DATA622_HW3

Group 5

2021-04-09

Contents

Authorship

Group 5:

- Don (Geeth) Padmaperuma,
- Subhalaxmi Rout,
- Isabel Ramesar, and
- Magnus Skonberg

Background

The purpose of this assignment was to explore classification via K-nearest neighbors, Decision Trees, Random Forests, and Gradient Boosting.

Classification

Classification is a supervised machine learning technique whose main purpose is to identify the category/class of provided input data. The model that is generated is trained using labeled data (hence the label "supervised") and then the trained model is used to predict our discrete output.

Our Approach

First, we're going to predict `species` of penguin using the KNN algorithm.

Then we're going to compare and contrast `loan approval status` prediction accuracy for Decision Trees, Random Forests and Gradient Boosting.

....

Palmer Penguins Data

Being that we've worked with the penguins dataset twice before, we perform *light EDA* to re-familiarize ourselves with the data prior to applying the KNN algorithm to it.

We load in the data, pre-process it, verify the first 6 observations, and utilize the built-in glimpse() function to gain insight into the dimensions, variable characteristics, and value range:

```r
#Load and tidy data
penguin_measurements <- penguins %>% drop_na() %>%
    dplyr::select(species, bill_length_mm, bill_depth_mm,
                  flipper_length_mm, body_mass_g)

head(penguin_measurements%>% as.data.frame())
```

```
##   species bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
## 1  Adelie           39.1          18.7               181        3750
## 2  Adelie           39.5          17.4               186        3800
## 3  Adelie           40.3          18.0               195        3250
## 4  Adelie           36.7          19.3               193        3450
## 5  Adelie           39.3          20.6               190        3650
## 6  Adelie           38.9          17.8               181        3625
```

```
glimpse(penguin_measurements)
```

```
## Rows: 333
## Columns: 5
## $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, A...
## $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, 36.7, 39.3, 38.9, 39.2, 41.1, 3...
## $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, 19.3, 20.6, 17.8, 19.6, 17.6, 2...
## $ flipper_length_mm <int> 181, 186, 195, 193, 190, 181, 195, 182, 191, 198,...
## $ body_mass_g       <int> 3750, 3800, 3250, 3450, 3650, 3625, 4675, 3200, 3...
```

Once we've dropped NA values and selected pertinent variables, we end up with a 333 observation x 5 variable dataframe with:

- `species`, a categorical variable of type factor, as our dependent variable and
- `bill_length_mm`, `bill_depth_mm`, `flipper_length_mm`, and `body_mass_g`, all quantitative variables of type dbl or int, as our independent variables.

1. K-Nearest Neighbors (KNN)

The KNN algorithm hinges on the idea that similar data will be near one another. When applying KNN, the distance between data points are used to classify data. Those nearer one another are "batched" together. The amount of batches we see and the distance between points is determined by the k value we select:

- Smaller k –> fewer batches and larger distance between "like" points.
- Larger k –> more batches and smaller distance between "like" points.

Due to the simplicity of calculations involved and ease of interpretability of the result, KNN is a popular classifier. For our purposes, we're going to apply the K-nearest neighbor algorithm to the Palmer penguins dataset to predict the `species` variable.

We perform an 80-20 split, center and scale our independent variables via built-in scale() function, and fit the KNN model to our training dataset:

```
# Splitting data into train and test data
set.seed(123)
training.individuals <- penguin_measurements$species %>%
    createDataPartition(p = 0.8, list = FALSE)

train_cl <- penguin_measurements[training.individuals, ]
test_cl <- penguin_measurements[-training.individuals, ]

# Feature Scaling
train_scale <- scale(train_cl[, 2:5])
test_scale <- scale(test_cl[, 2:5])

# Fitting KNN Model to training dataset
classifier_knn <- knn(train = train_scale,
                      test = test_scale,
                      cl = train_cl$species,
                      k = 1)

#classifier_knn #verify output
```

Performing an 80-20 split allocates 80% of our data for training the model and 20% of our data for testing it. Whereas applying the scale() function centers and scales our independent variables to reduce bias and improve predictive accuracy. *Although, we start with k=1 (neighbors) for our KNN model fit here. We'll vary this value later to interpret its impact on predictive accuracy.*

Once our model has been fit, we assess its accuracy via confusion matrix :

```
# Confusion Matrix
cm <- table(test_cl$species, classifier_knn)
cm
```

```
##            classifier_knn
##             Adelie Chinstrap Gentoo
##   Adelie        28         1      0
##   Chinstrap      3        10      0
##   Gentoo         0         0     23
```

```
# Model Evaluation - Choosing K Calculate out of Sample error
misClassError <- mean(classifier_knn != test_cl$species)
print(paste('Accuracy =', 1-misClassError))
```

```
## [1] "Accuracy = 0.938461538461538"
```

In the confusion matrix above, rows represent actual values while columns represent predicted values. With this in mind, we see that our test set results are:

- True Positive Result (TPR): 61 / 65 = 93.8%
- False Positive Result (FPR): 4 / 65 = 6.2%

Although our KNN classifier mis-predicted an Adelie as a Chinstrap and 3 Chinstraps as Adelies, it predicted Gentoo with 100% accuracy and produced a relatively favorable **93.8% accuracy**.

Let's explore the impact of increasing our k value to 3:

```
# K = 3
classifier_knn <- knn(train = train_scale,
                      test = test_scale,
                      cl = train_cl$species,
                      k = 3)
misClassError <- mean(classifier_knn != test_cl$species)
print(paste('Accuracy (k=3):', 1-misClassError))
```

```
## [1] "Accuracy (k=3): 0.953846153846154"
```

We don't visualize the output with a confusion matrix and instead calculate the sample error and just subtract this value from 1 to produce a predictive **accuracy of 95.4%**. A 1.6% improvement over our KNN classifier with k=1.

We, once again, explore the impact of increasing our k value. This time we increase it to 15:

```
# K = 15
classifier_knn <- knn(train = train_scale,
                      test = test_scale,
                      cl = train_cl$species,
                      k = 15)
misClassError <- mean(classifier_knn != test_cl$species)
print(paste('Accuracy (k=15):', 1-misClassError))
```

```
## [1] "Accuracy (k=15): 0.969230769230769"
```

With k=15 **our predictive accuracy climbs to 96.9%.** A 1.5% improvement over our KNN classifier with k=3. In our case, each increase in k value improved our predictive accuracy. This is not always the case though …

When choosing k values, smaller values are generally less computationally expensive *yet* they're also noisier and less accurate. Larger values, on the other hand, can result in smoother decision boundaries and a lower variance *yet* they increase the bias and processing demands.

Thus, more often than not we seek the "sweet spot". A k value that's not too large and not too small. Our choice in value is thus impacted by the number of observations, as well as the characteristics of the data we're classifying.

…………………………………………………………………..

Loan Approval Data

Being that we haven't worked with the loan approval dataset before, our exploratory data analysis (EDA) will be more in-depth. The depth will allow for greater understanding of the data at hand prior to applying a Decision Tree, Random Forest, and Gradient Boosting model to it.

We load in the data, replace empty strings with NAs, and observe the first 6 observations of our dataset:

```
#Load in data
loan <- read.csv("https://raw.githubusercontent.com/SubhalaxmiRout002/Data-622-Group-5/main/Loan_approva

loan[loan==""] <- NA #replace empty strings with NAs
#head(loan) #verify 1st 6 observations
```

The head() function provides some context regarding the format of our data. R's built-in glimpse() and summary() functions provide further insight:

```
#Light EDA
glimpse(loan)
```

```
## Rows: 614
## Columns: 13
## $ Loan_ID           <fct> LP001002, LP001003, LP001005, LP001006, LP001008,...
## $ Gender            <fct> Male, Male, Male, Male, Male, Male, Male, Male, M...
## $ Married           <fct> No, Yes, Yes, Yes, No, Yes, Yes, Yes, Yes, Yes, Y...
## $ Dependents        <fct> 0, 1, 0, 0, 0, 2, 0, 3+, 2, 1, 2, 2, 2, 0, 2, 0, ...
## $ Education         <fct> Graduate, Graduate, Graduate, Not Graduate, Gradu...
## $ Self_Employed     <fct> No, No, Yes, No, No, Yes, No, No, No, No, No, NA,...
## $ ApplicantIncome   <int> 5849, 4583, 3000, 2583, 6000, 5417, 2333, 3036, 4...
## $ CoapplicantIncome <dbl> 0, 1508, 0, 2358, 0, 4196, 1516, 2504, 1526, 1096...
## $ LoanAmount        <int> NA, 128, 66, 120, 141, 267, 95, 158, 168, 349, 70...
## $ Loan_Amount_Term  <int> 360, 360, 360, 360, 360, 360, 360, 360, 360, 360,...
## $ Credit_History    <int> 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, N...
## $ Property_Area     <fct> Urban, Rural, Urban, Urban, Urban, Urban, Urban, ...
## $ Loan_Status       <fct> Y, N, Y, Y, Y, Y, Y, N, Y, N, Y, Y, Y, N, Y, Y, Y...
```

```r
summary(loan)
```

```
##       Loan_ID       Gender    Married    Dependents        Education
##  LP001002:  1         :  0       :  0       :  0   Graduate    :480
##  LP001003:  1   Female:112   No  :213   0   :345   Not Graduate:134
##  LP001005:  1   Male  :489   Yes :398   1   :102
##  LP001006:  1   NA's  : 13   NA's:  3   2   :101
##  LP001008:  1                           3+  : 51
##  LP001011:  1                           NA's: 15
##  (Other) :608
##  Self_Employed ApplicantIncome CoapplicantIncome   LoanAmount
##      :  0      Min.   :  150   Min.   :    0     Min.   :  9.0
##  No  :500      1st Qu.: 2878   1st Qu.:    0     1st Qu.:100.0
##  Yes : 82      Median : 3812   Median : 1188     Median :128.0
##  NA's: 32      Mean   : 5403   Mean   : 1621     Mean   :146.4
##                3rd Qu.: 5795   3rd Qu.: 2297     3rd Qu.:168.0
##                Max.   :81000   Max.   :41667     Max.   :700.0
##                                                  NA's   :22
##  Loan_Amount_Term Credit_History     Property_Area Loan_Status
##  Min.   : 12      Min.   :0.0000   Rural    :179   N:192
##  1st Qu.:360      1st Qu.:1.0000   Semiurban:233   Y:422
##  Median :360      Median :1.0000   Urban    :202
##  Mean   :342      Mean   :0.8422
##  3rd Qu.:360      3rd Qu.:1.0000
##  Max.   :480      Max.   :1.0000
##  NA's   :14       NA's   :50
```

We're dealing with a 614 observation x 13 variable dataframe with:

- `Loan_Status`, a categorical, character-based variable, as our dependent variable,
- `ApplicantIncome`, `CoApplicantIncome`,`LoanAmount`, `Loan_Amount_Term`, and `Credit_History`, all quantitative variables of type dbl or int, as independent variables, and
- `Loan_ID`, `LoanGender`, `Married`, `Dependents`, `Education`, `Self_Employed`, `Property_Area`, and `Loan_Status`, all categorical, character-based variables, as independent variables.

From the above output, we also get an idea of proportions for our variables of type factor (ie.`Gender`: 489 male, 112 female applicants and `Loan_Status`: 422 Y, 192 N) which we can explore in greater depth later.

Of the above variables, we can see that `Loan_ID` does not appear to provide much insight. We'll drop this variable, explore a clearer visualization of NA counts and then deal with our NA values:

```r
loan <- subset(loan, select = -c(1) ) #drop Loan_ID from consideration

colSums(is.na(loan)) #visualize NA counts
```

```
##            Gender           Married        Dependents         Education
##                13                 3                15                 0
##     Self_Employed   ApplicantIncome CoapplicantIncome        LoanAmount
##                32                 0                 0                22
##  Loan_Amount_Term    Credit_History     Property_Area       Loan_Status
##                14                50                 0                 0
```

7 / 12 variables have NA values and 3 of these variables have more than 20 NA values (a relatively significant margin).

We *can* drop these values but dropping values means losing valuable observations and thus we elect to impute instead. From the mice library, we impute using the **pmm** method (predictive mean matching):

```
#relabel Dependents "3+" value as "3" so that we can impute values
loan$Dependents <- revalue(loan$Dependents, c("3+"="3"))

#apply predictive mean matching to loan data
loan <- mice(loan, m = 1, method = "pmm", seed = 500)
```

```
##
##  iter imp variable
##   1   1  Gender*  Married  Dependents  Self_Employed  LoanAmount  Loan_Amount_Term*  Credit_History*
##   2   1  Gender  Married  Dependents*  Self_Employed  LoanAmount*  Loan_Amount_Term  Credit_History*
##   3   1  Gender*  Married*  Dependents*  Self_Employed  LoanAmount*  Loan_Amount_Term*  Credit_Histo
##   4   1  Gender  Married  Dependents  Self_Employed  LoanAmount*  Loan_Amount_Term*  Credit_History
##   5   1  Gender  Married  Dependents*  Self_Employed  LoanAmount*  Loan_Amount_Term*  Credit_History
##  * Please inspect the loggedEvents
```

```
loan <- mice::complete(loan, 1)
```

We re-assign the "3+" value of the `Dependents` variable to provide consistent leveling and enable **pmm** and then we actually apply **pmm**.

Predictive mean matching calculates the predicted value for our target variable, and, for missing values, forms a small set of "candidate donors" from the complete cases that are closest to the predicted value for our missing entry. Donors are then randomly chosen from candidates and imputed where values were once missing. *To apply pmm we assume that the distribution is the same for missing cells as it is for observed data, and thus, the approach may be more limited when the % of missing values is higher.*

Once we've imputed missing values into our loan dataset and returned the data in proper form, we verify whether our operation was successful:

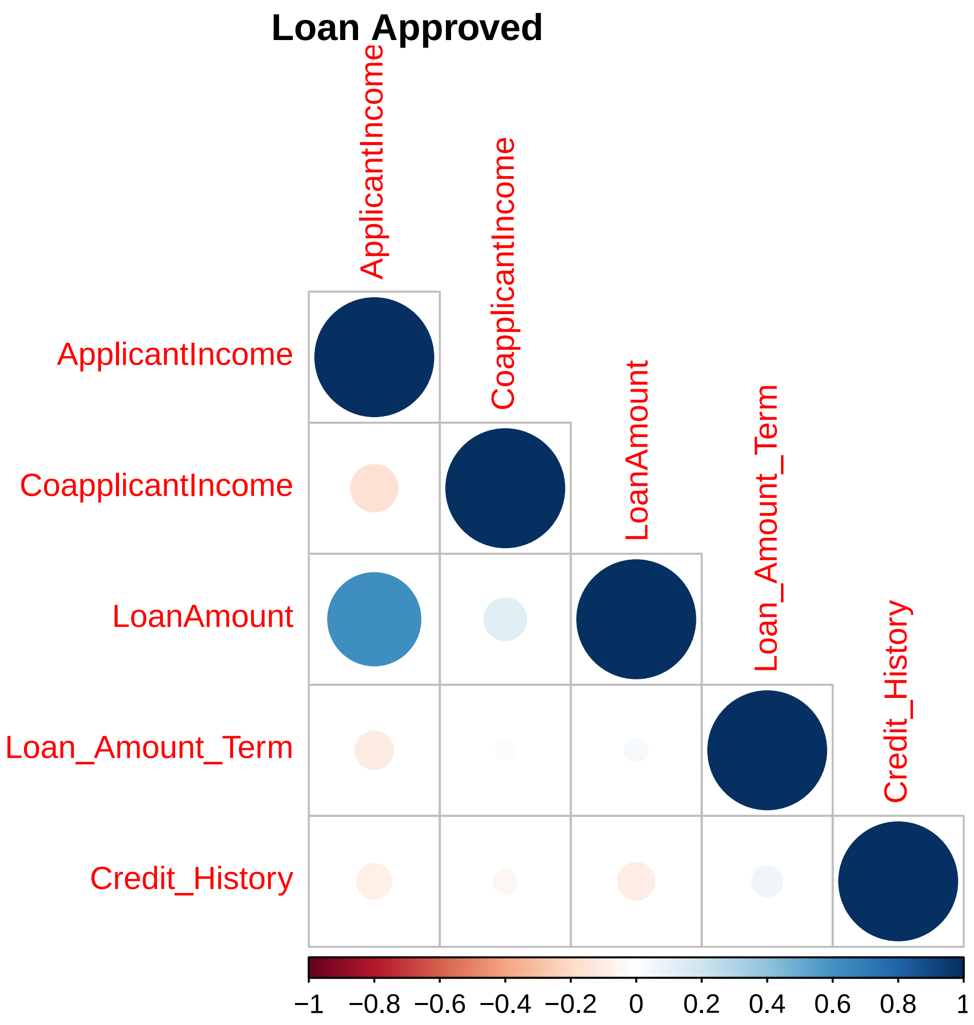
```
#verify absence of NA values in the dataset
colSums(is.na(loan))
```

```
##            Gender           Married        Dependents         Education
##                 0                 0                 0                 0
##     Self_Employed   ApplicantIncome CoapplicantIncome        LoanAmount
##                 0                 0                 0                 0
##  Loan_Amount_Term    Credit_History     Property_Area       Loan_Status
##                 0                 0                 0                 0
```

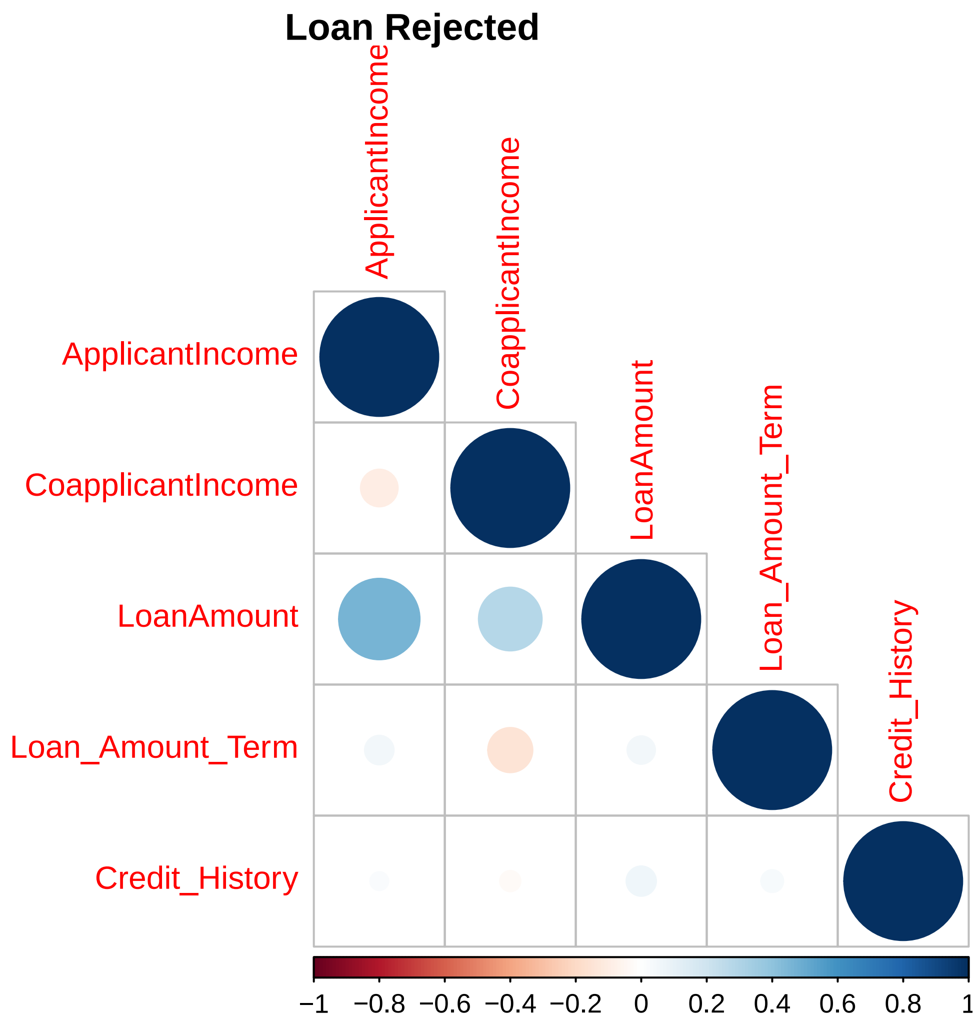
Imputation was a success and thus our data pre-processing has been completed. We can proceed with our exploratory data analysis (EDA).

To identify features that carry promise vs. those that may not, we consider a correlation matrix for our numeric variables:

```
#Correlation matrix for numeric variables
#Loan Approved correlation
loan_corr_y <- loan %>%
    filter(Loan_Status == "Y") %>%
    select_if(is.numeric) %>%
    cor()

corrplot(loan_corr_y, title="Loan Approved",type = "lower", mar=c(0,0,1,0))
```



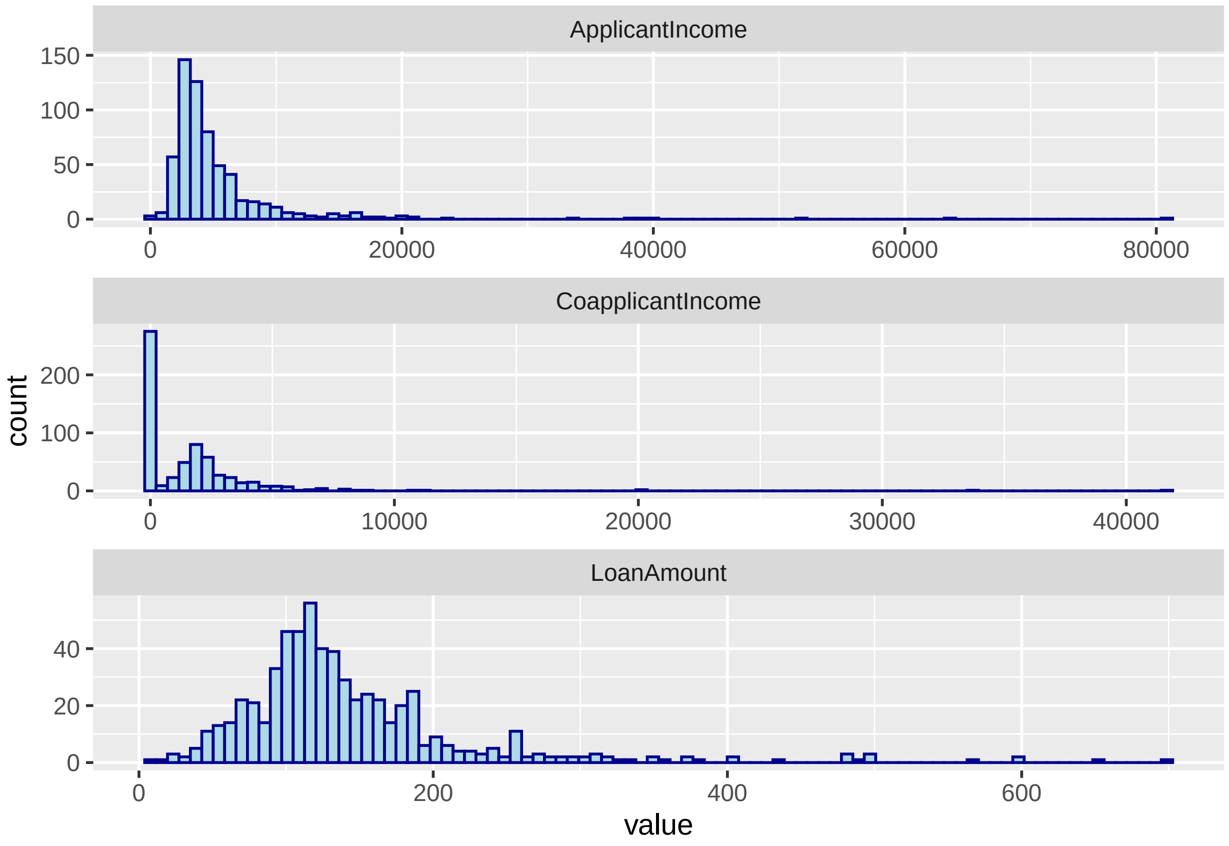
```
#Loan Rejected correlation
loan_corr_n <- loan %>%
    filter(Loan_Status == "N") %>%
    select_if(is.numeric) %>%
    cor()
corrplot(loan_corr_n, title="Loan Rejected", type = "lower", diag = T, mar=c(0,0,1,0))
```



When we consider loan approval and rejection, `LoanAmount` and `ApplicantIncome` have a relatively strong correlation. For loan rejection, `LoanAmount` also appears to have moderate correlation with `CoapplicantIncome`. Remaining features do not have strong correlation and thus we'll only note the aforementioned relationships for consideration for *possible* feature removal later.

We move on to explore histograms for our numeric variables:

```
#Histograms for all variables
loan %>%
    keep(is.numeric) %>%
    subset(select = -c(4,5)) %>% #drop CreditHistory, Loan_Amount_Term
    gather() %>%
    ggplot(aes(value)) +
        facet_wrap(~ key, scales = "free", ncol=1) +
        geom_histogram(bins=90,color="darkblue", fill="lightblue")
```



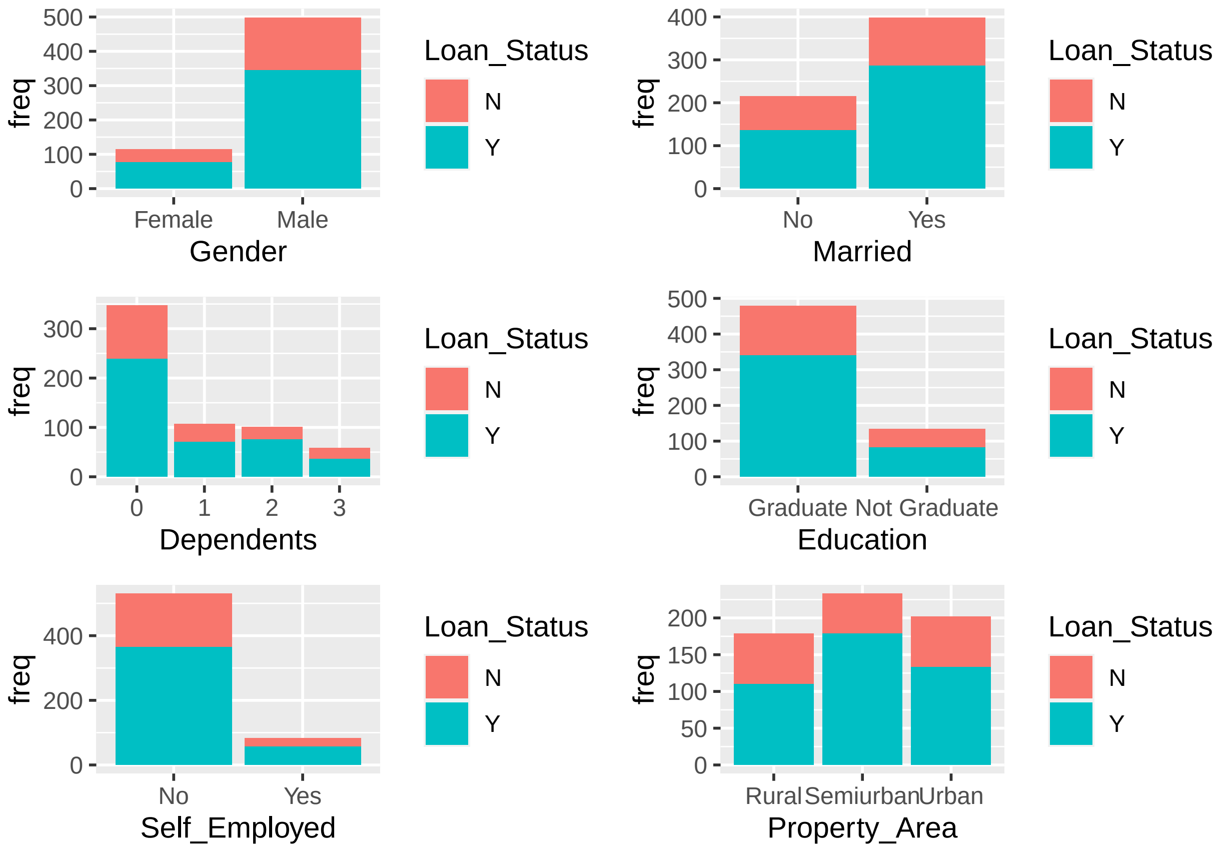
From the above figures we observe that:

- `ApplicantIncome` and `LoanAmount` appear to be right skewed normal with a number of noteworthy outliers, and
- `CoapplicantIncome` has a peak at 0 joined with an otherwise right skewed normal distribution.

Scaling is not necessary for Decision Trees or Random Forests but may prove useful for Gradient Descent. Thus, we won't act on transformation at this point, we'll just note the point for future consideration.

Next, we explore our categorical variables:

```r
#convert CreditHistory to type factor
loan$Credit_History <- factor(loan$Credit_History)
#levels(loan$Credit_History) #verify

#MS: if there's a way to facet wrap this / automate, would be preferred:

#Histograms for all categorical variables
##Gender
p1 <- loan %>%
    dplyr::select(1,12) %>% #dplyr::select(1:5,11:12)
    group_by(,Loan_Status) %>%
    #gather() %>%
    count() %>%

    ggplot(aes(x=Gender, y=freq, fill=Loan_Status)) +
        #facet_wrap(~ key) +
        geom_bar(stat='identity', position="stack")

##Married
p2 <- loan %>% dplyr::select(2,12) %>% group_by(,Loan_Status) %>% count() %>%
    ggplot(aes(x=Married, y=freq, fill=Loan_Status)) +
        geom_bar(stat='identity', position="stack")

##Dependents
p3 <- loan %>% dplyr::select(3,12) %>% group_by(,Loan_Status) %>% count() %>%
    ggplot(aes(x=Dependents, y=freq, fill=Loan_Status)) +
        geom_bar(stat='identity', position="stack")

##Education
p4 <- loan %>% dplyr::select(4,12) %>% group_by(,Loan_Status) %>% count() %>%
    ggplot(aes(x=Education, y=freq, fill=Loan_Status)) +
        geom_bar(stat='identity', position="stack")

##Self_Employed
p5 <- loan %>% dplyr::select(5,12) %>% group_by(,Loan_Status) %>% count() %>%
    ggplot(aes(x=Self_Employed, y=freq, fill=Loan_Status)) +
        geom_bar(stat='identity', position="stack")

##Property_Area
p6 <- loan %>% dplyr::select(11,12) %>% group_by(,Loan_Status) %>% count() %>%
    ggplot(aes(x=Property_Area, y=freq, fill=Loan_Status)) +
        geom_bar(stat='identity', position="stack")

grid.arrange(p1, p2, p3, p4, p5, p6, nrow = 3, ncol = 2)
```



From the above figures we can extend the following observations:

- **males outnumber females** and **non self employed outnumber self employed** on a 5:1 basis, **married outnumber non married** on a 2:1 basis, **those without dependents make up a majority**, those that have **graduated make up a majority**, and those with **properties in semiurban area make up a slight majority** of loan applications for the data under our consideration,
- with regard to loan approval, the only clearer takeaway is that it appears that **those in semiurban areas are approved at a greater rate than those in rural and urban areas**.

While the visualization of our six categorical variables distributions with regard to our dependent variable is enlightening, it doesn't appear to provide clear indication regarding one vs. another's predictive capabilities. There's the possibility that many of these variables will prove impertinent and thus we may be justified in their exclusion from our models later in the process.

With a relatively thorough exploratory analysis under our belt, we move on to the building and assessment of our Decision Tree, Random Forest, and Gradient Descent models.

2. Decision Trees

Decision trees build classification or regression models in the form of a tree structure. Datasets are broken into smaller and smaller subsets along chosen parameters, as the associated decision tree is developed.

The result is a tree with nodes and branches. Nodes denote a split point based on the attribute in question while branches denote the corresponding outcome. We start at a "root node", terminate at "leaf nodes", and use corresponding lead nodes to provide proportions regarding resulting class labels.

Due to the ease of interpretability, decision trees are a popular early classifier. For our purposes, we're dealing with a categorical dependent variable, and will build a Decision Tree model from the loan approval dataset to provide `Loan_Status` predictions (ie. Approve or Reject).

To prepare our data for Decision Tree modeling, we convert variables of type factor to numeric, remove the `Gender` variable from consideration because loan approval does not depend upon gender (as exemplified by our plots during EDA), and change variables `Education` and `Property_Area` from categorical to numerical type:

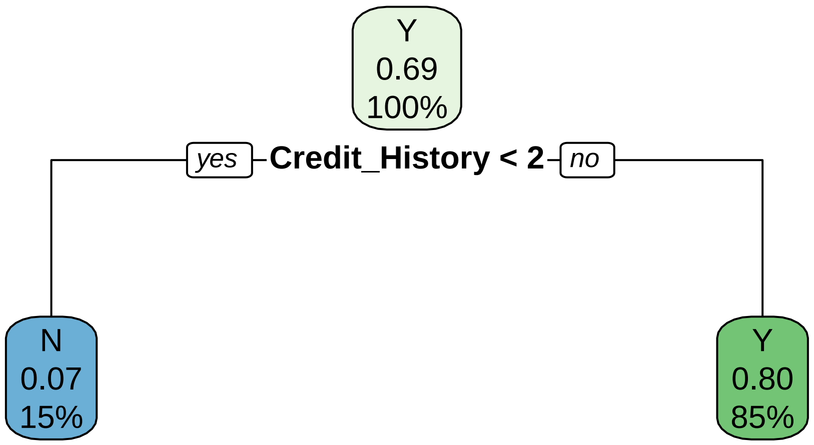
```r
# Remove Gender
loan_new <- subset(loan, select = -c (Gender))

# Convert Dependents, Credit_History numeric type
loan_new$Dependents <- as.numeric(loan_new$Dependents)
loan_new$Credit_History <- as.numeric(loan_new$Credit_History)

# Change Variables values
loan_new$Education <- ifelse(loan_new$Education=="Graduate", 1, 0)
loan_new$Married <- ifelse(loan_new$Married=="Yes", 1, 0)
loan_new$Self_Employed <- ifelse(loan_new$Self_Employed == "Yes", 1, 0)

if(loan_new$Property_Area=="Semiurban"){loan_new$Property_Area <- 2} else if(loan_new$Property_Area=="U
    loan_new$Property_Area <- 0}
```

Using the built-in **sample.split** function we perform a train test split such that 75% of our data will be training data and 25% will be testing data:

```r
#Split data into training and testing sets
set.seed(123)
sample_data = sample.split(loan_new, SplitRatio = 0.75)
train_data <- subset(loan_new, sample_data == TRUE)
test_data <- subset(loan_new, sample_data == FALSE)
```

With our data properly pre-processed and a train-test split performed, we recursively split our dataset using R's built-in **rpart** (short for Recursive Partitioning and Regression Tree) function, plot the resulting model, and train our model using R's built-in **ctree** (conditional inference tree) function:

```r
set.seed(144)

# Plot tree
binary.model <- rpart(Loan_Status ~ ., data = train_data, cp = .02)
rpart.plot(binary.model)
```



```
# Fit model
tree <- ctree(Loan_Status ~ ., data = train_data)
```

From the Decision tree above we observe that `Credit_History` **is the most important factor** when deciding if someone will be approved for a loan or not.

With our model fit to our training data and `Loan_Status` as the response variable, we can explore the model's accuracy via confusion matrix and mis-classification error rate for both our training and testing data:

```
# Confusion matrix and misclassification error rate for training data
tab <- table(Predicted = predict(tree), Actual = train_data$Loan_Status)
print(tab)
```

```
##          Actual
## Predicted   N   Y
##         N  63   5
##         Y  74 305
```

```
print(paste('Misclassification error for training data', round(1 - sum(diag(tab))/sum(tab),3)))
```

```
## [1] "Misclassification error for training data 0.177"
```

```
# Confusion matrix and misclassification error rate for testing data
testPred <- predict(tree, test_data)
testtab <- table(Predicted = testPred, Actual = test_data$Loan_Status)
print(testtab)
```

```
##          Actual
## Predicted   N   Y
##         N  24   2
##         Y  31 110
```

```
print(paste('Misclassification error for testing data', round(1 - sum(diag(testtab))/sum(testtab),3)))
```

```
## [1] "Misclassification error for testing data 0.198"
```

From the output statistics above, we can draw our model's performance as:

- Train dataset accuracy : 82.3%, error rate: 17.7%.
- Test dataset accuracy : 80.2, error rate:19.8%.

With these statistics in mind, we utilize the **confusionMatrix** function from the caret library and present our statistics as a kable table to glean more insight regarding our model's performance:

```r
#confusionMatrix(testPred, test_data$Loan_Status) #verify

DT_Model <- confusionMatrix(testPred, test_data$Loan_Status)$byClass
AccuracyDT <- confusionMatrix(testPred, test_data$Loan_Status)$overall['Accuracy']
DT_Model <- data.frame(DT_Model)
DT_Model <- rbind("Accuracy" = AccuracyDT, DT_Model)
tabview <- data.frame(DT_Model)

#present statistics as kable table
tabview %>%  kableExtra::kbl() %>% kable_styling(bootstrap_options = c("striped", "hover", "condensed",
```

	DT_Model
Accuracy	0.8023952
Sensitivity	0.4363636
Specificity	0.9821429
Pos Pred Value	0.9230769
Neg Pred Value	0.7801418
Precision	0.9230769
Recall	0.4363636
F1	0.5925926
Prevalence	0.3293413
Detection Rate	0.1437126
Detection Prevalence	0.1556886
Balanced Accuracy	0.7092532

For now, we note the Decision Tree model's performance based on these classification metrics (ie. stronger for `Pos Pred Value`, weaker for `Neg Pred Value`). Later, we'll provide an in-depth interpretation of these statistics vs. those of our Random Forest and Gradient Boosting models.

3. Random Forests

Random forests are one the most popular machine learning algorithms. They can be used for classification or regression, deal with a large number of features, and generally are so successful because they provide a good predictive performance, low incidence of over-fitting, and easy interpret-ability.

More features are generally better for this model, but we'll exclude `Gender` and `Dependents` from consideration because of their weak correlation with `Loan_Status`.

To prepare our data for Random Forest modeling, we convert variables of type factor to numeric, remove the `Gender` and `Dependents` variables from consideration, and change variables `Education` and `Property_Area` from categorical to numerical type:

```
# Remove Gender, Dependents
loan_RF <- subset(loan, select = -c(Gender, Dependents))

# Convert Credit_History to numeric type
loan_RF$Credit_History <- as.numeric(loan_RF$Credit_History)

# Change Variables values
loan_RF$Education <- ifelse(loan_RF$Education=="Graduate", 1, 0)
loan_RF$Married <- ifelse(loan_RF$Married=="Yes", 1, 0)
loan_RF$Self_Employed <- ifelse(loan_RF$Self_Employed=="Yes", 1, 0)

if(loan_RF$Property_Area=="Semiurban")
  {
    loan_RF$Property_Area <- 2
} else if(loan_RF$Property_Area=="Urban"){
    loan_RF$Property_Area <- 1
} else {
    loan_RF$Property_Area <- 0
}

#head(loan_RF) #verify output
```

Using the built-in **sample** function we perform a train test split *with replacement* such that 75% of our data will be training data and 25% will be testing data:

```
set.seed(1247)

ind <- sample(2, nrow(loan_RF), replace = TRUE, prob = c(0.75, 0.25))
train_RF <- loan_RF[ind == 1, ]
test_RF <- loan_RF[ind == 2, ]

#Verify dimensions of each set
#dim(train_RF)
#dim(test_RF)
```

The Random Forest model helps with feature selection based on importance and avoids over-fitting. We'll lean on this functionality in adapting our 2nd model based on the feature importance specified during the building of our first model.

With our data properly pre-processed and a train-test split performed, we utilize R's built-in **RandomForest** function to first help in determining the optimal value for `mtry`, and then train our model using this value.

RF Model 1

We determine the optimal `mtry` value (2) based on the lowest corresponding OOB rate, create the first model with all variables except `Gender` and `Dependents`, and then visit the corresponding Confusion Matrix and feature importance scores:

```
set.seed(222)

for (i in 1:10)
```

```r
{
rf = randomForest(Loan_Status ~ . , data = train_RF, mtry = i)
err <- rf$err.rate
oob_err <- err[nrow(err), "OOB"]
print(paste("For mtry : ", i , "OOB Error Rate : ", round(oob_err, 4)))
}
```

```
## [1] "For mtry :  1 OOB Error Rate :  0.2873"
## [1] "For mtry :  2 OOB Error Rate :  0.1849"
## [1] "For mtry :  3 OOB Error Rate :  0.2027"
## [1] "For mtry :  4 OOB Error Rate :  0.1938"
## [1] "For mtry :  5 OOB Error Rate :  0.2027"
## [1] "For mtry :  6 OOB Error Rate :  0.2027"
## [1] "For mtry :  7 OOB Error Rate :  0.2138"
## [1] "For mtry :  8 OOB Error Rate :  0.2116"
## [1] "For mtry :  9 OOB Error Rate :  0.2227"
## [1] "For mtry :  10 OOB Error Rate :  0.2183"
```

```r
#set.seed(35)
rf_1 = randomForest(Loan_Status ~ . , data = train_RF, mtry = 2)
print(rf_1)
```

```
##
## Call:
##  randomForest(formula = Loan_Status ~ ., data = train_RF, mtry = 2)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 17.82%
## Confusion matrix:
##    N   Y class.error
## N 66  77 0.538461538
## Y  3 303 0.009803922
```

```r
#Importance of each predictor
print(importance(rf_1, type=2))
```

```
##                   MeanDecreaseGini
## Married                   2.851203
## Education                 2.839618
## Self_Employed             1.812306
## ApplicantIncome          19.578320
## CoapplicantIncome        12.926162
## LoanAmount               18.687069
## Loan_Amount_Term          7.354080
## Credit_History           52.420703
## Property_Area             0.000000
```

We note an OOB (out of the box) error rate of 17.82% and will compare the output statistics of the confusion matrix with those of the second model to determine which has a greater predictive accuracy.

From the above feature importance scores, we observe that Credit_History, Applicant_Income, Loan_Amount, and CoapplicantIncome have the highest MeanDecreaseGini scores. They provide the largest average gain of purity and are thus the most important features. We will include these features and omit all others in our construction of model 2.

RF Model 2

We create the second model in the same manner as the first model. This time we only consider important factors such as Credit_History, Applicant_Income, CoapplicantIncome, and Loan_Amount:

```
set.seed(102)

for (i in 1:10)
{
rf = randomForest(Loan_Status ~ Credit_History + LoanAmount + CoapplicantIncome + ApplicantIncome, data
err <- rf$err.rate
oob_err <- err[nrow(err), "OOB"]
print(paste("For mtry : ", i , "OOB Error Rate : ", round(oob_err, 4)))
}
```

```
## [1] "For mtry :  1 OOB Error Rate :  0.1826"
## [1] "For mtry :  2 OOB Error Rate :  0.2094"
## [1] "For mtry :  3 OOB Error Rate :  0.2227"
## [1] "For mtry :  4 OOB Error Rate :  0.2294"
## [1] "For mtry :  5 OOB Error Rate :  0.2294"
## [1] "For mtry :  6 OOB Error Rate :  0.2294"
## [1] "For mtry :  7 OOB Error Rate :  0.2294"
## [1] "For mtry :  8 OOB Error Rate :  0.2249"
## [1] "For mtry :  9 OOB Error Rate :  0.2405"
## [1] "For mtry :  10 OOB Error Rate :  0.2339"
```

```
rf_2 = randomForest(Loan_Status ~ Credit_History + LoanAmount + CoapplicantIncome + ApplicantIncome , da
print(rf_2)
```

```
##
## Call:
##  randomForest(formula = Loan_Status ~ Credit_History + LoanAmount +      CoapplicantIncome + Applica
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##         OOB estimate of  error rate: 18.26%
## Confusion matrix:
##    N   Y class.error
## N 63  80 0.559440559
## Y  2 304 0.006535948
```

We note an OOB error rate of 18.26% and will compare the output statistics of the confusion matrix with those of the first model to determine which has a greater predictive accuracy.

Model Comparison

We evaluate our models by applying **RF Model 1** and **RF Model 2** to the test data set. We utilize the **confusionMatrix** function from the caret library and present our statistics as a kable table to glean more insight regarding the comparative statistics between each model's performance:

```r
#Tabular view of RF model 1 and Model 2 matrix comparison

rf_predict_1 <- predict(rf_1, newdata = test_RF)
#confusionMatrix(rf_predict_1, test_RF$Loan_Status)
RF_Model_1 <- confusionMatrix(rf_predict_1, test_RF$Loan_Status)$byClass
AccuracyRF1 <- confusionMatrix(rf_predict_1, test_RF$Loan_Status)$overall['Accuracy']
RF_Model_1 <- data.frame(RF_Model_1)
RF_Model_1 <- rbind("Accuracy" = AccuracyRF1, RF_Model_1)


rf_predict_2 <- predict(rf_2, newdata = test_RF)
#confusionMatrix(rf_predict_2, test_RF$Loan_Status)
RF_Model_2 <- confusionMatrix(rf_predict_2, test_RF$Loan_Status)$byClass
AccuracyRF2 <- confusionMatrix(rf_predict_2, test_RF$Loan_Status)$overall['Accuracy']
RF_Model_2 <- data.frame(RF_Model_2)
RF_Model_2 <- rbind("Accuracy" = AccuracyRF2, RF_Model_2)

tabularview <- data.frame(RF_Model_1, RF_Model_2)

tabularview %>%  kableExtra::kbl() %>% kable_styling(bootstrap_options = c("striped", "hover", "condens
```

	RF_Model_1	RF_Model_2
Accuracy	0.8000000	0.8181818
Sensitivity	0.4897959	0.4897959
Specificity	0.9310345	0.9568966
Pos Pred Value	0.7500000	0.8275862
Neg Pred Value	0.8120301	0.8161765
Precision	0.7500000	0.8275862
Recall	0.4897959	0.4897959
F1	0.5925926	0.6153846
Prevalence	0.2969697	0.2969697
Detection Rate	0.1454545	0.1454545
Detection Prevalence	0.1939394	0.1757576
Balanced Accuracy	0.7104152	0.7233462

The models share the same Sensitivity, Recall, Prevalence, and Detection Rates, but for every other metric

RF Model 2 outperforms RF Model 1. Based on the output statistics above **RF Model 2 is superior to RF Model 1** and will serve as our chosen Random Forest model. This is a noteworthy result being that more features are typically better and we elected the model with fewer features.

For now, we note the Random Forest model's performance based on these classification metrics. Later, we'll provide an in-depth interpretation of these statistics vs. those of our Decision Tree and Gradient Boosting models.

4. Gradient Boosting

Boosting is a method for creating an ensemble, or a combination of machine learning models. *Gradient* boosting relies on the belief that the best possible next model, when combined with prior models, reduces how often we misclassify our data.

It's heralded as one of the most powerful techniques for building predictive models and offers numerous options for implementation. We'll make use of the **XGBoost** (eXtreme Gradient Boosting) distributed gradient library due to its promise of flexibility, portability, and efficiency.

For our purposes, we're dealing with a categorical dependent variable, and will build a Gradient Boosting model from the loan approval dataset to provide `Loan_Status` predictions (ie. Approve or Reject).

To prepare our data for Gradient Boost modeling, we verify our numeric variable's skewness, perform a **log()** transform on each variable, and then re-verify the skewness to observe whether or not there was improvement:

```r
#Pre-process data: normalize independent numeric vars

#store sum of ApplicantIncome and CoapplicantIncome in ApplicantIncome
loan$ApplicantIncome <- loan$ApplicantIncome + loan$CoapplicantIncome

#calculate skewness prior to transformation
skewness(loan$ApplicantIncome, na.rm = TRUE)
```

```
## [1] 5.605953
```

```r
#skewness(loan$CoapplicantIncome, na.rm = TRUE)
skewness(loan$LoanAmount, na.rm = TRUE)
```

```
## [1] 2.685564
```

```r
#transformation: account for outliers with log transform
loan$ApplicantIncome <- log10(loan$ApplicantIncome)
#loan$CoapplicantIncome <- log10(loan$CoapplicantIncome)
loan$LoanAmount <- log10(loan$LoanAmount)

#calculate skewness after transformation
skewness(loan$ApplicantIncome, na.rm = TRUE)
```

```
## [1] 1.071447
```

```r
#skewness(loan$CoapplicantIncome, na.rm = TRUE) #produced NaN - dealt with via summing
skewness(loan$LoanAmount, na.rm = TRUE)
```

```
## [1] -0.1519333
```

After encountering issues transforming `CoapplicantIncome` and realizing a simplification may be in store, the sum of `ApplicantIncome` and `CoapplicantIncome` was stored as one in `ApplicantIncome`.

Based on the improved skewness values, we see that our transformation was a success and these (2) independent, numeric variables are now prepared to be included in our Gradient Boosting model.

As we continue processing our data, we drop impertinent variables, convert categorical variables to a binary scale, and convert our dependent variable to type factor:

```
#Data pre-processing: drop impertinent variables, convert to numeric, and change variable values
loan_GB <- subset(loan, select = -c(Gender, Property_Area, CoapplicantIncome)) #since CoapplicantIncome

# Convert Dependents, Credit_History numeric type
loan_GB$Dependents <- as.numeric(loan_GB$Dependents)
loan_GB$Credit_History <- as.numeric(loan_GB$Credit_History)

# Change Variables values
#loan_GB$Gender <- ifelse(loan_GB$Gender=="Male", 1, 0)
loan_GB$Education <- ifelse(loan_GB$Education=="Graduate", 1, 0)
loan_GB$Married <- ifelse(loan_GB$Married=="Yes", 1, 0)
loan_GB$Self_Employed <- ifelse(loan_GB$Self_Employed=="Yes", 1, 0)

#3. convert Loan_Status to type factor
loan_GB$Loan_Status <- ifelse(loan_GB$Loan_Status=="Y", 1, 0)
#loan$Loan_Status <- as.factor(loan$Loan_Status)

#head(loan_GB) #verify all numeric inputs
```

At this point pre-processing of our data is complete and we've verified that we will indeed be feeding our model all numeric inputs.

GB Model 1

We use the built-in **sample** function to perform a train-test split *with replacement* (75% training, 25% testing) and then convert our dataframes to matrices (with corresponding labels) as required to be modeled via **XGBoost**:

```
#Partition data
set.seed(1234)

#loan_GB
ind <- sample(2, nrow(loan_GB), replace = TRUE, prob = c(0.75, 0.25))
train_GB <- loan_GB[ind == 1, ]
test_GB <- loan_GB[ind == 2, ]

#dim(train_GB) #457 x 10
#dim(test_GB) #157 x 10

#Create train, test matrices - one hot encoding for factor variables
trainm <- sparse.model.matrix(Loan_Status ~ ., data = train_GB)
#head(trainm)
```

```
train_label <- train_GB[,"Loan_Status"]
train_matrix <- xgb.DMatrix(data = as.matrix(trainm),label = train_label )

testm <- sparse.model.matrix(Loan_Status ~ ., data = test_GB)
test_label <- test_GB[,"Loan_Status"]
test_matrix <- xgb.DMatrix(data = as.matrix(testm),label = test_label )
```

We specify our parameters, create our first model with all variables except `Gender` and `Dependents` (due to low correlation with `Loan_Status`), optimize `nrounds` (model parameter) for the lowest corresponding mlogloss rate, optimize `eta` (model parameter) for greatest perceived accuracy, visualize corresponding Confusion Matrices and misclassification error rates, and observe feature importance scores for our first model:

```
#Parameters
nc <- length(unique(train_label)) #number of classes
xgb_params <- list("objective" = "multi:softprob",
                   "eval_metric" = "mlogloss",
                   "num_class" = nc)
watchlist <- list(train = train_matrix, test = test_matrix)

#extreme Gradient Boosting Model
GB_model <- xgb.train(params = xgb_params,
                      data = train_matrix,
                      nrounds = 5, #run 100 iterations 1st then update based on test error value
                      watchlist = watchlist,
                      eta = 0.1, seed = 186
                      ) #inc eta value increased accuracy by 1
```

```
## [1]  train-mlogloss:0.645956 test-mlogloss:0.653761
## [2]  train-mlogloss:0.605647 test-mlogloss:0.619328
## [3]  train-mlogloss:0.572513 test-mlogloss:0.590422
## [4]  train-mlogloss:0.543785 test-mlogloss:0.565281
## [5]  train-mlogloss:0.518361 test-mlogloss:0.544616
```

```
#error plot
#e <- data.frame(GB_model$evaluation_log)
#plot(e$iter, e$train_mlogloss)
#lines(e$iter, e$test_mlogloss, col = 'red')

#determine when test error was lowest
#min(e$test_mlogloss) #0.456353 lowest error
#e[e$test_mlogloss == 0.456353,] #5th iteration

#prediction and confusion matrix from TRAIN data
p_train <- predict(GB_model, newdata = train_matrix)
pred_train <- matrix(p_train, nrow = nc, ncol = length(p_train)/nc) %>%
    t() %>% #matrix transpose
    data.frame() %>%
    mutate(label = train_label, max_prob = max.col(.,"last")-1)

tab_train <- table(Prediction = pred_train$max_prob, Actual = pred_train$label)
print(tab_train)
```

```
##           Actual
## Prediction   0   1
##          0  98   9
##          1  59 291
```

```r
print(paste('Misclassification Error with Train data', round(1 - sum(diag(tab_train))/sum(tab_train),3)
```

```
## [1] "Misclassification Error with Train data 0.149"
```

```r
#prediction and confusion matrix from TEST data
p_test <- predict(GB_model, newdata = test_matrix)
pred_test <- matrix(p_test, nrow = nc, ncol = length(p_test)/nc) %>%
    t() %>% #matrix transpose
    data.frame() %>%
    mutate(label = test_label, max_prob = max.col(.,"last")-1)

tab_test <- table(Prediction = pred_test$max_prob, Actual = pred_test$label)
print(tab_test)
```

```
##           Actual
## Prediction   0   1
##          0  16   5
##          1  19 117
```

```r
print(paste('Misclassification Error with Test data', round(1 - sum(diag(tab_test))/sum(tab_test),3)))
```

```
## [1] "Misclassification Error with Test data 0.153"
```

```r
#feature importance
imp <- xgb.importance(colnames(train_matrix), model=GB_model)
print(imp) #higher Gain means higher feature importance
```

```
##              Feature        Gain       Cover  Frequency
## 1:    Credit_History 0.679633852 0.191400894 0.06172840
## 2:   ApplicantIncome 0.144227526 0.307106444 0.35802469
## 3:        LoanAmount 0.115415220 0.328791702 0.27160494
## 4: Loan_Amount_Term 0.016045089 0.090222048 0.06172840
## 5:           Married 0.015001176 0.011367778 0.07407407
## 6:        Dependents 0.012202129 0.019761692 0.03703704
## 7:         Education 0.008809010 0.008240728 0.08641975
## 8:     Self_Employed 0.008665998 0.043108714 0.04938272
```

We note a train-mlogloss (multiclass log loss) value of 0.518361 on five iterations, training data misclassification error rate of 14.9%, and testing misclassification error rate of 15.3%. Later, we'll compare the output statistics of the confusion matrix with those of the second model to determine which has a greater predictive accuracy.

From the above feature importance scores, we observe that `Credit_History`, `ApplicantIncome`, and `LoanAmount` have the highest `Gain` scores. We'll omit all other features and utilize just these features to make up our second model.

Recall: `ApplicantIncome` *is now representative of* `ApplicantIncome` *and* `CoapplicantIncome`*.*

GB Model 2

We create the second model in the same manner as the first model. This time we only consider important factors such as `Credit_History`, `Applicant_Income`, and `Loan_Amount`:

```
#Create train, test matrices - one hot encoding for factor variables
trainm2 <- sparse.model.matrix(Loan_Status ~ Credit_History + LoanAmount + ApplicantIncome, data = trai
#head(trainm2)
train_label2 <- train_GB[,"Loan_Status"]
train_matrix2 <- xgb.DMatrix(data = as.matrix(trainm2),label = train_label2 )

testm2 <- sparse.model.matrix(Loan_Status ~ Credit_History + LoanAmount + ApplicantIncome, data = test_
test_label2 <- test_GB[,"Loan_Status"]
test_matrix2 <- xgb.DMatrix(data = as.matrix(testm2),label = test_label2 )

#Parameters
nc2 <- length(unique(train_label2)) #number of classes
xgb_params2 <- list("objective" = "multi:softprob",
                    "eval_metric" = "mlogloss",
                    "num_class" = nc2)
watchlist2 <- list(train = train_matrix2, test = test_matrix2)

#extreme Gradient Boosting Model
GB_model2 <- xgb.train(params = xgb_params2,
                       data = train_matrix2,
                       nrounds = 20, #run 100 iterations 1st then update based on test error value
                       watchlist = watchlist2,
                       eta = 0.1, seed = 606
                       ) #inc eta value increased accuracy by 1
```

```
## [1]  train-mlogloss:0.645587 test-mlogloss:0.654836
## [2]  train-mlogloss:0.605604 test-mlogloss:0.624365
## [3]  train-mlogloss:0.572721 test-mlogloss:0.595989
## [4]  train-mlogloss:0.544159 test-mlogloss:0.574349
## [5]  train-mlogloss:0.520744 test-mlogloss:0.554515
## [6]  train-mlogloss:0.498525 test-mlogloss:0.539389
## [7]  train-mlogloss:0.480195 test-mlogloss:0.527599
## [8]  train-mlogloss:0.462705 test-mlogloss:0.516761
## [9]  train-mlogloss:0.447150 test-mlogloss:0.509267
## [10] train-mlogloss:0.433480 test-mlogloss:0.502473
## [11] train-mlogloss:0.421272 test-mlogloss:0.496662
## [12] train-mlogloss:0.410621 test-mlogloss:0.491445
## [13] train-mlogloss:0.398767 test-mlogloss:0.487974
## [14] train-mlogloss:0.389052 test-mlogloss:0.484808
## [15] train-mlogloss:0.380588 test-mlogloss:0.483188
## [16] train-mlogloss:0.374175 test-mlogloss:0.482234
## [17] train-mlogloss:0.367937 test-mlogloss:0.479899
## [18] train-mlogloss:0.362150 test-mlogloss:0.479489
## [19] train-mlogloss:0.356925 test-mlogloss:0.478269
## [20] train-mlogloss:0.352221 test-mlogloss:0.477973
```

```
#error plot
#e2 <- data.frame(GB_model2$evaluation_log)
```

```r
#plot(e2$iter, e2$train_mlogloss)
#lines(e2$iter, e2$test_mlogloss, col = 'red')

#determine when test error was lowest
#min(e2$test_mlogloss) #0.478216 lowest error
#e2[e2$test_mlogloss == 0.478216,] #20th iteration

#prediction and confusion matrix from train data
p_train2 <- predict(GB_model2, newdata = train_matrix2)
pred_train2 <- matrix(p_train2, nrow = nc2, ncol = length(p_train2)/nc2) %>%
    t() %>% #matrix transpose
    data.frame() %>%
    mutate(label = train_label2, max_prob = max.col(.,"last")-1)

tab_train2 <- table(Prediction = pred_train2$max_prob, Actual = pred_train2$label)
print(tab_train2)
```

```
##             Actual
## Prediction   0   1
##          0  97   5
##          1  60 295
```

```r
print(paste('Misclassification Error with Train data', round(1 - sum(diag(tab_train2))/sum(tab_train2),
```

```
## [1] "Misclassification Error with Train data 0.142"
```

```r
#prediction and confusion matrix from test data
p_test2 <- predict(GB_model2, newdata = test_matrix2)
pred_test2 <- matrix(p_test2, nrow = nc2, ncol = length(p_test2)/nc2) %>%
    t() %>% #matrix transpose
    data.frame() %>%
    mutate(label = test_label2, max_prob = max.col(.,"last")-1)

tab_test2 <- table(Prediction = pred_test2$max_prob, Actual = pred_test2$label)
print(tab_test2)
```

```
##             Actual
## Prediction   0   1
##          0  17   9
##          1  18 113
```

```r
print(paste('Misclassification Error with Test data', round(1 - sum(diag(tab_test2))/sum(tab_test2),3))
```

```
## [1] "Misclassification Error with Test data 0.172"
```

We note a train-mlogloss (multiclass log loss) value of 0.352221 on twenty iterations, training data misclassification error rate of 14.2%, and testing misclassification error rate of 17.2%.

Next, we'll put our models side-by-side to determine which has a greater predictive accuracy.

Model Comparison

We evaluate our models by applying **GB Model 1** and **GB Model 2** to the test data set. We utilize the **confusionMatrix** function from the caret library and present our statistics as a kable table to glean more insight regarding the comparative statistics between each model's performance:

```
AccuracyGB1 <- confusionMatrix(factor(pred_test$max_prob),factor(pred_test$label))$overall['Accuracy']
AccuracyGB2 <- confusionMatrix(factor(pred_test2$max_prob),factor(pred_test2$label))$overall['Accuracy'

GB_Model_1 <- confusionMatrix(factor(pred_test$max_prob),factor(pred_test$label))$byClass
GB_Model_1 <- data.frame(GB_Model_1)
GB_Model_1 <- rbind("Accuracy" = AccuracyGB1, GB_Model_1)

GB_Model_2 <- confusionMatrix(factor(pred_test2$max_prob),factor(pred_test2$label))$byClass
GB_Model_2 <- data.frame(GB_Model_2)
GB_Model_2 <- rbind("Accuracy" = AccuracyGB2, GB_Model_2)

tabularview <- data.frame(GB_Model_1, GB_Model_2)

tabularview %>%  kableExtra::kbl() %>% kable_styling(bootstrap_options = c("striped", "hover", "condens
```

	GB_Model_1	GB_Model_2
Accuracy	0.8471338	0.8280255
Sensitivity	0.4571429	0.4857143
Specificity	0.9590164	0.9262295
Pos Pred Value	0.7619048	0.6538462
Neg Pred Value	0.8602941	0.8625954
Precision	0.7619048	0.6538462
Recall	0.4571429	0.4857143
F1	0.5714286	0.5573770
Prevalence	0.2229299	0.2229299
Detection Rate	0.1019108	0.1082803
Detection Prevalence	0.1337580	0.1656051
Balanced Accuracy	0.7080796	0.7059719

The models share the same Sensitivity, Recall, Prevalence, and Detection Rates, but for every other metric GB Model 1 outperforms GB Model 2. Based on the output statistics above **GB Model 1 is superior to GB Model 2** and will serve as our chosen Gradient Boosting model.

5. Overall Model Comparison

We put our strongest Decision Tree, Random Forest, and Gradient Boosting models side-by-side-by-side to interpret their common classification metrics and determine which has the greatest predictive accuracy.

We consider the following classification metrics in consulting each model's Confusion Matrix:

- **Accuracy** : $\frac{TP+TN}{TP+FP+TN+FN}$
- **Sensitivity (Recall)** : true positive rate. $\frac{TP}{TP+FN}$
- **Specificity**: true negative rate. $\frac{TN}{TN+FP}$
- **Pos Pred Value (Precision)** : probability that predicted positive is truly positive. $\frac{TP}{TP+FP}$
- **Neg Pred Value**: probability that predicted negative is truly negative. $\frac{TN}{(TN+FN)}$
- **F1**: harmonic mean of model's precision and recall. $\frac{2*(Precision*Recall)}{Precision+Recall}$
- **Prevalence**: truly positive observations as proportion of total number of observations. $\frac{TP+FN}{TP+FP+FN+TN}$
- **Detection Rate**: true positives detected as proportion of entire total population. $\frac{TP}{TP+FP+FN+TN}$
- **Detection Prevalence**: predicted positive events over total number of predictions. $\frac{TP+FP}{TP+FP+FN+TN}$
- **Balanced Accuracy**: measure of model's that is especially useful when classes are imbalanced. $\frac{Sensitivity+Specificity}{2}$

These models are all applied to the test data set and make use of the **confusionMatrix** function from the caret library. We present the corresponding common classification metrics as a kable table to glean more insight regarding the relative strength of each model's performance:

```r
#Present all model statistics on the same graphic
tabularview <- data.frame(DT_Model, RF_Model_2, GB_Model_1)

tabularview %>%  kableExtra::kbl() %>% kable_styling(bootstrap_options = c("striped", "hover", "condens
```

	DT_Model	RF_Model_2	GB_Model_1
Accuracy	0.8023952	0.8181818	0.8471338
Sensitivity	0.4363636	0.4897959	0.4571429
Specificity	0.9821429	0.9568966	0.9590164
Pos Pred Value	0.9230769	0.8275862	0.7619048
Neg Pred Value	0.7801418	0.8161765	0.8602941
Precision	0.9230769	0.8275862	0.7619048
Recall	0.4363636	0.4897959	0.4571429
F1	0.5925926	0.6153846	0.5714286
Prevalence	0.3293413	0.2969697	0.2229299
Detection Rate	0.1437126	0.1454545	0.1019108
Detection Prevalence	0.1556886	0.1757576	0.1337580
Balanced Accuracy	0.7092532	0.7233462	0.7080796

Consulting the above output table, we observe that **DT Model** has the strongest performance for `Specificity`, `Pos Pred Value`, `Precision`, and `F1` whereas **GB Model 1** has the strongest performance for `Sensitivity`, `Neg Pred Value` and `Balanced Accuracy`. Thus, **DT Model** performs better when predicting loan approvals while **GB Model 1** performs better when predicting loan rejections.

Analysis

Decision tree's often raise concerns regarding over-fitting, bias and variance error because of their simplicity, and random forests are meant to address these concerns by accounting for a collection of decision trees to come to a single, aggregated result. We found it surprising that the decision tree model outperformed the random forest model. This may have been because of how we implemented the model or it may have simply been a poor situation for random forests.

Where decision trees are lauded for ease of interpretability and speed and random forests are lauded for how they handle noise, gradient boosting is known to handle unbalanced data the best. **We would recommend gradient boosting** for the simple fact that we're dealing with imbalanced classes (recall 192 N's, 422 Y's) and **GB Model 1** is the best performing model across the board when this is taken into account.

…………………………………………………………………………..

References

In completing this assignment, reference was made to the following:

- Geeks for Geeks. (2020). **K-NN Classifier in R Programming** [article]. Retrieved from: https://www.geeksforgeeks.org/k-nn-classifier-in-r-programming/
- Guru 99. **Decision Tree in R** [article]. Retrieved from: https://guru99.com/r-decision-trees.html/
- Geeks for Geeks. (2020). **Decision Tree for Regression in R Programming** [article]. Retrieved from: https://www.geeksforgeeks.org/decision-tree-for-regression-in-r-programming/
- Data Flair. **R Decision Trees** [article]. Retrieved from: https://data-flair.training/blogs/r-decision-trees/
- Statology. **How to Build Random Forests in R** [article]. Retrieved from: https://statology.org/random-forest-in-r/
- Data Novia. **Transform Data to Normal Distribution in R** [article]. Retrieved from: https://www.datanovia.com/en/lessons/transform-data-to-normal-distribution-in-r/
- Dario Radecic. (2021). **Gradient Boosting with R** [article]. Retrieved from: https://appsilon.com/r-xgboost/
- ZXS107020. (2018). **Multi-Class Classification Using XGBOOST** [article]. Retrieved from: http://rstudio-pubs-static.s3.amazonaws.com/368478_bf9700befeba4283a4640a9a1285af22.html
- Bharatendra Rai. (2017). **eXtreme Gradient Boosting XGBoost Algorithm with R** [video]. Retrieved from: https://www.youtube.com/watch?v=woVTNwRrFHE
- Christian Thieme. (2021). **Understanding Common Classification Metrics** [article]. Retrieved from: https://towardsdatascience.com/understanding-common-classification-metrics-titanic-style-8b8a562d3e32