

Article

[David E Nelson](#) · Mar 9, 2017 9m read

Machine Learning with Spark and Caché

[Apache Spark](#) has rapidly become one of the most exciting technologies for big data analytics and machine learning. Spark is a general data processing engine created for use in clustered computing environments. Its heart is the Resilient Distributed Dataset (RDD) which represents a distributed, fault tolerant, collection of data that can be operated on in parallel across the nodes of a cluster. Spark is implemented using a combination of Java and Scala and so comes as a library that can run on any JVM. Spark also supports Python (PySpark) and R (SparkR) and includes libraries for SQL (SparkSQL), machine learning (MLlib), graph processing (GraphX), and stream processing (Spark Streaming).

When the Caché Spark connector is released, Spark-based applications will be able to make full use of Caché as an open analytics platform. That means taking full advantage of the capabilities of the underlying Caché database by optimizing throughput through parallelization and pushing down appropriate filtering work to the database, minimizing the amount of data that needs to be read. Today, using a plain JDBC connection to Caché, we can already begin using Spark with Caché and then transparently upgrade to the new connector when it becomes available.

The following is my attempt to demonstrate a machine learning "Hello World" using Spark and Caché running locally on my laptop. I show a couple of machine learning examples (linear regression and naive Bayes classification) using PySpark and a JDBC connection to Caché. Conveniently, Caché's SAMPLES namespace contains a copy of the [Iris dataset](#), a classic for machine learning demonstrations.

Setting up Spark and Caché on My Laptop

To install Spark locally on my laptop, I essentially followed the steps outlined here: [How to run Apache Spark on Windows 7 in Standalone Mode](#).

Here are some details for my Spark setup:

- Java 1.8
- Note: I did not install Scala separately as described in the above article. The pre-built Spark libraries I installed already contained the Scala executables and jars.
- Python 2.7.12. I installed this as part of the Anaconda 4.2 package.
- Spark version 2.6. Prebuilt package for Hadoop.
- findspark python package. I installed this using pip install findspark. This package uses the SPARK_HOME environment variable to locate the Spark installation. This makes it easier to import Spark into python code.
- Environment variables:
 - SPARK_HOME: pointing to the spark installation directory.
 - HADOOP_HOME: pointing to the Hadoop installation directory. Note that winutils.exe must be installed in the bin subdirectory. The above article discusses this dependency and provides a link for downloading the file.
- Directories: c:\spark\temp

Here is my Caché setup:

- Caché 2016.2.1
- I configured the following environment variable so that python would be able to find the Caché jdbc driver:
 - SPARK_CLASSPATH: cache-install-dir\dev\lib\java\lib\jdk18\cachejdbc.jar
- To prepare the Iris dataset, I ran the following Caché command in the SAMPLES namespace:

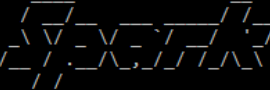
```
SAMPLES>Do ##class(DataMining.IrisDataset).load()
```

You can execute the following code samples from the command line. Open your [spark installation]/bin directory and enter pyspark. You should see output like the following:

```
C:\Spark\spark-2.0.0-bin-hadoop2.6\bin>pyspark
Python 2.7.12 |Anaconda 4.2.0 (64-bit)| (default, Jun 29 2016, 11:07:13) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
17/02/24 09:25:14 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
asses where applicable
17/02/24 09:25:14 WARN SparkConf:
SPARK_CLASSPATH was detected (set to 'C:\InterSystems\Ensemble\dev\java\lib\JDK18\cachejdbc.jar').
This is deprecated in Spark 1.0+.

Please instead use:
- ./spark-submit with --driver-class-path to augment the driver classpath
- spark.executor.extraClassPath to augment the executor classpath

17/02/24 09:25:14 WARN SparkConf: Setting 'spark.executor.extraClassPath' to 'C:\InterSystems\Ensemble\dev\java\lib\JDK1
8\cachejdbc.jar' as a work-around.
17/02/24 09:25:14 WARN SparkConf: Setting 'spark.driver.extraClassPath' to 'C:\InterSystems\Ensemble\dev\java\lib\JDK18\
cachejdbc.jar' as a work-around.
Welcome to

 version 2.0.0
```

I wrote this entire article, including all the code samples, in a jupyter notebook. You can get this on GitHub: [Machine Learning with Spark and Caché](#). Once your environment is ready you can launch the notebook and execute all the code samples directly from within it.

First we'll use findspark to do a quick test to verify that Spark is correctly configured and that we can import it into our environment.

```
import findspark
findspark.init()
import pyspark
sc = pyspark.SparkContext()
# If the Spark context was created, we should see output that looks something like th
e following.
sc
```

```
<pyspark.context.SparkContext at 0x2d8ceb8>
```

Loading and Examining Some Data

Next we will create a SparkSession instance and use it to connect to Caché. SparkSession is the starting point for using Spark. We will use it to load the Iris dataset into a Spark DataFrame. The Spark DataFrame extends the functionality of the original Spark RDD (discussed above). In addition to many optimizations, the DataFrame adds the ability to access and manipulate data through both a relational sql-like interface and a list of objects.

```
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()
iris = spark.read.format("jdbc").option("url", "jdbc:Cache://localhost:1972/SAMPLES").
```

```
option("driver",  
"com.intersys.jdbc.CacheDriver").option("dbtable","DataMining.IrisDataset").option("u  
ser",  
"_System").option("password","SYS").option("spark.sql.warehouse.dir","file:///C:/spar  
k/temp").load()
```

Here we can run a command to display the first 10 rows of Iris data as a table.

```
iris.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

By the way, a sepal is a leaf, usually green, that serves to protect a flower in its bud stage and then physically support the flower when it blooms.

We can do a variety of SQL-like operations, for example finding the number rows where PetalLength is greater than 6.0 or finding the counts of the different species:

```
iris.filter(iris["PetalLength"]>6.0).show()  
iris.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

Here are the first 10 rows displayed as a python list of Spark Row objects:

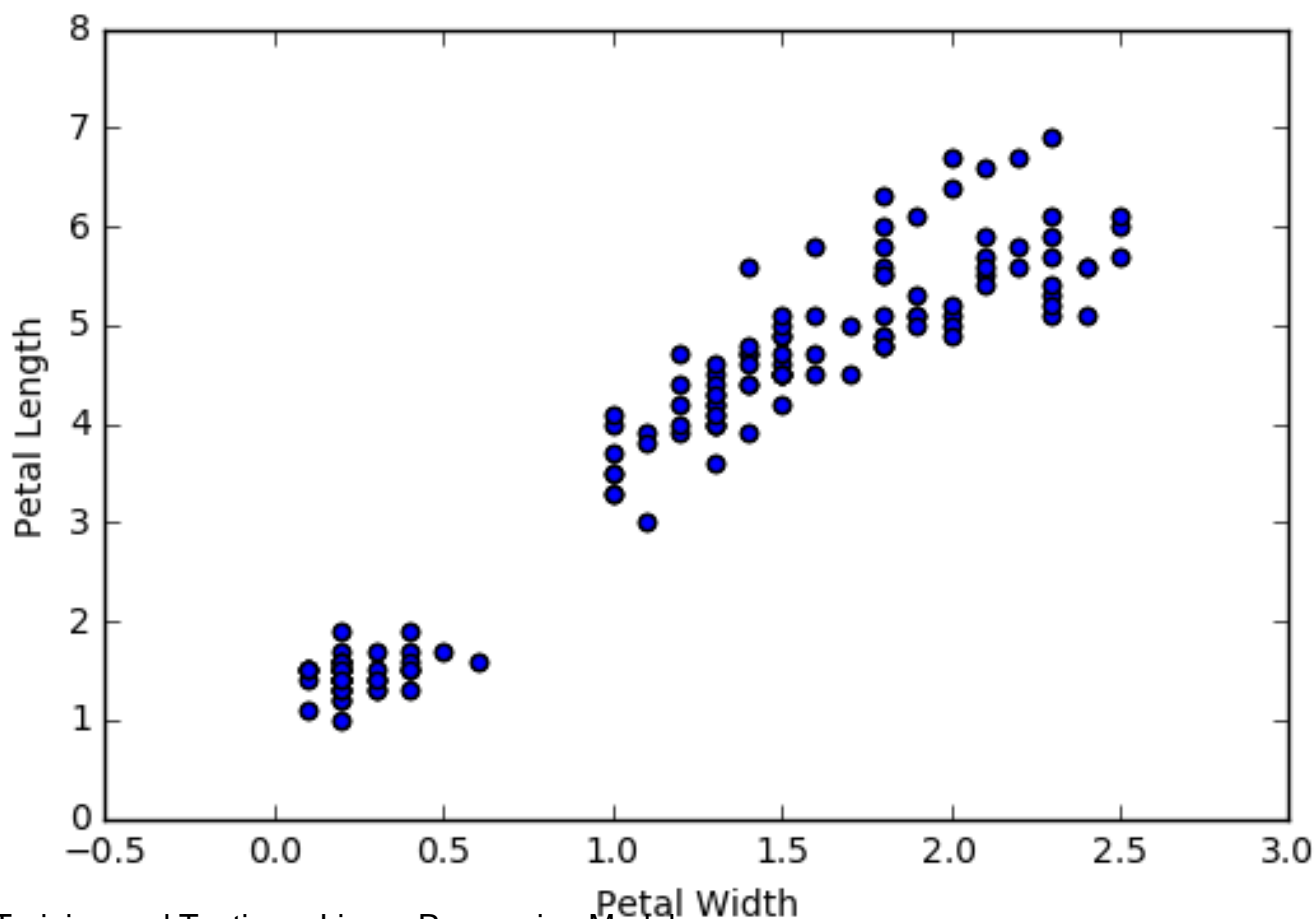
```
iris.head(10)?
```

```
[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')]
```

The following code accesses the Iris data through the list interface to create a pair of arrays to use with the matplotlib charting library. Spark unfortunately does not have its own charting library. The code creates a scatter plot showing PetalLength vs. PetalWidth.

```
import matplotlib.pyplot as plt
```

```
#Retrieve an array of row objects from the DataFrame
items = iris.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()
```



Training and Testing a Linear Regression Model

Looks like there is a pretty strong linear relationship between PetalWidth and PetalLength. I suppose that's not surprising. Let's investigate the relationship more closely using Spark's machine learning library. We will train a simple linear regression model to fit a line through the data. Once we have the model we can use it to predict the length of an Iris petal based on its width.

Here is an outline of the steps in the following code:

1. Create a new DataFrame and transform the PetalWidth or "features" column into the vector needed by the Spark library.
2. Randomly divide the Iris data into training (70%) and test (30%) sets.
3. Use the training data to fit a linear regression model, the actual machine learning.
4. Run the test data through the model and display the result.

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
# Transform the "Features" column(s) into the correct vector format
df = iris.select('PetalLength','PetalWidth')
```

```
vectorAssembler = VectorAssembler(inputCols=["PetalWidth"],
                                  outputCol="features")
data=vectorAssembler.transform(df)
# Split the data into training and test sets.
trainingData,testData = data.randomSplit([0.7, 0.3], 0.0)
# Configure the model.
lr = LinearRegression().setFeaturesCol("features").setLabelCol("PetalLength").setMaxI
ter(10)
# Train the model using the training data.
lrm = lr.fit(trainingData)
# Run the test data through the model and display its predictions for 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

The prediction column shows the petal length predicted by the model. We can compare it to the actual values in the PetalLength column.

The next piece of code evaluates the model by calculating the root mean squared error (RMSE) for its predictions on the test data. This provides one measure of the model's accuracy. The code also retrieves the slope and y-intercept of the regression line. We will use these to add the regression line to our earlier scatter plot.

```
from pyspark.ml.evaluation import RegressionEvaluator
# retrieve the slope and y-intercepts of the regression line from the model.
slope = lrm.coefficients[0]
```



```

intercept = lrm.intercept
print("slope of regression line: %s" % str(slope))
print("y-intercept of regression line: %s" % str(intercept))
# Select (prediction, true label) and compute test error
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

```

Based on this RMSE value it is not perfectly clear to me how well our model does at predicting petal length. We can compare the error to the average PetalLength value and perhaps get some sense of the error's significance.

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

```

+-----+-----+
|summary|      PetalLength|
+-----+-----+
|  count|              300|
|   mean| 3.75866666666666|
| stddev|1.761467412995684|
|   min|              1.0|
|   max|              6.9|

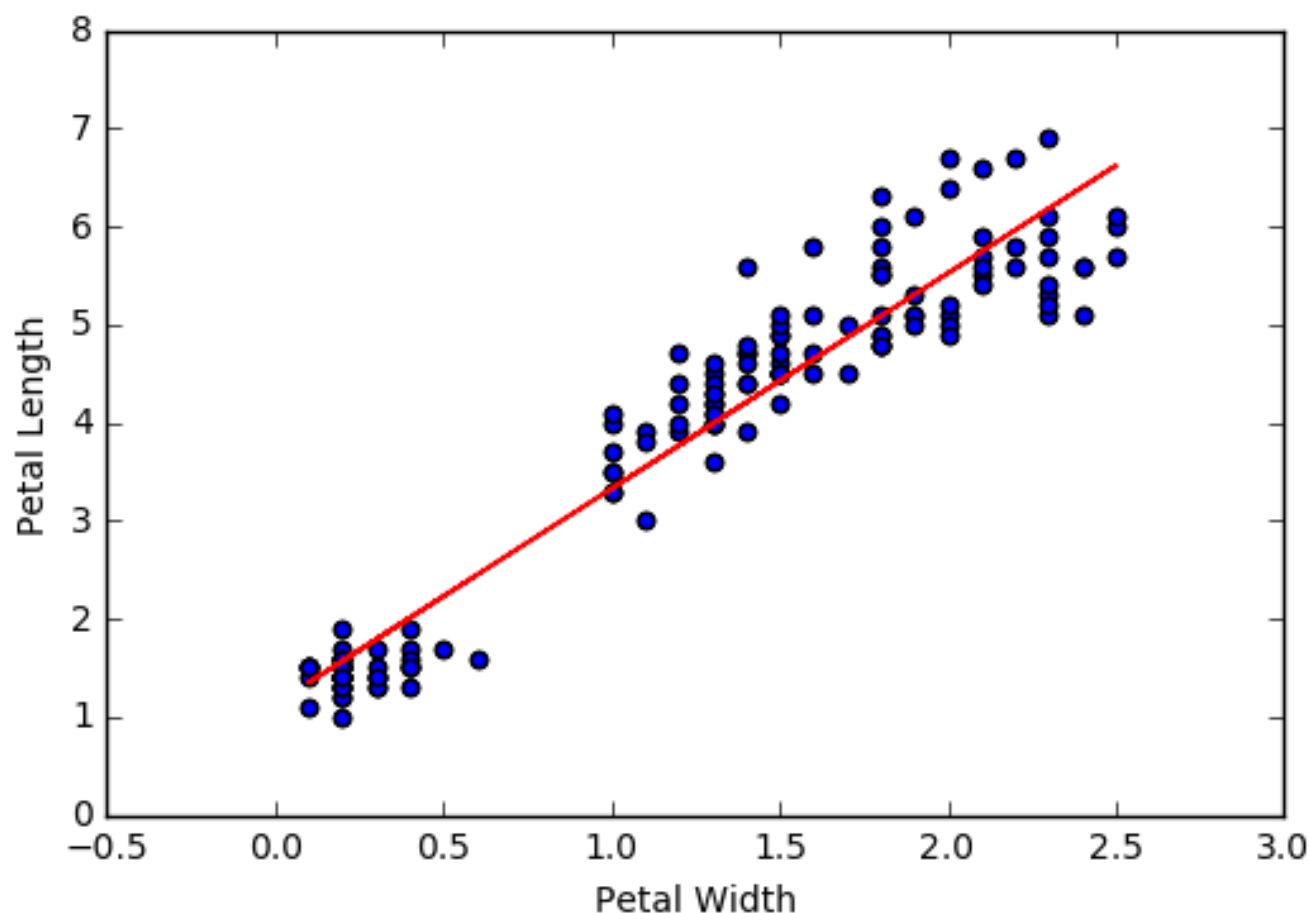
```

Finally, let's visualize the model by adding the regression line determined by the above slope and intercept to our original scatter plot.

```

import matplotlib.pyplot as plt
items = iris.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()

```



Training and Testing a Classification Model

The Iris data contains three different species of Iris: Iris-Setosa, Iris-Verisicolor, and Iris-Virginica. We can train a model to classify or predict which species a flower belongs to based on its features: PetalLength, PetalWidth, SepalLength, and SepalWidth. Spark supports several different classification algorithms. The following code uses the Naive Bayes algorithm, one of the simpler yet still very powerful classification algorithms.

Here is an outline of the steps:

1. Prepare the data for the model. This involves putting the features into a vector. It also involves indexing the classes, replacing "Iris-Setosa" with 0.0, "Iris-verisicolor" with 1.0, and "Iris-Virginica" with 2.0.
2. Randomly divide the Iris data into training (70%) and test (30%) sets.
3. Train the classifier on the training data.
4. Run the test data through the model to generate predicted classifications
5. Un-index the predictions so we can see the species names rather than the indexes in the output.
6. Display the actual and predicted species side-by-side.

```
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.feature import StringIndexer, IndexToString
# Prepare the data by indexing the classes and putting the features into a vector.
speciesIndexer = StringIndexer(inputCol="Species", outputCol="speciesIndex")
vectorAssembler = VectorAssembler(inputCols=["PetalWidth", "PetalLength", "SepalWidth",
"SepalLength"],
                                outputCol="features")
data = vectorAssembler.transform(iris)
index_model = speciesIndexer.fit(data)
data_indexed = index_model.transform(data)
# Split the data into training and test sets.
trainingData, testData = data_indexed.randomSplit([0.7, 0.3], 0.0)
# Configure the classifier and then train it using the training set.
nb = NaiveBayes().setFeaturesCol("features").setLabelCol("speciesIndex").setSmoothing
```



```
(1.0).setModelType("multinomial")
model = nb.fit(trainingData)
# Run the classifier on the test set
classifications = model.transform(testData)
# Un-index the data so we have the species names rather than the index numbers in our
  output.
converter = IndexToString(inputCol="prediction", outputCol="PredictedSpecies", labels
=index_model.labels)
converted = converter.transform(classifications)
# Display the actual and predicted species side-by-side
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-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

You can see that the classifier was not perfect. In the above subset of the data it misclassified two of the Iris-Verisicolor and one of the Iris-Virginica's. We can use an evaluator to calculate the exact accuracy of the classifier on the test data.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
# compute accuracy on the test set
evaluator = MulticlassClassificationEvaluator(labelCol="speciesIndex", predictionCol=
"prediction",
                                              metricName="accuracy")
accuracy = evaluator.evaluate(classifications)
print("Test set accuracy = " + str(accuracy))
```

Test set accuracy = 0.936708860759

If this accuracy is not sufficient, we could tune some parameters of the model or even try an entirely different classification algorithm

[#AI](#) [#Analytics](#) [#Big Data](#) [#JDBC](#) [#Machine Learning](#) [#Python](#) [#Caché](#)

Source URL: <https://community.intersystems.com/post/machine-learning-spark-and-cach%C3%A9>