UNIVERSIDAD DE GRANADA

Práctica 3. Procesamiento y minería de datos en Big Data con Spark sobre plataformas cloud.

Cloud Computing: Servicios y Aplicaciones

Máster Profesional en Ingeniería Informática

Curso académico 2022/2023

Autor

José Alberto Gómez García

26514779B

modej@correo.ugr.es

Índice

1. Introducción

En esta tercera práctica de la asignatura “Cloud Computing: Servicios y Aplicaciones” exploraremos y aplicaremos diversos métodos y técnicas del procesamiento de datos para analizar un conjunto de datos con un tamaño considerable. El objetivo final de esta práctica será la resolución de un problema de clasificación a través del desarrollo y entrenamiento de varios modelos.

Este trabajo implica el diseño conceptual, la programación y el despliegue de distintos modelos, apoyándonos en los métodos implementados en la biblioteca MLLib. Deberemos también usar las herramientas del ecosistema Spark. Una vez diseñados e implementados se comparará el rendimiento de los distintos clasificadores para identificar cuál resulta ser el más adecuado para el problema en cuestión realizando un estudio empírico comparativo.

Para la realización de esta práctica se hará uso del clúster “ulises.ugr.es” de la Universidad de Granada, que provee de un entorno con un sistema de archivos distribuido HDFS y una instalación de Spark con todas las herramientas que necesitaremos; listo para trabajar sobre él.

El problema a considerar es la realización de distintas tareas de análisis de datos sobre un conjunto de datos de objetos celestes, disponible en este enlace. El conjunto de dato define un problema de clasificación, para cuya resolución deberemos construir varios clasificadores.

2. Consideraciones iniciales

Para poder ejecutar el código desarrollado, este debe encontrarse en el clúster “ulises.ugr.es”. Dado que se ha desarrollado en una máquina local, el proyecto ha sido subido al servidor haciendo uso del comando SCP. Concretamente, se ha utilizado:

scp -r .* xxmodej@ulises.ugr.es:/home/xxmodej/practice3

El proyecto se divide en 4 ficheros principales, cuyo cometido se explica a continuación:

- main.py → Es el fichero de código principal, donde tiene lugar la carga de datos, parte del análisis exploratorio y preprocesamiento. En él se crea el vector de características, se realiza un proceso de normalización y se divide los datos en el conjunto de entrenamiento y prueba. Finalmente, se manda ejecutar la batería de entrenamientos y pruebas para los diferentes modelos (con diferentes conjuntos de parámetros).
- functions.py → En él se encuentran dos funciones. La primera de ella define los modelos (y conjunto de parámetros) a entrenar y probar, mientras que la segunda función es la encarga de realizar cálculos relativos a la bondad del modelo, como

precisión, área bajo la curva ROC, etc. Esta segunda función también salva los resultados en ficheros CSV.

- generate_graphs.py → En él está el código para generar una imagen con la matriz de confusión a partir de un fichero de texto que exportaremos haciendo uso de funcionalidades de PySpark y comandos propios de HDFS.
- Makefile → En él se tiene la orden de ejecución del fichero principal. Dado que es un tanto larga creamos un "atajo" para nuestra comodidad durante el desarrollo de la práctica. También tenemos atajos para mover el directorio con los ficheros que conforman la matriz de correlación a HDFS y obtener un TXT resultado.

Como puede haberse dado cuenta el lector, esta práctica se realizará haciendo uso del lenguaje de programación Python.

3. Resolución del problema planteado

3.1. Creación del entorno y carga de datos

Dado que no vamos a hacer uso de la consola interactiva de PySpark, deberemos crear un contexto y una sesión dentro de nuestro script. Para ello, crearemos una instancia de la clase SparkContext (a la que pasamos la URL del clúster al que conectarse y el nombre de la aplicación) y una instancia de la clase SparkSession.

Posteriormente, leeremos los datos del fichero CSV ubicado en el directorio del profesor dentro del sistema de archivos HDFS. Para ello, haremos uso del comando:

```
df = ss.read.csv("hdfs://ulises.imuds.es:8020/user/CCSA2223/ccano/train.csv",
header=True, sep=",", inferSchema=True)
```

3.2. Análisis exploratorio y preprocesamiento

Tras haber cargado los datos imprimiremos el esquema de los mismos y comprobaremos el número de filas, columnas y cuantas instancias tenemos de cada clase en la que se pueden clasificar las observaciones (estrellas o galaxias).

Todas las variables son numéricas de tipo "double" a excepción del ID del objeto, "run", "camcol" y "field" que son numéricas enteras. La variable de clasificación, "type", es de tipo texto, por lo que deberemos convertirla a tipo numérico haciendo uso del siguiente comando:

```
df = df.withColumn("type", when(col("type") == "galaxy",
0.0).otherwise(1.0).cast("double"))
```

Tenemos 4 millones de observaciones, 1.999.996 galaxias y 2.000.004 estrellas. El problema por lo tanto está balanceado y no requerirá de procesamiento adicional en este sentido (como podría ser under-sampling u over-sampling).

Si acudimos a la web de la que hemos obtenido el dataset, podremos consultar a qué hace referencia cada característica de las presentes en el conjunto de datos. No realizaremos un análisis exhaustivo dado que no somos astrofísicos ni tenemos estudios de astronomía, pero sí que podemos mencionar que muchas variables tienen valores similares en tanto que miden una determinada característica para cada una de las 5 bandas de color de la imagen analizada. Otras características parecen estar relacionada con la velocidad del objeto celeste y su posición en la galaxia.

Consultar esta web nos ha servido para darnos cuenta de que las características 'rowv' y 'colv' tienen valores nulos representados en forma de -9999. Esto sucede en 492 de los 4 millones de observaciones, por lo que decidimos eliminar dichas observaciones y no emplear otros métodos como imputación por media o mediana. En principio no hay más valores nulos, pero decidimos ejecutar el comando que se muestra a continuación para eliminar los posibles NaN, como medida de precaución.

```
df.select([count(when(isnan(c), c)).alias(c) for c in df.columns]).head()
```

Tras haber preprocesado los datos, seleccionaremos las características que queremos que tengan en cuenta los clasificadores y construiremos el "vector de características". Este vector no debe contener la variable objeto de la clasificación ("type").

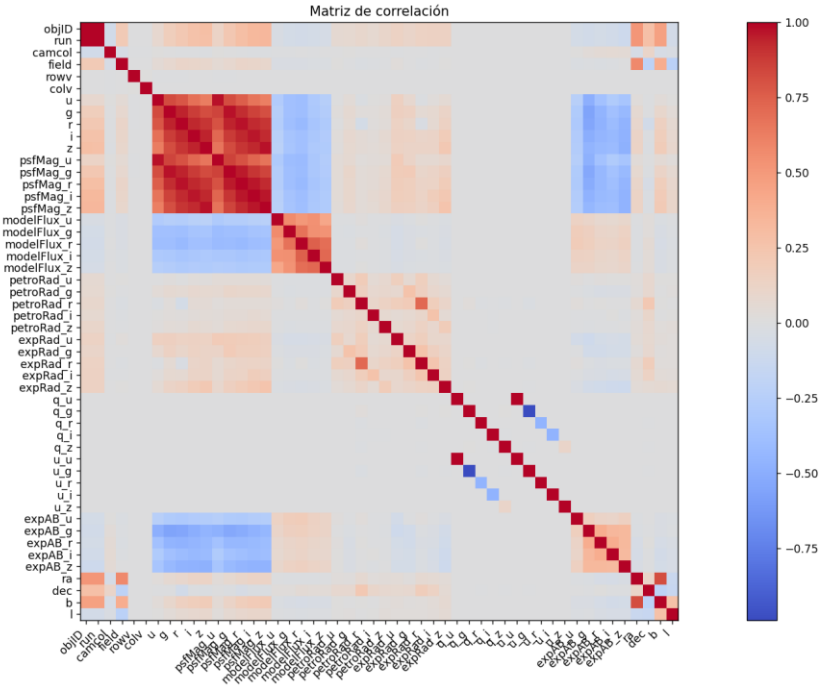
Se decide no incluir variables que en principio no parecen determinar el tipo de cuerpo celeste observado, como el ID de la observación, la cámara que ha realizado la observación, el "run number" y el "field number".

```
non_interesting = ['type', 'objID', 'run', 'camcol', 'field']
selected_features = [col for col in df.columns if col != 'type']
assembler = VectorAssembler(inputCols=selected_features, outputCol="features")
df = assembler.transform(df)
```

En otra variante de los experimentos que comentaremos más adelante, decidimos calcular la matriz de correlación de Pearson entre las variables y, para cada pareja con correlación superior a 0.7, eliminar una de ellas. Este proceso elimina las 17 variables que se listan a continuación. Posteriormente, volvemos a construir el vector de características como se ha mostrado anteriormente.

```
Number of features before removing correlated features:  52
Number of features after removing correlated features:  35
Features removed:  ['petroRad_g', 'psfMag_g', 'r', 'i', 'modelFlux_g',
'modelFlux_r', 'colv', 'objID', 'psfMag_i', 'g', 'q_u', 'psfMag_r', 'u', 'z',
'psfMag_u', 'expAB_z', 'expRad_z']
```

La matriz de correlación se puede observar en la siguiente imagen:



Tras haber creado el vector de características, deberemos normalizar los valores de las variables; práctica habitual para intentar mejorar los resultados de los clasificadores a entrenar.

Vamos a hacer uso de dos normalizadores distintos, el primero de ellos será el "StandardScaler" que, según la documentación "estandariza las características eliminado la media y escalando a la varianza unitaria". El segundo será el "MinMaxScaler", el cual escala los valores al rango [0,1]. Realizaremos una batería de pruebas con cada uno de ellos. Téngase en cuenta que en el caso de "StandarScaler" no podremos usar el método Naive-Bayes, dado que hay valores negativos no soportados por el algoritmo.

Finalmente, dividiremos los datos en el conjunto de entrenamiento y prueba con una proporción 80/20.

3.3. Modelos y ajustes

Un clasificador, a grandes rasgos, es un algoritmo que recibiendo como entrada cierta información de una observación es capaz de predecir la clase a la que pertenecerá. Estos clasificadores son entrenados a partir de un conjunto inicial de datos, diferente del que se usará para evaluarlos.

En esta sección se comentan brevemente los cuatro modelos de clasificación elegidos. También se comenta la función de evaluación que vamos a emplear.

3.3.1. Modelo de regresión logística

Los modelos de regresión logística calculan las variables de las observaciones para calcular la probabilidad de que dicha observación pertenezca a una clase específica. Luego, aplica una función llamada "función sigmoide" para convertir esas probabilidades en valores entre 0 y 1. Si el valor está por encima de un umbral específico (como 0.5), se clasifica en una clase determinada; en caso contrario se clasifica en la otra clase. Nótese que asumimos un problema binario, como es nuestro caso.

Este modelo se encuentra implementado en la librería PySpark, por lo que sólo tendríamos que crear un objeto de dicha clase y entrenarlo. Realmente el proceso que vamos a seguir es algo más complejo, ya que definimos una rejilla de parámetros para el coeficiente de regularización y usamos validación cruzada para estimar que valor de dicho parámetro proporcionaría mejor resultado. Dado que los tiempos de ejecución son algo altos nos limitamos a esta rejilla, pero podríamos definir otra para el parámetro “elasticNet”. Una vez tenemos la combinación de parámetros, entrenamos el modelo como tal.

Esto lo podemos hacer con los dos siguientes fragmentos de código:

```
lr = LogisticRegression(featuresCol='features', labelCol='type', maxIter=10, elasticNetParam=0.8)
grid = ParamGridBuilder().addGrid(lr.regParam, [0.1, 0.01, 0.001, 0.0001]).build()
evaluator = BinaryClassificationEvaluator(labelCol="type", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
cv = CrossValidator(estimator=lr, estimatorParamMaps=grid, evaluator=evaluator)
cv_model = cv.fit(data_train)
best_model = cv_model.bestModel
best_lambda = best_model._java_obj.getRegParam()
```

```
timeIni = time.time()
lr = LogisticRegression(featuresCol='features', labelCol='type', maxIter=10, regParam=best_lambda, elasticNetParam=0.8)
lr_model = lr.fit(data_train)
timeFin = time.time()
predictions = lr_model.transform(data_test)
lr_metrics = evaluate_model(predictions, ss, labels[0])
```

3.3.2. Naive Bayes

El clasificador Naive Bayes asume que el efecto de cada característica en una clase es independiente del efecto que puedan tener el resto de las características. Hace uso del teorema de Bayes para calcular la probabilidad de que la observación pertenezca a una clase u otra en función de las características observadas. Es un enfoque “ingenuo” y simple, pero que requiere de menos tiempo que otras alternativas.

Este modelo también se encuentra implementado en PySpark. Para su entrenamiento seguimos la filosofía del apartado anterior, buscar el parámetro adecuado mediante una rejilla y una vez lo tengamos entrenar el modelo. Este algoritmo solo tiene un parámetro numérico personalizable por el usuario, llamado “smoothing”.

El código empleado es el siguiente:

```
nb = NaiveBayes(featuresCol='features', labelCol='type', smoothing=1.0, modelType="multinomial")
evaluator = BinaryClassificationEvaluator(labelCol="type", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
grid = ParamGridBuilder().addGrid(nb.smoothing, [0.1, 0.5, 1.0, 2.0]).build()
cv = CrossValidator(estimator=nb, estimatorParamMaps=grid, evaluator=evaluator)
cv_model = cv.fit(data_train)
best_model = cv_model.bestModel
best_smoothing = best_model._java_obj.getSmoothing()
```

```
timeIni = time.time()
nb = NaiveBayes(featuresCol='features', labelCol='type', smoothing=best_smoothing, modelType="multinomial")
nb_model = nb.fit(data_train)
timeFin = time.time()
predictions = nb_model.transform(data_test)
nb_metrics = evaluate_model(predictions, ss, labels[1])
```

3.3.3. Árbol de clasificación

Un árbol de clasificación es un algoritmo que se utiliza para tomar decisiones basadas en características o atributos de los datos. Se parte de un nodo raíz que representa todo el conjunto de datos, y en cada nodo se hace una pregunta que va dividiendo los datos en diferentes ramas en función de la respuesta

Aunque se puede variar el algoritmo concreto que utiliza para calcular la condición óptima y el número mínimo de observaciones que debe haber en cada rama, dado los altos tiempos de ejecución nos limitaremos a variar el número de niveles máximo que puede tener el árbol. Presumiblemente mayor número de niveles proporcionaría mejores resultados, pero hay ocasiones en las que modelos más simples generan mejores predicciones.

El código empleado (similar al de apartados anteriores) se adjunta a continuación:

```
dt = DecisionTreeClassifier(featuresCol='features', labelCol='type', maxDepth=5)
evaluator = BinaryClassificationEvaluator(labelCol="type", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
grid = ParamGridBuilder().addGrid(dt.maxDepth, [5, 10, 15]).build()
cv = CrossValidator(estimator=dt, estimatorParamMaps=grid, evaluator=evaluator)
cv_model = cv.fit(data_train)
best_model = cv_model.bestModel
best_max_depth = best_model._java_obj.getMaxDepth()
```

```
timeIni = time.time()
dt = DecisionTreeClassifier(featuresCol='features', labelCol='type', maxDepth=best_max_depth)
dt_model = dt.fit(data_train)
timeFin = time.time()
predictions = dt_model.transform(data_test)
dt_metrics = evaluate_model(predictions, ss, labels[2])
```

3.3.4. Random Forest

Un "Random Forest" es un algoritmo de aprendizaje automático que combina múltiples árboles de decisión para obtener predicciones más precisas y estables. Se selecciona una muestra aleatoria con reemplazo del conjunto de datos de entrenamiento, y se utiliza dicha muestra para crear un árbol de decisión. Cada árbol se construye haciendo preguntas diferentes, lo que da lugar a divisiones de los datos diferentes.

Para hacer una predicción ante una nueva observación, se pasa por cada árbol de decisión individual y se obtiene la predicción de clasificación. Luego, se utiliza una estrategia de voto mayoritario para determinar la predicción final.

Al igual que en el caso anterior, podemos elegir la profundidad máxima de cada árbol. Adicionalmente, podemos elegir cuantos árboles conformaran el bosque. Así pues, definimos una doble rejilla de parámetros para buscar la combinación óptima.

El código empleado se muestra a continuación:

```
rf = RandomForestClassifier(featuresCol='features', labelCol='type', numTrees=10, maxDepth=5)
evaluator = BinaryClassificationEvaluator(labelCol="type", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
grid = ParamGridBuilder().addGrid(rf.maxDepth, [5, 10]).addGrid(rf.numTrees, [5, 10, 15]).build()
cv = CrossValidator(estimator=rf, estimatorParamMaps=grid, evaluator=evaluator)
cv_model = cv.fit(data_train)
best_model = cv_model.bestModel
best_max_depth = best_model._java_obj.getMaxDepth()
best_num_trees = best_model._java_obj.getNumTrees()
```

```
timeIni = time.time()
rf = RandomForestClassifier(featuresCol='features', labelCol='type', numTrees=best_num_trees, maxDepth=best_max_depth)
rf_model = rf.fit(data_train)
timeFin = time.time()
predictions = rf_model.transform(data_test)
rf_metrics = evaluate_model(predictions, ss, labels[3])
```

3.3.5. Función de evaluación

Tras determinar la combinación de parámetros óptima para cada modelo, se procede a su entrenamiento y evaluación. La evaluación se realiza a partir de las predicciones en una función diferente, encargada de calcular el área bajo la curva ROC (AUC-ROC), la precisión, el coeficiente Kappa, el "recall" y el "F1-Score".

Comentemos brevemente la utilidad de cada métrica.

- AUC-ROC → mide la capacidad de un modelo para distinguir entre clases positivas y negativas. La curva ROC representa la tasa de verdaderos positivos en función de la tasa de falsos positivos. Su rango oscila entre 0 y 1, siendo un valor mayor indicativo de un mejor rendimiento del modelo de clasificación. Nosotros lo expresaremos en el rango [0, 100]
- Precisión → mide la proporción de casos positivos clasificados correctamente en relación con todos los casos clasificados como positivos. Se suele representar en el rango [0, 1], siendo un mayor valor síntoma de un mejor rendimiento del modelo. Al igual que todas las medidas, las representaremos en el rango [0, 100].
- Coeficiente Kappa → mide la concordancia entre las predicciones del modelo y las etiquetas reales, teniendo en cuenta el acuerdo esperado por azar. Proporciona una medida de la precisión del modelo ajustado al azar. El coeficiente Kappa varía entre -1 y 1, donde 1 indica una concordancia perfecta y valores cercanos a 0 o negativos indican una concordancia pobre.
- Recall (o sensibilidad) → mide la proporción de casos positivos clasificados correctamente en relación con todos los casos positivos reales. Se calcula dividiendo el número de verdaderos positivos entre la suma de verdaderos positivos y falsos negativos. El recall es útil cuando el objetivo es minimizar los falsos negativos.
- F1-Score → combina la precisión y el recall en una sola medida. Es la media armónica de la precisión y el recall, y proporciona un equilibrio entre ambas métricas. Se mueve en el rango [0, 1]; siendo un valor mayor indicativo de un mejor rendimiento.

4. Experimentos y comparativa.

Para comprobar el rendimiento de los modelos se van a ejecutar tres baterías de tests.

En la primera de ellas utilizaremos todas las características (salvo las excepciones mencionadas en apartados anteriores) y el normalizador MinMaxScaler para entrenar los modelos con las diferentes rejillas de parámetros.

Una vez hallamos obtenido las combinaciones óptimas para cada modelo, realizaremos un entrenamiento y proceso de evaluación, cuyos resultados se muestran en la siguiente tabla:

	ROC	Precisión	Kappa	Recall	F1	T. Exec (segundos)
Logistic Regression	86.945	78.781	57.577	68.689	76.43	68.9963
Naive Bayes	78.057	76.679	53.361	75.161	76.35	21.2798
Decision Tree	98.163	96.966	93.932	97.195	96.978	264.1270
Random Forest	99.573	96.693	93.387	97.188	96.715	155.1989

Para la regresión logística, se utiliza un valor del parámetro lambda de 0.1. Para Naive Bayes se usa un valor de "smoothing" igual a 0.5. Para el árbol de decisión se emplea una profundidad máxima de 10 niveles, mientras que para el "random forest" se emplean 10 árbol de profundidad 10. Nótese que no se utiliza el máximo número de árboles que hemos definido en la rejilla (15).

Como podemos observar en la tabla, las predicciones de los modelos de regresión logística y Naive Bayes son bastante aceptables, con precisiones, recalls, F1 y ROC por encima de 0.68. Los resultados de Naive Bayes son los peores de la batería de tests (salvo para "recall", donde precisión logística presenta peor valor), lo cual era ciertamente previsible. Este algoritmo supone independencia entre variables, pero muchas de ellas se refieren a distintos canales de color de una imagen, por lo que podemos intuir que existe cierta dependencia. A cambio, es el algoritmo más rápido de ejecutar y testear.

Por otra parte, los modelos basados en árboles de clasificación (individual o en bosque) funcionan extraordinariamente bien, casi a la perfección, con precisiones cercanas al 97%. El resto de las métricas también presentan valores muy elevados. Sin embargo, cabe destacar que los tiempos necesarios para entrenar a los modelos son bastante mayores, pudiendo tardar el árbol de clasificación 13 veces más en entrenarse que Naive Bayes, o unas 4 veces más que la regresión logística. Es curioso también observar que el "random forest" se entrena más rápido que un único árbol de decisión de mayor profundidad.

Dado que Naive Bayes es el algoritmo con peores resultados, y viendo que podemos obtener resultados muy buenos con árboles de decisión, planteamos una segunda tanda de experimentos que hace uso del StandardScaler de PySpark. Nótese que este normalizador no se puede usar con Naive Bayes, dado que mantiene valores negativos que el algoritmo Naive Bayes no puede procesar. El objetivo es ver si podemos mejorar las métricas del algoritmo de precisión logística.

Los resultados de esta tanda de experimentos se resumen en la siguiente tabla:

	ROC	Precisión	Kappa	Recall	F1	T. Exec (segundos)
Logistic Regression	97.736	92.285	84.458	95.601	92.544	37.4708
Decision Tree	98.038	96.953	93.906	97.123	96.963	260.6748
Random Forest	99.573	96.693	93.387	97.188	96.715	155.7635

En esta ocasión, se ha usado un parámetro “lambda” con valor 0.0001 para regresión logística, árbol de decisión de profundidad máxima 15 y “random forest” con 10 árboles de profundidad máxima 10 niveles.

Para el caso de los métodos basados en árboles de decisión podemos observar que no existe gran diferencia, manteniéndose las métricas en valores muy altos. Sin embargo, podemos comprobar que usar este normalizador provoca una grandísima mejora en el modelo basado en regresión logística, el cual consigue alcanzar una precisión del 92% y un área bajo la curva del 97.74%. El tiempo de ejecución de este modelo se ha reducido aún más, siendo unas 4.16 y 7 veces más rápido que “random forest” y árbol de decisión individual respectivamente.

Por último, se plantea una tercera tanda de experimentos. Esta vez haremos uso del StandardScaler y el conjunto de variables resultado de tratar aquellas que presentan una alta correlación. El objetivo es comprobar si podemos obtener los mismos resultados, o incluso mejores, haciendo uso de un menor conjunto de variables.

En esta ocasión, se ha usado un parámetro “lambda” con valor 0.0001 para regresión logística, árbol de decisión de profundidad máxima 15 y “random forest” con 10 árboles de profundidad máxima 15 niveles.

Los resultados obtenidos se muestran en la siguiente tabla:

	ROC	Precisión	Kappa	Recall	F1	T. Exec (segundos)
Logistic Regression	98.42	94.287	88.573	94.872	94.329	33.1795
Decision Tree	98.404	96.757	93.515	97.1	96.774	212.3675
Random Forest	99.494	96.468	92.935	96.827	96.486	171.3733

Podemos ver como se obtienen resultados parecidos al apartado anterior. No obstante, observamos una ligera mejora de precisión en el modelo de regresión logística, el cual consigue alcanzar una precisión del 94.3% aproximadamente. Su desempeño es relativamente similar a los modelos basados en árbol de decisión, pero requiriendo tiempos de entrenamiento mucho menores.

5. Conclusiones.

En esta práctica se han aplicado diversos métodos y técnicas del procesamiento de datos para analizar un conjunto de datos considerablemente grande, haciendo uso de plataformas en la nube (clúster “ulises.ugr.es”).

En particular, se ha analizado la influencia del normalizador y de las características concretas seleccionadas a la hora de entrenar cuatro modelos de clasificación: regresión logística, Naive Bayes, árboles de decisión y Naive Bayes con diferentes rejillas de parámetros.

Se ha podido observar que el uso del normalizador StandardScaler y el modelo de regresión logística permite obtener resultados muy buenos con un tiempo de entrenamiento considerablemente menor al de otras alternativas. Los mejores resultados fueron obtenidos haciendo uso de modelos basados en árboles de clasificación.

También se ha podido comprobar que usar un conjunto de variables reducido, en el que se elimina una de las variables de aquellas parejas con alta correlación, permite mejorar ligeramente los resultados. Podemos obtener clasificadores con más de un 94% de precisión en tiempos asumibles con cierta facilidad.

6. Bibliografía.

- Apuntes de la asignatura “Cloud Computing: Servicios y Aplicaciones”.
- Apuntes de la asignatura “Tratamiento Inteligente de Datos”.
- Documentación de PySpark.