Week 5

Jordan Lian

3/2/2021

```
library(tidyverse)
```

Problem 1 (20 Points)

Build an R Notebook of the bank loan decision tree example in the textbook on pages 135 to 148; the CSV file is available for download below. Show each step and add appropriate documentation. Note that the provided dataset uses values *1* and *2* in default column whereas the book has *no* and *yes* in the default column. To fix any problems replace "*no*" with "*1*" and "*yes*" with "*2*" in the code that for matrix_dimensions. Alternatively, change the line below:

```
error_cost <- matrix(c(0, 1, 4, 0), nrow = 2, dimnames = matrix_dimensions)
```

to the following:

```
error_cost <- matrix(c(0, 1, 4, 0), nrow = 2)
```

If your tree produces poor results or runs slowly, add *control=Weka_control(R=TRUE)*.

Step 1 - collecting data

http://archive.ics.uci.edu/ml

Step 2 - exploring and preparing the data

```
credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
str(credit)
```

```
## 'data.frame':    1000 obs. of  21 variables:
##  $ checking_balance    : Factor w/ 4 levels "< 0 DM","> 200 DM",..: 1 3 4 1 1 4 4 3 4 3 ...
##  $ months_loan_duration: int  6 48 12 42 24 36 24 36 12 30 ...
##  $ credit_history      : Factor w/ 5 levels "critical","delayed",..: 1 5 1 5 2 5 5 5 5 1 ...
##  $ purpose             : Factor w/ 10 levels "business","car (new)",..: 8 8 5 6 2 5 6 3 8 2 ...
##  $ amount              : int  1169 5951 2096 7882 4870 9055 2835 6948 3059 5234 ...
##  $ savings_balance     : Factor w/ 5 levels "< 100 DM","> 1000 DM",..: 5 1 1 1 1 5 4 1 2 1 ...
##  $ employment_length   : Factor w/ 5 levels "> 7 yrs","0 - 1 yrs",..: 1 3 4 4 3 3 1 3 4 5 ...
##  $ installment_rate    : int  4 2 2 2 3 2 3 2 2 4 ...
##  $ personal_status     : Factor w/ 4 levels "divorced male",..: 4 2 4 4 4 4 4 4 1 3 ...
##  $ other_debtors       : Factor w/ 3 levels "co-applicant",..: 3 3 3 2 3 3 3 3 3 3 ...
##  $ residence_history   : int  4 2 3 4 4 4 4 2 4 2 ...
##  $ property            : Factor w/ 4 levels "building society savings",..: 3 3 3 1 4 4 1 2 3 2 ...
##  $ age                 : int  67 22 49 45 53 35 53 35 61 28 ...
##  $ installment_plan    : Factor w/ 3 levels "bank","none",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ housing             : Factor w/ 3 levels "for free","own",..: 2 2 2 1 1 1 2 3 2 2 ...
##  $ existing_credits    : int  2 1 1 1 2 1 1 1 1 2 ...
##  $ default             : int  1 2 1 1 2 1 1 1 1 2 ...
##  $ dependents          : int  1 1 2 2 2 2 1 1 1 1 ...
##  $ telephone           : Factor w/ 2 levels "none","yes": 2 1 1 1 1 2 1 2 1 1 ...
##  $ foreign_worker      : Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...
##  $ job                 : Factor w/ 4 levels "mangement self-employed",..: 2 2 4 2 2 4 2 1 4 1 ...
```

```
table(credit$checking_balance)
```

```
## 
##     < 0 DM   > 200 DM 1 - 200 DM    unknown 
##        274         63        269        394
```

```
table(credit$savings_balance)
```

```
## 
##      < 100 DM     > 1000 DM  101 - 500 DM 501 - 1000 DM       unknown 
##           603            48           103            63           183
```

```
summary(credit$months_loan_duration)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##     4.0    12.0    18.0    20.9    24.0    72.0
```

```
summary(credit$amount)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##     250    1366    2320    3271    3972   18424
```

```
table(credit$default)
```

```
## 
##   1   2 
## 700 300
```

Data preparation - creating random training and test datasets

```
set.seed(123)
train_sample <- sample(1000, 900)
str(train_sample)
```

```
##  int [1:900] 415 463 179 526 195 938 818 118 299 229 ...
```

```
credit_train <- credit[train_sample, ]
credit_test  <- credit[-train_sample, ]

prop.table(table(credit_train$default))
```

```
##
##         1         2
## 0.7055556 0.2944444
```

```
prop.table(table(credit_test$default))
```

```
##
##    1    2
## 0.65 0.35
```

Step 3 - training a model on the data

```
library(C50)
```

```
## Warning: package 'C50' was built under R version 4.0.4
```

```
credit_train$default<-as.factor(credit_train$default)
credit_model <- C5.0(credit_train[-17], credit_train$default)
credit_model
```

```
##
## Call:
## C5.0.default(x = credit_train[-17], y = credit_train$default)
##
## Classification Tree
## Number of samples: 900
## Number of predictors: 20
##
## Tree size: 42
##
## Non-standard options: attempt to group attributes
```

Step 4 - evaluating model performance

```
credit_pred <- predict(credit_model, credit_test)

library(gmodels)
```

```
## Warning: package 'gmodels' was built under R version 4.0.3
```

```r
CrossTable(credit_test$default, credit_pred,
           prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
           dnn = c('actual default', 'predicted default'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                | predicted default
## actual default |         1 |         2 | Row Total |
## ---------------|-----------|-----------|-----------|
##              1 |        55 |        10 |        65 |
##                |     0.550 |     0.100 |           |
## ---------------|-----------|-----------|-----------|
##              2 |        22 |        13 |        35 |
##                |     0.220 |     0.130 |           |
## ---------------|-----------|-----------|-----------|
##   Column Total |        77 |        23 |       100 |
## ---------------|-----------|-----------|-----------|
##
##
```

Step 5 - improving model performance

Boosting the accuracy of decision trees

```r
credit_boost10 <- C5.0(credit_train[-17], credit_train$default,
                         trials = 10)
credit_boost10
```

```
##
## Call:
## C5.0.default(x = credit_train[-17], y = credit_train$default, trials = 10)
##
## Classification Tree
## Number of samples: 900
## Number of predictors: 20
##
## Number of boosting iterations: 10
## Average tree size: 36.3
##
## Non-standard options: attempt to group attributes
```

```r
credit_boost_pred10 <- predict(credit_boost10, credit_test)
CrossTable(credit_test$default, credit_boost_pred10,
           prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
           dnn = c('actual default', 'predicted default'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                | predicted default
## actual default |         1 |         2 | Row Total |
## ---------------|-----------|-----------|-----------|
##              1 |        59 |         6 |        65 |
##                |     0.590 |     0.060 |           |
## ---------------|-----------|-----------|-----------|
##              2 |        17 |        18 |        35 |
##                |     0.170 |     0.180 |           |
## ---------------|-----------|-----------|-----------|
##   Column Total |        76 |        24 |       100 |
## ---------------|-----------|-----------|-----------|
##
##
```

Making mistakes more costliers than others

```r
matrix_dimensions <- list(c("no", "yes"), c("no", "yes"))
names(matrix_dimensions) <- c("predicted", "actual")
matrix_dimensions
```

```
## $predicted
## [1] "no"  "yes"
##
## $actual
## [1] "no"  "yes"
```

```r
error_cost <- matrix(c(0, 1, 4, 0), nrow = 2)
error_cost
```

```
##      [,1] [,2]
## [1,]    0    4
## [2,]    1    0
```

```r
credit_cost <- C5.0(credit_train[-17], credit_train$default,
                          costs = error_cost)
credit_cost_pred <- predict(credit_cost, credit_test)

CrossTable(credit_test$default, credit_cost_pred,
           prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
           dnn = c('actual default', 'predicted default'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                | predicted default
## actual default |         1 |         2 | Row Total |
## ---------------|-----------|-----------|-----------|
##              1 |        43 |        22 |        65 |
##                |     0.430 |     0.220 |           |
## ---------------|-----------|-----------|-----------|
##              2 |         8 |        27 |        35 |
##                |     0.080 |     0.270 |           |
## ---------------|-----------|-----------|-----------|
##   Column Total |        51 |        49 |       100 |
## ---------------|-----------|-----------|-----------|
##
##
```

Problem 2 (10 Points)

Build an R Notebook of the poisonous mushrooms example using rule learners in the textbook on pages 160 to 168. Show each step and add appropriate documentation. The CSV file is available below. If you have issues with the **RWeka** package on MacOS, consider using a Windows computer, RStudio.cloud or skip this question.

Tip: In case anyone gets this error on the 1R implementation:

```r
mushroom_1R <- OneR(type ~ ., data = mushrooms)
Error in .jcall(o, "Ljava/lang/Class;", "getClass") :
  weka.core.UnsupportedAttributeTypeException: weka.classifiers.rules.OneR: ...
```

Change your characters to factors. Here's an explanation why factors are needed.

Step 1 - collecting data

http://archive.ics.uci.edu/ml

Step 2 - exploring and preparing the data

```r
mushrooms <- read.csv("mushrooms.csv", stringsAsFactors = TRUE)
mushrooms$veil_type <- NULL
table(mushrooms$type)
```

```
##
##    edible poisonous
##      4208      3916
```

Step 3 - training a model on the data

```r
library(RWeka)
```

```
## Warning: package 'RWeka' was built under R version 4.0.4
```

```r
mushroom_1R <- OneR(type ~ ., data = mushrooms)
mushroom_1R
```

```
## odor:
##  almond  -> edible
##  anise   -> edible
##  creosote    -> poisonous
##  fishy   -> poisonous
##  foul    -> poisonous
##  musty   -> poisonous
##  none    -> edible
##  pungent -> poisonous
##  spicy   -> poisonous
## (8004/8124 instances correct)
```

Step 4 - evaluating model performance

```r
summary(mushroom_1R)
```

```
##
## === Summary ===
##
## Correctly Classified Instances        8004               98.5229 %
## Incorrectly Classified Instances       120                1.4771 %
## Kappa statistic                          0.9704
## Mean absolute error                      0.0148
## Root mean squared error                  0.1215
## Relative absolute error                  2.958  %
## Root relative squared error             24.323  %
## Total Number of Instances             8124
##
```

```
## === Confusion Matrix ===
##
##     a    b   <-- classified as
##  4208    0 |    a = edible
##   120 3796 |    b = poisonous
```

Step 5 - improving model performance

```
mushroom_JRip <- JRip(type ~ ., data = mushrooms)
mushroom_JRip
```

```
## JRIP rules:
## ===========
##
## (odor = foul) => type=poisonous (2160.0/0.0)
## (gill_size = narrow) and (gill_color = buff) => type=poisonous (1152.0/0.0)
## (gill_size = narrow) and (odor = pungent) => type=poisonous (256.0/0.0)
## (odor = creosote) => type=poisonous (192.0/0.0)
## (spore_print_color = green) => type=poisonous (72.0/0.0)
## (stalk_surface_below_ring = scaly) and (stalk_surface_above_ring = silky) => type=poisonous (68.0/0.
## (habitat = leaves) and (cap_color = white) => type=poisonous (8.0/0.0)
## (stalk_color_above_ring = yellow) => type=poisonous (8.0/0.0)
##  => type=edible (4208.0/0.0)
##
## Number of Rules : 9
```

Problem 3 (35 Points)

So far we have explored four different approaches to classification: kNN, Naive Bayes, C5.0 Decision Trees, and RIPPER Rules. Comment on the differences of the algorithms and when each is generally used. Provide examples of when they work well and when they do not work well. Add your comments to your R Notebook. Be specific and explicit; however, no code examples are needed.

kNN

k-nearest neighbors is a non-parametric classification that predicts the output based on the k closest training inputs in the data set. The algorithm uses distance for classifcation, so normalizing the training data will significantly improve its accuracy. Usually the distance is measured as a Euclidean distance, although there are other types like Manhattan, Hamming, etc. kNN is pretty unbiased, and the algorithm itself is simple. However, for data sets with higher dimensions, kNN is not good due to time and space complexity. kNN can be used for recommendation systems, outlier detection systems, and documents finders for semantic similarity.

Naive Bayes

Naive Bayes applies conditional probability to determine predicted values given their input. Spam filter emails are built off Naive Bayes, and they work surprisingly well in certain cases. Unlike kNN, Naive Bayes works well with high-dimensional data because it's easy to parallelize and it handles big data well. However, due to its Naivety (pun intended), other algorithms tend to outperform Naive Bayes.

C5.0 Decision Trees

Decision trees are simple and they are easy to understand and implement. Decision trees merely divide up the data into different levels to predict the output based on the inputs. However, they suffer from high variance, which means the smallest changes in the training data can screw up the entire decision tree. The C5.0 decision trees are the industry standard algorithm for producing decision trees, as they pretty well in comparison to the more advanced methods. It estimates the error rate of every internal node, and replaces it with a leaf node if the estimated error of the leaf is lower. C5.0 is the latest version, so there will most likely will be a new version soon.

RIPPER Rules

RIPPER rules, also known as "Repeated Incremental Pruning to Produce Error Reduction,"is a rule-based classification algorithm, where it gets a set of rules from the training data set itself, thus it is considered an induction algorithm. RIPPER works well with imbalanced class distributions. Imbalanced class distributions are present when most of the records in a data set belong to a particular class while the rest belong to different classes. RIPPER also works well with noisy data sets (corrupted data) because it uses a validation data set to prevent model over-fitting.

Problem 4 (35 Points)

Much of our focus so far has been on building a single model that is most accurate. In practice, data scientists often construct multiple models and then combine them into a single prediction model. This is referred to as a model ensemble. Two common techniques for assembling such models are boosting and bagging. Do some research and define what model ensembles are, why they are important, and how boosting and bagging function in the construction of assemble models. Be detailed and provide references to your research. You can use this excerpt from Kelleher, MacNamee, and D'Arcy, Fundamentals of Machine Learning for Predictive Data Analytics as a starting point. This book is an excellent resource for those who want to dig deeper into data mining and machine learning.

Model ensembles are prediction models composed of a set of models. The idea is to have multiple people working on the same model independently to guard against group think. Ensembles have two defining properties:

1. Ensembles induce each model using a modified version of the data set to build multiple and different models from the same data set.
2. Ensembles aggregate the predictions of the different models to make a final prediction. They use different voting mechanisms for categorical features, and they use central tendencies (mean/median) for continuous target features.

In addition to having two properties, ensembles have two standard methods of creation: boosting and bagging.

1. For **boosting**, each new model added is biased to pay more attention to the instances that the previous models misclassified. Boosting uses a weighted data set that can incrementally adapt the data set used to train the models. The weights are initially assigned as $\frac{1}{n}$, where n is the number of instances in the data set. These weights distribute across the data set to create a new replicated and proportionally weighted dataset.

2. **Bagging** (bootstrap aggregating) is where each model in the ensemble is based on a random sample of the data set where sampling replacement is used and the random sample is the exact same size as the data set. Sampling with replacement is used because there will be duplicates and some missing values,

which will make each bootstrap sample different. Bagging works well with decision tree induction algorithms because decision trees are highly sensitive to changes in the data set as I mentioned above. When bagging is used with decision trees, subspace sampling is quite frequent. Subspace sampling is where the sampling process is extended so each sample is only sampled from a randomly selected subset of the data set's descriptive features. The combination of bagging, subspace sampling, and decision trees is known as a **random forest model**. When the individual models are induced, the predictions are determined by a vote or the median (the median is less affected by outliers than the mean).

Overall, bagging is simpler than boosting, although they each have their different uses. For data sets with up to 4,000 descriptive features, boosted decision tree ensembles outperformed the bagging random forest ensembles. However, for anything with more than 4,000 features, random forest ensembles (bagging) performed better. Boosted ensembles are prone to over-fitting, so it becomes more of a problem when there are more features in a data set.

Model ensembles technically don't have to consist of decision trees, they can have any type of prediction model. However, decision trees are sensitive to changes in the data sets (induction sensitivity). The most important thing is that ensembles consist of an entire set of different models built independently from the other models. That way you can get the best of all of the models and combine them together.

Resources

https://da5030.weebly.com/rweka-hints.html

https://rstudio.cloud/