EAS508: Statistical learning and Data Mining I

Project_2

Team members:

- Sri Guna Kaushik Undru (srigunak)
- Dominic Parosh Yamarthi (dyamarth)
- Shri Harsha Adapala Thirumala (sadapala)
- Sushmitha Jakka (sjakka)

Q) Project #2 is a classification project. The data set is presented in the attached project2.txt file. The first four columns represent four quantitative predictors. The fifth column stands for a categorical response, with two categories coded as 0 and 1. Your task is to build a predictive model that minimizes test misclassification rate. Estimate this rate by cross-validation. Use at least four different classification methods learned in this class.

Solution:

We are setting seed to be 1 and read the data proved in *"project2.txt"* file and see the top 6 rows of the data by using *"head"* function.

```
set.seed(1)

my_data <- read.table("data/project2.txt", header = FALSE, sep = ",", stringsAsFactors = FALSE)

head(my_data)
```

Output:

```
A data.frame: 6 × 5

V1      V2      V3      V4      V5

<dbl>   <dbl>   <dbl>   <dbl>   <int>

1       3.62160 8.6661  -2.8073 -0.446990

2       4.54590 8.1674  -2.4586 -1.462100

3       3.86600 -2.6383 1.9242  0.10645 0

4       3.45660 9.5228  -4.0112 -3.594400

5       0.32924 -4.4552 4.5718  -0.988800

6       4.36840 9.6718  -3.9606 -3.162500
```

It is given the fifth column stands for a categorical response, with two categories coded as 0 and 1 so we make V5 as factor by using *"as.factor"* function.

```
my_data$V5 = as.factor(my_data$V5)
```

We get the dimensions of the given dataset using *"dim"* function.

```
print(paste('Dimension of the data:', dim(my_data)))
```

Output:

```
[1] "Dimestion of the data: 1372" "Dimestion of the data: 5"
```

Exploratory Data Analysis (EDA)

Exploratory data analysis (EDA) is a crucial initial step in data analysis that involves exploring and summarizing the key features of a dataset. The aim of EDA is to comprehend the data's nature, detect patterns, and pinpoint any possible outliers or anomalies.

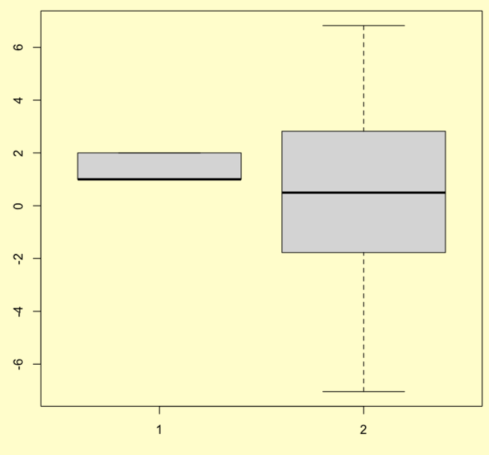
To perform EDA, one typically starts with data preparation, which involves cleaning the data, checking for missing values, and removing duplicates. Next, univariate analysis is conducted to examine the distribution of each variable in the dataset, including measures of central tendency and dispersion. Then, bivariate analysis is used to investigate the relationship between pairs of variables, while multivariate analysis is employed to explore the relationship between three or more variables.

Throughout the EDA process, it is crucial to create visualizations such as graphs, charts, and histograms to aid in identifying patterns or trends in the data and detecting any outliers or anomalies.

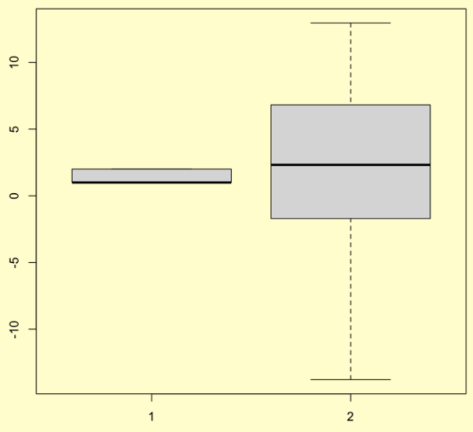
- EDA helps data analysts gain a better understand data, make informed decisions about which statistical methods to use, and identify any potential issues that need to be addressed before proceeding to more advanced analyses.
- EDA can help identify data quality issues: During EDA, data analysts may identify data quality issues such as missing or incorrect data. Addressing these issues early in the analysis process can help improve the accuracy of the results.
- EDA can help identify potential research questions: By exploring the data and discovering patterns or relationships between variables, data analysts may identify interesting research questions that were not initially considered.
- EDA can help select appropriate statistical methods: By understanding the distribution of the data and the relationship between variables, data analysts can choose appropriate statistical methods to use in subsequent analyses.
- EDA is an iterative process: EDA is not a one-time activity but rather an iterative process that involves continuously exploring the data as new insights are discovered or as the research question changes.
- EDA can be subjective: EDA involves making subjective decisions based on the data and the research question. Therefore, it is important for data analysts to document their decisions and justify their choices.
- Overall, EDA is a critical step in the data analysis process that can provide valuable insights into the data and help data analysts make informed decisions about subsequent analyses.

Using *"boxplot"* function plot the box plots for V1, V2, V3 and V4 against V5

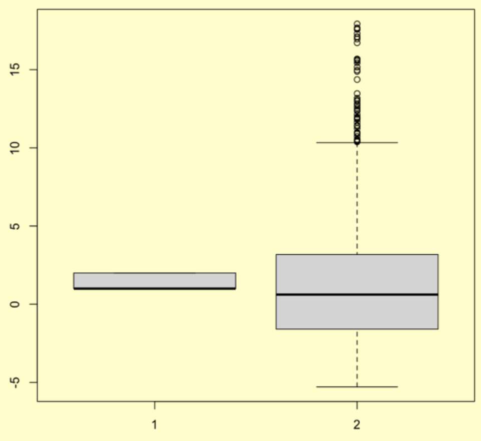
```
# V5 vs V1

boxplot(my_data$V5, my_data$V1)
```



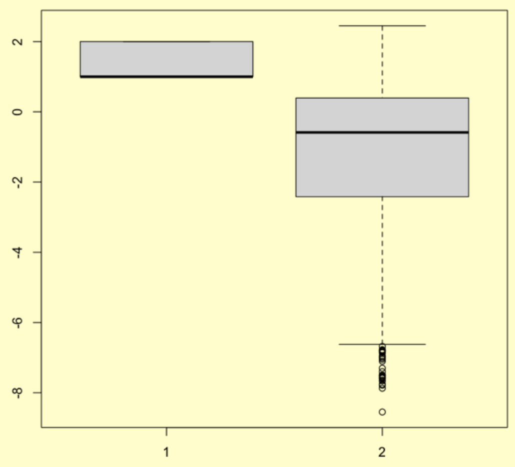
```
# V5 vs V2

boxplot(my_data$V5, my_data$V2)
```



```
# V5 vs V3

boxplot(my_data$V5, my_data$V3)
```



```
# V5 vs V4

boxplot(my_data$V5, my_data$V4)
```



Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is a statistical method that aims to find a linear combination of features that can effectively separate two or more classes in a dataset. It is a supervised learning technique used for classification tasks.

The basic idea of LDA is to project the original dataset onto a lower-dimensional space while preserving the maximum possible separation between the classes. In other words, it finds the linear combination of the original features that best discriminate between the classes.

LDA assumes that the data is normally distributed, and that the covariance matrix of each class is equal. It also assumes that the features are independent of each other.

The LDA algorithm involves calculating the mean and covariance matrix of each class, and then finding a set of coefficients that maximize the ratio of between-class variance to within-class variance. This set of coefficients is used to project the data onto a lower-dimensional space.

LDA can be used for dimensionality reduction as well as classification. In dimensionality reduction, LDA is used to project the data onto a lower-dimensional space while preserving the maximum possible separation between the classes. In classification, LDA is used to classify new data points based on their projected values.

LDA is a popular technique in various fields, including computer vision, pattern recognition, and bioinformatics. It has been used for face recognition, text classification, and gene expression analysis, among other applications.

LDA has several advantages over other classification algorithms, such as Naive Bayes and Logistic Regression. One advantage is that LDA can handle high-dimensional data well, as it reduces the number of features needed for classification. Another advantage is that LDA is a parametric method, which means that it makes assumptions about the distribution of the data. This can lead to better performance than non-parametric methods in some cases.

One potential disadvantage of LDA is that it assumes that the classes have equal covariance matrices. If this assumption is not met, the performance of LDA may suffer. Additionally, LDA may not perform well if the classes are highly overlapping or if the data is non-linearly separable.

To perform Linear Discriminant Analysis on the given dataset. Initially we must import the necessary libraries and we will be using *"MASS"* and *"caret"* libraries.

```
library(MASS)

library(caret)
```

Output:

```
Loading required package: ggplot2

Loading required package: lattice
```

We obtain the column names in the dataset by using *"names"* function.

```
names(my_data)
```

Output:

```
'V1"V2"V3"V4"V5'
```

We set training control object for 5-fold cross validation and train the LDA model for the given dataset using *"trainControl"* and *"train"* functions. We then will get the accuracy and error.

```
# Set up the training control object for 5-fold cross-validation

ctrl <- trainControl(method = "cv", number = 5)

# Train the LDA model using 5-fold cross-validation

lda_fit <- train(V5 ~ ., data = my_data, method = "lda", trControl = ctrl)

print(paste("Cross Validated Accuracy:",lda_fit$results$Accuracy))

print(paste("Cross Validated Error:",1-lda_fit$results$Accuracy))
```

Output:

```
[1] "Cross Validated Accuracy: 0.976684804246848"

[1] "Cross Validated Error: 0.023315195753152"
```

- ❖ On performing Linear Discriminant Analysis (LDA) classification on the given dataset we got a 5 fold cross-validation error of 0.0233 and an accuracy of 97%

<u>Quadratic Discriminant Analysis (QDA)</u>

Quadratic Discriminant Analysis (QDA) is a classification algorithm that is like Linear Discriminant Analysis (LDA) but allows for more flexible decision boundaries between classes. Like LDA, QDA is a supervised learning method used for classification tasks. The main difference between QDA and LDA is that QDA does not assume that the classes share the same covariance matrix, as LDA does. Instead, QDA allows each class to have its own covariance matrix, which can lead to more complex decision boundaries. QDA works by first estimating the mean and covariance matrix for each class. It then calculates the probability density function for each class and uses these functions to classify new data points.

Unlike LDA, which produces linear decision boundaries between classes, QDA produces quadratic decision boundaries. This allows QDA to model more complex relationships between the features and the class labels, but also makes it more prone to overfitting if the number of features is large. QDA can be useful in situations where the covariance matrices of the classes are different or when the decision boundary between classes is curved rather than linear. However, QDA can be computationally expensive, especially when the number of features is large, as it requires the estimation of a separate covariance matrix for each class.

Overall, QDA is a useful and flexible classification algorithm that can be a good alternative to LDA in certain situations. It can handle more complex decision boundaries than LDA, but its increased flexibility comes at the cost of increased complexity and computational expense. Like LDA, QDA can be used for dimensionality reduction. QDA can be used to project the data onto a lower-dimensional space while preserving the maximum possible separation between the classes.

One disadvantage of QDA is that it can be computationally expensive, especially when the number of features is large. This is because QDA requires the estimation of a separate covariance matrix for each class, which can be time-consuming and memory intensive. Another disadvantage of QDA is that it can be prone to overfitting when the number of features is large compared to the number of observations. This is because QDA has a higher model complexity than LDA, which can lead to overfitting if the number of features is not carefully controlled. Overall, QDA is a flexible and powerful classification algorithm that can be useful in certain situations where LDA may not be appropriate. However, it should be used with caution when the number of features is large or when the data is highly complex, as it can be prone to overfitting.

To perform Quadratic Discriminant Analysis on the given dataset we set training control object for 5-fold cross validation and train the QDA model for the given dataset using *"trainControl"* and *"train"* functions. We then will get the accuracy and error.

```
# Set up the training control object for 5-fold cross-validation

ctrl <- trainControl(method = "cv", number = 5)

# Train the QDA model using 5-fold cross-validation

qda_fit <- train(V5 ~ ., data = my_data, method = "qda", trControl = ctrl)

print(paste("Cross Validated Accuracy:",qda_fit$results$Accuracy))

print(paste("Cross Validated Error:",1-qda_fit$results$Accuracy))
```

Output:

```
[1] "Cross Validated Accuracy: 0.984698075646981"

[1] "Cross Validated Error: 0.0153019243530192"
```

❖ On performing Quadratic Discriminant Analysis (LDA) classification on the given dataset we got a 5-fold cross-validation error of 0.0153 and an accuracy of 98%

Naive Bayes

Naive Bayes is a classification algorithm based on Bayes' theorem and the assumption that the features are conditionally independent given the class label. The term "naive" comes from the assumption that the features are independent of each other, which is often not true in practice, but can still lead to good results in many cases.

Naive Bayes works by first estimating the probabilities of each class label based on the training data. It then calculates the conditional probability of each feature given the class label, based on the training data. Finally, it uses Bayes' theorem to calculate the posterior probability of each class given the observed features and chooses the class with the highest posterior probability as the predicted class label.

Naive Bayes can handle both binary and multiclass classification problems. It can also be used for both numerical and categorical data, if the assumptions of conditional independence and equal variances are met.

One advantage of Naive Bayes is that it is computationally efficient and can be trained quickly, even on large datasets. It also works well with high-dimensional data, as it reduces the number of parameters that need to be estimated. Another advantage of Naive Bayes is that it can perform well even with a small amount of training data. This makes it a good choice for applications where data is limited or expensive to collect.

However, the main disadvantage of Naive Bayes is that it relies on the assumption of conditional independence between features, which is often not true in practice. This can lead to suboptimal performance if the features are highly correlated. Additionally, Naive Bayes can be sensitive to irrelevant features or outliers in the data, which can lead to overfitting.

Overall, Naive Bayes is a simple yet powerful classification algorithm that can be a good choice in many situations. Its computational efficiency, ability to handle high-dimensional data, and good performance with limited training data make it a popular choice for many machine learning applications.

To perform Naïve Bayes on the given dataset. Initially we must import the necessary libraries and we will be using *"klaR"* library and then to perform Naïve Bayes on the given dataset we set training control object for 5-fold cross validation and train the model for the given dataset using *"trainControl"* and *"train"* functions. We then will get the accuracy and error.

```
library(klaR)

# Set up the training control object for 5-fold cross-validation

ctrl <- trainControl(method = "cv", number = 5)

# Train the Naive Bayes model using 5-fold cross-validation

nb_fit <- train(V5 ~ ., data = my_data, method = "nb", trControl = ctrl)

print(paste("Cross Validated Accuracy for 0:",nb_fit$results$Accuracy[1]))

print(paste("Cross Validated Accuracy for 1:",nb_fit$results$Accuracy[2]))

print(paste("Cross Validated Error for 0:",1-nb_fit$results$Accuracy[1]))

print(paste("Cross Validated Error for 1:",1-nb_fit$results$Accuracy[2]))

nb_fit$results
```

Output:

```
Warning message in FUN(X[[i]], ...):

“Numerical 0 probability for all classes with observation 96”

Warning message in FUN(X[[i]], ...):

“Numerical 0 probability for all classes with observation 235”

Warning message in FUN(X[[i]], ...):

“Numerical 0 probability for all classes with observation 245”

Warning message in FUN(X[[i]], ...):

“Numerical 0 probability for all classes with observation 240”

[1] "Cross Validated Accuracy for 0: 0.839633709356337"

[1] "Cross Validated Accuracy for 1: 0.916156602521566"

[1] "Cross Validated Error for 0: 0.160366290643663"
```

```
[1] "Cross Validated Error for 1: 0.0838433974784338"

A data.frame: 2 × 7

usekernel       fL      adjust   Accuracy        Kappa   AccuracySD     KappaSD

<lgl>   <dbl>   <dbl>   <dbl>    <dbl>   <dbl>   <dbl>

1       FALSE   0       1        0.8396337       0.6727459       0.03142926      0.06522405

2       TRUE    0       1        0.9161566       0.8290829       0.02243737      0.04628368
```

- On performing Naïve Bayes classification Algorithm on the given dataset, we get a 5-fold cross-validation accuracy as follows:
  - For 0: 83%
  - For 1: 91%
- We get a 5-fold cross-validation accuracy as follows:
  - For 0: 0.1604
  - For 1: 0.0838

K-Nearest Neighbours (KNN)

K-Nearest Neighbours (KNN) is a classification algorithm that is based on the idea of finding the k-nearest training examples in the feature space to a given test example, and assigning the test example the class label that is most common among its k-nearest neighbours.

The value of k is a hyperparameter that can be tuned to optimize the algorithm's performance. In general, larger values of k lead to smoother decision boundaries and can help to reduce the impact of noisy or irrelevant features but may also lead to misclassification of complex patterns. Smaller values of k lead to more complex decision boundaries that can capture more complex patterns but may be more sensitive to noise or outliers in the data.

KNN can handle both binary and multiclass classification problems, as well as regression problems. It can also work with both numerical and categorical data, although it requires a distance metric that is appropriate for the type of data being used.

One advantage of KNN is that it is a non-parametric algorithm, meaning that it does not make any assumptions about the underlying distribution of the data. This can make it a useful algorithm in situations where the distribution of the data is unknown or difficult to model.

Another advantage of KNN is that it is easy to understand and implement, making it a good choice for beginners or for situations where a quick and simple solution is needed.

However, the main disadvantage of KNN is that it can be computationally expensive, especially for large datasets or high-dimensional feature spaces. This is because KNN requires the computation of distances between the test example and all the training examples, which can be time-consuming and memory intensive.

Overall, KNN is a simple and versatile classification algorithm that can be a good choice in many situations. Its ability to handle a wide range of data types and problem types, as well as its non-parametric nature, make it a useful tool in many machine learning applications.

To perform K-Nearest Neighbours (KNN) is a classification algorithm we first need to import *"class"* library and set up the training control object for 5-fold cross validation and then train KNN model using 5-fold cross validation and we get accuracy and error.

```
library(class)

# Set up the training control object for 5-fold cross-validation

ctrl <- trainControl(method = "cv", number = 5)

# Train the KNN model using 5-fold cross-validation

knn_fit <- train(V5 ~ ., data = my_data, method = "knn", trControl = ctrl, tuneLength = 10)

knn_fit


print(paste("Cross Validated Accuracy:",max(knn_fit$results$Accuracy)))

print(paste("Cross Validated Error:",1-max(knn_fit$results$Accuracy)))
```

Output:

```
k-Nearest Neighbors

1372 samples

   4 predictor

   2 classes: '0', '1'

No pre-processing

Resampling: Cross-Validated (5 fold)

Summary of sample sizes: 1097, 1098, 1097, 1098, 1098

Resampling results across tuning parameters:

  k   Accuracy   Kappa

   5  1.0000000  1.0000000

   7  1.0000000  1.0000000

   9  1.0000000  1.0000000

  11  1.0000000  1.0000000

  13  1.0000000  1.0000000

  15  0.9970856  0.9941056

  17  0.9934466  0.9867604

  19  0.9927167  0.9852840

  21  0.9927167  0.9852840

  23  0.9927167  0.9852840

Accuracy was used to select the optimal model using the largest value.

The final value used for the model was k = 13.

[1] "Cross Validated Accuracy: 1"

[1] "Cross Validated Error: 0"
```

Logistic Regression

Logistic Regression is a classification algorithm that is used to predict the probability of a binary or categorical outcome based on one or more predictor variables. It is a type of regression analysis that uses a logistic function to model the relationship between the predictors and the probability of the outcome.

The logistic function is a sigmoidal curve that ranges from 0 to 1 and can be used to model the probability of a binary outcome such as "yes" or "no" or "true" or "false". Logistic Regression can also be extended to handle multiclass classification problems using one-vs-all or one-vs-one approaches.

Logistic Regression works by first estimating the parameters of the logistic function based on the training data. It then uses these parameters to predict the probability of the outcome for new data points. The predicted probability can then be converted into a binary decision using a threshold value, which can be adjusted to optimize the algorithm's performance.

Logistic Regression can handle both numerical and categorical data, and can be used for both binary and multiclass classification problems. It can also be extended to handle ordinal regression problems, where the outcome variable has a natural ordering.

One advantage of Logistic Regression is that it is a simple yet powerful algorithm that is relatively easy to interpret. The logistic function can be used to estimate the relationship between the predictor variables and the probability of the outcome, which can provide insight into the underlying factors that influence the outcome.

Another advantage of Logistic Regression is that it is computationally efficient and can be trained quickly, even on large datasets. This makes it a good choice for applications where data is limited or expensive to collect.

However, the main disadvantage of Logistic Regression is that it relies on the assumption of a linear relationship between the predictor variables and the logarithm of the odds ratio of the outcome. This assumption may not be true in all cases and can lead to suboptimal performance if the relationship between the predictor variables and the outcome is highly non-linear.

Overall, Logistic Regression is a powerful and widely used classification algorithm that can be a good choice in many situations. Its simplicity, interpretability, and computational efficiency make it a popular choice for many machine learning applications.

To perform Logistic Regression on the given dataset we set training control object for 5-fold cross validation and train the QDA model for the given dataset using *"trainControl"* and *"train"* functions. We then will get the accuracy and error.

```
# Set up the training control object for 5-fold cross-validation

ctrl <- trainControl(method = "cv", number = 5)


# Train the Logistic Regression model using 5-fold cross-validation

log_fit <- train(V5 ~ ., data = my_data, method = "glm", trControl = ctrl)


log_fit


print(paste("Cross Validated Accuracy:",log_fit$results$Accuracy))

print(paste("Cross Validated Error:",1-log_fit$results$Accuracy))
```

Output:

```
Warning message:
“glm.fit: fitted probabilities numerically 0 or 1 occurred”
Warning message:
“glm.fit: fitted probabilities numerically 0 or 1 occurred”
Warning message:
“glm.fit: fitted probabilities numerically 0 or 1 occurred”
Warning message:
“glm.fit: fitted probabilities numerically 0 or 1 occurred”
Warning message:
“glm.fit: fitted probabilities numerically 0 or 1 occurred”
Warning message:
“glm.fit: fitted probabilities numerically 0 or 1 occurred”

Generalized Linear Model

1372 samples
   4 predictor
   2 classes: '0', '1'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 1097, 1098, 1098, 1097, 1098
Resampling results:

  Accuracy   Kappa
  0.9897996  0.9793517

[1] "Cross Validated Accuracy: 0.989799601857996"
[1] "Cross Validated Error: 0.010200398142004"
```

- After performing Logistic Regression for classification of the given dataset we get a 5-fold cross-validation error of 0.012 and an accuracy of nearly 99%.

Classification Trees

Classification Trees, also known as Decision Trees, are a type of algorithm used for classification and prediction tasks. They build a tree-like model of decisions and their possible consequences, where the leaves represent the class labels or target values, and the branches represent the features or attributes that lead to those labels or values.

Classification Trees are built by recursively partitioning the data into subsets based on the values of the predictor variables. At each node of the tree, the algorithm selects the predictor variable that best separates the data into two or more homogeneous subsets, based on some criterion such as information gain or Gini impurity. The process continues until a stopping criterion is met, such as a maximum depth of the tree or a minimum number of samples per leaf.

Once the tree is built, it can be used to predict the class label or target value for new instances by traversing the tree from the root node to a leaf node, based on the values of the predictor variables.

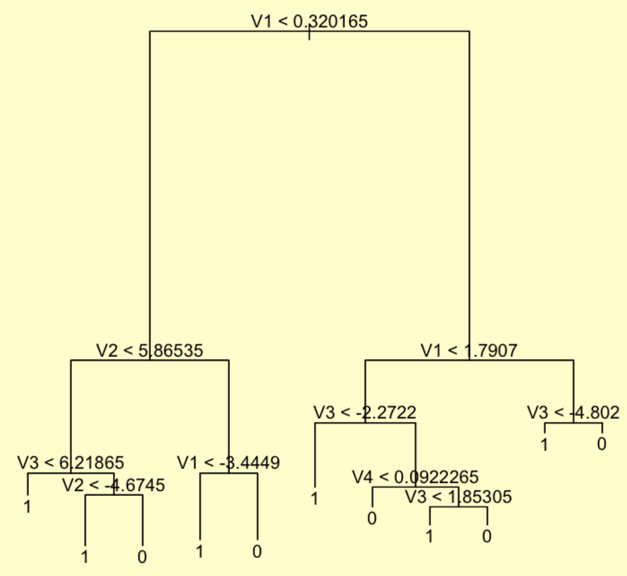
Classification Trees have several advantages over other classification algorithms. They are easy to interpret and visualize, making them useful for exploratory data analysis and for communicating results to non-technical stakeholders. They can also handle both numerical and categorical data, and can automatically handle missing values by assigning them to the most common class or by using surrogate splits.

However, Classification Trees also have some disadvantages. They can be sensitive to small variations in the training data, which can lead to overfitting if not properly regularized. They can also be biased towards features with many values or high cardinality and may not perform well on imbalanced datasets or datasets with overlapping classes.

To address these issues, several variants of Classification Trees have been developed, including Random Forests, Gradient Boosting Trees, and AdaBoost Trees. These algorithms combine multiple trees into an ensemble, which can reduce overfitting and improve the overall performance of the model.

To perform classification trees on the given dataset we first import *"tree"* library and set seed to be one and we use the function *"tree"* to perform classification trees and we plot the obtained tree using *"plot"* function.

```
library(tree)

set.seed(1)

tree_model = tree(V5 ~ ., my_data)

plot(tree_model)

text(tree_model, pretty=0)
```



We perform cross validation on the obtained tree using *"cv.tree"* function.

```
tree_model_cv = cv.tree(tree_model, FUN=prune.misclass)

tree_model_cv
```

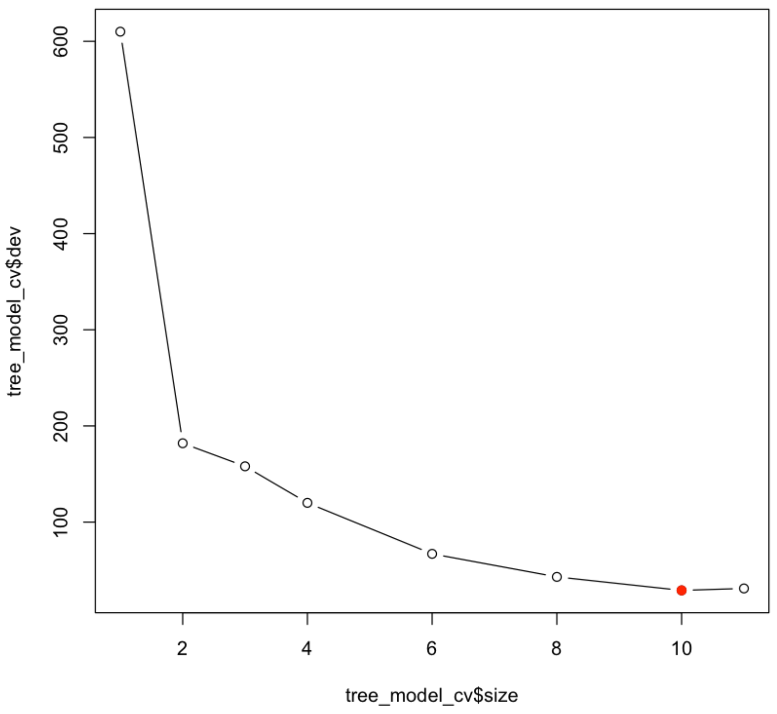
Output:

```
$size

[1] 11 10  8  6  4  3  2  1

$dev

[1]  31  29  43  67 120 158 182 610

$k

[1]  -Inf   1.0   7.0  11.5  25.5  38.0  58.0 409.0

$method

[1] "misclass"

attr(,"class")

[1] "prune"         "tree.sequence"
```

Now, we plot the cross-validation results.

```
plot(tree_model_cv$size, tree_model_cv$dev, type='b')

points(tree_model_cv$size[which.min(tree_model_cv$dev)], min(tree_model_cv$dev), col = "red", pch = 19)
```

Output:



Now we get cross-validation error and optimal tree size

```
# Extract cross-validation error and optimal tree size
print(paste('Minimum CV Error is :', min(tree_model_cv$dev), 'Occured for the tree size of',
tree_model_cv$size[which.min(tree_model_cv$dev)]))
```

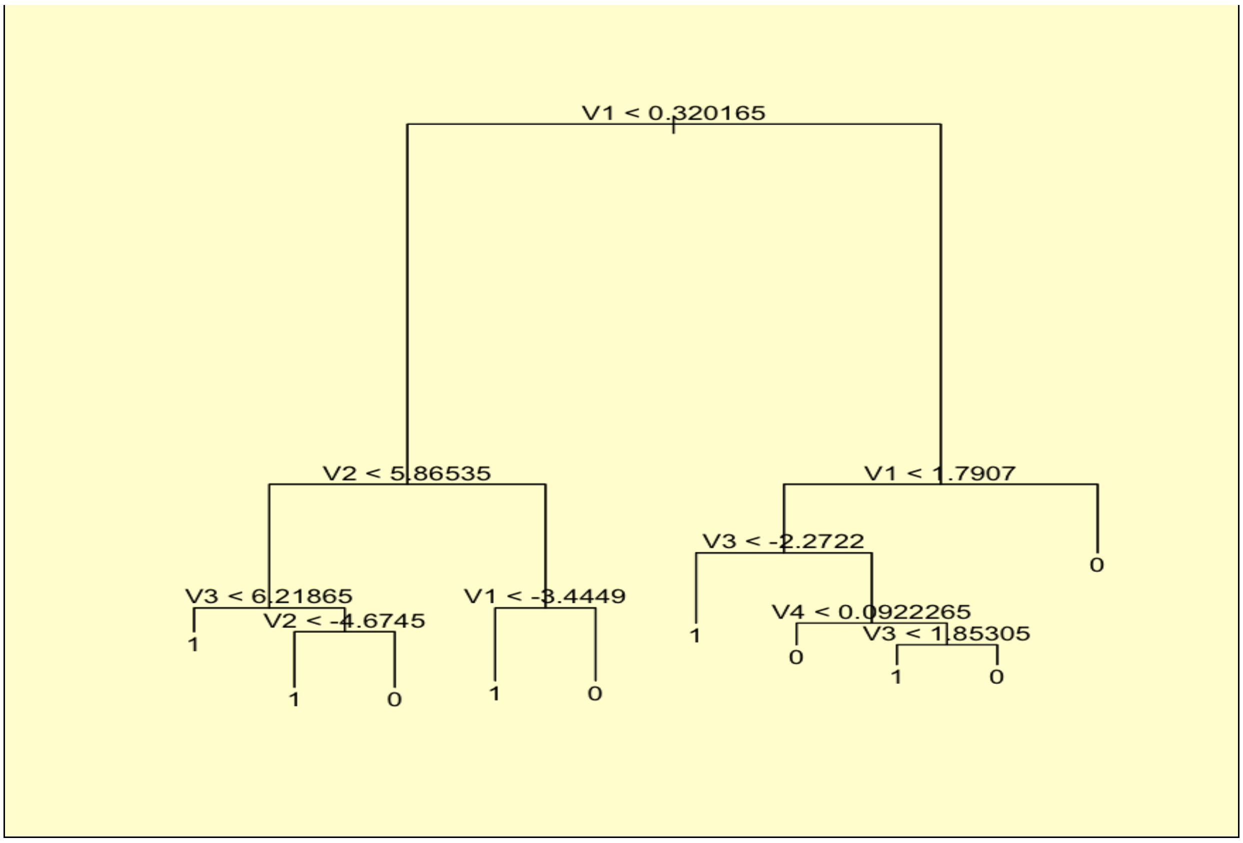
Output:

```
[1] "Minimum CV Error is : 29 Occured for the tree size of 10"
```

- After plotting tree size against error we get minimum cross-validation error as 29 for the tree size of 10

Now, we prune the tree using *"prune.tree"* and plot the pruned tree.

```
pruned_tree = prune.tree(tree_model, best = tree_model_cv$size[which.min(tree_model_cv$dev)])
plot(pruned_tree)
text(pruned_tree, pretty = 0)
```



Random Forest

Random Forest is a popular machine learning algorithm used for both classification and regression tasks. It is an ensemble learning method that combines multiple decision trees into a single model, where each tree is trained on a different random subset of the training data and a different subset of the predictor variables.

The basic idea behind Random Forest is to reduce the variance of the model by combining the predictions of multiple models with different sources of randomness. This can help to reduce overfitting and improve the accuracy and generalization performance of the model.

Random Forest works by first building a large number of decision trees, each trained on a different subset of the training data and a different subset of the predictor variables. To make a prediction for a new instance, the algorithm passes the instance through each of the decision trees and computes the average or majority vote of their predictions.

The randomization in Random Forest comes from the fact that each decision tree is trained on a different subset of the data and features. This helps to decorrelate the trees and reduce the overall variance of the model. The randomization can also help to reduce bias and improve the robustness of the model to noisy or irrelevant features.

Random Forest has several advantages over other machine learning algorithms. It is easy to use and computationally efficient, even on large datasets with high-dimensional features. It can handle both numerical and categorical data, and can automatically handle missing values and outliers. It is also robust to overfitting and can be easily tuned using cross-validation or other hyperparameter optimization methods.

However, Random Forest also has some disadvantages. It can be difficult to interpret and visualize the results, especially when dealing with high-dimensional features or interactions between features. It can also be sensitive to noisy or irrelevant features, which can reduce the accuracy and performance of the model. Finally, it can be computationally expensive to train and evaluate, especially on large datasets or with many trees.

To perform Random Forest model on the given dataset we first import libraries called *"randomForest"* and *"rfUtilities"*.

```
library(randomForest)

library(rfUtilities)
```

Output:

```
randomForest 4.7-1.1

Type rfNews() to see new features/changes/bug fixes.

Attaching package: 'randomForest'

The following object is masked from 'package:ggplot2':

   Margin
```

We get the number of predictors by using *"dim"* function and take the trees count. Then we find the cross-validation accuracy after fitting this model

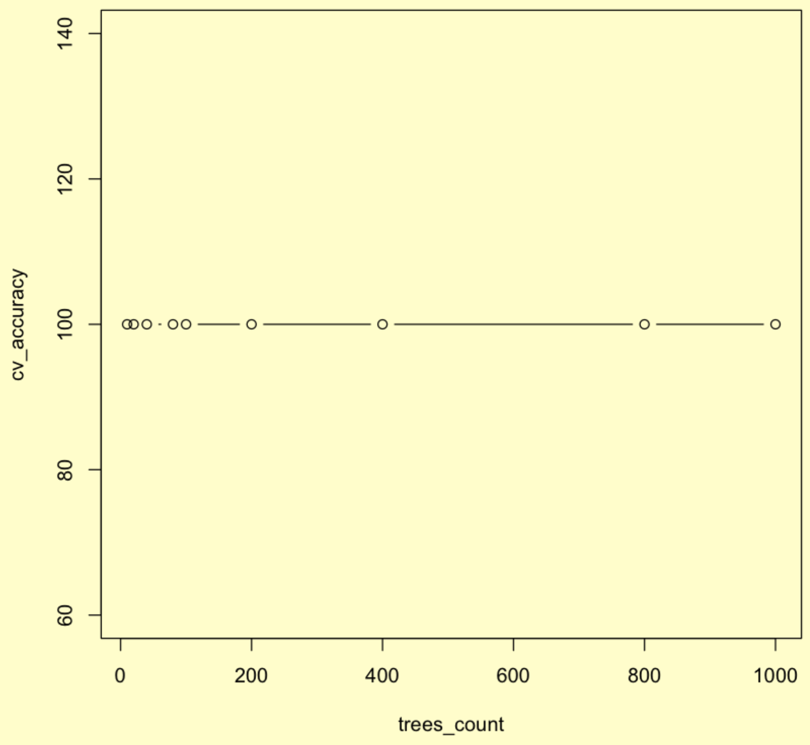
```
predictors = dim(my_data)[2] - 1

trees_count = c(10, 20, 40, 80, 100, 200, 400, 800, 1000)

cv_accuracy = c()

set.seed(1)

for (rf_tree_count in trees_count){

   # Fitting random forest model

   rf_model = randomForest(V5 ~ ., data = my_data, m_try = sqrt(predictors), ntree = rf_tree_count)

   # Perform Cross Validation

   cv_results = rf.crossValidation(rf_model, my_data, n=5)

   cv_accuracy = c(cv_accuracy, mean(apply(cv_results$cross.validation$cv.producers.accuracy, 1, mean)))

}
```

Output:

```
running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations

running: classification cross-validation with 5 iterations
```

We plot the trees count against the cross-validation accuracy.

```
plot(trees_count, cv_accuracy, type = "b")
```



We get the minimum cross-validation error and corresponding number of trees.

```
paste("Minimum CV error:", max(cv_accuracy), "at number of trees", which.max(cv_accuracy))
```

Output:

```
'Minimum CV error: 100 at number of trees 1'
```

- We obtain a minimum cross-validation error of 100 for a best selection of 1 tree

Now we get the best random forest model for the given dataset.

```
set.seed(1)

best_rf_model = randomForest(V5 ~ ., data = my_data, m_try = sqrt(predictors), ntree =
which.max(cv_accuracy))

best_rf_model
```

Output:

```
Call:

 randomForest(formula = V5 ~ ., data = my_data, m_try = sqrt(predictors),      ntree =
which.max(cv_accuracy))

               Type of random forest: classification

                     Number of trees: 1
```

```
No. of variables tried at each split: 2


        OOB estimate of  error rate: 1.17%
Confusion matrix:
    0   1 class.error
0 269   4 0.014652015
1   2 236 0.008403361
```

- We get true positive as 269, false positive as 2, true negative as 4, false negative as 236. With a class error of 0.014 and 0.008 for a dataset of 511 entries.

Support Vector Machines (SVM)

Support Vector Machines (SVM) is a popular machine learning algorithm used for both classification and regression tasks. It works by finding the optimal hyperplane that separates the data into different classes or predicts the target variable, while maximizing the margin between the classes or minimizing the regression error. In SVM, the hyperplane is defined by a set of weights (coefficients) and a bias term, which are learned from the training data using an optimization algorithm. The goal is to find the hyperplane that maximally separates the data while minimizing the classification error or regression error.

The choice of hyperplane is critical for the performance of the model. In SVM, the hyperplane that maximizes the margin between the classes is chosen, which means that the hyperplane is as far away as possible from the closest data points on each side. The data points that are closest to the hyperplane are called support vectors, and they are used to define the margin and the hyperplane.

SVM has several advantages over other machine learning algorithms. It can handle both linearly separable and non-linearly separable data by using different kernel functions, such as polynomial, radial basis function (RBF), or sigmoid. It can also handle high-dimensional data and can automatically select the most informative features using the kernel trick. Finally, SVM has a strong theoretical foundation, which can help to understand the behaviour and limitations of the model.

However, SVM also has some disadvantages. It can be sensitive to the choice of kernel function and its hyperparameters, which can affect the performance and generalization of the model. It can also be computationally expensive to train and evaluate, especially on large datasets or with complex kernels. Finally, SVM can be difficult to interpret and visualize, especially when dealing with high-dimensional data or non-linear decision boundaries.

To overcome some of the limitations of SVM, several variants and extensions have been proposed, such as support vector regression (SVR), support vector domain description (SVDD), and one-class SVM. SVR is used for regression tasks, where the goal is to predict a continuous target variable and works by finding the hyperplane that maximizes the margin between the predicted values and the actual values. SVDD is used for outlier detection and works by finding the smallest hypersphere that encloses most of the data points. One-class SVM is used for anomaly detection and works by finding the hyperplane that separates most of the data from the anomalous data points. These variants and extensions of SVM can be useful for specific applications where the data has different characteristics or requirements.

To perform support vector machine classification on the given dataset we first import *"e1071"* library and pass the data through *"svm"* function by setting the kernel to be radial, gamma to be 1 and cost to be 1e5

```
library(e1071)

svm_fit <- svm(V5 ~ ., data = my_data, kernel = "radial", gamma = 1, cost = 1e5)

summary(svm_fit)
```

Output:

```
Call:

svm(formula = V5 ~ ., data = my_data, kernel = "radial", gamma = 1,

    cost = 1e+05)

Parameters:

   SVM-Type:  C-classification

 SVM-Kernel:  radial

       cost:  1e+05

Number of Support Vectors:  79

 ( 39 40 )

Number of Classes:  2

Levels:

 0 1
```

We now tune the data using *"tune"* function.

```
tune.out <- tune(svm, V5 ~ ., data = my_data, kernel = "radial",ranges = list(cost = c(0.1, 1, 10, 100,
1000),gamma = c(0.5, 1, 2, 3, 4)))

summary(tune.out)
```

Output:

```
Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:

 cost gamma

    1   0.5

- best performance: 0

- Detailed performance results:

    cost gamma       error  dispersion

1  1e-01   0.5 0.005834127 0.009609057

2  1e+00   0.5 0.000000000 0.000000000

3  1e+01   0.5 0.000000000 0.000000000
```

```
4  1e+02   0.5 0.000000000 0.000000000
5  1e+03   0.5 0.000000000 0.000000000
6  1e-01   1.0 0.004374273 0.006152519
7  1e+00   1.0 0.000729927 0.002308232
8  1e+01   1.0 0.000729927 0.002308232
9  1e+02   1.0 0.000729927 0.002308232
10 1e+03   1.0 0.000729927 0.002308232
11 1e-01   2.0 0.003649635 0.006203187
12 1e+00   2.0 0.000729927 0.002308232
13 1e+01   2.0 0.000729927 0.002308232
14 1e+02   2.0 0.000729927 0.002308232
15 1e+03   2.0 0.000729927 0.002308232
16 1e-01   3.0 0.000000000 0.000000000
17 1e+00   3.0 0.000000000 0.000000000
18 1e+01   3.0 0.000000000 0.000000000
19 1e+02   3.0 0.000000000 0.000000000
20 1e+03   3.0 0.000000000 0.000000000
21 1e-01   4.0 0.000000000 0.000000000
22 1e+00   4.0 0.000000000 0.000000000
23 1e+01   4.0 0.000000000 0.000000000
24 1e+02   4.0 0.000000000 0.000000000
25 1e+03   4.0 0.000000000 0.000000000
```

Get the one with the minimum error

```
print(which.min(tune.out$performances[,"error"]))
```

Output:

```
[1] 2
```

Now we train the SVM model using cross-validation.

```
# Train the SVM model using cross-validation
svm_model <- svm(V5 ~ ., data = my_data,
            kernel = 'radial', cost = tune.out$performances['cost'][which.min(tune.out$performances[,"error"]), ],
            gamma = tune.out$performances['gamma'][which.min(tune.out$performances[,"error"]), ],
            type = "C-classification", cross = "10")
```

```
print(paste("Cross Validated Accuracy:",svm_model$tot.accuracy))

print(paste("Cross Validated Error:",100-svm_model$tot.accuracy))
```

Output:

```
[1] "Cross Validated Accuracy: 100"

[1] "Cross Validated Error: 0"
```

- After performing SVM model to classify the given dataset we get the cross-validation accuracy as 100% and cross-validation error as 0. This implies the data is perfectly classified using SVM model.

For the given data set, after performing LDA, QDA, Naïve Bayes, KNN, Logistic Regression, Classification Trees, Random Forest and SVM classification techniques, we see the following results.

Classification model	***LDA***	***QDA***	***Naïve Bayes***	***KNN***	***Logistic Regression***	***Classification Trees***	***Random Forest***	***SVM***
Error	***0.0233***	***0.0153***	***0: 0.1603 1: 0.0838***	***0***	***0.0102***	***0***	***0***	***0***
Accuracy	***97.67%***	***98.47%***	***0: 83.96% 1: 91.61%***	***100%***	***98.98%***	***100%***	***100%***	***100%***

From the above table we can observe that by using K-Nearest Neighbour classification, Classification trees, Random Forest classification and Support Vector classification, we classified the data in the given data set with 100% accuracy and by using other models like LDA, QDA, Logistic Regression we got accuracy more than 95% for classification. By any manner, we could successfully classify data given in the provided dataset by more than 95% accuracy.