Elaboration on the Working Process of the Codes Related to Our Current Research

The following context is a description of the Neena's pipeline written in R located in GroupData / Kiarash / Neena pipeline (Tiffany's version of the pipeline is very similar and is located in Git/Tiffany)

This pipeline uses GEO datasets (takes as input the GEOquery ids) and provides the required files (arff files) for weka.

GEOpipeline Documentation

The general purpose of this code is to use the packages in the GEO (Gene Expression Omnibus) to create 2 text files at the end that are able to be used by Albert's Multinet Builder java program (which uses weka). In order to run this code, first you'll need to install the GEOquery so that you could use the GEO database. You could Use the biocLite.R script to install Bioconductor packages. To install the GEOquery package just type the following script in R:

```
source("http://www.bioconductor.org/biocLite.R")

biocLite("GEOquery")
```

There's another package you'll need to install. This package is called "LIMMA". LIMMA is a library for the analysis of gene expression microarray data, especially the use of linear models for analysing designed experiments and the assessment of differential expression. LIMMA provides the ability to analyse comparisons between many RNA targets simultaneously in arbitrary complicated designed experiments. Empirical Bayesian methods are used to provide stable results even when the number of arrays is small. The normalization and data analysis functions are for two-colour spotted microarrays. The linear model and differential expression functions apply to all microarrays including Affymetrix and other multi-array oligonucleotide experiments.

There are three types of documentation available. (1) The *LIMMA User's Guide* can be reached through the "User Guides and Package Vignettes" links at the top of the LIMMA contents page. The functionlimmaUsersGuide gives the file location of the User's Guide. (2) An overview of limma functions grouped by purpose is contained in the numbered chapters at the top of the LIMMA contents page, of which this page is the first. (3) The LIMMA contents page gives an alphabetical index of detailed help topics.

To install limma, type the following in an R command window:

```
source("http://bioconductor.org/biocLite.R")

biocLite("limma")
```

There's another code called GetGEODescription.R which provides the IDs that were read from GDSIDsList.txt.

This pipeline works in this structure:

1. Determine which experiments are "interesting" through a variety of steps

 2. Find intersection of genes across all experiments

 3. Determine which samples correspond to control vs. noncontrol

 4. Find top differentially expressed genes for each experiment

 5. Find intersection of genes among top differentially expressed genes

 6. Create "interesting reduced experiments" (IREs), which are essentially

 data tables that represent each of the interesting experiments and their

 expression data from each GSM sample. They are "reduced" because only the data

 from the common genes is included in each data table.

 7. Normalize IREs by using a reference IRE, finding its median, and subtracting

 the difference between the reference median and the IRE's median from each

 value in the IRE

8. Create ARFF files with the data from these genes

Lines 75 to 80 we create a table that holds all the IDs as follows:

First it puts the ids into a character vector by "c(as.character(ids)" command. Then it uses read.csv command to create a table. Here's a brief description of the read.csv command:

read.csv

The data file is in the format called "comma separated values" (csv).

That is, each line contains a row of values which are experiment IDs here,

and each value is separated by a comma. Moreover the very first row must

contain a list of labels. The labels in the top row are used

to refer to the different columns of values.

From line 101 to 106 we Create a vector that holds the IDs by using the vectorIDs <- rep(NA, 1) command which creates a vector.

lines 111 and 112 make lists to store useful Ids and pc tags by using “list()” command.

In line 116 we build a list that holds all the keywords related to control. By using the c() command. c() In its default mode is a generic function that combines its arguments to form a vector.

Lines 123 and 124 make lists to store gds (geo dataset) list and eset (expression set object) list by using the “list() command.

Lines 129 downloads vectorIDs file put it in the current directory, and load it into gds.

Line 130 turns GDS object into an expression set object (using base 2 logarithms)

Line 131 extracts the phenotype data

Line 133 and 134 load gds and eset into gdsList and esetList respectively.

Line 140 gets the "names" attribute of the pData vector using the “names” command and loads it into colNames.

Line 145 checks if the word “time” exists in the colNames if it does we go to the next experiment, if not we go to the next step.

Lines 158 to 172

Checks if "sample & description" exists in a column, if the column contains "sample or description" it will be no longer an OC column. If a column contains "sample and description" ocContainsKeyword status will change to true. Also that specific column will be loaded into a vector called colsWithKeyword and colsWithKeywordCount will be added by one.

Lines 176 to determines which column will be the phenotype column. Those lines consider 3 different states that is

1. If there is only one column with keyword the column will be counted as the Phenotype Column (PC).

2. If more than 2 columns contain keywords, the column that contains "disease.state" will be counted as PC.
3. If there are exactly 2 columns containing keywords, it will assign numbers to each keyword in a vector with the same length as colsWithKeyword. These numbers play a priority role so that the column with the lower assigned number will be counted as phenotype column. If both numbers assigned to colNames were equal we'll check if either or neither of them contains "control" then we'll assume the one with 2 factors to be the pc. If only one of those columns contain the specific keyword "control", the column that contains "control" will be counted as pc. If none of the above conditions hold, PC will be the column with the smaller number.

Line 281 to 295 check the number of levels of phenotype data of the pc. If it equals to two, the program prints that the PC does not contain 2 factors and we’ll go to the next experiment.

If it equals to two, the id which was read from the file will be saved as a useful id in the usefulIDs vector and maps the phenotype column as the tag of that dataset.

From line 322 to we tend to find ALL common genes across all experiments.

In lines 326 to 336 we assign a symbol for each gene with the usefulIDs. The symbol would be the upper case of the level of the gds (GEO dataset).

Line 337 prints the number of genes in the gene list.

From line 343 to 363 we tend to find the intersection of all genes across all these platforms. To do so the code compares first and second list and find total Common Genes. Then it compares the third list to commonList to make new total Common Genes. And it continues until final list is reached; then totalCommonGenes will have all common genes.

For comparing the genes, the programmer has used “intersect” command.

A new part of the code begins in line 369. In this part we tend to merge all samples for control - create two lists, one to hold all the samples that correspond to control and one for non-control. Also we’ll create a list that holds each noncontrol sample's corresponding experiment tag.

Line 374 builds a list that holds all the keywords related to control. By using the c() command.

In lines 378, 381, and 383, the code builds three lists for control, nonControl, and experiment tag.

From lines 386 to 389 we get a sample of the pdata of each usefulID. After that we extract

the index and the tag of the sample. If the sample tag contains any of the control keywords,

the sample will be added to the control list otherwise it will be added to the nonControl list.

Moreover we add the pc tag to a vector called experimentTagList. This vector has the same size as the sample. For extracting the sample from the pdata, the code uses $ command as follows:

pdata$sample

From line 434 we tend to find differentially expressed genes for each experiment. First we create two lists of "listOfDiffExpGeneLists" and "listOfTabd" that hold each GDS's differentially expressed genes and GDS's topTable respectively.

From Line 438 to 487, we are searching in the usefulIDs which were found in the previous parts of the code.

From lines 441 to 443 we get the name of the tags of those IDs and load them into pcName. Also we get their expression set and load them into eset. Moreover their GEO data set will be loaded into gds in line 443.

In line 444 we build a table from the gds of line 443 by the "table" command.

By setting the useTotalCommonGenes to true or false we could decide whether to use all genes or not in line 446.

If the useTotalCommonGenes would be false then we use the topTable to find the top differentially expressed genes.

from line 453 to 455 we get gene indexes of top differentially expressed genes. To get the number of top genes, we use the following command:

numTopGenes <- ceiling(length(gdsTable[["IDENTIFIER"]]) * (percentTopGenes+1.5) / 100)

tabd <- topTable(fit, coef=2, adjust="fdr", n=numTopGenes)

geneIndexes <- as.integer(row.names(tabd))

The ceiling command takes a single numeric argument x and returns a numeric vector containing the smallest integers not less than the corresponding elements of x.

After that from line 456 to 486 we get the gene IDs and convert them into gene symbol.

From the line 488 we tend to find common top differentially expressed genes across all experiments.

From line 489 to 499 like the previous part we compare each list to each one and find all common genes using the “intersect” command.

From the line 502 we begin to create reduced experiments (IREs).

From line 505 to 547 we go through the useful Ids. For each useful ID, we first reduce the table based on common genes. To do so we first build a non reduced table and reduce it by picking commonTopExpGenes. For this to come true we use the “subset” command that **r**eturns subsets of vectors, matrices or data frames which meet conditions.

Then from line 514 to 545 we collapse the expressions values of all IDs corresponding to a single gene in each experiment to get one value for each gene for each sample. To do so we go through each gene symbol and generate a list of all expression values for each sample. Then we go through each sample and determine its max expression value for this gene, and add it to the list. Finally we have a list of Boolean values that represents each row of the IRE table

# TRUE if the row corresponds to this gene, FALSE otherwise

We will delete the rows that are FALSE except for the first one, which will be modified to hold the new list of max expression values for each sample for this gene symbol.

The next step would be normalization. To normalize we find the median across all gene expressions across all samples in each IRE. Lines 551 to 565 are written to find the median. After that we select one IRE as reference and for all other IREs, subtract the difference between its median and the reference median from all gene expressions. Lines 570 to 598 are written to do such thing.

The next part of the code that starts from line 600 688 considers the state that useTotalCommonGenes is TRUE. If we want to use all common genes, then the "commonGeneList" will be all common genes, otherwise, the list will be only common top differentially expressed genes.

From line 605 we get expression data from each experiment for each common gene and create new files for weka using the “write” command. Here’s an implementation of the write command:

```
write(x, file = "data",
     ncolumns = if(is.character(x)) 1 else 5,
     append = FALSE, sep = " ")
```

Arguments

x	the data to be written out.

file	A connection, or a character string naming the file to write to. If "", print to the standard output connection. If it is "|cmd", the output is piped to the command given by ‘cmd’.
ncolumns	the number of columns to write the data in.
append	if TRUE the data x are appended to the connection.
sep	a string used to separate columns. Using sep = "\t" gives tab delimited output; default is " ".

From line 696 to 731 we consider the state that useTotalCommonGenes is false. Here we make a table of the p-values for all common genes for each experiment.

In the following context we'll elaborate on a research that is done by Skanda Koppula using dbGaP data. There is a good documentation in the codes as well.

Skanda

Here is a brief instruction on how to use the **Skanda's Project** to **perform enrichment using the Fisher test**. This document references data that can be found in "skoppula / portable-fisher-pipeline.zip".

How to run?

This program uses some java classes and utilities that are in java archive files. So we need to put those jar files in the CLASSPATH. We have to copy the "pipeline_lib" directory into root directory (or somewhere else). Excecuting "pipeline.run" script will do all the work (in case of copying "pipeline_lib" somewhere else, we have to change that script to point to the "pipeline_lib" directory. It's very important).

Properties

This program read all the data that needs from "properties.txt" file. To make the program work we have to:

1. Set the appropriate values for properties.txt:

    a. Change train and trainFull to reference your two [hopefully distinct] training data sets. They both must have a file path assigned.
    Notice I set them in a folder called data so my variable assignments are as follows:

    ```
    train = ./data/coga-reduced-3.arff

    trainFull = ./data/coga-reduced-2.arff
    ```

    b. Choose which pathway libraries you want to use. I have a set of this pathway to gene mapping files in the folder pathway-libraries. In my example, I used the KEGG one:

```
conceptToGenesList = C:/Users/Skanda/Desktop/pathway-
libraries/c2.cp.kegg.v3.0.symbols.gmt        KEGG
```

c. Change the variables snpToGenes and backgroundModel to be the directory of your WGAViewer SNP annotation output. For example, right now I have the variables set to reference a file [snp-annotations-small.csv] in a folder called data. Both variables must be the same. You can assign these variables even if you select 'n' to isSNPData. I recommend assigning both variables to a larger map, which maps more number of SNPs.

```
snpToGenes = ./data/snp-annotations-large.csv

backgroundModel = ./data/snp-annotations-large.csv
```

d. Assign outputDir and savePValuesFileName to the folder you want created and your output saved in...I believe the reason why the output is empty for the sample data, is because the input train and trainFull data are both from the same source.

e. Change numEnvVariables to be the number of environmental variables you have in your two data files.

f. Increasing the variable 'iterations', increases the number of random classifiers in distributions created, which may increase significance of results. However, increasing iterations uses oodles of memory.

As I mentioned above, Do NOT disturb the “pathway_lib” folder and you should be good to go. The first six lines of “pipeline.run” should NOT be changed, unless you wish to relocate the location of these libraries. In which case just put the appropriate directories in place of pipeline_lib/***.jar

You can change parameters of the last line of “pipeline.run” script to utilize your system.

1. Change the 1 in Xmx1g to the amount of RAM you want to allocate to this program.
2. You can add the -d64 or -d32 flag after 'java' if you are using a 64-bit or 32-bit machine...but you don't have to.

Source Files

You can find the source code of this pipeline in “skoppula/code/pipeline-with-src-skanda-april2012/src” folder. This source files are using the jar files in “pathway_lib” too. So remember to add those jar files to your classpath in case you want to compile and run from source files.

The main function is in “main2/Main.java” that reads the properties from the specified file and does all the work. In the main function, first we extract gene symbols from the file that pointed in properties file as “snpToGenes”. Then we generate a map that maps concepts to genes based on the “ontologyPath” that set in the properties file, and then we create a map for maping concepts to genes from the result of two parts that mentioned above.

There are some packages in source code. Lots of files in package “main” and “main2” are the same. But as the other source files using some classes of “main” and the other files using some classes of “main2”, combining these tow packages needs reading the code carefuly.

Reading the comments in “main2/Main.java” will also help to understand what the parameters are and what their format is.

Utilities

There are some utilities that might be helpful. Note that most programs must be compiled and executed by the user, because the input directories/names were hardcoded into the source. That means, to use these programs in your project now, you'll need to change the value of some of the variables (named something along the lines of "inputfile") to suite your project's specifications.

- **skoppula / code /CommonPathwayFinder.java:**

  This file used to determine the pathways (a set of feature lists) in common, between to an arbitrary number of set of textfiles. These textfiles are generally pipeline output files, each of which is a listing of the significant pathways, each pathway's p-value and its AUROC. This is used to identify reproducible/robust pathways. The program outputs those pathways in common between the pipeline's results of significant pathways (the results after analyzing two or more data sets)

- **skoppula / code / DataPartition.java:**

The program was initially used to split and distribute the SNPs a large list of SNPs to an input, n, number of textfiles. This enables a method of 'manual parellization' each smaller list can be processed on a different machine to create a SNP to gene mapping. This would result in n gene-snp mapping files, which we combine using the small executable found:

Skoppula / code /merge-data.bat

Please note when using this tool to remerge, if you have a descriptor header row for each mapping file, it will be included multiple times in the merged file. Delete the descriptor header row on each spreadsheet.

DataPartition is one of the handful programs that 'generalized'. You can run the jar file found at "skoppula / code /DataPartition.jar" with parameters of the SNP list directory and num files.

- **skoppula / code /FileWritingPerfomanceTest.java:**

  This file used to optimize other IO intensive programs in the project. The program was a small test that used to compare the time-performance of the common file writing methods in Java.

- **skoppula / code /Format_SNP_List.java:**

  Converts the list of features preceding data in an .CSV file to a list of features in a seperate text file in the .wr format. This text file is commonly used later in analysis in WGAViewer.

- **skoppula / code /OntologySizeCount.java:**

  Determines the distribution of pathway size (measured by number of genes/SNPs), given a database listing of pathways. This is useful to determine the size of the parameters.

- **skoppula / code /ReducingDataset.java:**

  Takes in training/testing data, and given a number of features and number of patients, it reduces the data to contain those number of features for those number of patients.

- **skoppula / code /SelectFromCompleteDataSet.java:**

  Reads the first labels of the "n" features from the training data text file, then outputs just those labels into a seperate text file.

Converting Data Types

Here is a brief instruction on how to convert between data types: Binary [PLINK] files to .RAW files to .CSV files to .ARFF files, and RAW files to .ARFF files.

CONVERT FROM BINARY FILESET TO .RAW IN PLINK:

Use this command in command line:

```
plink --bfile [name of binary fileset without extensions, e.g. merge]
                                                            --recodeA --out [desired name of raw
file, e.g. merge-raw]
```

CONVERT FROM .RAW TO .CSV

After you have the raw file, you can run the code below to generate a CSV. Make sure you include the jar file I sent you in your classpath. Also, change the filepaths at the top of the code as needed (e.g. if you download those files from /scratch/James/Random to your local machine or some other supercomputer and run the code there).

You also need to change the lines:

```
        BufferedReader br = new BufferedReader(new FileReader(path + "snp.snplist"));
```

And

```
        ProduceCSVGeneric.writeEffEnd(path + "merge-raw.raw", path +"merge.csv",
env, setTitles, bins, snps);
```

to point to the appropriate files (snp.snplist should just be a list of all SNPs in the data file). merge-raw.raw should be changed to the name of the raw file to be converted and merge.csv should be changed to the desired name of the csv output.

```
        String path = "/scratch/James/Random/";
        String numeric = "/scratch/James/Random/envdiscnogen.txt";
        String envOne = "/scratch/James/Random/phs000092.v1.pht000116.v1." +
                "p1.c2.AlcoholDepAdd_Data.ARC.txt";
        String envTwo = "/scratch/James/Random/phs000092.v1.pht000116.v1." +
                "p1.c1.AlcoholDepAdd_Data.HR.txt";
        List<String> titles = new ArrayList<String>();
        Map<String, List<String>> env = new HashMap<String, List<String>>();
        Map<String, Boolean> storage = new HashMap<String, Boolean>();
        EnvUtils.readNumeric(storage, numeric); //which variables are continuous and
which are discrete
        for (String var : storage.keySet())
                titles.add(var); //all the demographic variables to include in csv
```

```
		Map<String, List<Integer>> bins = new HashMap<String,
			List<Integer>>();
		EnvUtils.getBins(bins, storage, envOne, envTwo); //bins for discretization of
continuous variables
		DataUtils.readEnv(envOne, env, null, titles); //read demographic variables from
the text files
		DataUtils.readEnv(envTwo, env, null, titles);
		Set<String> snps = new HashSet<String>();
		BufferedReader br = new BufferedReader(new FileReader(path + "snp.snplist"));
		String line = null;
		while ((line = br.readLine()) != null){
			snps.add(line); //read all the SNPs to include in the csv
		}
		br.close();
		Set<String> setTitles = new HashSet<String>();
		setTitles.addAll(titles); //adding titles to Set so they can be passed to
writeEffEnd()
		ProduceCSVGeneric.writeEffEnd(path + "merge-raw.raw",
			path + "merge.csv", env, setTitles, bins, snps);
```

CONVERT FROM .CSV TO .ARFF

You can use WEKAUtils.convert() or this command:

```
java -classpath Myweka.jar -Xmx..g  weka.core.converters.CSVLoader  file.sc >
file.arff
```

You should use the function WEKAUtils.convert([path to csv], [desired path to arff]) in my library to convert from csv to arff.

CONVERT FROM .RAW to .ARFF

Run the Yixin pipeline!

Yixin Pipeline

You can find Yixin pipeline in the “skoppula / code / pipeline-yixin-version” directory. And the source code for that in “Kiarash/”dbGaP to arff”

- You can learn how to use our pipeline by running the pipeline with the --help parameter. All other parameters will be ignored if you use this:

```
java -jar pipeline.jar –help
```

- The --fast parameter is a simple way to convert .raw to .arff, especially good for large files. However, it will only do the following changes:
    - Replace "NA" to "?"
    - Replace " " to ",", like a csv file
    - Generate the @attribute part(only all the SNPs, SEX and PHENOTYPE), but the value are all {0, 1, 2}
    - Generate the @relation part based on parameter
    - Generate the @data part.

  If you use this parameter, The --src, --des, and --name parameters are ALWAYS NECESSARY:

  --srcR or --srcA specifies the path of the .raw/.arff file, --des specifies the path of the .arff file. If file exists, the program will ask whether to overwrite it; if the file didn't exist but the directory existed, it will automatically create new file; if neither the file or directory exists, it will give the error. In the mean time, when the program doesn't use the fast method(--fast), it will also create a file with same name as --des specified but end with .temp, which is the temp file only record all the instances with the necessary SNPs .list file requires. --name(necessary for raw file) gives the relation name for the .arff file. The name can be separated by blank. --list(necessary for arff file) gives the .list file path. The names of the SNPs in this file don't need to be the same order as .raw file.

- --test replaces all the{0, 1}, {0, 2} and {1, 2} in SNPs with {0, 1, 2}

- You can find two sample input files in:

  testForRaw.list , testForRaw.raw

  For example, you can run the pipeline with these parameters via:

  java -jar pipeline.jar --srcR testForRaw.raw --des outForRaw.arff --list testForRaw.list --name MajorDepression –test

  The corresponding output of this can be found in:

  sample-output.txt

The context below is about the java files which are located in the multinet.zip folder. They make the Multinet Bayesian networks from arff files.

Multinet.zip is located in GroupData / Kiarash / Neena pipeline

BayesMultiNetTan file:

This file, loads the data file in CSV or ARFF with weka's internal format and implements the TAN Bayesian classifier. Much of this implementation is taken form weka.classifiers.bayes.BayesNet.java. Estimator associated with the class attribute, represented as a BayesNet with only one attribute. This allows easy computation of the class probability density as well as the score.

It has some function and describing it at follow:

globalInfo:

In the function of globalInfo, will return a string describing the classifier.

listOptions:

This function returns an enumeration describing the available options.

setOptions:

Function of setOptions, parses a given list of options. Parameters of this are the list of options as an array of strings. If an option is not supported throws the exception.

getOptions:

In the function of getOptions, gets the current settings of the classifier.

getCapabilities:

This function, returns default capabilities of the classifier.

buildClassifier:

Function of buildClassifier, generates the classifier and the parameter of this, is set of instances serving as training data and if the classifier has not been generated successfully throws exception. This function remove instances with missing class and ensure we have a data set with discrete variables only and with no missing values. Then filter all instances with a particular class label. At last, update probability of class label, using Bayesian network associated with the class attribute only.

toString:

Function of toString, returns a description of the classifier. For the multinet TAN classifier, this returns a list of structures prefaced by the corresponding class labels.

distributionForInstance:

In the function of distributionForInstance, calculates the class membership probabilities for the given test instance. The instance to be classified is the parameter of this function. If there is a problem generating the prediction the exception throws. This function Find maximum probability to facilitate normalization and transform from log-space to normal-space.

enumerateMeasures:

In the function of enumerateMeasures, returns an enumeration of the measure names. Additional measures must follow the naming convention of starting with "measure", eg. Double measureBlah().

Graph:

In the graph function, returns a graph representation of this classifier. This method outputs the structure associated with the class label given by the <code>-C</code> option.

And this file has some getter and setter function to set and get the value. At last, it has main function for testing it and ensures that all variables are nominal and that there are no missing values. Data set to check and quantize and/or fill in missing values are the parameters of this.

It creates the Bayesian network and computes the membership probability of each class and assign the class label that has maximum probabilities.

BMNTCCMultiNetSimpleEstimator file:

We have some functions that describe at follow:

globalInfo:

returns a string describing this object that is a description of the classifier suitable for displaying in the explorer/experimenter gui.

logDist:

The other function is logDist that calculates the class membership probabilities for the given test instance. For this particular implementation, a single probability is returned, since the MultiNetSimpleEstimator is associated with a single class label. It has two parameters; bayesNet that is the bayes net to use and instance that is the instance to be classified. This function ignore class index, since this estimator is associated with a single class label.

BMNTFeatureDiffrentiation file:

BMNTFeatureDifferentiation class:

This class extend from BMNTCaseControl class. Is has some function that we describe at follow:

splitInstances:

this function want to split the input Instances . It filters all instances with a particular class label. It means, it get iClass (inter parameter) as input and split instances with iClass class label. For doing this each instances just maintain selected feature. It generates a tree structure for iClass class label based on greedy search. And update probability of this class label.

Because output type is void this function doesn’t return anything.

MultinetSimpleEstimator file:

The other class in this file is MultinetSimpleEstimator that it extend from SimpleEstimator class.

It is the extension of simpleEstimator which can handle multinet approach of BayesMultiNetTAN.

MultiNetSimpleEstimator is used for estimating the conditional probability tables of a multinet Bayes network once the structure has been learned. Estimates probabilities directly from data.

GlobalInfo:

one function of this class is GlobalInfo that it just return some information about class .

distributionForInstance:

This function calculates the class membership probabilities for the given test instance. In this implementation, a single probability is returned.

Parameter: bayesNet is a bayes net to use.
                    instance is the instance to be classified
function return a predicted class probability for instance
if there is a problem in generating the prediction function throw a exception.

This function call another function in its body that, its name logDist function.

logDist:

that calculate the iCPT from value of the node parents up to bayes Net algorithm.

It has two parameters. One of them is bayesNet and the other instance.

BMNTCaseControl file:
in this file we have BMNTCaseControl class that it extend from BayesMultiNetTAN class.

The value for control in the class attribute in Instances must be "1N." The phenotypes of the Instances supplied to buildClassifier() should be differentiated (i.e. one for each comorbidity), but Instance objects supplied to distributionForInstance() should not (just "1N" for control and "2N" or something else for case).

BuildClassifier:
One of the function is buildClassifier. It removes instances with missing class and ensure we have a data set with discrete variables only and with no missing values. At last, it updates probability of class label, using Bayesian network associated with the class attribute only.

SplitInstances:
it filter all instances with a particular class label and generate tree structure for this class label.

LogDist:
this function ignore class index, since this estimator is associated with a single class label.

For run of each class in Linux you should have package of weka.jar near the code and write the command line of java -jar weka.jar Bayes.java BayesMultiNetTAN.java at console.

HIVMultinetBuilder file:

This java file wants to create a multinet using merged data from experiment that achieved. to run this file in NetBeans move all the text files from folder to the folder that has the build.xml file. !!

In HIVMultinetBuilder there is object from buildMultinet that used to build multinet classifier. You can change it to buildBayes to build simple Bayesian classifier.

buildBayes:

This function create simple Bayesian network (TAN). It read data in ARFF format. After that it descretize data to 50 bins.

After creating Bayesian network classifier give a index for each class and after that run a cross-validation of Bayesian network to validate a classifier that have been made. And after this all compare accuracy ratios.

buildMultinet:

It reads data from “filename” file that has ARFF format. And create filewriter with “output_filename” name. Classes are in the first file. It descretize data to 100 bins. After that it builds multinet using kent’s BayesMultiNetTAN class. After that again it run 10-fold cross-validation to get a better accuracy. And at last it again compares the accuracy ratios. And write to that output file.