Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Máster Profesional en Ingeniería Informática

Curso 2020/2021

PRÁCTICA 4

Cloud Computing: Servicios y Aplicaciones

Breve descripción

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

Autor

Álvaro de la Flor Bonilla (alvdebon@correo.ugr.es) 15408846-L

RESUMEN

En el desarrollo de esta práctica se estudiarán y pondrán en uso diferentes métodos y técnicas de procesamiento de datos para grandes volúmenes de datos. En concreto, el objetivo final será la resolución de un problema de clasificación a través del desarrollo de distintos modelos. Este trabajo implica el diseño conceptual, la programación y el despliegue de distintos modelos, apoyándose en los métodos implementados en la biblioteca MLLib. Para ello habrá que usar hábilmente distintas herramientas de los ecosistemas de Hadoop y Spark, desplegadas sobre distintos escenarios. 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 el desarrollo de la práctica se usará el cluster hadoop.ugr.es, que provee de HDFS y Spark ya preparado y listo para trabajar sobre él, y que proporciona todas las herramientas para el procesamiento. También se probará sobre otras plataformas: Azure, DataBricks, ...

El problema a considerar es la realización de distintas tareas de análisis de datos sobre un conjunto de datos concreto. Más concretamente, el conjunto define un problema de clasificación y se trata de construir varios clasificadores que los resuelvan.

ÍNDICE DEL PROYECTO

ÍNDICE DE ILUSTRACIONES

1 RESOLUCIÓN DE PROBLEMAS

En primer lugar, para ejecutar nuestro proyecto deberemos tenerlo localizado en el clúster *"hadoop.ugr.es"*. Una vez localizados todos los archivos basta con que ejecutemos la siguiente orden:

"make all"

En el caso de que ya se haya hecho con anterioridad el filtrado del *"dataset"* correspondiente al usuario, es decir, se haya ejecutado con anterioridad el archivo *"créate_new_df.py"* solo tendremos que usar la orden:

"make evaluate"

Es muy importante que con anterioridad se realice esta exportación de variables, mediante las siguientes órdenes:

"export PYSPARK_PYTHON=python3.8"

"export PYSPARK_DRIVER_PYTHON=python3.8"

Una vez configurado el entorno podremos empezar a trabajar, en primer lugar, extrayendo las columnas asignadas y construyento el nuevo *"dataframe"*.

Finalmente, destacar que todo el desarrollo ha sido realizado baje el lenguaje ***"Python"***.

1.1 Extracción de columnas y creación de *"dataframe"*

Todo lo realizado en este bloque corresponde al script del archivo *"create_new_df.py"*. El procedimiento que realizamos en el es leer el fichero original de HDFS (utilizando la dirección de caberas y datos cedidos en el enunciado de la práctica).

Una vez contamos con este dataset filtramos seleccionando las columnas que se nos requieren, en nuestro caso han sido:

'PSSM_r1_3_E', 'PSSM_central_0_Y', 'PSSM_r2_-4_S', 'PSSM_central_-2_Q', 'PSSM_r1_-2_Y', 'PSSM_r1_0_G'.

Además, evidentemente en este caso tendremos que añadir la variable de clase que en este conjunto en concreto queda identificada con la variable *"class"*.

```
if __name__ == '__main__':
    data = read_data()
    selected_columns = ['PSSM_r1_3_E', 'PSSM_central_0_Y', 'PSSM_r2_-4_S', 'PSSM_central_-2_Q', 'PSSM_r1_-2_Y', 'PSSM_r1_0_G']
    selected_columns.append('class')
    create_new_df(data, selected_columns)
    spark_context.stop()
```

Ilustración 1 – Resumen de creación del nuevo *"dataframe"*

1.2 Balanceo de clases

El problema del desbalance de clases implica que la información no se encuentre distribuida equitativamente entre todas las clases que la componen, por lo que se generan efectos no deseados en el proceso de clasificación. Esto significa que alguna de las clases del conjunto de datos tiene una cantidad mucho mayor al resto.

Para solucionar este problema se puede optar a estrategias como *“under-sampling”* o *“over-sampling”*.

1.2.1 Under-Sampling

Eliminar aleatoriamente las muestras de la clase mayoritaria, con o sin reemplazo. Esta es una de las primeras técnicas utilizadas para aliviar el desequilibrio en el conjunto de datos, sin embargo, puede aumentar la varianza del clasificador y es muy probable que descarte muestras útiles o importantes.

```
def under_sampling(df):
    """Tecnica undersampling para balancear las clases"""
    negativos = df[df['class'] == 0]
    positivos = df[df['class'] == 1]

    porcentaje = float(positivos.count()) / float(negativos.count())

    nuevo_dataframe = negativos.sample(withReplacement=False, fraction=porcentaje, seed=2020)
    dataframe_balanceado = nuevo_dataframe.union(positivos)
```

Ilustración 2 - Método *"Under-sampling"*

1.2.2 Over-Samping

El sobremuestreo (*“over-sampiling”*) aleatorio consiste en complementar los datos de entrenamiento con múltiples copias de algunas de las clases minoritarias.

```
def over_sampling(df):
    """Tecnica oversampling para balancear las clases"""
    negativos = df[df['class'] == 0]
    positivos = df[df['class'] == 1]

    porcentaje = float(negativos.count()) / float(positivos.count())

    nuevo_dataframe = positivos.sample(withReplacement=True, fraction=porcentaje, seed=2020)
    dataframe_balanceado = nuevo_dataframe.union(negativos)

    return dataframe_balanceado
```

Ilustración 3 – Método *“Over-sampling”*

En nuestro caso hemos elaborado hemos elaborado un sencillo método tal y como puede ver en la ilustración superior.

1.3 Preprocesamiento

En este método lo que hemos hecho es preprocesar los datos para que Spark posteriormente puede comenzar las labores de entrenamiento. Mediante el uso de la librería *“pyspark”* se ha reducido a configurar una serie de parámetros simples.

1.4 Clasificadores

Podemos decir que un clasificador es un algoritmo que, recibiendo como entrada cierta información de un objeto, es capaz indicar la categoría o clase a que pertenece de entre un número acotado de clases posibles.

En nuestro caso hemos usado cuatro diferentes en todos los que ha sido posible hemos usado distintas parametrizaciones.

1.4.1 Modelos de regresión logística binario

El modelo de regresión logística binario o bimodal esta incluido en la librería *"pyspark"* y en el tan solo deberemos configurar el número de iteraciones que deseamos que se realicen así como un parámetro de regularización.

El algoritmo pasará por una serie de etapas y finalmente nos devolverá el nuevo *"dataframe"* procesado.

```
def regresion_logistica_binaria(train, test, iters, regularization):
    """Siguiendo el tutorial https://stackoverflow.com/questions/36697304/how-to-extract-model-hyper-parameters-from-spar
    lr = LogisticRegression(featuresCol='features', labelCol='label', maxIter=iters, elasticNetParam=regularization)

    grid = ParamGridBuilder().addGrid(lr.regParam, [0.1, 0.01, 0.001, 0.0001]).build()
    evaluator = BinaryClassificationEvaluator()
    cv = CrossValidator(estimator=lr, estimatorParamMaps=grid, evaluator=evaluator)
    cv_model = cv.fit(train)
    best_model = cv_model.bestModel

    best_lambda = best_model._java_obj.getRegParam()
    lr = LogisticRegression(featuresCol='scaledFeatures', labelCol='label',
                            maxIter=iters, regParam=best_lambda, elasticNetParam=regularization)
    lr_model = lr.fit(train)
    """Summary of the model and predictions"""
    predictions = lr_model.transform(test)
```

Ilustración 4 – Algoritmo de regresión logística

1.4.2 Naive Bayes

El clasificador Naive Bayes asume que el efecto de una característica particular en una clase es independiente de otras características. Por ejemplo, un solicitante de préstamo es deseable o no dependiendo de sus ingresos, historial de préstamos y transacciones anteriores, edad y ubicación.

En este caso no es configurable ningún parámetro adicional.

```
def naive_bayes(train, test):
    """Siguiendo el tutorial https://ai.plainenglish.io/build-naive-bayes-spam-classifier-on-pyspark-58aa3352e244"""
    naive_bayes_construct = NaiveBayes(modelType="multinomial", featuresCol='scaledFeatures')
    evaluador = BinaryClassificationEvaluator()
    grid = ParamGridBuilder().addGrid(naive_bayes_construct.smoothing, [0.0, 0.5, 1.0]).build()
    cross_validator = CrossValidator(estimator=naive_bayes_construct, estimatorParamMaps=grid, evaluator=evaluador)
    cross_validation_model = cross_validator.fit(train)
    best_model = cross_validation_model.bestModel
    smoothing = best_model._java_obj.getSmoothing()

    naive_bayes_mejorado = NaiveBayes(smoothing=smoothing, modelType="multinomial", featuresCol='scaledFeatures')
    naive_bayes_model = naive_bayes_mejorado.fit(train)
    predicciones = naive_bayes_model.transform(test)

    return predicciones
```

Ilustración 5 – Algoritmo de *"Naive Bayes"*

1.4.3 Decision Tree

Un árbol de decisión es un mapa de los posibles resultados de una serie de decisiones relacionadas. Permite que un individuo o una organización comparen posibles acciones entre sí según sus costos, probabilidades y beneficios.

Este clasificador una vez más viene incluido en el paquete de la librería *"pyspark"* y únicamente tendremos que parametrizar la etiqueta clasificadora, la impureza (medida en la que se basa la elección de la condición óptima), la profundidad de los árboles generados y el número de semillas.

```
def decision_tree(train, test, imp, depth):
    decision_tree = DecisionTreeClassifier(labelCol="label", featuresCol="scaledFeatures", impurity=imp, maxDepth=depth, seed=2020)
    decision_tree_model = decision_tree.fit(train)
    predicciones = decision_tree_model.transform(test)

    return predicciones
```

Ilustración 6 – Algoritmo de Árbol de decisión

1.4.4 Random Forest

Es una combinación de árboles tal que cada árbol depende de los valores de un vector aleatorio probado independientemente y con la misma distribución para cada uno de estos.

La configuración es prácticamente igual al caso anterior.

```
def random_forest(train, test, imp, depth, n_trees):
    random_forest = RandomForestClassifier(labelCol="label", featuresCol="scaledFeatures", maxDepth=depth, impurity=imp, seed=2020, numTrees=n_trees)
    random_forest_model = random_forest.fit(train)
    predicciones = random_forest_model.transform(test)

    return predicciones
```

Ilustración 7 – Algoritmo de *"Random Forest"*

1.4.5 Evaluador

Para todos los algoritmos que hemos visto en este bloque le hemos añadido una función evaluadora que determine su rendimiento. En concreto, hemos determinado los siguientes elementos a analizar:

- ***ROC***. Valor del área bajo la curva ROC donde 0 es el peor y 100 el mejor.
- ***Matriz de confusión***. Indica el número de aciertos y errores que se han obtenido, tanto para los casos positivos como para los negativos.
- ***Precisión***. Indica la precisión del clasificador que se ha evaluado.
- ***Coeficiente Kappa***. Se usa para evaluar la concordancia o reproducibilidad de instrumentos de medida cuyo resultado es categórico (2 o más categorías).

Todos estos resultados quedan almacenados en una carpeta con el nombre *"results"* y que contiene un archivo del tipo *"csv"* con los resultados de las estrategias explicadas anteriormente.

```
ccsa15408846@hadoop-master:~/cc_p4$ ls
create_new_df.py  filteredC.small.training  main.py  Makefile  original.df.balanced.classes  README.md  results  schemas
ccsa15408846@hadoop-master:~/cc_p4$ cd results/
ccsa15408846@hadoop-master:~/cc_p4/results$ ls
dt.entropy  dt.gini  nb  rf  rlb.1  rlb.2
ccsa15408846@hadoop-master:~/cc_p4/results$ cd rlb.1/
ccsa15408846@hadoop-master:~/cc_p4/results/rlb.1$ ls
part-00000-af6c5ddc-f88b-429a-8727-67d30556fa67-c000.csv  _SUCCESS
ccsa15408846@hadoop-master:~/cc_p4/results/rlb.1$ cat part-00000-af6c5ddc-f88b-429a-8727-67d30556fa67-c000.csv
ROC,Precision,Kappa,Negativos acierto,Negativos error,Positivos error,Positivos acierto
100.0,100.0,100.0,413153,0,0,206112
ccsa15408846@hadoop-master:~/cc_p4/results/rlb.1$
```

Ilustración 8 - Ejemplo de resultados

2 CONCLUSIONES

En este apartado compararemos el conjunto de algoritmos en función del algoritmo evaluador que se ha comentado en la sección anterior.

2.1 Modelos de regresión logística binario

```
ccsa15408846@hadoop-master: ~/cc_p4/results
ccsa15408846@hadoop-master:~/cc_p4/results$ cat rlb.1/part-00000-af6c5ddc-f88b-429a-8727-67d30556fa67-c000.csv
ROC,Precision,Kappa,Negativos acierto,Negativos error,Positivos error,Positivos acierto
100.0,100.0,100.0,413153,0,0,206112
ccsa15408846@hadoop-master:~/cc_p4/results$ cat rlb.2/part-00000-1890ca42-5fbe-4934-b75c-a5ee27b7524e-c000.csv
ROC,Precision,Kappa,Negativos acierto,Negativos error,Positivos error,Positivos acierto
100.0,100.0,100.0,413153,0,0,206112
ccsa15408846@hadoop-master:~/cc_p4/results$
```

Ilustración 9 – Resultados para RLB en ambos casos

En la captura anterior se puede observar cómo se ha obtenido un 100% de acierto (ningún error) tanto para el caso donde se ha parametrizado el algoritmo con un valor de regularización 0 (*"rlb.1"*) como cuando se ha utilizado un valor 1.0 (*"rbl.2"*).

2.2 Naive Bayes

```
ccsa15408846@hadoop-master: ~/cc_p4/results
ccsa15408846@hadoop-master:~/cc_p4/results$ cat nb/part-00000-1bd0e099-e4bb-46ad-a1d8-ec7c5264734b-c000.csv
ROC,Precision,Kappa,Negativos acierto,Negativos error,Positivos error,Positivos acierto
9.448,100.0,100.0,413153,0,0,206112
ccsa15408846@hadoop-master:~/cc_p4/results$
```

Ilustración 10 – Resultados para Naive Bayes

En esta ocasión podemos observar en primer lugar que no se requiere de ninguna parametrización (por lo que solo hay una modalidad de resultado) y que, aunque se ha obtenido un 100% de acierto, en este caso el valor del área bajo la curva ROC es inferior, por lo que presuponemos un resultado algo inferior, aunque mínimo, en comparación con el caso anterior.

2.3 Decision Tree

```
ccsa15408846@hadoop-master: ~/cc_p4/results
ccsa15408846@hadoop-master:~/cc_p4/results$ cat dt.entropy/part-00000-90df22a1-734a-491a-a5ec-d54db122284c-c000.csv
ROC,Precision,Kappa,Negativos acierto,Negativos error,Positivos error,Positivos acierto
100.0,100.0,100.0,413153,0,0,206112
ccsa15408846@hadoop-master:~/cc_p4/results$ cat dt.gini/part-00000-06608da3-d26a-4de4-a280-c9abb197b169-c000.csv
ROC,Precision,Kappa,Negativos acierto,Negativos error,Positivos error,Positivos acierto
100.0,100.0,100.0,413153,0,0,206112
ccsa15408846@hadoop-master:~/cc_p4/results$
```

Ilustración 11 – Resultados para *"Decision Tree"*

En este caso observamos que los resultados con completamente iguales a los obtenidos utilizando la metodología de regresión y de forma particular, para las dos parametrizaciones utilizadas.

2.4 Random Forest

```
ccsa15408846@hadoop-master: ~/cc_p4/results
ccsa15408846@hadoop-master:~/cc_p4/results$ cat rf/part-00000-3dbbe569-34a2-49e1-aa80-5f2f3d5403c3-c000.csv
ROC,Precision,Kappa,Negativos acierto,Negativos error,Positivos error,Positivos acierto
100.0,100.0,100.0,413153,0,0,206112
ccsa15408846@hadoop-master:~/cc_p4/results$
```

Ilustración 12 - Resultados para *"Random Forest"*

Igual que en el caso anterior, se han obtenido los mismos resultados que en la metodología de regresión.

Presuponemos que necesitaríamos un conjunto más grandes y diferenciado para apreciar mayores diferencias entre ambos modelos.