How to determine which feature to split in the Breiman's random forest algorithms is from the following references: https://stats.stackexchange.com/questions/144818/does-breimans-random-forest-use-information-gain-or-gini-index and https://en.wikipedia.org/wiki/Decision\_tree\_learning#Gini\_impurity

How to find the frequent itemsets and association rules are from http://r-statistics.co/Association-Mining-With-R.html#Caveat%20with%20using%20Lift

**Feature selection– Random Forest (RF) based**

Class for control data is set to be 1 and -1 is set for case data before building the RFs. All the control data and the patient data are combined to be the training data. The training data with missing value imputed

will be used in this homework to identify the important features. The method to impute the missing value is the same as in homework 1. The method is as follows: First, setA and setB are created for dataset

without missing data and with missing data. Then, knnImputation in R is used to fill in missing data using 5 nearest neighbors. For every missing value to be imputed in set B, it identifies 5 closest observations in setA based on the euclidean distance and computes the weighted average (weighted based on distance) of these ‘5’ neighbors as the imputed values. The imputed data for WGAAD2 in GI\_10047093-S and GI\_10047133-A are showed below as an example.

Classes GI\_10047091-S GI\_10047093-S GI\_10047103-S GI\_10047133-A

WGAAD2 -1 -1.9848353 -0.37700698 0.076412305 -1.3768435

Finally, I combined setA and setB with missing data imputed as the training data.

To build random forests using the training data, randomForest function inside the randomForest package in R is used. 500 Classification trees were built by setting the parameter “ntree” to be 500. Classification trees are built using factor Classes as the response vector and all the 800 genes as the predictors. To build RFs by fixing the number of features (genes) at 70% of the total number of features, “mtry” parameter is set to be 560 (=0.7\*800), which means 560 genes are randomly sampled as candidates at each split. To vary the number of instances from 50% to 90%, with an increment of 5%, a loop and sample function is used. Sampling via “sample” takes place from 1 to 364 observations and chooses desired number of instances randomly by setting the corresponding “size” argument inside the “sample” function. Since the required interval (5%) sometimes will give us a noninteger, so a “round” function is used to round the value to have no decimal. The IEC 60559 standard is used, which means ‘go to the even digit’. Therefore, round(0.5) is 0 and round(0.6) is 1. To get the frequency of each gene appearing in each RFs, “varUsed” function is used. It takes in an object of class randomForest and by setting the parameter “count = TRUE”, an integer vector containing frequencies that variables are used in the forest is returned by this function. Top 10 frequent genes appearing in the RFs using different number of instances are showed below as an example of the results from “varUsed” function.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RFs(70% features) | Top 10 frequent genes | | | | | | | | | |
| 182 instances | GI\_14249537-S | GI\_14277689-S | GI\_13899314-S | GI\_14150038-S | GI\_13259519-S | GI\_13375790-S | GI\_13259551-A | GI\_14149782-S | GI\_14249161-S | GI\_13325059-S |
| 218 instances | GI\_14249537-S | GI\_14591908-S | GI\_11037060-A | GI\_13435144-S | GI\_10864046-S | GI\_14195617-A | GI\_13375790-S | GI\_13259551-A | GI\_13514821-A | GI\_11559926-S |
| 255 instances | GI\_14249537-S | GI\_14702170-I | GI\_13375790-S | GI\_13236552-S | GI\_14277689-S | GI\_13259551-A | GI\_13518031-S | GI\_13489082-S | GI\_14670363-I | GI\_11056011-S |
| 291 instances | GI\_14249537-S | GI\_13514821-A | GI\_14702170-I | GI\_13129091-S | GI\_11968044-S | GI\_13325059-S | GI\_12056970-S | GI\_14670363-I | GI\_13489082-S | GI\_13899304-S |
| 328 instances | GI\_14249537-S | GI\_13259551-A | GI\_13375790-S | GI\_13129091-S | GI\_13775223-S | GI\_14589934-I | GI\_14249499-S | GI\_12056477-S | GI\_14702170-I | GI\_14149980-S |
| 200 instances | GI\_14249537-S | GI\_14670363-I | GI\_11037060-A | GI\_11641417-S | GI\_10801344-S | GI\_14149858-S | GI\_13899314-S | GI\_13259551-A | GI\_13569947-S | GI\_13899232-S |
| 237 instances | GI\_14249537-S | GI\_14591908-S | GI\_13129091-S | GI\_14574565-A | GI\_14702170-I | GI\_10864046-S | GI\_11038661-S | GI\_11967986-S | GI\_14150038-S | GI\_14149858-S |
| 273 instances | GI\_14249537-S | GI\_14249499-S | GI\_14702170-I | GI\_11545846-I | GI\_14249161-S | GI\_13937360-S | GI\_14149980-S | GI\_11321616-S | GI\_13899314-S | GI\_10835229-S |
| 309 instances | GI\_14249537-S | GI\_13375790-S | GI\_12056477-S | GI\_10864046-S | GI\_13775223-S | GI\_14149980-S | GI\_11321616-S | GI\_13129091-S | GI\_14149858-S | GI\_14589934-I |

Using the same method described above, I can compute the frequency of each gene appearing in the RFs when I fix the number of instances at 70% of the total number of instances, vary the number of features (genes) from 50% to 90%, with an increment of 5%. Top 10 frequent genes are tabulated below as an example of the results.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RFs(70% instances) | Top 10 frequent genes | | | | | | | | | |
| 400 features | GI\_14249537-S | GI\_13375790-S | GI\_13375925-S | GI\_13236552-S | GI\_13129091-S | GI\_13787213-I | GI\_11496880-S | GI\_13435144-S | GI\_13775223-S | GI\_14149858-S |
| 480 features | GI\_14249537-S | GI\_14149858-S | GI\_11496977-S | GI\_14249161-S | GI\_14589934-I | GI\_12751496-S | GI\_13899314-S | GI\_11056053-S | GI\_12232472-S | GI\_11545846-I |
| 560 features | GI\_14249537-S | GI\_13375790-S | GI\_13899314-S | GI\_13259551-A | GI\_13325059-S | GI\_14249449-S | GI\_13514821-A | GI\_14589934-I | GI\_14149980-S | GI\_13376368-S |
| 640 features | GI\_13375790-S | GI\_14249537-S | GI\_11321616-S | GI\_14149980-S | GI\_13375737-S | GI\_10864046-S | GI\_11968044-S | GI\_13775201-S | GI\_13775223-S | GI\_14149858-S |
| 720 features | GI\_14249537-S | GI\_11321616-S | GI\_13899304-S | GI\_13430867-S | GI\_13775223-S | GI\_13375790-S | GI\_11968044-S | GI\_13325074-S | GI\_14149994-S | GI\_13375925-S |
| 440 features | GI\_14249537-S | GI\_13375790-S | GI\_13899314-S | GI\_11321616-S | GI\_13775223-S | GI\_10864046-S | GI\_14702170-I | GI\_13514816-A | GI\_13325074-S | GI\_14150038-S |
| 520 features | GI\_14249537-S | GI\_13899314-S | GI\_13375790-S | GI\_12232414-S | GI\_13325059-S | GI\_11545760-S | GI\_10801344-S | GI\_14249383-S | GI\_13375978-S | GI\_12965190-S |
| 600 features | GI\_14249537-S | GI\_13384601-S | GI\_10835100-S | GI\_14149858-S | GI\_13124878-A | GI\_11496977-S | GI\_10801344-S | GI\_14149980-S | GI\_13775223-S | GI\_14249177-S |
| 680 features | GI\_14249537-S | GI\_13259551-A | GI\_13375790-S | GI\_11321616-S | GI\_12056477-S | GI\_11038650-S | GI\_13435384-S | GI\_13376839-S | GI\_14149858-S | GI\_14149980-S |

The frequency of a gene in a RF tells us how important this gene is for the classifier. When there are two genes with different frequencies, generally, we can say that the one with higher frequency is more important than the other one to determine whether the observation will have AD disease. The reason is as follows: Since the algorithms randomly select the genes, all the genes are equally likely to be selected as candidates to build a tree. As I mentioned above, I have used randomForest function inside randomForest package in R. randomForest implements Breiman’s random forest algorithm. In this algorithm, to decide the overall best split across mtry variables, the code uses a scoring function similar to gini-gain (Reference: https://stats.stackexchange.com/questions/144818/does-breimans-random-forest-use-information-gain-or-gini-index and https://en.wikipedia.org/wiki/Decision\_tree\_learning#Gini\_impurity).

GiniGain(N,X)=Gini(N)− Gini(N1)−Gini(N2), Where X is a given feature, N is the node on which the split is to be made, and N1 and N2 are the two child nodes created by splitting N.

and Gini(N) = 1−, where K is the number of categories in the node

Gini impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. GiniGain is pretty similar to the information gain. Splitting the tree using one feature indicates the importance of that feature.

Not all 8,000+ (or 800 in this problem) genes are important or useful. If we can only use a small number of genes, say 100. I will first sort the genes according to their frequency (from largest to smallest). Then I will use the first 100 genes. However, as we see the orderings of genes in different RFs are different, so I will use the first 100 genes (as determined from the frequency) from the best RF model with number of features and instances tuned. The mean of the out of bag error is selected as the indicator to tune those parameters. The results are as follows:

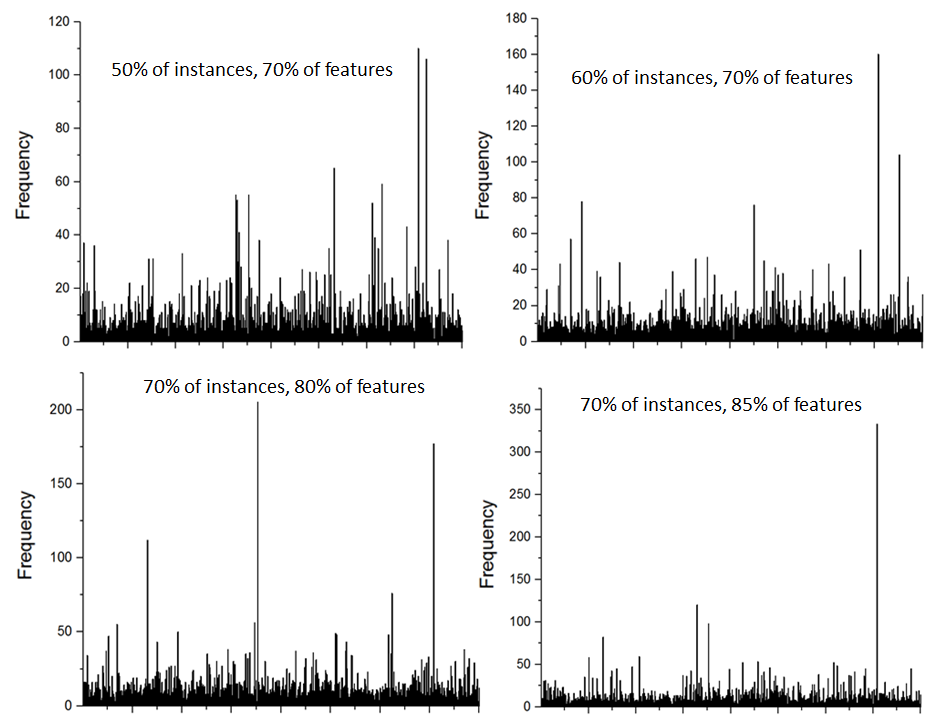
|  |  |  |  |
| --- | --- | --- | --- |
| Percentage of instances | Mean of out of bag error rate | Percentage of features | Mean of out of bag error rate |
| 0.5 | 0.340867549348496 | 0.5 | 0.358316645994261 |
| 0.55 | 0.340491786529378 | 0.55 | 0.331435870370839 |
| 0.6 | 0.357775818183236 | 0.6 | 0.367714924752537 |
| 0.65 | 0.407456187527614 | 0.65 | 0.382122673775259 |
| 0.7 | 0.307616186621753 | 0.7 | 0.324881473349868 |
| 0.75 | 0.342543723194622 | 0.75 | 0.359023765990902 |
| 0.8 | 0.341763491215925 | 0.8 | 0.323572032706278 |
| 0.85 | 0.322683308183123 | 0.85 | 0.323961303012913 |
| 0.9 | 0.323293751082005 | 0.9 | 0.313437695334443 |

The above table suggests 70% of instances and 90% of features have smallest out of bag error rate. So the best RF will be the one using these two parameters. And the 100 genes will be the top 100 frequent genes appearing in it. The top 10 genes are showed below.

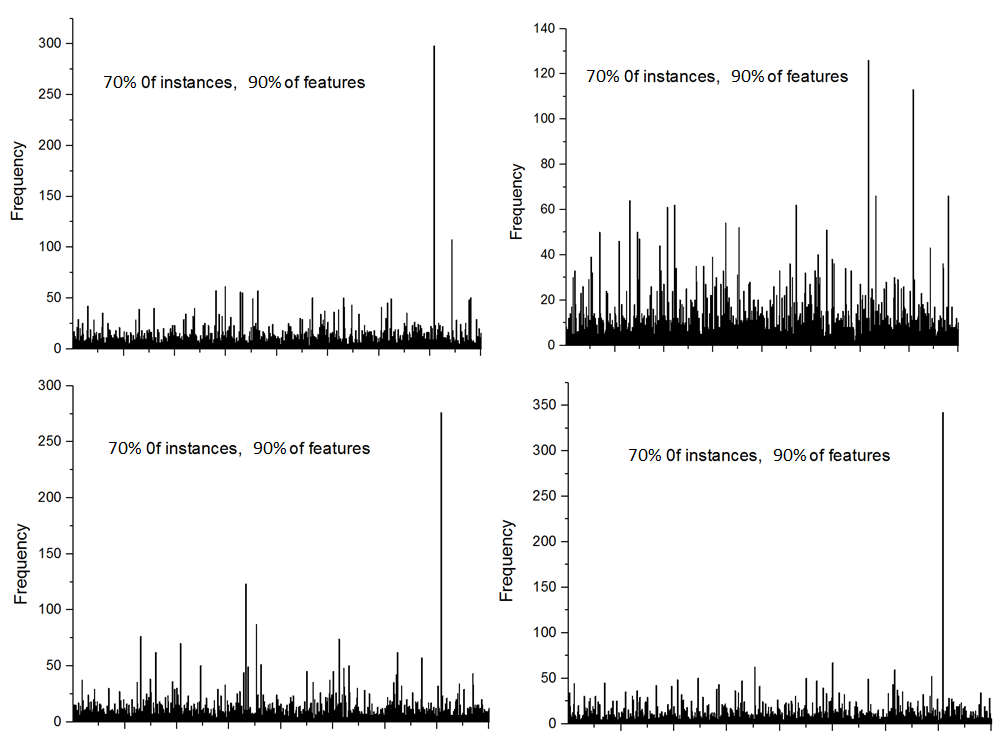
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| GI\_14249537-S | GI\_11321616-S | GI\_13899304-S | GI\_13430867-S | GI\_13775223-S | GI\_13375790-S | GI\_11968044-S | GI\_13325074-S | GI\_14149994-S | GI\_13375925-S |

The above tables showing the top 10 frequent genes using different instances and different features suggest that GI\_14249537-S is very likely the most important gene. However, the orderings of genes are not stable. For example, gene GI\_13375790-S is the second frequent gene in the RF with 50% of features and 70%of instances, but it is not always the second frequent gene in other RFs. It is more obvious when we plot the frequencies of genes in RFs with varying number of instances and features (see plot 1, x\_axis in the plot is the genes in sequence as showed in “case.gex”), we can see the orderings of the genes using the RF method are not stable. This suggests the tuning variables (e.g. the number of the instances or the number of features) will affect the stability. This is reasonable, because the observations have variance and different instances may suggest different strong features. With different number of the instances selected, we may have different datasets used for RF tree building which may suggest different strong features. Especially, when the number of the instances is small, the result will be less stable. In terms of the number of features, they will affect the feature candidates so they will also affect the stability of the orderings of the genes. Some other factors that can affect the stability are the number of trees, the fraction of relevant features, the variation among instances, and how strong some features are compared with other features. Smaller fraction of relevant features and big variation among instances will make the orderings of genes less stable. When the relevant features are equally strong, the orderings of genes will also be less stable. When the number of trees is not big enough, we can’t get enough statistics. These factors lead us to have different orderings of the genes even if we fix the number of features and instances, like 70% of instances and 90% of features. The results are as shown in plot 2. Generally, with very large number of instances and features, we would expect to get very stable orderings of the trees. It then is just a decision tree.

I would suggest the following procedure to quantify the stability if we care about the orderings for each gene. First, tune the variables in the RFs to find the best parameters according to the out\_of\_bag error. Then using these parameters, we run the RFs several times (like *m* times). Each time, we record the order of the gene according to its frequency and assign its order to the corresponding gene. We then check standard deviation of the order for specific gene throughout *m* runs. Finally, we sum up the standard deviation for the first 100 frequent genes (determined from the median value of the order throughout *m* runs) and use it to represent the stability. If we just care about what the top genes are and we don’t care about their orderings, we can run the tuned RFs several times (like m times) and record the top frequent genes each time (for example top 100 frequent genes). Then we can check how many percentages of genes are top 100 frequent genes for all the m runs. And we can use it to quantify the stability.



Plot 1: The frequency histogram of genes appearing in the RFs by varying the number of features and number of instances.

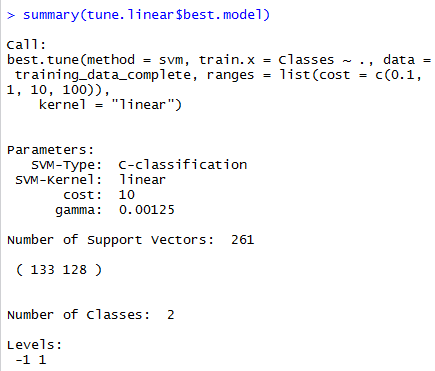


Plot 2: The frequency genes appearing in four independent RFs with 70% of features and 90% of instances

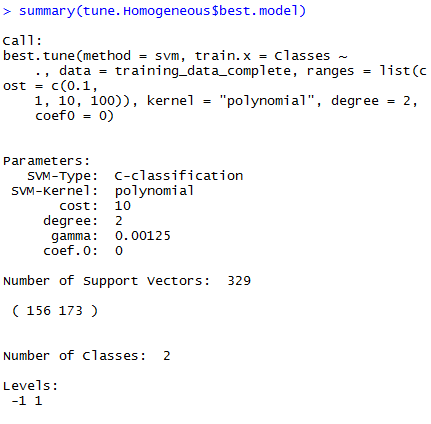
**Feature selection – SVM**

SVM model was built using factor Classes as the response and 800 genes as the predictors. The data for the AD cases and normal controls combined with missing data imputed were used for training data. Moreover, the predictor gene data were scaled internally to zero mean and unit variance during the model building. Soft margin SVMs are used here. A cost argument allows us to specify the cost of a violation to the margin. When the cost argument is small, then the margins will be wide and many support vectors will be on the margin or will violate the margin. When the cost argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin. The e1071 library in R includes a built-in function, tune(), to perform cross validation and provide the best model with cost parameter tuned from a range of c(0.1,1,10,100).

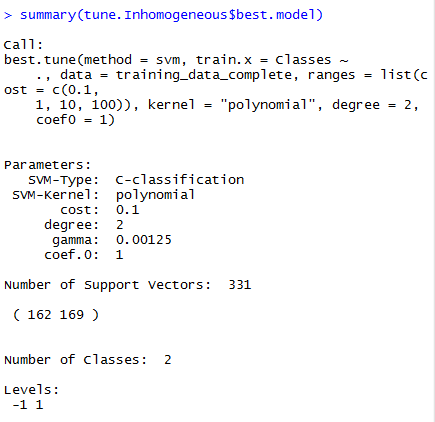
“tune. linear$best.model”, “tune. Homogeneous$best.model”, and “tune. Inhomogeneous$best.model” store the best model with cost parameter tuned using linear kernel, Homogeneous quadratic kernel, and Inhomogeneous quadratic kernel. These models have the lowest cross-validation error rate. Summary function gives the parameters of SVM and number of support vectors. They are showed below.



The tuned cost parameter for linear kernel is 10. The number of support vectors is 261



The tuned cost parameter for homogeneous quadratic kernel is 10. For the polynomial kernel, it has the form . Setting “degree”=2 and “coef0” =0 suggests the kernel used here is k(x, x’) = (xTx’)2 . The gamma used is also shown above. Moreover, the number of support vectors is 329.



The tuned cost parameter for Inhomogeneous quadratic kernel is 0.1. degree i=2 and coef.0=1 suggests the kernel used is k(x, x’) = (xTx’+1)2 . The number of support vectors is 331. It has more support vectors than the homogeneous quadratic kernel.

Given an SVM model f(x) = (wTv+b), i.e., w and b are given, the larger absolute values in vector w, the more important those dimensions (features) are. The decision function is f(x) = sgn (wTv+b). When building the SVM model, predictor gene data were scaled internally to zero mean and unit variance. Then with larger positive w and v is positive, it is more likely the sign of f(x) will be positive. When w is negative and the absolute value of w is larger and v is positive, the sign of f(x) will more likely to be negative. Thus the larger absolute values in vector w, the more important those dimensions (features) are.

As I mentioned above, the absolute values in vector *w* can tell us the important features. With the linear

kernel, we can also use the absolute values in vector w to get the top features. From class, we

know w= where is the coefficients which can determine which vectors are support vectors (if are not zero, the corresponding vectors are support vectors), is the training label, and . *w* sums up for all the support vectors. tune.linear$best.model$coefs can provide the corresponding coefficients times the training labels, which are inside *w*. Meanwhile, tune.linear$best.model$SV gives us the support vectors, which are inside *w.* When we multiply them, we can get *w.* Moreover*,* tune.linear$best.model$coefs is 261\*1 matrix and tune.linear$best.model$SV is 261\* 800 matrix. To compute *w*, I first transpose tune.linear$best.model$coefs to be 1\*261 matrix then do the matrix multiplication. I have usedt(tune.linear$best.model$coefs) %\*% tune.linear$best.model$SV to do it. The resulting w is a 1 \* 800 matrix. For linear kernel, it is just a linear regression. Because the input space has 800 dimensions, we would expect the feature space also has 800 dimensions, so the vector will be 1\*800 matrix. For each observation, v (1\*800 matrix) multiple wT (800 \* 1 matrix) will be a scaler and the sign of (wTv + b) can give us the classifier.

I then calculate the absolute values in w to determine the top 100 genes. The top 10 genes are showed below

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| GI\_14249383-S | GI\_10190745-S | GI\_12232388-S | GI\_10442821-S | GI\_13540514-S | GI\_13129143-S | GI\_10862697-S | GI\_14591917-S | GI\_13514821-A | GI\_14195610-S |

As we see here, the top features identified by linear kernel are very different than that of Random forest. They two are using different methods to determine the classifier and have different error rate, thus the top features identified are different.

For the polynomial kernel, it has the form . If then the kernel is homogenous and if it is not zero, then it is inhomogeneous. Inhomogeneous quadratic kernel is better for feature selection than homogeneous quadratic kernel. As we have learned from class, for k(x, x’) = (x • x’)2  and x has 2 dimensions, mapping φ(x) = () while for k(x, x’) = (x • x’+1)2 ; mapping φ(x) =(). Inhomogeneous quadratic kernel has two more dimensions than homogeneous quadratic kernel. These two more dimensions are the original dimensions in the input space. That is why the number of support vectors in inhomogeneous kernel is greater than that in homogenous kernel. Thus inhomogeneous quadratic kernel can better determine the importance of features.

With the two quadratic kernels, we cannot find correlated features (genes). Some people may think the absolute values of w for correlated features should be higher and we can use it to find the correlated features. However, that is not always the truth. For example, x has 3 dimensions (x, y, z), x and y are correlated with each other but they are much less important than z feature. For the homogenous kernel, the mapping φ(x) = (x2, y2,z2, xy,xz,yz). Since z is much more important than x and y, we would expect the absolute value of w for x2, y2 , xy are small while that for z2, xz and yz are big. If we use the absolute values of w to determine the correlated features, we would determine x and z are correlated and y is also correlated with z. However, from the assumption, we know that the reason for xz and yz have big absolute values of w is z is much more important than x and y. The same explanation can be applied to inhomogeneous quadratic kernel.

**Association Rule (AR) mining**

The training data with missing data imputed were used here. However, they were separated into two dataset, one for AD case data and the other one for control data. Then in one dataset (for AD or controls), for each feature (gene), order function in R was used to find the 10% people whose expressions of that gene are among the most abundant. I then turn the gene expression matrix of the given data into a new 1/0 matrix, where (i, j) = 1 if the i-th person has the expression of the j-th gene (item) among the top 10% highest expression levels, or 0 otherwise. The converted data for the first 4 genes for the case data and the total expressed genes (entries with 1 in the matrix) for WGAAD1, WGAAD2, WGAAD3, and WGAAD4 are showed below.

GI\_10047091-S GI\_10047093-S GI\_10047103-S GI\_10047133-A

1 WGAAD1 0 0 0 0

2 WGAAD2 0 0 0 0

3 WGAAD3 1 1 0 0

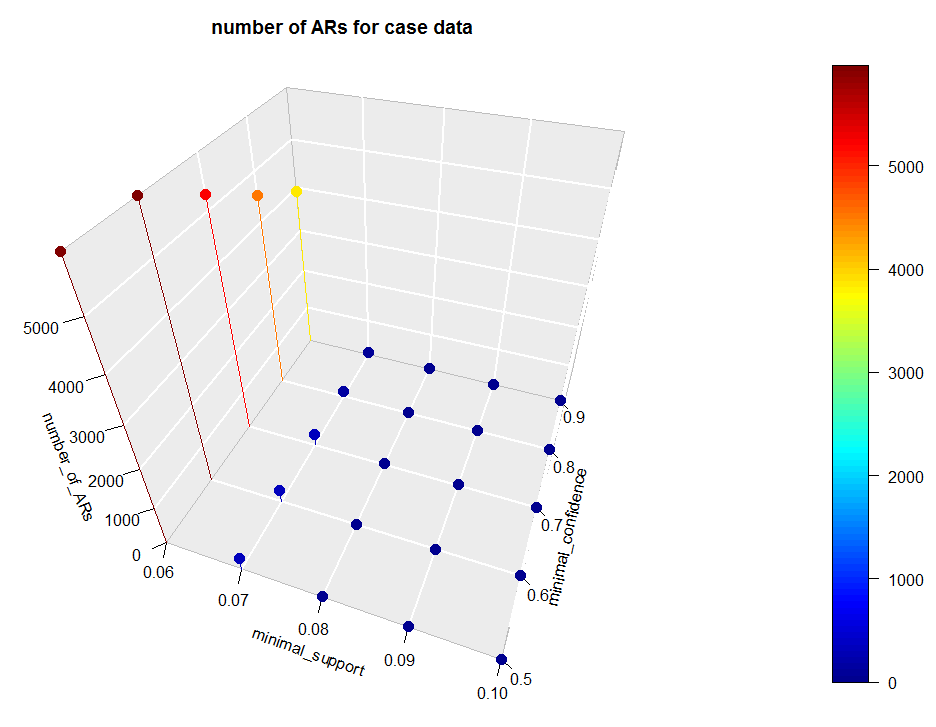
4 WGAAD4 0 0 0 0

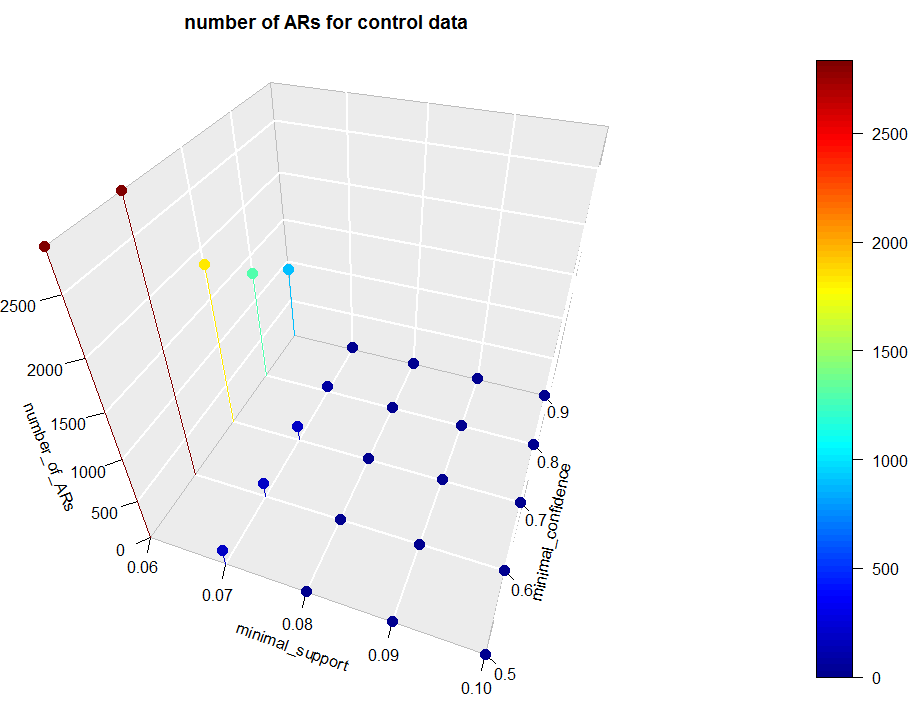
The converted data were first changed to be matrixes and then converted to "transactions" class using as function in R. This format can be used in arules package in R to find the frequent itemsets and association rules. Frequent itemsets were found using eclat function. The eclat() takes in a transactions object and gives the most frequent items in the data based the support you provide to the “supp” argument. The maxlen defines the maximum number of items in each itemset of frequent items, which were set to be 800. ARs were found using apriori function. It takes in the minimal support and minimal confidence in the parameter argument and gives the association rules. By default, Apriori only creates rules with one item in the right hand side(RHS) because the minlen parameter is 1 by default. I have used the default minlen value. In the first place, association rule mining usually produces too many rules even if one confines oneself to rules with only one item in the consequent. So why should one make the situation worse by allowing more than one item in the consequent? Secondly, I do not know any application in which rules with more than one item in the consequent are of any real use. The reason, in my opinion, is that such more complex rules add almost nothing to the insights about the data set. (Reference: http://www.borgelt.net/doc/apriori/apriori.html). Thus, I used the default value and only consider rules with one item in the RHS. One thing needs to be noted is if the minimum support is chosen too low for the dataset, then the algorithm will try to create an extremely large set of rules. This will result in very long run time and eventually the process will run out of memory. To prevent this, the default maximal length of rules is restricted to 10 items (via the parameter element maxlen=10) and the time for checking subsets is limited to 5 seconds (via maxtime=5) (Reference: Arules package documents about Apriori function). The total numbers of frequent itemsets (at least two items inside) and ARs discovered (in separate figures) as a function of the min support and min confidence for control and case data are listed and plotted as follows.

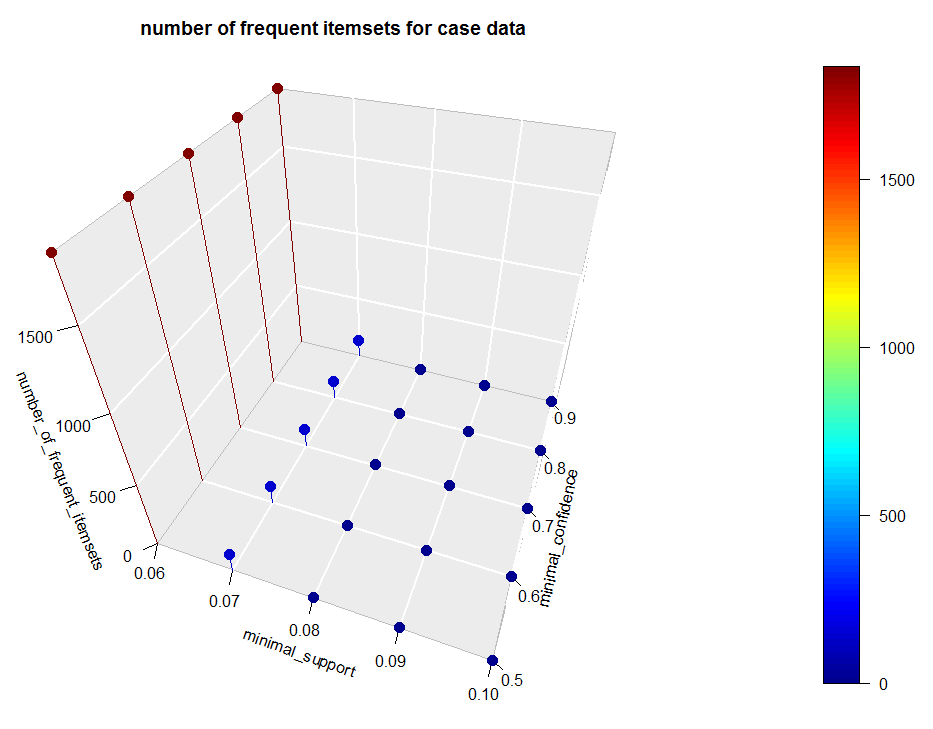
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| minimal support | minimal confidence | number of ARs for control | minimal support | minimal confidence | number of ARs for case |
| 0.06 | 0.5 | 2837 | 0.06 | 0.5 | 5977 |
| 0.06 | 0.6 | 2837 | 0.06 | 0.6 | 5977 |
| 0.06 | 0.7 | 1835 | 0.06 | 0.7 | 5277 |
| 0.06 | 0.8 | 1299 | 0.06 | 0.8 | 4548 |
| 0.06 | 0.9 | 903 | 0.06 | 0.9 | 3900 |
| 0.07 | 0.5 | 187 | 0.07 | 0.5 | 312 |
| 0.07 | 0.6 | 187 | 0.07 | 0.6 | 312 |
| 0.07 | 0.7 | 187 | 0.07 | 0.7 | 312 |
| 0.07 | 0.8 | 63 | 0.07 | 0.8 | 150 |
| 0.07 | 0.9 | 20 | 0.07 | 0.9 | 70 |
| 0.08 | 0.5 | 8 | 0.08 | 0.5 | 8 |
| 0.08 | 0.6 | 8 | 0.08 | 0.6 | 8 |
| 0.08 | 0.7 | 8 | 0.08 | 0.7 | 8 |
| 0.08 | 0.8 | 8 | 0.08 | 0.8 | 8 |
| 0.08 | 0.9 | 2 | 0.08 | 0.9 | 0 |
| 0.09 | 0.5 | 2 | 0.09 | 0.5 | 0 |
| 0.09 | 0.6 | 2 | 0.09 | 0.6 | 0 |
| 0.09 | 0.7 | 2 | 0.09 | 0.7 | 0 |
| 0.09 | 0.8 | 2 | 0.09 | 0.8 | 0 |
| 0.09 | 0.9 | 2 | 0.09 | 0.9 | 0 |
| 0.1 | 0.5 | 0 | 0.1 | 0.5 | 0 |
| 0.1 | 0.6 | 0 | 0.1 | 0.6 | 0 |
| 0.1 | 0.7 | 0 | 0.1 | 0.7 | 0 |
| 0.1 | 0.8 | 0 | 0.1 | 0.8 | 0 |
| 0.1 | 0.9 | 0 | 0.1 | 0.9 | 0 |

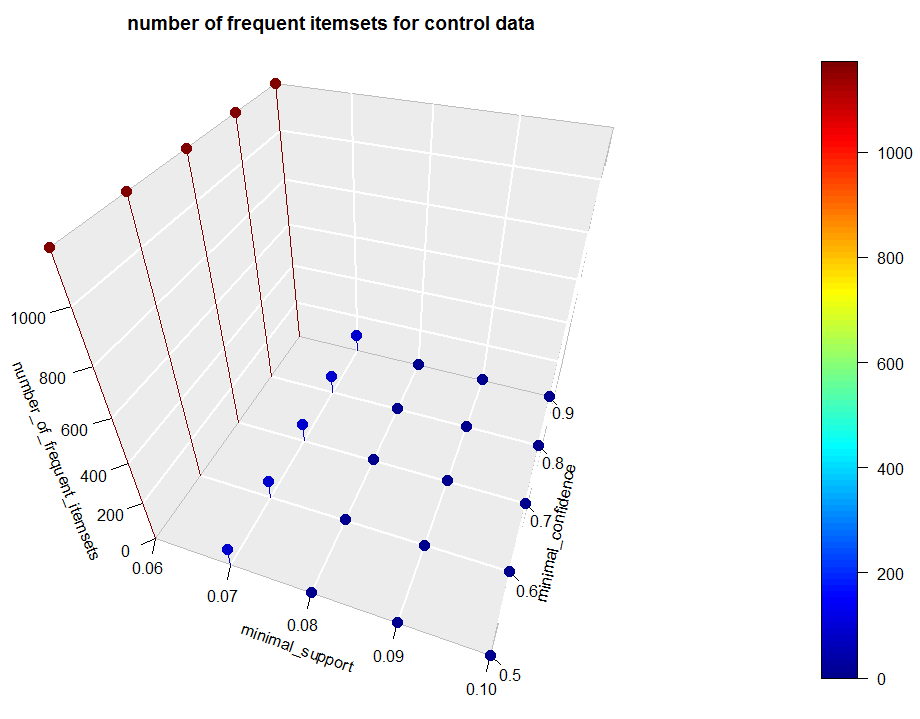
|  |  |  |  |
| --- | --- | --- | --- |
| minimal support | number of frequent items for control data | minimal support | number of frequent items for case data |
| 0.06 | 1173 | 0.06 | 1836 |
| 0.07 | 90 | 0.07 | 142 |
| 0.08 | 4 | 0.08 | 4 |
| 0.09 | 1 | 0.09 | 0 |
| 0.1 | 0 | 0.1 | 0 |

3D plots, for AD and controls separately, showing the total number of frequent itemsets and ARs discovered as a function of the min support and min confidence are as follows. I want to mention that the total number of frequent itemsets is not a function of min confidence because the definition of frequent itemsets is not related to min confidence.

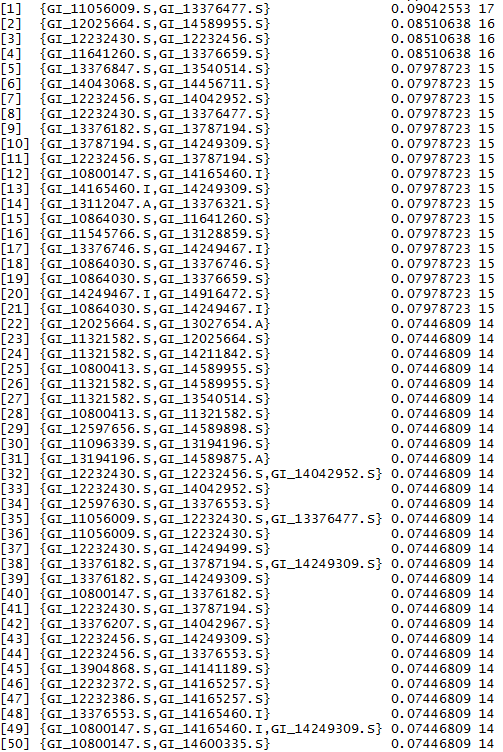




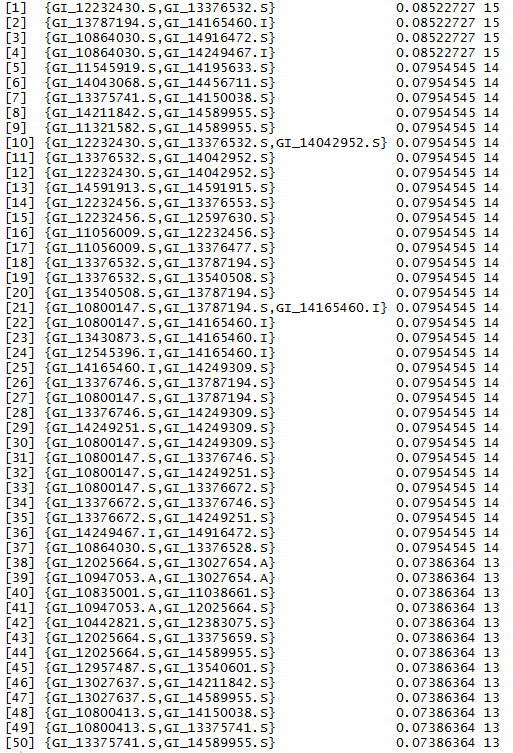




The frequent itemsets with supports greater than 5% were sorted from the highest frequency to the lowest by sort function in R with the arguments “by=count” and decreasing = TRUE. Since the top 800 frequent items contain only one item, the 801th to 851th frequent items will be the top 50 most frequent itemsets containing at least two items. They can be seen using inspect function. For control data, the results are as follows.

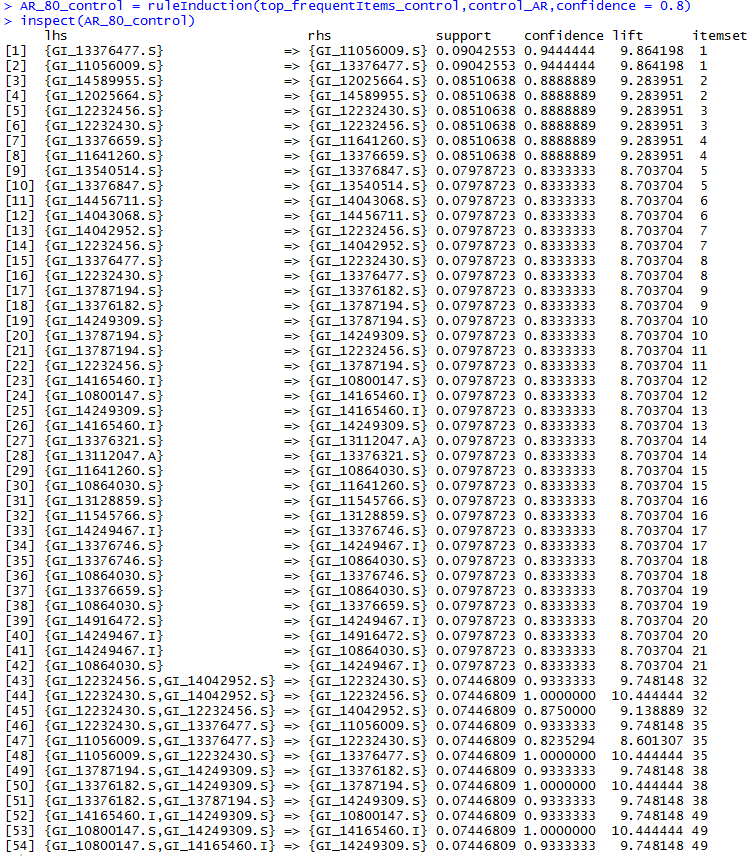


For the case data, the 50 most frequent itemsets containing at least two items with minimal support 5% are showed below

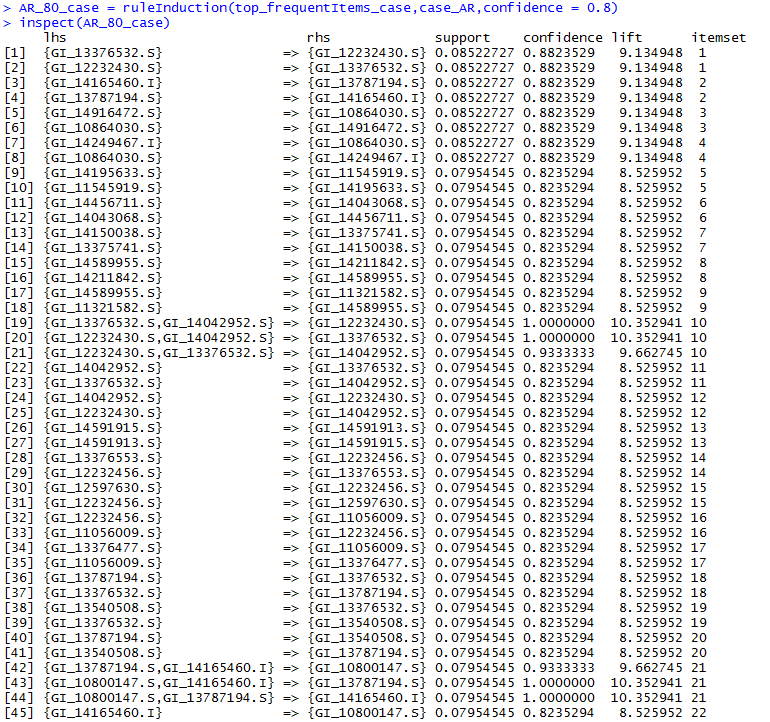


ARs (confidence > 80%) from the top 50 most frequent itemsets with support > 5%) were found using ruleInduction function in R. This function induces all rules which can be generated by the given set of itemsets from a transactions dataset. It takes in three parameters: itemsets from which rules will be induced; the transaction dataset used to mine the itemsets and a numeric value giving the minimum confidence for the rules. The top 50 most frequent itemsets with support > 5% which were shown above were provided as the itemsets used in ruleInduction function. The transaction dataset are the original dataset from the AD cases and normal controls. Finally, 0.8 was set for the minimum confidence.

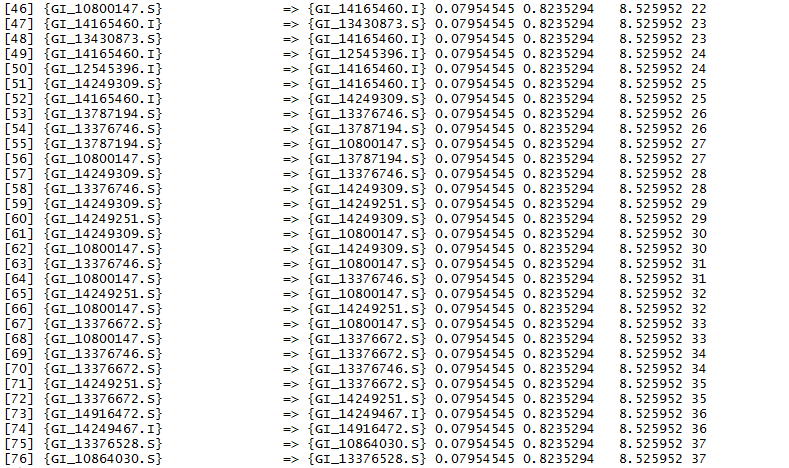
The resulting ARs for control data are showed below.



For case dataset, the resulting ARs (confidence > 80%) from the top 50 most frequent itemsets with support > 5%) were listed below.



continues in next page



The meaning of an AR (A=>B) here is how often we saw A and B together and how often we observed that if we saw A, we would expect to see B as well.

The meaning of an AR (A => B) implies that A and B are correlated. And generally speaking, A will be the independent feature while B is dependent feature. However, we can’t trust all the ARs. For example, as discussed in class for the examples of computer games and video games, although buy(game)=> buy(video) is an AR, they two are actually negatively correlated. To solve this problem, we can use “lift” function to check the ARs. Lift is the factor by which, the co-occurence of A and B exceeds the expected probability of A and B co-occuring, had they been independent. It is calculated as follows:

So, higher the lift, higher the chance of A and B occurring together. The lift values for all the ARs are showed above, all of them are much bigger than 1. This suggests that we can trust the above ARs. One more situation we need to pay attention. If there are ARs (A => B) and (B => A), neither A nor B itemsets are independent features. Similary, if there are ARs({A,B} => C), ( C => A), and ( C => B), then neither A, B nor {A,B} itemsets are independent features. However, if there are only two ARs ({A,B} => C) and ( C => A)), C ≠> B, then I think {A,B} is still the independent feature while C is the associated redundant feature. In this case, {A, B} is a new feature and it is not the same as A or B.

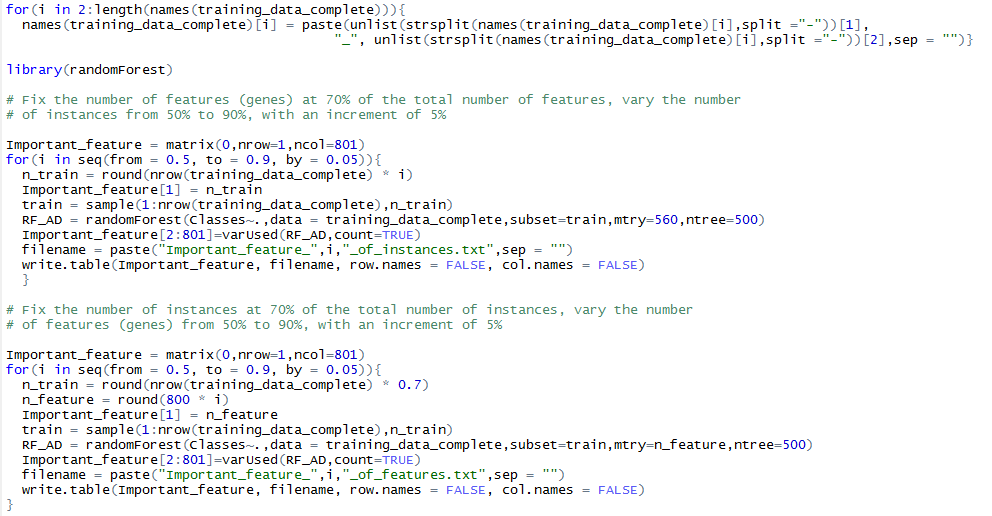
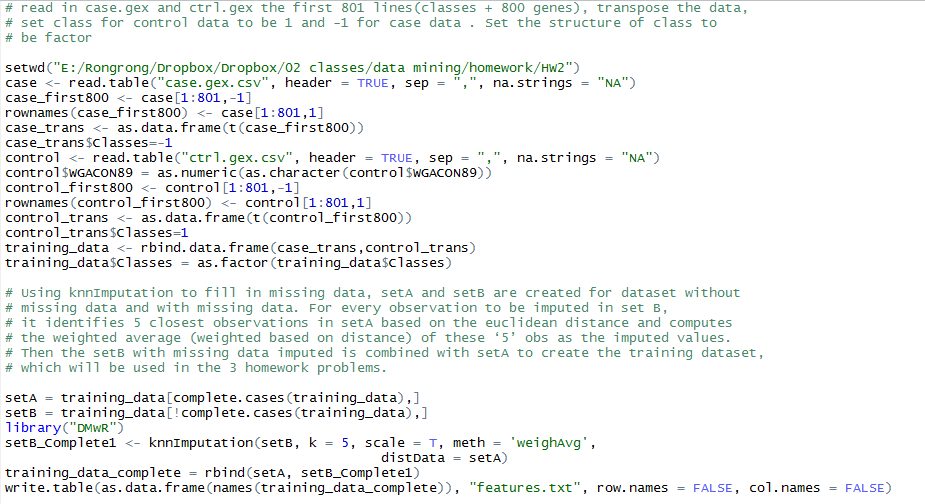
Apply my strategy to the results of 2) above (i.e., ARs (confidence > 80%) from the top 50 most frequent itemsets with support > 5%), for the control data, the independent features and their associated redundant features (genes) are listed below.

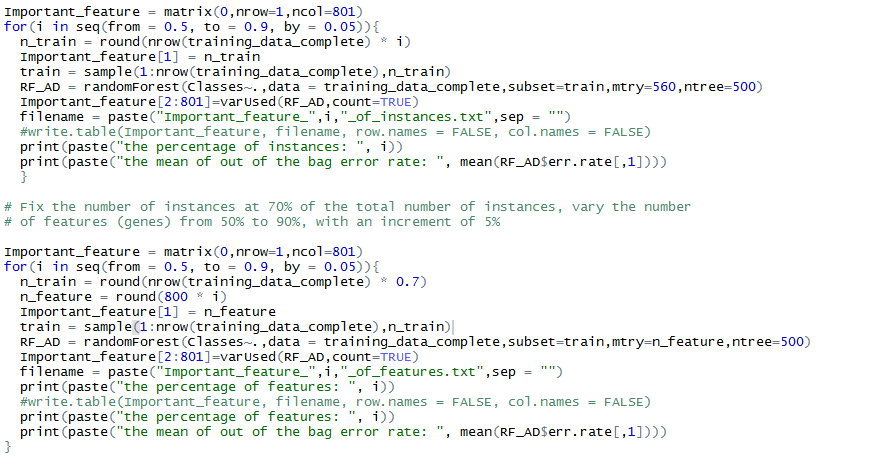
|  |  |
| --- | --- |
| Independent features | Associated redundant features (genes) |
| {GI\_12232456.S,GI\_14042952.S} | {GI\_12232430.S} |
| {GI\_12232430.S,GI\_12232456.S} | {GI\_14042952.S} |
| {GI\_12232430.S,GI\_13376477.S} | {GI\_11056009.S} |
| {GI\_11056009.S,GI\_13376477.S} | {GI\_12232430.S} |
| {GI\_13787194.S,GI\_14249309.S} | {GI\_13376182.S} |
| {GI\_13376182.S,GI\_13787194.S} | {GI\_14249309.S} |
| {GI\_14165460.I,GI\_14249309.S} | {GI\_10800147.S} |
| {GI\_10800147.S,GI\_14165460.I} | {GI\_14249309.S} |

For case data, I didn’t find any independent features.

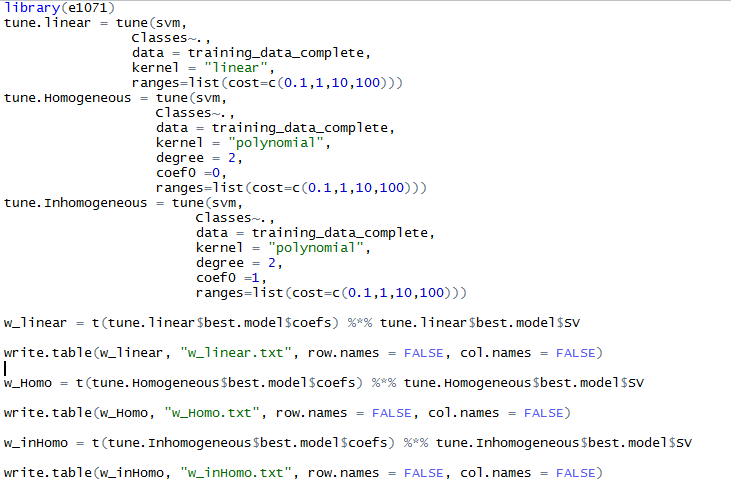
Appendix (Code)

P1





P2



P3

