Tecnológico Nacional de México

Instituto Tecnológico de Tijuana

Subdirección Académica

Departamento de Sistemas y Computación

SEMESTRE SEPTIEMBRE – ENERO 2020-2021

Ing. en Sistemas Computacionales

Materia: Datos Masivos

Serie: BDD-1704 TI9A

Profesor: JOSE CHRISTIAN ROMERO HERNANDEZ

Unidad#4:
Proyecto Final

Ceballos Bobadilla Aide	15211282
Rodriguez Medrano Marco Antonio	17210635

Fecha de Entrega 16 de Enero del 2021

Índice

1-Introducción

En el siguiente documento se presenta el reporte de el trabajo realizado como Proyecto Final de la materia de Datos Masivos , el cual consta de la investigación e implementación de diferentes tipos de algoritmos de Machine Learning (SVM,Decision Three,Logistic Regresion, Multilayer perceptron) de Spark en su versión 2.4.7 y someternos a una comparación utilizando a una misma base de datos dados por el docente y esto con el objetivo de hacer una comparación del rendimiento de los algoritmos y analizar cada uno de ellos.

2.-Marco Teórico

¿Qué es SVM?

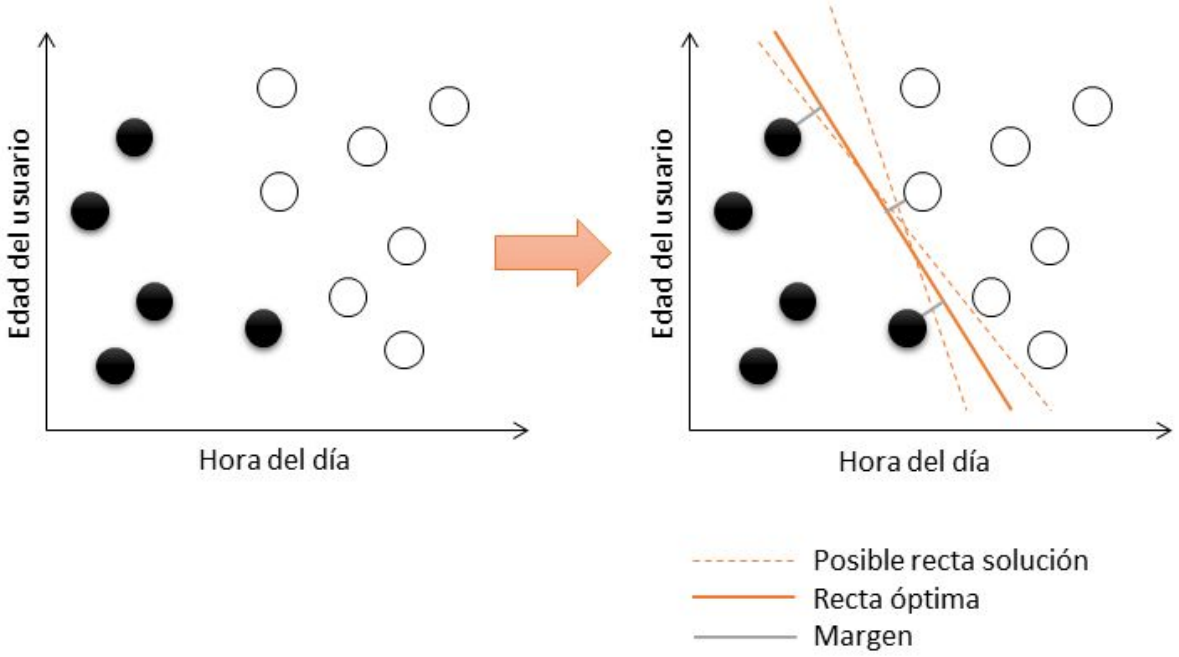
Las **Máquinas de Vectores Soporte** (creadas por Vladimir Vapnik) constituyen un método basado en aprendizaje para la resolución de problemas de clasificación y regresión. En ambos casos, esta resolución se basa en una primera fase de *entrenamiento* (donde se les informa con múltiples ejemplos ya resueltos, en forma de pares {problema, solución}) y una segunda fase de uso para la resolución de problemas. En ella, las SVM se convierten en una "caja negra" que proporciona una respuesta (salida) a un problema dado (entrada).

¿Para qué podemos usar las SVM?

Los conceptos fundamentales son modelado y predicción en dos vertientes: clasificación y regresión.

Clasificación

Imaginemos que buscamos encontrar qué tipo de usuario tiene más probabilidad de hacer clic en un determinado *banner*. Está claro que esta decisión implica varias variables a tener en cuenta: no solo de las características del propio usuario, sino también podremos considerar su región geográfica, la tecnología empleada, día/hora en que se encuentra con el *banner*, etc. Si solo dos de estas variables fueran determinantes, podríamos encontrarnos en una situación similar a la de la imagen, donde el círculo negro identifica que el usuario hace clic, y el blanco que no. Con SVM podemos obtener la "superficie óptima" que delimitará el comportamiento *clic* – *noclic* de un usuario:



¿Qué es Decision Tree?

Árbol de decisión o Decision Tree Classification es un tipo de algoritmo de aprendizaje supervisado que se utiliza principalmente en problemas de clasificación, aunque funciona para variables de entrada y salida categóricas como continuas.

En esta técnica, dividimos la data en dos o más conjuntos homogéneos basados en el diferenciador más significativos en las variables de entrada. El árbol de decisión identifica la variable más significativa y su valor que proporciona los mejores conjuntos homogéneos de población. Todas las variables de entrada y todos los puntos de división posibles se evalúan y se elige la que tenga mejor resultado.

Los algoritmos de aprendizaje basados en árbol se consideran uno de los mejores y más utilizados métodos de aprendizaje supervisado. Los métodos basados en árboles potencian modelos predictivos con alta precisión, estabilidad y facilidad de interpretación. A diferencia de los modelos lineales, mapean bastante bien las relaciones no lineales.

¿Qué es la regresión logística?

La regresión logística es una técnica de aprendizaje automático que proviene del campo de la estadística. A pesar de su nombre no es un algoritmo para aplicar en problemas de regresión, en los que se busca un valor continuo, sino que es un método para problemas de clasificación, en los que se obtienen un valor binario entre 0 y 1. Por ejemplo, un problema de clasificación es identificar si una operación dada es fraudulenta o no. Asociándolo una etiqueta “fraude” a unos registros y “no fraude” a otros. Simplificando mucho es identificar si al realizar una afirmación sobre registro esta es cierta o no.

Con la regresión logística se mide la relación entre la variable dependiente, la afirmación que se desea predecir, con una o más variables independientes, el conjunto de características disponibles para el modelo. Para ello utiliza una función logística que determina la probabilidad de la variable dependiente. Como se ha comentado anteriormente, lo que se busca en estos problemas es una clasificación, por lo que la probabilidad se ha de traducir en valores binarios. Para lo que se utiliza un valor umbral. Los valores de probabilidad por encima del valor umbral la afirmación es cierta y por debajo es falsa. Generalmente este valor es 0,5, aunque se puede aumentar o reducir para gestionar el número de falsos positivos o falsos negativos.

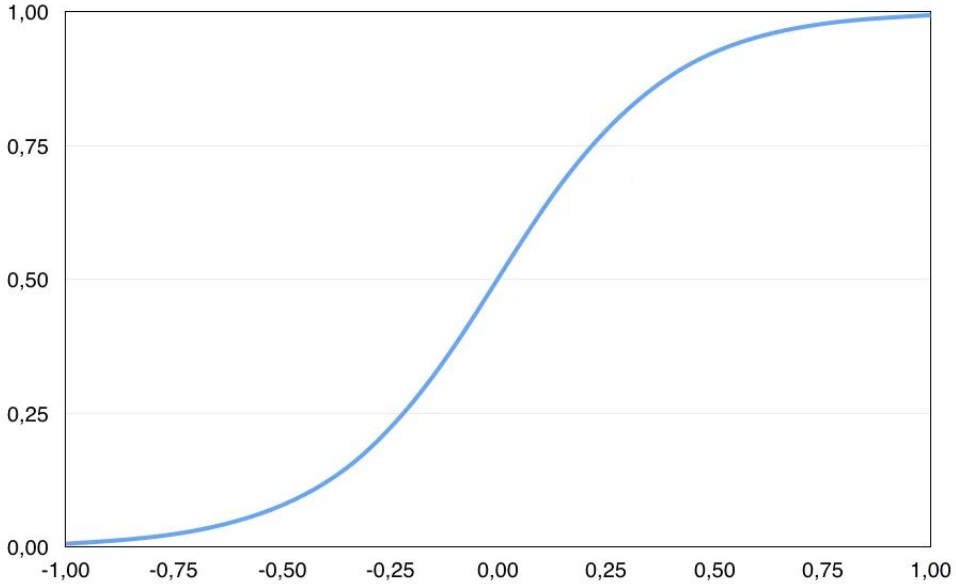
Formulación matemática de la función logística

A la función que relaciona la variable dependiente con las independientes también se le llama función sigmoidea. La función sigmoidea es una curva en forma de S que puede tomar cualquier valor entre 0 y 1, pero nunca valores fuera de estos límites. La ecuación que define la función sigmoidea es

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

donde x es un número real. En la ecuación se puede ver que cuando x tiene a menos infinito el cociente tiende a cero. Por otro lado, cuando x tiende a infinito el cociente tiende a la unidad.

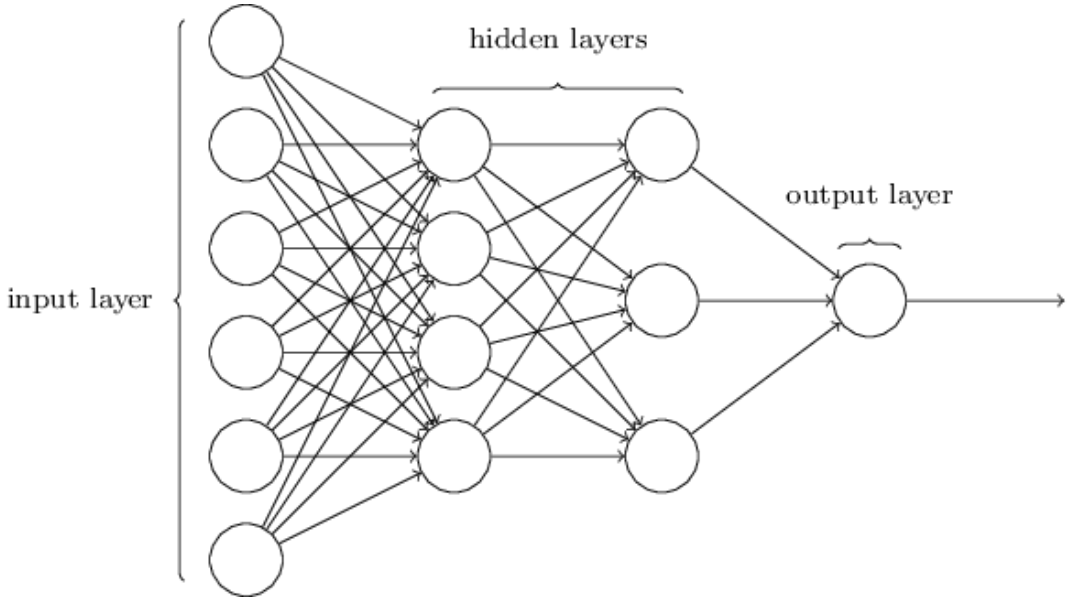
En esta imagen se puede observar la representación gráfica de la función logística (función sigmoide).



¿Qué es el multilayer perceptron?

El clasificador de perceptrones multicapa (MLPC) es un clasificador basado en la red neuronal artificial feedforward . MLPC consta de múltiples capas de nodos. Cada capa está completamente conectada a la siguiente capa de la red. Los nodos de la capa de entrada representan los datos de entrada. Todos los demás nodos mapean

entradas a las salidas realizando una combinación lineal de las entradas con los pesos y el sesgo del nodo y aplicando una función de activación.



Como se describe en la Imagen, MLPC consta de múltiples capas de nodos, incluida la capa de entrada, las capas ocultas (también llamadas capas intermedias) y las capas de salida. Cada capa está completamente conectada a la siguiente capa de la red. Donde la capa de entrada, las capas intermedias y la capa de salida se pueden definir de la siguiente manera:

La capa de entrada consta de neuronas que aceptan los valores de entrada. La salida de estas neuronas es la misma que la de los predictores de entrada. Los nodos de 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 del nodo y el sesgo y aplicando una función de activación. Esto se puede escribir en forma de matriz para MLPC con capas K + 1 de la siguiente manera: Input_Layer

$$\mathbf{y}(\mathbf{x}) = \mathrm{f_K}(\ldots \mathrm{f_2}(\mathbf{w}_2^T \mathrm{f_1}(\mathbf{w}_1^T \mathbf{x} + b_1) + b_2) \ldots + b_K)$$

Las capas ocultas se encuentran entre las capas de entrada y salida. Normalmente, el número de capas ocultas varía de una a muchas. Es la capa de cálculo central que tiene las funciones que asignan la entrada a la salida de un nodo. Los nodos de las capas intermedias utilizan la función sigmoidea (logística), de la siguiente manera Hidden_Layer

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

La capa de salida es la capa final de una red neuronal que devuelve el resultado al entorno del usuario. Basado en el diseño de una red neuronal, también indica a las capas anteriores cómo se han desempeñado en el aprendizaje de la información y, en consecuencia, mejoraron sus funciones. Los nodos de la capa de salida utilizan la función softmax. Output_Layer

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

El número de nodos N, en la capa de salida, corresponde al número de clases.

3.-Implementación

Para la elaboración de este proyecto se utilizó Apache Spark el cual es un framework de programación para procesamiento de datos distribuidos diseñado para ser rápido y de propósito general. Como su propio nombre indica, ha sido desarrollada en el marco del proyecto Apache, lo que garantiza su licencia Open Source.

Además, podremos contar con que su mantenimiento y evolución se llevarán a cabo por grupos de trabajo de gran prestigio, y existirá una gran flexibilidad e interconexión con otros módulos de Apache como Hadoop, Hive o Kafka.
Parte de la esencia de Spark es su carácter generalista. Consta de diferentes APIs y módulos que permiten que sea utilizado por una gran variedad de profesionales en todas las etapas del ciclo de vida del dato.
Dichas etapas pueden incluir desde soporte para análisis interactivo de datos con SQL a la creación de complejos pipelines de machine learning y procesamiento en streaming, todo usando el mismo motor de procesamiento y las mismas APIs.

Apache Spark es un motor de procesamiento distribuido responsable de orquestar, distribuir y monitorear aplicaciones que constan de múltiples tareas de procesamiento de datos sobre varias máquinas de trabajo, que forman un cluster.

Para utilizar Apache se utilizó Scala el cual es un lenguaje de programación moderno multi-paradigma diseñado para expresar patrones de programación comunes de una forma concisa, elegante, y con tipado seguro. Integra fácilmente características de lenguajes orientados a objetos y funcionales.
Scala es también un lenguaje funcional en el sentido que toda función es un valor. Scala provee una sintaxis ligera para definir funciones anónimas. Soporta funciones de orden superior, permite funciones anidadas, y soporta currying. Las clases Case de Scala y las construcciones incorporadas al lenguaje para reconocimiento de patrones modelan tipos algebraicos usados en muchos lenguajes de programación funcionales.

Linear Support Vector Machine

```
//Importamos la libreria de SVC y las demas que utilizaremos
import org.apache.spark.ml.classification.LinearSVC
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer, VectorIndexer, OneHotEncoder}
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.mllib.evaluation.MulticlassMetrics

//Eliminamos los errores
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

//Creamos las variables runtime y startTimeMillis para obtener el tiempo de ejecución
val runtime = Runtime.getRuntime
val startTimeMillis = System.currentTimeMillis()

// creamos un dataframe donde le cargaremos los datos del dataset bank-full.csv
val training  = spark.read.option("header","true").option("inferSchema",
"true").option("delimiter",";").format("csv").load("C:/Users/DELL/Desktop/logicRegresion/bank-full.csv")
val labelIndexer = new StringIndexer().setInputCol("y").setOutputCol("indexedLabel").fit(training)
val indexed = labelIndexer.transform(training).drop("y").withColumnRenamed("indexedLabel", "label")

//creamos la variable vectorFeatures con la columna features
val vectorFeatures = (new
VectorAssembler().setInputCols(Array("balance","day","duration","pdays","previous")).setOutputCol("f
eatures"))

//hacemos la transformacion de los resultados
val features = vectorFeatures.transform(indexed)
//renombramos la columna y label
val featuresLabel = features.withColumnRenamed("y", "label")
//seleccionamos las columnas label y featues y las agregamos a la nueva variable dataIndexed
val dataIndexed = featuresLabel.select("label","features")

//creamos las variables labelIndexer y featureIndexer donde agregaremos las columnas label y
features
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(dataIndexed)
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(4).fit(data
Indexed)

//crearemos el arreglo de entrenamiento y testeo con 70% de entrenamiento y 30% de testeo
val Array(training, test) = dataIndexed.randomSplit(Array(0.7, 0.3), seed = 12345)

//estas tres variables se utilizaran para hacer el entrenamiento y testeo
val lsvc = new LinearSVC().setMaxIter(10).setRegParam(0.1)

val lsvcModel = lsvc.fit(training)
```

```
val results = lsvcModel.transform(test)
//creamos la predictionAndLabels para hacer la prediccion
val predictionAndLabels = results.select($"prediction",$"label").as[(Double, Double)].rdd
val metrics = new MulticlassMetrics(predictionAndLabels)

//imprimimos la matrix de confusion
println("Confusion matrix:")
println(metrics.confusionMatrix)

//imprimimos la exactitud
println("Accurancy: " + metrics.accuracy)
println(s"Test Error = ${(1.0 - metrics.accuracy)}")

//obtenemos las diferentes valores de memoria utilizados
val mb = 0.000001
println("Used Memory: " + (runtime.totalMemory - runtime.freeMemory) * mb)
println("Free Memory: " + runtime.freeMemory * mb)
println("Total Memory: " + runtime.totalMemory * mb)
println("Max Memory: " + runtime.maxMemory * mb)

//obtenemos el tiempo de ejecucion
val endTimeMillis = System.currentTimeMillis()
val durationSeconds = (endTimeMillis - startTimeMillis) / 1000
//imprimimos los coeficientes
println(s"Coefficients: ${lsvcModel.coefficients} Intercept: ${lsvcModel.intercept}")
```

Decision tree classifier

```
//Importamos las librerias
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}

val runtime = Runtime.getRuntime
val startTimeMillis = System.currentTimeMillis()

// Iniciamos una sesion de spark
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder.appName("DecisionTreeClassificationExample").getOrCreate()

//cargamos la base de datos y ordenamos
val data = spark.read.option("header","true").option("inferSchema",
"true").option("delimiter",";").format("csv").load("C:/Users/DELL/Desktop/logicRegresion/bank-full.csv")
val labelIndexer = new StringIndexer().setInputCol("y").setOutputCol("indexedLabel").fit(df)
val indexed = labelIndexer.transform(df).drop("y").withColumnRenamed("indexedLabel", "label")

//creamos la variable vectorFeatures y le agregamos el arreglo
val vectorFeatures = (new
VectorAssembler().setInputCols(Array("balance","day","duration","pdays","previous")).setOutputCol("f
eatures"))
//creamos la varaible features donde se guardara la transformacion de los resultados
val features = vectorFeatures.transform(indexed)
```

```
//se renombramos las columnas Y y label
val featuresLabel = features.withColumnRenamed("y", "label")

//selecionamos las columnas label y features
val dataIndexed = featuresLabel.select("label","features")

// se crean las variables labelIndexer y featureIndexer donde les agregaremos los valores de label y features
val labelIndexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(dataIndexed)
val featureIndexer = new VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(4).fit(dataIndexed)

//entrenamos el modelo con 70% de entrenamiento y 30% de testeo
val Array(trainingData, testData) = dataIndexed.randomSplit(Array(0.7, 0.3))

//creamos un dataframe donde guardaremos las columnas indexedLabel y indexedFeatures
val dt = new DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures")

//convertimos la columna label para hacer la prediccion
val labelConverter = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(labelIndexer.labels)

val pipeline = new Pipeline().setStages(Array(labelIndexer, featureIndexer, dt, labelConverter))

//creamos el modelo
val model = pipeline.fit(trainingData)

//creamos la prediccion
val predictions = model.transform(testData)

//vemos los primeros 5 columnas
predictions.select("predictedLabel", "label", "features").show(5)

//creamos la variable evaluator la cual nos servirá para obtener la exactitud
val evaluator = new MulticlassClassificationEvaluator().setLabelCol("indexedLabel").setPredictionCol("prediction").setMetricName("accuracy")

//creamos la variable exactitud
val accuracy = evaluator.evaluate(predictions)
//imprimimos el error de testeo
println(s"Test Error = ${(1.0 - accuracy)}")

val treeModel = model.stages(2).asInstanceOf[DecisionTreeClassificationModel]

println(s"Learned classification tree model:\n ${treeModel.toDebugString}")
//imprimimos los valores de la memoria utilizada
val mb = 0.000001
println("Used Memory: " + (runtime.totalMemory - runtime.freeMemory) * mb)
println("Free Memory: " + runtime.freeMemory * mb)
```

```
println("Total Memory: " + runtime.totalMemory * mb)
println("Max Memory: " + runtime.maxMemory * mb)

//obtenemos el tiempo de ejecucion
val endTimeMillis = System.currentTimeMillis()
val durationSeconds = (endTimeMillis - startTimeMillis) / 1000
```

Logic Regression

```
//Importamos las librerias necesarias con las que vamos a trabajar
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer, VectorIndexer, OneHotEncoder}
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.sql.SparkSession

//eliminamos los errores
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

//creamos las variables runtime y startTimeMillis para obtener el tiempo de ejecución
val runtime = Runtime.getRuntime
val startTimeMillis = System.currentTimeMillis()

//creamos la sesison de spark
val spark = SparkSession.builder().getOrCreate()

//creamoa la variable data y cargamos el dataset bank-full.csv y limpiamos
val data = spark.read.option("header","true").option("inferSchema",
"true").option("delimiter",";").format("csv").load("C:/Users/DELL/Desktop/logicRegresion/bank-full.csv")
val labelIndexer = new StringIndexer().setInputCol("y").setOutputCol("indexedLabel").fit(data)
val indexed = labelIndexer.transform(data).drop("y").withColumnRenamed("indexedLabel", "label")

//creamos la variable vectorFeatures dondel crearemos un arreglo con las columnas balance, day,
duration, pdays, previous y la columna features
val vectorFeatures = (new
VectorAssembler().setInputCols(Array("balance","day","duration","pdays","previous")).setOutputCol("f
eatures"))

//creamos la variable features donde transformaremos los valores obtenidos en la variable
vectorFeatures
val features = vectorFeatures.transform(indexed)

//creamos la variable featuresLabel donde renoblaremos la columna y por label
val featuresLabel = features.withColumnRenamed("y", "label")

//creamos la variable dataIndexed, seleccionaremos los datos de label y label y las guardaremos en
dataIndexed
val dataIndexed = featuresLabel.select("label","label")

//arreglo donde tomaremos un 70% training y 30% para el testeo
val Array(training, test) = dataIndexed.randomSplit(Array(0.7, 0.3), seed = 12345)
//creamos la variable logisticAlgorithm donde haremos 10 iteracciones
```

```
val logisticAlgorithm = new
LogisticRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8).setFamily("multinomi
al")

//creamos el modelo y lo entrenamos
val logisticModel = logisticAlgorithm.fit(training)

//vemos los resultados  de transformar el modelo ya entrenado con el testeo
val results = logisticModel.transform(test)

/*importamos la libreria MulticlassMetrics, nota profe tuve que investigar porque solo poniendo
aqui la importación de esta librería me arrojaba los resultados y eso me extraño*/
import org.apache.spark.mllib.evaluation.MulticlassMetrics

//hacemos la predicción
val predictionAndLabels = results.select($"prediction",$"label").as[(Double, Double)].rdd
val metrics = new MulticlassMetrics(predictionAndLabels)

//imprimimos la matriz de confusión
println("Confusion matrix:")
println(metrics.confusionMatrix)

//imprimimos la exactitud y el error de testeo
println("Accuracy: "+metrics.accuracy)
println(s"Test Error = ${(1.0 - metrics.accuracy)}")

//creamos la variable mb que nos ayuda a obtener diferentes valores de memoria
val mb = 0.000001
println("Used Memory: " + (runtime.totalMemory - runtime.freeMemory) * mb)
println("Free Memory: " + runtime.freeMemory * mb)
println("Total Memory: " + runtime.totalMemory * mb)
println("Max Memory: " + runtime.maxMemory * mb)

// variables que arrojan el tiempo de ejecución del código
val endTimeMillis = System.currentTimeMillis()
val durationSeconds = (endTimeMillis - startTimeMillis) / 1000
```

Multilayer Perceptron

```
//importamos las librerias que vamos a utilizar
import org.apache.spark.sql.types.IntegerType
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.classification.MultilayerPerceptronClassifier
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.ml.feature.VectorAssembler

//eliminamos cualquier error
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

/*creamos la variable runtime y starttimemillis para ser capaces de
obtener el tiempo en el que el codigo obtiene los resultados*/
val runtime = Runtime.getRuntime
```

```
val startTimeMillis = System.currentTimeMillis()

//creamos la sesion de spark
val spark = SparkSession.builder.appName("MultilayerPerceptronClassifierExample").getOrCreate()

// creamos un dataframe donde le cargaremos los datos del dataset bank-full.csv
val df  = spark.read.option("header","true").option("inferSchema",
"true").option("delimiter",";").format("csv").load("C:/Users/DELL/Desktop/MultiplayerPerceptron/bank-full.csv")

//operaciones básicas con el dataframe
//ver las columnas
df.columns
//ver el schema
df.printSchema()
//ver los primeros 5 columnas
df.head(5)
//ver la descripción
df.describe().show()

//creamos las variables labelindexer y indexed para hacer limpieza de datos
val labelIndexer = new StringIndexer().setInputCol("y").setOutputCol("indexedLabel").fit(df)
val indexed = labelIndexer.transform(df).drop("y").withColumnRenamed("indexedLabel", "label")
indexed.describe().show()
/*creamos las variables assembler y features para transformar, agregar una columna llamada
features
y para acomodar los datos ya una vez limpiados en el paso anterior*/
val assembler = new
VectorAssembler().setInputCols(Array("balance","day","duration","pdays","previous")).setOutputCol("features")
val  features = assembler.transform(indexed)

//creamos la variable labelindexer para encontrar los valores los label y agregar una columna llamada
label con esos datos
val labelIndexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(indexed)
println(s"Found labels: ${labelIndexer.labels.mkString("[", ", ", "]")}")
features.show()

// creamos la variables de testeo train y splits
 val splits = features.randomSplit(Array(0.6, 0.4), seed = 1234L)
 val train = splits(0)
 val test = splits(1)

 //creamos la variable layers que llevará los valores de los nodos, nodo de entrada 5, dos nodos
internos 4, 1 y un nodo de salida 2
val layers = Array[Int](5, 4, 1, 2)

//hacemos el entrenamiento con 100 iteraciones
val trainer = new
MultilayerPerceptronClassifier().setLayers(layers).setBlockSize(128).setSeed(1234L).setMaxIter(100)

//creamos el modelo
val model = trainer.fit(train)
//transformamos los resultados guardados en el modelo
```

```
val result = model.transform(test)
//creamos la variable predictionAndLabels para ver los resultados en una nueva tabla con la columnas prediction y label
val predictionAndLabels = result.select("prediction", "label")
predictionAndLabels.show

/*creamos la variable evuador donde utilizando la libreria de multiclass obtendremos
la exactitud de prueba */
val evaluator = new MulticlassClassificationEvaluator().setMetricName("accuracy")
println(s"Test set accuracy = ${evaluator.evaluate(predictionAndLabels)}")

//creamos la variable mb la cual utilizaremos para obtener el Used Memory, Free Memory, Total Memory y Max Memory
val mb = 0.000001
println("Used Memory: " + (runtime.totalMemory - runtime.freeMemory) * mb)
println("Free Memory: " + runtime.freeMemory * mb)
println("Total Memory: " + runtime.totalMemory * mb)
println("Max Memory: " + runtime.maxMemory * mb)

//variables donde veremos el tiempo que tardo el código en ejecutarse
val endTimeMillis = System.currentTimeMillis()
val durationSeconds = (endTimeMillis - startTimeMillis) / 1000

spark.stop()
```

4.-Resultados

Performance

	Memoria Usada MB	Segundos
Logic Regression	198.91	145
SVM	441.48	111
Decision Three	504.79	47
Multilayer perceptron	227.36	219

Accuracy

	Accuracy	Error
Logic Regression	0.884	0.115
SVM	0.885	0.114
Decision Three	0.895	0.104
Multilayer perceptron	0.882	0.117

5.-Conclusión

Marco Antonio Rodriguez Medrano

En conclusión los cuatro métodos que se investigaron, se creó un código ejemplo y posteriormente se observan los resultados son excelentes métodos para obtener un cierto valor, pero al final el usuario buscará cuál método es más eficiente, (claro dependiendo del problema que quiera resolver), pero al compararlos podemos observar que en cuestión de velocidad Decision Three fue el que obtuvo el resultado más rápido, pero a muchos la velocidad no implica precisión o exactitud, pero en este caso nuevamente Decision Three tuvo la exactitud más alta con un error estimado del 0.10% lo cual es excelente ya que eso nos indica que tiene menos probabilidad de fallar, pero siendo de los cuatro el que más memoria utilizó.

Aide Ceballos Bobadilla

Como conclusión considero que esta práctica nos permitió hacer una buena comparación entre diferentes métodos, esto nos permite comparar el comportamiento de ellos y ver cuales son sus diferencias, considero que todos los métodos son buenos, y el decidir cual utilizar es dependiendo de el problema que estemos tratando de resolver , sería cuestión de analizar cuál nos conviene más o inclusive con cual nos sentimos más confiados de sus resultados.

6.-Referencias(No wikipedia)

CinthiaBV. (27 de noviembre de 2020). *Github*. Obtenido de https://github.com/CinthiaBV/EXPODM

Rodríguez, D. (23 de julio de 2018). *analyticslane*. Obtenido de https://www.analyticslane.com/2018/07/23/la-regresion-logistica/

ESIC Business & Marketing School.(Octubre 2018). Esic de https://www.esic.edu/rethink/tecnologia/apache-spark-introduccion-que-es-y-como-funciona

Alvar Arnaiz Gonzales.(N.A). Scala. Obtenido de https://docs.scala-lang.org/es/tour/tour-of-scala.html

Anónimo (1 Septiembre 2020).Merkle . Obtenido de https://www.merkleinc.com/es/es/blog/machine-learning-support-vector-machines