



TECNOLÓGICO NACIONAL DE MÉXICO INSTITUTO

TECNOLÓGICO DE TIJUANA  
SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

SEMESTRE FEBRERO-JUNIO 2022

**CARRERA**

Ingeniería en informática e Ingeniería en Sistemas

Computacionales

**MATERIA**

Datos masivos

**TÍTULO**

Proyecto final.

**Integrantes:**

Munguía Silva Edgar Geovanny #17212344

Pérez López Alicia Guadalupe ##18210514

**NOMBRE DEL MAESTRO**

Jose Christian Romero Hernandez

Tijuana, Baja California, 01 de Junio del 2022



# Índice

Introducción .....	3
Marco teórico de los algoritmos .....	4-7
Implementación .....	8
Algoritmos .....	9-21
Resultados .....	22-29
Conclusiones.....	30
Referencias .....	31



## Introducción

El presente proyecto consta de 4 algoritmos los cuales vamos a comparar para determinar cuál es el más eficiente y conveniente de utilizar en diversas situaciones.

Culminamos la materia de Datos masivos la cual consta de 4 unidades, las cuales fueron muy interesantes y nos dejaron mucho conocimiento respecto al tema del big data, se aprendió bastante y lo que más destacamos, son los temas de machine learning, ya que como sabemos, ML es uno de los temas que más suenan en nuestra sociedad, entonces es necesario, estar a la vanguardia para implementar dichas tecnologías y no quedarnos atrás tecnológicamente hablando, nosotros, como futuros científicos de datos, debemos conocer mejor que nadie, este tipo de algoritmos, a continuación, vamos a presentar dichos algoritmos , los definiremos, los mostraremos, los compararemos para determinar, cuál es el mejor, en términos de rapidez y eficiencia.

## Marco teórico de los algoritmos.

**Support Vector Machine (SVM):** es un algoritmo de aprendizaje supervisado que se utiliza en muchos problemas de clasificación y regresión, incluidas aplicaciones médicas de procesamiento de señales, procesamiento del lenguaje natural y reconocimiento de imágenes y voz.

El objetivo del algoritmo SVM es encontrar un hiperplano que separe de la mejor forma posible dos clases diferentes de puntos de datos. "De la mejor forma posible" implica el hiperplano con el margen más amplio entre las dos clases, representado por los signos más y menos en la siguiente figura. El margen se define como la anchura máxima de la región paralela al hiperplano que no tiene puntos de datos interiores. El algoritmo solo puede encontrar este hiperplano en problemas que permiten separación lineal; en la mayoría de los problemas prácticos, el algoritmo maximiza el margen flexible permitiendo un pequeño número de clasificaciones erróneas.

### Referencia:

MATLAB. (2015, 2 marzo). *Support Vector Machine (SVM)*. MATLAB & Simulink. Recuperado 5 de junio de 2022, de <https://es.mathworks.com/discovery/support-vector-machine.html>



### Decision Tree:

Los árboles de decisión son algoritmos estadísticos o técnicas de machine learning que nos permiten la construcción de modelos predictivos de analítica de datos para el Big Data basados en su clasificación según ciertas características o propiedades, o en la regresión mediante la relación entre distintas variables para predecir el valor de otra.

En los modelos de clasificación queremos predecir el valor de una variable mediante la clasificación de la información en función de otras variables (tipo, pertenencia a un grupo...). Por ejemplo, queremos pronosticar qué personas comprarán un determinado producto, clasificando entre clientes y no clientes, o qué marcas de portátiles comprará cada persona mediante la clasificación entre las distintas marcas. Los valores a predecir son predefinidos, es decir, los resultados están definidos en un conjunto de posibles valores.


En los modelos de regresión se intenta predecir el valor de una variable en función de otras variables que son independientes entre sí. Por ejemplo, queremos predecir el precio de venta del terreno en función de variables como su localización, superficie, distancia a la playa, etc. El posible resultado no forma parte de un conjunto predefinido, sino que puede tomar cualquier posible valor.

**Referencia:** *Unir, V. (2021, 19 octubre). Árboles de decisión: en qué consisten y aplicación en Big Data. UNIR. Recuperado 5 de junio de 2022, de <https://www.unir.net/ingenieria/revista/arboles-de-decision/>*



**Logistic Regression:** La Regresión Logística es un método estadístico para predecir clases binarias. El resultado o variable objetivo es de naturaleza dicotómica. Dicotómica significa que solo hay dos clases posibles. Por ejemplo, se puede utilizar para problemas de detección de cáncer o calcular la probabilidad de que ocurra un evento.

La Regresión Logística es uno de los algoritmos de Machine Learning más simples y más utilizados para la clasificación de dos clases. Es fácil de implementar y se puede usar como línea de base para cualquier problema de clasificación binaria. La Regresión Logística describe y estima la relación entre una variable binaria dependiente y las variables independientes.

**Referencia:** Gonzalez, L. (2020, 21 agosto). Regresión Logística - Teoría.  Aprende IA. Recuperado 5 de junio de 2022, de <https://aprendeia.com/regresion-logistica-multiple-machine-learning-teoria/#:%7E:text=La%20Regresi%C3%B3n%20Log%C3%ADstica%20es%20uno,cualquier%20problema%20de%20clasificaci%C3%B3n%20binaria.>

**Multilayer perceptron:**

El perceptrón multicapa (MLP) es un complemento de la red neuronal de avance. Consta de tres tipos de capas: la capa de entrada, la capa de salida y la capa oculta. La capa de entrada recibe la señal de entrada para ser procesada. La capa de salida realiza la tarea requerida, como la predicción y la clasificación. Un número arbitrario de capas ocultas que se colocan entre la capa de entrada y la de salida son el verdadero motor computacional del MLP. De manera similar a una red de avance en un MLP, los datos fluyen en la dirección de avance desde la capa de entrada a la de salida. Las neuronas en el MLP se entrenan con el algoritmo de aprendizaje de retropropagación. Los MLP están diseñados para aproximar cualquier función continua y pueden resolver problemas que no son linealmente separables. Los principales casos de uso de MLP son la clasificación, el reconocimiento, la predicción y la aproximación de patrones.

**Referencia:**

*Sciencedirect. (2014a, abril 1). Multilayer Perceptron. Recuperado 5 de junio de 2022, de <https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron>*



### Implementación.

Para llevar a cabo la implementación de los algoritmos anteriormente mencionados, hicimos uso del lenguaje de programación spark/scala, ya que se trata de una herramienta muy poderosa para los tópicos de Big Data (Datos masivos) y además, es relativamente sencilla de utilizar, en realidad, la comparamos más o menos con python, ya que no se trata de un lenguaje tan complejo en comparación con otros. Las posibilidades que nos ofrece, son infinitas, siendo para nosotros, una de las herramientas top para trabajar, con datos masivos.





```
scala> //In order to avoid error we need to create 2 new columns label and features

scala> //Here we create them using StringIndexer and VectorAssembler

scala> val assembler = new VectorAssembler().setInputCols(Array("age", "balance", "day", "duration", "previous")).setOutputCol("features")
assembler: org.apache.spark.ml.feature.VectorAssembler = vecAssembler_0a8c20955563

scala> val features = assembler.transform(indexed)
features: org.apache.spark.sql.DataFrame = [age: int, job: string ... 17 more fields]

scala>

scala> val Array(training, test) = features.randomSplit(Array(0.7, 0.3), seed = 12345)
training: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [age: int, job: string ... 17 more fields]
test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [age: int, job: string ... 17 more fields]

scala>

scala> val lsvcModel = lsvc.fit(training)
22/06/07 21:19:46 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
22/06/07 21:19:46 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
lsvcModel: org.apache.spark.ml.classification.LinearSVCModel = linearsvc_54c3de7a3daa

scala>

scala> val results = lsvcModel.transform(test)
results: org.apache.spark.sql.DataFrame = [age: int, job: string ... 19 more fields]

scala> val predictionAndLabels = results.select($"prediction", $"label").as[(Double, Double)].rdd
predictionAndLabels: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[58] at rdd at <console>:32

scala> val metrics = new MulticlassMetrics(predictionAndLabels)
metrics: org.apache.spark.mllib.evaluation.MulticlassMetrics = org.apache.spark.mllib.evaluation.MulticlassMetrics@7260f47f
```

```
scala> println("Confusion matrix:")
Confusion matrix:

scala> println(metrics.confusionMatrix)
11528.0  0.0
2213.0   0.0

scala>

scala> metrics.accuracy
res3: Double = 0.8389491303398589

scala>

scala> val error = 1 - metrics.accuracy
error: Double = 0.16105086966014115
```





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



```
scala> val spark = SparkSession.builder().getOrCreate()
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@6a577564

scala>

scala> val data = spark.read.option("header", "true").option("inferSchema", "true").option("delimiter", ";").csv("C:/Users/x/Documents/projet/bank-full.csv")
data: org.apache.spark.sql.DataFrame = [age: int, job: string ... 15 more fields]
```

```
scala> //Transform the categorical data to numeric, merges the new data with the previous values

scala> //this time with the numeric data and renames the loan column as label to use it in the execution

scala> val labelIndexer = new StringIndexer().setInputCol("loan").setOutputCol("indexedLabel").fit(data)
labelIndexer: org.apache.spark.ml.feature.StringIndexerModel = strIdx_1c1d0bd38d03

scala> val indexed = labelIndexer.transform(data).drop("loan").withColumnRenamed("indexedLabel", "label")
indexed: org.apache.spark.sql.DataFrame = [age: int, job: string ... 15 more fields]

scala> val assembler = (new VectorAssembler()).setInputCols(Array("age", "balance", "day", "duration", "previous")).setOutputCol("features")
assembler: org.apache.spark.ml.feature.VectorAssembler = vecAssembler_bdccc90e5fe7

scala> val features = assembler.transform(indexed)
features: org.apache.spark.sql.DataFrame = [age: int, job: string ... 16 more fields]

scala> val filter = features.withColumnRenamed("loan", "label")
filter: org.apache.spark.sql.DataFrame = [age: int, job: string ... 16 more fields]
```

```
scala> val finalData = filter.select("label", "features")
finalData: org.apache.spark.sql.DataFrame = [label: double, features: vector]

scala>

scala> // Index labels, adding metadata to the label column.

scala> // Fit on whole dataset to include all labels into the index.

scala> val labelIndexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(finalData)
labelIndexer: org.apache.spark.ml.feature.StringIndexerModel = strIdx_66b61990098d

scala> // Automatically identify categorical features and then index them.

scala> val featureIndexer = new VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(4).fit(finalData)
featureIndexer: org.apache.spark.ml.feature.VectorIndexerModel = vecIdx_c1fcfdad2d7b

scala> // Split the data into training and test sets (30% held out for testing).

scala> val Array(trainingData, testData) = finalData.randomSplit(Array(0.7, 0.3))
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label: double, features: vector]
testData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label: double, features: vector]

scala>

scala> // Train a DecisionTree model.

scala> val dt = new DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures")
dt: org.apache.spark.ml.classification.DecisionTreeClassifier = dtc_b11fd1116bed
```



```
scala> // Convert indexed labels back to original labels.

scala> val labelConverter = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(labelIndexer.labels)
labelConverter: org.apache.spark.ml.feature.IndexToString = idxToStr_783253135231

scala>

scala> // Chain indexers and tree in a Pipeline.

scala> val pipeline = new Pipeline().setStages(Array(labelIndexer, featureIndexer, dt, labelConverter))
pipeline: org.apache.spark.ml.Pipeline = pipeline_dd11ff66c29f

scala>

scala> // Train the model, this also runs the indexers.

scala> val model = pipeline.fit(trainingData)
model: org.apache.spark.ml.PipelineModel = pipeline_dd11ff66c29f

scala>

scala> // Make the predictions.

scala> val predictions = model.transform(testData)
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 6 more fields]
```

```
scala> // Select example rows to display. In this case there was only 5 rows to show.

scala> predictions.select("predictedLabel", "label", "features").show(5)
+-----+-----+-----+
|predictedLabel|label|features|
+-----+-----+-----+
|0.0|0.0|[18.0,3.0,25.0,13...|
|0.0|0.0|[18.0,35.0,21.0,1...|
|0.0|0.0|[18.0,108.0,10.0,...|
|0.0|0.0|[18.0,156.0,4.0,2...|
|0.0|0.0|[19.0,4.0,3.0,114...|
+-----+-----+-----+
only showing top 5 rows

scala>

scala> // Select (prediction, true label)

scala> val evaluator = new MulticlassClassificationEvaluator().setLabelCol("indexedLabel").setPredictionCol("prediction").setMetricName("accuracy")
evaluator: org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator = mcEval_a2575191c2b2

scala> // Compute the test error.

scala> val accuracy = evaluator.evaluate(predictions)
accuracy: Double = 0.8383978911913305

scala> println(s"Test Error = ${(1.0 - accuracy)}")
Test Error = 0.16160210880866954
```

```
scala> // Show by stages the classification of the tree model

scala> val treeModel = model.stages(2).asInstanceOf[DecisionTreeClassificationModel]
treeModel: org.apache.spark.ml.classification.DecisionTreeClassificationModel = DecisionTreeClassificationModel (uid=dtc_b11fd1116bed) of depth 5 with 29 nodes
```

```
scala> println(s"Learned classification tree model:\n ${treeModel.toDebugString}")
Learned classification tree model:
DecisionTreeClassificationModel (uid=dtc_b11fd1116bed) of depth 5 with 29 nodes
  If (feature 1 <= -0.5)
    If (feature 1 <= -312.5)
      If (feature 3 <= 904.5)
        Predict: 0.0
      Else (feature 3 > 904.5)
        If (feature 0 <= 54.5)
          If (feature 2 <= 12.5)
            Predict: 0.0
          Else (feature 2 > 12.5)
            Predict: 1.0
        Else (feature 0 > 54.5)
          Predict: 0.0
    Else (feature 1 > -312.5)
      If (feature 2 <= 1.5)
        If (feature 4 <= 0.5)
          If (feature 3 <= 66.5)
            Predict: 0.0
          Else (feature 3 > 66.5)
            Predict: 1.0
        Else (feature 4 > 0.5)
          Predict: 0.0
      Else (feature 2 > 1.5)
        If (feature 2 <= 22.5)
          Predict: 0.0
        Else (feature 2 > 22.5)
          If (feature 0 <= 28.5)
            Predict: 1.0
          Else (feature 0 > 28.5)
            Predict: 0.0
      Else (feature 1 > -0.5)
```

```
  If (feature 1 <= 930.5)
    Predict: 0.0
  Else (feature 1 > 930.5)
    If (feature 1 <= 2501.5)
      If (feature 4 <= 17.5)
        Predict: 0.0
      Else (feature 4 > 17.5)
        If (feature 0 <= 56.5)
          Predict: 1.0
        Else (feature 0 > 56.5)
          Predict: 0.0
    Else (feature 1 > 2501.5)
      Predict: 0.0
```



```
scala> //Get the total time of the program execution

scala> val totalTime = System.currentTimeMillis - start
totalTime: Long = 77556

scala> println("Elapsed time: %1d ms".format(totalTime))
Elapsed time: 77556 ms

scala>

scala> //Get the total of MB used

scala> val runtime = Runtime.getRuntime
runtime: Runtime = java.lang.Runtime@39aef4b8

scala> val mb = 1024*1024
mb: Int = 1048576

scala> println("Used memory: " + (runtime.totalMemory - runtime.freeMemory) / mb + " MB")
Used memory: 309 MB
```

## Logistic Regression.

```
// Starting timer
val timerstar = System.currentTimeMillis()

//Import necessary libraries

import
org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.sql.Session
import org.apache.spark.ml.feature.{VectorAssembler,
StringIndexer, VectorIndexer, OneHotEncoder}
import org.apache.spark.ml.linalg.Vectors
import
org.apache.spark.mllib.evaluation.MulticlassMetrics

// Create spark session
val spark = SparkSession.builder().getOrCreate()

// Load dataframe
val data =
spark.read.option("header","true").option("inferSchema",
"true").option("delimiter",";").format("csv").load("bank-
full.csv")
```



```
// Adding index labels
val labelIndexer = new
StringIndexer().setInputCol("y").setOutputCol("indexedLabel").fit(data)
val indexed =
labelIndexer.transform(data).drop("y").withColumnRenamed(
"indexedLabel", "label")

// Creating a vector to add the info into the array
val vectorFeatures = (new
VectorAssembler().setInputCols(Array("balance", "day", "duration", "pdays", "previous")).setOutputCol("features"))

// Transforming the previous vector into a new index variable
val features = vectorFeatures.transform(indexed)

// Renaming label of features
val featuresLabel = features.withColumnRenamed("y", "label")

// A new variable is created selecting some columns
val dataIndexed =
featuresLabel.select("label", "features")

// Use randomSplit to create 70/30 split test and train data
val Array(training, test) =
dataIndexed.randomSplit(Array(0.7, 0.3), seed = 12345)

// The Logistic Regression is created with the parameters sent
val logisticAlgorithm = new
LogisticRegression().setMaxIter(10).setRegParam(0.3).setE
```

```
lasticNetParam(0.8).setFamily("multinomial")

// The model is trained
val logisticModel = logisticAlgorithm.fit(training)

// Calculating precision of the test data
val results = logisticModel.transform(test)
val predictionAndLabels =
results.select($"prediction", $"label").as[(Double,
Double)].rdd
val metrics = new MulticlassMetrics(predictionAndLabels)

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

// Showing Accuracy and error
println("Accuracy: "+metrics.accuracy)
println(s"Test Error = ${1.0 - metrics.accuracy}")
// Stopping the timer and calculated the time that took to
run the algorithm
val timerstop = System.currentTimeMillis()
val duration = (timerstop - timerstar) / 1000

// Printing the time in seconds that took the algorithm
to run
println(duration)
```

### Multilayer perceptron.

```
//Start timer
val starttimer = System.currentTimeMillis()
```

```
// Import necessary libraries
import
org.apache.spark.ml.classification.MultilayerPerceptronCl
assifier
import
org.apache.spark.ml.evaluation.MulticlassClassificationEv
aluator
import org.apache.spark.sql.Session
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.sql.types.IntegerType

// Creating spark session
val spark =
SparkSession.builder.appName("MultilayerPerceptronClassif
ierExample").getOrCreate()

// Loading dataframe
val dataframeMP =
spark.read.option("header","true").option("inferSchema",
"true").option("delimiter",";").format("csv").load("bank-
full.csv")

// displaying the data
dataframeMP.columns
dataframeMP.printSchema()
dataframeMP.head(5)
dataframeMP.describe().show()

// Indexing labels
val labelIndexer = new
StringIndexer().setInputCol("y").setOutputCol("indexedLab
el").fit(dataframeMP)
val indexed =
labelIndexer.transform(dataframeMP).drop("y").withColumnR
```

```
enamed("indexedLabel", "label")
indexed.describe().show()

// Creating assemble vector
val assembler = new
VectorAssembler().setInputCols(Array("balance", "day", "dur
ation", "pdays", "previous")).setOutputCol("features")
val features = assembler.transform(indexed)

// The label columns are indexed and the data is
displayed
val labelIndexer = new
StringIndexer().setInputCol("label").setOutputCol("indexe
dLabel").fit(indexed)
println(s"Found labels:
${labelIndexer.labels.mkString("[", ", ", ", "]")}")
features.show()

// Data is divided into training and testing
val splits = features.randomSplit(Array(0.6, 0.4), seed =
1234L)
val train = splits(0)
val test = splits(1)

// The layers of the neural network are specified:
val layers = Array[Int](5, 4, 1, 2)

// Training parameters are set
val trainer = new
MultilayerPerceptronClassifier().setLayers(layers).setBlo
ckSize(128).setSeed(1234L).setMaxIter(100)

// Training the model
val model = trainer.fit(train)
```

```
// Calculating precision
val result = model.transform(test)
val predictionAndLabels = result.select("prediction",
"label")
predictionAndLabels.show
val evaluator = new
MulticlassClassificationEvaluator().setMetricName("accuracy")

// Printing the model accuracy
println(s"Test Accuracy =
${evaluator.evaluate(predictionAndLabels)}")
println(s"Test Error = ${1.0 -
evaluator.evaluate(predictionAndLabels)}")

// Stopping the timer
val endtimer = System.currentTimeMillis()
val duration = (endtimer - starttimer) / 1000

//Print the time that took the algorithm to run
println(duration)
```

## Resultados.

SVM

Corrida	Time	Memory	Error	Precisión
1	106310mls	279 MB	0.161050869 66014115	0.838949130 3398589
2	71875 ms	204 MB	0.161050869 66014115	0.838949130 3398589
3	66305 ms	217 MB	0.161050869 66014115	0.838949130 3398589
4	69009 ms	282 MB	0.161050869 66014115	0.838949130 3398589
5	39890 ms	410 MB	0.161050869 66014115	0.838949130 3398589
6	59618 ms	532 MB	0.161050869 66014115	0.838949130 3398589
7	63861 ms	261 MB	0.838949130 3398589	0.161050869 66014115
8	53880 ms	437 MB	0.161050869 66014115	0.838949130 3398589
9	55165 ms	339 MB	0.161050869 66014115	0.838949130 3398589
10	69009 ms	282 MB	0.161050869 66014115	0.838949130 3398589
11	60375 ms	406 MB	0.161050869 66014115	0.838949130 3398589
12	43956 ms	455 MB	0.161050869 66014115	0.838949130 3398589
13	68579 ms	377 MB	0.161050869 66014115	0.838949130 3398589
14	66839 ms	322 MB	0.161050869	0.838949130

			66014115	3398589
15	71875 ms	204 MB	0.161050869 66014115	0.838949130 3398589
16	48871 ms	337 MB	0.161050869 66014115	0.838949130 3398589
17	63076 ms	542 MB	0.161050869 66014115	0.838949130 3398589
18	80110 ms	480 MB	0.161050869 66014115	0.838949130 3398589
19	91839 ms	667 MB	0.161050869 66014115	0.838949130 3398589
20	63861 ms	261 MB	0.838949130 3398589	0.161050869 66014115
21	71037 ms	565 MB	0.161050869 66014115	0.838949130 3398589
22	61093 ms	482 MB	0.160680391 29799971	0.839319608 7020003
23	100936 ms	677 MB	0.161050869 66014115	0.838949130 3398589
24	66294 ms	705 MB	0.161050869 66014115	0.838949130 3398589
25	63861 ms	261 MB	0.838949130 3398589	0.161050869 66014115
26	74832 ms	362 MB	0.161050869 66014115	0.838949130 3398589
27	68479 ms	357 MB	0.161050869 66014115	0.838949130 3398589
28	48971 ms	437 MB	0.161050869 66014116	0.838949130 3398589

29	106311 ms	229 MB	0.161050869 66014115	0.838949130 3398589
30	67815 ms	409 MB	0.161050869 66014115	0.838949130 3398589
Promedio	68131,06667	393,8666667	0	0

## DTC

Corrida	Time	Memory	Error	Precisión
1	77556 ms	309 MB	0.1616021088 0866954	0.838397891 1913305

2	90001 ms	442 MB	0.157969603 95298343	0.842030396 0470166
3	57335 ms	472 MB	0.160767934 34454462	0.839232065 6554554
4	54572 ms	247 MB	0.162184249 62852895	0.837815750 371471
5	55400 ms	250 MB	0.1600088118 6664706	0.8399911881 333529
6	50709 ms	392 MB	0.160516880 78876603	0.839483119 211234
7	54623 ms	261 MB	0.159566572 6584533	0.84043342 73415467
8	41382 ms	436 MB	0.160264900 66225163	0.839735099 3377484
9	32821 ms	394 MB	0.1570915619 3895868	0.84290843 80610413
10	60266 ms	289 MB	0.157494407	0.84250559



			15883667	28411633
11	47868 ms	331 MB	0.164868685 36567223	0.835131314 6343278
12	61093 ms	482 MB	0.160680391 29799971	0.839319608 7020003
13	88445 ms	571 MB	0.161050869 66014115	0.838949130 3398589
14	44382 ms	396 MB	0.160264900 76225163	0.839735099 3477484
15	51583 ms	496 MB	0.158853976 81708824	0.841146023 1829118
16	50709 ms	392 MB	0.160516880 78876603	0.839483119 211234
17	71343 ms	77 MB	0.160988643 9545758	0.839011356 0454242
18	41282 ms	434 MB	0.160264900 66224163	0.839735099 3377484
19	77849 ms	496 MB	0.167152009 31857888	0.832847990 6814211
20	81975 ms	437 MB	0.1608515671 2004733	0.839148432 8799527
21	95823 ms	529 MB	0.1615961199 2945322	0.83840388 00705468
22	54629 ms	261 MB	0.159566572 6584533	0.84043342 73415467
23	60457 ms	542 MB	0.1613715470 6361545	0.83862845 29363845
24	67120 ms	614 MB	0.161364994 02628435	0.83863500 59737157
25	74589 ms	658 MB	0.161649944	0.83835005 57413601

			25863988	
26	77556 ms	309 MB	0.1616021088 0866954	0.838397891 1913305
27	37849 ms	532 MB	0.1607341078 961001	0.839265892 1038999
28	41391 ms	441 MB	0.160264900 66226163	0.839735099 3378484
29	129213 ms	541 MB	0.163625574 8405281	0.836374425 1594719
30	63448 ms	569 MB	0.157097465 45481418	0.84290253 45451858
Prom	63108,966 67 420	420	0	0

### Logistic Regression:

Corrida	Tiempo en segundos	Precisión	Prueba de error
1	10	0.884833502980951	0.11516649701904902
2	4	0.884833502980951	0.11516649701904902
3	4	0.884833502980951	0.11516649701904902
4	3	0.884833502980951	0.11516649701904902
5	3	0.884833502980951	0.11516649701904902
6	3	0.884833502980951	0.11516649701904902
7	3	0.884833502980951	0.11516649701904902
8	3	0.884833502980951	0.11516649701904902
9	3	0.884833502980951	0.11516649701904902



10	4	0.884833502980951	0.11516649701904902
11	3	0.884833502980951	0.11516649701904902
12	3	0.884833502980951	0.11516649701904902
13	3	0.884833502980951	0.11516649701904902
14	3	0.884833502980951	0.11516649701904902
15	3	0.884833502980951	0.11516649701904902
16	3	0.884833502980951	0.11516649701904902
17	4	0.884833502980951	0.11516649701904902
18	3	0.884833502980951	0.11516649701904902
19	4	0.884833502980951	0.11516649701904902
20	3	0.884833502980951	0.11516649701904902
21	4	0.884833502980951	0.11516649701904902
22	3	0.884833502980951	0.11516649701904902
23	3	0.884833502980951	0.11516649701904902
24	3	0.884833502980951	0.11516649701904902
25	3	0.884833502980951	0.11516649701904902
26	3	0.884833502980951	0.11516649701904902
27	4	0.884833502980951	0.11516649701904902
28	3	0.884833502980951	0.11516649701904902
29	3	0.884833502980951	0.11516649701904902
30	3	0.884833502980951	0.11516649701904902
Promedio	3.46666666	0.884833502980951	0.115166449701904902

### Multilayer Perceptron:

Corrida	Tiempo en	Precisión	Prueba de error
---------	-----------	-----------	-----------------



	segundos		
1	22	0.8829233550649208	0.11707664493507919
2	12	0.8829233550649208	0.11707664493507919
3	12	0.8829233550649208	0.11707664493507919
4	12	0.8829233550649208	0.11707664493507919
5	11	0.8829233550649208	0.11707664493507919
6	12	0.8829233550649208	0.11707664493507919
7	12	0.8829233550649208	0.11707664493507919
8	12	0.8829233550649208	0.11707664493507919
9	12	0.8829233550649208	0.11707664493507919
10	12	0.8829233550649208	0.11707664493507919
11	11	0.8829233550649208	0.11707664493507919
12	12	0.8829233550649208	0.11707664493507919
13	12	0.8829233550649208	0.11707664493507919
14	12	0.8829233550649208	0.11707664493507919
15	12	0.8829233550649208	0.11707664493507919
16	12	0.8829233550649208	0.11707664493507919
17	12	0.8829233550649208	0.11707664493507919
18	11	0.8829233550649208	0.11707664493507919
19	12	0.8829233550649208	0.11707664493507919
20	12	0.8829233550649208	0.11707664493507919
21	11	0.8829233550649208	0.11707664493507919
22	12	0.8829233550649208	0.11707664493507919
23	13	0.8829233550649208	0.11707664493507919
24	12	0.8829233550649208	0.11707664493507919
25	12	0.8829233550649208	0.11707664493507919
26	12	0.8829233550649208	0.11707664493507919



27	11	0.8829233550649208	0.11707664493507919
28	12	0.8829233550649208	0.11707664493507919
29	12	0.8829233550649208	0.11707664493507919
30	12	0.8829233550649208	0.11707664493507919
Promedio	12.2	0.8829233550649208	0.11707664493507919



## Conclusiones.

Podemos notar como cada algoritmo tiene distintas predicciones y cual es su margen de error, lo cual es interesante ya que se trabajó con el mismo archivo de datos (csv), es interesante el cómo se puede manejar una gran cantidad de datos con, este tipo de algoritmos, nos llevamos un aprendizaje de cómo usar estas herramientas y la gran cantidad de cosas que podemos realizar. En conclusión conforme efectividad el mejor de los algoritmos es el de SVM, ya que en los promedios muestra menor rango de error.



## Referencias.

MATLAB. (2015, 2 marzo). Support Vector Machine (SVM). MATLAB & Simulink. Recuperado 5 de junio de 2022, de <https://es.mathworks.com/discovery/support-vector-machine.html>

Unir, V. (2021, 19 octubre). Árboles de decisión: en qué consisten y aplicación en Big Data. UNIR. Recuperado 5 de junio de 2022, de <https://www.unir.net/ingenieria/revista/arboles-de-decision/>

Gonzalez, L. (2020, 21 agosto). Regresión Logística - Teoría. Aprende IA. Recuperado 5 de junio de 2022, de <https://aprendeia.com/regresion-logistica-multiple-machine-learning-teoria/#:%7E:text=La%20Regresi%C3%B3n%20Log%C3%ADstica%20es%20uno,cualquier%20problema%20de%20clasificaci%C3%B3n%20binaria.>

Sciencedirect. (2014a, abril 1). Multilayer Perceptron. Recuperado 5 de junio de 2022, de <https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron>

Link Youtube:

<https://www.youtube.com/watch?v=WDdLq3Ogg3Y>

Link GitHub:

<https://github.com/Aliciap26/DATOS-MASIVOS/blob/Unit-4/README.md>



**SEP**  
SECRETARÍA DE  
EDUCACIÓN PÚBLICA



TECNOLÓGICO NACIONAL DE MÉXICO

