PySpark ML Pipeline

The basis for constructing ML models in Spark

Spark ML Pipeline

- Like Python's pipeline
- Simplifies data transformation and modeling
- Simplifies running multiple models or datasets
  - E.g., Using Cross Validation

Python provides an API to
aid in coding machine learning (ML) models

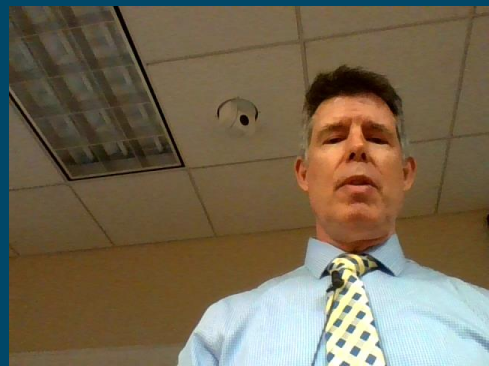


Illustration from Python Scikit-learn

same steps, but simplified

No Pipeline

```
vect = CountVectorizer()
tfidf = TfidfTransformer()
clf = SGDClassifier()

vX = vect.fit_transform(Xtrain)
tfidfX = tfidf.fit_transform(vX)
predicted = clf.fit_predict(tfidfX)

# Now evaluate all steps on test set
vX = vect.fit_transform(Xtest)
tfidfX = tfidf.fit_transform(vX)
predicted = clf.fit_predict(tfidfX)
```

Pipeline

```
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', SGDClassifier()),
])
predicted =
pipeline.fit(Xtrain).predict(Xtrain)
# Now evaluate all steps on test set
predicted = pipeline.predict(X
```

https://stackoverflow.com/questions/33091376/python-what-is-exactly-sklearn-pipeline-pipeline

Spark ML Pipeline

- Inspired by Python's Scikit-learn ML API

- A Spark ML Pipeline chains multiple Transformers and Estimators together to specify an ML workflow
  - Pipeline is a class that is part of the ML API for simplifying the specification of programming steps need to do ML modeling

https://spark.apache.org/docs/latest/ml-pipeline.html#how-it-works

A Pipeline simplifies software development

- Fewer lines of code with Pipeline
  - Faster to write
  - Easier to understand and maintain
- Pipeline is a programming object
  - Code can change it, and thus it simplifies creating ML models dynamically
  - It can be passed to functions like CrossValidate, which automatically modify parameters, which simplifies running multiple models (Hypertuning)

Fit() and Transform() methods

- Fit
  - Reviews the data according to the specified pipeline elements
  - Creates a model (for predictions) and transforms the DataFrame to have features and label columns
  - Often applied to training DataFrame
- Transform
  - Using the created model, processes a DataFrame having features and label columns, to create a new prediction column
  - Often applied to test DataFrame

Two object types in a pipeline

- A Transformer implements a transform() method, which converts one DataFrame into another, by appending one or more columns.
  - A feature transformer might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new DataFrame with the mapped column appended
- An Estimator implements a fit() method, which accepts a DataFrame and produces a Model (which is a Transformer)
  - LogisticRegression is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model (Transformer)

https://spark.apache.org/docs/latest/ml-pipeline.html#how-it-works

Spark ML Pipeline execution

- Objects in the pipeline are
  - Ordered
  - of type Transform or Estimator
- When pipeline.transform() is called, each of its objects are called
  - If the object is a Transformation, then call transform()
  - If the object is an Estimator, then call fit(), which produces a Model. Then, call transform() on that model
- Without a pipeline, your code does the same
  - You create Transforms and Estimators
  - Then you transform() or fit().transform() on each

Feature construction & model building is the same

- With or without pipeline, the steps are the same
  - Specify (feature) Transformations (e.g., Tokenizer) and Estimators (e.g., LinearRegression)
  - For each of these objects
    - If the object is a Transformation, then call transform()
    - If the object is an Estimator, then call fit(), which produces a Model. Then, call transform() on that model
- The result is a Model fitted to the data

PySpark regression without a pipeline

The original data frame

```
display(df)
```

▸ (1) Spark Jobs

Avg_Area_Income	Avg_Area_House_Age	Avg_Area_Number_of_Rooms	Avg_Area_Number_of_Bedrooms	Area_Population	Price	Address
79545.45857431678	5.682861321615587	7.009188142792237	4.09	23086.800502686456	1059033.5578701235	208 Michael Ferry Apt. 674
79248.64245482568	6.0028998082752425	6.730821019094919	3.09	40173.07217364482	1505890.91484695	188 Johnson Views Suite 079
61287.067178656784	5.865889840310001	8.512727430375099	5.13	36882.15939970458	1058987.9878760849	9127

Manual regression in two stages

- RFormula transforms the DataFrame (in manual “pipeline”)

```
formula = "{} ~ {}".format("Price", " + ".join(columns))
print("Formula : {}".format(formula))
supervised = RFormula(formula = formula)
# Most transformers don't need to be fitted first (they just transform)
# However, RFormula must review the data to handle categorical variables before it does its transformation
train_fittedRF = supervised.fit(train_data) # this is the prepared RForumla model
# Now, transform the data using the fitted RForumla, resulting in new columns (features, label)
train_preparedDF = train_fittedRF.transform(train_data)
```

- LinearRegression (an estimator), uses fit() to create model

```
lr = LinearRegression(labelCol ="label", featuresCol ="features")
train_fittedLR = lr.fit(train_preparedDF)
labeledPredictions = train_fittedLR.transform(test_preparedDF).select("labe
```

Let’s look at the details of these two stages

RFormula is first stage: the fit()
(a 2-phase estimator)

- RFormula is fit() against the data
  - 1st (fit) it reads all data, checking for categorical values
  - 2nd (transform, next slide) it transforms the data according to the R-formula in preparation for input to a model

```
supervised = RFormula(formula = formula)
# Most transformers don't need to be fitted first (they just transform)
# However, RFormula must review the data to handle categorical variables before it does t
train_fittedRF = supervised.fit(train_data) # this is the prepared RForumla model
train_fittedRF
```

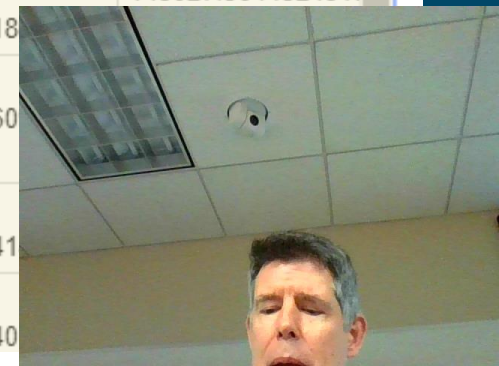
```
Out[64]: RFormula_e25eac586cdf
```

RFormula is first stage: the transform()

```
train_preparedDF = train_fittedRF.transform(train_data)
display(train_preparedDF)
```

- Calling transform() on the RFormula Model (Transformation) generates a new DataFrame with the original columns and now, the new Features and label columns generated by RFormula (transform)
- Each transformation adds columns to a DataFrame

er_of_Bedrooms	Area_Population	Price	Address	features	label
	24435.777301862807	143027.36445248185	166 Terry Grove	▸ [1,5,[], [35963.330809062856,3.4385465143530936,8.264121726134206,3.28,24435.7773018	143027.3644524818
	33267.767727560946	31140.517620186045	98398 Terrance Pines	▸ [1,5,[], [37971.20756623529,4.291223903128535,5.807509527238798,3.24,33267.767727560	
	38067.552184144304	899609.3001428025	783 Stacey Glen	▸ [1,5,[], [38122.524488262905,6.3361089421009655,7.762550883071808,5.12,38067.5521841	
	45899.73840240165	723750.0652577134	2899 Katherine	▸ [1,5,[], [38139.919044520015,5.577267494234432,6.348067700382508,2.13,45899.73840240	



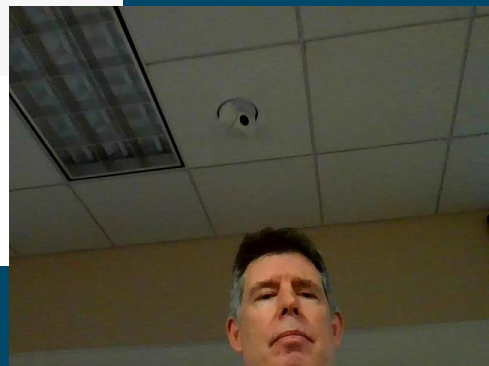
LinearRegression is second stage, fit()

- On the LinearRegression (Estimator) fit() is called to generate a model that is fitted to the DataFrame
- Notice that the output is a Model (which is a kind of Transformation)

```
lr = LinearRegression(labelCol ="label", featuresCol ="features")
train_fittedLR = lr.fit(train_preparedDF)
train_fittedLR
```

▸ (5) Spark Jobs

```
Out[63]: LinearRegression_72e362f0bfc6
```



LinearRegression is second stage, transform()

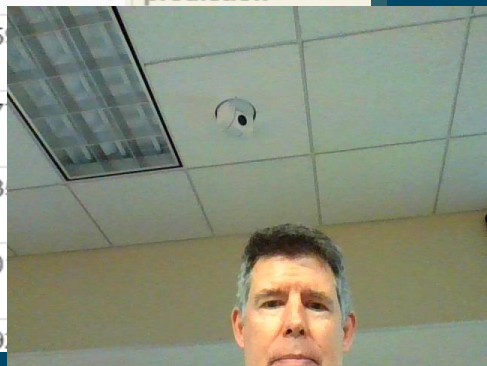
- After LinearRegression's fit() creates its model, its transform() is called on the test data, which produces the final DataFrame
- This transformation, like others, adds a column to the preceding DataFrame, here adding the prediction column

```
lr = LinearRegression(labelCol ="label", featuresCol ="features")
train_fittedLR = lr.fit(train_preparedDF)
lrDF = train_fittedLR.transform(test_preparedDF)
display(lrDF)
```

▸ (6) Spark Jobs

▸ lrDF: pyspark.sql.dataframe.DataFrame = [Avg_Area_Income: double, Avg_Area_House_Age: double ... 8 more fields]

rea_Population	Price	Address	features	label	prediction
7162.183643191434	302355.83597895555	9932 Eric Circles	▸[1,5,[], [17796.631189543397,4.9495570055571125,6.713905444702088,2.5,47162.183643191434]]	302355.835	
9636.40255302499	1077805.577726322	Unit 4700 Box 1880	▸[1,5,[], [35454.714659475445,6.855708363901107,6.018646502679608,4.5,59636.40255302499]]	1077805.57	
0833.007623149297	449331.5835333807	652 Stanton Island	▸[1,5,[], [35608.986237077515,6.935838865796822,7.827588622388147,6.35,20833.007623149297]]	449331.583	
4844.200190072384	299863.0401311839	645 Mary Radial	▸[1,5,[], [35797.323121548245,5.544221046634432,7.795138241804151,5,24844.200190072384]]	299863.040	
4901.857337630565	599504.0192866956	842 Duane	▸[1,5,[],	599504.019	



A fitted LinearRegression Model is the result

- Properties of the model can be presented, such as coefficients, and intercept

```
lrModel = train_fittedLR
# Print the coefficients and intercept for linear regression
print("Coefficients: %s" % str(lrModel.coefficients))
print("Intercept: %s" % str(lrModel.intercept))
```

```
Coefficients: [21.65762387655863,166392.77127254306,120250.65990624814,1400.6428044458557,15.336207832664162]
Intercept: -2648259.32117
```

```
trainingSummary = lrModel.summary
#trainingSummary.residuals.show()
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)
```

```
RMSE: 100937.153678
r2: 0.920383
```

PySpark regression with a pipeline

Feature construction & model building is the same

- With or without pipeline, the steps are the same
  - Specify (feature) Transformations (e.g., Tokenizer) and Estimators (e.g., LinearRegression)
  - For each of these objects
    - If the object is a Transformation, then call transform()
    - If the object is an Estimator, then call fit(), which produces a Model. Then, call transform() on that model
- The result is a Model fitted to the data

Same regression but in a pipeline: fit()

- Specify the pipeline (a list passed to Pipeline)

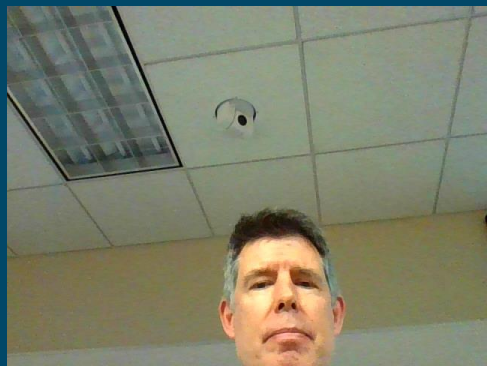
```
rformula = RFormula(formula = formula)
lr = LinearRegression(labelCol ="label", featuresCol ="features")
# The ML pipeline
pipeline = Pipeline(stages=[rformula, lr])
```

- fit() the pipeline
  - For each object in pipeline, in sequence,
    - call fit() & transform() for estimator object
    - call transform() for transform objects
  - The result is a Pipeline of fitted objects
    - Now, our model is ready to be applied to data

```
fittedPipe = pipeline.fit(train_data)
fittedPipe.stages
```

▸ (5) Spark Jobs

```
Out[18]: [RFormula_74d26a00e123, LinearRegression_80592590fb3a]
```



Details of Pipeline.fit()

- Pipeline(stages =[transformerA, estimatorB, transformerC])
  - Example [Tokenizer, Word2Vec, n-gram]
- Pipeline.fit() → fit() & transform()

1. `dataset = transformerA.transform(dataset)` // lazy call
2. `transformerB = estimatorB.fit(dataset)` // it will perform execution of previous lazy calls
3. `dataset = transformerB.transform(dataset)` // lazy call, never executed
4. `dataset = transformerC.transform(dataset)` // lazy call, never executed

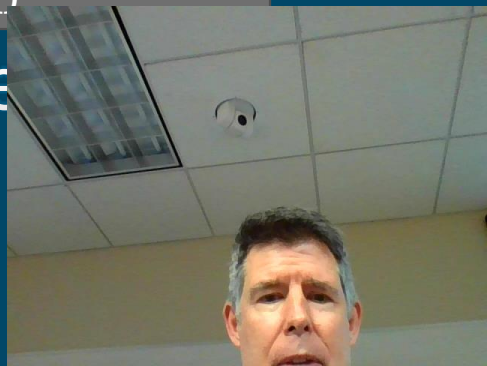
No need to output Tokens, so lazy

Word2Vec has Tokens as input, so run Tokenizer.transform() & Word2Vec.fit()

No need for output, so lazy transform()

- Later, we'll Pipeline.transform() and view output, the
  #4 executed

https://stackoverflow.com/questions/44375206/pyspark-ml-feature-transform-calls-the-fit-transform-method-twice-or-overide-i



Now Pipeline.transform()

- A fitted Pipeline represents our Model
  - We use transform() on the test data, which adds the prediction column to the DataFrame

```
predictions = fittedPipe.transform(test_data)
display(predictions)
```

▸ (1) Spark Jobs

▸ predictions: pyspark.sql.dataframe.DataFrame = [Avg_Area_Income: double, Avg_Area_House_Age: double ... 8 more fields]

pulation	Price	Address	features	label	prediction
3643191434	302355.83597895555	9932 Eric Circles	▸[1,5,[], [17796.631189543397,4.9495570055571125,6.713905444702088,2.5,47162.183643191434]]	302355.83597895555	94886.1476044762
255302499	1077805.577726322	Unit 4700 Box 1880	▸[1,5,[], [35454.714659475445,6.855708363901107,6.018646502679608,4.5,59636.40255302499]]	1077805.	
7623149297	449331.5835333807	652 Stanton Island	▸[1,5,[], [35608.986237077515,6.935838865796822,7.827588622388147,6.35,20833.007623149297]]	449331.	
0190072384	299863.0401311839	645 Mary Radial	▸[1,5,[], [35797.323121548245,5.544221046634432,7.795138241804151,5,24844.200190072384]]	299863.	
7337630565	599504.0192866956	842 Duane	▸[1,5,[],	599504.	

Our fitted LinearRegression Model is in the Pipeline

- Properties of the model can be presented, such as coefficients, and intercept

```
print("Pipeline: ", fittedPipe.stages)
lrModel1 = fittedPipe.stages[1]
# Print the coefficients and intercept for linear regression
print("Coefficients: %s" % str(lrModel1.coefficients))
print("Intercept: %s" % str(lrModel1.intercept))
```

```
Pipeline:  [RFormula_5424ab8b6451, LinearRegression_aaab6e3c4dfc]
Coefficients: [21.65762387655863,166392.77127254306,120250.65990624814,1400.6428044458557,15.336207832664162]
Intercept: -2648259.32117
```

```
trainingSummary = lrModel1.summary
#trainingSummary.residuals.show()
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)
```

```
RMSE: 100937.153678
r2: 0.920383
```

Another example

Exam

Predict outdoor equipment purcl

Example:
Predict outdoor equipment purchase

- Collect data

```
%sh
wget 'https://apsportal.ibm.com/exchange-api/v1/entries/8044492073eb964f46597b4be06ff5ea/data?accessKey=9561295fa407698694b1e254d0099600' -O GoSales_Tx_NaiveBayes.csv
```

Explore and Visualize data

```
1 display(df_data)
```

▸ (1) Spark Jobs

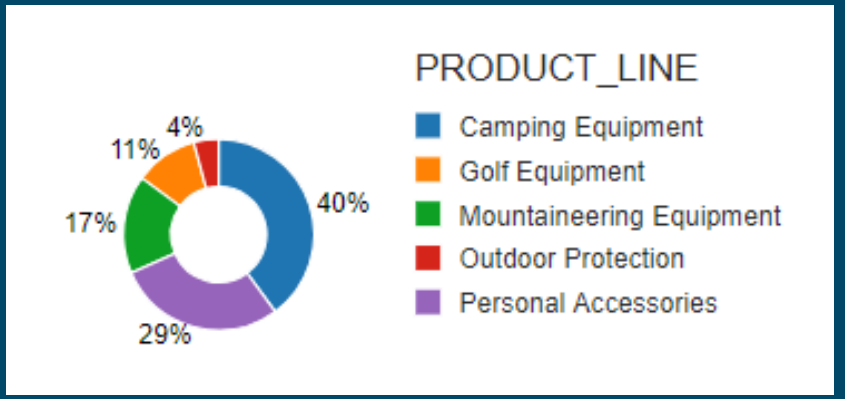
PRODUCT_LINE	GENDER	AGE	MARITAL_STATUS	PROFESSION
Personal Accessories	M	27	Single	Professional
Personal Accessories	F	39	Married	Other
Mountaineering Equipment	F	39	Married	Other
Personal Accessories	F	56	Unspecified	Hospitality
Golf Equipment	M	45	Married	Retired
Golf Equipment	M	45	Married	Retired
Camping Equipment	F	39	Married	Other
Camping Equipment	F	49	Married	Other
Outdoor Protection	F	49	Married	Other

Showing the first 1000 rows.

```
1 # The labels to be predicted...
2 df_data.select('PRODUCT_LINE').distinct().show()
```

▸ (5) Spark Jobs

```
+--------------------+
|        PRODUCT_LINE|
+--------------------+
|   Camping Equipment|
|     Golf Equipment|
|Mountaineering Eq...|
|  Outdoor Protection|
|Personal Accessories|
+--------------------+
```



Transform data for modeling

- Common to have to:
  - Format data as number, date, etc.
  - Remove nulls (using filter)
  - Split data into training and test sets

```
splitted_data = df_data.randomSplit([0.8, 0.18, 0.02], 24)# proportions [], seed for random
train_data = splitted_data[0]
test_data = splitted_data[1]
predict_data = splitted_data[2]

print "Number of training records: " + str(train_data.count())
print "Number of testing records : " + str(test_data.count())
print "Number of prediction records : " + str(predict_data.count())
```

Model data: creating features

- Convert text features
  - Index (0…n) using StringIndexer
  - Categorical variables using OneHotEncoder
  - Various text measures, such as term frequency
- Here, simple index

```
6 stringIndexer_label = StringIndexer(inputCol="PRODUCT_LINE", outputCol="label").fit(df_data)  # product_line -> Double
7 # Note the above outputCol is label (the predicted column). Here we predict the product line (above) from the attributes below.
8 stringIndexer_prof = StringIndexer(inputCol="PROFESSION", outputCol="PROFESSION_IX")
```

- Put them into a feature vector (for ML API use)

```
1 # Select the input columns for the model (and put them into one features column)
2 vectorAssembler_features = VectorAssembler(inputCols=["AGE", "PROFESSION_IX"], outputCol="features")
```

ML DataFrame: data + features

PRODUCT_LINE	GENDER	AGE	MARITAL_STATUS	PROFESSION	label	PROFESSION_IX	features
Camping Equipment	F	18	Single	Other	0	0	["1","2",[],[18,0]]
Camping Equipment	F	18	Single	Retail	0	7	["1","2",[],[18,7]]
Camping Equipment	F	19	Single	Hospitality	0	5	["1","2",[],[19,5]]
Camping Equipment	F	19	Single	Hospitality	0	5	["1","2",[],[19,5]]
Camping Equipment	F	19	Single	Hospitality	0	5	["1","2",[],

Showing the first 1000 rows.

Model: Random Forrest Classifier

- Create the model

```
rf = RandomForestClassifier(labelCol="label", featuresCol="features")
```

- Create a pipeline to process & model data

```
pipeline_rf = Pipeline(stages=[stringIndexer_label, stringIndexer_prof, vectorAssembler_features, rf, labelConverter])
```

- Train (fit) the model to the data

```
model_rf = pipeline_rf.fit(train_data)
```

Predict & evaluate using model

```
predictions = model_rf.transform(test_data)
evaluatorRF = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")
accuracy = evaluatorRF.evaluate(predictions)
print("Accuracy = %g" % accuracy)
print("Test Error = %g" % (1.0 - accuracy))
```

ML Result DataFrame:
data + features + prediction

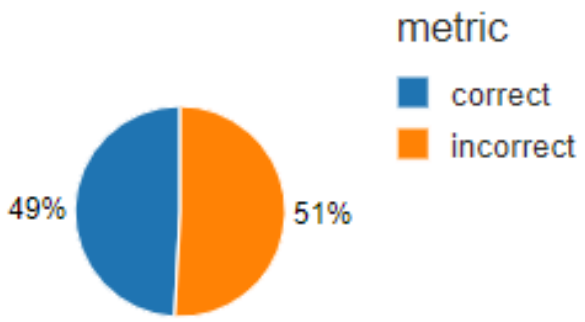
features	rawPrediction	probability	prediction	predictedLabel
["1","2",[], [18,0]]	["1","5",[], [7.693058582762915,6.599430908635922,4.8603764097170155,0.3611703600858441,0.4859637387983046]]	["1","5",[], [0.38465292913814575,0.3299715454317961,0.24301882048585077,0.018058518004292205,0.024298186939915232]]	0	Camping Equipment
["1","2",[], [18,7]]	["1","5",[], [2.7073422103482856,12.018381140731401,3.7414520693879227,0.10987147574280688,1.4229531037895848]]	["1","5",[], [0.1353671105174143,0.6009190570365701,0.18707260346939614,0.005493573787140344,0.07114765518947924]]	1	Personal Accessories
["1","2",[], [19,5]]	["1","5",[], [17.106650217071977,0.9441316035501558,0.3347717374466458,1.1117917984817347,0.5026546434494862]]	["1","5",[], [0.855332510853599,0.0472065801775078,0.016738586872332293,0.055589589924086746,0.025132732172474314]]	0	Camping Equipment
["1","2",[], [19,5]]	["1","5",[], [17.106650217071977,0.9441316035501558,0.3347717374466458,1.1117917984817347,0.5026546434494862]]	["1","5",[], [0.855332510853599,0.0472065801775078,0.016738586872332293,0.055589589924086746,0.025132732172474314]]	0	Camping Equipment
["1","2",[],	["1","5",[],	["1","5",[],	0	Camping

Visualize results

```
1 correct = predictions.where("(label = prediction)").count()
2 incorrect = predictions.where("(label != prediction)").count()
3
4 resultDF = sqlContext.createDataFrame([['correct', correct], ['incorrect', incorrect]], ['metric', 'value'])
5 display(resultDF)
```

▸ (5) Spark Jobs

▸ resultDF: pyspark.sql.dataframe.DataFrame = [metric: string, value: long]



Important to remember

- Pipeline is better because
  - fewer lines of code
  - easier to understand
  - is a programmable object
  - supports multiple models simultaneously