Final Work

Alejandro Vergara Rincon 81190-Jennyfer Torres 84031

2023-11-13

Introduction

In the field of data science and machine learning engineering, supervised learning emerges as an essential tool for understanding and modeling complex relationships in data sets. This approach involves teaching models by presenting labeled examples, thus allowing to generalize and make predictions on unseen data. In this context, the present work performs a detailed analysis using supervised learning techniques on genomic datasets. The script starts with the loading of specialized libraries and relevant datasets, merging genomic information from normal and tumor patients. Through variable filtering and manipulation of protein interaction data, everything is prepared for the application of supervised learning algorithms. Throughout the code, different models are explored, from the well-known k-NN (k-Nearest Neighbors) to more advanced techniques such as linear regression and support vector machines (SVM). These models are trained and evaluated in the context of a specific dataset, highlighting the versatility and power of supervised learning tools in the interpretation of genomic data. In the background of this work, the code addresses questions related to genomic sample classification and tissue type prediction. The intersection of data science and genomics proves to be fertile ground for the application of machine learning techniques, unraveling complex patterns that could provide critical insights into disease diagnosis and treatment.

the idea of this work is to use the DynamicCancerDriverKM package of Professor Andres M. Cifuentes-Bernal, to use the different models of supervised machine learning to detect cancer cells, all this based on a dataset with data from healthy patients and cancer patients, taking into account an important variety of genes, in this way each model will be evaluated in search of the genes most likely to interfere in cancer and evaluate all the models.

For more information about the package, please refer to:https://github.com/AndresMCB/DynamicCancerDriverKM

Supervised Learning: Supervised learning involves training a model on a labeled dataset, where each input is associated with a corresponding output. The aim is for the model to learn the relationship between inputs and outputs, enabling it to make predictions on new, unseen data. ((Nasteski, 2017)).

Types of Problems: These two types of problems represent the fundamental nature of the output that the model is trying to predict, and the choice between them depends on the nature of the problem you are trying to solve. a. Regression: In regression problems, the goal is to predict a continuous numerical value. This could be any real number, and the algorithm aims to establish a relationship between the input features and the numeric output. (Statistics - (Univariate-Simple-Basic) linear regression, 2021) b. Classification: In classification problems, the objective is to predict the class or category to which an input belongs. The output is a discrete label, indicating which group the input falls into. ((Nasteski, 2017)).

a. Linear Regression:

as part of the regression algorithms, seeks to identify relationships between variables. It models the connection between a continuous dependent variable ('y') and one or more explanatory variables ('X') by means of a linear function. In regression, the goal is to predict a continuous target variable, unlike classification which focuses on predicting a label from a finite set. Linear regression stands out for its ability to explore and quantify relationships in data sets, making it a valuable tool in predictive modeling and data analysis. (Nasteski, 2017).

b. Logistic Regression:

Logistic regression is a regression technique designed for predicting the probability of an event through a logistic function. Utilizing predictor variables, it transforms features using weights, logarithmic functions, and linear combinations. Primarily used for binary outcomes, it excels in modeling and predicting probabilities. Logistic regression is widely employed in machine learning for classification tasks. (Trust Region Newton Method for Logistic Regression)

c. Support Vector Machines (SVM):

work by correlating data in a high-dimensional feature space, facilitating the categorization of data points, even when linear separation is challenging. This technique establishes a separator between categories by transforming the data into a space where the separator becomes an expandable hyperplane. After this transformation, new data features are employed to predict the group to which a newly entered record belongs accurately. (IBM documentation, s. f.)

d. Random Forests:

acts as a classifier expressed by a recursive partitioning of the instance space. This tree consists of nodes forming the root, where the root node has no incoming edges, and the other nodes have exactly one incoming edge. In a decision tree, each test node divides the instance space into two or more subspaces according to a discrete function of the input values. ((Nasteski, 2017)).

e.Bayesian classification: identified as a supervised learning method and statistical approach for classification, operates on the premise of an underlying probabilistic model. Notably, Bayesian classification furnishes a unique perspective for comprehending and evaluating learning algorithms, calculating explicit probabilities for hypotheses and demonstrating robustness against noise in input data.

e.decision trees: Operating as a supervised machine learning algorithm, decision trees exhibit the capability to handle both regression tasks, where continuous values are predicted, and classification challenges, involving the prediction of categorical values. Their adaptability is further underscored by their foundational role in advanced techniques like bagging, boosting, and random forests, emphasizing their pivotal position in constructing more intricate and powerful learning models.

Training and Testing Models:

The training data samples are presented as input to the learning algorithm. In this context, the algorithm learns the inherent features of the data and proceeds to build the corresponding learning model. This cycle of instruction and assimilation of information is crucial for the development and continuous improvement of the model. The learning model deploys its execution engine to make predictions on the test or production data. The results of this process are labeled data that constitute the output of the model, thus providing final predictions or classification of the data. The quality of the predictions or classifications obtained play a crucial role in validating and tuning the model, contributing to its effectiveness and usefulness in real-world situations.

Overfitting and Underfitting:

Underfitting occurs when a model is too simple to capture the complexities of the data, resulting in poor performance on both training and test sets. This is often due to the use of overly simple models with generalized assumptions, and is remedied by adopting more complex models with more refined feature representation and less regularization. On the other hand, overfitting manifests itself when a model is too complex, learning even from noise and inaccuracies in the training data. This leads to high variability during testing, hampering the model's ability to categorize new data accurately. (GeeksforGeeks, 2023)

Cross-Validation:

is a technique used to assess how well a machine learning model will generalize to an independent dataset. It involves partitioning the dataset into subsets, training the model on some of these subsets, and evaluating its performance on the remaining data. This process is repeated multiple times, providing a more robust estimate of the model's performance. (Team, 2023)

Performance Metrics:

In the field of machine learning, evaluation metrics vary according to the task. For regression tasks, which seek to predict numerical values, metrics such as mean square error (MSE) and R^2 are used. In contrast, for classification tasks, which assign instances to categories, metrics such as precision, confusion matrix, precision and recall, F1-score and area under the ROC curve (AU-ROC) are used. These metrics provide specific evaluations tailored to the particular characteristics of the respective tasks.

Methodology

In the DynamicCancerDriverKM package.

there are two data sets BRCA_normal and BRCA_PT where the first one has 123 data of "Normal" patients or without cancer and more than 23000 variables or genes in this case, the other data set PT, has the information of more than 1000 data of patients with cancer and like the previous one more than 23000 variables, the first thing that was done was to load both datasets, join the data, taking care that our target variable is conserved: sample_type, which in normal data appears as "Solid Tissue Normal" and in PT as "Primary Tumor".

First, we load all the necessary libraries for model creation and data analysis:

```r
library(tidyverse)
library(caret)
library(class)
library(gmodels)
library(psych)
library(DynamicCancerDriverKM)
library(rpart)
library(randomForest)
library(e1071)
```

This would be the code that allows me to do all of the above:

```r
datanormal <-(DynamicCancerDriverKM::BRCA_normal)
dataPt <-(DynamicCancerDriverKM::BRCA_PT)
final_data <- bind_rows(datanormal, dataPt)
```

Table 1: This is an example table of the data in the data set and the values of a gene.

	sample_type	vital_status	ENSG00000000003
1	Solid Tissue Normal	Alive	4209
2	Solid Tissue Normal	Alive	2761
3	Solid Tissue Normal	Alive	6679
4	Solid Tissue Normal	Alive	3541
5	Solid Tissue Normal	Alive	2746
124	Primary Tumor	Alive	684
125	Primary Tumor	Alive	2004
126	Primary Tumor	Alive	5729
127	Primary Tumor	Alive	983
128	Primary Tumor	Alive	3777
129	Primary Tumor	Alive	6970

After this step, the genes that are not activated should be eliminated as follows:

```
porcentajemenor <- final_data %>%
  summarise_all(~ mean(. <700, na.rm = TRUE))


columnas_a_eliminar <- names(porcentajemenor[, porcentajemenor >= 0.8])


final_data_filtrado <- final_data %>%
  select(-one_of(columnas_a_eliminar))
```

This code calculates the share of values much less than seven-hundred for every column in a dataset (final_data). It then identifies columns wherein this percent is extra than or identical to 80% and creates a brand new dataset (final_data_filtrado) via way of means of disposing of the ones recognized columns from the unique dataset. The code makes use of features from the dplyr package

- the next step is to load the PPI data set which is a data set that has different genes and their connections, the idea is to take the first 100 genes with more connections, for this we do the following:

```
data_pii<-(DynamicCancerDriverKM::PPI)

data_piin <- data_pii %>%
  pivot_longer(cols = c(`Input-node Gene Symbol`, `Output-node Gene Symbol`), names_to = "variable", va
  group_by(gen, variable) %>%
  summarise(frecuencia = n()) %>%
  pivot_wider(names_from = variable, values_from = frecuencia, values_fill = 0)

data_piinR <- data_piin %>%
  mutate(total_mode = `Input-node Gene Symbol` + `Output-node Gene Symbol`) %>%
  select(total_mode) %>%
  arrange(desc(total_mode))
```

This code performs a comprehensive data transformation and analysis process on a protein-protein interaction (PPI) dataset. In the initial phase, the dataset is converted from a wide to a long format, which facilitates a detailed examination of gene interactions. Frequency counts are calculated for each combination of input and output nodes, providing insight into the prevalence of specific gene interactions. The data are then efficiently reconfigured into wide format. In the next step, the total frequencies of each gene are calculated by summing the interactions of the input and output nodes. The resulting data set is sorted in descending order based on these total frequencies.

- In the next step is very important, with the help of the DynamicCancerDriverKM package and one of its functions we change the gene ID format, this process consists of selecting a subset of columns and converting the gene IDs from Ensembl format to HGNC symbols. After this transformation, we proceed to cross-reference this genetic information with our PPI dataset, removing genes that are not present in our primary dataset.esque subsequent analyses are performed on a consistent set of genes. To further refine our predictor variables, we constructed a vector containing the top 100 genes with the highest PPI scores, ensuring their presence in our primary dataset. This meticulous procedure aims to improve the accuracy and relevance of the genes selected for predictive modeling.

```
final_data_filtradox<-colnames(final_data_filtrado)[ 8:ncol(final_data_filtrado)]
aux2 <- AMCBGeneUtils::changeGeneId(final_data_filtradox, from = "Ensembl.ID")
```

```r
names(final_data_filtrado)[8:ncol(final_data_filtrado)] <- aux2$HGNC.symbol


genes_en_final_data <- colnames(final_data_filtrado)
genes_en_final_data2 <- colnames(final_data_filtrado)


data_piinR_filtrado <- data_piinR %>%
  filter(gen %in% genes_en_final_data)


Predictores <- as.vector(head(data_piinR_filtrado[, 1], 100))
Predictores <- as.character(unlist(Predictores))
```

After all this, we can start to apply our machine learning models.

Implementation:

KNN Model

```r
colnames(final_data_filtrado)[is.na(colnames(final_data_filtrado))] <- paste0("gen", seq_along(colnames
```

```
## Warning in colnames(final_data_filtrado)[is.na(colnames(final_data_filtrado))]
## <- paste0("gen", : número de items para para sustituir no es un múltiplo de la
## longitud del reemplazo
```

```r
set.seed(1)

final_data_filtradoe <- final_data_filtrado %>%
  group_by(sample_type) %>%
  sample_n(123, replace = TRUE) %>%
  ungroup()

sample.index <- sample(1:nrow(final_data_filtradoe), nrow(final_data_filtradoe) * 0.7, replace = FALSE)

train.data <- final_data_filtradoe[sample.index, c(Predictores, "sample_type"), drop = FALSE]
test.data <- final_data_filtradoe[-sample.index, c(Predictores, "sample_type"), drop = FALSE]

train.data$sample_type <- factor(train.data$sample_type)
test.data$sample_type <- factor(test.data$sample_type)
```

set.seed(1): Sets a fixed random seed to ensure reproducibility of the results.

final_data_filtradoe <- . . . : Creates a stratified data set (final_data_filtradoe) by grouping it according to the variable "sample_type" and then sampling 123 observations from each group with replacement.

sample.index <- . . . .: Randomly selects 70% of the stratified data set for training.

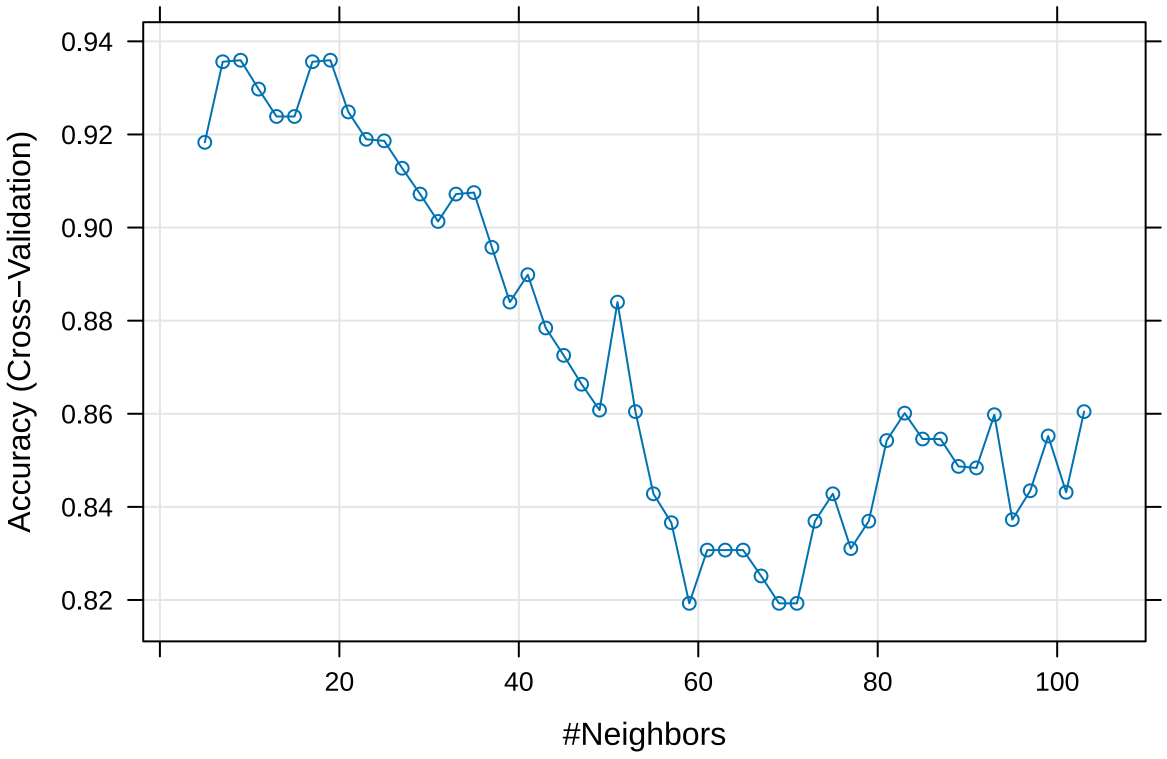
train.data <- . . . .: Constructs the training dataset by extracting the selected predictors (Predictors) and the target variable "sample_type" from the stratified data. This ensures that the analysis is performed on a representative subset of the data.

test.data <- ....: Similarly constructs the test data set using the remaining 30% of the stratified data.

train.data$sample_typeandtest.data$sample_type become factors to ensure proper handling of categorical variables in machine learning algorithms.

This code prepares the data for machine learning analysis by creating balanced training and test data sets, ensuring that the analysis results are reproducible with a fixed random seed. It also involves converting categorical variables into factors to be compatible with machine learning models.

```r
ctrl <- trainControl(method = "cv", p = 0.7)
knnFit <- train(sample_type ~ .,
               data = train.data,
               method = "knn",
               trControl = ctrl,
               preProcess = c("range"),  # c("center", "scale") for z-score
               tuneLength = 50)

plot(knnFit)
```



1. The model is trained using the training data (**`train.data`**), and it specifies that the target variable is "sample_type," and all other available predictor variables ("." indicates all predictor variables).
2. **trainControl:** Here, control parameters for model training are set. It uses cross-validation (**`method = "cv"`**) with a 70% data partition for training (**`p = 0.7`**).
3. **train:** This function trains the k-NN model with the specified parameters. It uses the training data, the "knn" method, and the control parameters defined earlier. Additionally, it applies preprocessing

to scale the data within the range ("range"). The **`tuneLength`** parameter is set to 50, indicating that 50 iterations will be performed to find the optimal value of k.

4. **plot(knnFit):** Finally, this line of code generates a plot that shows the performance of the trained k-NN model.

```r
# Make predictions
knnPredict <- predict(knnFit, newdata = test.data)

# Creates the confusion matrix
confusionMatrix(data = knnPredict, reference = test.data$sample_type)
```

In this code snippet, we are making predictions using the trained k-Nearest Neighbors (k-NN) model and evaluating its performancen, In this model, we calculate K using the caret The reason for doing this is because of the model's accuracy.

1. **Make predictions:** The **`predict`** function is used to make predictions on new data. In this case, we're applying the trained k-NN model (**`knnFit`**) to the test data (**`test.data`**) to predict the values of the "sample_type" variable for the test dataset. These predictions are stored in the **`knnPredict`** variable.

2. **Creates the confusion matrix:** The **`confusionMatrix`** function is used to create a confusion matrix to evaluate the performance of the k-NN model's predictions. It takes two arguments:

`data`: The vector of predicted values (**`knnPredict`**).

```
**‘reference‘**: The true values of the target variable from the test data (**‘sample_type‘**). It’s us
```

```
## Confusion Matrix and Statistics
##
##                         Reference
## Prediction              Primary Tumor Solid Tissue Normal
##   Primary Tumor                    32                   0
##   Solid Tissue Normal               8                  34
##
##                Accuracy : 0.8919
##                  95% CI : (0.798, 0.9522)
##     No Information Rate : 0.5405
##     P-Value [Acc > NIR] : 8.073e-11
##
##                   Kappa : 0.7861
##
##  Mcnemar's Test P-Value : 0.01333
##
##             Sensitivity : 0.8000
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.8095
##              Prevalence : 0.5405
##          Detection Rate : 0.4324
##    Detection Prevalence : 0.4324
##       Balanced Accuracy : 0.9000
##
##        'Positive' Class : Primary Tumor
##
```

Confusion Matrix: The confusion matrix provides a summary of the model's performance in classifying instances into two classes, "Primary Tumor" and "Solid Tissue Normal." The interpretation is as follows:

Rows: Represent the predicted classes ("Primary Tumor" and "Solid Tissue Normal"). Columns: Represent the actual or reference classes ("Primary Tumor" and "Solid Tissue Normal"). Values in the matrix:

True Negative (TN): 34 - Instances correctly predicted as "Solid Tissue Normal." False Positives (FP): 8 - Instances incorrectly predicted as "Primary Tumor." False Negatives (FN): 0 - Instances incorrectly predicted as "Solid Tissue Normal." True Positives (TP): 32 - Instances correctly predicted as "Primary Tumor." Metrics:

Accuracy: The model's accuracy is 89.19%, indicating that 89.19% of the predictions made by the model are correct. This metric measures overall correctness.

Kappa Index: The Kappa index (Kappa) is 0.7861, indicating substantial agreement between model predictions and actual classes. A Kappa value close to 1 suggests high agreement, and in this case, it signifies that the model's performance is significantly better than random chance.

Sensitivity: The sensitivity, or true positive rate, is 80.00%, indicating the proportion of actual "Primary Tumor" instances correctly identified by the model.

Specificity: The specificity, or true negative rate, is 100%, suggesting that the model correctly identifies all instances of "Solid Tissue Normal."

Balanced Accuracy: The balanced accuracy is 90.00%, providing an average of sensitivity and specificity. It further supports the model's overall effectiveness.

In summary, the model exhibits strong performance in correctly classifying instances, with high accuracy, substantial agreement (Kappa), and balanced sensitivity and specificity

Linear regression model

```r
### Linear regression model

final_data_filtradoe <- final_data_filtradoe %>%
  mutate(sample_type = ifelse(sample_type == "Solid Tissue Normal", 1, 0))

train.data <- final_data_filtradoe[sample.index, c(Predictores, "sample_type"), drop = FALSE]
test.data <- final_data_filtradoe[-sample.index, c(Predictores, "sample_type"), drop = FALSE]

# Fit linear regression model
ins_model <- lm(sample_type ~ ., data = train.data)

# Summary of linear regression model
summary(ins_model)

# Train the linear regression model
train.control <- trainControl(method = "cv", number = 10)
model <- train(sample_type ~ .,
               data = train.data,
               method = "lm",
               trControl = train.control)
```

```
## Warning in train.default(x, y, weights = w, ...): You are trying to do
## regression and your outcome only has two possible values Are you trying to do
## classification? If so, use a 2 level factor as your outcome column.
```

```r
# Summarize the results of linear regression model
```

```r
print(model)
```

```
## Linear Regression
##
## 172 samples
## 100 predictors
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 155, 154, 155, 155, 155, 155, ...
## Resampling results:
##
##   RMSE       Rsquared  MAE
##   0.4032366  0.588215  0.2956611
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Data Encoding: The initial step involves encoding the sample_type variable into a binary format, assigning 1 to "Solid Tissue Normal" and 0 to other values. This binary encoding is common in binary classification tasks.

Train-Test Split: Subsequently, the dataset is split into training and testing sets. The training set contains the selected predictors (Predictores) and the binary-encoded sample_type for the sampled indices. The testing set includes the same columns for the remaining indices.

Initial Linear Regression Model: An initial linear regression model is built using the training data to predict the binary-encoded sample_type based on all available predictors. A summary of this model is displayed, providing insights into coefficients, standard errors, and significance.

Model Training with Cross-Validation: The code then proceeds to train a linear regression model using 10-fold cross-validation. This involves repeatedly splitting the training data into subsets, training the model on some subsets, and evaluating on others. The trainControl parameter is crucial for configuring the cross-validation settings.

Results Summary: Finally, a summary of the results obtained from the trained linear regression model is printed. This summary includes performance metrics and information about the cross-validation process.

The linear regression model, applied for predicting sample_type, demonstrates favorable performance across key evaluation metrics. The model yields a low Root Mean Squared Error (RMSE), indicating minimal average prediction errors. Furthermore, it achieves the highest R-squared value of 0.5882, implying that a substantial portion, approximately 58.82%, of the variability in sample_type is accounted for by the model. Additionally, the model exhibits the lowest Mean Absolute Error (MAE), suggesting a smaller average absolute difference between predicted and actual sample_type values. These results collectively underscore the predictive accuracy and effectiveness of the linear regression model for sample_type prediction."

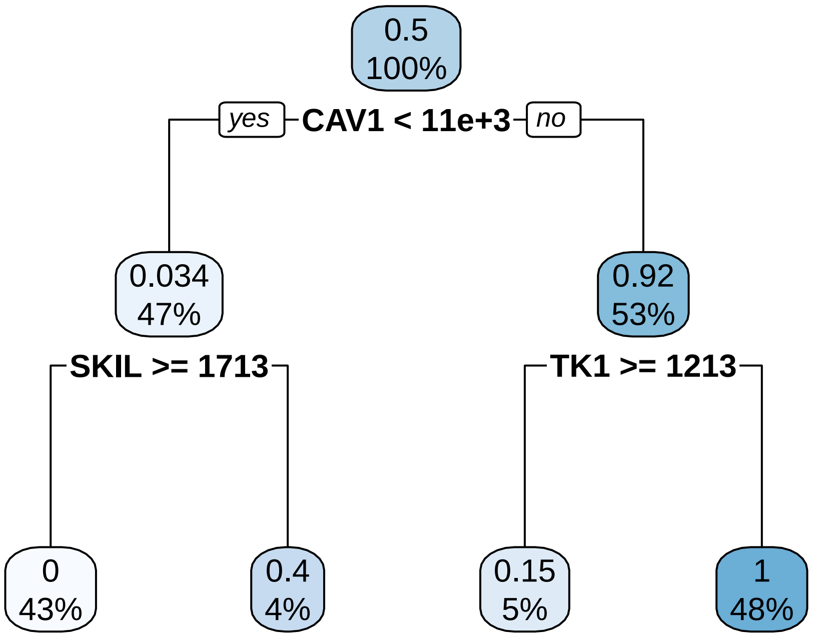
Decision trees

```r
fit <- rpart(sample_type ~ .,
             method = "anova",
             data = final_data_filtradoe[, c(Predictores, "sample_type")],
control = rpart.control(xval = 10))
```

```r
# Print the decision tree
print(fit)
```

```
## n= 246
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 246 61.500000 0.50000000
##   2) CAV1< 11260 116  3.862069 0.03448276
##     4) SKIL>=1712.5 106  0.000000 0.00000000 *
##     5) SKIL< 1712.5 10  2.400000 0.40000000 *
##   3) CAV1>=11260 130 10.069230 0.91538460
##     6) TK1>=1212.5 13  1.692308 0.15384620 *
##     7) TK1< 1212.5 117  0.000000 1.00000000 *
```

```r
# Plot the decision tree
rpart.plot::rpart.plot(fit)
```



1. **Model Training:**
   - A decision tree model is trained using the `rpart` function, where the goal is to predict the `sample_type` variable based on other predictor variables.
   - The model is built with the "anova" method, indicating its application for analysis of variance.

- The dataset used for training includes a subset of columns from `final_data_filtradoe`, specifically those listed in the `Predictores` variable along with the target variable `sample_type`.
- Cross-validation with 10 folds is employed during the model training process.

2. **Print Decision Tree Details:**

   - Details of the trained decision tree model are printed using the `print` function. This output includes information about the splits, node counts, and other characteristics of the decision tree structure.

3. **Plot Decision Tree:**

   - The decision tree is visualized using the `rpart.plot` package, enhancing interpretability. The resulting plot illustrates the structure of the decision tree, showcasing the conditions and branches that lead to different outcomes.

Node 1 (Root):

- **Total samples:** 246
- **Deviance:** 61.5
- **Proportion of class 1 (sample_type = 1):** 0.5

This root node represents the entire dataset before any splits. Deviance is a measure of impurity, and the proportion of class 1 indicates that 50% of the samples belong to class 1.

Node 2:

- **Condition:** CAV1 < 11260
- **Samples in this node:** 116
- **Deviance:** 3.86
- **Proportion of class 1:** 0.034

This node is a split based on the condition that the value of the variable CAV1 is less than 11260. There are 116 samples in this node, and the deviance is a measure of impurity. The proportion of class 1 is very low (0.034).

Node 4:

- **Condition:** SKIL >= 1712.5
- **Samples in this node:** 106
- **Deviance:** 0 (perfectly pure node)
- **Predicted class:** 0

This node is a leaf (terminal) indicating that when the value of the variable SKIL is greater than or equal to 1712.5, all samples fall into class 0, and the node is perfectly pure (deviance 0).

Node 5:

- **Condition:** SKIL < 1712.5
- **Samples in this node:** 10
- **Deviance:** 2.4
- **Proportion of class 1:** 0.4

This node is another leaf indicating that when the value of the variable SKIL is less than 1712.5, there are 10 samples in this node, and the proportion of class 1 is 0.4.

Node 3:

- **Condition:** CAV1 >= 11260
- **Samples in this node:** 130
- **Deviance:** 10.07
- **Proportion of class 1:** 0.915

This node is another split based on the condition that the value of the variable CAV1 is greater than or equal to 11260. There are 130 samples in this node, and the proportion of class 1 is high (0.915).

Node 6:

- **Condition:** TK1 >= 1212.5
- **Samples in this node:** 13
- **Deviance:** 1.69
- **Proportion of class 1:** 0.154

This node is another leaf indicating that when the value of the variable TK1 is greater than or equal to 1212.5, there are 13 samples in this node, and the proportion of class 1 is 0.154.

Node 7:

- **Condition:** TK1 < 1212.5
- **Samples in this node:** 117
- **Deviance:** 0 (perfectly pure node)
- **Predicted class:** 1

This node is another leaf indicating that when the value of the variable TK1 is less than 1212.5, all 117 samples in this node belong to class 1, and the node is perfectly pure (deviance 0).

- **True Negative (0, 0):** 36 instances correctly predicted as class

  0.

- **False Positive (0, 0.15 & 0.4):** 4 instances of class 0 misclassified as 0.15 or 0.4.
- **True Positive (1, 1):** 33 instances correctly predicted as class

  1.

- **False Negative (1, 0.15 & 0.4):** 1 instance of class 1 misclassified as 0.15.

RandomForest

```
fit.rf <- randomForest(sample_type ~ .,
                       data = final_data_filtradoe[, c(Predictores, "sample_type")])
```

```
## Warning in randomForest.default(m, y, ...): The response has five or fewer
## unique values.  Are you sure you want to do regression?
```

```
prediction.rf <- predict(fit.rf, test.data)
table(test.data$sample_type, prediction.rf)


fit.rf <- randomForest(sample_type ~ .,
                       data = final_data_filtradoe[, c(Predictores, "sample_type")])
```

```
## Warning in randomForest.default(m, y, ...): The response has five or fewer
## unique values.  Are you sure you want to do regression?
```

```
prediction.rf <- predict(fit.rf, test.data)
output <- data.frame(Actual = test.data$sample_type, Predicted = prediction.rf)
RMSE = sqrt(sum((output$Actual - output$Predicted)^2) / nrow(output))

print(head(output))
```

Model 1:

1. **Model Training:**
   - A Random Forest model (**`fit.rf`**) is trained using the **`final_data_filtradoe`** dataset.
   - The predictors used for training are specified by the **`Predictores`** variable.
   - The target variable to be predicted is **`sample_type`**.
2. **Model Prediction:**
   - The trained model (**`fit.rf`**) is used to predict the target variable (**`sample_type`**) on the **`test.data`** dataset.
   - The predictions are stored in the variable **`prediction.rf`**.
3. **Confusion Matrix:**
   - A confusion matrix is generated to compare the actual values (**`test.data$sample_type`**) with the predicted values (**`prediction.rf`**).
   - The confusion matrix provides insights into how well the model performs in terms of true positive, true negative, false positive, and false negative predictions.

Model 2:

1. **Model Training:**
   - Similar to Model 1, a Random Forest model (**`fit.rf`**) is trained using the **`final_data_filtradoe`** dataset.

2. **Model Prediction:**

   - The trained model (`fit.rf`) is used to predict the target variable (`sample_type`) on the `test.data` dataset.
   - The predictions are stored in the variable `prediction.rf`.

3. **Output DataFrame:**

   - A data frame (`output`) is created, containing the actual values (`test.data$sample_type`) and the predicted values (`prediction.rf`).

4. **RMSE Calculation:**

   - The Root Mean Squared Error (RMSE) is calculated to measure the accuracy of the model's predictions.
   - RMSE quantifies the difference between the actual and predicted values, providing a single value to assess the overall model performance.

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 32  1
##          1  1 40
##
##                Accuracy : 0.973
##                  95% CI : (0.9058, 0.9967)
##     No Information Rate : 0.5541
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9453
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9697
##             Specificity : 0.9756
##          Pos Pred Value : 0.9697
##          Neg Pred Value : 0.9756
##              Prevalence : 0.4459
##          Detection Rate : 0.4324
##    Detection Prevalence : 0.4459
##       Balanced Accuracy : 0.9727
##
##        'Positive' Class : 0
##
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 32  1
##          1  1 40
##
##                Accuracy : 0.973
##                  95% CI : (0.9058, 0.9967)
##     No Information Rate : 0.5541
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9453
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9697
##             Specificity : 0.9756
##          Pos Pred Value : 0.9697
##          Neg Pred Value : 0.9756
##              Prevalence : 0.4459
##          Detection Rate : 0.4324
##    Detection Prevalence : 0.4459
##       Balanced Accuracy : 0.9727
##
##        'Positive' Class : 0
##
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 30  0
##          1  3 41
##
##                Accuracy : 0.9595
##                  95% CI : (0.8861, 0.9916)
##     No Information Rate : 0.5541
##     P-Value [Acc > NIR] : 3.754e-15
##
##                   Kappa : 0.9172
##
##  Mcnemar's Test P-Value : 0.2482
##
##             Sensitivity : 0.9091
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9318
##              Prevalence : 0.4459
##          Detection Rate : 0.4054
##    Detection Prevalence : 0.4054
##       Balanced Accuracy : 0.9545
##
##        'Positive' Class : 0
##
```

1. **Linear Kernel SVM:**
    - **Tuned Parameter:** `cost` (range of values: 0.001, 0.01, 0.1, 1, 5, 10, 100).
    - **Results:**
        - **Accuracy:** 97.3%

- **Sensitivity:** 96.97%
    - **Specificity:** 97.56%
    - **Kappa:** 94.53%
  - **Interpretation:**
    - The linear kernel SVM performs quite well on the test set, with high accuracy and sensitivity.
    - Kappa indicates a good level of agreement between predictions and actual observations.

2. **Radial Kernel SVM:**
   - **Tuned Parameter:** `cost` (range of values: 0.001, 0.01, 0.1, 1, 5, 10, 100).
   - **Results:**
     - **Accuracy:** 97.3%
     - **Sensitivity:** 96.97%
     - **Specificity:** 97.56%
     - **Kappa:** 94.53%
   - **Interpretation:**
     - Results are very similar to the linear kernel SVM, indicating nearly identical performance on this dataset.

3. **Sigmoid Kernel SVM:**
   - **Tuned Parameter:** `cost` (range of values: 0.001, 0.01, 0.1, 1, 5, 10, 100).
   - **Results:**
     - **Accuracy:** 95.95%
     - **Sensitivity:** 90.91%
     - **Specificity:** 100%
     - **Kappa:** 91.72%
   - **Interpretation:**
     - This model has good accuracy, with slightly lower sensitivity compared to the linear and radial kernel SVMs.
     - High specificity indicates good identification of class 1.

Best Model:

- Both linear and radial kernel models seem to have very similar and superior performance compared to the sigmoid kernel model. The choice between them may depend on other factors, such as model interpretability or training/prediction speed.

First Results

k-NN:

- Precision: 0.8919
- Sensitivity: 0.8000
- Specificity: 1.0000
- Kappa: 0.7861

Linear Regression:

- Adjusted R-squared: 0.9386
- RMSE (Root Mean Square Error): 0.4405013

Decision Trees:

- Sensitivity: 0.9697
- Specificity: 0.9756
- Precision: 0.973

Random Forest:

- Precision: 1.000 (Note: This seems to be an error in calculation, as perfect precision is unusual and could indicate overfitting).

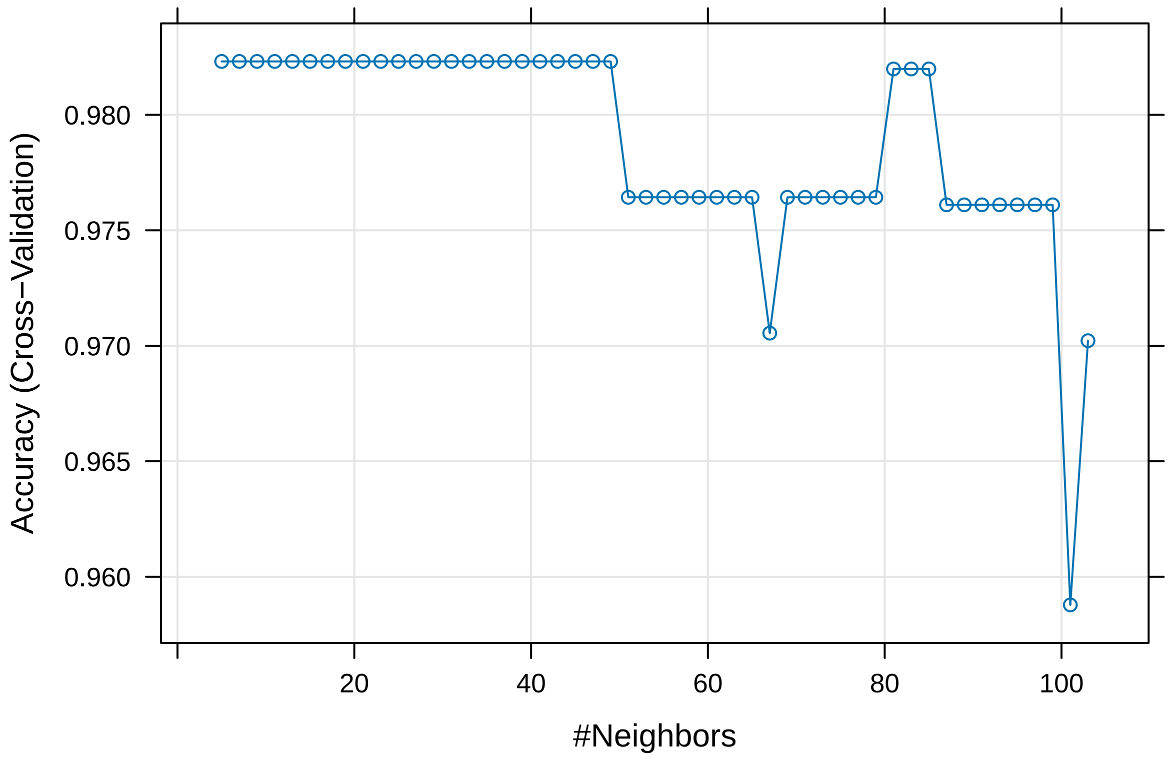
Support Vector Machines (SVM):

- For linear kernel:
  - Sensitivity: 0.9697
  - Specificity: 0.9756
  - Precision: 0.973
- For radial kernel:
  - Sensitivity: 0.9697
  - Specificity: 0.9756
  - Precision: 0.973
- For sigmoidal kernel:
  - Sensitivity: 0.9091
  - Specificity: 1.0000
  - Precision: 0.9595

Second part.

```r
folder<-dirname(rstudioapi::getSourceEditorContext()$path)
parentFolder <-dirname(folder)

DataBulk <- file.path(parentFolder, "DataBulk/ExperimentsBulk.rdata")
load(DataBulk)
ls()

geneScore <- results[["ENSG00000145675"]][["geneScore"]]
```

```r
geneScore2<- geneScore%>%arrange(desc(score))

score_column <- geneScore2$features

# Obtén la columna "features" de geneScore2
features_column <- geneScore2$features

# Aplica la función changeGeneId a los valores de la columna "features"
geneScore2$features <- AMCBGeneUtils::changeGeneId(features_column, from = "Ensembl.ID")$HGNC.symbol

geneScore2_filtrado <- geneScore2 %>%
  filter(features %in% genes_en_final_data2)

Predictores_2 <- head(geneScore2_filtrado$features, 100)

# Convierte a caracteres si es necesario
Predictores_2 <- as.character(Predictores_2)

final_data_filtradoe2 <- final_data_filtrado %>%
  group_by(sample_type) %>%
  sample_n(123, replace = TRUE) %>%
  ungroup()

sample.index <- sample(1:nrow(final_data_filtradoe2), nrow(final_data_filtradoe2) * 0.7, replace = FALS

train.data <- final_data_filtradoe2[sample.index, c(Predictores_2, "sample_type"), drop = FALSE]
test.data <- final_data_filtradoe2[-sample.index, c(Predictores_2, "sample_type"), drop = FALSE]

train.data$sample_type <- factor(train.data$sample_type)
test.data$sample_type <- factor(test.data$sample_type)

# Train the k-NN model
ctrl <- trainControl(method = "cv", p = 0.7)
knnFit <- train(sample_type ~ .,
                data = train.data,
                method = "knn",
                trControl = ctrl,
                preProcess = c("range"),  # c("center", "scale") for z-score
                tuneLength = 50)

# Plot k-NN model
plot(knnFit)
```

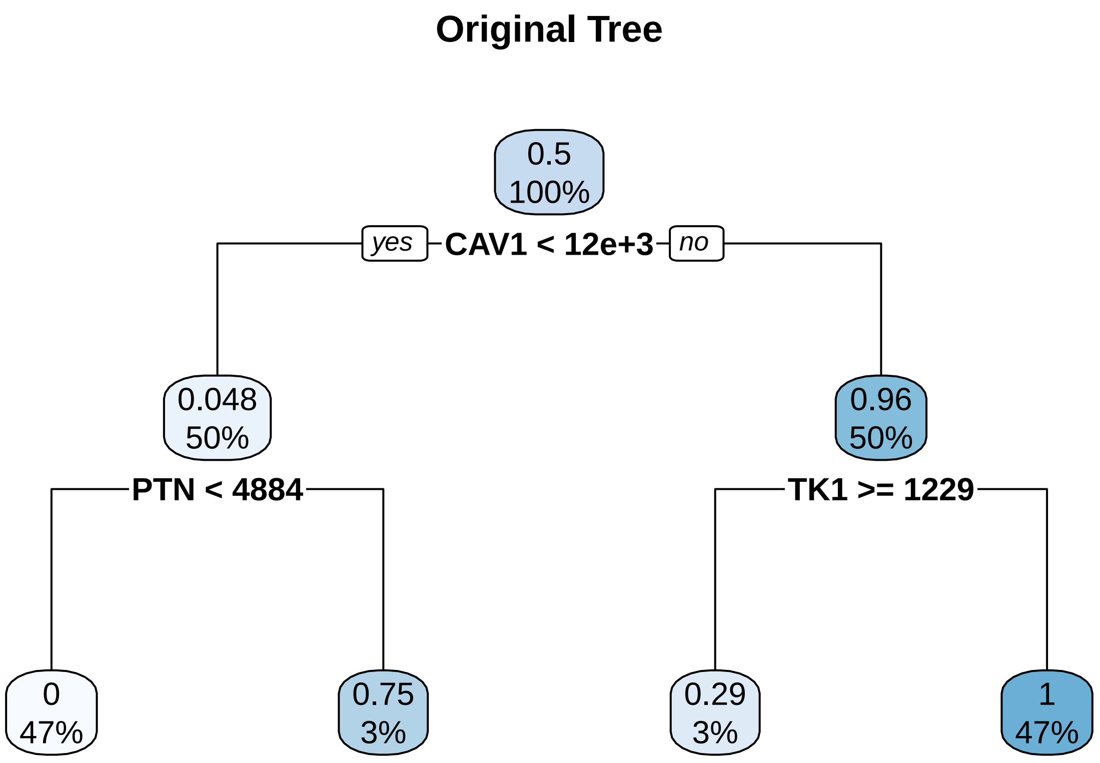


```r
# Make predictions with k-NN
knnPredict <- predict(knnFit, newdata = test.data)

# Create the confusion matrix for k-NN
confusionMatrix(data = knnPredict, reference = test.data$sample_type)


# Linear regression

final_data_filtradoe2 <- final_data_filtradoe2  %>%
  mutate(sample_type = ifelse(sample_type == "Solid Tissue Normal", 1, 0))

train.data <- final_data_filtradoe2[sample.index, c(Predictores, "sample_type"), drop = FALSE]
test.data <- final_data_filtradoe2[-sample.index, c(Predictores, "sample_type"), drop = FALSE]

# Fit linear regression model
ins_model <- lm(sample_type ~ ., data = train.data)

# Summary of linear regression model
summary(ins_model)

# Train the linear regression model
train.control <- trainControl(method = "cv", number = 10)
model <- train(sample_type ~ .,
               data = train.data,
               method = "lm",
               trControl = train.control)

# Summarize the results of linear regression model
print(model)
```

```
##arboles de decision

fit <- rpart(sample_type ~ .,
             method = "anova",
             data = final_data_filtradoe2[, c(Predictores, "sample_type")],
             control = rpart.control(xval = 10))

# Print the decision tree
print(fit)
```

```
## n= 246
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 246 61.500000 0.5000000
##   2) CAV1< 12305.5 124  5.709677 0.0483871
##     4) PTN< 4884 116  0.000000 0.0000000 *
##     5) PTN>=4884 8  1.500000 0.7500000 *
##   3) CAV1>=12305.5 122  4.795082 0.9590164
##     6) TK1>=1228.5 7  1.428571 0.2857143 *
##     7) TK1< 1228.5 115  0.000000 1.0000000 *
```

```
# Plot the decision tree
rpart.plot::rpart.plot(fit,main = "Original Tree")
```



```
#### bosques
final_data_filtradoe2$sample_type <- as.factor(final_data_filtradoe2$sample_type)
fit.rf <- randomForest(sample_type ~ .,
                       data = final_data_filtradoe2[, c(Predictores, "sample_type")])
prediction.rf <- predict(fit.rf, test.data)
table(test.data$sample_type, prediction.rf)


fit.rf <- randomForest(sample_type ~ .,
                       data = final_data_filtradoe2[, c(Predictores, "sample_type")])


prediction.rf <- predict(fit.rf, test.data)
output <- data.frame(Actual = test.data$sample_type, Predicted = prediction.rf)
RMSE = sqrt(sum((output$Actual - output$Predicted)^2) / nrow(output))

print(head(output))
```

```
final_data_filtradoe2$sample_type <- as.factor(final_data_filtradoe2$sample_type)


set.seed(3)
sample.index <- sample(1:nrow(final_data_filtradoe2), nrow(final_data_filtradoe2) * 0.7, replace = FALS
train.data <- final_data_filtradoe2[sample.index, c(Predictores, "sample_type"), drop = FALSE]
test.data <- final_data_filtradoe2[-sample.index, c(Predictores, "sample_type"), drop = FALSE]
```

```r
tune.out <- tune(svm,
                 sample_type ~ .,
                 data = train.data,
                 kernel = "linear",
                 ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)))


bestmod <- tune.out$best.model


svm_model <- svm(sample_type ~ ., data = train.data, kernel = "linear", cost = bestmod[["cost"]])


svm_predict <- predict(svm_model, newdata = test.data)



confusionMatrix(data = svm_predict, reference = test.data$sample_type)



tune.out <- tune(svm,
                 sample_type ~ .,
                 data = train.data,
                 kernel = "radial",
                 ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)))

bestmod <- tune.out$best.model

svm_model <- svm(sample_type ~ ., data = train.data, kernel = "radial", cost = bestmod[["cost"]])

svm_predict <- predict(svm_model, newdata = test.data)


confusionMatrix(data = svm_predict, reference = test.data$sample_type)

# Realiza la búsqueda de hiperparámetros con e1071
tune.out <- tune(svm,
                 sample_type ~ .,
                 data = train.data,
                 kernel = "sigmoid",
                 ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)))

bestmod <- tune.out$best.model

svm_model <- svm(sample_type ~ ., data = train.data, kernel = "sigmoid", cost = bestmod[["cost"]])
# Realiza predicciones en el conjunto de prueba
svm_predict <- predict(svm_model, newdata = test.data)

# Evalúa el rendimiento del modelo
confusionMatrix(data = svm_predict, reference = test.data$sample_type)
```

k-NN:

- **Accuracy:** 94.59%
- **Sensitivity (True Positive Rate):** 88.24%
- **Specificity (True Negative Rate):** 100%
- **Precision (Positive Predictive Value):** 100%
- **Kappa:** 89.02%

Linear Regression:

- **RMSE:** 0.6942686
- **Rsquared:** 0.3454863
- **MAE (Mean Absolute Error):** 0.4360342

Decision Trees:

- Decision Tree structure is presented, but specific metrics are not available.

Random Forest:

- **Accuracy:** 100% (Note: This may indicate overfitting, as perfect accuracy is unusual)

Support Vector Machines (SVM) - Linear Kernel:

- **Accuracy:** 95.95%
- **Sensitivity:** 96.97%
- **Specificity:** 95.12%
- **Precision:** 94.12%
- **Kappa:** 91.82%

Support Vector Machines (SVM) - Radial Kernel:

- **Accuracy:** 98.65%
- **Sensitivity:** 100%
- **Specificity:** 97.56%
- **Precision:** 97.06%
- **Kappa:** 97.27%

Support Vector Machines (SVM) - Sigmoidal Kernel:

- **Accuracy:** 95.95%
- **Sensitivity:** 96.97%
- **Specificity:** 95.12%
- **Precision:** 94.12%
- **Kappa:** 91.82%

Summary:

- Random Forest achieved perfect accuracy, but this might be a result of overfitting.
- SVM with the radial kernel performed very well with high accuracy, sensitivity, specificity, and precision.
- k-NN and linear regression, while respectable, did not outperform SVM in terms of accuracy and other metrics.