Tecnológico Nacional de México

Instituto Tecnológico de Tijuana

Subdirección académica

Departamento de Sistemas y Computación

Semestre: AGOSTO - DICIEMBRE 2019

Ingeniería en Tecnologías de la Información y Comunicación

Materia: Datos Masivos

# Proyecto Final

Nombre: Vite Hernández José Omar

No. Control: 15211706

Docente: José Christian Romero Hernandez

# Índice

## Introducción

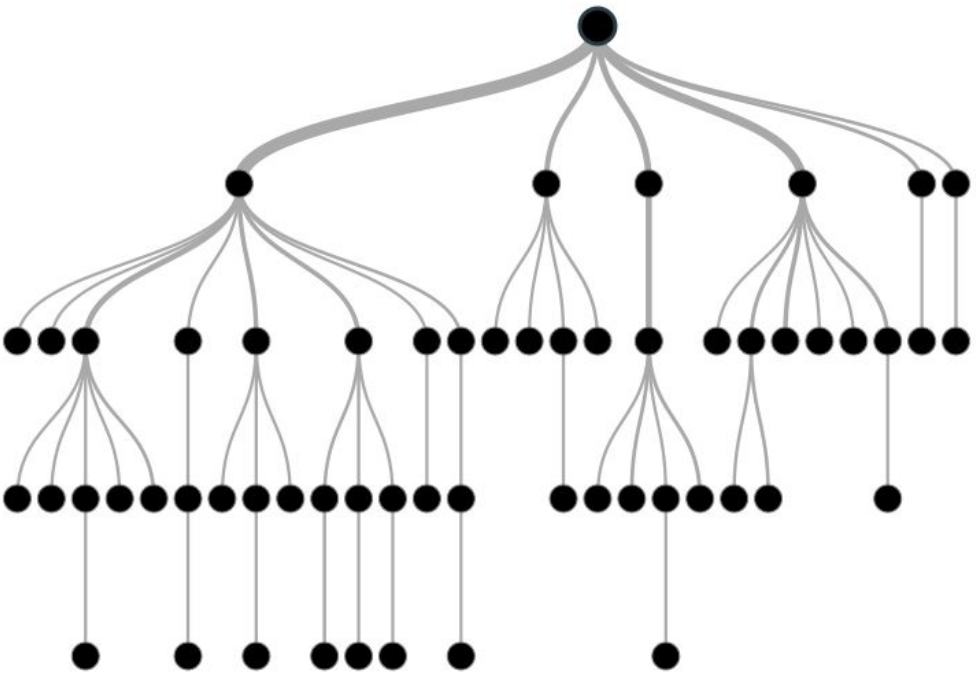
El presente documento es el resultado de una investigación sobre 3 diferentes tipos de aprendizaje automático en el rubro de la clasificación, estos algoritmos fueron desarrollados en el lenguaje de programación Scala con librerías (Machine Learning Library) de Spark en su versión 2.4.4

## Marco teórico de los algoritmos

## Arbol de decision

Un árbol de decisión es una serie de nodos, un gráfico direccional que comienza en la base con un solo nodo y se extiende a los muchos nodos hoja que representan las categorías que el árbol puede clasificar. Otra forma de pensar en un árbol de decisión es como un diagrama de flujo, donde el flujo comienza en el nodo raíz y termina con una decisión tomada en las hojas. Es una herramienta de apoyo a la decisión. Utiliza un gráfico en forma de árbol para mostrar las predicciones que resultan de una serie de divisiones basadas en características.

Aquí hay algunos términos útiles para describir un árbol de decisión:



Nodo raíz: un nodo raíz está al comienzo de un árbol. Representa a toda la población que se analiza. Desde el nodo raíz, la población se divide de acuerdo con varias características, y esos subgrupos se dividen a su vez en cada nodo de decisión debajo del nodo raíz.

División: es un proceso de división de un nodo en dos o más subnodos.

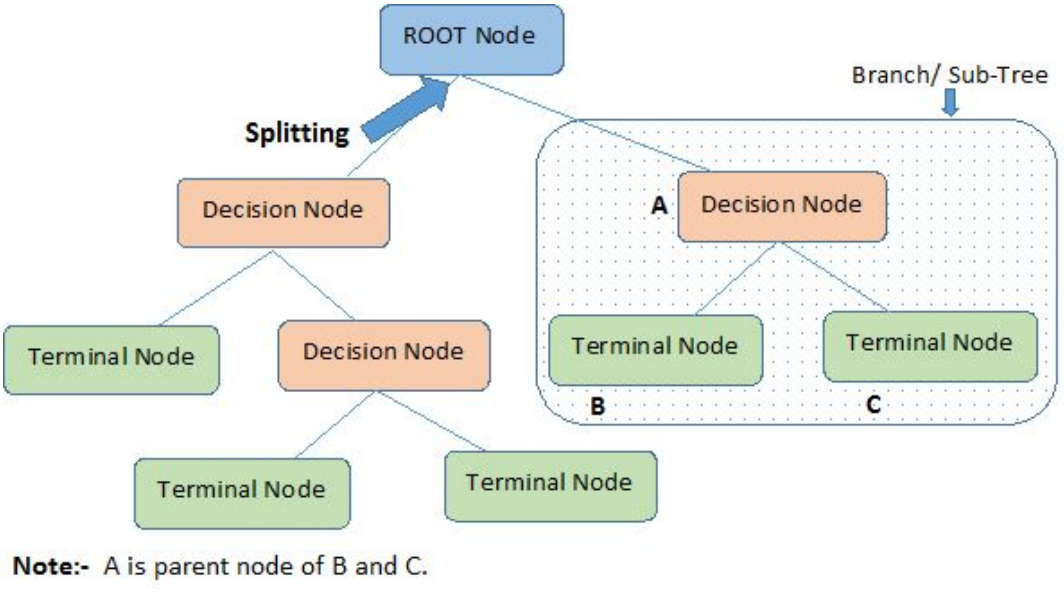
Nodo de decisión: cuando un subnodo se divide en subnodos adicionales, es un nodo de decisión.

Nodo hoja o nodo terminal: los nodos que no se dividen se denominan nodos hoja o terminal.

Poda: la eliminación de los subnodos de un nodo primario se denomina poda. Un árbol se cultiva mediante la división y se contrae mediante la poda.

Rama o subárbol: una subsección del árbol de decisión se denomina rama o subárbol, del mismo modo que una parte de un gráfico se denomina subgrafo.

Nodo padre y nodo hijo: estos son términos relativos. Cualquier nodo que se encuentre debajo de otro nodo es un nodo secundario o subnodo, y cualquier nodo que preceda a esos nodos secundarios se llama nodo primario.



## Regresión Logística

La regresión logística resulta útil para los casos en los que se desea predecir la presencia o ausencia de una característica o resultado según los valores de un conjunto de predictores. Es similar a un modelo de regresión lineal pero está adaptado para modelos en los que la variable dependiente es dicotómica. Los coeficientes de regresión logística pueden utilizarse para estimar la odds ratio de cada variable independiente del modelo. La regresión logística se puede aplicar a un rango más amplio de situaciones de investigación que el análisis discriminante.

Ejemplo. ¿Qué características del estilo de vida son factores de riesgo de enfermedad cardiovascular ? Dada una muestra de pacientes a los que se mide la situación de fumador, dieta, ejercicio, consumo de alcohol, y estado de enfermedad cardiovascular, se puede generar un modelo utilizando las cuatro variables de estilo de vida para predecir la presencia o ausencia de enfermedad cardiovascular en una muestra de pacientes. El modelo puede utilizarse posteriormente para derivar estimaciones de la odds ratio para cada uno de los factores y así indicarle, por ejemplo, cuánto más probable es que los fumadores desarrollan una enfermedad cardiovascular frente a los no fumadores.

Estadísticos. Casos totales, Casos seleccionados, Casos válidos. Para cada variable categórica: parámetro coding. Para cada paso: variables introducidas o eliminadas, historial de iteraciones, -2 log de la verosimilitud, bondad de ajuste, estadístico de bondad de ajuste de Hosmer-Lemeshow, chi-cuadrado del modelo ¡, chi-cuadrado de la mejora, tabla de clasificación, correlaciones entre las variables, gráfico de las probabilidades pronosticadas y los grupos observados, chi-cuadrado residual. Para las variables de la ecuación: coeficiente (B), error estándar de B, Estadístico de Wald, razón de las ventajas estimada (exp(B)), intervalo de confianza para exp(B), log de la verosimilitud si el término se ha eliminado del modelo. Para cada variable que no esté en la ecuación: estadístico de puntuación. Para cada caso: grupo observado, probabilidad pronosticada, grupo pronosticado, residuo, residuo estandarizado.

Métodos. Puede estimar modelos utilizando la entrada en bloque de las variables o cualquiera de los siguientes métodos por pasos: condicional hacia delante, LR hacia delante, Wald hacia delante, Condicional hacia atrás, LR hacia atrás o Wald hacia atrás.

Regresión logística: Consideraciones sobre los datos

Datos. La variable dependiente debe ser dicotómica. Las variables independientes pueden estar a nivel de intervalo o ser categóricas; si son categóricas, deben ser variables auxiliares o estar codificadas como indicadores (existe una opción en el procedimiento para codificar automáticamente las variables categóricas).

## Perceptrón multicapa

El clasificador de perceptrón multicapa (MLPC) es un clasificador basado en la red neuronal artificial de alimentación directa. MLPC consta de múltiples capas de nodos. Cada capa está completamente conectada a la siguiente capa en la red. Los nodos en la capa de entrada representan los datos de entrada. Todos los demás nodos asignan entradas a salidas mediante una combinación lineal de las entradas con los pesos w y sesgo del nodo y aplicando una función de activación. Esto se puede escribir en forma de matriz para MLPC con capas K + 1 de la siguiente manera:

$$\mathrm{y}(\mathbf{x}) = \mathrm{f}_{\mathrm{K}}(\ldots \mathrm{f}_2(\mathbf{w}_2^T \mathrm{f}_1(\mathbf{w}_1^T \mathbf{x} + b_1) + b_2) \ldots + b_K)$$

Los nodos en las capas intermedias utilizan la función sigmoidea (logística):

$$\mathrm{f}(z_i) = \frac{1}{1 + e^{-z_i}}$$

Los nodos en la capa de salida utilizan la función softmax:

$$\mathrm{f}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}$$

El número de nodos N en la capa de salida corresponde al número de clases.MLPC emplea la propagación hacia atrás para aprender el modelo. Utilizamos la función de pérdida logística para la optimización y L-BFGS como rutina de optimización.

## Implementación

Para su implementación se utilizó el lenguaje Scala  con la librerías de Spark en su versión 2.4.4 y se utilizaron los datos del dataset “bank-full” en formato “.csv”, los cuales pueden ser consultados en la siguiente dirección:
https://github.com/joseomarvite/BigData/tree/Unidad_4/Evaluacion

## Códigos

### Arbol de decision

```
//Importamos las librerias necesarias con las que vamos a trabajar
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.DateType
import org.apache.spark.sql.{SparkSession, SQLContext}
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.Transformer
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.ml.classification.DecisionTreeClassifier
import org.apache.spark.ml.feature.IndexToString
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.log4j._
//Quita los warnings
Logger.getLogger("org").setLevel(Level.ERROR)

//Creamos una sesion de spark y cargamos los datos del CSV en un datraframe
val spark = SparkSession.builder().getOrCreate()
val df =
spark.read.option("header","true").option("inferSchema","true").option("deli
miter",";").format("csv").load("bank-full.csv")
//Desblegamos los tipos de datos.
df.printSchema()
df.show(1)

//Cambiamos la columna y por una con datos binarios.
```

```
val change1 =
df.withColumn("y",when(col("y").equalTo("yes"),1).otherwise(col("y")))
val change2 =
change1.withColumn("y",when(col("y").equalTo("no"),2).otherwise(col("y")))
val newcolumn = change2.withColumn("y",'y.cast("Int"))
//Desplegamos la nueva columna
newcolumn.show(1)

//Generamos la tabla features
val assembler = new
VectorAssembler().setInputCols(Array("balance","day","duration","pdays","pre
vious")).setOutputCol("features")
val fea = assembler.transform(newcolumn)
//Mostramos la nueva columna
fea.show(1)

//Cambiamos la columna y a la columna label
val cambio = fea.withColumnRenamed("y", "label")
val feat = cambio.select("label","features")
feat.show(1)

//DecisionTree
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(feat)
// features con mas de 4 valores distinctivos son tomados como continuos
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setM
axCategories(4)
//Division de los datos entre 70% y 30% en un arreglo
val Array(trainingData, testData) = feat.randomSplit(Array(0.7, 0.3))
//Creamos un objeto DecisionTree
val dt = new
DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexed
Features")
//Rama de prediccion
val labelConverter = new
IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").set
Labels(labelIndexer.labels)
//Juntamos los datos en un pipeline
val pipeline = new Pipeline().setStages(Array(labelIndexer, featureIndexer,
dt, labelConverter))
//Create a model of the entraining
val model = pipeline.fit(trainingData)
```

```
//Transformacion de datos en el modelo
val predictions = model.transform(testData)
//Desplegamos predicciones
predictions.select("predictedLabel", "label", "features").show(5)
//Evaluamos la exactitud
val evaluator = new
MulticlassClassificationEvaluator().setLabelCol("indexedLabel").setPredictio
nCol("prediction").setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${(1.0 - accuracy)}")

val treeModel =
model.stages(2).asInstanceOf[DecisionTreeClassificationModel]
println(s"Learned classification tree model:\n ${treeModel.toDebugString}")
```

## Regresión Logística

```
//Importamos las librerias necesarias con las que vamos a trabajar
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.DateType
import org.apache.spark.sql.{SparkSession, SQLContext}
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.Transformer
import org.apache.spark.ml.classification.LinearSVC
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.log4j._
//Quita los warnings
Logger.getLogger("org").setLevel(Level.ERROR)

//Creamos una sesion de spark y cargamos los datos del CSV en un datraframe
val spark = SparkSession.builder().getOrCreate()
val df =
spark.read.option("header","true").option("inferSchema","true").option("deli
miter",";").format("csv").load("bank-full.csv")
//Desblegamos los tipos de datos.
df.printSchema()
df.show(1)

//Cambiamos la columna y por una con datos binarios.
```

```
val change1 =
df.withColumn("y",when(col("y").equalTo("yes"),1).otherwise(col("y")))
val change2 =
change1.withColumn("y",when(col("y").equalTo("no"),2).otherwise(col("y")))
val newcolumn = change2.withColumn("y",'y.cast("Int"))
//Desplegamos la nueva columna
newcolumn.show(1)

//Generamos la tabla features
val assembler = new
VectorAssembler().setInputCols(Array("balance","day","duration","pdays","pre
vious")).setOutputCol("features")
val fea = assembler.transform(newcolumn)
//Mostramos la nueva columna
fea.show(1)
//Cambiamos la columna y a la columna label
val cambio = fea.withColumnRenamed("y", "label")
val feat = cambio.select("label","features")
feat.show(1)

//Logistic Regresion
val logistic = new
LogisticRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
// Fit del modelo
val logisticModel = logistic.fit(feat)
//Impresion de los coegicientes y de la intercepcion
println(s"Coefficients: ${logisticModel.coefficients} Intercept:
${logisticModel.intercept}")
val logisticMult = new
LogisticRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
.setFamily("multinomial")
val logisticMultModel = logisticMult.fit(feat)
println(s"Multinomial coefficients: ${logisticMultModel.coefficientMatrix}")
println(s"Multinomial intercepts: ${logisticMultModel.interceptVector}")
```

## Perceptrón multicapa

```
//Importamos las librerias necesarias con las que vamos a trabajar
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.DateType
```

```
import org.apache.spark.sql.{SparkSession, SQLContext}
import org.apache.spark.ml.feature.VectorIndexer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.Transformer
import org.apache.spark.ml.classification.LinearSVC
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.classification.MultilayerPerceptronClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.log4j._
//Quita los warnings
Logger.getLogger("org").setLevel(Level.ERROR)

//Creamos una sesion de spark y cargamos los datos del CSV en un datraframe
val spark = SparkSession.builder().getOrCreate()
val df =
spark.read.option("header","true").option("inferSchema","true").option("deli
miter",";").format("csv").load("bank-full.csv")
//Desblegamos los tipos de datos.
df.printSchema()
df.show(1)

//Cambiamos la columna y por una con datos binarios.
val change1 =
df.withColumn("y",when(col("y").equalTo("yes"),1).otherwise(col("y")))
val change2 =
change1.withColumn("y",when(col("y").equalTo("no"),2).otherwise(col("y")))
val newcolumn = change2.withColumn("y",'y.cast("Int"))
//Desplegamos la nueva columna
newcolumn.show(1)

//Generamos la tabla features
val assembler = new
VectorAssembler().setInputCols(Array("balance","day","duration","pdays","pre
vious")).setOutputCol("features")
val fea = assembler.transform(newcolumn)
//Mostramos la nueva columna
fea.show(1)
//Cambiamos la columna y a la columna label
val cambio = fea.withColumnRenamed("y", "label")
val feat = cambio.select("label","features")
feat.show(1)

//Multilayer perceptron
```

```
//Dividimos los datos en un arreglo en partes de 70% y 30%
val split = feat.randomSplit(Array(0.6, 0.4), seed = 1234L)
val train = split(0)
val test = split(1)

// Especificamos las capas para la red neuronal
//De entrada 5 por el numero de datos de las features
//2 capas ocultas de dos neuronas
//La salida de 4  asi lo marca las clases
val layers = Array[Int](5, 2, 2, 4)

//Creamos el entrenador con sus parametros
val trainer = new
MultilayerPerceptronClassifier().setLayers(layers).setBlockSize(128).setSeed
(1234L).setMaxIter(100)
//Entrenamos el modelo
val model = trainer.fit(train)
//Imprimimos la exactitud
val result = model.transform(test)
val predictionAndLabels = result.select("prediction", "label")
val evaluator = new
MulticlassClassificationEvaluator().setMetricName("accuracy")
println(s"Test set accuracy = ${evaluator.evaluate(predictionAndLabels)}")
```

## Resultados

## Arbol de decision

```
scala> val accuracy = evaluator.evaluate(predictions)
accuracy: Double = 0.891364699006429

scala> println(s"Test Error = ${(1.0 - accuracy)}")
Test Error = 0.108635300993571

scala>

scala> val treeModel = model.stages(2).asInstanceOf[DecisionTreeClassificationModel]
treeModel: org.apache.spark.ml.classification.DecisionTreeClassificationModel = DecisionTreeClassificationModel (uid=dtc db9167c
f35b0) of depth 5 with 37 nodes
```

```
scala> println(s"Learned classification tree model:\n ${treeModel.toDebugString}")
Learned classification tree model:
 DecisionTreeClassificationModel (uid=dtc db9167cf35b0) of depth 5 with 37 nodes
  If (feature 2 <= 479.5)
   If (feature 3 <= 8.5)
    Predict: 0.0
   Else (feature 3 > 8.5)
    If (feature 2 <= 159.5)
     Predict: 0.0
    Else (feature 2 > 159.5)
     If (feature 3 <= 188.5)
      If (feature 3 <= 95.5)
       Predict: 1.0
      Else (feature 3 > 95.5)
       Predict: 0.0
     Else (feature 3 > 188.5)
      If (feature 3 <= 490.5)
       Predict: 0.0
      Else (feature 3 > 490.5)
       Predict: 1.0
  Else (feature 2 > 479.5)
   If (feature 2 <= 873.5)
    If (feature 3 <= 8.5)
     If (feature 2 <= 673.5)
      Predict: 0.0
     Else (feature 2 > 673.5)
      If (feature 1 <= 29.5)
       Predict: 0.0
      Else (feature 1 > 29.5)
       Predict: 1.0
```

```
       Predict: 1.0
     Else (feature 3 > 8.5)
      If (feature 3 <= 95.5)
       Predict: 1.0
      Else (feature 3 > 95.5)
       If (feature 1 <= 21.5)
        Predict: 0.0
       Else (feature 1 > 21.5)
        Predict: 1.0
    Else (feature 2 > 873.5)
     If (feature 0 <= -296.5)
      If (feature 1 <= 20.5)
       If (feature 1 <= 1.5)
        Predict: 1.0
       Else (feature 1 > 1.5)
        Predict: 0.0
      Else (feature 1 > 20.5)
       If (feature 1 <= 25.5)
        Predict: 1.0
       Else (feature 1 > 25.5)
        Predict: 0.0
     Else (feature 0 > -296.5)
      If (feature 0 <= 7380.0)
       Predict: 1.0
      Else (feature 0 > 7380.0)
       If (feature 3 <= 271.5)
        Predict: 0.0
       Else (feature 3 > 271.5)
        Predict: 1.0
```

## Regresión logística

```
scala> val logisticMultModel = logisticMult.fit(feat)
logisticMultModel: org.apache.spark.ml.classification.LogisticRegressionModel = LogisticRegressionModel: uid = logreg 6438759e85
21, numClasses = 3, numFeatures = 5

scala> println(s"Multinomial coefficients: ${logisticMultModel.coefficientMatrix}")
Multinomial coefficients: 3 x 5 CSCMatrix

scala> println(s"Multinomial intercepts: ${logisticMultModel.interceptVector}")
Multinomial intercepts: [-7.827431229384973,2.903059293515478,4.924371935869495]
```

## Perceptrón multicapa

```
scala> println(s"Test set accuracy = ${evaluator.evaluate(predictionAndLabels)}")
Test set accuracy = 0.8848956335944776
```

### Tabulación de resultados

| | Arbol de decision | Regresión logística | Perceptrón multicapa |
|---|---|---|---|
| Porcentaje de los datos a entrenar | 70% | 70% | 60% |
| Porcentaje de los datos a entrenar | 30% | 30% | 40% |
| Exactitud del algoritmo | 0.90 | 0.89 | 0.88 |
| Error del algoritmo | 0.10 | 0.11 | 0.12 |
| Capas neuronales | N/A | N/A | (5, 2, 2, 4) |

## Conclusión

Podemos concluir en base a los resultados obtenidos que al algoritmo de “Árboles de decisión” es más efectivo debido a la exactitud obtenida, también cabe destacar que los datos deben de tener sentido para que el algoritmo pueda funcionar correctamente.
Es importante recordar que los datos son tomados de la siguiente forma:

```
val Array(trainingData, testData) = feat.randomSplit(Array(0.7, 0.3))
```

“randomSplit” nos permite partir los datos de forma aleatoria, esto quiere decir que si compilamos el algoritmo más de una vez, existe la posibilidad que la exactitud aumente o disminuya ya que los datos fueron tomados de forma aleatoria, hay una gran posibilidad que en el 70% de los datos que se van a la variable “trainingData” todos tengan sentido al momentos de procesarlos, pero si en la variable “”testData” (30%) es todo lo contrario la exactitud disminuye de forma notable.

El resultado de “Árboles de decisión” o el despliegue del árbol está formulado en las condicionales “if and else” que nos permite graficar de forma sencilla mostrando la predicción en cada hoja terminal, esta sencillez hace que cualquier persona con nociones basicas de matematicas pueda entenderlo y explicarlo.

Los árboles de decisión son de gran herramienta para instituciones medicinales, bancarios, administrativos, y sobre todo para la toma de decisiones.

Por ejemplo, en Colombia estos algoritmos tienen gran relevancia en bancos, donde hacen un análisis previo para determinar el riesgo crediticio cuando una persona física o moral hace uso de un préstamo mostrando probabilidades (Naive Bayes), ventajas y desventajas.

Hernández, P. A. C. (2004). Aplicación de árboles de decisión en modelos de riesgo crediticio. Revista colombiana de estadística, 27(2), 139-151.

## Referencias

Hernández, P. A. C. (2004). Aplicación de árboles de decisión en modelos de riesgo crediticio. Revista colombiana de estadística, 27(2), 139-151.

Barandiaran, I. (1998). The random subspace method for constructing decision forests. IEEE Trans. Pattern Anal. Mach. Intell, 20(8), 1-22.

Magerman, D. M. (1995, June). Statistical decision-tree models for parsing. In Proceedings of the 33rd annual meeting on Association for Computational Linguistics (pp. 276-283). Association for Computational Linguistics.

Freund, Y., & Mason, L. (1999, June). The alternating decision tree learning algorithm. In icml (Vol. 99, pp. 124-133).

Yang, T., & Xie, J. Y. (2002, November). A multiuser detection method based on support vector machine. In Proceedings. International Conference on Machine Learning and Cybernetics (Vol. 1, pp. 373-375). IEEE.

Weber, B. G., & Mateas, M. (2009, September). A data mining approach to strategy prediction. In 2009 IEEE Symposium on Computational Intelligence and Games (pp. 140-147). IEEE.

Bhargava, N., Sharma, G., Bhargava, R., & Mathuria, M. (2013). Decision tree analysis on j48 algorithm for data mining. Proceedings of International Journal of Advanced Research in Computer Science and Software Engineering, 3(6).