Applying forestRK Package To The Soybean Dataset

Hyunjin Cho

2019-07-16

```
# set seed
set.seed(3672)

library(forestRK)
library(mlbench)
```

In this vignette, we demonstrate the implementation of **forestRK** functions to a dataset other than the **iris** dataset.

For the demonstration, we will work with the **Soybean** dataset from the package **mlbench**. The **Soybean** dataset contains 26 nominal (categorical) attributes of 683 different soybeans for its input; the output of the dataset is the 19 different class labels of the soybeans.

```
data(Soybean)
dim(Soybean)
```

```
## [1] 683  36
```

```
levels(Soybean$Class)
```

```
##  [1] "2-4-d-injury"               "alternarialeaf-spot"
##  [3] "anthracnose"                "bacterial-blight"
##  [5] "bacterial-pustule"          "brown-spot"
##  [7] "brown-stem-rot"             "charcoal-rot"
##  [9] "cyst-nematode"              "diaporthe-pod-&-stem-blight"
## [11] "diaporthe-stem-canker"      "downy-mildew"
## [13] "frog-eye-leaf-spot"         "herbicide-injury"
## [15] "phyllosticta-leaf-spot"     "phytophthora-rot"
## [17] "powdery-mildew"             "purple-seed-stain"
## [19] "rhizoctonia-root-rot"
```

```
head(Soybean)
```

```
##                  Class date plant.stand precip temp hail crop.hist
## 1 diaporthe-stem-canker    6           0      2    1    0         1
## 2 diaporthe-stem-canker    4           0      2    1    0         2
## 3 diaporthe-stem-canker    3           0      2    1    0         1
## 4 diaporthe-stem-canker    3           0      2    1    0         1
## 5 diaporthe-stem-canker    6           0      2    1    0         2
## 6 diaporthe-stem-canker    5           0      2    1    0         3
##   area.dam sever seed.tmt germ plant.growth leaves leaf.halo leaf.marg
## 1        1     1        0    0           1      1         0         2
## 2        0     2        1    1           1      1         0         2
## 3        0     2        1    2           1      1         0         2
## 4        0     2        0    1           1      1         0         2
## 5        0     1        0    2           1      1         0         2
## 6        0     1        0    1           1      1         0         2
##   leaf.size leaf.shread leaf.malf leaf.mild stem lodging stem.cankers
## 1         2           0         0         0    1       1            3
## 2         2           0         0         0    1       0            3
## 3         2           0         0         0    1       0            3
## 4         2           0         0         0    1       0            3
## 5         2           0         0         0    1       0            3
## 6         2           0         0         0    1       0            3
##   canker.lesion fruiting.bodies ext.decay mycelium int.discolor sclerotia
## 1             1               1         1        0            0         0
## 2             1               1         1        0            0         0
## 3             0               1         1        0            0         0
## 4             0               1         1        0            0         0
## 5             1               1         1        0            0         0
## 6             0               1         1        0            0         0
##   fruit.pods fruit.spots seed mold.growth seed.discolor seed.size
```

```
## 1          0           4    0           0             0         0
## 2          0           4    0           0             0         0
## 3          0           4    0           0             0         0
## 4          0           4    0           0             0         0
## 5          0           4    0           0             0         0
## 6          0           4    0           0             0         0
##   shriveling roots
## 1          0     0
## 2          0     0
## 3          0     0
## 4          0     0
## 5          0     0
## 6          0     0
```

```
summary(Soybean)
```

```
##                 Class          date     plant.stand  precip      temp
##  brown-spot         : 92   5      :149   0   :354    0   : 74   0   : 80
##  alternarialeaf-spot: 91   4      :131   1   :293    1   :112   1   :374
##  frog-eye-leaf-spot : 91   3      :118   NA's: 36    2   :459   2   :199
##  phytophthora-rot   : 88   2      : 93               NA's: 38   NA's: 30
##  anthracnose        : 44   6      : 90
##  brown-stem-rot     : 44   (Other):101
##  (Other)            :233   NA's   :  1
##    hail     crop.hist  area.dam    sever     seed.tmt     germ
##  0   :435   0   : 65   0   :123   0   :195   0   :305   0   :165
##  1   :127   1   :165   1   :227   1   :322   1   :222   1   :213
##  NA's:121   2   :219   2   :145   2   : 45   2   : 35   2   :193
##             3   :218   3   :187   NA's:121   NA's:121   NA's:112
##             NA's: 16   NA's:  1
##
##
##  plant.growth leaves  leaf.halo  leaf.marg  leaf.size  leaf.shread
##  0   :441     0: 77   0   :221   0   :357   0   : 51   0   :487
##  1   :226     1:606   1   : 36   1   : 21   1   :327   1   : 96
##  NA's: 16             2   :342   2   :221   2   :221   NA's:100
##                       NA's: 84   NA's: 84   NA's: 84
##
##
##
##  leaf.malf  leaf.mild    stem     lodging    stem.cankers canker.lesion
##  0   :554   0   :535   0   :296   0   :520   0   :379     0   :320
##  1   : 45   1   : 20   1   :371   1   : 42   1   : 39     1   : 83
##  NA's: 84   2   : 20   NA's: 16   NA's:121   2   : 36     2   :177
##             NA's:108                         3   :191     3   : 65
##                                              NA's: 38     NA's: 38
##
##
##  fruiting.bodies ext.decay  mycelium   int.discolor sclerotia  fruit.pods
##  0   :473          0   :497   0   :639   0   :581     0   :625   0   :407
##  1   :104          1   :135   1   :  6   1   : 44     1   : 20   1   :130
##  NA's:106          2   : 13   NA's: 38   2   : 20     NA's: 38   2   : 14
##                    NA's: 38              NA's: 38                3   : 48
##                                                                  NA's: 84
##
##
##  fruit.spots   seed     mold.growth seed.discolor seed.size  shriveling
##  0   :345    0   :476   0   :524    0   :513      0   :532   0   :539
##  1   : 75    1   :115   1   : 67    1   : 64      1   : 59   1   : 38
##  2   : 57    NA's: 92   NA's: 92    NA's:106      NA's: 92   NA's:106
##  4   :100
##  NA's:106
##
##
##   roots
##  0   :551
##  1   : 86
##  2   : 15
##  NA's: 31
##
##
##
##
```

0. Forest-RK

The new Forest-RK algorithm is from the paper: **"Forest-RK: A New Random Forest Induction Method" by Simon Bernard, Laurent Heutte, Sebastien Adam, 4th International Conference on Intelligent Computing (ICIC), Sep 2008, Shanghai, China, pp.430-437** .

Please consult the paper above for detailed information about the new Forest-RK algorithm.

To briefly summarize, Forest-RK algorithm is identical to the classical random forest model except that in Forest-RK model, we treat the parameter K, the number of covariates that we consider for each split, to be random.

1. Data Cleaning

The very first thing that we want to do before implementing **forestRK** functions is the proper data cleaning. The functions **y.organizer** and **x.organizer** were made for these data cleaning tasks. To be more specific, we need to follow the following data cleaning procedures:

1. Remove all NAs and NaNs from the original dataset, as **forestRK** functions will throw an error if the data contains any missing records.
2. Seperate the dataset into a chunk that stores covariates of all training and test observations (**X**) and a vector that stores class types of the training observations (**y**).
3. Numericize the data frame **X** by applying the **x.organizer**, and seperate **X** into a training and a test set as needed.
4. Numericize the vector **y** from the training set by applying the **y.organizer** function. The attribute **y.new** of the **y.organizer** output contains the numericized class types of the training observations.

These steps are outlined in the example code below; note that we are going to perform a Binary Encoding onto the categorical covariates of **Soybean** dataset. When **K**, the number of covariates that we consider at each split is fixed, **Binary Encoding** is known to be effective for the dataset that has categorical features with cardinality greater than 1000, and when the cardinality of categorical feature is less than 1000, **Numeric Encoding** is known to perform better than the **Binary Encoding**. For more information about Binary and Numeric Encoding, please visit the website:

https://medium.com/data-design/visiting-categorical-features-and-encoding-in-decision-trees-53400fa65931

The cardinality of **Soybean** covariates are definately less than 1000, but we implement both **Numeric** and **Binary Encoding** just to give an example.

```
# Step 1: Remove all NA's and NaN's from the original dataset
Soybean <- na.omit(Soybean)

# Step 2: Seperate the dataset into a chunk that stores covariates of both
# training and test observations X and a vector y that stores class types of the
# training observations
vec <- seq.int(1,562, by=3) # indices of the training observations
X <- Soybean[,2:36]
y <- Soybean[vec,1]

# Step 3: Numericize the data frame X by applying the x.organizer function
# and split X into a training and a test set
X1 <- x.organizer(X, encoding = "bin") # Binary Encoding applied
X2 <- x.organizer(X, encoding = "num") # Numeric Encoding applied

## train and test set from the Binary Encoding
x.train1 <- X1[vec,]
x.test1 <- X1[-vec,]

## train and test set from the Numeric Encoding
x.train2 <- X2[vec,]
x.test2 <- X2[-vec,]

# Step 4: Numericize the vector y by applying the y.organizer function
y.train <- y.organizer(y)$y.new # a vector storing the numericized class type
y.factor.levels <- y.organizer(y)$y.factor.levels
```

The lines R code below extracted directly from the **x.organizer** function implements the **Binary Encoding**:

```
    ## Numercize the data frame of covariates via Binary Encoding

    if(encoding == "bin"){
        ## x.dat is the data frame containing the covariates of both training and test observations,
        ## x.dat is one of the required input to call the x.organizer function
        x.new <- rep(0, dim(x.dat)[1])
        n.cov <- dim(x.dat)[2]
        for(j in 1:n.cov){
                if(!(is.factor(x.dat[,j]) || is.character(x.dat[,j]))){
                    x.new <- data.frame(x.new, x.dat[,j])
                    colnames(x.new)[dim(x.new)[2]] <- colnames(x.dat)[j]
                } # if all of the covariates in the dataset are not categorical,
```

```
            # then the function will simply return the original numeric dataset.

              else if(is.factor(x.dat[,j]) || is.character(x.dat[,j])){
                    x.bin  <-
data.frame(matrix(as.integer(intToBits(as.integer(as.factor(x.dat[,j])))), ncol = 32,
                                          nrow = length(x.dat[,j]),byrow = TRUE)[
,1:ceiling(log(length(unique(x.dat[,j])) + 1)/log(2))])
                    colnames(x.bin) <- paste(colnames(x.dat)[j], c(1:(dim(x.bin)[2])))
                    x.new <- data.frame(x.new, x.bin)
              } # end of else if(is.factor(x.dat[,j]) || is.character(x.dat[,j]))
      } # end of for(j in 1:n.cov)

      x.new <- x.new[,-1]
```

The chunk of R code below from the function implements **Numeric Encoding**:

```
  else if(encoding == "num"){
      ## Convert the original data frame of covariates into a numericized one via Numeric Encoding
      ## Numeric Encoding simply assigns an arbitrary number to each class category
      x.new <- data.frame(data.matrix(x.dat))
  }
```

2. Building A rkTree

Once the data cleaning has been performed successfully, we can start implementing **forestRK** functions to construct trees, forests, and related plots.

The function **construct.treeRK** builds a single rkTree based on the training data, the minimum number of observations that the user want each end node of his rkTree to contain (default is 5), and the type of criteria that the user would like to use (Entropy vs. Gini Index).

User can specify the type of splitting criteria that he or she is going to use by adjusting the boolean argument **entropy** in the function call. If **entropy** is set to **TRUE**, then the function will use the Entropy as the splitting criteria; if **entropy** is set to **FALSE**, then the function will use the Gini Index as the splitting criteria.

The R code below constructs a rkTree from the Binary-Encoded training data by using the Gini Index as the splitting criteria, and by using 6 as the minimum number of observations that its end node should contain.

```
tree.gini <- construct.treeRK(x.train1, y.train, min.num.obs.end.node.tree = 6,
                              entropy = FALSE)
```

We also make a rkTree from the Numeric-Encoded training data by using the Entropy as the splitting criteria, and by using 5 (default) as the minimum number of observations that its end node should contain.

```
## default for entropy is TRUE
## default for min.num.obs.end.node.tree is 5
tree.entropy <- construct.treeRK(x.train2, y.train)
```

The formula that the function uses for computing **Entropy** of a node is:

$$Entropy = \sum_{i=1}^{C} - p_i \dot{\log_2}(p_i)$$

, where p_i =(proportion of observations from the node that falls into class i)

The formula that the function uses for computing **Gini Index** of a node is:

$$Gini = 1 - \sum_{i=1}^{C} p_i^2$$

, where p_i =(proportion of observations from the node that falls into class i)

The formula that the function uses for computing the value of the splitting criteria after a certain split is:

$$E(T, X) = \sum_{c \in X} P_c E_c$$

, where P_c =(proportion of number of observations contained in the child node ‘c’) E_c =(value of the splitting criteria of the child node ‘c’)

If the number of observations included in the original training set is already less than the minimum number of observations that the user specified each end node to contain, then the **construct.treeRK** function will not perform any split on the original training dataset.

The **construct.treeRK** function will not perform any extra split on a child node if:

1. the number of observations contained in the child node is less than the minimum number of observations that the user wants each end node to contain;
2. the value of the splitting criteria of the node in question before a split is already 0 (i.e. the node is perfectly pure); OR
3. the value of the splitting criteria of the node in question after an optimal split is greater than the value of the splitting criteria before the split.

One of the most useful output produced by **construct.treeRK** function is the hierarchical flag. The hirarchical flag of a rktree (**construct.treeRK()$flag**) is constructed in the following manner:

1. the first entry of the flag, “r” denotes for “root” or the original training dataset provided by the user;
2. the subsequent strings of the flag is constructed in the way that last “x” denotes for the left child node of the node represented by the series of characters that are before the last “x”, and the last “y” denotes for the right child node of the node represented by the series of characters that are before the last “y”. For example, if a node represented as “rx” was split, its left child node would be represented as “rxx” and its right child node would be represented as “rxy” somewhere down the list of the hierarchical flag.

Below is the example of hierarchical flag produced by **construct.treeRK** function:

```
tree.gini$flag
```

```
##       [,1]
##  [1,] "r"
##  [2,] "rx"
##  [3,] "ry"
##  [4,] "rxx"
##  [5,] "rxy"
##  [6,] "ryx"
##  [7,] "ryy"
##  [8,] "rxxx"
##  [9,] "rxxy"
## [10,] "ryxx"
## [11,] "ryxy"
## [12,] "ryyx"
## [13,] "ryyy"
## [14,] "ryxyx"
## [15,] "ryxyy"
## [16,] "ryyxx"
## [17,] "ryyxy"
## [18,] "ryyyx"
## [19,] "ryyyy"
## [20,] "ryyyyx"
## [21,] "ryyyyy"
```

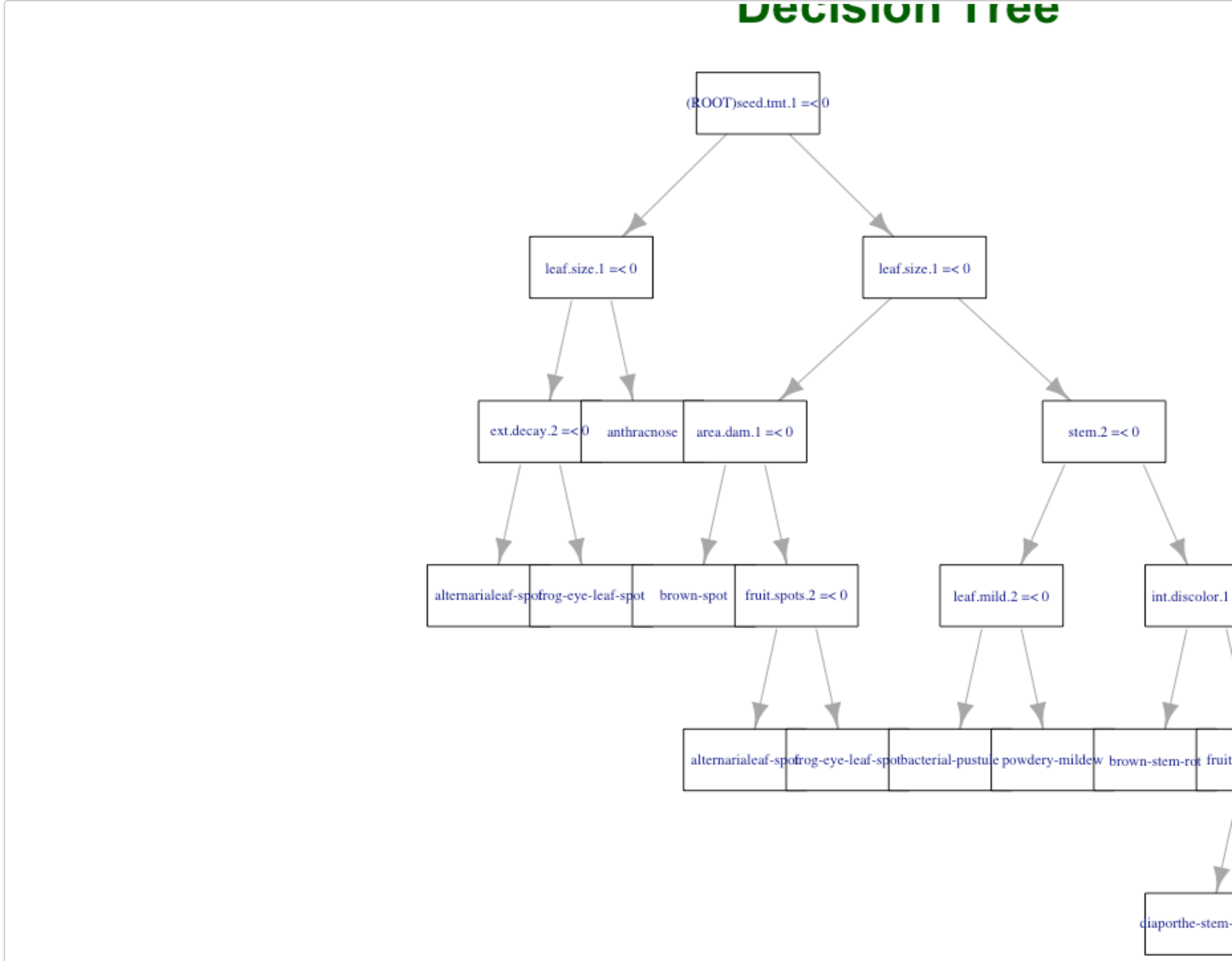
By closely examining the structure of the hierarchical flag, we can do many useful things such as identifying where to place an edge when drawing a plot of the rkTree model that we constructed, etc (discussed further down the road).

For more details about how to interpret **construct.treeRK** output, please see the **construct.treeRK** section of the **forestRK** documentation.

3. Plot The rkTree

Once a rkTree is built from the training data, the user can be interested in plotting the tree. The **draw.treeRK** function plots the **igraph** diagram of the rkTree.

```
## plotting the tree.gini rkTree obtained via construct.treeRK function
draw.treeRK(tree.gini, y.factor.levels, font="Times", node.colour = "white",
            text.colour = "dark blue", text.size = 0.6, tree.vertex.size = 30,
            tree.title = "Decision Tree", title.colour = "dark green")
```




```
## NULL
```

The plot should be interpreted as the following:

The rectangular nodes (or vertices) that contain " **=<** " symbol are used to describe the splitting criteria applied to that very node while constructing the **rktree**; for example, if a rectangle (which represents an individual node) has the label " **date.1 =< 1.6** ", this would indicate that this node was split into a chunk that contains observations with the value of the covariate **date.1** is less than or equal to 1.6 and a chunk that contains observations with their **date.1** values greater than 1.6, while the rkTree was built.

Any other rectangular nodes (or vertices) that do not contain the " **=<** " symbol indicate that we have reached an end node, and the text displayed in such node is the actual name of the class type that the rkTree model assigns to the observations belonging to that node; for example, the rectangle (again, a node) with the label " **diaporthe-stem-canker** " indicates that the rkTree in question assigns the class type " **diaporthe-stem-canker** " to all observations that belong to that particular node.

The hardest part of making this function is to determine where the edges should be placed in the rkTree diagram. The basic algorithm is the following:

if the substring of the current flag before the very last character matches exactly with the entire string of the one of the previous flags, and if the length of the current flag is greater than the length of the previous flag by 1 character, this means that when we draw the diagram of this rkTree, we should place an edge between the node that is represented by the current flag and the node that is represented by the previous flag. This makes senses since those criteria that I just described wouldn't be satisfied unless the node represented by the current flag is a direct child node of the node that is represented by the previous flag.

The R code below, which I extracted from the **draw.treeRK** function implements this edge-placing algorithm:

```
  ## Make edges
  for(i in 1:(length(f)-1)){
      branch <- f[i]

      for (j in (i+1):(length(f))){
          ch <- f[j]
          if ((substr(ch,1,nchar(branch))==branch) && (nchar(ch)==(nchar(branch)+1))){ed <-
c(ed,i,j)}
      }
  }
```

```
    # creating empty graph (i.e. a graph with no edges)
    i.graph <- make_empty_graph(length(unlist(tr$flag)), directed = T)
    # adding edges onto the empty graph
    i.graph <- add_edges(i.graph, ed)
```

4. Make Predictions Based On A rkTree

Users may want to make predictions on the test observations based on the rkTree that he or she constructed by using **construct.treeRK** function; the function **pred.treeRK** was made to perform this task.

```
## display numericized predicted class type for the first 5 observations (in the
## order of increasing original observation index number) from the test set with
## Binary Encoding
prediction.df <- pred.treeRK(X = x.test1, tree.gini)$prediction.df
prediction.df[1:5, dim(prediction.df)[2]]
```

```
## [1]  3  3 11 11 11
```

```
## display the list of hierarchical flag from the test set with Numeric Encoding
pred.treeRK(X = x.test2, tree.entropy)$flag.pred
```

```
##       [,1]
##  [1,] "r"
##  [2,] "rx"
##  [3,] "ry"
##  [4,] "rxx"
##  [5,] "rxy"
##  [6,] "ryx"
##  [7,] "ryy"
##  [8,] "rxxx"
##  [9,] "rxxy"
## [10,] "rxyx"
## [11,] "rxyy"
## [12,] "ryxx"
## [13,] "ryxy"
## [14,] "ryyx"
## [15,] "ryyy"
## [16,] "rxxxx"
## [17,] "rxxxy"
## [18,] "rxxyx"
## [19,] "rxxyy"
## [20,] "ryxxx"
## [21,] "ryxxy"
## [22,] "ryxyx"
## [23,] "ryxyy"
## [24,] "rxxxxx"
## [25,] "rxxxxy"
## [26,] "rxxxyx"
## [27,] "rxxxyy"
## [28,] "rxxyyx"
## [29,] "rxxyyy"
## [30,] "rxxxxyx"
## [31,] "rxxxxyy"
## [32,] "rxxxxyyx"
## [33,] "rxxxxyyy"
## [34,] "rxxxxyyxx"
## [35,] "rxxxxyyxy"
```

prediction.df a data frame of test observations their predicted class type that is returned by the **pred.treeRK** function. If **prediction.df** has **n** columns, the first **n-1** columns will contain the numericized covariates of the test observations, and the very last **n**-th column will contain the predicted numericized class type for each of those test observations. Users of this function may be interested in identifying the original name of the numericized predicted class type shown in the last column of data frame **prediction.df**. This can easily be done by extracting the attribute **y.factor.levels** from the **y.organizer** object. For example, if the data frame **prediction.df** indicates that the predicted class type of the 1st test observation is “2”, that means the actual name of the predicted class type for that 1st test observation is indicated as the 2nd element of the vector **y.organizer.object$y.factor.levels** that we can obtain during the data cleaning phase.

NOTE: At the end of the **pred.treeRK** function, **the test data points in prediction.df are re-ordered by the increasing original row index number (the integer row index of a data point from the original dataset that the user had right before splitting them into a training and a test set) of those test observations.** So if you shuffled the data before seperating them into a training and a test set, the order of the data points in which they are presented under the data frame **prediction.df** may not be same as the shuffled order of data points in your test set.

The **pred.treeRK** function makes a use of the list of hierarchical flags generated by the **construct.treeRK** function; the function uses the list of hierarchical flag as a guide to how it should split the test set to make predictions. The function **pred.treeRK** itself actually generates a list of hierarchical flag of its own as it splits the test set, and at the end of the function **pred.treeRK** tries to match the list of hierarchical flag it generated with the list of hierarchical flag from the **construct.treeRK** function. If the two flags match exactly, then it is a good sign since this would imply that the splitting on the test set was done in the manner consistent with how the training set was split when the rkTree in question was built. If there is any difference in the two flags, however, this is not a good sign since it would signal that the splitting on the test set has done in a different manner than how the splitting was done on the training set; if the mismatch occurs, the **pred.treeRK** function will stop and throw an error.

5. Generating A forestRK

But our inquiry shouldn't end with just examining a single rkTree; normally we would be interested in building a forest that is comprised of many trees to make predictions. The R code below generates **forestRK** models based on the training set, the number of bags (= number of bootstrap samples), the sample size for each bag, etc., that were provided by the user:

```
## make a forestRK model based on the training data with Binary Encoding and with Gini Index as the
  splitting criteria
## normally nbags and samp.size would be much larger than 30 and 50
forestRK.1 <- forestRK(x.train1, y.train, nbags = 30, samp.size = 50, entropy = FALSE)

## make a forestRK model based on the training data with Numeric Encoding and with Entropy as the
  splitting criteria
## make a forestRK model based on the training data with Binary Encoding and with Gini Index as the
  splitting criteria
## normally nbags and samp.size would be much larger than 30 and 50
forestRK.2 <- forestRK(x.train2, y.train, nbags = 30, samp.size = 50)
```

The basic mechanism behind the **forestRK** function is pretty simple: the function first generates bootstrap samples of the training dataset based on the paramemters that the user provided, and then it calls the function **construct.treeRK** in order to build a rkTree on each of those bootstrap samples, to form a bigger forest.

6. Making Predictions Based On A forestRK Model

Again, the main interest of users who use the **forestRK** package may be making predictions on the test observations based on a **forestRK** model. The function **pred.forestRK** is for making predictions on the test observations based on a forestRK algorithm. The R code shown below would make predictions on the test observation that the user provided (**x.test1**), based on the **forestRK** model that it builds on the training data (**x.train1** and **y.train1**) that the user specified:

```
# make predictions on the test set x.test1 based on the forestRK model constructed from
# x.train1 and y.train (recall: these are the training data with Binary Encoding)
# after using Gini Index as splitting criteria (entropy == FALSE)
pred.forest.rk1 <- pred.forestRK(x.test = x.test1, x.training = x.train1,
                                  y.training = y.train, nbags = 100,
                                 samp.size = 100, y.factor.levels = y.factor.levels,
                                 entropy = FALSE)

# make predictions on the test set x.test2 based on the forestRK model constructed from
# x.train2 and y.train (recall: these are the training data with Numeric Encoding)
# after using Entropy as splitting criteria (entropy == TRUE)
pred.forest.rk2 <- pred.forestRK(x.test = x.test2, x.training = x.train2,
                                 y.training = y.train, nbags = 100,
                                 samp.size = 100, y.factor.levels = y.factor.levels,
                                 entropy = TRUE)
```

The basic mechanism behind **pred.forestRK** function is the following: When the function is called, it calls **forestRK** function after passing the user-specified training data as an argument, in order to first generate the **forestRK** object. After that, the function uses **pred.treeRK** function to make predictions on the test observations based on each individual tree in the **forestRK** object. Once the individual prediction from each tree are obtained for all of the test observations, the function stores those individual predictions under a big dataframe. Once that data frame is complete, then the function collapses the results by the rule of the majority votes. For example, for the m-th observation from the test set, if the most frequently predicted class type for that m-th test observation by all of the rkTrees in the forest is class type 'A', then by the rule of the majority votes, the **pred.forestRK** function will assign class 'A' as the predicted class type for that m-th test observation based on the **forestRK** model.

When I was making this function, I had an option of directly using the output of the **forestRK** function as the required input for calling **pred.forestRK**; however, I decided to make the function **pred.forestRK** to generate a **forestRK** model internally based on the training data provided by the user, in order to reduce the number of functions that the user is required to use to make predictions.

NOTE: **pred.forestRK** outputs (**test.prediction.df.list**, , **pred.for.obs.forest.rk**, and **num.pred.for.obs.forest.rk**) are **generated after re-ordering test data points by the increasing original row index number (the integer row index of a data point from the original dataset that the user had right before splitting them into a training and a test set) of those test observations**. So if you shuffled the data

before seperating them into a training and a test set, the order of the data points in which they are presented in the **pred.forestRK** outputs may not be same as the shuffled order of data points in your test set.

7. Extracting Individual Tree(s) From The forestRK Object

The users may be interested in examining the structure of an individual tree(s) that is contained within the **forestRK** object. The functions **get.tree.forestRK** and **var.used.forestRK** help with this type of tasks. **get.tree.forestRK** is used to extract information about a particular tree(s) that is contained in the **forestRK**. The function **var.used.forestRK** is used to extract the list of names of the covariates that were used at each split while that particular rkTree was built. The R code shown below demonstrate the use of **get.tree.forestRK** and **var.used.forestRK** functions:

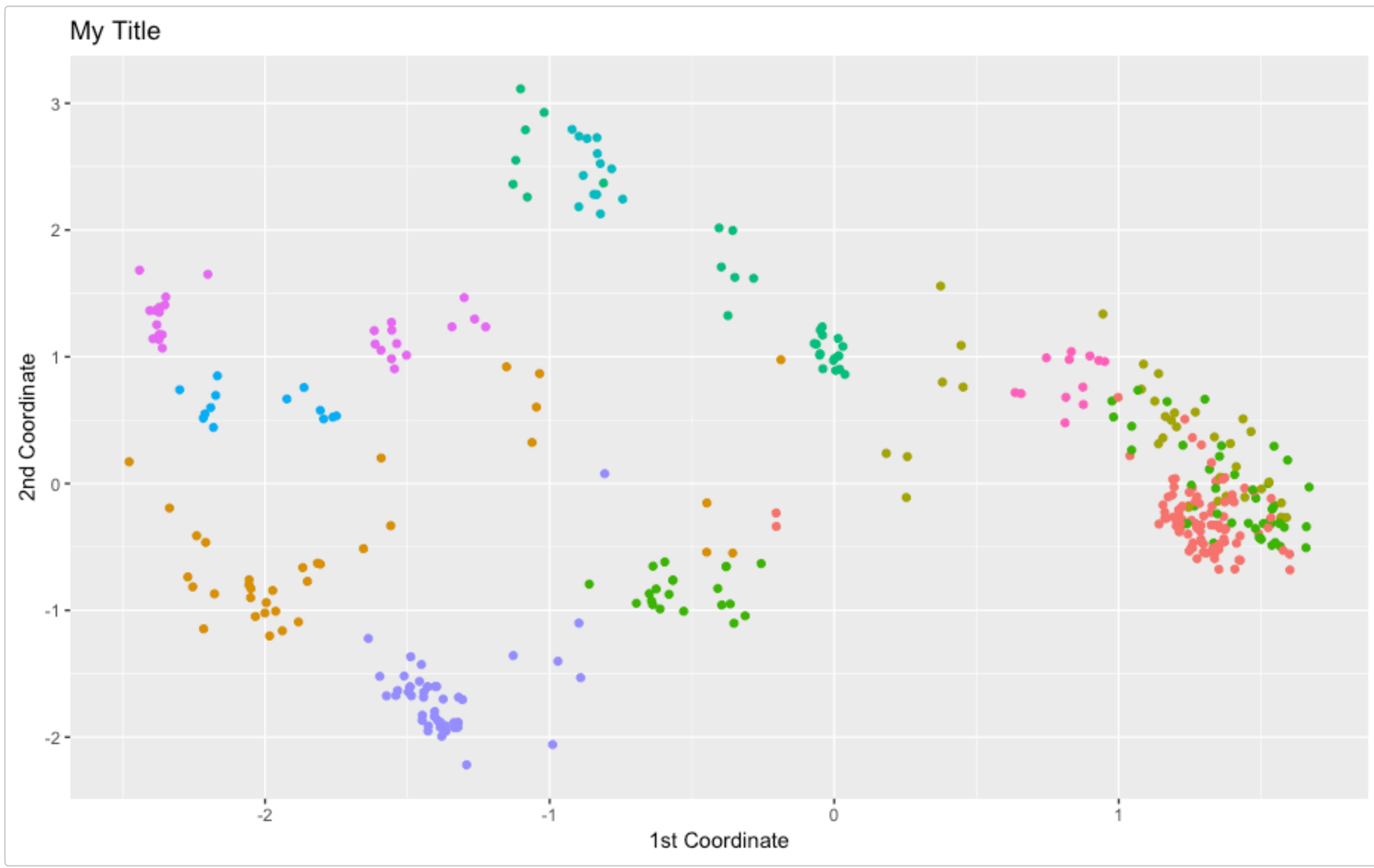
```
## get tree
tree.index.ex <- c(1,3,8)
# extract information about the 1st, 3rd, and 8th tree in the forestRK.1
get.tree <- get.tree.forestRK(forestRK.1, tree.index = tree.index.ex)
# display 8th tree in the forest (which is the third element of the vector 'tree.index.ex')
get.tree[["8"]]

## get the list of variable used for splitting
covariate.used.for.split.tree <- var.used.forestRK(forestRK.1, tree.index = c(4,5,6))
# get the list of names of covariates used for splitting to construct tree #6 in the forestRK.1
covariate.used.for.split.tree[["6"]]
```

The **var.used.forestRK** lists the actual name of the covariate used for a split (not their numericized ones), consistent to the exact order of the split; for instance, the 1st element of the vector **covariate.used.for.split.tree[[“6”]]** from the example above is the covariate on which the 1st split had occured while the 6th tree in the **forestRK.1** object was built.

8. Other Useful Functions From ForestRK

- **mds.plot.forestRK** function generates 2-dimensional Multi-Dimensional Scaling (MDS) plot of the test observations, and colour codes each test observation by their predicted class type. Already existing R functions **dist** and **cmdscale** were used in this function to compute the Multi- Dimensional Scales of the test data. The R code below demonstrates how to use this function:

```
## Generate 2-dimensional MDS plot of the test observations colour coded by the
## predictions stored under the object pred.forest.rk1
mds.plot.forestRK(pred.forest.rk1, plot.title = "My Title", xlab ="1st Coordinate",
                  ylab = "2nd Coordinate", colour.lab = "Predictions By The forestRK.1")
```



- **importance.forestRK** function calculates the Gini Importance (sometimes also known as Mean Decrease in Impurity) of each covariate that we consider in the **forestRK** model that the user provided, and lists the covariate names and values in the order of most important to the least important. The Gini Importance

algorithm is also used in 'scikit-learn'.

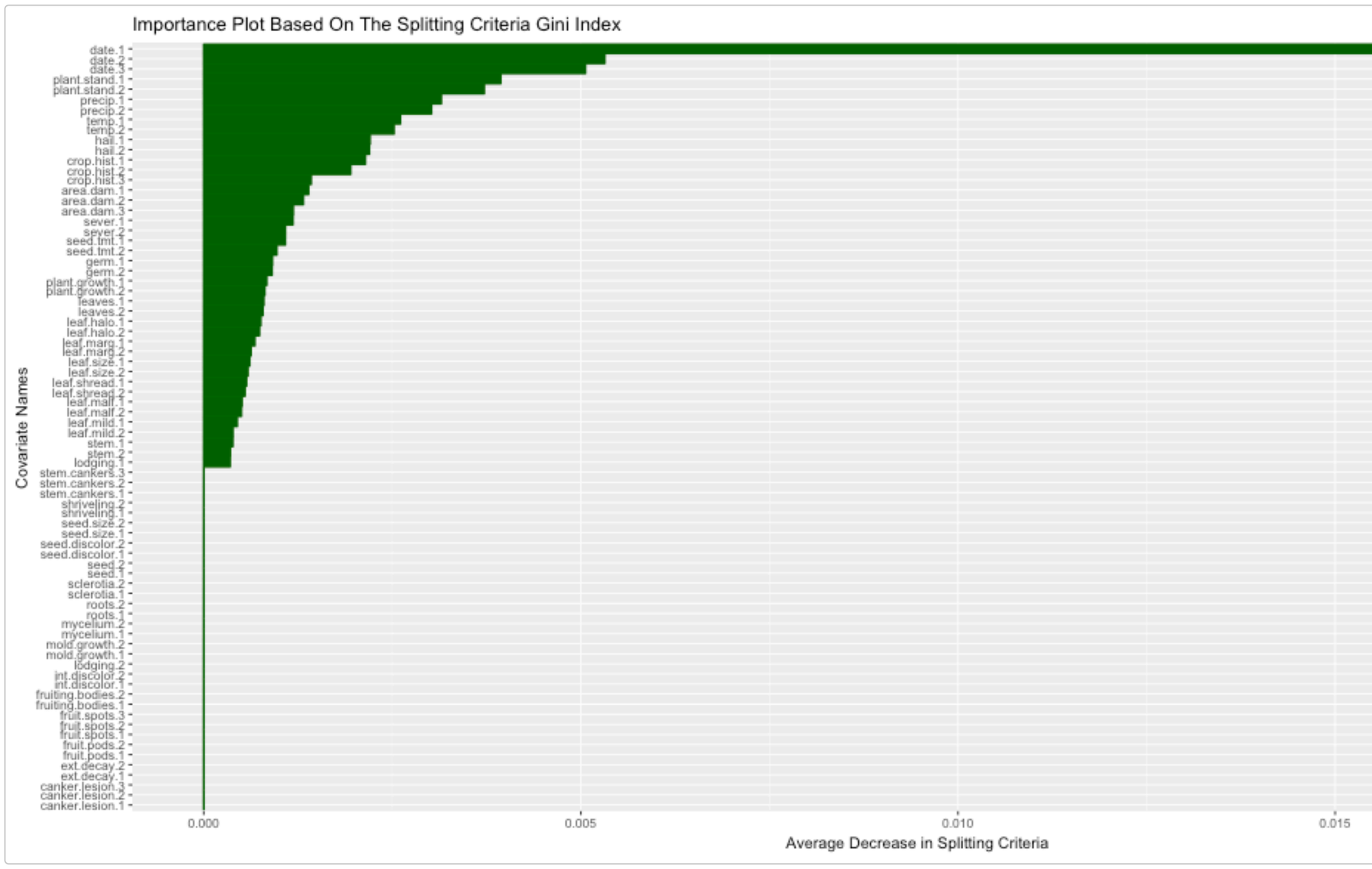
Gini Importance is defined as the total decrease in node impurity averaged over all trees of the ensemble, where the decrease in node impurity is obtained after weighting by the probability for an observation to reach that node (which is approximated by the proportion of samples reaching that node).

The R code below illustrates how to use **importance.forestRK** function:

```
## Generate 2-dimensional MDS plot of the test observations colour coded by the
## predictions stored under the object pred.forest.rk1
imp <- importance.forestRK(forestRK.1)

# gives the list of covariate names ordered from most important to least important
covariate.names <- imp$importance.covariate.names

# gives the list of average decrease in (weighted) node impurity across all trees in the forest
# ordered from the highest to lowest in value.
# that is, the first element of this vector pertains to the first covariate listed under
# imp$importance.covariate.names
ave.decrease.criteria <- imp$average.decrease.in.criteria.vec
```

- **importance.plot.forestRK** function takes **importance.forestRK** function as an input and generates an Importance Plot based on the Gini Importance of each covariates. The covariates are ordered from the one with the highest importance to the one with the lowest importance. The R code below demonstrates an example for using the **importance.plot.forestRK** function:

```
## Generate importance plot of the forestRK.1 object
p <- importance.plot.forestRK(imp, colour.used="dark green", fill.colour="dark green", label.size=8)

p
```



9. Comparing Performances Of Different Types of Encodings and Splitting Criteria

At times users may want to compare the overall performances of different types of encodings (Numeric vs. Binary) and splitting criteria (Entropy vs. Gini Index) by the predictive power of their **forestRK** models.

The R code below shows how to compare the **forestRK** models with different encodings and splitting criterias:

```
## overall prediction accuracy (in percentage) when training data was modified with Binary Encoding
## and the Gini Index was used as splitting criteria.
y.test <- Soybean[-vec,1] # this is an actual class type of test observations
sum(as.vector(pred.forest.rk1$pred.for.obs.forest.rk) == as.vector(y.test)) * 100 / length(y.test)
```

```
## [1] 75.66845
```

```
## overall prediction accuracy (in percentage) when training data was modified with numeric Encoding
## and the Entropy was used as splitting criteria.
y.test <- Soybean[-vec,1] # this is an actual class type of test observations
sum(as.vector(pred.forest.rk2$pred.for.obs.forest.rk) == as.vector(y.test)) * 100 / length(y.test)
```

```
## [1] 71.12299
```

```
### NOTE: IF YOU SHUFFLED THE DATA before dividing them into a training and a test
###       set, since the observations in pred.forest.rk1 and pred.forest.rk2 are
###       re-ordered by the increasing original observation index number (the
###       original row index of each test observation),
###       the order of the test data points presented in pred.forest.rk objects
###       may not be same as the shuffled order of the observations in your
###       test set.
###
###       In this case, you have to reorder the vector y.test in the order of
###       increasing observation number before you make the comparison between
###       y.test and pred.for.obs.forest.rk
###
###       In this case, simply do:
###
###       names(y.test) <- rownames(x.test)
###       sum(as.vector(pred.forest.rk$pred.for.obs.forest.rk) ==
###           as.vector(y.test[order(as.numeric(names(y.test)))])) / length(y.test)
###
###
###  All of the functions in the forestRK package works well with the shuffled
###  datasets; it's just that when the user calculates the accuracy of the
###  prediction by the rktree or forestRK models by the methods shown here, the
###  user needs to rearrange the vector y.test in the order of
###  increasing observation index.
```