Apache Spark Machine Learning

SMU, 2019

Nikita Neveditsin
nikita.neveditsin@smu.ca

What is Spark ML?

MLlib is a Spark machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

- **ML Algorithms**: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- **Featurization**: feature extraction, transformation, dimensionality reduction, and selection
- **Pipelines**: tools for constructing, evaluating, and tuning ML Pipelines
- **Persistence**: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.[l]

[link]

Clustering

k-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters. The MLlib implementation includes a parallelized variant of the *k-means++* method called *kmeans||*[I]

Prepare your notebook

- **Open Zeppelin Notebook (http://sandbox-hdp.hortonworks.com:9995)**
- **Download a notebook http://wolly.cs.smu.ca/tmp/spml.json and import it into Zeppelin**
- **Run the first 5 cells**

Clustering: try k-means

First try: run kmeans as is (will give an error)

```
%spark2.pyspark
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

df = spark.read.format("csv").option("inferSchema", "true").load("/__dsets/kmeans/kmeans_example.csv")

df.show(2)

kmeans = KMeans().setK(2).setSeed(1L)
model = kmeans.fit(df)

# Results of clustering
results = model.transform(df)

results.show(2)
# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(results)
print(silhouette)

# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

Clustering: field “features” does not exist

```
+-------------------+-------------------+
|                _c0|                _c1|
+-------------------+-------------------+
|-1.1827670320796768|-1.1407926475386723|
|-1.9793751488152271|-0.5969386524684088|
+-------------------+-------------------+
only showing top 2 rows

Traceback (most recent call last):
  File "/tmp/zeppelin_pyspark-1970741528662838763.py", line 367, in <module>
    raise Exception(traceback.format_exc())
Exception: Traceback (most recent call last):
  File "/tmp/zeppelin_pyspark-1970741528662838763.py", line 355, in <module>
    exec(code, _zcUserQueryNameSpace)
  File "<stdin>", line 6, in <module>
  File "/usr/hdp/current/spark2-client/python/pyspark/ml/base.py", line 132, in fit
    return self._fit(dataset)
  File "/usr/hdp/current/spark2-client/python/pyspark/ml/wrapper.py", line 288, in _fit
    java_model = self._fit_java(dataset)
  File "/usr/hdp/current/spark2-client/python/pyspark/ml/wrapper.py", line 285, in _fit_java
    return self._java_obj.fit(dataset._jdf)
  File "/usr/hdp/current/spark2-client/python/lib/py4j-0.10.6-src.zip/py4j/java_gateway.py", line 1160, in __call__
    answer, self.gateway_client, self.target_id, self.name)
  File "/usr/hdp/current/spark2-client/python/pyspark/sql/utils.py", line 79, in deco
    raise IllegalArgumentException(s.split(': ', 1)[1], stackTrace)
IllegalArgumentException: u'Field "features" does not exist.\nAvailable fields: _c0, _c1'
```

Clustering: let's check the official manual

K-means

k-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters. The MLlib implementation includes a parallelized variant of the k-means++ method called kmeans||.

`KMeans` is implemented as an `Estimator` and generates a `KMeansModel` as the base model.

Input Columns

Param name	Type(s)	Default	Description
featuresCol	Vector	"features"	Feature vector

Output Columns

Param name	Type(s)	Default	Description
predictionCol	Int	"prediction"	Predicted cluster center

[link]

Feature? Vector?

- A **feature** is an individual measurable property or characteristic of a phenomenon being observed [l]
- For most of the machine learning algorithms, Spark ML requires converting features into Vectors. **Vector** is a *data type* that is used by Spark ML.

Vector as a data type in Spark ML

- Spark ML works with the following data types: Vectors, Labeled points, various types of Matrices
- A local **vector** has *integer-typed and 0-based indices* and *double-typed values*, stored on a single machine. MLlib supports two types of local vectors: **dense** and **sparse**. A dense vector is backed by a double array representing its entry values, while a sparse vector is backed by two parallel arrays: indices and values. For example, a vector (1.0, 0.0, 3.0) can be represented in dense format as [1.0, 0.0, 3.0] or in sparse format as (3, [0, 2], [1.0, 3.0]), where 3 is the size of the vector[I]

So how to create a “features” column?

VectorAssembler is a *transformer* that combines a given **list of columns** into a **single vector column**. It is useful for combining raw features and features generated by different feature transformers into a single feature vector. VectorAssembler accepts the following input column types:

- all numeric types,
- boolean type,
- vector type [I]

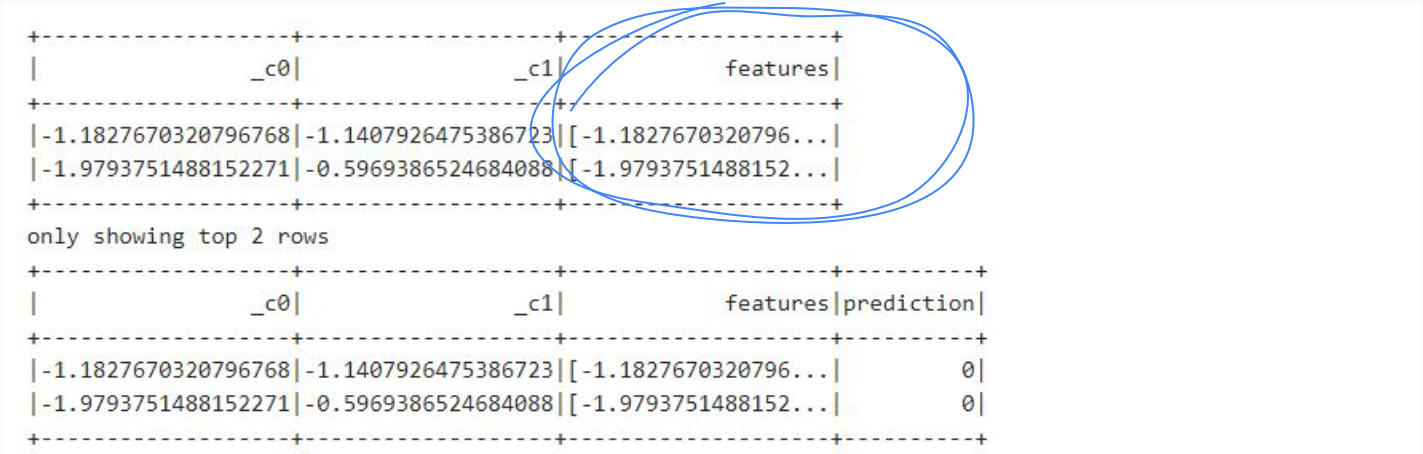
Let’s go back to our k-means example

We transformed our DataFrame to a new DataFrame with a new column “features” of the Vector type

Second try: convert features to a Vector

```
%spark2.pyspark
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

df = spark.read.format("csv").option("inferSchema", "true").load("/__dsets/kmeans/kmeans_example.csv")

assembler = VectorAssembler(
    inputCols=["_c0", "_c1"],
    outputCol="features")

df = assembler.transform(df)

df.show(2)

kmeans = KMeans().setK(2).setSeed(1L)
model = kmeans.fit(df)

# Results of clustering
results = model.transform(df)


results.show(2)
# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(results)
print(silhouette)

# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

“Features” column is just a combination of _c0,_c1

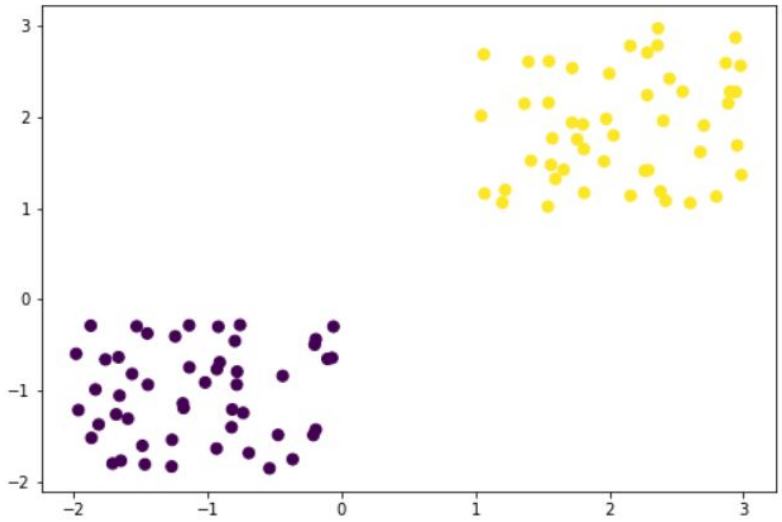


```
+-------------------+-------------------+--------------------+
|                _c0|                _c1|            features|
+-------------------+-------------------+--------------------+
|-1.1827670320796768|-1.1407926475386723|[-1.1827670320796...|
|-1.9793751488152271|-0.5969386524684088|[-1.9793751488152...|
+-------------------+-------------------+--------------------+
only showing top 2 rows
+-------------------+-------------------+--------------------+----------+
|                _c0|                _c1|            features|prediction|
+-------------------+-------------------+--------------------+----------+
|-1.1827670320796768|-1.1407926475386723|[-1.1827670320796...|         0|
|-1.9793751488152271|-0.5969386524684088|[-1.9793751488152...|         0|
+-------------------+-------------------+--------------------+----------+
```

K-means: convert to Pandas and plot the results

Plot the results

```
%spark2.pyspark
pred = results.toPandas()
import matplotlib.pyplot as plt
from numpy import genfromtxt

plt.scatter(pred["_c0"], pred["_c1"], s = 50, c = pred["prediction"])
plt.show()
```



Exercise

Using the existing code:

- try the following number of clusters: 3, 4, 8
- place clustering results on the plots
- which one gives the best *silhouette* score?

What is a transformer? ML Pipelines

MLlib standardizes APIs for machine learning algorithms to make it easier to ***combine multiple algorithms into a single pipeline***, or workflow. ML Pipelines work with **DataFrames** (covered previously). Components:

- **Transformer**: A Transformer is an algorithm which can *transform one DataFrame into another DataFrame*. E.g., an ML model is a Transformer which transforms a DataFrame with features into a DataFrame with predictions. [DF -> DF]
- **Estimator**: An Estimator is an algorithm which can be **fit** on a DataFrame *to produce a Transformer*. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model. [DF -> Transformer]
- **Pipeline**: A Pipeline **chains** multiple Transformers and Estimators together to specify an ML workflow [DF -> Transformer]
- **Parameter**: All Transformers and Estimators now share a common API for specifying parameters.

More on Spark ML Pipeline components

A Transformer is an abstraction that includes **feature transformers** and **learned models**. Technically, a Transformer implements a method transform(), which converts one DataFrame into another, generally by appending one or more columns. For example:

- 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.
- A learning model might take a DataFrame, read the column containing feature vectors, predict the label for each feature vector, and output a new DataFrame with predicted labels appended as a column[l].

An Estimator abstracts the concept of a learning algorithm or any algorithm that **fits** or **trains** on data. Technically, an Estimator implements a method *fit()*, which accepts a DataFrame and produces a Model, which is a Transformer. For example, a learning algorithm such as LogisticRegression is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model and hence a Transformer[l].

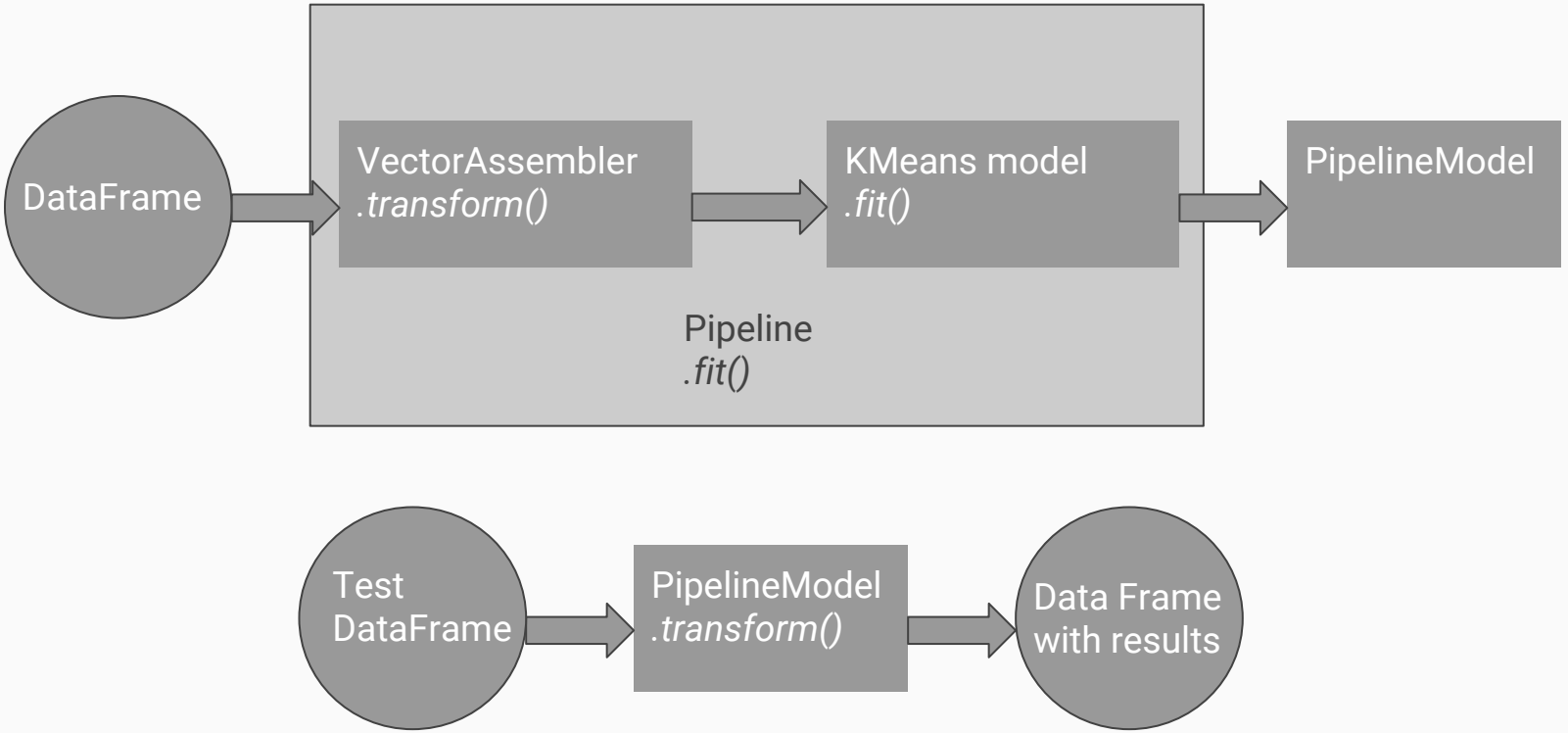
A Pipeline is specified as a *sequence of stages*, and each stage is either a Transformer or an Estimator. These stages are run **in order**, and the input DataFrame is transformed as it passes through each stage[l].
A Pipeline is an **estimator.** It produces a pipeline model which is **a transformer**.

Clustering with a Pipeline

Third try: use Pipeline

```
%spark2.pyspark
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

df = spark.read.format("csv").option("inferSchema", "true").load("/__dsets/kmeans/kmeans_example.csv")

va = VectorAssembler(
    inputCols=["_c0", "_c1"],
    outputCol="features")

kmeans = KMeans().setK(2).setSeed(1L)

pipeline = Pipeline(stages=[va, kmeans])

model = pipeline.fit(df)

res = model.transform(df)
res.show(2)

# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()

silhouette = evaluator.evaluate(res)
print(silhouette)

# Shows the result.
centers = model.stages[-1].clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

Clustering with a Pipeline (cont-d)



Prediction and classification: Random Forest

Random forests are ensembles of decision trees. Random forests combine many decision trees in order to reduce the risk of overfitting. The spark.ml implementation supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features[l].

Random Forest: Inputs and Outputs

Input Columns

Param name	Type(s)	Default	Description
labelCol	Double	"label"	Label to predict
featuresCol	Vector	"features"	Feature vector

Output Columns (Predictions)

Param name	Type(s)	Default	Description	Notes
predictionCol	Double	"prediction"	Predicted label	
rawPredictionCol	Vector	"rawPrediction"	Vector of length # classes, with the counts of training instance labels at the tree node which makes the prediction	Classification only
probabilityCol	Vector	"probability"	Vector of length # classes equal to rawPrediction normalized to a multinomial distribution	Classification only

[link]

RF example. Adults dataset (from prev. lecture)

```
%spark2.pyspark
df = spark.read.format("csv").option("inferSchema", "true").option("header", "true").load("/__dsets/census/adult.data")
df.show(2)
```

FINISHED

```
+---+----------------+-------+---------+-------------+------------------+----------------+-------------+------+-----+------------+------------+--------------+--------------+------+
|age|       workclass| fnlwgt| education| education-num|     marital-status|      occupation|  relationship|  race|  sex| capital-gain| capital-loss| hours-per-week| native-country| income|
+---+----------------+-------+---------+-------------+------------------+----------------+-------------+------+-----+------------+------------+--------------+--------------+------+
| 39|       State-gov|77516.0| Bachelors|         13.0|      Never-married|    Adm-clerical| Not-in-family| White| Male|      2174.0|         0.0|          40.0|  United-States|  <=50K|
| 50| Self-emp-not-inc|83311.0| Bachelors|         13.0| Married-civ-spouse| Exec-managerial|       Husband| White| Male|         0.0|         0.0|          13.0|  United-States|  <=50K|
+---+----------------+-------+---------+-------------+------------------+----------------+-------------+------+-----+------------+------------+--------------+--------------+------+
only showing top 2 rows
```

Took 1 sec. Last updated by anonymous at March 22 2019, 1:37:40 PM.

```
df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |--  workclass: string (nullable = true)
 |--  fnlwgt: double (nullable = true)
 |--  education: string (nullable = true)
 |--  education-num: double (nullable = true)
 |--  marital-status: string (nullable = true)
 |--  occupation: string (nullable = true)
 |--  relationship: string (nullable = true)
 |--  race: string (nullable = true)
 |--  sex: string (nullable = true)
 |--  capital-gain: double (nullable = true)
 |--  capital-loss: double (nullable = true)
 |--  hours-per-week: double (nullable = true)
 |--  native-country: string (nullable = true)
 |--  income: string (nullable = true)
```

RF Example: step 1 - index output (label)

```
#index output
out_indexer = StringIndexer(inputCol="income", outputCol="label")
#out_indexer.fit(df).transform(df).show(2)
pipe_stages += [out_indexer]
```

StringIndexer encodes a string column of labels to a column of label indices. The indices are in [0, numLabels), and four ordering options are supported: “frequencyDesc”: descending order by label frequency (most frequent label assigned 0), “frequencyAsc”: ascending order by label frequency (least frequent label assigned 0), “alphabetDesc”: descending alphabetical order, and “alphabetAsc”: ascending alphabetical order (default = “frequencyDesc”).

Additionally, there are three strategies regarding how StringIndexer will handle unseen labels when you have fit a StringIndexer on one dataset and then use it to transform another:

- throw an exception (which is the default)
- skip the row containing the unseen label entirely
- put unseen labels in a special additional bucket, at index numLabels

RF Example: step 1 - index output (label): out

```
---------+------------------+----------------+--------------+------+-----+------------+------------+--------------+--------------+------+
ation_num|     marital_status|      occupation|  relationship|  race|  sex|capital_gain|capital_loss|hours_per_week|native_country|income|
---------+------------------+----------------+--------------+------+-----+------------+------------+--------------+--------------+------+
     13.0|      Never-married|    Adm-clerical| Not-in-family| White| Male|      2174.0|         0.0|          40.0| United-States| <=50K|
     13.0| Married-civ-spouse| Exec-managerial|       Husband| White| Male|         0.0|         0.0|          13.0| United-States| <=50K|
---------+------------------+----------------+--------------+------+-----+------------+------------+--------------+--------------+------+


---------+------------------+----------------+--------------+------+-----+------------+------------+--------------+--------------+------+-----+
ation_num|     marital_status|      occupation|  relationship|  race|  sex|capital_gain|capital_loss|hours_per_week|native_country|income|label|
---------+------------------+----------------+--------------+------+-----+------------+------------+--------------+--------------+------+-----+
     13.0|      Never-married|    Adm-clerical| Not-in-family| White| Male|      2174.0|         0.0|          40.0| United-States| <=50K|  0.0|
     13.0| Married-civ-spouse| Exec-managerial|       Husband| White| Male|         0.0|         0.0|          13.0| United-States| <=50K|  0.0|
---------+------------------+----------------+--------------+------+-----+------------+------------+--------------+--------------+------+-----+
```

RF Example: step 2 - select features

```
#define columns what will be used for prediction of income in our model
#age, workclass, education|education_num, marital_status, occupation, race, sex, hours_per_week, native_country

#which of the columns are numeric?
numeric = ["age", "hours_per_week"]

#which of the columns need to be converted to numeric format?
nominal = ["workclass", "education", "marital_status", "occupation", "race", "sex", "native_country"]
```

Here we have 2 basic types of features (variables): ordinal (numeric) and nominal (categorical)

- Numeric features are good to go right to the Vector Assembler
- Nominal (categorical) features have to be converted to a numeric type

RF Example: step 3 - convert the nominal features

```
#create indexer to create index columns
indexers = []
for col in nominal:
    indexers += [StringIndexer(inputCol=col, outputCol=col + "Idx").setHandleInvalid("keep")]
pipe_stages += indexers

#create OneHot encoder to encode indexed columns to vectors
in_names = map(lambda col: col + "Idx", nominal)
out_names = map(lambda col: col + "Vec", nominal)
ohe = OneHotEncoderEstimator(inputCols=in_names, outputCols=out_names)
pipe_stages += [ohe]
```

- First, we index the nominal columns with StringIndexer (the same one that we used for creating labels)
- Then, we use One Hot encoder to encode Index columns

What is One Hot encoder?

One-hot encoding is used to map a categorical feature, represented as a *label index*, to a ***binary vector*** with at most a single one-value indicating the presence of a specific feature value from among the set of all feature values. This encoding allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features. For string type input data, it is common to encode categorical features using StringIndexer first[].

```
+-------------+
|  educationVec|
+-------------+
|(16,[5],[1.0])|
|(16,[7],[1.0])|
|(16,[5],[1.0])|
|(16,[7],[1.0])|
+-------------+
only showing top 4 rows
```

Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

[link]

RF Example: step 4 - assembling a feature vector

```
#now assemble all columns into a feature vector
assembler = VectorAssembler(inputCols=out_names + numeric, outputCol="features")
pipe_stages += [assembler]
```

- Note that numeric columns are assembled as is

RF Example: step 5 - create RF model (estimator)

```
#create a Random Forest classifier
rf = RandomForestClassifier(labelCol="label", featuresCol="features")
pipe_stages += [rf]
```

RF Example: step 6 - create a pipeline

```
#create a pipeline
pipeline = Pipeline(stages=pipe_stages)
```

RF Example: train and evaluate

```
#split data into training and test
(trainingData, testData) = df.randomSplit([0.75, 0.25], 1L)
model = pipeline.fit(trainingData)

pred = model.transform(testData)
pred.select("label", "income", "prediction", "probability", "education", "occupation").show(3)
#pred.select("educationVec").show(4)

#Evaluate model - gives AUC (the higher value is better)
evaluator = BinaryClassificationEvaluator()
evaluator.evaluate(pred)
```

Exercise

Using the existing code:

- instead of using "**education**" column, use "**education_num**" column as a feature
- compare AUC. Which option gives better AUC: "education" or "education_num"?
- try to use SVM Linear instead of Random forest. Which model gives better classification results?

  For linear SVM code: https://spark.apache.org/docs/latest/ml-classification-regression.html#linear-support-vector-machine

Model tuning: train validation split

Train-Validation split

```
%spark2.pyspark
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit(estimator=pipeline,
                           estimatorParamMaps=paramGrid,
                           evaluator=evaluator,
                           # 75% of the data will be used for training, 25% for validation.
                           trainRatio=0.75)

model = tvs.fit(trainingData)
predTvs = model.transform(testData)
evaluator.evaluate(predTvs)
```

Model tuning: k-folds cross-validation

Model Tuning

```
%spark2.pyspark
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

paramGrid = (ParamGridBuilder()
             .addGrid(rf.maxDepth, [2, 4, 6])
             .addGrid(rf.numTrees, [1, 5])
             .build())

cv = CrossValidator(estimator=pipeline, estimatorParamMaps=paramGrid, evaluator=evaluator, numFolds=5)

model = cv.fit(trainingData)
predCv = model.transform(testData)
evaluator.evaluate(predCv)
```

Model tuning: getting the best model

Getting the best model

```
%spark2.pyspark
rf_best = model.bestModel.stages[-1]
print(rf_best._java_obj.getMaxDepth())
print(rf_best._java_obj.getNumTrees())
```

Took 0 sec. Last updated by anonymous at March 21 2019, 4:42:56 PM. (outdated)

Exercise

Using the existing code:

- try to tune a *Linear SVM* model by setting the following *regParam* values: 0.05, 0.1, 1.0
- Which one gives the best result?

Association mining

Association mining example

```
%spark2.pyspark
from pyspark.ml.fpm import FPGrowth

df = spark.createDataFrame([
    (0, [1, 2, 5]),
    (1, [1, 2, 3, 5]),
    (2, [2, 1])
], ["id", "items"])

fpGrowth = FPGrowth(itemsCol="items", minSupport=0.5, minConfidence=0.6)
model = fpGrowth.fit(df)

# Display frequent itemsets.
model.freqItemsets.show()

# Display generated association rules.
model.associationRules.show()

# transform examines the input items against all the association rules and summarize the
# consequents as prediction
model.transform(df).show()
```

The example is taken from
https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html

Association mining: results

```
+---------+----+
|    items|freq|
+---------+----+
|      [5]|   2|
|   [5, 1]|   2|
|[5, 1, 2]|   2|
|   [5, 2]|   2|
|      [2]|   3|
|      [1]|   3|
|   [1, 2]|   3|
+---------+----+
```

```
+----------+----------+------------------+
|antecedent|consequent|        confidence|
+----------+----------+------------------+
|       [5]|       [1]|               1.0|
|       [5]|       [2]|               1.0|
|    [1, 2]|       [5]|0.6666666666666666|
|    [5, 2]|       [1]|               1.0|
|    [5, 1]|       [2]|               1.0|
|       [2]|       [5]|0.6666666666666666|
|       [2]|       [1]|               1.0|
|       [1]|       [5]|0.6666666666666666|
|       [1]|       [2]|               1.0|
+----------+----------+------------------+
```

```
+---+------------+----------+
| id|       items|prediction|
+---+------------+----------+
|  0|   [1, 2, 5]|        []|
|  1|[1, 2, 3, 5]|        []|
|  2|      [2, 1]|       [5]|
+---+------------+----------+
```

End of the class

Thanks!