Introduction

Building classification models is one of the most important data science use cases. *Classification models* are models that predict a categorical label. A few examples of this include predicting whether a customer will churn or whether a bank loan will default. In this guide, you will learn how to build and evaluate a classification model in R. We will train the logistic regression algorithm, which is one of the oldest yet most powerful classification algorithms.

0. Data

In this exercise, we will use a fictitious dataset of loan applicants containing about 614 observations and 12 variables, as described below:

1. `Gender`: Whether the applicant is a male ("Male") or a female ("Female")
2. `Marital_status`: Whether the applicant is married or not ("Yes") or not ("No")
3. `Dependent`: Total number of dependents
4. `Education`: Whether the applicant is a graduate (“Graduate”) or not (“Not Graduate”)
5. `Self_employed`: Whether the applicant is a self-employed ("Yes") or not (“No”)
6. `ApplicantIncome`: Monthly Income of the applicant (in USD)
7. `CoapplicantIncome:` Monthly Income of the coapplicant (in USD)
8. `Loan_amount`: Loan amount (in USD) for which the application was submitted
9. `Loan_amount_term:` Terms of the loan in months
10. `Credit_history`: Whether the applicant has a credit history ("1") or not ("0")
11. `Property_area`: Where property is located – rural (“Rural”), semiurban (“Semiurban”) or urban ("Urban")
12. `Loan_status`: Whether the loan application was approved ("Y") or not ("N")

Let's start by loading the required libraries and the data.

```
library(readr)
library(dplyr)

dat <- read_csv("data.csv") #this is path to where your “data.csv” file is
glimpse(dat)
```

Output:

```
Rows: 614
Columns: 12
$ Gender            <fct> Male, Male, Male, Male, Male, Male, Male, Male, Male, Male…
$ Marital_status    <fct> No, Yes, Yes, Yes, No, Yes, Yes, Yes, Yes, Yes, Yes, Yes, …
$ Dependents        <fct> 0, 1, 0, 0, 0, 2, 0, 3+, 2, 1, 2, 2, 2, 0, 2, 0, 1, 0, 0, …
$ Education         <fct> Graduate, Graduate, Graduate, Not Graduate, Graduate, Grad…
$ Self_employed     <fct> No, No, Yes, No, No, Yes, No, No, No, No, No, No, No, No, …
$ ApplicantIncome   <int> 5849, 4583, 3000, 2583, 6000, 5417, 2333, 3036, 4006, 1284…
$ CoapplicantIncome <dbl> 0, 1508, 0, 2358, 0, 4196, 1516, 2504, 1526, 10968, 700, 1…
$ Loan_amount       <int> 230, 128, 66, 120, 141, 267, 95, 158, 168, 349, 70, 109, 2…
$ Loan_amount_term  <int> 360, 360, 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, 1, 0, 1, 1…
$ Property_area      <fct> Urban, Rural, Urban, Urban, Urban, Urban, Urban, Semiurban…
$ Loan_status        <fct> Y, N, Y, Y, Y, Y, Y, N, Y, N, Y, Y, Y, N, Y, Y, Y, N, N, Y…
```

The output shows that the dataset has four numerical (labeled as int). If the other six character variables are labeled as chr, we will convert these into factor variables using the line of code below.

```
names <- c(1,2,3,4,5,11,12)
dat[,names] <-lapply(dat[,names], factor)
glimpse(dat)
```

This should now show as <fct>

We will proceed as follow:

- Step 1: Check continuous variables
- Step 2: Check factor variables
- Step 3: Summary statistic
- Step 4: Train/test set
- Step 5: Build the model
- Step 6: Assess the performance of the model

1. Step 1) Check continuous variables

In the first step, you can see the distribution of the continuous variables.

```
continuous <-select_if(dat, is.numeric)
summary(continuous)
```

Code Explanation

- continuous <- select_if(dat, is.numeric): Use the function select_if() from the dplyr library to select only the numerical columns
- summary(continuous): Print the summary statistic

Output:

```
ApplicantIncome CoapplicantIncome  Loan_amount    Loan_amount_term Credit_history
 Min.   :  150   Min.   :    0     Min.   :  9.0   Min.   : 12.0    Min.   :0.0000
 1st Qu.: 2884   1st Qu.:    0     1st Qu.:100.0   1st Qu.:360.0    1st Qu.:1.0000
 Median : 3814   Median : 1149     Median :128.5   Median :360.0    Median :1.0000
 Mean   : 5408   Mean   : 1618     Mean   :148.8   Mean   :341.2    Mean   :0.8355
 3rd Qu.: 5795   3rd Qu.: 2297     3rd Qu.:172.0   3rd Qu.:360.0    3rd Qu.:1.0000
 Max.   :81000   Max.   :41667     Max.   :700.0   Max.   :480.0    Max.   :1.0000
```

From the above table, you can see that the data have totally different scales. `ApplicantIncome` & `CoapplicantIncome` have large outliers (i.e., look at the last quartile and maximum value).

You can deal with it following two steps:

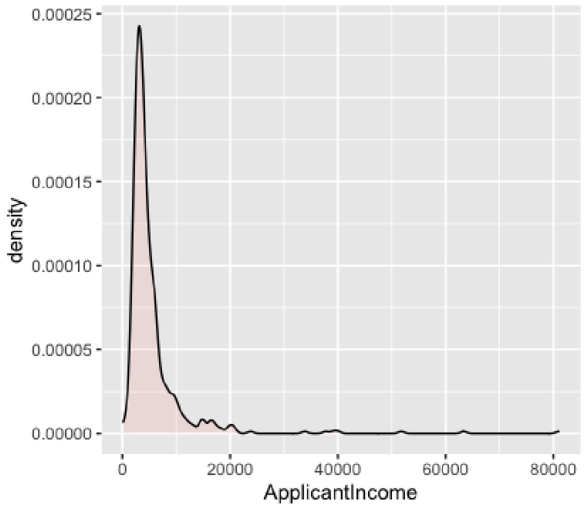
- 1: Plot the distribution of the variables with the outliers (`ApplicantIncome` & `CoapplicantIncome`)
- 2: Standardize the continuous variables

1. Plot the distribution

Let’s look closer at the distribution of `ApplicantIncome`

```
# Histogram with kernel density curve
library(ggplot2)
ggplot(continuous, aes(x = ApplicantIncome)) +
    geom_density(alpha = .2, fill = "#FF6666")
```

Output:



The variable has some outliers. You can partially tackle this problem by deleting the top 0.02 percent of the ApplicantIncome.

Basic syntax of quantile:

```
quantile(variable, percentile)
arguments:
-variable:  Select the variable in the data frame to compute the percentile
-percentile:  Can be a single value between 0 and 1 or multiple value. If multiple,
use this format:  `c(A,B,C, ...)
- `A`,`B`,`C` and `...` are all integer from 0 to 1.
```

We compute the top 2 percent percentile

```
applicant_income <- quantile(dat$ApplicantIncome, .98)
applicant_income
```

Code Explanation

- quantile(dat$ApplicantIncome, .98): Compute the value of the 98 percent of the income time

Output:

```
     98%
19666.04
```

98 percent of the population makes under $19666.04 per month.

You can drop the observations above this threshold. You use the filter from the dplyr library.

```
dat_drop <- dat %>%
  filter(ApplicantIncome<applicant_income)
dim(dat_drop)
```

Output:

```
[1] 601   12
```

Compare this with the original “dat”. Observe that some rows have been dropped.

2. Standardize the continuous variables

You can standardize each column to improve the performance because your data do not have the same scale. You can use the function mutate_if from the dplyr library. The basic syntax is:

```
mutate_if(df, condition, funs(function))
arguments:
-`df`: Data frame used to compute the function
- `condition`: Statement used. Do not use parenthesis
- funs(function):  Return the function to apply. Do not use parenthesis for the
function
```

You can standardize the numeric columns as follow:

```
 dat_rescale <- dat_drop %>%
     mutate_if(is.numeric, list(~as.numeric(scale(.))))
head(dat_rescale)
```

Code Explanation

- mutate_if(is.numeric, funs(scale)): The condition is only numeric column and the function is scale

Output:

```
  Gender Marital_status Dependents    Education Self_employed
1   Male             No          0     Graduate            No
2   Male            Yes          1     Graduate            No
3   Male            Yes          0     Graduate           Yes
4   Male            Yes          0 Not Graduate            No
5   Male             No          0     Graduate            No
6   Male            Yes          2     Graduate           Yes
  ApplicantIncome CoapplicantIncome Loan_amount Loan_amount_term
1       0.3586198       -0.55326660  1.23072033        0.2875715
2      -0.0506540       -0.03987195 -0.20246263        0.2875715
3      -0.5624079       -0.55326660 -1.07361306        0.2875715
4      -0.6972161        0.24950832 -0.31486914        0.2875715
5       0.4074353       -0.55326660 -0.01980206        0.2875715
6       0.2189624        0.87525061  1.75060042        0.2875715
  Credit_history Property_area Loan_status
1       0.443715         Urban           Y
2       0.443715         Rural           N
3       0.443715         Urban           Y
4       0.443715         Urban           Y
5       0.443715         Urban           Y
6       0.443715         Urban           Y
```

2. Step 2) Check factor variables

This step has two objectives:

- Check the level in each categorical column
- Define new levels

We will divide this step into three parts:

- Select the categorical columns
- Store the bar chart of each column in a list
- Print the graphs

We can select the factor columns with the code below:

```
# Select categorical column
factor <- data.frame(select_if(dat_rescale, is.factor))
     ncol(factor)
```

Code Explanation

data.frame(select_if(dat_rescale, is.factor)): We store the factor columns in factor in a data frame type. The library ggplot2 requires a data frame object.

Output:

```
##  [1]   7
```

The dataset contains 7 categorical variables

The second step is more skilled. You want to plot a bar chart for each column in the data frame factor. It is more convenient to automate the process, especially in situations where there are lots of columns.

```
library(ggplot2)
# Create graph for each column
graph <- lapply(names(factor),
    function(x)
     ggplot(factor, aes(get(x))) +
          geom_bar() +
          theme(axis.text.x = element_text(angle = 90)))
```

Code Explanation

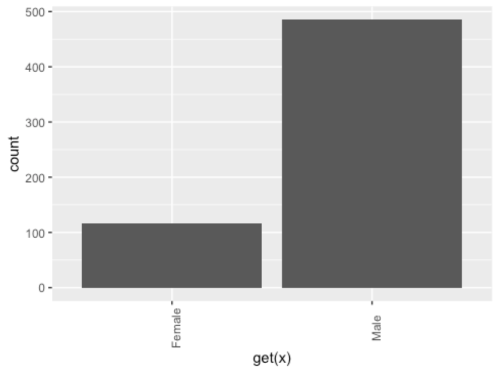
- lapply(): Use the function lapply() to pass a function in all the columns of the dataset. You store the output in a list
- function(x): The function will be processed for each x. Here x is the columns
- ggplot(factor, aes(get(x))) + geom_bar()+ theme(axis.text.x = element_text(angle = 90)): Create a bar char chart for each x element. Note, to return x as a column, you need to include it inside the get()

The last step is relatively easy. You want to print the 7 graphs.

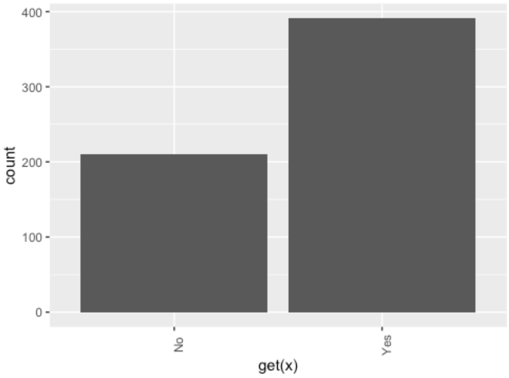
```
# Print the graph
Graph
```

Output:

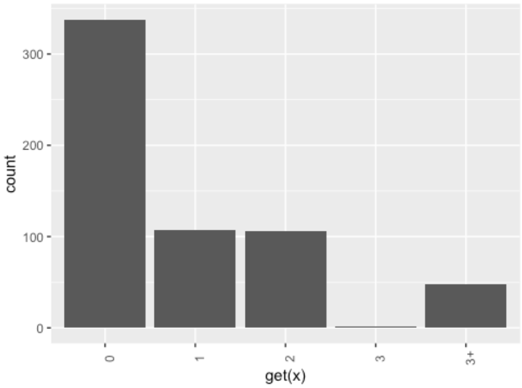
```
## [[1]]
```



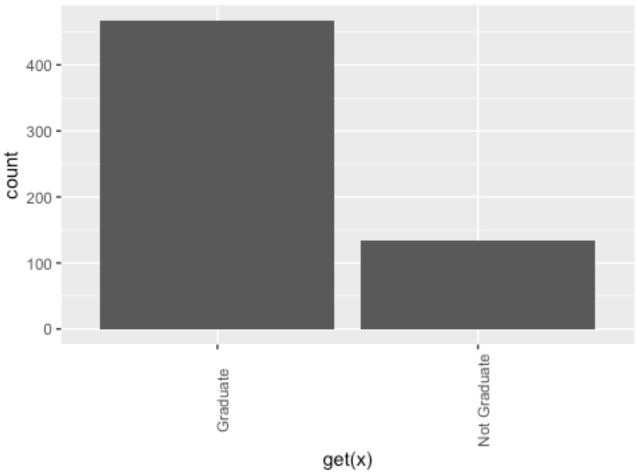
```
## ## [[2]]
```


400
300
200
100
0
count
No
Yes
get(x)

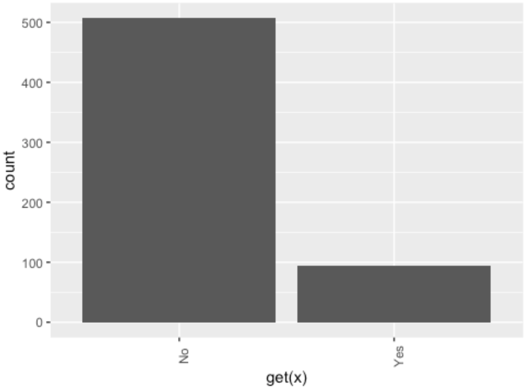
```
## ## [[3]]
```


300
200
100
0
count
0
1
2
3
3+
get(x)

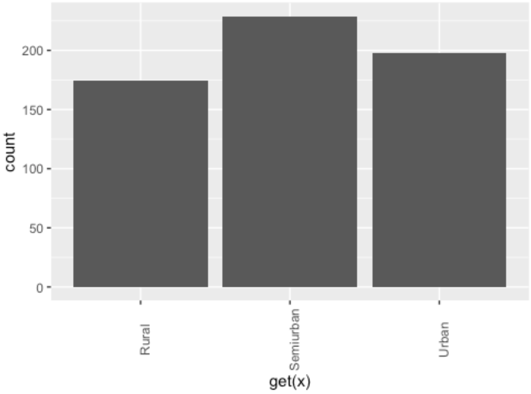
```
## ## [[4]]
```


400
300
200
100
0
count
Graduate
Not Graduate
get(x)

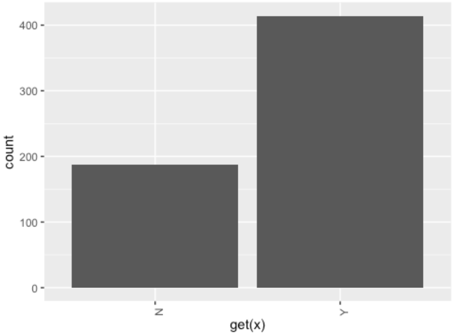
```
## ## [[5]]
```


count
500
400
300
200
100
0
No
Yes
get(x)

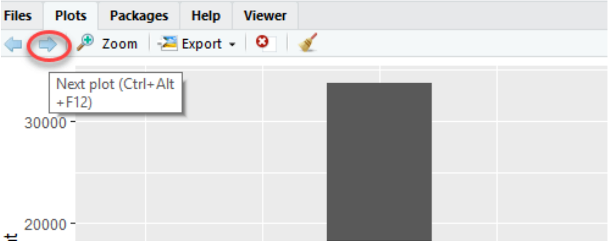
```
## ## [[6]]
```


count
200
150
100
50
0
Rural
Semiurban
Urban
get(x)

```
## ## [[7]]
```


count
400
300
200
100
0
N
Y
get(x)

Note: Use the next button to navigate to the next graph



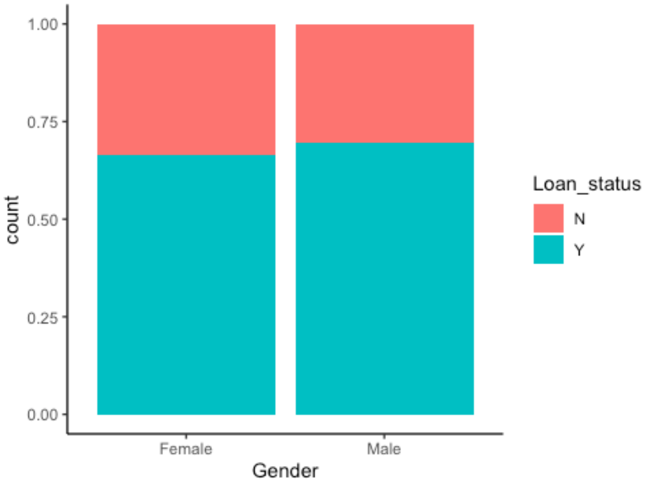
Next plot (Ctrl+Alt +F12)

3. Step 3) Summary Statistic

It is time to check some statistics about our target variables. In the graph below, you count the percentage of individuals with loan approval given their gender.

```
# Plot gender income
ggplot(dat_rescale, aes(x = Gender, fill = Loan_status)) +
    geom_bar(position = "fill") +
    theme_classic()
```

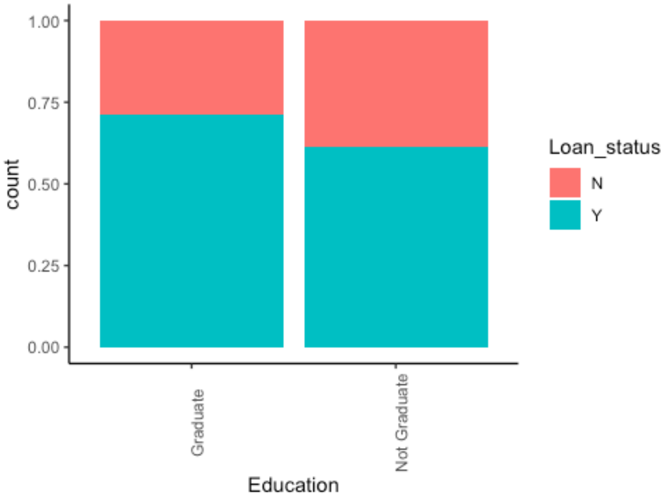
Output:



Next, check if the level of education affects their loan approval.

```
# Plot Education Loan_status
ggplot(dat_rescale, aes(x = Education, fill = Loan_status)) +
    geom_bar(position = "fill") +
    theme_classic() +
    theme(axis.text.x = element_text(angle = 90))
```

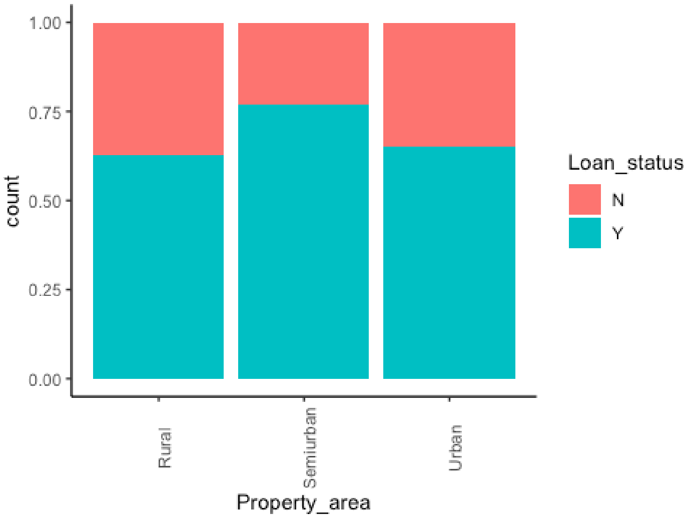
Output:



Next, check if the property area affects their loan approval.

```
# Plot Property_area Loan_status
ggplot(dat_rescale, aes(x = Property_area, fill = Loan_status))
+
    geom_bar(position = "fill") +
    theme_classic() +
    theme(axis.text.x = element_text(angle = 90))
```

Output:



Non-linearity

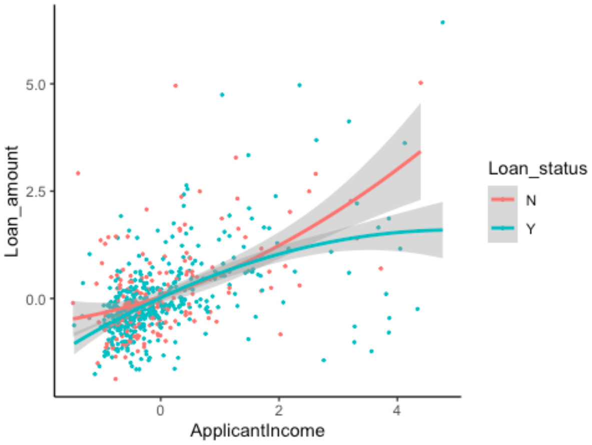
Before you run the model, you can see if the applicant income is related to loan amount.

```
ggplot(dat_rescale, aes(x = ApplicantIncome, y = Loan_amount))
+
    geom_point(aes(color = Loan_status),
        size = 0.5) +
    stat_smooth(method = 'lm',
        formula = y~poly(x, 2),
        se = TRUE,
        aes(color = Loan_status)) +
    theme_classic()
```

Code Explanation

- gplot(dat_rescale, aes(x = ApplicantIncome, y = Loan_amount)): Set the aesthetic of the graph
- geom_point(aes(color= income), size =0.5): Construct the dot plot
- stat_smooth(): Add the trend line with the following arguments:
    - method='lm': Plot the fitted value if the linear regression
    - formula = y~poly(x,2): Fit a polynomial regression
    - se = TRUE: Add the standard error
    - aes(color= income): Break the model by income

Output:



In a nutshell, you can test interaction terms in the model to pick up the non-linearity effect between the applicant income and other features. It is important to detect under which condition the applicant income differs.

Correlation

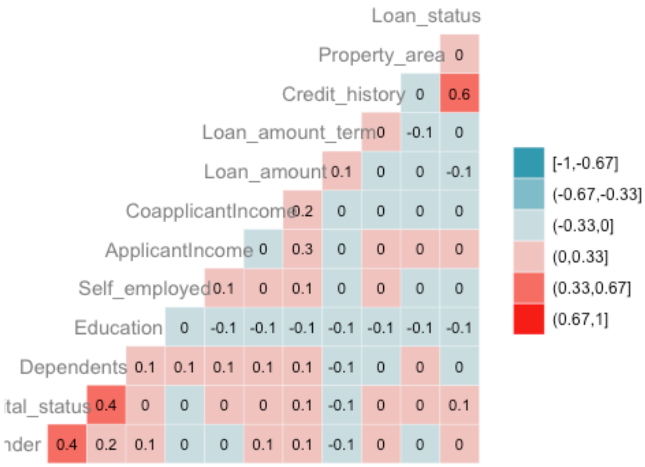
The next check is to visualize the correlation between the variables. You convert the factor level type to numeric so that you can plot a heat map containing the coefficient of correlation computed with the Spearman method.

```
library(GGally)
# Convert data to numeric
corr <- data.frame(lapply(dat_rescale, as.integer))
# Plot the graph
ggcorr(corr,
method = c("pairwise", "spearman"),
nbreaks = 6,
hjust = 0.8,
label = TRUE,
label_size = 3,
color = "grey50")
```

Code Explanation

- data.frame(lapply(dat_rescale, as.integer)): Convert data to numeric
- ggcorr() plot the heat map with the following arguments:
    - method: Method to compute the correlation
    - nbreaks = 6: Number of break
    - hjust = 0.8: Control position of the variable name in the plot
    - label = TRUE: Add labels in the center of the windows
    - label_size = 3: Size labels
    - color = “grey50”): Color of the label

Output:



4. Step 4) Train/test set

Any supervised machine learning task requires splitting the data between a train set and a test set.

```
library(caret)
set.seed(1234)

trainIndex <- createDataPartition(dat$Loan_status, p = 0.7,
list = FALSE, times = 1)

trainData <- dat[trainIndex,]
testData <- dat[-trainIndex,]

print(dim(trainData)); print(dim(testData))
```

Output:

```
[1]  431   12
[1]  183   12
```

Code Explanation:
The first line loads the `caret` package that will be used for data partitioning, The second line of code below sets the random seed for reproducibility of results. while the third to fifth lines create the training and test datasets. The train dataset contains 70 percent of the data (431 observations of 12 variables) while the test data contains the remaining 30 percent (183 observations of 12 variables).

5. Step 5) Build the model

To fit the logistic regression model, the first step is to instantiate the algorithm. This is done in the first line of code below with the `glm()` function. The second line prints the summary of the trained model.

```
model_glm = glm(Loan_status~., family="binomial", data =
trainData)
summary(model_glm)
```

Output:

```
Call:
glm(formula = Loan_status ~ ., family = "binomial", data = trainData)

Deviance Residuals: 
    Min       1Q   Median       3Q      Max  
-2.3449  -0.3804   0.4734   0.6766   2.4101  

Coefficients:
                         Estimate Std. Error z value
(Intercept)            -1.039e+00  9.628e-01  -1.080
GenderMale             -8.175e-02  3.664e-01  -0.223
Marital_statusYes       6.698e-01  3.187e-01   2.102
Dependents1            -8.481e-01  3.553e-01  -2.387
Dependents2            -9.540e-02  4.076e-01  -0.234
Dependents3             1.246e+01  8.827e+02   0.014
Dependents3+           -8.190e-02  5.634e-01  -0.145
EducationNot Graduate  -6.883e-01  3.160e-01  -2.178
Self_employedYes       -1.536e-01  3.551e-01  -0.433
ApplicantIncome         1.801e-05  2.770e-05   0.650
CoapplicantIncome      -4.750e-05  4.355e-05  -1.091
Loan_amount            -2.717e-03  1.750e-03  -1.553
Loan_amount_term       -3.702e-03  2.333e-03  -1.586
Credit_history          3.838e+00  4.278e-01   8.973
Property_areaSemiurban  8.824e-01  3.362e-01   2.625
Property_areaUrban      2.179e-01  3.345e-01   0.651
                       Pr(>|z|)    
(Intercept)             0.28036    
GenderMale              0.82346    
Marital_statusYes       0.03558 *  
Dependents1             0.01700 *  
Dependents2             0.81494    
Dependents3             0.98874    
Dependents3+            0.88443    
EducationNot Graduate   0.02940 *  
Self_employedYes        0.66536    
ApplicantIncome         0.51552    
CoapplicantIncome       0.27539    
Loan_amount             0.12047    
Loan_amount_term        0.11266    
Credit_history          < 2e-16 ***
Property_areaSemiurban  0.00868 ** 
Property_areaUrban      0.51473    
---
Signif. codes:  
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 535.87  on 430  degrees of freedom
Residual deviance: 366.22  on 415  degrees of freedom
AIC: 398.22

Number of Fisher Scoring iterations: 13
```

The significance code ‘***’ in the above output shows the relative importance of the feature variables. Let's evaluate the model further, starting by setting the baseline accuracy using the code below. Since the majority class of the target variable has a proportion of 0.68, the baseline accuracy is 68 percent.

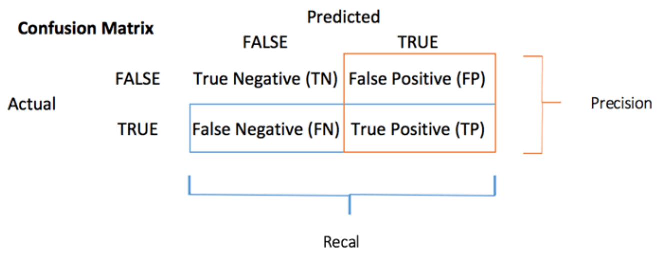
```
#Baseline Accuracy
prop.table(table(trainData$Loan_status))
```

Output:

```
        N         Y
0.3166667 0.6833333
```

6. Step 6) Assess the performance of the model

The confusion matrix is a better choice to evaluate the classification performance compared with the different metrics you saw before. The general idea is to count the number of times True instances are classified are False.



Confusion Matrix		Predicted	
		FALSE	TRUE
Actual	FALSE	True Negative (TN)	False Positive (FP)
	TRUE	False Negative (FN)	True Positive (TP)

Let's now evaluate the model performance on the training and test data, which should ideally be better than the baseline accuracy. We start by generating predictions on the training data, using the first line of code below. The second line creates the confusion matrix with a threshold of 0.5, which means that for probability predictions equal to or greater than 0.5, the algorithm will predict the `Y` response for the `Loan_status` variable. The third line prints the accuracy of the model on the training data, using the confusion matrix, and the accuracy comes out to be 81 percent.

We then repeat this process on the test data, and the accuracy comes out to be 82 percent.

```
# Predictions on the training set
predictTrain <- predict(model_glm, trainData, type = "response")

# Confusion matrix on training data
train_mat <- table(trainData$Loan_status, predictTrain >= 0.5)
train_mat


# Predictions on the test set
predictTest <- predict(model_glm, testData, type = "response")

# Confusion matrix on test data
```

```
test_mat <- table(testData$Loan_status, predictTest >= 0.5)
test_mat
```

Output:

```
# Confusion matrix and accuracy on training data

     FALSE TRUE
  N    73    62
  Y    15   281


# Confusion matrix and accuracy on testing data

     FALSE TRUE
  N     25   32
  Y      2  124
```

You can calculate the model accuracy by summing the true positive + true negative over the total observation

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

```
accuracy_train <- sum(diag(train_mat)) / sum(train_mat)
accuracy_train

accuracy_test <- sum(diag(test_mat)) / sum(test_mat)
accuracy_test
```

Code Explanation

- sum(diag(ttain_mat)): Sum of the diagonal
- sum(train_mat): Sum of the matrix.

Output:

```
[1]   0.8213457
[1]   0.8142077
```

7. Conclusion

In this guide, you have learned techniques of building a classification model in R using the powerful logistic regression algorithm. The baseline accuracy for the data was 68 percent, while the accuracy on the training and test data was 91 percent, and 88 percent, respectively. Overall, the logistic regression model is beating the baseline accuracy by a big margin on both the train and test datasets, and the results are very good.

Definitions:

- plyr – splitting, applying and combining
- dplyr – next iteration of plyr working with dataframes
- readr – read rectangular data like csv, tsv, etc
- caret – classification and regression training - data preparation, model building, and model evaluation
- ggplot2 - is a system for declaratively creating graphics, based on The Grammar of Graphics
- ggally - is designed to be a helper to ggplot2