Artículo
Pierre-Yves Duq... · Jul 6, 2020 Lectura de 11 min

Machine Learning con Spark e InterSystems IRIS

Apache Spark se ha convertido rápidamente en una de las tecnologías más atractivas para la analítica de big data y el machine learning. Spark es un motor de procesamiento de datos generales, creado para usar con entornos de procesamiento en clúster. Su corazón es el RDD (Resilient Distributed Dataset), que representa un conjunto de datos distribuido con tolerancia a fallos, sobre el que se puede operar en paralelo entre los nodos de un clúster. Spark se implementa con una combinación de Java y Scala, por lo que viene como una biblioteca que puede ejecutarse sobre cualquier JVM. Spark también es compatible con Python (PySpark) y R (SparkR) e incluye bibliotecas para SQL (SparkSQL), machine learning (MLlib), procesamiento de gráficas (GraphX) y procesamiento de flujos (Spark Streaming).

El conector Spark de IRIS permite sacar el máximo provecho de las capacidades de la plataforma InterSystems IRIS mediante la optimización del throughput usando paralelización, y trasladando el trabajo de filtrado correspondiente a la base de datos, lo que minimiza la cantidad de datos que es necesario leer.

A continuación les presento mi intento de demostrar un "Hola mundo" de machine learning con Spark e IRIS ejecutándose localmente en un portátil. Les mostraré un par de ejemplos de machine learning(regresión lineal y clasificación naive Bayes) con Spark y una conexión a IRIS.

Una alternativa a la instalación local es hacer estas pruebas en un entorno proporcionado por InterSystems online en la plataforma de e-learning:

https://learning.intersystems.com/course/view.php?id=796

Los datos que usaremos aquí están disponibles en el repositorio github de ejemplos de InterSystems, y se pueden descargar como descrito en la documentación de producto:

https://docs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls?KEY=ASAMPLES

Aquí, solo necesitamos una copia del conjunto de datos Iris, un clásico ejemplo de flores que se usa para las demostraciones de machine learning.

Configuración de Spark e IRIS local

A continuación, los requisitos de Instalación y la configuración local de los componentes, en un servidor Windows sin nada anterior:

- Editor VSCode (Microsoft Visual Studio Core), como editor.
- Cliente Git, (git for Windows): https://gitforwindows.org/
- Java OpenJDK 1.8: https://adoptopenjdk.net/upstream.html?variant=openjdk8&jvmVariant=hotspot
- Python 3.7.7
- Apache Spark 2.4.5 pre-construido para Apache Hadoop 2.7
- WinUtils
- IRIS 2020.1 Community Edition

Configuración de Java OpenJDK 1.8

Después de descargar la versión OpenJDK 1.8 pre-compilada para Windows

(https://adoptopenjdk.net/upstream.html?variant=openjdk8&jvmVariant=hotspot) en c: \ openjdk, se tienen que definir 3 variables de entorno:

- JAVA_HOME=C: \ openjdk
- PATH=$PATH;C: \ OpenJDK \ JRE \ bin
- _JAVA_OPTIONS=-Xmx512M -Xms512M

Descarga y configuración de Python

Al usar Apache spark versión 2.4 en esta configuración, es importante notar que pyspark 2.4 aún no soporta a python 3.8, por lo cual la versión que se descarga e instala es Python 3.7.7, usando el instalador para Windows diponible en:

https://www.python.org/downloads/windows/

Descarga de Spark y WinUtils

La descarga de Spark se hace desde el sitio https://spark.apache.org/downloads.html, escogiendo la versión spark-2.4.6-bin-hadoop2.7.tgz y se instala en c: \ Spark, y a continuación, se definen estas variables de entorno:

- SPARK_HOME=C: \ spark
- PATH=$PATH;c: \ spark \ bin
- HADOOP_HOME=C: \ spark

La utilidad WinUtils.EXE para esta versión de Spark esta disponible en: https://github.com/cdarlint/winutils

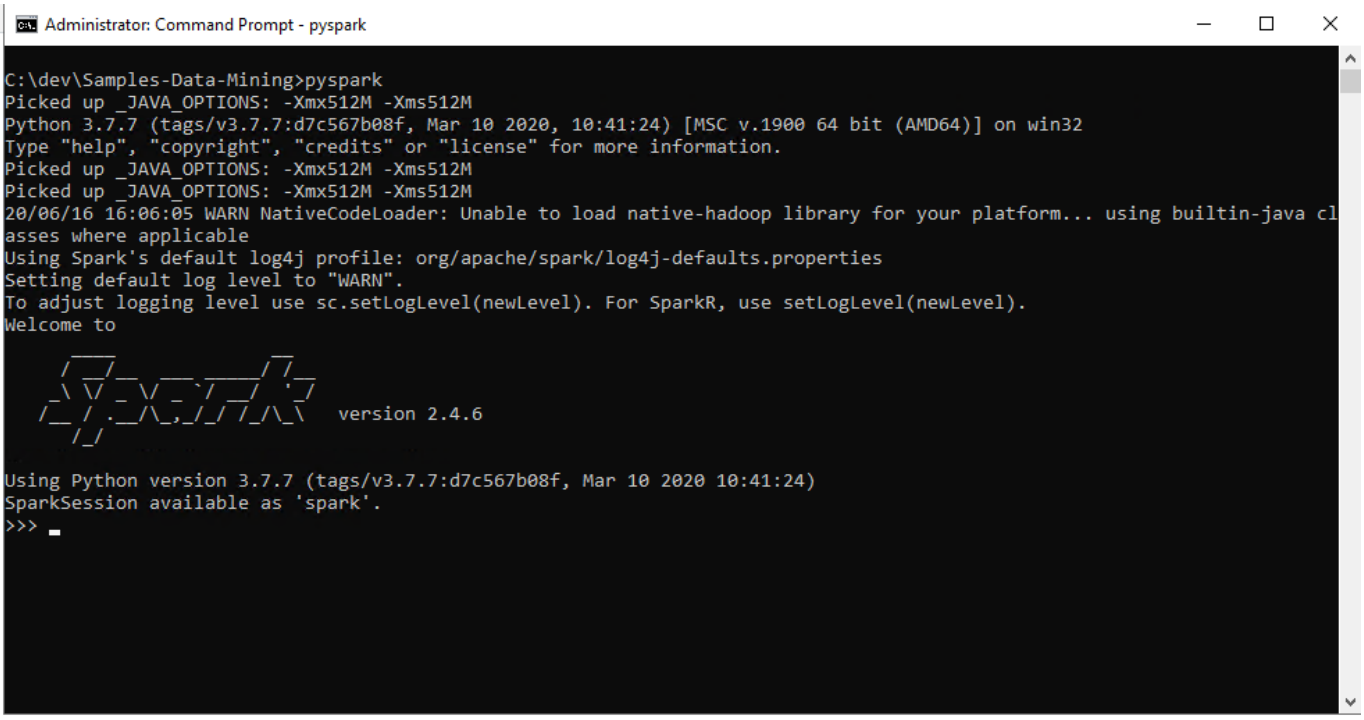
Se copia el WinUtils.EXE en c: \ Spark \ bin

Librerias Python y Validación de la instalación:

Para usar spark desde python, usaremos un editor "jupyter notebook" y la libreria findspark. Ambos se pueden instalar desde un command prompt con "pip":

```
c:\>pip install findspark
c:\>pip install jupyter
C:\>pip install matplotlib
```

Ahora se puede validar la instalación de pyspark desde la línea de comando con pyspark. Debería ver una salida como esta:



Instalación de InterSystems IRIS y carga del dataset de ejemplo

Se descarga IRIS Community("IRIS_Community-2020.1.0.215.0-win_x64") del sitio web de InterSystems y se instala en C: \ intersystems \ iris.

El dataset de ejempl se puede descargar de github con git:

c: \ dev>git clonhttps://github.com/intersystems/Samples-Data-Mining.git

Y se puede cargar en el namespace USER de IRIS desde un terminal, con una carga recursiva

```
USER>do $system.OBJ.LoadDir("c:\dev\","ck",1,1)
USER>do ##class(Build.DataMiningSample).Build()
Your input: c:\dev\Samples-Data-Mining
```

Copia de Librerías de IRIS para uso desde Spark

Para que las librerías del conector nativo spark de IRIS estén accesibles, el metodo que usamos aquí es copiar estas librerías de directorio de instalaión iris "c: y adjuntarlas al directorio c: \ spark \ jars:

intersystems-jdbc-3.1.0.jar

intersystems-spark-1.0.0.jar

El articulo y las muestras de código se han escrito en un notebook jupyter sobre IRIS. El notebook original para Caché (usa el driver JDBC que precede al conector nativo Spark) está disponible en GitHub: Machine learning con Spark y Caché. Cuando su entorno esté listo, puede ejecutar el bloc de notas y ejecutar todas las muestras de código directamente desde el mismo.

Primero usaremos findspark para hacer una prueba rápida para verificar que Spark está configurado correctamente y que podemos importarlo a nuestro entorno. Desde un prompt, se arranca con:

```
c:\dev>jupyter notebook
```

```
In [1]: #Inicilización del contexto spark
        import findspark
        findspark.init()
```

```
In [2]: #Obtención del contexto spark
        import pyspark
        sc=pyspark.SparkContext()

        #viualización de las caracteristicas del contexto spark
        sc
```

Out[2]:

SparkContext

Spark UI

Version

`v2.4.6`

Master

`local[*]`

AppName

`pyspark-shell`

Cargar y examinar datos

Ahora crearemos una instancia de SparkSession y la usaremos para conectarnos a IRIS. SparkSession es el punto de partida para usar Spark. Lo usaremos para cargar el conjunto de datos Iris en un DataFrame de Spark. El DataFrame de Spark extiende la funcionalidad del RDD Spark original (discutido antes). Además de muchas optimizaciones, el DataFrame agrega la capacidad de acceder y manipular datos tanto a través de una interfaz de estilo SQL como con una lista de objetos.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local")
spark = SparkSession.builder.config('spark.sql.warehouse.dir','file:///C:/spark/temp'
)
spark = SparkSession.builder.getOrCreate()
irisdf = spark.read.format("com.intersystems.spark") \
    .option("url","IRIS://localhost:51773/USER") \
    .option("user","_system") \
    .option("password","SYS") \
    .option("spark.sql.warehouse.dir","file:///C:/spark/temp") \
    .option("dbtable","DataMining.IrisDataset").load()
```

Aquí podemos ejecutar un comando para mostrar las primeras 10 filas de datos de Iris como una tabla.

```
irisdf.show(10)
```

```
+---+-----------+----------+-----------+----------+-----------+
| ID|PetalLength|PetalWidth|SepalLength|SepalWidth|    Species|
+---+-----------+----------+-----------+----------+-----------+
|  1|        1.4|       0.2|        5.1|       3.5|Iris-setosa|
|  2|        1.4|       0.2|        4.9|       3.0|Iris-setosa|
|  3|        1.3|       0.2|        4.7|       3.2|Iris-setosa|
|  4|        1.5|       0.2|        4.6|       3.1|Iris-setosa|
|  5|        1.4|       0.2|        5.0|       3.6|Iris-setosa|
|  6|        1.7|       0.4|        5.4|       3.9|Iris-setosa|
|  7|        1.4|       0.3|        4.6|       3.4|Iris-setosa|
|  8|        1.5|       0.2|        5.0|       3.4|Iris-setosa|
|  9|        1.4|       0.2|        4.4|       2.9|Iris-setosa|
| 10|        1.5|       0.1|        4.9|       3.1|Iris-setosa|
+---+-----------+----------+-----------+----------+-----------+
only showing top 10 rows
```

Por cierto, un sépalo es una hoja, generalmente verde, que sirve para proteger a una flor en su etapa de capullo, y luego para soportar físicamente la flor cuando se abre.

Podemos realizar una variedad de operaciones tipo SQL, por ejemplo encontrar el número de filas en las que PetalLength es mayor a 6.0, o encontrar los recuentos de las distintas especies:

```
irisdf.filter(irisdf["PetalLength"]>6.0).show()
irisdf.groupBy("Species").count().show()
```

ID	PetalLength	PetalWidth	SepalLength	SepalWidth	Species
106	6.6	2.1	7.6	3.0	Iris-virginica
108	6.3	1.8	7.3	2.9	Iris-virginica
110	6.1	2.5	7.2	3.6	Iris-virginica
118	6.7	2.2	7.7	3.8	Iris-virginica
119	6.9	2.3	7.7	2.6	Iris-virginica
123	6.7	2.0	7.7	2.8	Iris-virginica
131	6.1	1.9	7.4	2.8	Iris-virginica
132	6.4	2.0	7.9	3.8	Iris-virginica
136	6.1	2.3	7.7	3.0	Iris-virginica
256	6.6	2.1	7.6	3.0	Iris-virginica
258	6.3	1.8	7.3	2.9	Iris-virginica
260	6.1	2.5	7.2	3.6	Iris-virginica
268	6.7	2.2	7.7	3.8	Iris-virginica
269	6.9	2.3	7.7	2.6	Iris-virginica
273	6.7	2.0	7.7	2.8	Iris-virginica
281	6.1	1.9	7.4	2.8	Iris-virginica
282	6.4	2.0	7.9	3.8	Iris-virginica
286	6.1	2.3	7.7	3.0	Iris-virginica

Species	count
Iris-virginica	100
Iris-setosa	100
Iris-versicolor	100

Estas son las primeras 10 filas que se muestran como lista de Python de objetos de fila Spark:

```
irisdf.head(10)?
```

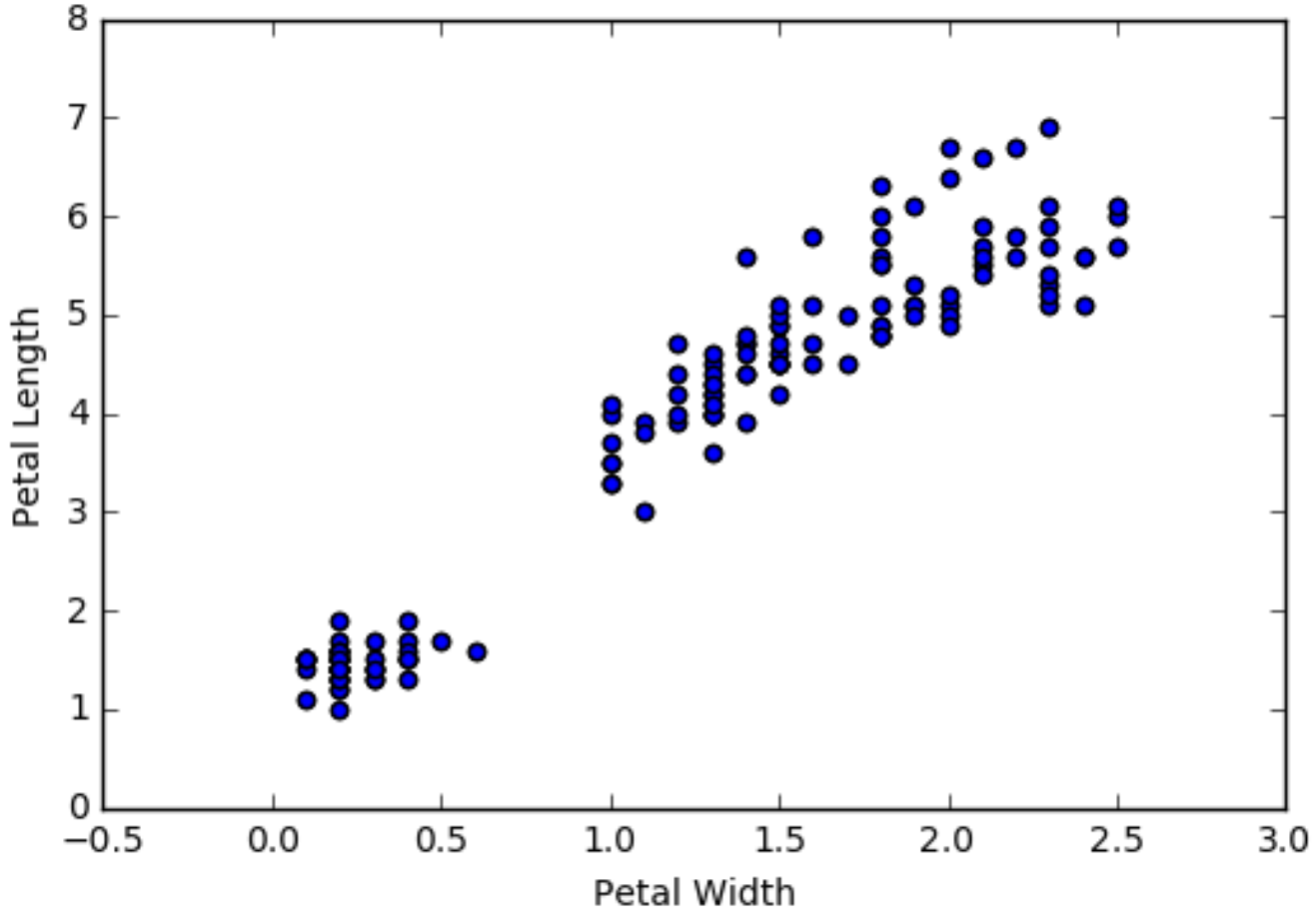
```
[Row(ID=1, PetalLength=1.4, PetalWidth=0.2, SepalLength=5.1, SepalWidth=3.5, Species=u'Iris-setosa'),
 Row(ID=2, PetalLength=1.4, PetalWidth=0.2, SepalLength=4.9, SepalWidth=3.0, Species=u'Iris-setosa'),
 Row(ID=3, PetalLength=1.3, PetalWidth=0.2, SepalLength=4.7, SepalWidth=3.2, Species=u'Iris-setosa'),
 Row(ID=4, PetalLength=1.5, PetalWidth=0.2, SepalLength=4.6, SepalWidth=3.1, Species=u'Iris-setosa'),
 Row(ID=5, PetalLength=1.4, PetalWidth=0.2, SepalLength=5.0, SepalWidth=3.6, Species=u'Iris-setosa'),
 Row(ID=6, PetalLength=1.7, PetalWidth=0.4, SepalLength=5.4, SepalWidth=3.9, Species=u'Iris-setosa'),
 Row(ID=7, PetalLength=1.4, PetalWidth=0.3, SepalLength=4.6, SepalWidth=3.4, Species=u'Iris-setosa'),
 Row(ID=8, PetalLength=1.5, PetalWidth=0.2, SepalLength=5.0, SepalWidth=3.4, Species=u'Iris-setosa'),
 Row(ID=9, PetalLength=1.4, PetalWidth=0.2, SepalLength=4.4, SepalWidth=2.9, Species=u'Iris-setosa'),
 Row(ID=10, PetalLength=1.5, PetalWidth=0.1, SepalLength=4.9, SepalWidth=3.1, Species=u'Iris-setosa')]
```

El siguiente código accede a los datos de Iris a través de la interfaz de lista para crear un par de conjuntos

que podamos usar con la biblioteca de creación de gráficas matplotlib. Lamentablemente Spark no tiene su propia biblioteca de creación de gráficas. El código crea un diagrama de dispersión que muestra PetalLength vs PetalWidth.

```
import matplotlib.pyplot as plt
#Recuperar una matriz de objetos de fila desde el DataFrame
items = irisdf.collect()
petal_length = []
petal_width = []

for item in items:
    petal_length.append(item['PetalLength'])
    petal_width.append(item['PetalWidth'])
plt.scatter(petal_width,petal_length)
plt.xlabel("Petal Width")
plt.ylabel("Petal Length")
plt.show()
```



Entrenamiento y prueba de un modelo de regresión lineal

Parece haber una relación lineal bastante fuerte entre PetalWidth y PetalLength. Supongo que eso no puede sorprender a nadie. Investiguemos la relación más de cerca con la biblioteca de machine learning de Spark. Entrenaremos un modelo simple de regresión lineal para que ajuste una línea a través de los datos. Cuando tengamos el modelo, podremos usarlo para predecir el largo de un pétalo de Iris en base a su ancho.

Este es un esquema de los pasos del siguiente código:

1. Crear un nuevo DataFrame y transformar la columna PetalWidth o "features" en el vector que requiere la biblioteca de Spark.

2. Dividir los datos de Iris al azar entre un conjunto de entrenamiento (70%) y uno de prueba (30%).
3. Usar los datos de entrenamiento para ajustar un modelo de regresión lineal, el machine learning.
4. Pasar los datos de prueba por el modelo y mostrar el resultado.

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
# Transformar la(s) columna(s) de "Características" (Features) al formato vectorial c
orrecto
df = irisdf.select('PetalLength','PetalWidth')
vectorAssembler = VectorAssembler(inputCols=["PetalWidth"],
                                  outputCol="features")
data=vectorAssembler.transform(df)
# Dividir los datos entre conjuntos de entrenamiento y prueba.
trainingData,testData = data.randomSplit([0.7, 0.3], 0.0)
#Configurar el modelo.
lr = LinearRegression().setFeaturesCol("features").setLabelCol(
"PetalLength").setMaxIter(10)
# Entrenar el modelo con los datos de entrenamiento.
lrm = lr.fit(trainingData)
# Aplicar el modelo a los datos de prueba y mostrar sus predicciones de largo de péta
lo (PetalLength).
predictions = lrm.transform(testData)
predictions.show()
```

```
+-----------+----------+--------+------------------+
|PetalLength|PetalWidth|features|        prediction|
+-----------+----------+--------+------------------+
|        1.0|       0.2|   [0.2]|1.5653385837963045|
|        1.1|       0.1|   [0.1]|1.3456666610929295|
|        1.2|       0.2|   [0.2]|1.5653385837963045|
|        1.2|       0.2|   [0.2]|1.5653385837963045|
|        1.2|       0.2|   [0.2]|1.5653385837963045|
|        1.3|       0.3|   [0.3]|1.7850105064996793|
|        1.4|       0.1|   [0.1]|1.3456666610929295|
|        1.4|       0.1|   [0.1]|1.3456666610929295|
|        1.4|       0.2|   [0.2]|1.5653385837963045|
|        1.4|       0.2|   [0.2]|1.5653385837963045|
|        1.4|       0.2|   [0.2]|1.5653385837963045|
|        1.4|       0.2|   [0.2]|1.5653385837963045|
|        1.4|       0.3|   [0.3]|1.7850105064996793|
|        1.4|       0.3|   [0.3]|1.7850105064996793|
|        1.5|       0.1|   [0.1]|1.3456666610929295|
|        1.5|       0.1|   [0.1]|1.3456666610929295|
|        1.5|       0.1|   [0.1]|1.3456666610929295|
|        1.5|       0.2|   [0.2]|1.5653385837963045|
|        1.5|       0.2|   [0.2]|1.5653385837963045|
|        1.5|       0.2|   [0.2]|1.5653385837963045|
+-----------+----------+--------+------------------+
only showing top 20 rows
```

La columna de predicción muestra el largo de pétalo pronosticado por el modelo. Podemos compararlo con

los valores reales de la columna PetalLength.

Para evaluar el modelo, la siguiente parte del código calcula el error cuadrático medio (RMSE) para sus predicciones sobre los datos de prueba. Esto brinda una medida de la precisión del modelo. El código también recupera la pendiente e intersección con el eje vertical de la línea de regresión. Usaremos esto para agregar la línea de regresión a nuestro diagrama de dispersión anterior.

```
from pyspark.ml.evaluation import RegressionEvaluator
# recuperar la pendiente e intersección con el eje vertical de la línea de regresión del modelo.
slope = lrm.coefficients[0]
intercept = lrm.intercept
print("slope of regression line: %s" % str(slope))
print("y-intercept of regression line: %s" % str(intercept))
# Seleccionar (predicción, etiqueta verdadero) y calcular el error de la prueba
evaluator = RegressionEvaluator(
    labelCol="PetalLength", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
slope of regression line: 2.19671922703
y-intercept of regression line: 1.12599473839
Root Mean Squared Error (RMSE) on test data = 0.401258
```

En base a este valor de RMSE, no es totalmente claro para mí qué tan bien nuestro modelo logró predecir el largo de pétalo. Podemos comparar el error con el valor promedio de PetalLength para quizás darnos una idea de lo significativo del error.

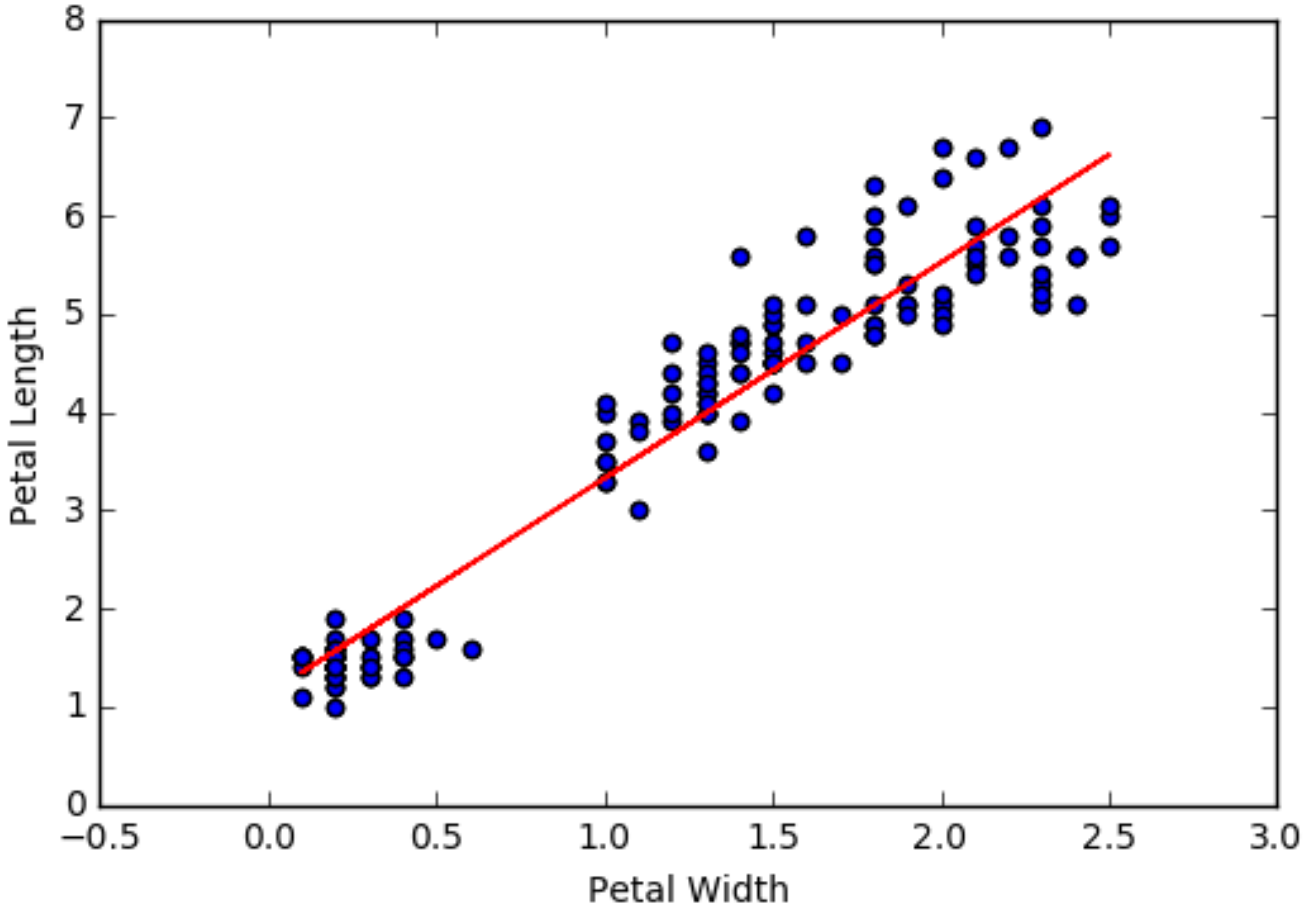
```
iris.describe(["PetalLength"]).show()
```

```
+-------+------------------+
|summary|       PetalLength|
+-------+------------------+
|  count|               300|
|   mean|3.758666666666666|
| stddev|1.761467412995684|
|    min|               1.0|
|    max|               6.9|
+-------+------------------+
```

Finalmente, para visualizar el modelo, agregaremos la línea de regresión determinada por la pendiente e intersección con el eje vertical anteriores a nuestro diagrama de dispersión original.

```
import matplotlib.pyplot as plt
items = irisdf.collect()
petal_length = []
petal_width = []
petal_features = []
for item in items:
    petal_length.append(item['PetalLength'])
    petal_width.append(item['PetalWidth'])
fig, ax = plt.subplots()
ax.scatter(petal_width,petal_length)
plt.xlabel("Petal Width")
```

```
plt.ylabel("Petal Length")
y = [slope*x+intercept for x in petal_width]
ax.plot(petal_width, y, color='red')
plt.show()
```



Entrenamiento y prueba de un modelo de clasificación

Los datos de Iris contienen tres especies distintas de Iris: Iris-Setosa, Iris-Verisicolor y Iris-Virginica. Podemos entrenar un modelo para que clasifique o prediga a qué especie pertenece una flor en base a sus características: PetalLength, PetalWidth, SepalLength y SepalWidth. Spark admite varios algoritmos de clasificación distintos. El siguiente código usa el algoritmo Naive Bayes, uno de los algoritmos más simples, a pesar de lo cual es muy potente.

Este es un esquema de los pasos a seguir:

1. Preparar los datos para el modelo. Esto implica poner las características en forma de vector. También implica indexar las clases, sustituyendo "Iris-Setosa" por 0.0, "Iris-verisicolor" por 1.0 y "Iris-Virginica" por 2.0.
2. Dividir los datos de Iris al azar entre un conjunto de entrenamiento (70%) y uno de prueba (30%).
3. Entrenar el clasificador con los datos de entrenamiento.
4. Pasar los datos de prueba por el modelo para generar clasificaciones pronosticadas.
5. Desindexar las predicciones para poder ver los nombres de las especies en lugar de los índices en la salida.
6. Mostrar las especies reales y las pronosticadas, lado a lado.

```
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.feature import StringIndexer,IndexToString
# Para preparar los datos, indexar las clases y colocar las características en un vec
tor.
```

```
speciesIndexer = StringIndexer(inputCol="Species", outputCol="speciesIndex")
vectorAssembler = VectorAssembler(inputC
ols=["PetalWidth","PetalLength","SepalWidth","SepalLength"],
                                  outputCol="features")
data = vectorAssembler.transform(irisdf)
index_model = speciesIndexer.fit(data)
data_indexed = index_model.transform(data)
# Dividir los datos entre conjuntos de entrenamiento y prueba.
trainingData, testData =  data_indexed.randomSplit([0.7, 0.3],0.0)
# Configurar el clasificador y luego entrenarlo con el conjunto de entrenamiento.
nb = NaiveBayes().setFeaturesCol("features").setLabelCol("speciesIndex"
).setSmoothing(1.0).setModelType("multinomial")
model = nb.fit(trainingData)
# Pasar el conjunto de prueba por el clasificador
classifications = model.transform(testData)
# Desindexar los datos para ver los nombres de las especies en lugar de los números d
e índice en la salida.
converter = IndexToString(inputCol="prediction", outputCol=
"PredictedSpecies", labels=index_model.labels)
converted = converter.transform(classifications)
# Mostrar las especies reales y las pronosticadas, lado a lado
converted.select(['Species','PredictedSpecies']).show(45)
```

```
+---------------+----------------+
|        Species|PredictedSpecies|
+---------------+----------------+
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|    Iris-setosa|     Iris-setosa|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor|  Iris-virginica|
|Iris-versicolor|  Iris-virginica|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
|Iris-versicolor| Iris-versicolor|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica| Iris-versicolor|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
| Iris-virginica|  Iris-virginica|
+---------------+----------------+
only showing top 45 rows
```

Puede ver que el clasificador no fue perfecto. En el subconjunto de datos de arriba, clasificó erróneamente dos de las Iris-Verisicolor y una de las Iris-Virginica's. Podemos usar un evaluador para calcular la precisión exacta del clasificador con los datos de prueba.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
# calcular la precisión con el conjunto de prueba
evaluator = MulticlassClassificationEva
luator(labelCol="speciesIndex", predictionCol="prediction",
                                                metricName="accuracy")
accuracy = evaluator.evaluate(classifications)
print("Test set accuracy = " + str(accuracy))
```

```
Test set accuracy = 0.936708860759
```

Si esta precisión no es suficiente, podemos ajustar algunos parámetros del modelo o incluso probar con una algoritmo de clasificación totalmente distinto

#Analítica #Big Data #Inteligencia Artificial #JDBC #Machine learning #Python #InterSystems IRIS

URL de fuente: https://es.community.intersystems.com/post/machine-learning-con-spark-e-intersystems-iris