K-Nearest Neigbors

Code

Taylor Shrode

Regis University MSDS 680

September 6, 2020

Introduction

One simple supervised machine learning algorithm that is used for regression and classification is the *K*-Nearest Neighbors (KNN) algorithm (Harrison, 2019). A supervised machine learning algorithm is an algorithm that uses labeled input data to learn functions and produce appropriate outputs when given unlabeled data (Harrison, 2019). After training the algorithm with labeled data, the *K*-Nearest Neighbors algorithm classifies unlabeled data by a majority vote of its *k* neighbors (Knn algorithm, 2015). In other words, the KNN takes the unlabeled data points and assumes that the similar data points exist in close proximity. Thus, KNN places the data points into well defined groups (Knn algorithm, 2015). To choose an appropriate number of nearest neighbors, *k*, we need to run the KNN algorithm several times with different *k* values (Harrison, 2019). Then, we determine the *k* that reduces the number of errors we encounter while also maintaining the ability to make accurare predictions. A large *k* results in more accurate predictions, but also increases the number of errors as *k* gets too large. Too small of a *k* value leads to less errors, but also less accurate predictions (Harrison, 2019).

For this assignment, we will applying the KNN algorithm to a dataset that contains information used to predict whether a patient has heart disease or not. In particular, we will be using the dataset located in the Cleveland database which contains 14 attributes (Janosi et al, n.d.). These attributes include (Janosi et al, n.d.):

1. *age*: Age of patient
2. *sex*: Sex of patient (1 = male, 0 = female)
3. *cp*: Chest pain type (1 = typical angina, 2 = atypical angina, 3 = non-anginal pain, 4 = asymptomatic)
4. *trestbps*: Resting blood pressure (in mm Hg on admission to the hospital)
5. *chol*: Cholestoral in mg/dl
6. *fbs*: Fasting blood sugar > 120 mg/dl (1 = true; 0 = false)
7. *restecg*: Resting electrocardiographic results
8. *thalach*: Maximum heart rate achieved
9. *exang*: Exercise induced angina (1 = yes; 0 = no)
10. *oldpeak*: ST depression induced by exercise relative to rest
11. *slope*: Slope of the peak exercise ST segment (1 = upsloping, 2 = flat, 3 = downsloping)
12. *ca*: Number of major vessels (0-3) colored by flourosopy
13. *thal*: 3 = normal; 6 = fixed defect; 7 = reversable defect
14. *num*: Diagnosis of heart disease (variable we are attempting to predict)

The goal of this experiment is to attempt to distinguish the presence of heart disease (values 1,2,3,4) from the absence of heart disease (value 0).

Load Libraries

Before we begin our experiment, we need to load the necessary libraries into R.

Hide

```
library(DataExplorer)
```

```
Registered S3 methods overwritten by 'htmltools':
  method               from
  print.html           tools:rstudio
  print.shiny.tag      tools:rstudio
  print.shiny.tag.list tools:rstudio
Registered S3 method overwritten by 'htmlwidgets':
  method           from
  print.htmlwidget tools:rstudio
```

Hide

```
library("caret")
```

```
Loading required package: lattice
Loading required package: ggplot2
```

Hide

```
library(class)
library("e1071")
```

The **DataExplorer** library allows us to perform data exploration analysis on our Heart Disease dataset (Cui, 2020). The **class** library is loaded so we can use the *K*-Nearest Neighbors, **knn()**, function (Torgo, n.d.). The **caret** and **e1071** packages are loaded so we can create a confusion matrix for our KNN models (Kuhn, n.d.).

Load Data

Now, we can load our data into R using the **read.csv()** function. Then, to confirm that our data was loaded properly, and to view the first few rows of our data, we can use the **head()** function.

Hide

```
heart <- read.csv(file.choose(), header = T)
head(heart)
```

	X63.0 <dbl>	X1.0 <dbl>	X1.0.1 <dbl>	X145.0 <dbl>	X233.0 <dbl>	X1.0.2 <dbl>	X2.0 <dbl>	X150.0 <dbl>	X0.0 <dbl>
1	67	1	4	160	286	0	2	108	1
2	67	1	4	120	229	0	2	129	1
3	37	1	3	130	250	0	0	187	0
4	41	0	2	130	204	0	2	172	0
5	56	1	2	120	236	0	0	178	0
6	62	0	4	140	268	0	2	160	0

6 rows | 1-10 of 14 columns

Notice, the column names do not provide any insight as to what the data in each column is. Thus, we can change the names of the columns to make them easier to understand and easier to access.

Rename Columns

To change the column names in our **heart** dataframe, we can use the **colnames()** function (Nguyen, 2016).

Hide

```
colnames(heart) <- c( "age", "sex", "cp", "trestbps", "chol", "fbs", "restecg", "thalach", "e
xang", "oldpeak", "slope", "ca", "thal", "num")
```

Now, we can easily access the columns in our dataframe.

Exploratory Data Analysis

Before we build our model, we need to explore our data. Exploratory data analysis “refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis and to check assumptions with the help of summary statistics and graphical representations (Patil, 2018).” Thus, we will begin by viewing a summary of our data.

Hide

```
summary(heart)
```

```
      age             sex               cp       
 Min.   :29.00   Min.   :0.0000   Min.   :1.000  
 1st Qu.:48.00   1st Qu.:0.0000   1st Qu.:3.000  
 Median :55.50   Median :1.0000   Median :3.000  
 Mean   :54.41   Mean   :0.6788   Mean   :3.166  
 3rd Qu.:61.00   3rd Qu.:1.0000   3rd Qu.:4.000  
 Max.   :77.00   Max.   :1.0000   Max.   :4.000  
    trestbps          chol            fbs        
 Min.   : 94.0   Min.   :126.0   Min.   :0.0000  
 1st Qu.:120.0   1st Qu.:211.0   1st Qu.:0.0000  
 Median :130.0   Median :241.5   Median :0.0000  
 Mean   :131.6   Mean   :246.7   Mean   :0.1457  
 3rd Qu.:140.0   3rd Qu.:275.0   3rd Qu.:0.0000  
 Max.   :200.0   Max.   :564.0   Max.   :1.0000  
    restecg          thalach          exang       
 Min.   :0.0000   Min.   : 71.0   Min.   :0.0000  
 1st Qu.:0.0000   1st Qu.:133.2   1st Qu.:0.0000  
 Median :0.5000   Median :153.0   Median :0.0000  
 Mean   :0.9868   Mean   :149.6   Mean   :0.3278  
 3rd Qu.:2.0000   3rd Qu.:166.0   3rd Qu.:1.0000  
 Max.   :2.0000   Max.   :202.0   Max.   :1.0000  
    oldpeak          slope            ca           
 Min.   :0.000   Min.   :1.000   Length:302        
 1st Qu.:0.000   1st Qu.:1.000   Class :character  
 Median :0.800   Median :2.000   Mode  :character  
 Mean   :1.035   Mean   :1.596                     
 3rd Qu.:1.600   3rd Qu.:2.000                     
 Max.   :6.200   Max.   :3.000                     
     thal                num        
 Length:302         Min.   :0.0000  
 Class :character   1st Qu.:0.0000  
 Mode  :character   Median :0.0000  
                    Mean   :0.9404  
                    3rd Qu.:2.0000  
                    Max.   :4.0000  
```

First, we notice that the columns **ca** and **thal** contain character values. We will further analyze this later. Also, we can see that the columns **age**, **trestbps**, **chol**, and **thalach** each contain a wide range of values. The column **chol** has the widest range of values, and may *dominate* the other features. When there is a dominating feature, this can result in skewed regressions or classifications (Harrison, 2019). Therefore, we will need to normalize these features.

Now, we can view the structure of our data with the **str()** function.

Hide

```
str(heart)
```

```
'data.frame':   302 obs. of  14 variables:
 $ age     : num  67 67 37 41 56 62 57 63 53 57 ...
 $ sex     : num  1 1 1 0 1 0 0 1 1 1 ...
 $ cp      : num  4 4 3 2 2 4 4 4 4 4 ...
 $ trestbps: num  160 120 130 130 120 140 120 130 140 140 ...
 $ chol    : num  286 229 250 204 236 268 354 254 203 192 ...
 $ fbs     : num  0 0 0 0 0 0 0 0 1 0 ...
 $ restecg : num  2 2 0 2 0 2 0 2 2 0 ...
 $ thalach : num  108 129 187 172 178 160 163 147 155 148 ...
 $ exang   : num  1 1 0 0 0 0 1 0 1 0 ...
 $ oldpeak : num  1.5 2.6 3.5 1.4 0.8 3.6 0.6 1.4 3.1 0.4 ...
 $ slope   : num  2 2 3 1 1 3 1 2 3 2 ...
 $ ca      : chr  "3.0" "2.0" "0.0" "0.0" ...
 $ thal    : chr  "3.0" "7.0" "3.0" "3.0" ...
 $ num     : int  2 1 0 0 0 3 0 2 1 0 ...
```

From this output, we can determine the type of data that is in each column (numeric, character, or integer) and view the structure of the data in each column.

Now, using the **DataExplorer** library, we can perform more data exploration. First, we can get “introduced” to our dataset with the **introduce()** function. Then, we can visualize the output with the **plot_intro()** function (Cui, 2020). We can also plot the missing values to visualize the missing profile for each feature (Cui, 2020).

Hide

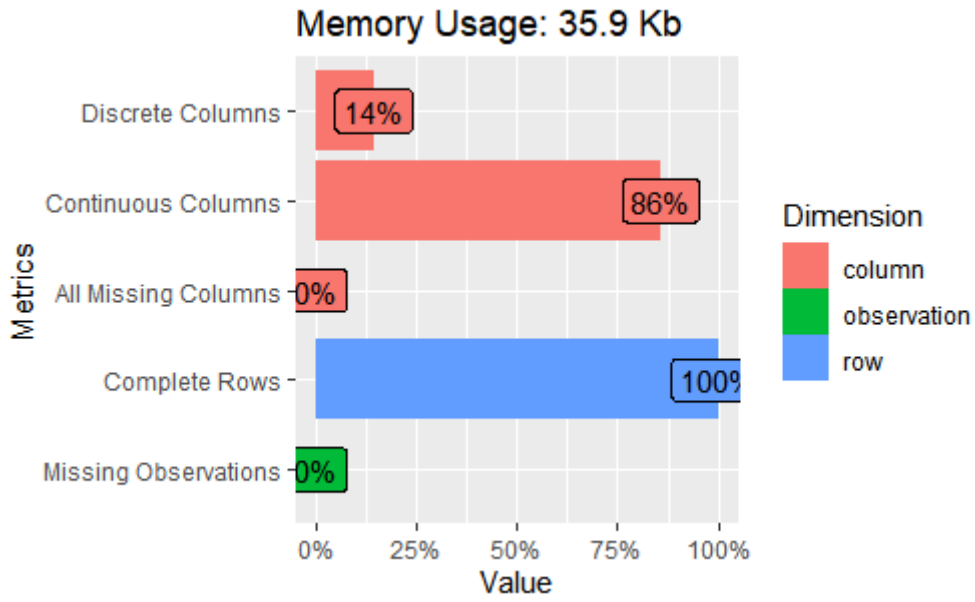
```
introduce(heart)
```

r… <int>	colu… <int>	discrete_columns <int>	continuous_columns <int>	all_missing_columns <int>	total_miss
302	14	2	12	0	

1 row | 1-6 of 9 columns

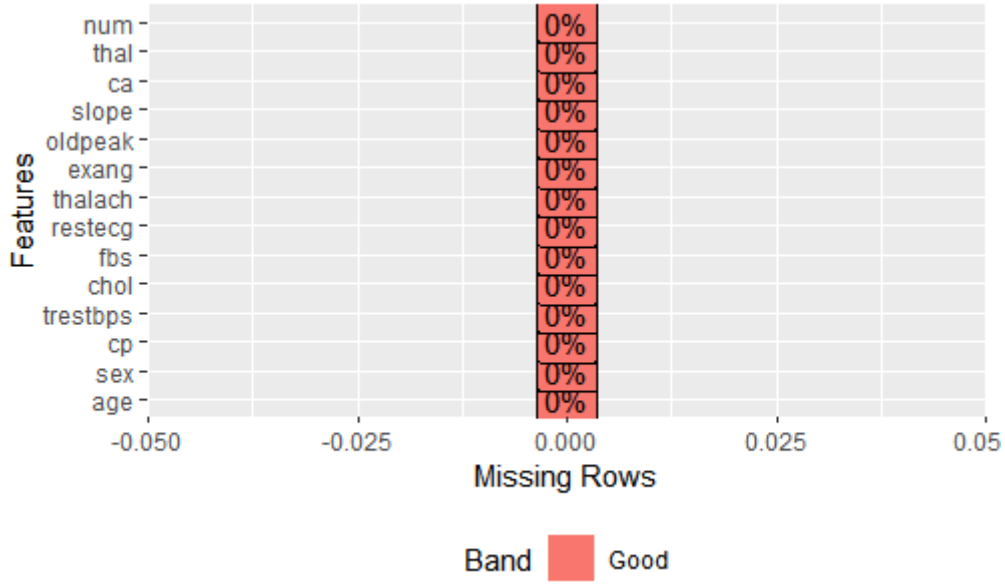
Hide

```
plot_intro(heart)
```



Hide

```
plot_missing(heart)
```

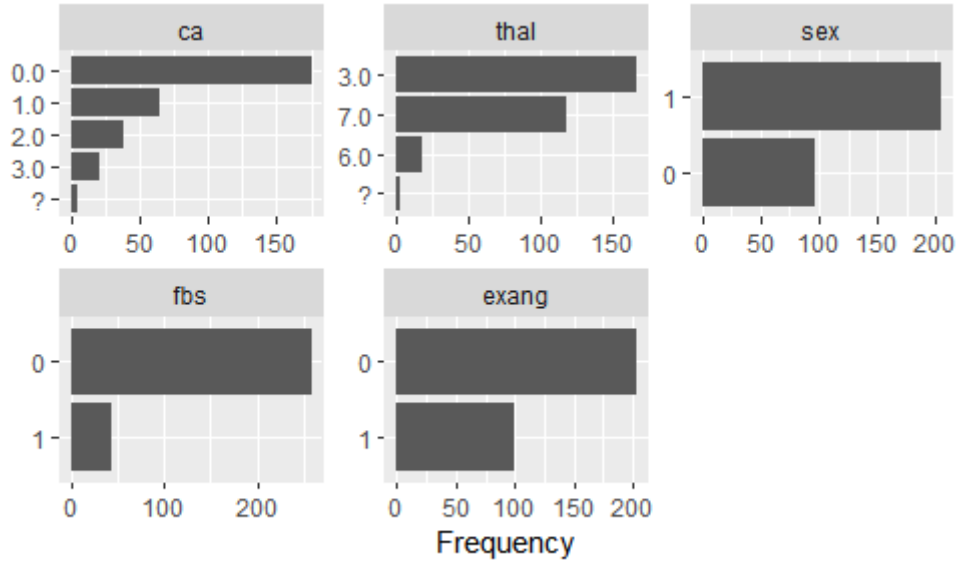


The output above shows that there is no missing columns or missing observations. Recall, we determined above that there are character values in two columns, which we will analyze later.

Further, we can visualize the frequency distributions for the discrete and continuous features. We can use a bar chart to visualize discrete features and a histogram for continuous features (Cui, 2020).

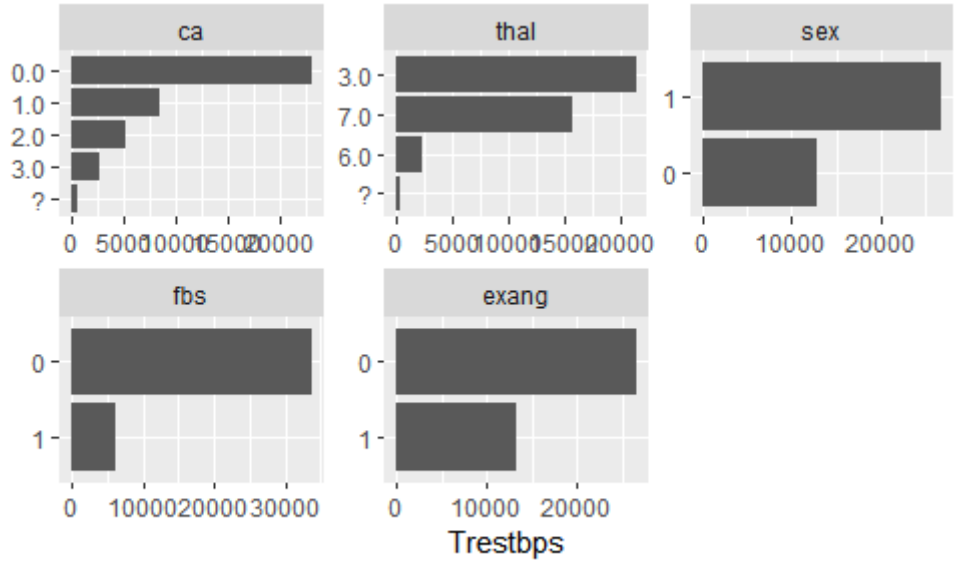
Hide

```
plot_bar(heart) #visualize frequency distributions for all discrete features
```



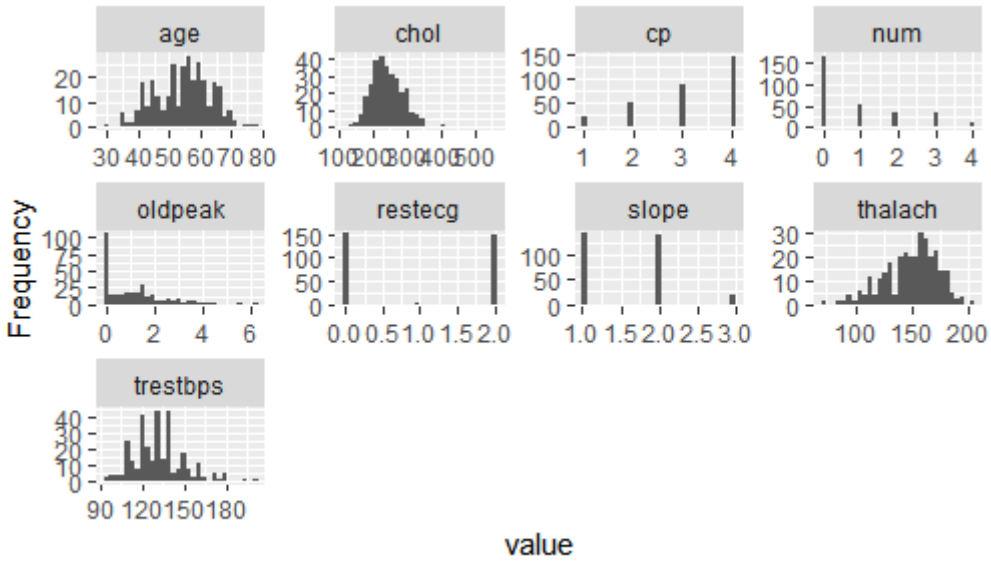
Hide

```
plot_bar(heart, with = "trestbps") # look at bivariate frequency distribution
```



Hide

```
plot_histogram(heart) #visualize distributions for all continuous features
```

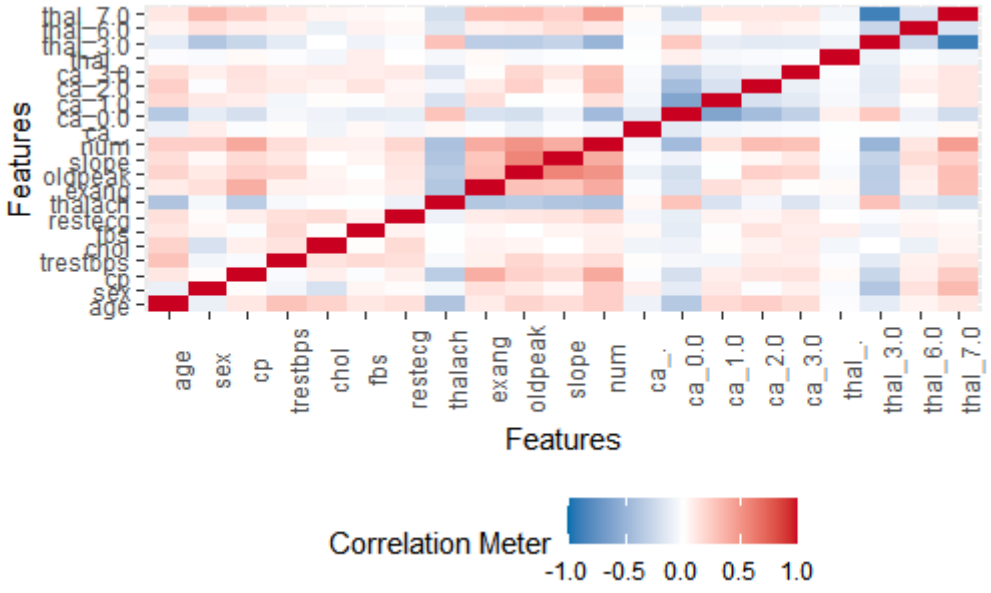


Recall, the two columns that contain character values are the **ca** and **thal** column. From the bar chart above, we can determine that the character values are *"?"*. Thus, we will need to convert these to *NA* values and remove them. We can also use a bar chart to look at bivariate frequency distributions (Cui, 2020). Above, we look at our discrete features by **trestbps**. From our histogram plot, we can see that the **oldpeak**, **chol**, and **thalach** features are not normally distributed. From both the barcharts and histograms, we can also see that there are features that we will have to treate as categorical/dummy, variables.

Next, we will view the correlation heatmap for all non-missing features (Cui, 2020).

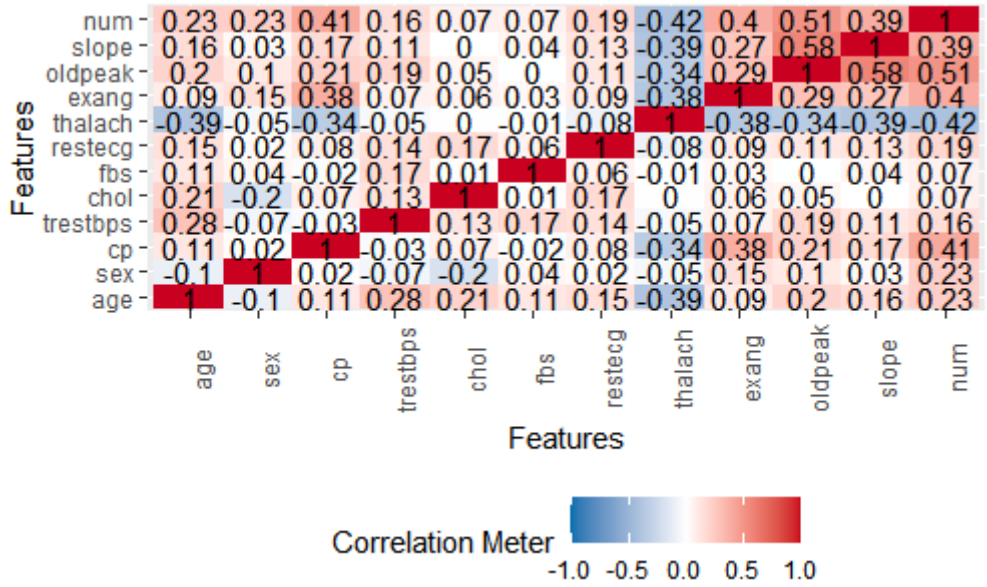
Hide

```
plot_correlation(na.omit(heart)) #visualize correlation heatmap for all non-missing features
```



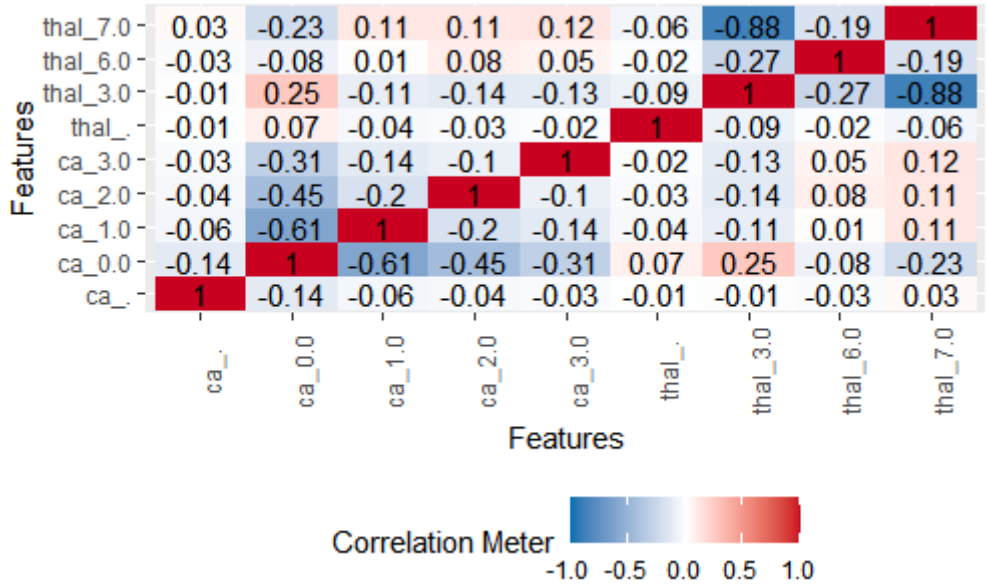
Hide

```
plot_correlation(na.omit(heart), type = "c") #continuous variables
```



Hide

```
plot_correlation(na.omit(heart), type = "d") #discrete variables
```

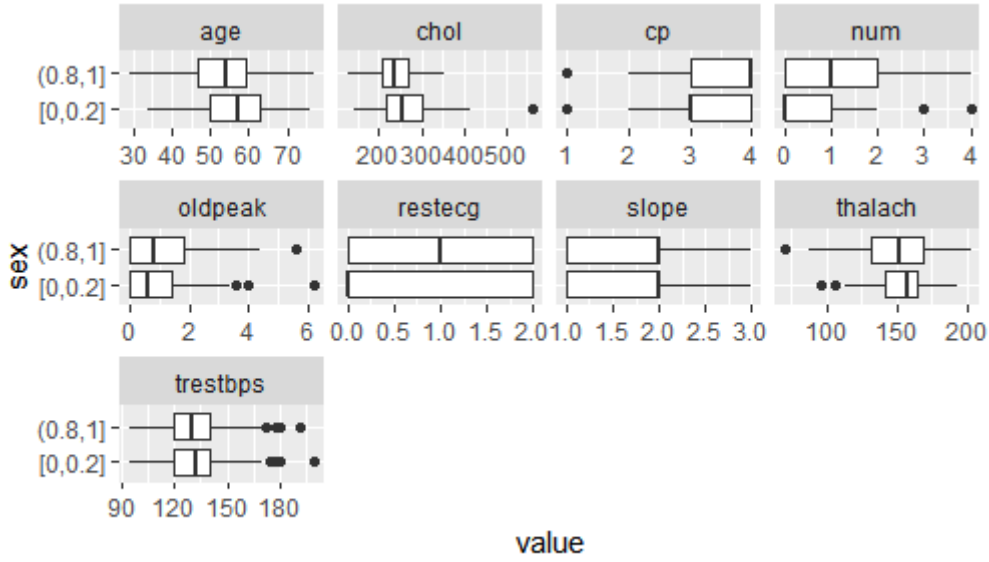


A correlation heatmap allow us to visualize a correlation matrix. The main reason for using a correlation heatmap/matrix is to summarize data so we can recognize patterns (Bock, 2018). From our outputs above, we can see that most of the features are positively correlated with each other. We can also add the argument **type = "d"** to view the heatmap for discrete features or the argument **type = "c"** to view the heatmap for continuous features.

Finally, we can visualize the distributions of the continuous features based on another feature by using a boxplot (Cui, 2020). Below, we view the distributions of the continuous features based on **sex**. An alternative visualization is a scatterplot.

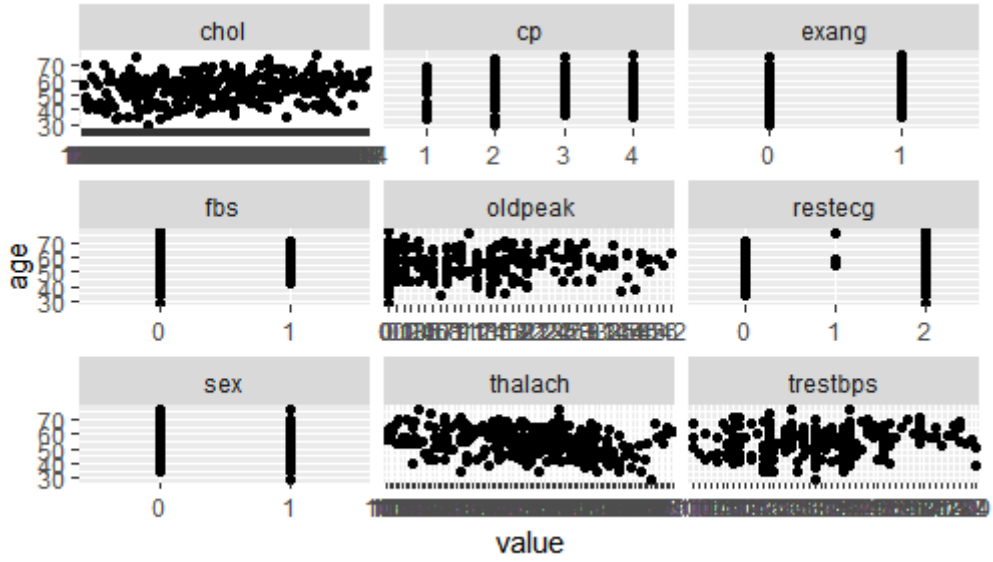
Hide

```
plot_boxplot(heart, by = 'sex') # visualize the distribution of all continuous features based
on sex with a boxplot
```

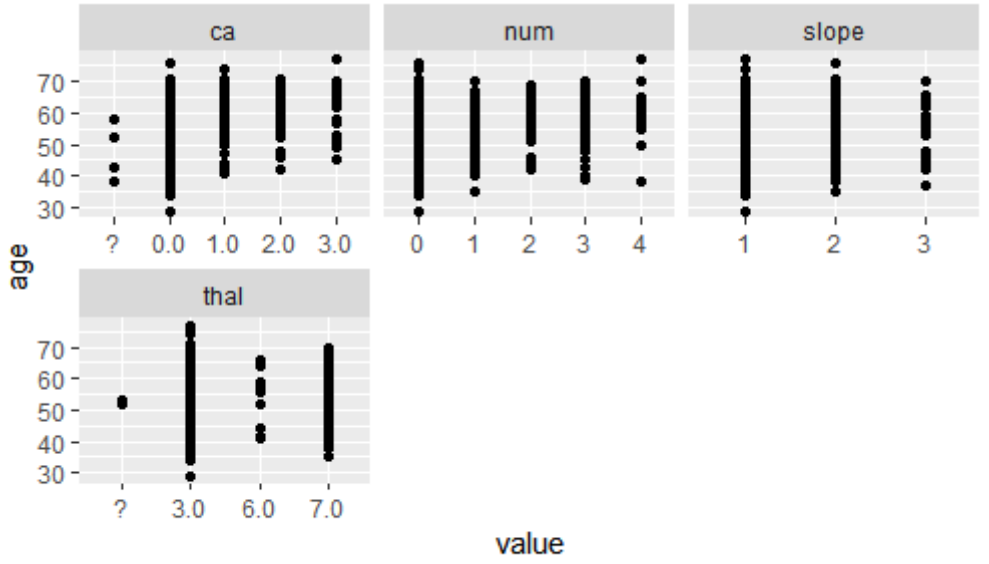


Hide

```
plot_scatterplot(heart, by = 'age')
```



Page 1



Page 2

The boxplots suggest that for the continuous features, the distributions between male and female data are similar.

Handle Missing Data

We have determined above that two of our columns include *"?"* characters and need to be converted to null values and then removed. To convert to null values, we can "get" the data that contains the "?" character and then replace them with **NA** (Vainora, 2012).

Hide

```
#recall, "ca" and "thal" columns contain "?" (both are discrete features)
unique(heart$ca)
```

```
[1] "3.0" "2.0" "0.0" "1.0" "?"
```

Hide

```
unique(heart$thal)
```

```
[1] "3.0" "7.0" "6.0" "?"
```

Hide

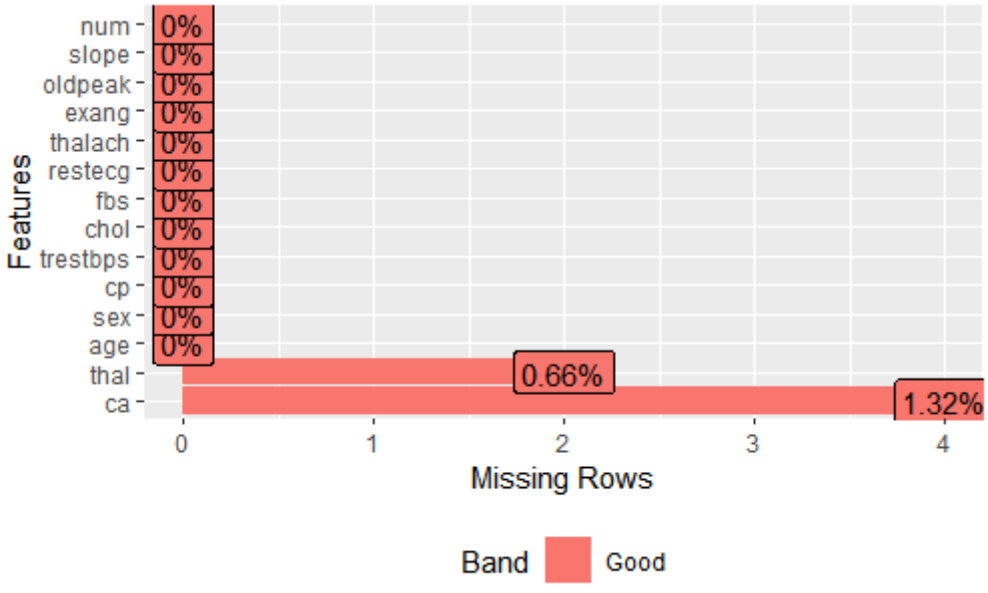
```
#replace ? with NA
heart[heart == "?"] <- NA

sum(is.na(heart))
```

```
[1] 6
```

Hide

```
plot_missing(heart)
```



We can now remove the null values using the **complete.cases()** function, which returns a dataframe with only complete rows.

Hide

```
complete_heart <- heart[complete.cases(heart), ]
sum(is.na(complete_heart))
```

```
[1] 0
```

Normalize Data

Next, we need to normalize particular columns in our dataset, or else we will have features that dominate the other features. This could result in skewed and inaccurate results. Using the normalizing function from last week, we can ensure that each feature contributes equally to our analysis (Zach, 2019). We will want to normalize the numerical columns. To view the datatypes of each column, we can use the **str()** function, like we did above, or we can use the **sapply()** function (R - determine the data types of a data frame's columns, n.d.).

The features that need to be normalized are **age**, **chol**, **oldpeak**, **thalach**, and **trestbp**. We can exclude **sex**, **fbs**, and **exang** because they are already values between 0 and 1. We will also exclude **cp**, **num**, **restecg**, **slope**, **ca**, and **thal** because these are our categorical variables. The **lapply()** function is used so that we can apply our normalize function to the selected features (Knn algorithm, 2015).

Hide

```
norm <- function(x) {return ((x - min(x)) / (max(x) - min(x)))}
sapply(complete_heart, class) #view data type of each column
```

```
       age         sex          cp    trestbps        chol 
 "numeric"   "numeric"   "numeric"   "numeric"   "numeric" 
       fbs     restecg     thalach       exang     oldpeak 
 "numeric"   "numeric"   "numeric"   "numeric"   "numeric" 
     slope          ca        thal         num 
 "numeric" "character" "character"   "integer" 
```

Hide

```
  #want to normalize numerical columns: age, chol, oldpeak, thalach, trestbp
  #exclude sex, fbs, exang; already 0, 1 values
  #exclude cp, num, restecg, slope, ca, thal; factors
norm_heart <- as.data.frame(lapply(complete_heart[,c(1,4,5,8,10)],norm))
head(norm_heart)
```

	age <dbl>	trestbps <dbl>	chol <dbl>	thalach <dbl>	oldpeak <dbl>
1	0.7916667	0.6226415	0.3652968	0.2824427	0.2419355
2	0.7916667	0.2452830	0.2351598	0.4427481	0.4193548
3	0.1666667	0.3396226	0.2831050	0.8854962	0.5645161
4	0.2500000	0.3396226	0.1780822	0.7709924	0.2258065
5	0.5625000	0.2452830	0.2511416	0.8167939	0.1290323

	age <dbl>	trestbps <dbl>	chol <dbl>	thalach <dbl>	oldpeak <dbl>
6	0.6875000	0.4339623	0.3242009	0.6793893	0.5806452

6 rows

Notice, we have saved our normalized features in a separate dataframe.

Convert to Dummy Variables

In machine learning, where modeling techniques are used, some data may need to be transformed into dummy variables (Amunategui, n.d.). There are numerous different ways to accomplish this, but the **dummyVars()** function located in the **caret** package allows a user to easily transform text data into numerical data (Amunategui, n.d.). The **dummyVars** function breaks out unique values from a column and transforms them into individual columns (Amunategui, n.d.). We will convert the factors **cp**, **num**, **restecg**, **slope**, **ca**, and **thal** to dummy variables because they do not provide any mathematical value. We will exclude **sex**, **fbs**, **exang** because the columns already 0 and 1 values.

First, we will convert the necessary columns to factors and save them in their own dataframe. Then, we will use the **dummyVars()** function to convert our features to dummy variables.

Hide

```
  #exclude sex, fbs, exang; already 0, 1 values
  #cp, num, restecg, slope, ca, thal; factors
cat_heart <- as.data.frame(lapply(complete_heart[,c(3,7,11:13)], as.factor))
dmy <- dummyVars("~.", data = cat_heart, fullRank = TRUE)
dmy_df <- as.data.frame(predict(dmy, newdata = cat_heart))
head(dmy_df)
```

	cp.2 <dbl>	cp.3 <dbl>	cp.4 <dbl>	restecg.1 <dbl>	restecg.2 <dbl>	slope.2 <dbl>	slope.3 <dbl>	ca.1.0 <dbl>	ca.2.0 <dbl>
1	0	0	1	0	1	1	0	0	0
2	0	0	1	0	1	1	0	0	1
3	0	1	0	0	0	0	1	0	0
4	1	0	0	0	1	0	0	0	0
5	1	0	0	0	0	0	0	0	0
6	0	0	1	0	1	0	1	0	1

6 rows | 1-10 of 12 columns

Now, our categorical features have been transformed into dummy variables. Next, recall from our dataset description above, that the absence of heart disease is represented with a 0 and the presence of heart disease is represented by a 1, 2, 3, or 4. To simplify this, we will represent the presence of heart disease with a 1, so that we only have two classes. To do this, we will utilize the **ifelse()** function. This allows us to convert any value greater than 1 in our **num** (predicted/response) column to a 1 and leave the 0 values as is.

Hide

```
#Change predicted/response variable (num) to 2 classes: 0 = No heart disease, Value >= 1 = He
art disease
#Then, convert to factor variable with as.factor
unique(complete_heart$num)
```

```
[1] 2 1 0 3 4
```

Hide

```
complete_heart$num <- as.factor(with(complete_heart, ifelse(num >=1, 1,0)))
  #if num >= 1, then convert to 1, else, leave num = 0
unique(complete_heart$num)
```

```
[1] 1 0
Levels: 0 1
```

Now, can combine all the features together in one dataframe.

Combine All Variables Back Together

To combine all the features together to create our final dataframe, we will use the **cbind()** function which allows us to combine vectors, matrices, and/or dataframes by columns (Schork, n.d.).

Hide

```
#norm_heart, dmy_df, cat_heart, complete_heart$sex, complete_heart$fbs, complete_heart$exang,
complete_heart$num

final_df <- cbind(complete_heart$sex, complete_heart$fbs, complete_heart$exang, norm_heart, d
my_df, cat_heart, complete_heart$num)
head(final_df)
```

	complete_heart$sex <dbl>	complete_heart$fbs <dbl>	complete_heart$exang <dbl>	age <dbl>	trestbps <dbl>
1	1	0	1	0.7916667	0.6226415
2	1	0	1	0.7916667	0.2452830
3	1	0	0	0.1666667	0.3396226
4	0	0	0	0.2500000	0.3396226
5	1	0	0	0.5625000	0.2452830
6	0	0	0	0.6875000	0.4339623

6 rows | 1-7 of 26 columns

Hide

```
dim(final_df) #dimensions of dataframe
```

```
[1] 296  26
```

Now, we have a dataframe that contains 26 columns and 296 rows. This is important to know for when we divide our data into a training and test set.

Divide Data into Train/Test Set

Using our **final_df**, we can split our data into a training and test set. The training set will be used to train out KNN model with labels provided. Then, we will use the KNN model with the test set with unlabeled data and then evaluate the accuracy of our model. To split the data, we will use the **createDataPartition** function located in the **caret** library. This allows us to create balanced splits of our data (Kuhn, n.d.). We will also need to set the seed for the random generator so our results can be reproduced. If we do not do this, the results will change every time the code is ran. We will create a two samples with a 70:30 ratio. In other words, our training set will be comprised of 70% of the final dataset and the test set will be 30% of the final dataset.

Hide

```
#Want 70/30 Split
set.seed(346)
index <- createDataPartition(final_df$`complete_heart$num`, p =0.7, list = FALSE)
#optimal k usually is sqrt(n)
heartTrain <- final_df[index,] #index for training set
cat("Training set dimensions:", dim(heartTrain), "\n")
```

```
Training set dimensions: 208 26
```

Hide

```
cat("Training set Split:", (NROW(heartTrain)/NROW(final_df))*100,"%", "\n")
```

```
Training set Split: 70.27027 %
```

Hide

```
heartTest <- final_df[-index,] #not index for test set
cat("Test set dimensions:", dim(heartTest), "\n")
```

```
Test set dimensions: 88 26
```

Hide

```
cat("Test set Split:", (NROW(heartTest)/NROW(final_df))*100,"%", "\n")
```

```
Test set Split: 29.72973 %
```

To confirm the split of the data, we can divide the number of rows (**NROW**) in our train or test set by the number of rows in our **final_df**. To print our output, we use the **cat()** function which outputs our objects and concatenates the representations (Cat function, n.d.).

KNN Algorithm

Now that we have our cleaned, transformed, and split data, we can build out *K*-Nearest Neighbors model. First, we need to define our training labels and our test labels, which will be used to compare to our predicted labels. Recall, our labels are located in the **num** column (the 26th column). We will obtain our training and test labels in a similar way we obtained our training and test sets (Evan, 2019).

Next, we can build our model. We will use the **knn()** function in the **class** library to build our initial model. Our initial model will have **k = 1**, which is the number of nearest neighbors.

Hide

```
labels <- final_df[26][index,]
test_labels <- final_df[-index,26]
knn_1 <- knn(train = heartTrain, test = heartTest, cl =labels, k =1)
table_1 <- table(knn_1, test_labels) #confusion matrix
table_1
```

```
     test_labels
knn_1  0  1
    0 44  6
    1  3 35
```

Hide

```
accuracy_1 <- 100*sum(test_labels == knn_1)/NROW(test_labels)
accuracy_1
```

```
[1] 89.77273
```

Hide

```
confusionMatrix(table_1)
```

```
Confusion Matrix and Statistics

     test_labels
knn_1  0  1
    0 44  6
    1  3 35

               Accuracy : 0.8977
                 95% CI : (0.8147, 0.9522)
    No Information Rate : 0.5341
    P-Value [Acc > NIR] : 2.051e-13

                  Kappa : 0.7935

 Mcnemar's Test P-Value : 0.505

            Sensitivity : 0.9362
            Specificity : 0.8537
         Pos Pred Value : 0.8800
         Neg Pred Value : 0.9211
             Prevalence : 0.5341
         Detection Rate : 0.5000
   Detection Prevalence : 0.5682
      Balanced Accuracy : 0.8949

       'Positive' Class : 0
```

Above, we use the **table()** function to check our predictions against our actual values (test labels) (Skand, 2017). The output shows that our model predicted 44 correct instances for the absence of heart disease and 35 correct instances for the presence of heart disease. There were 7 incorrect predictions. To calculate the accuracy, we can caluclate the proportion of correct classifications (Skand, 2017). This can also be done by using the **confusionMatrix()** function in the **caret** library. A confusion matrix outputs the accuracy of the model and a table that is used to visualize the performance of the model, similar to the **table()** output (Confusion matrix in r, 2020). The accuracy of out model according to the confusion matrix is ~89% when **k = 1**.

Now, the optimal value for *k* can be determined by using the elbow method or the maximum percent accuracy method. To find the highest accurate *k* value, we can create a loop. Now, as a “rule of thumb”, the value of *k* is chosen as the square root of the number of observations. Thus, in this case, the value for *k* should be 17.2046505.

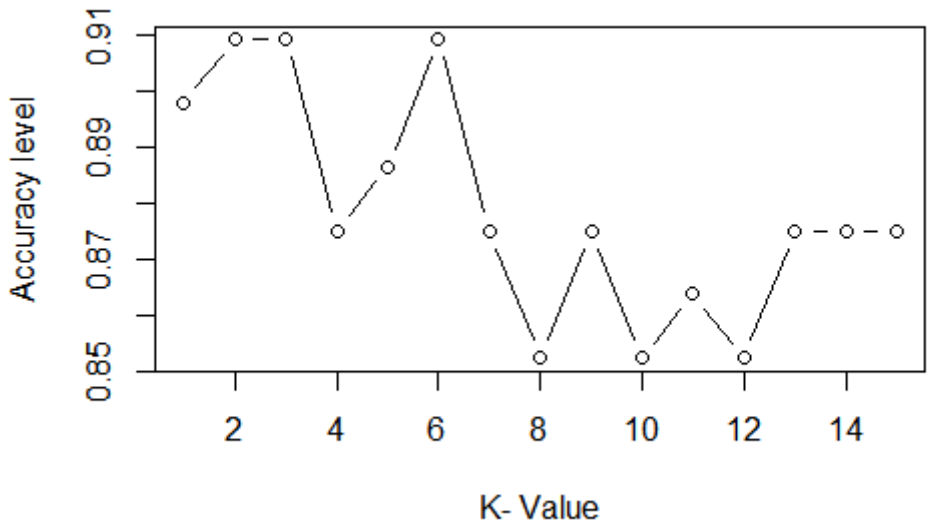
Hide

```
i = 1
k_optimal = 1
for(i in 1:15){
  knn_model <- knn(heartTrain, heartTest, cl=labels, k=i)
  k_optimal[i] <- confusionMatrix(knn_model, test_labels)$overall['Accuracy']
  k=i
  cat(k,'=',k_optimal[i], '\n')
}
```

```
1 = 0.8977273
2 = 0.9090909
3 = 0.9090909
4 = 0.875
5 = 0.8863636
6 = 0.9090909
7 = 0.875
8 = 0.8522727
9 = 0.875
10 = 0.8522727
11 = 0.8636364
12 = 0.8522727
13 = 0.875
14 = 0.875
15 = 0.875
```

Hide

```
plot(k_optimal, type="b", xlab="K- Value",ylab="Accuracy level")
```



We can see that our maximum accuracy occurs when **k = 3** according to the loop created above. .

Optimal *K=3*

Now, we will build our KNN model with *k* equal to 3.

Hide

```
knn_3 <- knn(train = heartTrain, test = heartTest, cl =labels, k =3)
table(knn_3, test_labels) #confusion matrix
```

```
     test_labels
knn_3  0  1
    0 43  4
    1  4 37
```

Hide

```
100*sum(test_labels == knn_3)/NROW(test_labels)
```

```
[1] 90.90909
```

Hide

```
confusionMatrix(knn_3, test_labels)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 43  4
         1  4 37

               Accuracy : 0.9091
                 95% CI : (0.8287, 0.9599)
    No Information Rate : 0.5341
    P-Value [Acc > NIR] : 2.6e-14

                  Kappa : 0.8173

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.9149
            Specificity : 0.9024
         Pos Pred Value : 0.9149
         Neg Pred Value : 0.9024
             Prevalence : 0.5341
         Detection Rate : 0.4886
   Detection Prevalence : 0.5341
      Balanced Accuracy : 0.9087

       'Positive' Class : 0
```

According to the confusion matrix above, the KNN model predicted 43 correct instances for the absence of heart disease, 37 correct instances for the presence of heart disease, and a total of 8 false predictions.

Conclusion

The goal of this assignment was to build a *K*-Nearest Neighbors model to classify heart disease patient based on certain attributes. After gathering the necessary data and loading it into R, we performed exploratory data analysis. During this analysis, we identified missing values that were not identified earlier. After finding these missing values, we converted them from a “?” to a null value. Then, we removed the rows that contained missing values. From here, we normalized features in our dataset, or we would have had dominating features which would have led to skewed results. Next, we converted our categorical features to dummy variables. After we combined all of our variables back into a final dataframe, we were able to build our KNN model. After our first model, we created a loop to find the most accurate number of classes. From here, we found the most accurate *k* value was 3. After using this value in our KNN model, we saw that the model was ~90% accurate with 80 correct predictions and 8 false predictions.

Resources

Amunategui, M. (n.d.). Data exploration & machine learning, hands-on. Retrieved September 5, 2020, from https://amunategui.github.io/dummyVar-Walkthrough/ (https://amunategui.github.io/dummyVar-Walkthrough/)

Bock, T. (2018, August 16). What is a correlation matrix? Displayr. https://www.displayr.com/what-is-a-correlation-matrix/ (https://www.displayr.com/what-is-a-correlation-matrix/)

Cat function. (n.d.). Retrieved September 5, 2020, from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/cat (https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/cat)

Confusion matrix in r. (2020, February 18). Honing Data Science. https://honingds.com/blog/confusion-matrix-in-r/ (https://honingds.com/blog/confusion-matrix-in-r/)

Cui, B. (2020, January 7). Introduction to dataexplorer. https://cran.r-project.org/web/packages/DataExplorer/vignettes/dataexplorer-intro.html#alternative (https://cran.r-project.org/web/packages/DataExplorer/vignettes/dataexplorer-intro.html#alternative)

Evan, P. (2019, February 23). R - error “train” and “class” have different lengths. Stack Overflow. https://stackoverflow.com/questions/54837161/error-train-and-class-have-different-lengths (https://stackoverflow.com/questions/54837161/error-train-and-class-have-different-lengths)

Gorelik, B. (2014, June 22). How to retrieve overall accuracy value from confusionmatrix in r. Stack Overflow. https://stackoverflow.com/questions/24348973/how-to-retrieve-overall-accuracy-value-from-confusionmatrix-in-r (https://stackoverflow.com/questions/24348973/how-to-retrieve-overall-accuracy-value-from-confusionmatrix-in-r)

Harrison, O. (2019, July 14). Machine learning basics with the k-nearest neighbors algorithm. Medium. https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761 (https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761)

Janosi, A., Steinbrunn, W., Pfisterer, M., & Detrano, R. (n.d.). Heart disease data set. Retrieved September 4, 2020, from https://archive.ics.uci.edu/ml/datasets/Heart+Disease (https://archive.ics.uci.edu/ml/datasets/Heart+Disease)

Knn algorithm. (2015, August 19). Analytics Vidhya. https://www.analyticsvidhya.com/blog/2015/08/learning-concept-knn-algorithms-programming/ (https://www.analyticsvidhya.com/blog/2015/08/learning-concept-knn-algorithms-programming/)

Kuhn, M. (n.d.-a). 4 data splitting | the caret package. Retrieved September 5, 2020, from https://topepo.github.io/caret/data-splitting.html (https://topepo.github.io/caret/data-splitting.html)

Nguyen , T. (2016, July 20). Changing column names of a data frame. https://stackoverflow.com/questions/6081439/changing-column-names-of-a-data-frame (https://stackoverflow.com/questions/6081439/changing-column-names-of-a-data-frame)

Patil, P. (2018, May 23). What is exploratory data analysis? Medium. https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15 (https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15)

R - determine the data types of a data frame’s columns. (n.d.). Stack Overflow. Retrieved September 5, 2020, from https://stackoverflow.com/questions/21125222/determine-the-data-types-of-a-data-frames-columns (https://stackoverflow.com/questions/21125222/determine-the-data-types-of-a-data-frames-columns)

Schork, J. (n.d.). Cbind r command. Statistics Globe. Retrieved September 5, 2020, from https://statisticsglobe.com/cbind-r-command-example/ (https://statisticsglobe.com/cbind-r-command-example/)

Skand, K. (2017, October 8). Knn(K-nearest neighbour) algorithm in r. https://rstudio-pubs-static.s3.amazonaws.com/316172_a857ca788d1441f8be1bcd1e31f0e875.html (https://rstudio-pubs-static.s3.amazonaws.com/316172_a857ca788d1441f8be1bcd1e31f0e875.html)

Torgo, L. (n.d.). Knn function | r documentation. Retrieved September 4, 2020, from https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/kNN (https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/kNN)

Vainora, J. (2012, June 14). R - replace all 0 values to na. Stack Overflow. https://stackoverflow.com/questions/11036989/replace-all-0-values-to-na (https://stackoverflow.com/questions/11036989/replace-all-0-values-to-na)

Zach. (2019, April 20). How to normalize data in r. Statology. https://www.statology.org/how-to-normalize-data-in-r/ (https://www.statology.org/how-to-normalize-data-in-r/)