EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

SEMESTRE ENERO - JUNIO 2020

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES

DATOS MASIVOS

BDD-1704 TI9A

PROYECTO FINAL

UNIDAD 4

PARDINI CORONADO SAMANTHA 15210339

REYNOSO SILVA ANGÉLICA 15211080

JOSE CHRISTIAN ROMERO HERNANDEZ

18 / JUNIO / 2020

ÍNDICE

Comparativo del Rendimiento de los Algoritmos de Machine Learning: SVM, Decision Tree, Logistic Regression y Multilayer Perceptron en Spark Scala

Pardini Coronado Samantha[1], Reynoso Silva Angélica[2]

Instituto Tecnológico de Tijuana
Tijuana, México
samantha.pardini16@tectijuana.edu.mx[1], angelica.reynoso@tectijuana.edu.mx[2]

Resumen.
Hoy en día nos encontramos rodeados de la generación de grandes cantidades de datos, las cuales nosotros como humanos no somos capaces de procesar en su totalidad; es por esta razón que nos vemos en la necesidad de buscar herramientas que nos ayuden a procesar, extraer y/o almacenar información, brindándonos la capacidad de obtener el beneficio necesario para alcanzar un objetivo concreto.
Estos objetivos se pueden conseguir mediante un enfoque en "Machine Learning" o Aprendizaje Automático, el cual permite a las máquinas aprender y tomar decisiones con datos proporcionados de manera automática. Este aprendizaje se lleva a cabo gracias a la detección de patrones dentro del conjunto de datos, de forma que el mismo algoritmo sea capaz de predecir cualquier tipo de situaciones que pudieran presentarse.
Entre los algoritmos de este tipo se encuentran el Support Vector Machine o SVM, el Árbol de Decisión o Decision Tree, la Regresión Logística o Logistic Regression, y el Multilayer Perceptron, por mencionar algunos.

Palabras Clave: Rendimiento, Algoritmos, Machine Learning, SVM, Support Vector Machine, Decision Tree, Árbol de Decisión, Logistic Regression, Regresión Logística, Multilayer Perceptron, Spark, Scala.

1. INTRODUCCIÓN

El término Machine Learning se refiere a la detección automatizada de patrones significativos en los datos. En las últimas dos décadas se ha convertido en una herramienta común en casi cualquier tarea que requiera la extracción de información de grandes conjuntos de datos. [6]

El presente artículo se encuentra dividido en cinco secciones, más los anexos. En la segunda sección se encuentra el Marco Teórico en donde se exponen las definiciones y conceptos correspondientes a los algoritmos empleados para el análisis comparativo. En la tercera sección se mencionan las herramientas utilizadas para la elaboración del análisis. La cuarta sección describe los resultados obtenidos de las pruebas elaboradas con cada tipo de algoritmo, y finalmente en la quinta sección se presentan las conclusiones obtenidas para todo el trabajo.

2. MARCO TEÓRICO

2.1 SVM

Una máquina de vectores de soporte (SVM) es un algoritmo de aprendizaje supervisado que se puede emplear para clasificación binaria o regresión. Las máquinas de vectores de soporte son muy populares en aplicaciones

como el procesamiento del lenguaje natural, el habla, el reconocimiento de imágenes y la visión artificial. [2]

2.1.1 Ventajas

- Eficaz en espacios de grandes dimensiones.
- Todavía eficaz en casos donde el número de dimensiones es mayor que el número de muestras.
- Versátil ya que se pueden especificar diferentes funciones del núcleo para la función de decisión. Se proporcionan kernels comunes, pero también es posible especificar kernels personalizados. [5]

2.1.2 Desventajas

- Si el número de características es mucho mayor que el número de muestras, evite el exceso de ajuste al elegir las funciones del Kernel y el término de regularización es crucial.
- Los SVMs no proporcionan directamente estimaciones de probabilidad, éstas se calculan utilizando una validación cruzada quíntuple. [5]

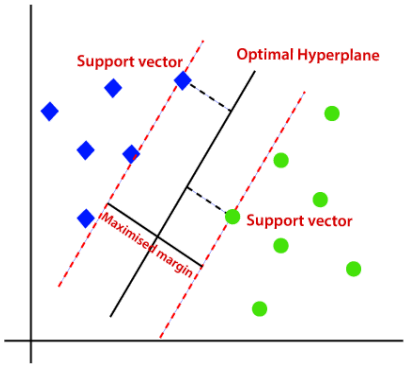

Fig. 1.- Representación de SVM.

2.2 Decision Tree

Los árboles de decisión son una metodología de clasificación, en la que el proceso de clasificación se modela con el uso de un conjunto de decisiones jerárquicas sobre las variables de características, organizadas en una estructura similar a un árbol. La decisión en un nodo particular del árbol, que se conoce como el criterio de división, es típicamente una condición en una o más variables de características en los datos de entrenamiento. El criterio de división divide los datos de entrenamiento en dos o más partes.

El objetivo es identificar un criterio dividido para que el nivel de "mezcla" de las variables de clase en cada rama del árbol se reduzca tanto como sea posible. Cada nodo en el árbol de decisión representa lógicamente un subconjunto del espacio de datos definido por la combinación de criterios divididos en los nodos por encima de él. El árbol de decisión se construye típicamente como una partición jerárquica de los ejemplos de entrenamiento, de la misma manera que un algoritmo de agrupamiento en la parte superior divide los datos jerárquicamente. [4]

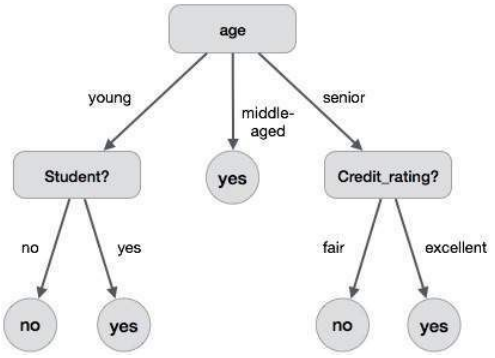

Fig. 2.- Representación de Decision Tree.

2.3 Logistic Regression

A pesar de su nombre, Regresión Logística es un modelo para clasificación, no para regresión. Este algoritmo es muy simple de implementar pero sin embargo es difícil que converja si los datos que estemos tratando no son separables linealmente. Es un clasificador lineal binario pero que usa la técnica OvR (One versus Rest).

En estadística, Regresión Logística es un modelo de regresión donde la variable dependiente es categórica, aquella que puede coger valores fijos o un número de valores posibles. Estas variables dependientes son aquellas clases objetivos que queremos predecir mientras que las variables independientes son las distintas características que componen nuestro conjunto de datos y que vamos a usar para entrenar nuestro modelo.

Este algoritmo es muy usado en muchos campos, incluyendo entre ellos el aprendizaje automático, la mayoría de ellos usados a nivel médico, como por ejemplo para averiguar la gravedad de un paciente se ha llegado a usar este tipo de algoritmo o por ejemplo para predicción de que se produzca una tormenta geomagnética usando modelos de este tipo. Su uso también se puede ver reflejado en el campo de la ingeniería, especialmente para saber la probabilidad de fallo de un proceso, un producto o un sistema. [6]

2.3.1 Ventajas

- Fácil de entender y explicar.
- Rara vez existe sobreajuste.
- El uso de la regularización es efectivo en la selección de funciones.
- Rápido para entrenar.
- Fácil de entrenar sobre grandes datos gracias a su versión estocástica. [5]

2.3.2 Desventajas

- Tienes que trabajar duro para que se ajuste a los datos no lineales.
- Puede sufrir con valores atípicos.
- En algunas ocasiones es muy simple para captar relaciones complejas entre variables. [5]

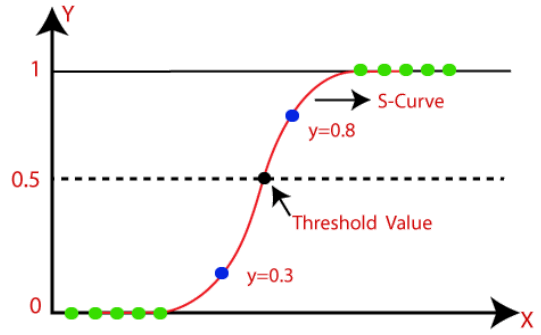

Fig. 3.- Representación de Logistic Regression.

2.4 Multilayer Perceptron

Un perceptrón multicapa (MLP) es una red neuronal artificial de alimentación directa que genera un conjunto de salidas a partir de un conjunto de entradas. Un MLP se caracteriza por varias capas de nodos de entrada conectados como un gráfico dirigido entre las capas de entrada y salida. MLP utiliza la retropropagación para capacitar a la red. MLP es un método de aprendizaje profundo. [1,9]

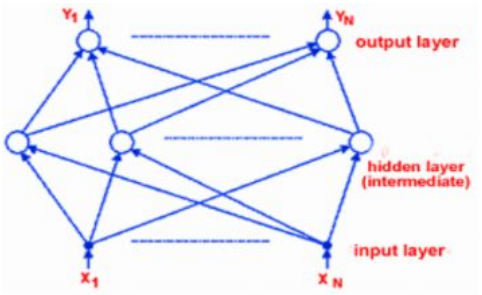

Fig. 4.- Multicapa de alimentación de red neuronal.

2.4.1 Características

- Este tipo de redes son muy potentes y pueden ser extremadamente complicadas.
- Son dinámicos, cambian su condición todo el tiempo, hasta que la red alcanza un estado de equilibrio, y la búsqueda de un nuevo equilibrio ocurre con cada cambio de entrada.
- La introducción de varias capas se determinó por la necesidad de aumentar la complejidad de regiones de decisión. [9]

2.4.2 Ventajas

- Es capaz de crear modelos no lineales.
- Puede entrenar un modelo en tiempo real haciendo uso del método partial_fit, que actualiza el modelo con una única iteración con los datos que se incluyan como argumentos. [9]

2.4.3 Desventajas

- Posee distintos problemas y limitaciones como por ejemplo el proceso de aprendizaje para problemas complejos con gran cantidad de variables.
- La existencia de mínimos locales en la función de error dificulta considerablemente el entrenamiento, pues una vez alcanzado un mínimo el entrenamiento se detiene aunque no se haya alcanzado la tasa de convergencia fijada. [9]

3. IMPLEMENTACIÓN

Para el desarrollo del proyecto, se utilizaron las siguientes herramientas:

3.1 Spark

Apache Spark 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. [3]

3.2 Scala

Scala es un lenguaje de programación de propósito general, diseñado para programar utilizando patrones de una forma concisa, elegante y utilizando tipos. De la misma forma, íntegra principios de orientación a objetos y programación funcional, permitiendo a los programadores ser más productivo. [10]

Se optó por Spark debido a que nos brinda una herramienta 'Todo en uno', esto es que permite la combinación de consultar datos (Spark sql), procesar datos en lotes, y contiene MLlib (Machine Learning).
Una de las ventajas de trabajar con Spark son las consolas interactivas que tiene para dos de los lenguajes (Scala y Phyton) con los que se puede programar, en este caso se optó por Scala, ya que esta consola permite analizar los datos de forma interactiva, y con la conexión a los clústeres.

4. RESULTADOS

Se realizaron un total de 10 pruebas para cada uno de los algoritmos a analizar; en todos los casos se evaluaron los mismos parámetros:

- Accuracy o nivel de exactitud.
- Test Error o porcentaje de error.
- Tiempo en segundos.

Los resultados obtenidos para cada ejecución del código se registraron en las siguientes tablas.

La siguiente tabla (Tabla 1) corresponde a los resultados obtenidos para el algoritmo de Support Vector Machine o SVM. En este caso se puede observar que los valores tanto para Accuracy como el Test Error no tuvieron variaciones, es decir se mantuvieron constantes, sin embargo, el tiempo si obtuvo un aumento, o se vió disminuido, según se avanzaba con la cantidad de pruebas realizadas.

No. Prueba	Accuracy (Exactitud)	Test Error	Tiempo (seg.)
No.1	0.8901808785529716	0.1098191214470284	375.6718289
No. 2	0.8901808785529716	0.1098191214470284	363.9957071
No. 3	0.8901808785529716	0.1098191214470284	363.0835897
No. 4	0.8901808785529716	0.1098191214470284	390.4538277
No. 5	0.8901808785529716	0.1098191214470284	358.3278896
No. 6	0.8901808785529716	0.1098191214470284	381.9333628
No. 7	0.8901808785529716	0.1098191214470284	366.4412427
No. 8	0.8901808785529716	0.1098191214470284	359.8022371
No. 9	0.8901808785529716	0.1098191214470284	363.1190171
No. 10	0.8901808785529716	0.1098191214470284	356.9475381

Tabla 1.- Resultados del Algoritmo SVM.

La siguiente tabla (Tabla 2) corresponde a los resultados obtenidos para el algoritmo de Árboles de Decisión (Decision Tree). Al igual que en el caso anterior, los valores para

Accuracy y Test Error se mantuvieron constantes, y el tiempo fue el valor con más cambios.
En esta ocasión, los tiempos obtenidos fueron mucho menores, a comparación del algoritmo anterior, lo cual nos indica que el modelo fue capaz de trabajar de manera más rápida. También, por los valores obtenidos para accuracy y test error, se puede inferir que el modelo es más preciso, y comete una menor cantidad de errores.

No. Prueba	Accuracy (Exactitud)	Test Error	Tiempo (seg.)
No.1	0.914001937984496 1	0.085998062015503 86	14.938472985
No. 2	0.914001937984496 1	0.085998062015503 86	8.864550009
No. 3	0.914001937984496 1	0.085998062015503 86	7.862777911
No. 4	0.914001937984496 1	0.085998062015503 86	7.064012839
No. 5	0.914001937984496 1	0.085998062015503 86	6.58651064
No. 6	0.914001937984496 1	0.085998062015503 86	6.768532076
No. 7	0.914001937984496 1	0.085998062015503 86	6.988422528
No. 8	0.914001937984496 1	0.085998062015503 86	6.846584689
No. 9	0.914001937984496 1	0.085998062015503 86	6.946460378
No. 10	0.914001937984496 1	0.085998062015503 86	7.113414449

Tabla 2.- Resultados del Algoritmo Decision Tree.

La siguiente tabla (Tabla 3) corresponde a los resultados obtenidos para el algoritmo de Regresión Logística (Logistic Regression). De la misma manera que los casos anteriores, los valores para Accuracy y Test Error se mantuvieron constantes, y el tiempo fue el valor con más cambios.
En esta ocasión, a primera vista, los tiempos obtenidos fueron un tanto similares a los del algoritmo anterior, sin embargo, en un análisis más profundo, se puede observar que el modelo anterior fue capaz de trabajar, todavía, de manera un poco más rápida. También, por los valores obtenidos para accuracy y test error, se puede inferir que el modelo anterior (Decision Tree) es, por poco, aún más preciso, y comete una menor cantidad de errores.

No. Prueba	Accuracy (Exactitud)	Test Error	Tiempo (seg.)
No.1	0.909964470284237 8	0.090035529715 77	23.132142516
No. 2	0.909964470284237 8	0.090035529715 77	10.20661212
No. 3	0.909964470284237 8	0.090035529715 77	8.324216133
No. 4	0.909964470284237 8	0.090035529715 77	7.82093714
No. 5	0.909964470284237 8	0.090035529715 77	7.569859223
No. 6	0.909964470284237 8	0.090035529715 77	7.470137036
No. 7	0.909964470284237 8	0.090035529715 77	7.917381598
No. 8	0.909964470284237 8	0.090035529715 77	7.841127548

No. 9	0.9099644702842378	0.090035529715 7	8.191968182
No. 10	0.9099644702842378	0.090035529715 7	7.886081448

Tabla 3.- Resultados del Algoritmo Logistic Regression.

La siguiente tabla (Tabla 4) corresponde a los resultados obtenidos para el algoritmo de Multilayer Perceptron. Del mismo modo que los casos anteriores, los valores para Accuracy y Test Error se mantuvieron constantes, y el tiempo fue el valor con más cambios.

En esta ocasión, los tiempos obtenidos volvieron a aumentar, lo cual nos indica que a comparación de los dos casos anteriores, este modelo no fue capaz de trabajar más rápido o a la par de los mismos.

También, por los valores obtenidos para accuracy y test error, se puede inferir que los modelos anteriores (Decision Tree y Logistic Regression) fueron más precisos, y cometieron una menor cantidad de errores.

No. Prueba	Accuracy (Exactitud)	Test Error	Tiempo (seg.)
No.1	0.8901808785529716	0.10981912144702843	36.986344957
No. 2	0.8901808785529716	0.10981912144702843	25.344787648
No. 3	0.8901808785529716	0.10981912144702843	23.817277119
No. 4	0.8901808785529716	0.10981912144702843	23.405495415
No. 5	0.8901808785529716	0.10981912144702843	23.039295538
No. 6	0.8901808785529716	0.10981912144702843	23.231313813
No. 7	0.8901808785529716	0.10981912144702843	23.396428692
No. 8	0.8901808785529716	0.10981912144702843	23.992974404
No. 9	0.8901808785529716	0.10981912144702843	23.451315893
No. 10	0.8901808785529716	0.10981912144702843	23.489903215

Tabla 4.- Resultados del Algoritmo Multilayer Perceptron.

La siguiente tabla (Tabla 5) corresponde a los promedios de todos los resultados obtenidos para cada algoritmo.

Aquí se puede apreciar de forma más clara cuál fue el modelo con los mejores resultados tanto para accuracy, como test error y tiempo de ejecución.

Algoritmo	Valores Promedio		
	Accuracy	Test Error	Tiempo (seg.)
SVM	0.8901808785529716	0.10981912144702843	367.97762408
Decision Tree	0.9140019379844961	0.085998062015503 86	7.9979738504
Logistic Regression	0.9099644702842378	0.090035529 7157	9.6360462944
Multilayer Perceptron	0.8901808785529716	0.10981912144702843	25.0155136536

Tabla 5.- Comparación de Resultados de los Algoritmos: SVM, Decision Tree, Logistic Regression y Multilayer Perceptron.

5. CONCLUSIONES

Una vez realizadas las pruebas se obtuvo que el algoritmo de aprendizaje automático de "Árboles de Decisión" es el que presenta un mejor rendimiento, desde su nivel de exactitud (con un promedio de 0.9140019379844961 ≈ 91.40%), hasta su porcentaje de error (con el valor más bajo de 0.08599806201550386 ≈ 8.60%), y tiempo de ejecución (siendo el más rápido con un promedio de 7.9979738504 segundos); por lo anterior, se puede decir que este algoritmo cuenta con una alta precisión, estabilidad y sobre todo que es fácil de interpretar.

Como segundo puesto se obtuvo al algoritmo de Logistic Regression, del cual también se obtuvo un buen rendimiento, con un porcentaje de exactitud del 91%, mientras que el promedio del porcentaje de error fue aproximadamente de un 9%; por otro lado también se puede considerar que es un algoritmo rápido de ejecutar, ya que este tardó un promedio de 9.63 segundos, lo cual se debe a que no requiere de muchos recursos para su ejecución.

Finalmente se pudo observar que los algoritmos de SVM y Multilayer Perceptron arrojaron tanto los mismos niveles de exactitud (0.8901808785529716 ≈ 90%), como los mismos porcentajes de error (0.10981912144702843 ≈ 11%), siendo así, el tiempo de ejecución el único valor que diferencia a ambos: para el Multilayer Perceptron se obtuvo un tiempo promedio de 25.015 segundos, mientras que para el SVM se obtuvo un promedio de 367.977 segundos, lo cual colocaría a este último, como el algoritmo con el rendimiento más bajo de todos.

REFERENCIAS

[1] Aggarwal, C. C. (2015). Data Mining: The Textbook (2015.ª ed.). Berlín, Alemania: Springer.

[2] Awad, Mariette & Khanna, Rahul. (2015). Support Vector Machines for Classification. 10.1007/978-1-4302-5990-9_3.

[3] ICEMD. (2019, 31 agosto). Apache Spark: Introducción, qué es y cómo funciona. Recuperado 12 de junio de 2020, de https://www.icemd.com/digital-knowledge/articulos/apache-spark-introduccion-que-es-y-como-funciona/

[4] Friedl, M. A., & Brodley, C. E. (1997). Decision tree classification of land cover from remotely sensed data. Remote sensing of environment, 61(3), 399-409.

[5] González, L. (2019, 20 diciembre). Ventajas y Desventajas de los Algoritmos de Clasificación. Recuperado 12 de junio de 2020, de https://ligdigonzalez.com/ventajas-y-desventajas-de-los-algoritmos-de-clasificacion-machine-learning/

[6] Shalev-Shwartz, S., & Ben-David, S. (2014). Understanding Machine Learning: From Theory to Algorithms (1.ª ed.). Cambridge, Inglaterra: Cambridge University Press.

[7] Marius, Popescu & Balas, Valentina & Perescu-Popescu, Liliana & Mastorakis, Nikos. (2009). Multilayer perceptron and neural networks. WSEAS Transactions on Circuits and Systems. 8.

[8] S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, Junio 2014.

[9] Cortés P."Multilayer Perceptron (mlp)".Universidad de Minho.[en línea] link:http://www3.dsi.uminho.pt/pcortez/mg.pdf (24 de abril del 2015)

[10] What's in Apache Spark & Scala? - Quora. (2018, 14 junio). Recuperado 12 de junio de 2020, de https://www.quora.com/Whats-in-Apache-Spark-Scala

ANEXOS

A. DATASET UTILIZADO

Bank Marketing Data Set

- **Resumen:**
  Los datos están relacionados con campañas de marketing directo (llamadas telefónicas) de una institución bancaria portuguesa. El objetivo de clasificación es predecir si el cliente se suscribirá a un depósito a plazo (variable y).

- **Características:**

Características del Dataset:	Multivariante	Número de instancias:	45211	Área:	Negocios
Características del atributo:	Real	Número de atributos:	17	Fecha de donación:	2012-02-14
Tareas asociadas:	Clasificación	¿Valores faltantes?	N/A	Número de visitas web:	1214749

- **Link:**
  https://archive.ics.uci.edu/ml/datasets/Bank+Marketing

B. ALGORITMO: SVM

1. CÓDIGO

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()

import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

val dataset =
spark.read.option("header","true").option("inferSchema","true").csv("ba
nk-additional-full.csv")

import org.apache.spark.ml.classification.LinearSVC
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

val coly = when($"y".contains("yes"), 1.0).otherwise(0.0)
val df = dataset.withColumn("y", coly)
```

```
val data = df.select(df("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")

import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

val assembler = new
VectorAssembler().setInputCols(Array("age","duration","campaign","pdays
","previous","emp_var_rate","cons_price_idx","cons_conf_idx","euribor3m
","nr_employed")).setOutputCol("features")

val features = assembler.transform(data)

import org.apache.spark.ml.feature.StringIndexer
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(f
eatures)

import org.apache.spark.ml.feature.VectorIndexer
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures")
.setMaxCategories(4).fit(features)

val splits = features.randomSplit(Array(0.7, 0.3), seed = 1234L)
val train = splits(0)
val test = splits(1)

val t1 = System.nanoTime

val lsvc = new LinearSVC().setMaxIter(100).setRegParam(0.1)

val lsvcModel = lsvc.fit(test)

import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(labelIndexer,
featureIndexer, lsvc))
val model = pipeline.fit(train)

val predictions = model.transform(test)
```

```
val predictionAndLabels = predictions.select("prediction", "label")
val evaluator = new MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel")
.setPredictionCol("prediction")
.setMetricName("accuracy")

val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))
println("Accuracy = " + accuracy)

println(s"Coefficients: ${lsvcModel.coefficients} Intercept:
${lsvcModel.intercept}")

val duration = (System.nanoTime - t1) / 1e9d
```

2. CÓDIGO CON EXPLICACIÓN

Support Vector Machine (SVM)

1. Iniciar nueva sesión de spark.

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()
```

2. Código para reducir errores durante la ejecución.

```
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)
```

3. Cargar el archivo con el dataset.

```
val dataset =
spark.read.option("header","true").option("inferSchema","true").csv("ba
nk-additional-full.csv")
```

> Se utilizan la opciones "header" e "inferSchema" para obtener el título y tipo de dato de cada columna en el dataset.

4. Importar las librerías para trabajar con SVM.

```
import org.apache.spark.ml.classification.LinearSVC
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

La primera librería sí corresponde y es necesaria para la implementación del modelo SVM, no obstante, la segunda librería se utiliza para evaluar la precisión de los modelos de clasificación, ya sea que se trate de SVM o algún otro algoritmo.

5. Transformación de los datos.

```
val coly = when($"y".contains("yes"), 1.0).otherwise(0.0)
val df = dataset.withColumn("y", coly)
```

Se buscan los valores iguales a "yes" de la columna "y", y se sustituyen por 1, o en el caso contrario por 0.

Una vez cambiados los valores, estos se vuelven a insertar en la columna "y".

6. Transformación para los datos categóricos (etiquetas a clasificar).

```
val data = df.select(df("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")
```

Se seleccionan las columnas del dataframe y se renombra la columna "y" como "label".

7. Importar las librerías VectorAssembler y Vectors.

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
```

8. Crear nuevo objeto VectorAssembler.

```
val assembler = new
VectorAssembler().setInputCols(Array("age","duration","campaign","pdays
","previous","emp_var_rate","cons_price_idx","cons_conf_idx","euribor3m
","nr_employed")).setOutputCol("features")
val features = assembler.transform(data)
```

Se crea un nuevo objeto VectorAssembler llamado assembler para guardar el resto de las columnas como features. Se crea la variable features para guardar el dataframe con los cambios anteriores.

9. Importar la libreria StringIndexer.

```
import org.apache.spark.ml.feature.StringIndexer
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(f
eatures)
```

Indexamos la columna label y añadimos metadata, esto es para cambiar los valores tipo texto de las etiquetas por valores numéricos.

También se hace un ajuste a todo el dataset para incluir todas las etiquetas en el índice.

10. Importar la librería VectorIndexer.

```
import org.apache.spark.ml.feature.VectorIndexer
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures")
.setMaxCategories(4).fit(features)
```

Se indexa la columna "features" y se establece el número máximo de categorías a tomar como 4.

También se hace un ajuste a todo el dataset para incluir todas las etiquetas en el índice.

11. Dividir datos.

```
val splits = features.randomSplit(Array(0.7, 0.3), seed = 1234L)
val train = splits(0)
val test = splits(1)
```

Se dividen los datos en datos de entrenamiento (0.7) y de prueba (0.3) con randomSplit.

Se crean las variables train y test para guardar los datos divididos.

12. Tiempo de ejecución.

```
val t1 = System.nanoTime
```

System.nanoTime se utiliza para medir la diferencia de tiempo transcurrido.

Esta línea se coloca antes de que comience el código del cual se desea tomar el tiempo de ejecución.

13. Crear modelo.

```
val lsvc = new LinearSVC().setMaxIter(100).setRegParam(0.1)
```

Se crea el modelo (lsvc) y se establecen sus parámetros.

- setMaxIter = número máximo de iteraciones a realizar por el modelo.
- setRegParam = establecer el parámetro de regularización. El valor predeterminado es 0.0.

14. Ajustar el modelo.

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

Se ajusta el modelo para que trabaje con los datos de prueba y, posteriormente se puedan obtener los coeficientes y la intercepción.

15. Importar la librería para utilizar Pipeline.

```
import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(labelIndexer,
featureIndexer, lsvc))
val model = pipeline.fit(train)
```

Se unen los indexadores y el modelo en una Pipeline.

Se entrena el modelo, y se ajusta la Pipeline para trabajar con los datos de entrenamiento.

16. Imprimir los resultados del modelo.

```
val predictions = model.transform(test)
val predictionAndLabels = predictions.select("prediction", "label")
val evaluator = new MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel")
.setPredictionCol("prediction")
.setMetricName("accuracy")
```

Se utiliza transform para cambiar los datos a utilizar, de train a test.

Se seleccionan las columnas de predicción y "label", y se guardan dentro de la variable "predictionAndLabels".

Se crea un evaluador con la función MulticlassClassificationEvaluator(), en donde se seleccionan las columnas "indexedLabel" y "prediction", y se calcula el nivel de exactitud (accuracy) del modelo.

```
val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))
println("Accuracy = " + accuracy)
```

Se utiliza evaluate para valorar las predicciones hechas por el modelo.

Se imprime en consola el porcentaje de error como resultado de la resta: 1.0 - accuracy; así como el valor obtenido para el accuracy.

```
println(s"Coefficients: ${lsvcModel.coefficients} Intercept:
${lsvcModel.intercept}")
```

Se imprimen los coeficientes y la intercepción obtenidas para el modelo.

17. Imprimir tiempo total de ejecución.

```
val duration = (System.nanoTime - t1) / 1e9d
```

Esta línea se coloca al final del código del cual se desea tomar el tiempo de ejecución.

El tiempo se obtiene como resultado de la resta: tiempo actual en nanosegundos (System.nanoTime) menos el tiempo al iniciar el código (en este caso la variable denominada t1).

Como el resultado se encuentra en nanosegundos se utiliza una división entre 1e9d para obtener el tiempo en segundos.

- 1e9d = operación 10^9, y la "d" es para indicar que el resultado sea de tipo double.

Resultados:

- Porcentaje de error promedio = 0.10981912144702843 ≈ 11%
- Nivel de exactitud promedio = 0.8901808785529716 ≈ 89%
- Tiempo de ejecución promedio = 367.97762408

C. CÓDIGO PARA EL ALGORITMO: DECISION TREE

1. CÓDIGO

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()

import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

val dataset =
spark.read.option("header","true").option("inferSchema","true").csv("ba
nk-additional-full.csv")

import
org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

```
val data = dataset.select(dataset("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")

import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

val assembler = new
VectorAssembler().setInputCols(Array("age","duration","campaign","pdays
","previous","emp_var_rate","cons_price_idx","cons_conf_idx","euribor3m
","nr_employed")).setOutputCol("features")

val df = assembler.transform(data)

import org.apache.spark.ml.feature.StringIndexer
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(d
f)

import org.apache.spark.ml.feature.VectorIndexer
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures")
.setMaxCategories(4).fit(df)

val Array(trainingData, testData) = df.randomSplit(Array(0.7, 0.3),
seed = 1234L)

val t1 = System.nanoTime

val dt = new
DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("in
dexedFeatures")

import org.apache.spark.ml.feature.IndexToString
val labelConverter = new IndexToString()
.setInputCol("prediction")
.setOutputCol("predictedLabel")
.setLabels(labelIndexer.labels)

import org.apache.spark.ml.Pipeline
```

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

val model = pipeline.fit(trainingData)

val predictions = model.transform(testData)
predictions.select("predictedLabel", "label", "features").show(5)
val evaluator = new MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel")
.setPredictionCol("prediction")
.setMetricName("accuracy")

val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))

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

val duration = (System.nanoTime - t1) / 1e9d
```

2. CÓDIGO CON EXPLICACIÓN

Decision Tree

1. Iniciar nueva sesión de spark.

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()
```

2. Código para reducir errores durante la ejecución.

```
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)
```

3. Cargar el archivo con el dataset.

```
val dataset =
spark.read.option("header","true").option("inferSchema","true").csv("ba
nk-additional-full.csv")
```

Se utilizan la opciones "header" e "inferSchema" para obtener el título y tipo de dato de cada columna en el dataset.

4. Importar las librerías para trabajar con Decision Tree.

```
import
org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

Las primeras dos librerías sí corresponden y son necesarias para la implementación del modelo de Decision Tree, no obstante, la tercera librería se utiliza para evaluar la precisión de los modelos de clasificación, ya sea que se trate de Decision Tree o algún otro algoritmo.

5. Transformación para los datos categóricos (etiquetas a clasificar).

```
val data = dataset.select(dataset("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")
```

Se seleccionan las columnas del dataframe y se renombra la columna "y" como "label".

6. Importar las librerías VectorAssembler y Vectors.

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
```

7. Crear nuevo objeto VectorAssembler.

```
val assembler = new
VectorAssembler().setInputCols(Array("age","duration","campaign","pdays
","previous","emp_var_rate","cons_price_idx","cons_conf_idx","euribor3m
","nr_employed")).setOutputCol("features")
val df = assembler.transform(data)
```

Se crea un nuevo objeto VectorAssembler llamado assembler para guardar el resto de las columnas como features. Se crea la variable df para guardar el dataframe con los cambios anteriores.

8. Importar la libreria StringIndexer.

```
import org.apache.spark.ml.feature.StringIndexer
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(d
f)
```

Indexamos la columna label y añadimos metadata, esto es para cambiar los valores tipo texto de las etiquetas por valores numéricos.

También se hace un ajuste a todo el dataset para incluir todas las etiquetas en el índice.

9. Importar la librería VectorIndexer.

```
import org.apache.spark.ml.feature.VectorIndexer
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures")
.setMaxCategories(4).fit(df)
```

Se indexa la columna "features" y se establece el número máximo de categorías a tomar como 4.

También se hace un ajuste a todo el dataset para incluir todas las etiquetas en el índice.

10. Dividir datos.

```
val Array(trainingData, testData) = df.randomSplit(Array(0.7, 0.3),
seed = 1234L)
```

Se dividen los datos en datos de entrenamiento (0.7) y de prueba (0.3) con randomSplit.

Los datos divididos se guardan como trainingData y testData.

11. Tiempo de ejecución.

```
val t1 = System.nanoTime
```

System.nanoTime se utiliza para medir la diferencia de tiempo transcurrido.

Esta línea se coloca antes de que comience el código del cual se desea tomar el tiempo de ejecución.

12. Crear modelo.

```
val dt = new
DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("in
dexedFeatures")
```

Se crea el modelo (dt) y se establecen sus parámetros.

- setLabelCol = columna label indexada indexedLabel.
- setFeaturesCol = columna features indexada indexedFeatures.

13. Importar la librería IndexToString.

```
import org.apache.spark.ml.feature.IndexToString
val labelConverter = new IndexToString()
.setInputCol("prediction")
.setOutputCol("predictedLabel")
.setLabels(labelIndexer.labels)
```

Se convierten los valores de las etiquetas indexadas en los de las etiquetas originales, y se les asigna el nombre de "predictedLabel".

14. Importar la librería para utilizar Pipeline.

```
import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(labelIndexer,
featureIndexer, dt, labelConverter))
val model = pipeline.fit(train)
```

Se unen los indexadores y el modelo en una Pipeline.

Se entrena el modelo, y se ajusta la Pipeline para trabajar con los datos de entrenamiento.

15. Imprimir los resultados del modelo.

```
val predictions = model.transform(testData)
predictions.select("predictedLabel", "label", "features").show(5)
val evaluator = new MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel")
.setPredictionCol("prediction")
.setMetricName("accuracy")
```

Se utiliza transform para cambiar los datos a utilizar, de train a test.

Se seleccionan 5 filas de ejemplo para mostrar en consola.

Se crea un evaluador con la función MulticlassClassificationEvaluator(), en donde se seleccionan las columnas "indexedLabel" y "prediction", y se calcula el nivel de exactitud (accuracy) del modelo.

```
val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))
```

Se utiliza evaluate para valorar las predicciones hechas por el modelo.

Se imprime en consola el porcentaje de error como resultado de la resta: 1.0 - accuracy.

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

```
println(s"Learned classification tree model:\n
${treeModel.toDebugString}")
```

Se imprime el árbol generado por el modelo.

16. Imprimir tiempo total de ejecución.

```
val duration = (System.nanoTime - t1) / 1e9d
```

Esta línea se coloca al final del código del cual se desea tomar el tiempo de ejecución.

El tiempo se obtiene como resultado de la resta: tiempo actual en nanosegundos (System.nanoTime) menos el tiempo al iniciar el código (en este caso la variable denominada t1).

Como el resultado se encuentra en nanosegundos se utiliza una división entre 1e9d para obtener el tiempo en segundos.

- 1e9d = operación 10^9, y la "d" es para indicar que el resultado sea de tipo double.

Resultados:

- Porcentaje de error promedio = 0.08599806201550386 ≈ 9%
- Nivel de exactitud promedio = 0.9140019379844961 ≈ 91%
- Tiempo de ejecución promedio = 7.9979738504

D. CÓDIGO PARA EL ALGORITMO: LOGISTIC REGRESSION

1. CÓDIGO

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()

import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.mllib.evaluation.MulticlassMetrics

val dataset  = spark.read.option("header","true").option("inferSchema",
"true").format("csv").load("bank-additional-full.csv")

val coly = when($"y".contains("yes"), 1.0).otherwise(0.0)
```

```
val df = dataset.withColumn("y", coly)

val data = df.select(df("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")

import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

val assembler = (new
VectorAssembler().setInputCols(Array("age","duration","campaign","pdays
","previous","emp_var_rate","cons_price_idx","cons_conf_idx","euribor3m
","nr_employed")).setOutputCol("features"))

val Array(training, test) = data.randomSplit(Array(0.7, 0.3), seed =
1234L)

val t1 = System.nanoTime

val lr = new LogisticRegression().setMaxIter(100)

import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(assembler,lr))
val model = pipeline.fit(training)
val results = model.transform(test)
val predictionAndLabels =
results.select($"prediction",$"label").as[(Double, Double)].rdd
val metrics = new MulticlassMetrics(predictionAndLabels)

println("Confusion matrix:")
println(metrics.confusionMatrix)

metrics.accuracy

val duration = (System.nanoTime - t1) / 1e9d
```

2. CÓDIGO CON EXPLICACIÓN

Logistic Regression

1. Iniciar nueva sesión de spark.

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()
```

2. Código para reducir errores durante la ejecución.

```
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)
```

3. Cargar el archivo con el dataset.

```
val dataset  = spark.read.option("header","true").option("inferSchema",
"true").format("csv").load("bank-additional-full.csv")
```

Se utilizan la opciones "header" e "inferSchema" para obtener el título y tipo de dato de cada columna en el dataset.

4. Importar las librerías para trabajar con Logistic Regression.

```
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.mllib.evaluation.MulticlassMetrics
```

La primera librería corresponde y es necesaria para la implementación del modelo de Logistic Regression.

La segunda librería es un tanto vieja, sin embargo, es la que se utiliza para evaluar la precisión de este tipo de algoritmo.

5. Transformación de los datos.

```
val coly = when($"y".contains("yes"), 1.0).otherwise(0.0)
val df = dataset.withColumn("y", coly)
```

Se buscan los valores iguales a "yes" de la columna "y", y se sustituyen por 1, o en el caso contrario por 0.

Una vez cambiados los valores, estos se vuelven a insertar en la columna "y".

6. Transformación para los datos categóricos (etiquetas a clasificar).

```
val data = df.select(df("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")
```

Se seleccionan las columnas del dataframe y se renombra la columna "y" como "label".

7. Importar las librerías VectorAssembler y Vectors.

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
```

8. Crear nuevo objeto VectorAssembler.

```
val assembler = (new VectorAssembler()
.setInputCols(Array("age","duration","campaign","pdays","previous","emp
_var_rate","cons_price_idx","cons_conf_idx","euribor3m","nr_employed"))
.setOutputCol("features"))
```

Se crea un nuevo objeto VectorAssembler llamado assembler para guardar el resto de las columnas como features.

9. Dividir datos.

```
val Array(training, test) = data.randomSplit(Array(0.7, 0.3), seed =
1234L)
```

Se dividen los datos en datos de entrenamiento (0.7) y de prueba (0.3) con randomSplit.

Los datos divididos se guardan como training y test.

10. Tiempo de ejecución.

```
val t1 = System.nanoTime
```

System.nanoTime se utiliza para medir la diferencia de tiempo transcurrido.

Esta línea se coloca antes de que comience el código del cual se desea tomar el tiempo de ejecución.

11. Crear modelo.

```
val lr = new LogisticRegression().setMaxIter(100)
```

Se crea el modelo (lr) y se establecen sus parámetros.

- setMaxIter = número máximo de iteraciones a realizar por el modelo.

12. Importar la librería para utilizar Pipeline.

```
import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline().setStages(Array(assembler,lr))
val model = pipeline.fit(training)
```

Se unen el assembler y el modelo en una Pipeline.

Se entrena el modelo, y se ajusta la Pipeline para trabajar con los datos de entrenamiento.

13. Imprimir los resultados del modelo.

```
val results = model.transform(test)
val predictionAndLabels =
results.select($"prediction",$"label").as[(Double, Double)].rdd
val metrics = new MulticlassMetrics(predictionAndLabels)
```

Se utiliza transform para cambiar los datos a utilizar, de train a test.

Se convierten los resultados de prueba (test) en RDD utilizando .as y .rdd.

- .rdd = estructura de datos de spark para ver los resultados.

Se inicializa un objeto MulticlassMetrics().

- metrics = se utiliza para realizar distintas pruebas para comprobar la precisión del modelo.

```
println("Confusion matrix:")
println(metrics.confusionMatrix)
```

Se imprime la matriz de confusión para las predicciones del modelo.

```
metrics.accuracy
```

Se imprime el nivel de exactitud (accuracy) del modelo.

14. Imprimir tiempo total de ejecución.

```
val duration = (System.nanoTime - t1) / 1e9d
```

Esta línea se coloca al final del código del cual se desea tomar el tiempo de ejecución.

El tiempo se obtiene como resultado de la resta: tiempo actual en nanosegundos (System.nanoTime) menos el tiempo al iniciar el código (en este caso la variable denominada t1).

Como el resultado se encuentra en nanosegundos se utiliza una división entre 1e9d para obtener el tiempo en segundos.

- 1e9d = operación 10^9, y la "d" es para indicar que el resultado sea de tipo double.

Resultados:

- Porcentaje de error promedio = 0.0900355297157 ≈ 9%
- Nivel de exactitud promedio = 0.9099644702842378 ≈ 91%
- Tiempo de ejecución promedio = 9.6360462944

E. CÓDIGO PARA EL ALGORITMO: MULTILAYER PERCEPTRON

1. CÓDIGO

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()

import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)

val df = spark.read.option("header",
"true").option("inferSchema","true")csv("bank-additional-full.csv")

import
org.apache.spark.ml.classification.MultilayerPerceptronClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

val data = df.select(df("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")

import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

val assembler = new
VectorAssembler().setInputCols(Array("age","duration","campaign","pdays
","previous","emp_var_rate","cons_price_idx","cons_conf_idx","euribor3m
","nr_employed")).setOutputCol("features")

val features = assembler.transform(data)

import org.apache.spark.ml.feature.StringIndexer
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(f
eatures)
```

```
import org.apache.spark.ml.feature.VectorIndexer
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures")
.setMaxCategories(4).fit(features)

val splits = features.randomSplit(Array(0.7, 0.3), seed = 1234L)
val train = splits(0)
val test = splits(1)

val t1 = System.nanoTime

val layers = Array[Int](10, 11, 3, 2)

val trainer = new MultilayerPerceptronClassifier()
.setLayers(layers)
.setLabelCol("indexedLabel")
.setFeaturesCol("indexedFeatures")
.setBlockSize(128)
.setSeed(1234L)
.setMaxIter(100)

import org.apache.spark.ml.feature.IndexToString
val labelConverter = new IndexToString()
.setInputCol("prediction")
.setOutputCol("predictedLabel")
.setLabels(labelIndexer.labels)

import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline()
.setStages(Array(labelIndexer, featureIndexer, trainer,
labelConverter))
val model = pipeline.fit(train)
val predictions = model.transform(test)
val predictionAndLabels = predictions.select("prediction", "label")

val evaluator = new MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel")
.setPredictionCol("prediction")
.setMetricName("accuracy")
```

```
val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))

val duration = (System.nanoTime - t1) / 1e9d
```

2. CÓDIGO CON EXPLICACIÓN

Multilayer Perceptron

1. Iniciar nueva sesión de spark.

```
import org.apache.spark.sql.SparkSession
val spar = SparkSession.builder().getOrCreate()
```

2. Código para reducir errores durante la ejecución.

```
import org.apache.log4j._
Logger.getLogger("org").setLevel(Level.ERROR)
```

3. Cargar el archivo con el dataset.

```
val df = spark.read.option("header",
"true").option("inferSchema","true")csv("bank-additional-full.csv")
```

Se utilizan la opciones "header" e "inferSchema" para obtener el título y tipo de dato de cada columna en el dataset.

4. Importar las librerías para trabajar con Multilayer Perceptron.

```
import
org.apache.spark.ml.classification.MultilayerPerceptronClassifier
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

La primera librería sí corresponde y es necesaria para la implementación del modelo de Multilayer Perceptron, no obstante, la segunda librería se utiliza para evaluar la precisión de los modelos de clasificación, ya sea que se trate de Multilayer Perceptron o algún otro algoritmo.

5. Transformación para los datos categóricos (etiquetas a clasificar).

```
val data = df.select(df("y").as("label"),
$"age",$"duration",$"campaign",$"pdays",$"previous",$"emp_var_rate",$"c
ons_price_idx",$"cons_conf_idx",$"euribor3m",$"nr_employed")
```

Se seleccionan las columnas del dataframe y se renombra la columna "y" como "label".

6. Importar las librerías VectorAssembler y Vectors.

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
```

7. Crear nuevo objeto VectorAssembler.

```
val assembler = new
VectorAssembler().setInputCols(Array("age","duration","campaign","pdays
","previous","emp_var_rate","cons_price_idx","cons_conf_idx","euribor3m
","nr_employed")).setOutputCol("features")
val features = assembler.transform(data)
```

Se crea un nuevo objeto VectorAssembler llamado assembler para guardar el resto de las columnas como features. Se crea la variable features para guardar el dataframe con los cambios anteriores.

8. Importar la libreria StringIndexer.

```
import org.apache.spark.ml.feature.StringIndexer
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(f
eatures)
```

Indexamos la columna label y añadimos metadata, esto es para cambiar los valores tipo texto de las etiquetas por valores numéricos.

También se hace un ajuste a todo el dataset para incluir todas las etiquetas en el índice.

9. Importar la librería VectorIndexer.

```
import org.apache.spark.ml.feature.VectorIndexer
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures")
.setMaxCategories(4).fit(features)
```

Se indexa la columna "features" y se establece el número máximo de categorías a tomar como 4.

También se hace un ajuste a todo el dataset para incluir todas las etiquetas en el índice.

10. Dividir datos.

```
val splits = features.randomSplit(Array(0.7, 0.3), seed = 1234L)
```

```
val train = splits(0)
val test = splits(1)
```

Se dividen los datos en datos de entrenamiento (0.7) y de prueba (0.3) con randomSplit.

Se crean las variables train y test para guardar los datos divididos.

11. Tiempo de ejecución.

```
val t1 = System.nanoTime
```

System.nanoTime se utiliza para medir la diferencia de tiempo transcurrido.

Esta línea se coloca antes de que comience el código del cual se desea tomar el tiempo de ejecución.

12. Especificar capas para la red neuronal.

```
val layers = Array[Int](10, 11, 3, 2)
```

Se especifican las capas para la red neuronal del modelo:

- Capa de entrada de tamaño 10 (características).
- Dos capas intermedias de tamaño 11 y 3.
- Capa de salida de tamaño 2 (clases).

13. Crear modelo.

```
val trainer = new MultilayerPerceptronClassifier()
.setLayers(layers)
.setLabelCol("indexedLabel")
.setFeaturesCol("indexedFeatures")
.setBlockSize(128)
.setSeed(1234L)
.setMaxIter(100)
```

Se crea el entrenador y se establecen sus parámetros.

- setLayers = variable creada en el paso anterior layers.
- setLabelCol = columna label indexada indexedLabel.
- setFeaturesCol = columna features indexada indexedFeatures.
- setBlockSize = tamaño del bloque por defecto en kilobytes (128).
- setSeed = aleatoriedad en los datos (1234L).
- setMaxIter = número máximo de iteraciones a realizar por el modelo (100 es el valor predeterminado).

14. Importar librería IndexToString.

```
import org.apache.spark.ml.feature.IndexToString
```

```
val labelConverter = new IndexToString()
.setInputCol("prediction")
.setOutputCol("predictedLabel")
.setLabels(labelIndexer.labels)
```

Se convierten los valores de las etiquetas indexadas en los de las etiquetas originales, y se les asigna el nombre de "predictedLabel".

15. Importar la librería para utilizar Pipeline.

```
import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline()
.setStages(Array(labelIndexer, featureIndexer, trainer,
labelConverter))
val model = pipeline.fit(train)
```

Se unen los indexadores y el modelo (trainer) en una Pipeline.

Se entrena el modelo, y se ajusta la Pipeline para trabajar con los datos de entrenamiento.

16. Imprimir los resultados del modelo.

```
val predictions = model.transform(test)
val predictionAndLabels = predictions.select("prediction", "label")
val evaluator = new MulticlassClassificationEvaluator()
.setLabelCol("indexedLabel")
.setPredictionCol("prediction")
.setMetricName("accuracy")
```

Se utiliza transform para cambiar los datos a utilizar, de train a test.

Se seleccionan las columnas de predicción y "label", y se guardan dentro de la variable "predictionAndLabels".

Se crea un evaluador con la función MulticlassClassificationEvaluator(), en donde se seleccionan las columnas "indexedLabel" y "prediction", y se calcula el nivel de exactitud (accuracy) del modelo.

```
val accuracy = evaluator.evaluate(predictions)
println("Test Error = " + (1.0 - accuracy))
```

Se utiliza evaluate para valorar las predicciones hechas por el modelo.

Se imprime en consola el porcentaje de error como resultado de la resta: 1.0 - accuracy.

17. Imprimir tiempo total de ejecución.

```
val duration = (System.nanoTime - t1) / 1e9d
```

Esta línea se coloca al final del código del cual se desea tomar el tiempo de ejecución.

El tiempo se obtiene como resultado de la resta: tiempo actual en nanosegundos (System.nanoTime) menos el tiempo al iniciar el código (en este caso la variable denominada t1).

Como el resultado se encuentra en nanosegundos se utiliza una división entre 1e9d para obtener el tiempo en segundos.

- 1e9d = operación 10^9, y la "d" es para indicar que el resultado sea de tipo double.

Resultados:

- Porcentaje de error promedio = 0.10981912144702843 ≈ 11%
- Nivel de exactitud promedio = 0.8901808785529716 ≈ 89%
- Tiempo de ejecución promedio = 25.0155136536

Para más información consultar:
https://github.com/Angi-Reynoso/Datos_Masivos/tree/Unidad_4/Unit_4