Máster Universitario en Ingeniería Informática

CLOUD COMPUTING

Práctica 4:
Procesamiento y minería de datos en Big Data con Spark sobre plataformas cloud

UNIVERSIDAD DE GRANADA

Carlos Morales Aguilera
75925767-F
carlos7ma@correo.ugr.es

Curso Académico 2020-2021

Índice

1. Introducción

1.1. Objetivo

En esta práctica se pretende resolver un problema de *Machine Learning* haciendo uso de herramientas y plataformas Cloud, las cuales se encuentran relacionadas con **Hadoop**, tal y como se ha podido obsrevar en la asignatura.

El objetivo de la práctica es resolver un problema sobre clasificación binaria de diferentes datos de proteínas, para el cual se deben utilizar una serie limitada de columnas, obtener dicha información del sistema **HDFS** de **Hadoop**, y procesarlos en **Databricks** utilizando **Spark** con las herramientas de la **MLLib**.

1.2. HDFS

Es el sistema de ficheros de **Hadoop**, para el cual debemos conectarnos al mismo, y obtener el conjunto de datos, para ello obteniendo el fichero de cabecera y a continuación el fichero de datos **ECBDL14_IR2.header** y **ECBDL14_IR2.data**, respectivamente.

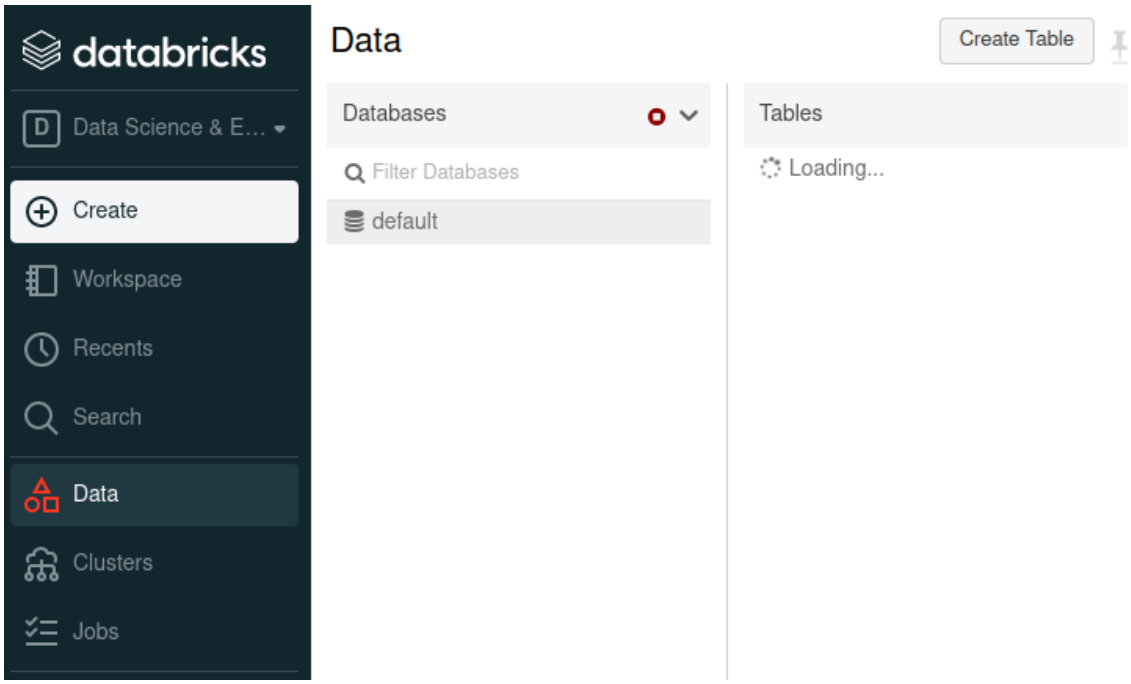
Para ello simplemente se han descargado ambos ficheros, y debido a la escasez de recursos en el servidor de **Hadoop**, se ha hecho el procedimiento de extracción de las columnas en un entorno local.

En este caso las columnas a utilizar son: **'PSSM_central_-1_I', 'PSSM_r2_-1_R', 'AA_freq_global_S', 'PSSM_central_-2_Y', 'PSSM_central_-1_R', 'PSSM_r1_0_K'**.

Una vez obtenido el fichero, descargándolo con **sqlContext**, se procesan y se transforman a un fichero **.csv** que se importará en **Databricks**.

1.3. Databricks

Inicialmente, tras crearnos una cuenta, se ha de crear un **Cluster**, y en ese cluster definir un **Notebook** para la realización de la práctica utilizando **Spark**. Inicialmente cargamos los datos seleccionando la opción de **Data** en el menú desplegable y a continuación en **Create Table**:



A continuación se define el **Cluster** y se guardan los datos en el **Notebook**. Cabe señalar que la realización de la práctica se ha llevado a cabo utilizando **Python**.

2. Resolución de tareas

2.1. Preprocesamiento

Tras el procesamiento que se ha llevado a cabo previamente de los datos e importarlos como **.csv**, la lectura de los mismos es una tarea sencilla utilizando un **sqlReader.read.csv**, al cual se le indica la ruta en **Databricks**:

```
# Leer conjunto de datos
df =
    sqlContext.read.csv("/FileStore/tables/carlos_morales_aguilera-training.csv",
    sep=",", header=True, inferSchema=True)
```

A continuación, una de las técnicas más comunes es la omisión de valores perdidos o valores **NA**, los cuales no aportan información y pueden llevar a conclusiones erróneas, por lo que lo mejor es eliminarlos:

```
# Omision de valores perdidos
df = df.na.drop()
```

Tras analizar los diferentes modelos de la **MLLib** de **Spark** se ha podido observar que trabajan con dataframes con dos columnas que contienen toda la información, siendo estas columnas **label** (equivalente a nuestra **class**) y **features** (que reune todas las caracteristicas). Para obtener dichas se agrupan con la clase **VectorAssembler** y se renombra la columna **class** por **label**:

```
# Crear columna features
assembler = VectorAssembler(inputCols=df.columns[:-1],outputCol="features")
df = assembler.transform(df)

# Cambiar columna class por label
df = df.withColumnRenamed("class", "label")
drop_cols = ['PSSM_central_-1_I', 'PSSM_r2_-1_R', 'AA_freq_global_S',
    'PSSM_central_-2_Y', 'PSSM_central_-1_R', 'PSSM_r1_0_K']
df = df.drop(*drop_cols)
print(df.columns)
```

El siguiente paso en cuanto al preprocesamiento sería el balanceo de los datos, para los cuales se ha tomado la opción de aplicar **RUS** (Random Under-Sampling), el cual consiste en aleatoriamente eliminar un conjunto de datos aleatorios de la clase con mayor cantidad de casos, siendo esta la clase *0* e igualandola a la otra clase:

```
# Balancear las clases, para ello filtramos segun clases
df_0 = df.filter("label = 0")
df_1 = df.filter("label = 1")
# Comprobamos que conjunto posee mas elementos
print("Clase 0: " + str(df_0.count()) + "\nClase 1: " + str(df_1.count()))

# Aplicamos downsampling aleatorio sobre el conjunto de la clase 0
df_0_resample = df_0.sample(withReplacement=False,
    fraction=1.0,seed=0).limit(df_1.count())

# Unimos finalmente el conjunto de entrenamiento
df = df_0_resample.union(df_1)
```

Por último, algunos modelos requieren del uso de un indexador de las diferentes variables, determinando variables categóricas o continuas y la indexación de la variable de clasificación, por lo que para estas funciones se utilizan las clases **StringIndexer** y **VectorIndexer**:

```
# Indexamos las etiquetas (clase)
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(df)
# Indexamos las caracteristicas (max 4 categorias o sino se considera variables
    continuas)
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures",
        maxCategories=4).fit(df)
```

2.2. Particionamiento de los datos

Se ha optado por realizar un particionamiento de los datos, para el cual se utiliza la función **randomSplit** de **Spark**, la cual permite definir los porcentajes de cada subconjunto y una semilla con el objetivo de realizar los resultados de forma reproducible (ya que los modelos son no determinísticos):

```
# Realizar particiones de train (0.8) y test (0.2)
data_train, data_test = df.randomSplit([0.8, 0.2], 0)
```

2.3. Clasificación con diferentes modelos

A continuación, se muestran los diferentes modelos que se han aplicado en la práctica, con dos parametrizaciones diferentes estudiadas, siendo estos:

- Árbol de decisión.
- Random Forest.
- SVM Lineal.
- Regresión Logística.

A continuación se estudiarán las diferentes implementaciones de los mismos haciendo uso de la **MLLib** de **Spark** y se evaluará en una sección final el desempeño de los mismos, mostrando tanto su *accuracy* como el tiempo de ejecución.

2.3.1. Árbol de decisión

Para el modelo del árbol de decisión se aplican los indexadores mencionados previamente con el modelo de la clase **DecisionTreeClassifier**, y se indican las diferentes parametrizaciones, haciendo uso de un **Pipeline** que encauza las diferentes tareas para la indexación y entrenamiento del modelo con el conjunto de datos.

En cuanto a la parametrización del árbol de decisión se consideran los siguientes parámetros:

- **maxDepth**: Controla la profundidad máxima que puede alcanzar un árbol al ir creciendo hacia abajo en las diferentes separaciones.
- **maxBins**: Controla el número máximo de nodos que se pueden crear en el árbol, por lo que números reducidos limitarían el número de conjuntos finales.

A continuación se entrena con la función **fit** y se definen las predicciones sobre el **RDD** con la función **transform**, para finalmente con la clase **MultiClassificationEvaluator** obtener la comparativa y métrica de *accuracy* obtenida.

```
# Modelo de Arbol de decision
dt = DecisionTreeClassifier(labelCol="indexedLabel",
    featuresCol="indexedFeatures", maxDepth=5, maxBins=32)

# Definimos el Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dt])

# Entrenar modelo
model = pipeline.fit(data_train)

# Realizar predicciones
predictions = model.transform(data_test)

# Calcular accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Accuracy = %g " % (accuracy))
```

2.3.2. Random Forest

Para el modelo de Random Forest se aplican los indexadores mencionados previamente con el modelo de la clase **RandomForestClassifier**, y se indican las diferentes parametrizaciones, haciendo uso de un **Pipeline** que encauza las diferentes tareas para la indexación y entrenamiento del modelo con el conjunto de datos.

En cuanto a la parametrización del modelo se consideran los siguientes parámetros:

- **numTrees**: Controla la cantidad total de árboles de decisión que se emplean en el algoritmo para promediar.

A continuación se entrena con la función **fit** y se definen las predicciones sobre el **RDD** con la función **transform**, para finalmente con la clase **MultiClassificationEvaluator** obtener la comparativa y métrica de *accuracy* obtenida.

```
# Modelo de RandomForest
rf = RandomForestClassifier(labelCol="indexedLabel",
    featuresCol="indexedFeatures", numTrees=10)

# Transformar las etiquetas
labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel",
                               labels=labelIndexer.labels)

# Definimos el Pipeline
pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])

# Entrenar modelo
model = pipeline.fit(data_train)

# Realizar predicciones
predictions = model.transform(data_test)

# Calcular accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Accuracy = %g " % (accuracy))
```

2.3.3. SVM Lineal

Para el modelo de SVM Lineal se aplica el modelo de la clase **LinearSVC**, y se indican las diferentes parametrizaciones, en cuanto a la parametrización del modelo se consideran los siguientes parámetros:

- **maxIter**: Controla el número máximo de iteraciones que realiza el modelo.
- **regParam**: **MLlib** usa el parámetro de regularización como escalado de la penalización.

A continuación se entrena con la función **fit** y se definen las predicciones sobre el **RDD** con la función **transform**, para finalmente con la clase **MultiClassificationEvaluator** obtener la comparativa y métrica de *accuracy* obtenida.

```
# Modelo de SVM Lineal
lsvc = LinearSVC(maxIter=10, regParam=0.1)

# Entrenar modelo
model = lsvc.fit(data_train)

# Realizar predicciones
predictions = model.transform(data_test)

# Calcular accuracy
evaluator = MulticlassClassificationEvaluator(labelCol="label",
    predictionCol="prediction",
                                              metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Accuracy = " + str(accuracy))
```

2.3.4. Regresión Logística

Para el modelo de Regresión Logística se aplica el modelo de la clase **LogisticRegression**, y se indican las diferentes parametrizaciones, en cuanto a la parametrización del modelo se consideran los siguientes parámetros:

- **maxIter**: Controla el número máximo de iteraciones que realiza el modelo.
- **regParam**: **MLlib** usa el parámetro de regularización como escalado de la penalización.
- **elasticNetParam**: Controla el parámetro de regularización elástica de la red.

A continuación se entrena con la función **fit** y se definen las predicciones sobre el **RDD** con la función **transform**, para finalmente con la clase **MultiClassificationEvaluator** obtener la comparativa y métrica de *accuracy* obtenida.

```
# Modelo de Regresion Logistica
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Entrenar modelo
model = lr.fit(data_train)

# Realizar predicciones
predictions = model.transform(data_test)

# Calcular accuracy
evaluator = MulticlassClassificationEvaluator(labelCol="label",
    predictionCol="prediction",
                                              metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Accuracy = " + str(accuracy))
```

2.4. Resultados

A continuación, se realiza una tabla comparativa de los diferentes resultados obtenidos con la parametrización escogida, para comprobar que algoritmos obtienen mejores resultados:

Modelo	**Parametrización**	**Accuracy**	**Tiempo**
Decision Tree	maxDepth=5 maxBins=32	0.57914	2.56
Decision Tree	maxDepth=10 maxBins=64	0.586755	1.78
Random Forest	numTrees=10	0.577644	2.17
Random Forest	numTrees=25	0.578573	2.53
Linear SVC	maxIter=10 regParam=0.1	0.5732565536836762	1.62
Linear SVC	maxIter=20 regParam=0.3	0.5694725079850615	1.65
Logistic Regression	maxIter=10 regParam=0.3 elasticNetParam=0.8	0.4988578642556011	1.76
Logistic Regression	maxIter=30 regParam=0.2 elasticNetParam=0.9	0.4988578642556011	1.51

Se observa que los modelos asociados a árboles de decisión que detectan patrones en las características, aunque los modelos de SVM lineales obtienen un resultado en menor tiempo considerablemente buenos para los resultados finales.

Finalmente el resultado más prometedor es el de **Decision Tree** con profundidad máxima de 10 y número máximo de nodos de 64.

3. Conclusiones

Durante la realización de la práctica me he encontrado con que en diferentes momentos el servidor de **Hadoop** estaba ocupado por otros procesos por los que decidí trabajar el preprocesamiento en local, y posteriormente trasladar los datos a **Databricks**.

Se han podido comprobar las ventajas que suponen herramientas como **Spark** para resolver problemas de **Machine Learning** de forma sencilla y con pocos recursos, el cual considero que era el objetivo de la práctica. Aunque evidentemente existen una serie de limitaciones en el problema planteado.

En cuanto a la parametrización de modelos, se han probado una serie de parametrizaciones comprobando como afectaban estas a los diferentes resultados y tratando de optimizarlos, pero al trabajar con una serie limitada de columnas, no se puede asegurar que se obtengan los mejores resultados y el preprocesamiento sencillo ha sido básico, por lo que realmente habría que aplicar mejores técnicas y decisiones pero por cuestiones de limitación de tiempo no se han podido llevar a cabo.

Se ve que los modelos que se adaptan mejor están relacionados con los árboles decisión porque las características deben ser separables, y además el tiempo de ejecución es menor en árboles de decisión, por lo que modelos sencillos de aprendizaje resultan efectivos dando el conjunto de datos, realizándose en una plataforma que procesa en menor tiempo de ejecución.