

TECNOLÓGICO NACIONAL DE MÉXICO.

INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN.

SEMESTRE AGOSTO – DICIEMBRE 2018.

ING. Tecnologías de la información y comunicaciones.

**Datos masivos.**

**Proyecto.**

Unidad 4.

Alumna.

**Vidal Cadena Ana Crisel #14211197.**

Profesor.

**José Christian Romero.**

Tijuana, B.C; a 30 de noviembre de 2018.



# **Objetivo.**

Se llevará a cabo la comparación del rendimiento de los siguientes algoritmos de machine learning.

* SVM.
* Árboles de decisión.
* Logistic Regression.
* Multilayer Perceptron.

Con el siguiente data set:<https://archive.ics.uci.edu/ml/datasets/Bank+Marketing>

Para observar que tan óptimos son con una cantidad de datos superior a los 40’000 y con 17 columnas al finalizar se optara por un solo algoritmo de machine learning y se justificará la selección.

# **Limpieza de Datos.**

Para que exista una coherencia con los datos que se van a manejar va a ser necesario que se lleve a cabo una depuración de todo aquello que pueda simplificar la corrida de los diferentes códigos y no entorpezca el procedimiento de cada uno de ellos, para esto se realizará lo siguiente.

|  |
| --- |
| // ~~~~~~Limpieza de datos~~~~~ //Importación para la creacion de una sesion spark import org.apache.spark.sql.SparkSession //Lo siguiente se aplica para evitar errores ya que nos va marcando cada uno // y los va categorizando. import org.apache.log4j.\_ Logger.getLogger("org").setLevel(Level.ERROR) //importación de las librerías a utilizar import org.apache.spark.sql.functions.\_ import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType};  // Se crea la variable Spark para crear una sesion y permitira //dar lectura al archivo csv . val spark = SparkSession.builder().getOrCreate()  //division de dataset en columnas gracias al la delimitacion del ";" a partir de la lectura del csv a utilizar. val df = spark.read.option("header", true).option("inferSchema", "true").option("delimiter", ";").csv("bank-full.csv")  //se pivotea la insercion del dataset para que valide los "yes" como 1 ó 0 en caso contrario val df2 = when($"y".contains("yes"), 1.0).otherwise(0.0) //se vuelve a pivotear los resultados insertándose en la columna "y" nuevamente val df3 = df.withColumn("y", df2)  // pruebas para análisis y visualización del dataset // df3.printSchema // df3.show(3) // df3.columns  //creación de variables donde se redefinen las columnas en el dataset df3 //para mostrarlo con una sola variable // val colnames = df3.columns // //para cuestiones de estética nos arroja el primer renglón con sus respectivos encabezados // val firstrow = df3.head(1)(0) // println("\n") // println("Example data row") // for(ind <- Range(1, colnames.length)){ // println(colnames(ind)) // println(firstrow(ind)) // println("\n") // }  //importaciones que nos ayudaran para transformar los valores en datos binarios //para la ejecución. import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer, VectorIndexer, OneHotEncoderEstimator} import org.apache.spark.ml.linalg.Vectors  val jobIndexer = new StringIndexer().setInputCol("job").setOutputCol("jobIndex").fit(df3) val maritalIndexer = new StringIndexer().setInputCol("marital").setOutputCol("maritalIndex").fit(df3) val eduIndexer = new StringIndexer().setInputCol("education").setOutputCol("educationIndex").fit(df3) val defaultIndexer = new StringIndexer().setInputCol("default").setOutputCol("defaultIndex").fit(df3) val housingIndexer = new StringIndexer().setInputCol("housing").setOutputCol("housingIndex").fit(df3) val loanIndexer = new StringIndexer().setInputCol("loan").setOutputCol("loanIndex").fit(df3) val contactIndexer = new StringIndexer().setInputCol("contact").setOutputCol("contactIndex").fit(df3) val monthIndexer = new StringIndexer().setInputCol("month").setOutputCol("monthIndex").fit(df3)  val jobIndexed = jobIndexer.transform(df3) val maritalIndexed = maritalIndexer.transform(df3) val eduIndexed = eduIndexer.transform(df3) val defaultIndexed = defaultIndexer.transform(df3) val housingIndexed = housingIndexer.transform(df3) val loanIndexed = loanIndexer.transform(df3) val contactIndexed = contactIndexer.transform(df3) val monthIndexed = monthIndexer.transform(df3)  val Encoder = new OneHotEncoderEstimator().setInputCols(Array("jobIndex", "maritalIndex", "educationIndex", "defaultIndex", "housingIndex", "loanIndex", "contactIndex", "monthIndex")).setOutputCols(Array("jobVec", "maritalVec", "educationVec", "defaultVec", "housingVec", "loanVec", "contactVec", "monthVec"))  val assembler = (new VectorAssembler().setInputCols(Array("age","duration", "balance","day","campaign", "previous", "jobVec", "maritalVec", "educationVec", "defaultVec", "housingVec", "loanVec", "contactVec", "monthVec")).setOutputCol("features"))  val Array(training, test) = df3.randomSplit(Array(0.7, 0.3), seed = 12345) |

# **Logistic Regression.**

La regresión logística es un algoritmo de **clasificación** utilizado para asignar observaciones a un conjunto discreto de clases. A diferencia de la regresión lineal que genera valores numéricos continuos, la regresión logística transforma su salida utilizando la función sigmoide logística para devolver un valor de probabilidad que luego puede asignarse a dos o más clases discretas.

|  |
| --- |
| /////////// ~~~~~~~ logistic Regression ~~~~~~~~ //////////////// //Importacion de librerias a utilizar para la corrida del codigo llamando funciones //y la libreria de la regresion logistica.. import org.apache.spark.sql.expressions.\_ import org.apache.spark.sql.functions.\_ import org.apache.spark.ml.classification.LogisticRegression import org.apache.spark.ml.Pipeline  //Se declara la variable log\_Re para utilizar la regresion logistica en la columna "Y" //con un maximo de 10 iteraciones. val log\_Re = new LogisticRegression().setLabelCol("y").setMaxIter(10)  //Si le damos un parametro para el setMaxIter y el setRegParam no nos arroja una matriz de confusion correcta //val lr = new LogisticRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)  //utilizamos el pipeline(embudo) para tomar en un arreglo los indexer creados val pipelinelog\_Re = new Pipeline().setStages(Array(jobIndexer,maritalIndexer, eduIndexer, defaultIndexer, housingIndexer, loanIndexer, contactIndexer, monthIndexer, Encoder, assembler,log\_Re))  //se realiza el modelo de entrenamiento y se transforman los datos para arrojarlos como resultados de pruebas val modellog\_Re = pipelinelog\_Re.fit(training) val resultslog\_Re = modellog\_Re.transform(test)  //Variable de prediccion y mediciones. val predictionAndLabelslog\_Re = resultslog\_Re.select($"prediction",$"y").as[(Double, Double)].rdd  import org.apache.spark.mllib.evaluation.MulticlassMetrics val metricsLog\_Re = new MulticlassMetrics(predictionAndLabelslog\_Re)  //Matriz de confusion donde nos indicara cuales fueron los falsos positivos y falsos negativos. println("Matriz de confusion: ") println(metricsLog\_Re.confusionMatrix)  //Resultado de la Exactitud println("Exactitud")  println(metricsLog\_Re.accuracy) |

# **Support Vector Machine.**

Las máquinas de soporte vectorial, máquinas de vectores de soporte o máquinas de vector soporte (Support Vector Machines, SVMs) son un conjunto de [algoritmos](https://es.wikipedia.org/wiki/Algoritmo) de [aprendizaje supervisado](https://es.wikipedia.org/wiki/Aprendizaje_supervisado) desarrollados por [Vladimir Vapnik](https://es.wikipedia.org/w/index.php?title=Vladimir_Vapnik&action=edit&redlink=1) y su equipo en los laboratorios AT&T.

Estos métodos están propiamente relacionados con problemas de [clasificación](https://es.wikipedia.org/wiki/Clasificaci%C3%B3n_estad%C3%ADstica) y [regresión](https://es.wikipedia.org/wiki/An%C3%A1lisis_de_la_regresi%C3%B3n). Dado un conjunto de ejemplos de entrenamiento (de muestras) podemos etiquetar las clases y entrenar una SVM para construir un modelo que prediga la clase de una nueva muestra. Intuitivamente, una SVM es un modelo que representa a los puntos de muestra en el espacio, separando las clases a 2 espacios lo más amplios posibles mediante un [hiperplano](https://es.wikipedia.org/wiki/Hiperplano) de separación definido como el vector entre los 2 puntos, de las 2 clases, más cercanos al que se llama vector soporte. Cuando las nuevas muestras se ponen en correspondencia con dicho modelo, en función de los espacios a los que pertenezcan, pueden ser clasificadas a una o la otra clase.

Más formalmente, una SVM construye un [hiperplano](https://es.wikipedia.org/wiki/Hiperplano) o conjunto de hiperplanos en un espacio de [dimensionalidad](https://es.wikipedia.org/wiki/Dimensi%C3%B3n) muy alta (o incluso infinita) que puede ser utilizado en problemas de **clasificación** o regresión. Una buena separación entre las clases permitirá una clasificación correcta.

|  |
| --- |
| //////////////// ~~~~~~~ SVM ~~~~~~~~ ////////////////////  import org.apache.spark.ml.classification.LinearSVC  //Debido a que ya hemos precargado el dataset con el que se va a trabajar //Se evita la redundancia de informacion y la siguiente linea de codigo //val training = spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt") val l\_SVM = new LinearSVC().setLabelCol("y").setMaxIter(10)  val pipelinel\_SVM = new Pipeline().setStages(Array(jobIndexer,maritalIndexer, eduIndexer, defaultIndexer, housingIndexer, loanIndexer, contactIndexer, monthIndexer, Encoder, assembler, l\_SVM))  val modell\_SVM = pipelinel\_SVM.fit(training) val resultsl\_SVM = modell\_SVM.transform(test)  val predictionAndLabelsl\_SVM = resultsl\_SVM.select($"prediction",$"y").as[(Double, Double)].rdd  val metricsl\_SVM = new MulticlassMetrics(predictionAndLabelsl\_SVM)  //println("Confusion matrix:") println("Matriz de confusion: ") println(metricsl\_SVM.confusionMatrix) println("Exactitud: ") println(metricsl\_SVM.accuracy) |

# 

# **Multilayer Perceptron.**

Puede verse como un clasificador de regresión logística donde la entrada se transforma primero utilizando una transformación no lineal aprendida \ Phi. Esta transformación proyecta los datos de entrada en un espacio donde se puede separar linealmente. Esta capa intermedia se conoce como una capa oculta. Una sola capa oculta es suficiente para hacer de los MLP un aproximador universal. Sin embargo, más adelante veremos que hay beneficios sustanciales en el uso de muchas de estas capas ocultas, es decir, la premisa misma del aprendizaje profundo. Vea estas notas del curso para una introducción a los MLP, el algoritmo de propagación hacia atrás y cómo entrenar a los MLP.

|  |
| --- |
| ///////////// ~~~~~~~ MultilayerPerceptron ~~~~~~~~ ////////////  import org.apache.spark.sql.SparkSession import org.apache.spark.sql.functions.\_ import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType};  import org.apache.log4j.\_ Logger.getLogger("org").setLevel(Level.ERROR)  val spark = SparkSession.builder().getOrCreate()  val df = spark.read.option("header", true).option("inferSchema", "true").option("delimiter", ";").csv("bank-full.csv")  //Ver Estructura de los datos. // df.printSchema // df.show(3) // df.head // df.columns  //Eliminar posibles columnas con datos nulos. val data = df.na.drop()  //Reemplazar los datos de tipo String a numéricos. import org.apache.spark.ml.feature.StringIndexer  val indexer\_age = new StringIndexer().setInputCol("age").setOutputCol("InAge") val indexer\_job = new StringIndexer().setInputCol("job").setOutputCol("InJob") val indexer\_marital = new StringIndexer().setInputCol("marital").setOutputCol("InMarital") val indexer\_education = new StringIndexer().setInputCol("education").setOutputCol("InEducation") val indexer\_default = new StringIndexer().setInputCol("default").setOutputCol("InDefault") val indexer\_balance = new StringIndexer().setInputCol("balance").setOutputCol("InBalance") val indexer\_housing = new StringIndexer().setInputCol("housing").setOutputCol("InHousing") val indexer\_loan = new StringIndexer().setInputCol("loan").setOutputCol("InLoan") val indexer\_contact = new StringIndexer().setInputCol("contact").setOutputCol("InContact") val indexer\_day = new StringIndexer().setInputCol("day").setOutputCol("InDay") val indexer\_month = new StringIndexer().setInputCol("month").setOutputCol("InMonth") val indexer\_duration = new StringIndexer().setInputCol("duration").setOutputCol("InDuration") val indexer\_campaign = new StringIndexer().setInputCol("campaign").setOutputCol("InCampaign") val indexer\_pdays = new StringIndexer().setInputCol("pdays").setOutputCol("InPdays") val indexer\_previous = new StringIndexer().setInputCol("previous").setOutputCol("InPrevious") val indexer\_poutcome = new StringIndexer().setInputCol("poutcome").setOutputCol("InPoutcome") val indexer\_y = new StringIndexer().setInputCol("y").setOutputCol("label")  val indexed = indexer\_job.fit(data).transform(data) val data2 = indexed val indexed = indexer\_marital.fit(data2).transform(data2) val data3 = indexed val indexed = indexer\_education.fit(data3).transform(data3) val data4 = indexed val indexed = indexer\_default.fit(data4).transform(data4) val data5 = indexed val indexed = indexer\_housing.fit(data5).transform(data5) val data6 = indexed val indexed = indexer\_loan.fit(data6).transform(data6) val data7 = indexed val indexed = indexer\_contact.fit(data7).transform(data7) val data8 = indexed val indexed = indexer\_month.fit(data8).transform(data8) val data9 = indexed val indexed = indexer\_poutcome.fit(data9).transform(data9) val data10 = indexed val indexed = indexer\_y.fit(data10).transform(data10) val data11 = indexed val indexed = indexer\_age.fit(data11).transform(data11) val data12 = indexed val indexed = indexer\_balance.fit(data12).transform(data12) val data13 = indexed val indexed = indexer\_day.fit(data13).transform(data13) val data14 = indexed val indexed = indexer\_duration.fit(data14).transform(data14) val data15 = indexed val indexed = indexer\_campaign.fit(data15).transform(data15) val data16 = indexed val indexed = indexer\_pdays.fit(data16).transform(data16) val data17 = indexed val indexed = indexer\_previous.fit(data17).transform(data17)  val df2 = indexed df2.show()  val df3 = df2.select($"label", $"InAge", $"InJob", $"InMarital", $"InEducation", $"InDefault", $"InBalance", $"InHousing", $"InLoan", $"InContact", $"InDay", $"InMonth", $"InDuration", $"InCampaign", $"InPdays", $"InPrevious", $"InPoutcome")  ///////////////////hasta obtenemos la tabla con datos numericos //////////////////////// import org.apache.spark.ml.feature.VectorAssembler val assembler = new VectorAssembler().setInputCols(Array("InAge", "InJob", "InMarital", "InEducation", "InDefault", "InBalance", "InHousing", "InLoan", "InContact", "InDay", "InMonth", "InDuration", "InCampaign", "InPdays", "InPrevious", "InPoutcome")).setOutputCol("features")  val output = assembler.transform(df3)  output.select("label", "features").show(false)   val mlp = output.select("label", "features") mlp.show(false) mlp.printSchema   import org.apache.spark.ml.classification.MultilayerPerceptronClassifier import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator import org.apache.spark.mllib.util.MLUtils import org.apache.spark.sql.Row  // Load training data //val data = MLUtils.loadLibSVMFile(sc, "bank-full.csv").toDF() //val mlpc = new MLUtils().setLabelCol("y").setMaxIter(10).setRegParam(0.1)   // Load training data //val data = output //data.show(2) // Load training data // Split the data into train and test val splits = mlp.randomSplit(Array(0.6, 0.4), seed = 1234L) val train = splits(0) val test = splits(1) // specify layers for the neural network: // input layer of size 4 (features), two intermediate of size 5 and 4 and output of size 3 (classes) val layers = Array[Int](16, 2, 2, 5) // create the trainer and set its parameters val trainer = new MultilayerPerceptronClassifier().setLayers(layers).setBlockSize(128).setSeed(1234L).setMaxIter(100) // train the model val model = trainer.fit(train)  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)}") |

# 

# **Árbol de decisiones.**

Los modelos de árbol, donde la variable de destino puede tomar un conjunto finito de valores se denominan árboles de clasificación. En estas estructuras de árbol, las hojas representan etiquetas de clase y las ramas representan las conjunciones de características que conducen a esas etiquetas de clase. Los árboles de decisión, donde la variable de destino puede tomar valores continuos (por lo general números reales) se llaman árboles de regresión.  
  
En análisis de decisión, un árbol de decisión se puede utilizar para representar visualmente y de forma explícita decisiones y toma de decisiones. En minería de datos, un árbol de decisión describe datos, pero no las decisiones; más bien el árbol de clasificación resultante puede ser un usado como entrada para la toma de decisiones. Esta página se ocupa de los árboles de decisión en la minería de datos.

|  |
| --- |
| //////////////~~~~ Arbol de Desiciones ~~~~ /////////////////  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 label\_Indexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(df) // Automatically identify categorical features, and index them. val feature\_Indexer = new VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(4)// features with > 4 distinct values are treated as continuous. .fit(data) // Split the data into training and test sets (30% held out for testing). val Array(trainingData, testData) = df.randomSplit(Array(0.6, 0.4)) // Train a DecisionTree model. val d4 = new DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures") // Convert indexed labels back to original labels. val label\_Converter = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(label\_Indexer.labels) // Chain indexers and tree in a Pipeline. val pipeline = new Pipeline().setStages(Array(label\_Indexer, feature\_Indexer, d4, label\_Converter)) // Train model. This also runs the indexers. val model = pipeline.fit(trainingData) // Make predictions. val predictions = model.transform(testData) // Select example rows to display. predictions.select("predictedLabel", "label", "features").show(5) // Select (prediction, true label) and compute test error. val evaluator = new MulticlassClassificationEvaluator().setLabelCol("indexedLabel").setPredictionCol("prediction").setMetricName("accuracy") val accuracy = evaluator.evaluate(predictions) println(s"Test Error = ${(1.0 - accuracy)}")  val 3Model = model.stages(2).asInstanceOf[DecisionTreeClassificationModel] println(s"Learned classification tree model:\n ${3Model.toDebugString}") |

**El mejor hasta el momento es Regresión logística. por la exactitud que ha manejado con los datos**