
GetPrincipleComponents.R
*PACE
†what do with NAs?
Isomap.R
*PACE
†what do with NAs?
▷pcs_all.png
▷pcs_top_20.png
3_residPCs_SZ_vs_control.csv
▷isos2.png
▷Rplots.pdf
3_residiso_k2_SZ_vs_control.csv
classify.py
*tweak
†what k?
▷a plot in python
‡the ability to classify new data
progression_GUI.m
more .mat files

2.2 How to read the diagram

Boxed items are "units", pieces of code made for processing things (located in `/code`). Unboxed text represents files (located in `/data`). I've prefixed datasets with numbers to roughly indicate their stage in the process. Outputs prefixed by a ▷ are side-effects intended to give some insight in to what just happened. Smaller text under units or files are "notes". Those beginning with * are directions for use: "PACE" means "This unit should be run in the cluster if you want it to finish."; "tweak" means "You will have to modify the actual text of the unit (and sometimes then recompile) to make it generate different outputs." Notes beginning with † are essential questions that plague us. If they appear below a unit, I had to find an answer in order to create that unit. You should know that my solutions to these problems—in all cases—are not necessarily right, but I spent a great deal of time trying to understand the issues and ensuring my answers were at least good. I make a few arguments for my choices in this document as well as in code comments.

3 Stepwise instructions for how to do what the diagram shows + What things are and mean

3.1 Preliminaries

We begin with the 0s: `GSE74193_GEO_procData.csv` is directly from Jaffe, et al.; `jaffe_metadata.csv` is created by passing a minorly-modified `GSE74193_series_matrix.tsv` through `CSVTransposer.java`; and `Probes_to_exclude_64410.txt` is made by joining a couple of other exclusion lists together without duplicates.

I find it most convenient to interact with `.csv` files through a program like LibreOffice or Excel, since they can put the comma-separated data in cells and make it more readable. But do not try to open large datasets with these programs, or your computer will run out of memory and freeze. LibreOffice's 'text import' screen (that appears when opening `.csvs`) is nice because it gives a preview of the data without trying to read all of it, so the schema is visible. `more <filename>` on the command line can be used to print large files piece by piece, not as neat as an office-type program but often equally as effective.

3.2 Basic processing

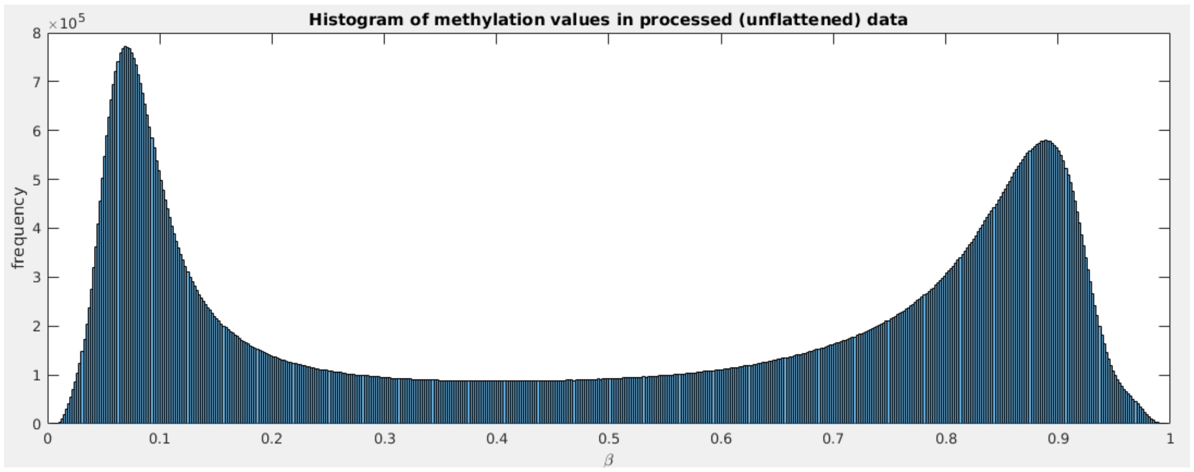
At stage 1 I take all 0s as inputs to `JaffeParser.java` and output what I call "processed" data. That is: Exclude all the probes listed in the exclude file; set all "beta" methylation values with a corresponding p value > 0.05 to be `NA`; exclude samples from plate 244 (known to suffer from exceptionally bad batch effects); only include the best samples from each subject; and only include the subjects who meet certain criteria (like exceed some age or have a certain disease state). Note that this last distinction is the difference between the three outputs of this unit.

To use the `JaffeParser`, first modify the subject-inclusion criteria on lines 42-47. Then recompile with `javac JaffeParser.java`, and run with `java JaffeParser <data> <metadata> <excludelist> <out>`, where everything labelled with <> is replaced by literal file paths. Next run the `MetaMatcher` with `java MetaMatcher <data> <metadata> <out>` to generate cleaned-up metadata files corresponding perfectly to the datasets generated by the parser.

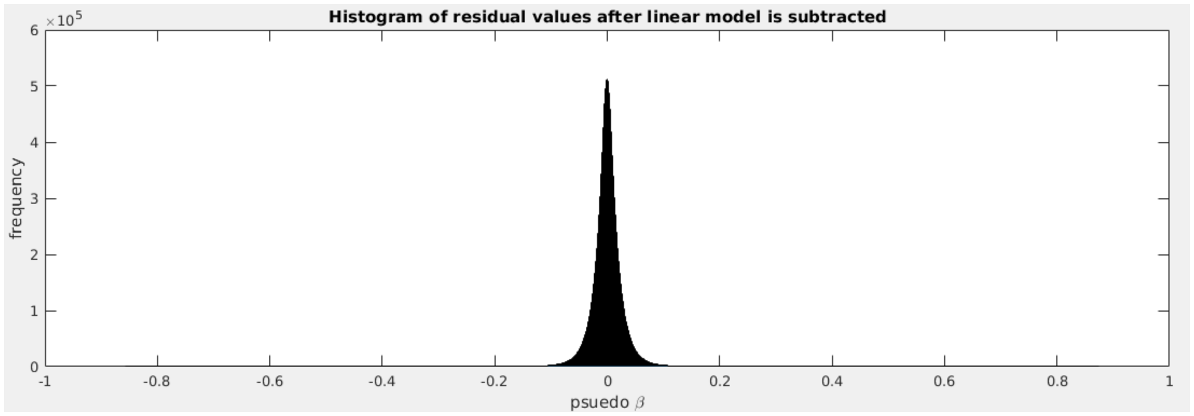
3.3 Getting rid of everything that isn't disease state

At stage 2 I generate what I term the "flattened" datasets by passing the processed data and metadata to `GetResiduals.R`. The computation is done in the cluster because it takes loads of memory and can crash your system (as it did mine several times). Open your root directory in PACE and make sure `GetResiduals.R` is in a subdirectory called `/code`. Open this R file and make sure that it is looking for `.csv` files that exist in the `/data` subdirectory. Then `ssh` in to PACE and submit the job with `msub GetResiduals.pbs`. If that didn't make perfect sense, see Section 4 where I've laid out some details.

Conceptually, this step fits a linear model to explain why the data varies as it does according to the Sex, Age, and Ethnicity of individuals, the four PCs that best characterize batch effects (provided by Jaffe, et al. in the metadata), and the cellular composition of samples (embrionic stem, dopaminergic neuron, et cetera). Once a fit is computed, the script subtracts this fit from the real data and returns the difference, termed the "residuals". This leftover variation is unexplained by the model, or in other words may be due to causes not accounted for in the model. But a few problems arise here: What if a linear model is not the right choice so the leftovers we see are the artefacts of an intrinsically poor fit instead of an unseen cause (like disease state)? What if the leftover variation is small? Might it just be noise? How can we be sure?



(a) processed



(b) flattened

Figure 1: Histograms of SZ_vs_control datasets (which include samples from Schizophrenic brains and controls over 16.)

I hope Figure 1 demonstrates in stark terms what this step has done and helps you recognize how to answer some of the questions posed above: After flattening, the leftover data is sharply centered around zero. It's basically a Dirac delta. Whereas before the values intuitively correspond to degrees of methylation at loci, the new values can be positive or negative depending upon whether the model was larger or smaller than the input data at a locus. This outcome means a few things: (1) Sex, Age, Ethnicity, et cetera explain most (if not all, considering noise) of the variation in the original dataset. (2) Because a linear model can match the processed data so closely as to return such small residuals, its characterization power of this dataset is at least very good. (3) There isn't much left to be explained by disease state, or there isn't much left to explain disease state (depending upon which way you think the arrow of causality goes).

Whether the little bit of variation left is noise is still unclear. Ideally we would like the subtraction performed at this step to yield a fairly sharp, bimodal distribution, so we might clearly see a single divide the model could not explain yet be somewhat confident its characterization power was otherwise quite good. Since this is not the case, and since there is no simple mathematical or scientific proof that what remains is entirely noise, we need to rephrase the question. Let's pose it in engineering terms: Given the flattened data, can we build an agent that can distinguish between diseased and control at a better-than-random rate? If we can, then we have found some signal in the noise and can know that it is not all garbage.

3.4 Machine Learning: Definitions and how to approach the problem

We could try to be the agent ourselves or build some rule-based algorithm that can distinguish samples based on disease state, but the data is so large and conceptually inaccesssible that we need a different approach. Thankfully, we live in the age of Machine Learning.

There are many classes of Machine Learning algorithm, and it is important to know what they are and what they do because: (1) Not every algorithm is suitable to every problem, and (2) the choice of algorithm has a large impact on how expensive training or querying will be.

In the interest of making this document as practical as possible for you biologists, I include some explanations now.

1. *Regression vs Classification* If we wish to assign some number from a continuous domain to each sample, then we should use regression; if we wish to assign one of a discrete set of answers to each sample, then we should use classification.

2. *Supervised vs Unsupervised* Supervised means we feed the algorithm (x, y) pairs, where x is the information we might see at query-time and y is the answer we would have to find. The algorithm takes this input-output information and adjusts itself to increase the chance it outputs y in the future given this x. It takes hundreds of thousands (if not millions) of pairs to create a generalizable model (one that is performant on data not included in the training set). Unsupervised means we don't have the answers to start with; all we know is that there may be some pattern in the data and and want to uncover it, with no regard (at least inside the algorithm) for what that pattern may mean.

3. *Parametric vs Instance* Parametric learners find a model that can be represented by a "small" set of parameters (i.e. much smaller than the size of the training data). In this paradigm, training takes a long time, there is no need to keep the training data around after training, and using the model is fast. Instance learners memorize and use the training data directly, which means training is fast, but the model is relatively large and querying it is slow.

4. *Batch vs Online* Batch learners train on a batch of data and must be completely retrained to accomodate any new information. Online learners do all their decision-making with query-time calculations, so incorporating new data doesn't involve a complete retrain.

5. *Generative vs Discriminative* A Generative learner seeks to find the "generator" function, the joint probability distribution of all input-output pairs: $P(x, y)$. If we know the generator function has a certain form, then this can be done easily with little data and then even be used to give us additional, simulated data. Discriminative learners are useful when we don't know what form the underlying model should have. They seek to find conditional probability distributions: $P(y|x)$. For example, a generative model might attempt to solve the problem "Is this sentence in English or French?" by trying to learn a theory of language that can model French and English with parameters, whereas a discriminative model would solve the problem by learning English and French and trying to fit any given sentence.

To make these disctinctions concrete, let's review a few common algorithms with notes on their operation and how it determines their categorizations.

- *K-Nearest Neighbors (kNN)* Training simply involves memorizing the data, and querying involves finding the k most similar examples from training. This makes kNN an *instance* learner. Including additional data does not require a retrain, so kNN is *online.* kNN can either return some average or weighted combination of these k training examples, in which case it would be doing *regression*, or it can implement a voting-scheme and return only one *classification.* kNN requires labels (answers) for each training point, so it is *supervised.* And since kNN models the conditional probability of belonging to a class (discrete case) or of having a value on some domain (continuous case), it is *discriminative.*

- *Clustering* Combining things in to new classes based on their apparent similarity requires no knowledge of labels, so any form of clustering is *unsupervised.* (Actually it's difficult to come up with an example of unsupervised learning that is *not* some kind of clustering.) Clusters might end up different if we re-run the algorithm after including new points, in which case it would be *batched*, or the algorithm might simply lump new points with the nearest cluster, in which case it might operate *online.* The other distinctions listed do not apply.

- *Neural Networks* Neural Nets are *supervised*, requiring many training examples and thousands of rounds of backpropagation to tune. Backpropagation adjusts input weights of neurons, a set of numbers of a fixed size, the network's *parameters.* A neural network can be trained *online* if each new (x, y) is used, but they are often simply trained on a *batch* of data and then deployed without further adjustment. A network can do *regression* if there is a single output neuron connected to the last hidden layer, but in practise a 1-of-k *classification* scheme often works better. For example, if you wanted to find a number between 0 and 9, you might have 10 output neurons and pass the number 5 as 0000010000 at the beginning of backpropagation, so that one neuron is trained to activate and the others to be silenced. As opposed to having to worry about how close the output is to some value, you can then interpret the intensity of the output on each of the 10 neurons as the network's confidence for each digit. Finally, Neural Nets are *discriminative* because they find boundaries between classes instead of explicit models of joint probability.

So we have options (many more than listed), but which is the best option for gene microarray data? The short answer is it's difficult to know right away. And how do we even define "best"? Does it mean fastest, most accurate, most ellucidating of some underlying pattern, most conceptually tractable?

I've taken two main paths, one the easiest-to-understand classification scheme I could think of that does not require tens of thousands of training points (to answer the question "Are these residuals just noise?") and one based on pattern-finding, which is a bit more complicated. I will start with the first, and I begin that discussion with a preliminary issue: feature extraction.

3.5 Turning too many features in to something manageable

Let's make the concept of an (x, y) pair more specific: An algorithm gets a load of information stacked in to a vector called x and must find an output y (often a single value). Elements of x might be continuous (like a temperature measurement) or discrete (like a genre), but all can be mapped to numbers. If we call the length of x D, then x constitutes a point in D-dimensional space.

In this hyperspace, the task of distinguishing known classes (supervised learning) becomes the task of drawing boundaries (hyper-surfaces) to divide groups of points with the same y from each other, and a regression task becomes the task of finding some continuous function to describe y throughout the space.

Here's the rub: All methods to accomplish such jobs suffer from "the curse of dimensionality". That is, as the dimensionality of the input data D grows, the computation and number of samples N required to find a generalizable answer explodes.

In our case we have a feature-space in 421103 dimensions (one for each CpG site after exclusions), far too large to work with, especially considering that (due to the high cost of biological experiments) our $N \sim 100$. Note that this dimension is actually quite small in a biological sense since every individual has far, far more CpG sites. In the future we can expect both D and N to increase for this problem.

The way to resolve this dilemma is by recognizing that not all these dimensions carry useful information, and some carry redundant information. So, intuitively, we should be able to recreate a somewhat faithful portrait of the data in a lower dimension, d, that captures its key elements. This is process of going from $N \times D$ to $N \times d$ is called "feature reduction", and to say it's a well-studied area is an understatement.

As a consequence, many reduction methods have been developed. I used three: PCA, Isomap, and Hierarchical Clustering.

3.5.1 Principle Component Analysis

PCA is maybe the most common feature reduction method. It works as follows:

```
1. Subtract the mean in every dimension and scale by the variance to get
   adjusted data where features are more equally scaled.
2. Calculate the covariance matrix $C$.
3. Find eigenvalues, $\lambda$, and eigenvectors, $\vec{v}$, of $C$.
4. Choose $n$ $\vec{v_i}$ with the largest corresponding $|\lambda_i|$. These are the PCs.
5. $New\_data = adjusted\_data \cdot [\vec{v_1}\vec{v_2}...\vec{v_n}]$
```

This isn't the only formulation of PCA—the method employed by R uses Singular Value Decomposition (SVD) to get the eigenthings—, but I think it gives a nice intuition. Essentially, you are looking for the vectors which best characterize the directions in which the data varies most, recording how much each datapoint varies in those directions, and throwing away all the other directions/dimensions.

To accomplish this feature reduction on our data, I used R's `prcomp()` function and wrote a nice script to package it all up. To run it, put `GetPrincipleComponents.R` and `GetPrincipleComponents.pbs` in your `/code` subdirectory in PACE, make sure it's reading the right input file from `/data`, and submit the job (details in Section 4). Note that SVD and other methods of finding eigenvectors and -values do not work if there are `NA`s in the data, so I set them to be 0 after centering and scaling the data. For a more complete discussion of this choice, see the comments in `GetPrincipleComponents.R`.

PCA returns up to N PCs, so PCA reduces the dimension from 21103 to 244 right away (in the case of SZ vs control), but most of the variance is usually captured in the top few PCs. Figure 2 displays the variance in the direction of each PC across the N samples.

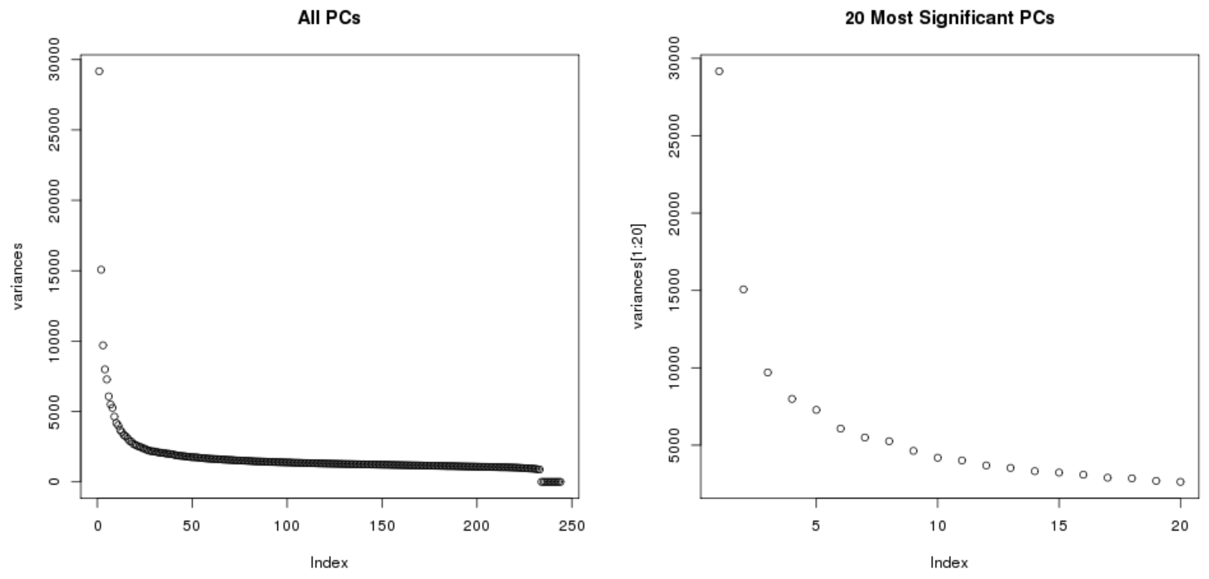


Figure 2: Variance of the data in the direction of PCs.

Notice that beyond some index, the curve flattens out. The PCs beyond this point contain comparatively little descriminative information, so if we wish to further reduce the dimension of the data (to make $N > d$), then we can ignore them.

To make this clearer, Figure 3 visualizes the samples as points in a space defined by the most-varying 3 PCs. Note that I've turned the figure to show the best separation I can, so it's not a perfectly honest measure, but some kind of difference between the groups appears to be visible. If you wish to generate this plot and play with it yourself, run `first3PCs.R`.

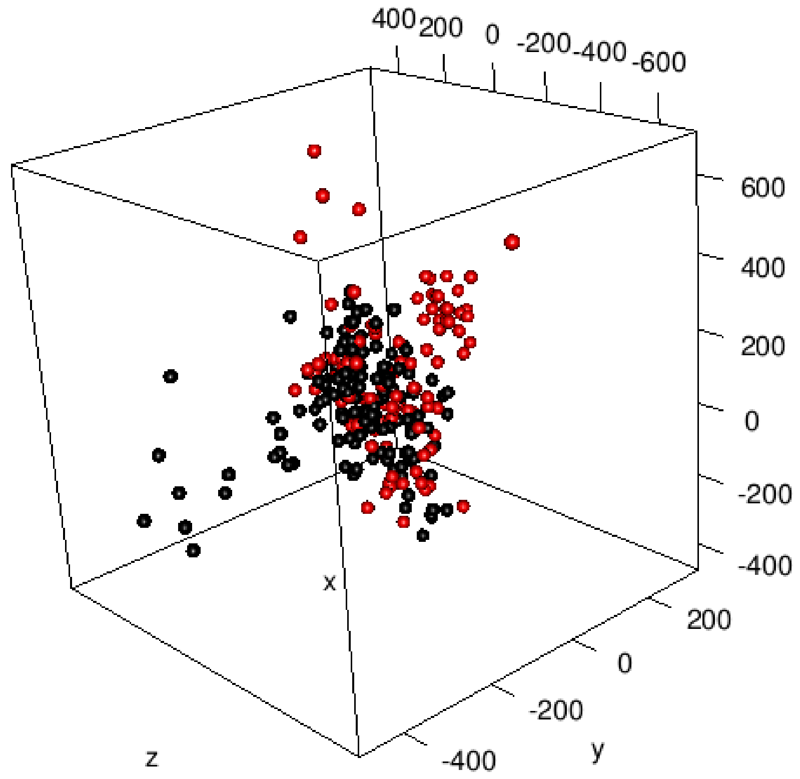


Figure 3: Visualization of Control samples (black) and SZ samples (red) in a space of the 3 most significant PCs.

But there is a possible problem: PCA is inherently linear, so it can be terrible at discovering good features in highly nonlinear contexts. A common example is the "swiss roll" dataset shown in Figure 1 here. What if microarray data is nonlinear?

3.5.2 Isomap

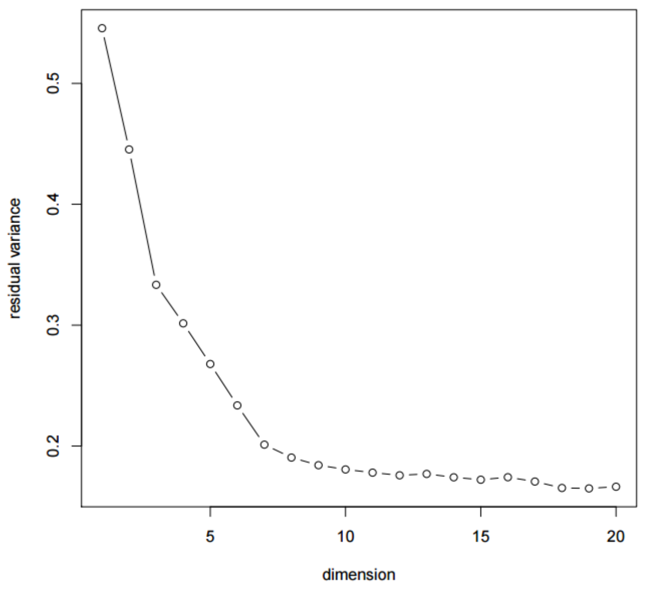
Isomap is an increasingly popular way to characterize nonlinear data. Here is an outline:

```
1. Construct a neighborhood graph as follows:
   for all $(x_i, x_j)$
       if distance$(x_i, x_j)$ $< \epsilon$
           add edge $(x_i, x_j)$ to G with weight 1
2. Compute shortest distances along the graph edges $\delta_G(x_i, x_j)$ by Floyd-Warshall
3. Apply multidimensional scaling to $\delta_G(x_i, x_j)$
```

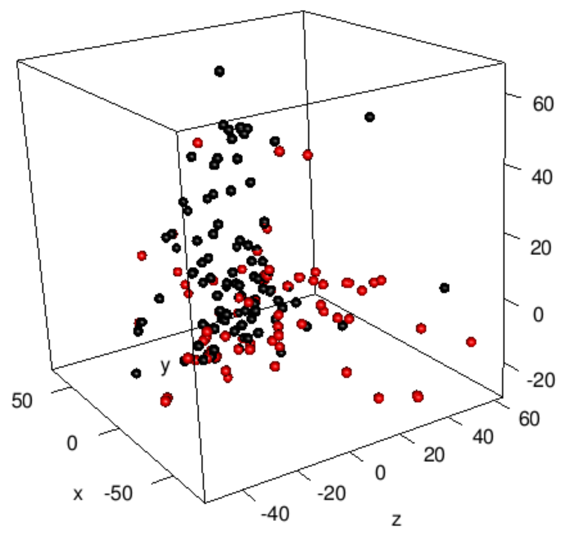
Multidimensional scaling takes a set of things that all have set distances from each other (like graph edges) and tries to assign all of them locations in d-dimensional space such that these distances will be best preserved. I highly recommend Wikipedia's article for more details.

But what you probably care more about is `Isomap.R`, an R script I have written to perform the above on our data utilizing the absurdly convenient `Isomap()` function from the `Bioconductor RDRToolbox` package. In order to get this to run, you will need to install that library in PACE (See instructions in Section 4.3), and then follow job submission steps analogous to those for `GetPrincipleComponents.R`.

Note that the version of Isomap in this package uses the k nearest neighbors when constructing the neighborhood graph instead of a distance cutoff ϵ. According to Wikipedia, k too large or too small can cause problems, so I ran the script a few times with different k and determined that $k = 2$ (the smallest value possible) yields the smallest residual variance in the fewest dimensions.



(a) Residual variance not captured by a multidimensional scaling in to x dimensions.



(b) Separation of the data when multidimensional scaling to 3 dimensions.

Figure 4: Isomap results.

The Isomap function appears to construct the adjacency graph once using the given k and does multidimensional scaling to however-many dimensions we specify. Figure 4a depicts the variance of residual information not captured by scaling to 1 through 20 dimensions. Notice that more dimensions is better up until some point where the residual variance does not decrease any more and can even jostle back upward (20 as opposed to 18, for example). For comparison with PCA, I include a point-cloud for the scaling to 3 dimensions. Notice there is a little separation, despite a still relatively high residual variance. To generate this plot yourself, use `first3isos.R`.

3.5.3 Hierarchical clustering

The final dimensionality reduction technique I have used is iterative hierarchical clustering. It works as follows:

1. Each item (CpG) begins in its own group with a variance across N people.
2. Drop groups with a variance that is below some threshold.
3. Combine all groups in to one and divide it in to k groups ($k = 2$) by k–means clustering if its 'coherence', defined as the average covariance between every locus in the group and the group's mean, is not above some threshold (c_1). Repeat L times or until no more groups can be divided.
4. Compare groups to each other and merge them if the correlation of their centers is greater than some cutoff (c_2).

In some ways this method is more intuitive and interpretable than the others: We take gene loci, make sure that they are varying across our samples, and then combine them in to "modules" that vary together. The results are clusters of loci, something biologically meaningful, and the d features for each of the N samples are degrees to which they express those modules.

3.6 Learning to distinguish samples based on disease state

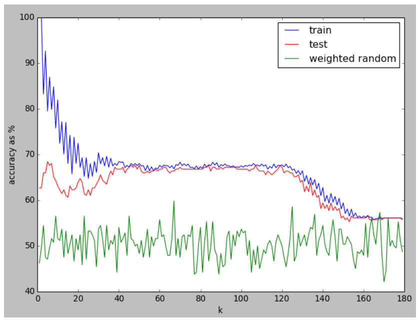
The last stage on the left of the flow chart is classification. Files beginning with 3s are features-files, literally $N \times d$ tables of numbers that we can use to characterize samples in d-dimensional space. They are now quite small and manageable, so we can use use an algorithm to classify them on any local machine.

In the interest of making a solution that is extremely intuitive, quick to code, and does not require millions of examples, I decided to use a kNN learner, which classifies samples based on which of their neighbors are closest in the d-space. `classify.py` makes use of my `KNNLearner.py` object as well as the `numpy` and `pandas` packages in `Python 2.7`, so install these. Open the code and look at the upper section where the `data` is read in; ensure you are reading the data you want and either normalizing it or not normalizing it depending upon your choice. Run the program with `python classify.py` in a command line pointing to the `/code` subdirectory.

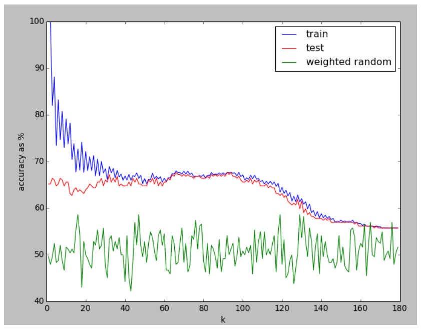
If you read the code, you will see there are two loops. The first is to find the best setting for the kNN "hyperparameter" k. Most machine learners have these kinds of parameters, values you the user have to set in the beginning without much guidance. For neural networks these include the number of layers and the number of hidden neurons per layer. For decision trees there is the max depth and the branching factor. The general rule is to try lots of different hyperparameters to find which ones work best on the training data and then stick with these.

I use the second loop as part of a "leave-one-out" testing scheme. The idea is that we don't have very many samples at all, so to divide the data in to training and testing sets might leave both sets too small. Instead, we can withold a single sample, train a model on the rest, test on that one sample, and repeat for all samples in the data. For an idea of accuracy, the number of models that correctly classified the one left out is divided by the N models created.

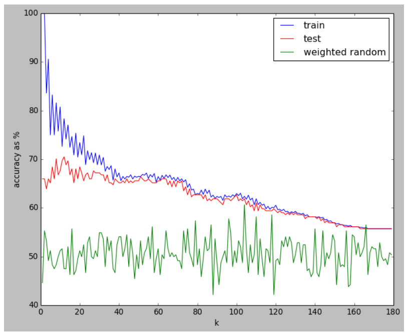
The plot displayed when execution finishes shows three things vs k: (blue) the average accuracy of the models on the data on which they were trained, (red) the average accuracy of the models on the data point left out, (green) the acccuracy of a random classifier that just picks 'Control' or 'Schizo' at a rate proportionate to their occurence in the data (should be about 50%).



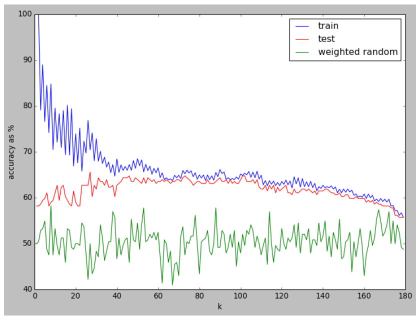
(a) 3 PCs



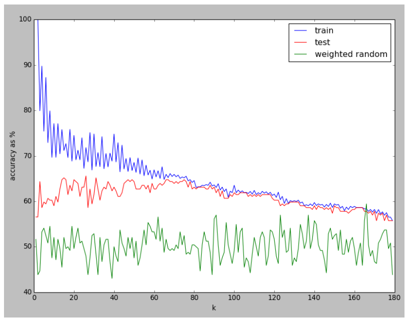
(b) 10 PCs



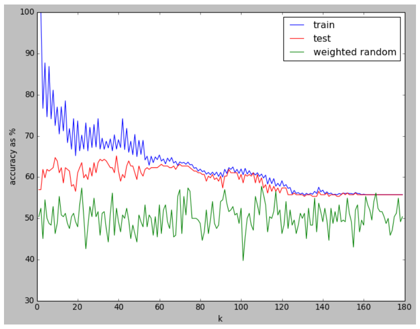
(c) 10 normed PCs



(d) 3-d isomap features



(e) 10-d isomap features



(f) 10-d normed isomap features

Figure 5: kNN classifier results.

Figure 5 displays some results. The upper row makes use of features found via PCA, and the lower row makes use of features found with Isomap. The left column is classifying with 3-dimensional data; the middle column uses 10-dimensional data; and the right column uses 10-dimensional data that has been divided in each dimension by the standard deviation across all points in that dimension (to make the point-cloud more equally stretched in all dimensions, a preprocessing step which is often recommended for kNN).

Notice a few things:

1. For small k, the training accuracy greatly exceeds the testing accuracy. This is called "overfit" and happens because whatever model we find to describe the training data does not generalize to the testing

data; it is too attuned to the data it has seen. At larger k there is a smoothing effect, so overfit goes away, and the red and blue lines converge.

2. In all cases the kNN classifier is doing better than random! This means there *appears* to be some signal left in the flattened data. Caution! We have very few data points, so this performance, no better than about 70%, may be an artefact of some oddities in the small dataset. (See Section 5.) What can say right away given the classifier's imperfect performance is that whatever signal there may be is either not present for all samples, is overwhelmed by noise much of the time, or can not be perfectly-described by the surfaces kNN is finding. (We may need something else 5.)

3. Norming the data doesn't have much of an effect, aside maybe from marginally depressing the accuracy of a 10-PCA classifier at high k and raising it at lower k. (I suspect this is an artefact of our dataset and will not generalize.)

4. PCA might be a better feature-extraction method than Isomap, as classifiers based on PCA features to do consistently slightly better. Maybe the microarray data actually correlates linearly with disease state.

3.7 Learning to identify progressions

So we suspect that there is *at least a little* signal left in the flattened data and that we can see some of it in lower dimension with PCA or Isomap. But what can we do with such a system? Long-term, the chief clinical application is diagnostic: Can we take tissue that is easily accessible (blood, for example), run it through a series of assays, and then use the resulting raw data to determine disease state for a variety of conditions? That sort of technology would require faster, cheaper tests on the wet-lab side and more honed learners on the analytical side, and it's years away.

Let's rephrase the question "Can we predict disease state?" to "Can we identify the underlying progression of a disease?". That might allow us to see what the most significant common threads are among diseased individuals and connect them to genes and the like, a much more interesting prospect for biologists. Now let's think like engineers: Can we build a system that can identify a meaningful progression from non-diseased to diseased?

Fortunately, Dr. Peng Qiu actually built the system in question (already intended for microarray data) and wrote a great paper about it [2]. The algorithm, called Sample Progression Discovery (SPD), is, as I understand it, as follows:

```
1. Perform Hierarchical Clustering as described above to obtain 'gene
   modules' that vary more than some threshold and are highly correlated
   to themselves across samples.
2. For each gene module:
   a. Place the samples in a space defined by the genes in that module
      at a location based upon their expression of those genes.
   b. Do density-dependent thinning of the samples in that space (only
      necessary when there are far more samples than we have).
   c. Create a Minimum Spanning Tree (MST) between all the remaining
      points in that space with weights proportional to Euclidean
      distance.
   d. Remember the distances between all samples along edges of the MST
      in a distances matrix (where samples that were thinned out get the
      same values as their nearest included neighbor).
3. Compare the distance matrices generated by different modules in step 2
   to obtain a similarity metric based upon their 'earth mover's distance'
   to one another.
4. Generate a progression-similarity matrix between gene modules using a
   threshold for whether a fit between a module and a tree is significant.
   '0.05 means that, among all the module-tree pairs, the top 5% with most
   significant [smallest] earth mover's distances are considered to 'fit
   well with each other''.
5. Choose a group of gene modules that support a similar progression, and
```

```
   construct the MST that describes that common progression.
6. Perform Multidimensional Scaling to place the samples in 2D according
   to their tree-distances, and draw tree edges to connect them.
```

Did you understand? Read it again if you have to; it's really brilliant. There is also an outline in the Methods section of the PLoS paper [2] that is simultaneously more and less detailed.

Unlike the methods explored in the last section, which were supervised, SPD is unsupervised, relying only on blind pattern-finding. In order to make sense of the patterns it identifies, we have to overlay information about the samples *post hoc*. I've written a Matlab script called `ConstructSPDData.m` to read in both flattened data and metadata and save them in a format SPD can understand. To modify which files it reads, change the input strings ($T =$ ..., $M =$ ...), making sure the data and metadata match.

Using SPD is not difficult but takes a few steps and can be a bit opaque unless you understand the algorithm. Follow the guide for maximal efficiency. Here are some hickups I ran in to:

1. This software was originally intended for smaller microarrays. As a consequence, 420k is too many loci. Solve this problem by throwing away loci with more than some number of NAs (If the number is >0, then the missing values are imputed.) and by throwing away loci with a standard deviation (across samples) below some threshold. You may have to be aggressive, otherwise clustering may not finish or may yield too many modules.

2. I am not actually certain how the iterative Hierarchical Clustering algorithm works. "Agglomerative" clustering is only the second half of what I and [2] describe.

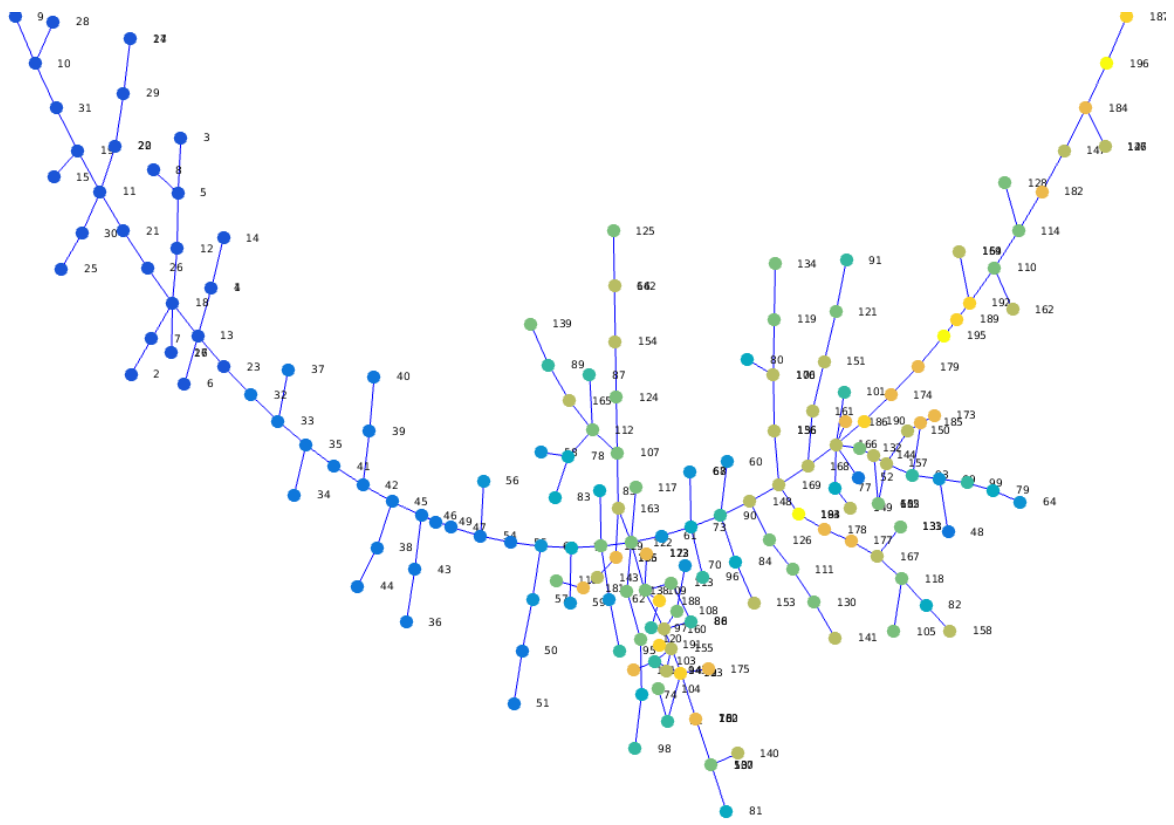
3. Ideally, the modules at the end are 50-200 loci, each with a very high coherence (see plots by clicking "module quality"). But in practise I have found it difficult to obtain modules like this. Either they are of wildly varying sizes (usually a few much much larcher than the others) or there are too many modules, all just above the minimum module size.

4. Beware too many modules! Computing fits between modules and trees (via earth mover's distance and distance matrices) is $O(n^2)$ with the number of modules. If n goes much over 100, it will take a very long time to finish.

5. Dr. Qiu's recommendation is that you vary the percentage of module-tree pairs used to construct the similarity matrix between 5% and 15%. The idea is that for a few different values you should see some of the same groups of gene modules making up the bright, highly-correlated blocks. Those groups are what you should enter manually.

6. If you don't want to do all the computation again, click "Save Results" and wait. (It takes a while.) If you want to read the data in the future, use the "Load result file" option instead of "Load raw data".

At the end of this process, clicking "View Progression" displays a tree that is constructed to fit the best progression commonly supported by the group of highlighted modules. Clicking the "Color code according to" check-box allows you to quickly see whether the progression matches well with any known features as recorded in metadata.


(a) Progression with age overlaid.

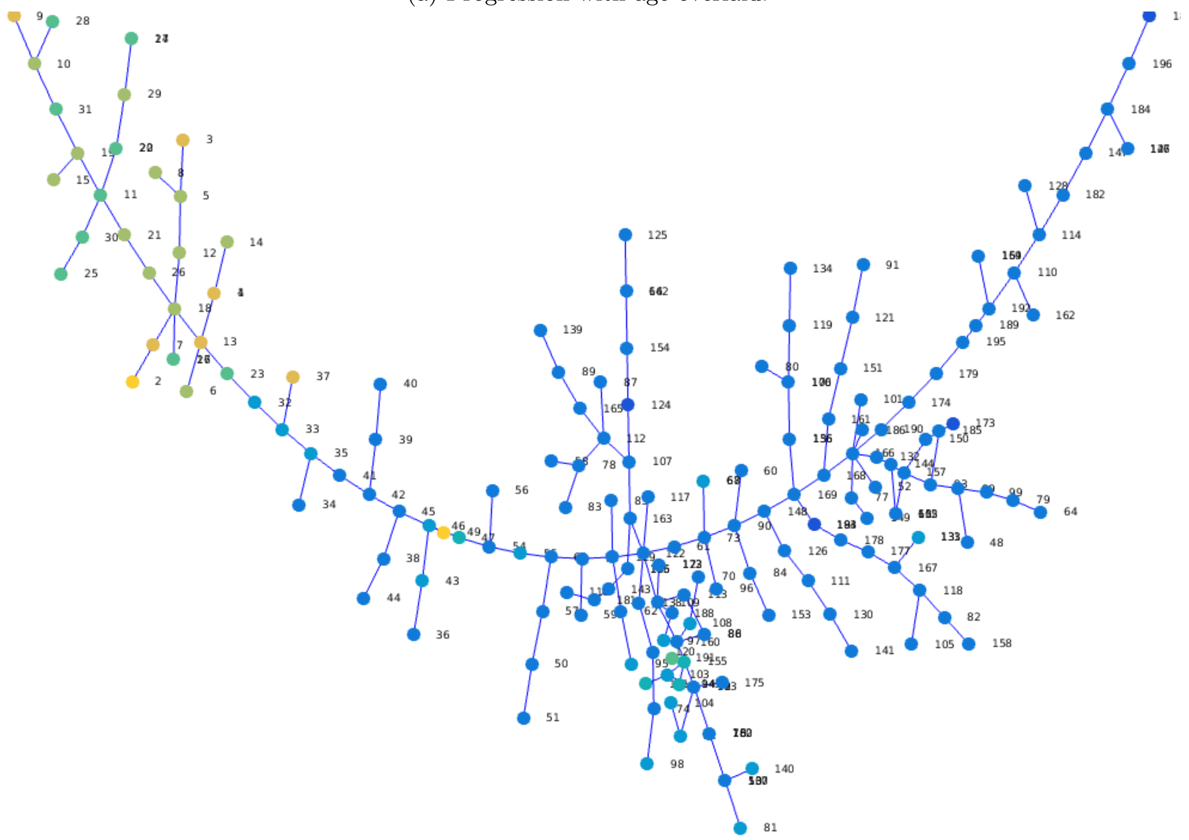

(b) Progression with embrionic stem cell concentration overlaid.

Figure 6: The most significant progression identified by SPD on all not-schizophrenic data (1 module, 1365 loci).

Figure 6 shows SPD's best results. This is on not-flattened data including every sample that is not from a schizophrenic person. Since overlaying age and age-related factors definitely agrees with the identified progression, we know that SPD has found the very obvious epigenetic trends associated with ageing.

So SPD is not broken. But does it work for identifying more subtle trends? The short answer is that I tried many, many different settings and never witnessed a progression that would diffinitively explain disease state. Figure 7 shows a typical example.

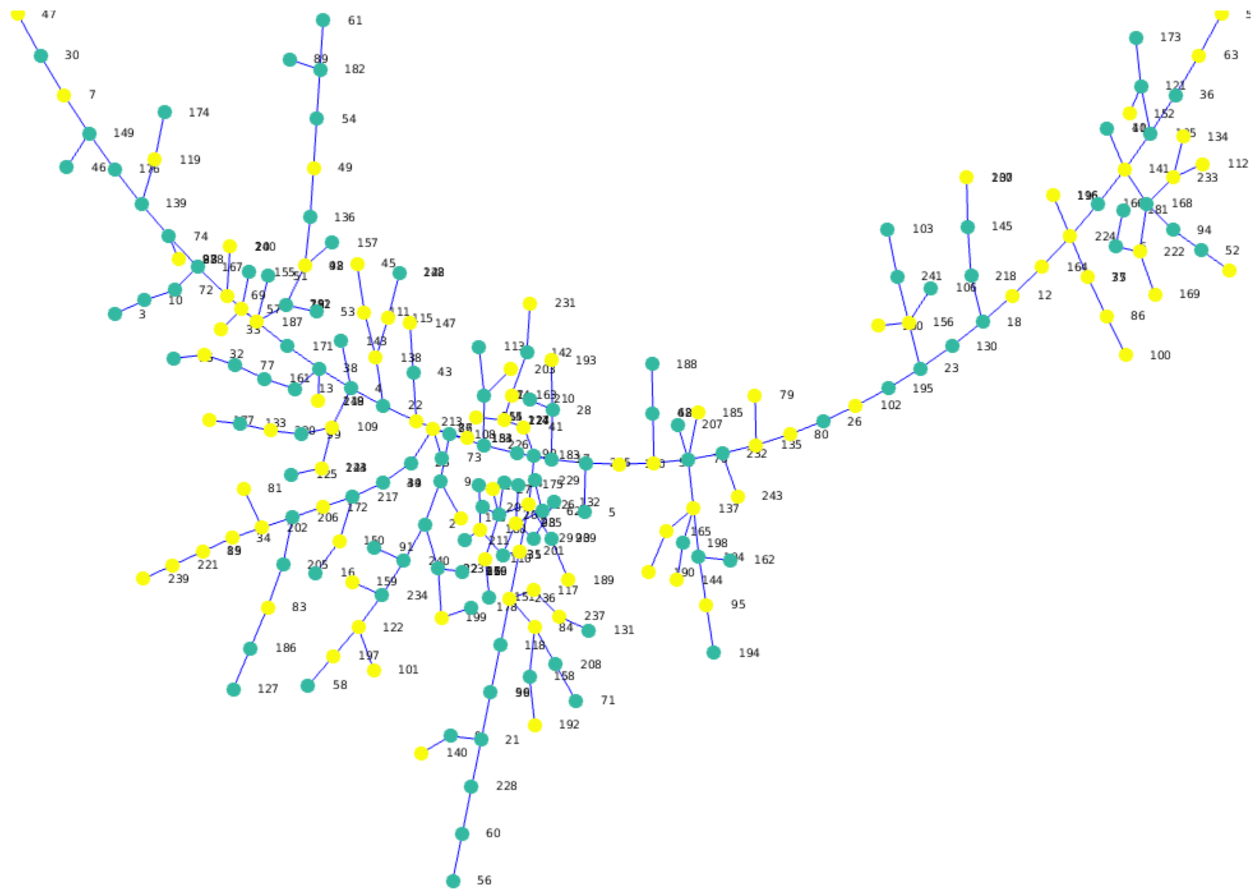


Figure 7: The most substantial progression identified (6 modules, 42 genes) from flattened >16yr control and schizophrenic samples with disease state overliad (yellow is control).

Although SPD can clearly identify progressions due to plate (batch effects) and age when I feed it the processed data (and *maybe* a couple subtle things due to sex and ethnicity, but remember we removed loci we thought would vary much with those factors), it largely fails to find such patterns in the flattened data. This is good news: The flattening process is doing what we want. The bad news is that there appears to be no pattern *strongly* correlated with disease state in any data set. However, there may be some *weak* relationships here, and it may be worth more exploration 5.

4 The Cluster

In in the interest of not distracting you the reader with too many things at once, I have so far neglected to cover the cluster at a granular level. But I would have liked a compressed introduction myself, and you will need it to carry out the steps in the last section, so I've generously given you this gift.

If you don't know what a computer cluster is or have never heard of map-reduce, I recommend falling down a Wikipedia rabbit-hole for a few days or reading the original paper from Google. Today's networked computing environments do amazing things, and understanding parallelization can be useful.

4.1 File I/O

First, open a terminal and type `ssh pkomarov3@biocluster-6.pace.gatech.edu`, replacing username and cluster name as necessary. I can access bioforce-6, biohimem-6, iw-shared-6, and testflight; you might use different ones. Give your password to log in, and then type `pwd` to determine the path to your root directory, something like `/nv/hp10/pkomarov3`. To quit the ssh connection at any time, use `exit`.

Next, open nemo (or any other file browser that supports sftp) and type something homologous to `sftp://pkomarov3@biocluster-6.pace.gatech.edu/nv/hp10/pkomarov3/` in to the url bar. It may require a password the first time and will take you to the 5GB user directory. You can do whatever you want here. I created a `/code` subdirectory to hold all scripts and job submission (`.pbs`) files. `/data` is 200GB of storage that will be saved (unlike the 7TB `/scratch` which is wiped every 60 days). Be aware that uploading to and downloading from the cloud is slow unless plugged in to LAWN.

4.2 Submitting jobs

Pbs files are to provide the cluster with job details, because R scripts and such do not contain instructions for this layer. Here is a simple example:

```
#PBS -N pavelGetResiduals
#PBS -q iw-shared-6
#PBS -l nodes=1:ppn=1
#PBS -l walltime=12:00:00
#PBS -l mem=20gb

cd $PBS_O_WORKDIR
module load R/3.1.0
Rscript GetResiduals.R
```

Anything beginning with `#` is an instruction to the order-processsing node like "When the job terminates, give me everything the program printed in files named 'pavelGetResiduals'.", "Submit to the iw-shared-6 queue.", "Use 1 node with a minimum of 1 processor each.", "Make sure this thing doesn't run more than 12 hours.", and "Make sure the nodes have 20GBs of memory.". `cd $PACE_O_WORKDIR` ensures the nodes that execute the job are looking in the directory whence a job was submitted (the current directory of the terminal at submit-time). Hereafter follow any instructions a node needs to execute to complete the job.

To submit a job, use the command `msub script.pbs` in a terminal sshed to the cluster. If successful, it should print the job number. To cancel the job, use `qdel <job number>` (for example `qdel 100200300`). To view the status of a job, use `qstat | grep <username>`. Q means the job is currently queued and will be run when there is a free node. R means the job is running. C means the job has completed.

4.3 Additional libraries

If you require libraries that are not native to whichever code modules you are loading, then you must install them in the cluster. This is done from the command line through an `ssh` connection. Since PACE is connected to the internet, you can download anything you need to your 5GB user directory. For example, `Isomap.R` requires the `RDRToolbox`. To get this library and supporting libraries: (1) Load an R module of your choice (listed in `/etc/modulefiles/R/` on the PACE node) with a call like `module load R/3.2.4`. (2) Enter the R environment with `R`. (3) Get the Bioconductor installer with `source("https://bioconductor.org/biocLite.R")`, call `biocLite("RDRToolbox")`, and answer `y` to its questions. (4) Make sure your R file has the appropriate `library(RDRToolbox)` reference. (5) Make sure the corresponding `.pbs` file is set to load the same version of R that you just used to install the packages.

Note that the cluster offers more possibility than this. For a full introduction, visit http://www.pace.gatech.edu/getting-started-pace-clusters.

5 Suggested direction and future work items

In the course of this project and in organizing the work well enough to write this document, we ran upon a great many possible paths. That's part of what makes this project exciting. Here are some ideas, a few specific and actionable and a few general:

1. Use better preliminary data. That is, use more reliable data that includes more loci. This would largely solve "Are we seeing the correct loci?" and "What should we do about `NA`s?". Right now Paramita is working very hard to generate such data.

2. Use more data. This one is maybe intractible, because good biological data is currently absurdly expensive to generate. But in order to be able to train a supervised learner like a neural net to distinguished diseased from not-diseased, we fundamentally (see 7.) need tens of thousands, even hundreds of thousands of samples. Keep this theoretical limitation in mind, and try to side-step this problem wherever possible. It may be possibe to do "boot-strapping" on a bit less data or to accomplish what you need with a learner that doesn't require so many training examples. It may be possible to pull data from multiple studies, but then you run in to batch effects of the worst possible kind, and we are unsure how to surmount them.

3. It is possible the linear regression model we used to fit the data and cancel strong methylation effects is not the best option. I am not sure it is advisable to try something else, because the flattened data is so flat that it appears the linear model fits exceedingly well. Plus, Ixa has said that linear models of this sort are the norm, so doing anything else requires biological justification. Depending upon the learner, it might not be necessary or advantageous to flatten the data because doing so undoubtedly deforms features pertaining to disease state. SPD, for example, is intended to find multiple, unrelated, progressions, so we should ideally be able to see one due to age, one due to sex, one due to disease state. Likewise, some learners can be trained to look for subtle cues and ignore broad effects. Consider the options carefully in each new situation.

4. It is tempting to say that an classifier accuracy of $\approx$68% when random does no better than $\approx$50% is a positive result. (Actually, until today I have been saying it is.) But it is important to remember that our $n = 244$, and that is very small. What is the probability this result is just an artefact of the data? How can we prove statistical significance? We need a strong mathematical argument to be able to confidently assert the flattened data is not entirely noise. Only then will the sanity check be complete.

5. I built units to output $N \times d$ tables of samples versus their features for use with a kNN classifier, but no such unit exists for hierarchical clustering features (to what degrees a sample expresses genes in the various modules). Right now these numbers are just floating mysteriously inside SPD somewhere, and I am not sure how to get them out. I would like to run `classify.py` on such a features and do a couple of visualizations to get an idea of how useful these features are for sample-separation in the modules-hyperspace. My guess is they will not be as good as those found via PCA or Isomap, but since they have the advantage of being biologically meaningful, maybe the tradeoff is worth something.

6. To me the results of SPD look random with respect to disease state, but looking random doesn't rigorously imply randomness. I would like to have a mathematical argument that can demonstrate that SPD is definitively not picking up trends that have anything to do with disease state or, stronger, that the patterns are entirely random and warrant no further exploration. You may or may not agree. Certainly SPD is still worth thinking about, but beware apophenia.

7. Think bigger.

   I am slightly unhappy that kNN seems to be the best algorithm for our classification problem, because kNN is relatively simplistic and is almost always outperformed by other algorithms when there is more data available. Do some thinking about alternatives and how and why and in what suitable ways.

   Note that the typical path to high performance is by framing a classification or regression problem as optimization of some loss function $f(\theta, x)$ with respect to parameters θ. In supervised, parametric learning (like neural nets), this is done by gradient descent upon reciept of each new (x, y) pair according to:

   $$\theta_{t+1} = \theta_t - \lambda \nabla f(\theta(t)), \qquad \nabla f(\theta(t)) = [\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots \frac{\partial f}{\partial \theta_n}]$$

   That is, find the direction (in the parameter-space) that will cause the algorithm to be more likely to output y given x, and adjust the parameters by a small step (size λ) in that direction.

This explanation leaves out all the best parts. I recommend Andrey Karpathy's notes for a more complete explanation of the calculus of backpropagation. But the point is that to make such a scheme work, we have to take lots of steps, and that requires WAY more samples than we have (see 2.).

Still, we should think about all the possibilities. For example, it seems to me that just as upper-layer 'neurons' in Convolutional Neural Networks (CNNs) trained on image data adjust themselves to pick out facial structures or lines in some direction or certain colors in particular sections of an image, a CNN trained on genetic data might come to recognize promoters, enhancers, genes, or even relationships between them.

If you find this list lacking, see Dr. Yi. I guarantee she has at least twice as many ideas.

6 References

1. Jaffe, Andrew E., et al. "Mapping DNA methylation across development, genotype and schizophrenia in the human frontal cortex." Nature neuroscience (2015).

2. Qiu, Peng, Andrew J. Gentles, and Sylvia K. Plevritis. "Discovering biological progression underlying microarray samples." PLoS Comput Biol 7.4 (2011): e1001123.