Big data: architectures and data analytics

Spark MLlib – Part 3

Regression problem

Regression problem

- Predict the value of a continuous attribute (the target attribute)
- Regression is a logical extension of classification:
  - Harder from a mathematical perspective since there are infinite number of possible output values
  - Aim at optimizing metrics of error between predicted and true values (no accuracy rate)

Regression applications

- Use cases can be
  - Predicting usage of bike-sharing in an area and in a day
  - Predicting throughput of an internet connection
  - Predicting movie box-office takings
  - Predicting company revenue
  - ...

Regression algorithms

- Spark MLlib provides also a set of regression algorithms
  - E.g., Linear regression
- A regression algorithm is used to predict the target continuous attribute by applying a model on the predictive attributes
- The model is trained on a set of training data
  - i.e., a set of data for which the value of the target attribute is know

Regression algorithms

- The regression algorithms available in Spark work only on numerical data
  - They work similarly to classification algorithms, but they **predict continuous numerical values** (the target attribute is a continuous numerical attribute)
- The input data must be transformed in a DataFrame having the following attributes:
  - label: double
    - The continuous numerical value to be predicted
  - features: Vector of doubles
    - Predictive features

Regression algorithms

- Many regression algorithms are available in Mllib
  - Linear regression
  - Decision tree regression
  - Random forest regression
  - Survival regression
  - Isotonic regression
  - ...

Regression algorithms

- Many regression algorithms are available in Mllib
  - Linear regression
  - Decision tree regression
  - Random forest regression
  - Survival regression
  - Isotonic regression
  - ...

These algos are shown in the slides

Regression algorithms: scalability

Model	Number of features	Training size
Linear regression	> millions	No limit
Decision tree regression	> 1 000	No limit
Random forest regression	> 10 000	No limit
...	...	...

Regression and parameter setting

- The tuning approach that we used for the classification problem can also be used to optimize the regression problem
  - **CrossValidation**
  - **TrainValidationSplit**
- The only difference is given by the used evaluator
  - In this case the difference between the actual value and the predicted one must be computed

Regression and performance evaluation

- As for classification, in order to test the goodness of algorithms there is an **evaluator**
- The evaluator is **RegressionEvaluator** from **pyspark.ml.evaluation**
- The instantiated estimator has the method **evaluate()** that is applied to a dataframe
- It compares the prediction with the true value
- Output: the double value of the performance

Regression and Performance evaluation

- Parameters of **RegressionEvaluator**:
  - **MetricName:** string for metric name in evaluation. Supports:
    - "rmse" (default): root mean squared error
    - "mse": mean squared error
    - "r2": R2 metric
    - "mae": mean absolute error
  - **labelCol**: input column with the true double target
  - **predictionCol**: input column with the predicted double value

Linear regression

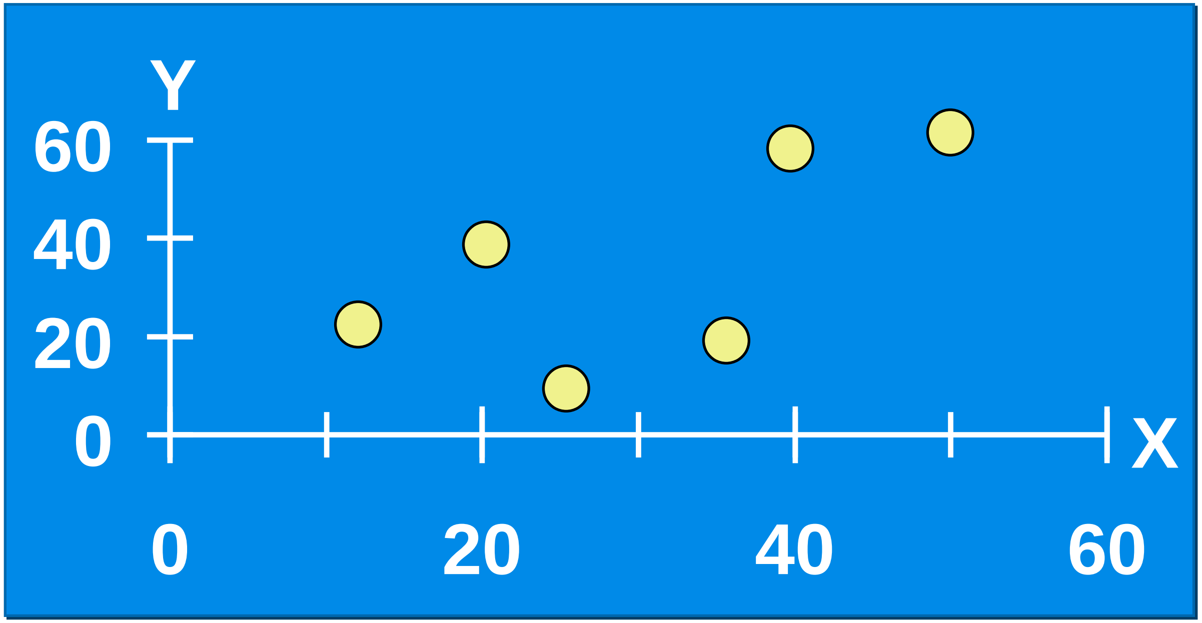
Linear regression

- Linear regression is a popular, effective and efficient regression algorithm
- It assumes a linear combination of your input features – sum of each feature multiplied by a weight
- The input features can be preprocessed (e.g, you can apply a non-linear function of them)
- Produce Gaussian error in the output

Linear regression: 1-D

1-D example: suppose X is the feature and Y the target value

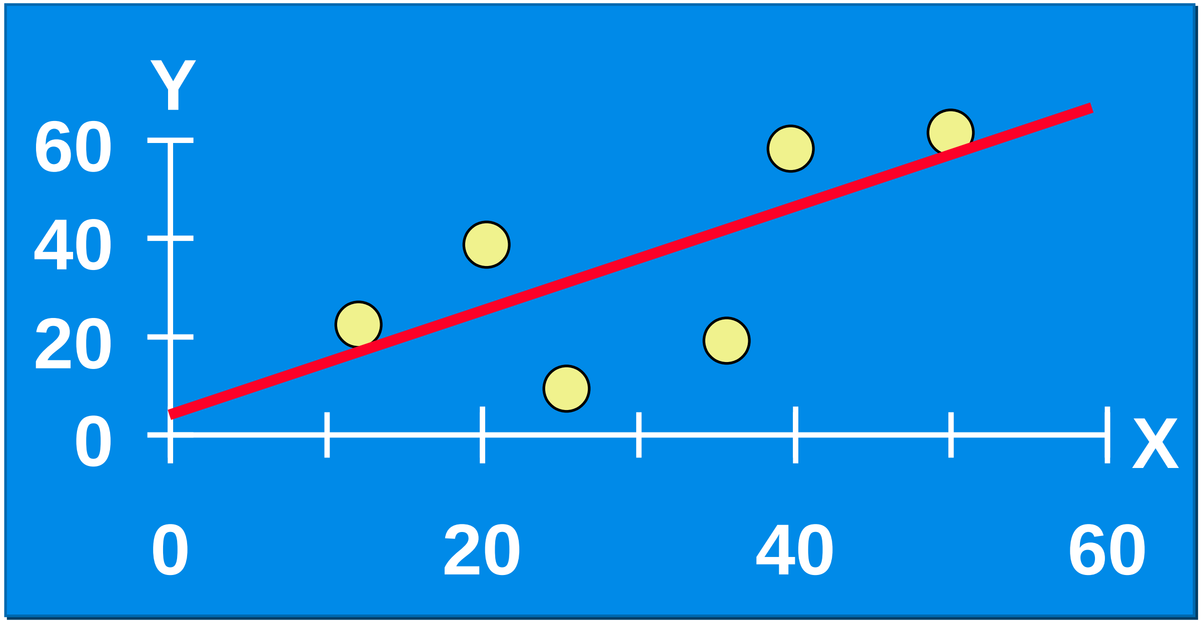
$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$



Linear regression: 1-D

How would you draw a line through the points? How do you determine which line 'fits best'?

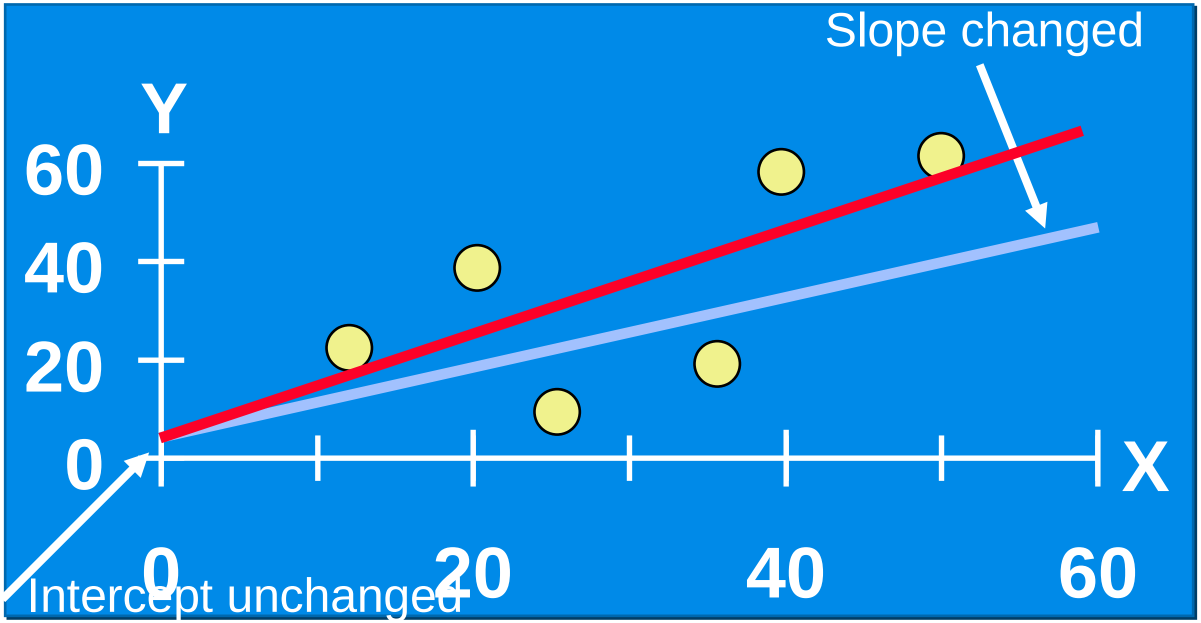
$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$



Linear regression: 1-D

How would you draw a line through the points? How do you determine which line 'fits best'?

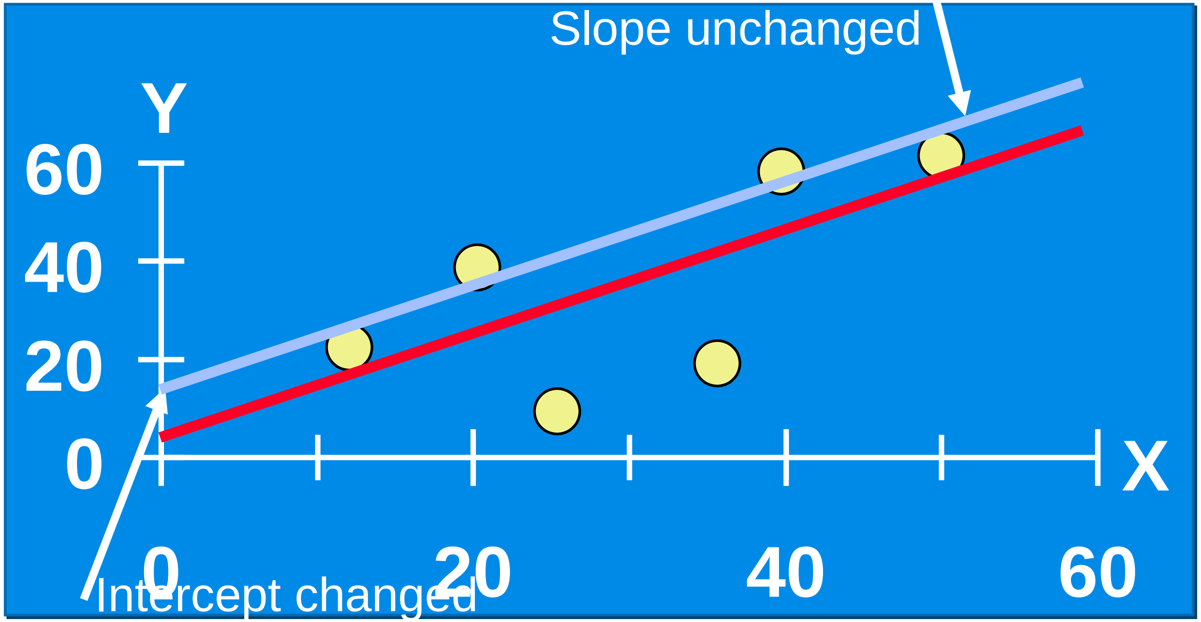
$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$



Linear regression: 1-D

How would you draw a line through the points? How do you determine which line 'fits best'?

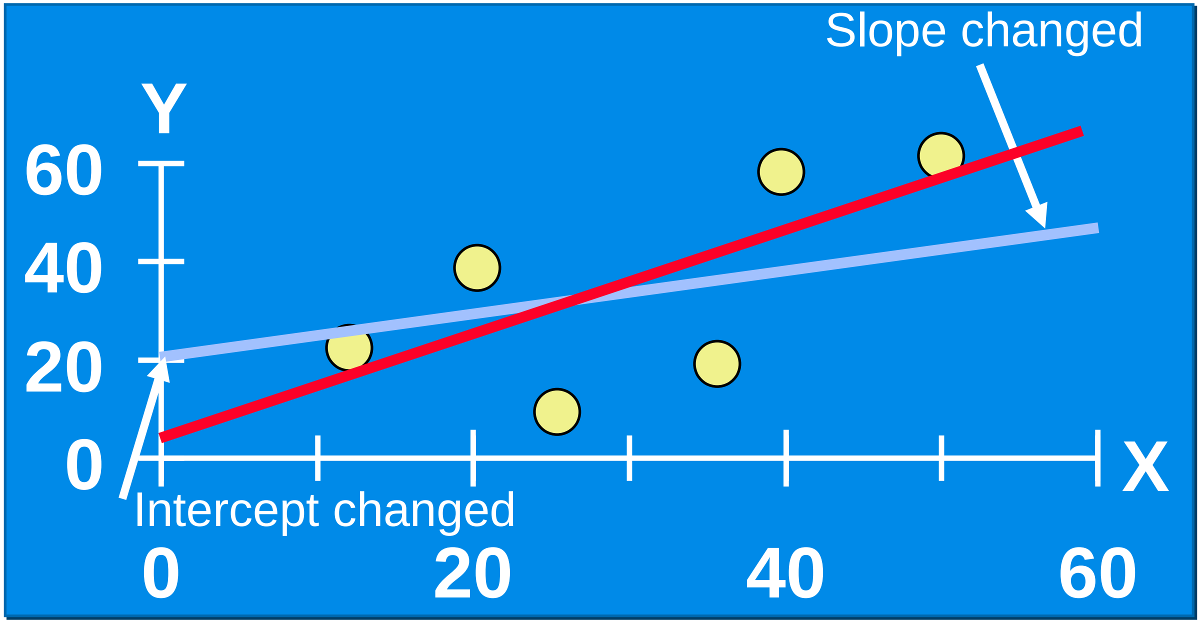
$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$



Linear regression: 1-D

How would you draw a line through the points? How do you determine which line ‘fits best’?

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$



Linear regression: Least Squares

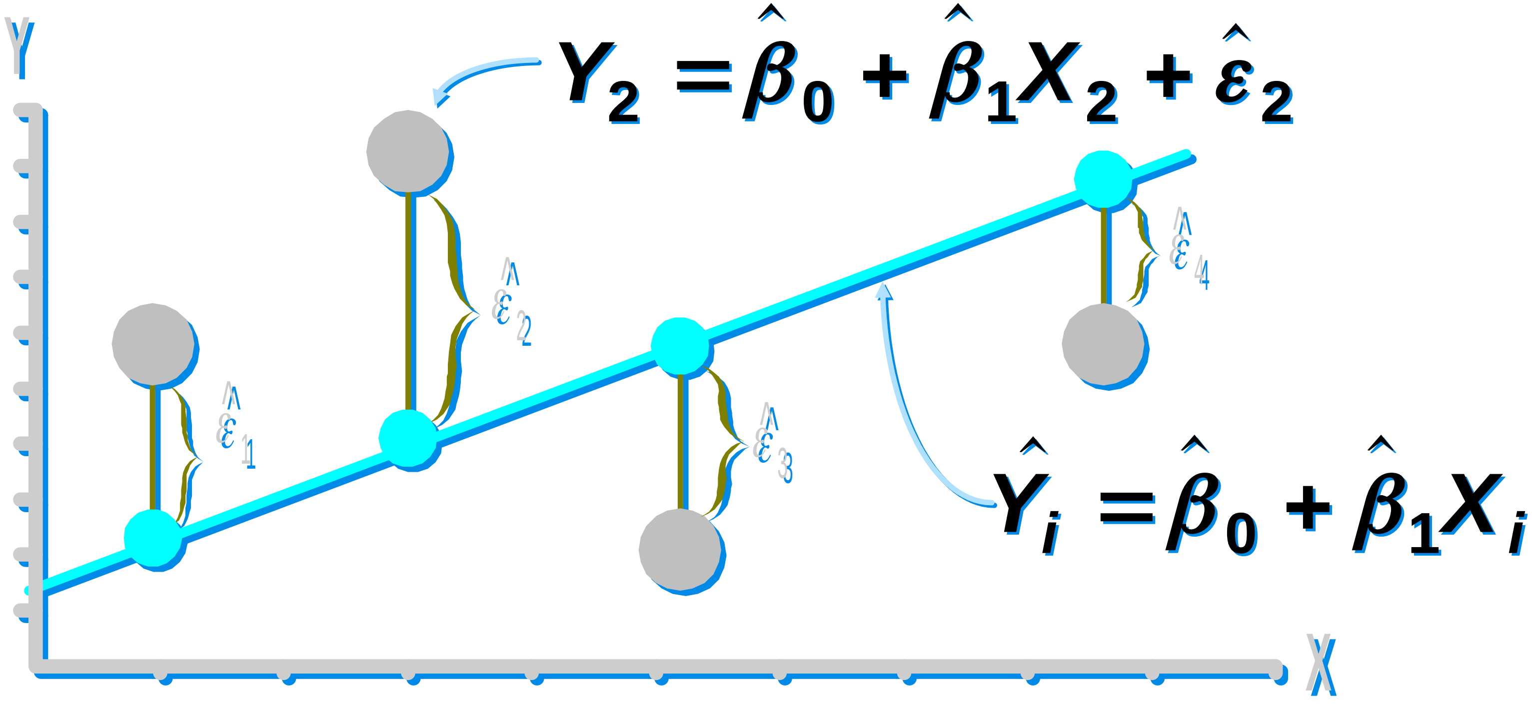
'Best Fit' means difference between actual Y values and predicted Y values are a minimum.

$$\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 = \sum_{i=1}^{n}\hat{\varepsilon}_i^2$$

Least squares minimizes the sum of the squared differences (errors)

Linear regression: Least Squares

LS minimizes $\sum_{i=1}^{n} \hat{\varepsilon}_i^2 = \hat{\varepsilon}_1^2 + \hat{\varepsilon}_2^2 + \hat{\varepsilon}_3^2 + \hat{\varepsilon}_4^2$



Linear regression: interpretation of coefficients

1) **Slope (β_1)**

– Estimated Y changes by β_1 for each unit increase in X

– If $\beta_1 = 2$, then Y increase by 2 for each 1 unit increase in X

2) **Y-Intercept (β_0)**

– Value of Y when $X = 0$

– If $\beta_0 = 4$, then Y is 4 when X is 0

Linear regression

- Strong points
  - Very fast to train
  - Simple and interpretable model
  - Hard to overfit
- Weak points
  - Linear assumption often does not hold
  - Even if it holds, the errors (the noise) might not be gaussian

Linear regression in MLlib

- How to instantiate a linear regression algorithm in Spark an apply it on unlabeled data using MLlib
- The input dataset is a structured dataset with a fixed number of attributes
  - One attribute is the target attribute (the label)
  - The others are predictive attributes that are used to predict the value of the target attribute

Linear regression in MLlib

- Use the **LinearRegression** estimator from **pyspark.ml.regression** on a DataFrame
- Explicitly specify input columns **featuresCol** (vector) and **labelCol** (double)
- Output column:
  - **predictionCol** with the predicted double value

Linear regression in MLlib

- (Some) parameters:
  - **maxIter**: maximum number of iterations to fit the data (>0)
  - **fitIntercept**: whether to fit an intercept term ("True" or "False")
  - **loss:** "squaredError" or "huber", i.e., the function to minimize
  - ...

Linear regression in Mllib: model characteristics and performance

- Model characteristics
  - lrModel**.coefficients** return a python list of the coefficients of the linear regressor
  - lrModel**.intercept** return the double value of the intercept
- We can get detailed information about the regressor we trained. The **summary** method of the transformer returns a summary object with many fields
- For examples:
  - Residuals of each training data (**summary.residuals** is a dataframe)
  - Root mean square error of residuals (**summary.rootMeanSquaredError** is a double)

Linear regression: example

- Consider the following example file about bike sharing usage

weekDay	distanceCenter	rentals
Monday	1.5	358
Saturday	1.0	272
Saturday	0.5	390
Monday	3.0	120
Saturday	0.3	439
Monday	0.9	509
Saturday	1.9	102
Saturday	2.7	43
Monday	0.6	597

- It contains 10 records
- Each record has two predictive attributes and the target attribute
  - “rentals” is the target attribute
  - The other attributes are predictive attributes
  - They represent the day of the week and the distance of the station from the center in km.

Linear regression: example

Preprocessing the data

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.feature import VectorAssembler

# Load training data
rentalsDF=spark.read.csv('rentals.txt',header=True,inferSchema=True)

indexer = StringIndexer(inputCol="weekDay",
  outputCol="weekDayIndex", handleInvalid="keep")
indexerModel = indexer.fit(rentalsDF)
indexedDF=indexerModel.transform(rentalsDF)

va=VectorAssembler(inputCols=["weekDayIndex","distanceCenter"],
  outputCol="features")
assembledDF=va.transform(indexedDF)
```

Linear regression: example

Input DataFrame

weekDay	distanceCenter	rentals
Monday	1.5	358
Saturday	1.0	272
Saturday	0.5	390
Monday	3.0	120
Saturday	0.3	439
Monday	0.9	509
Saturday	1.9	102
Saturday	2.7	43
Monday	0.6	597

Preprocessed DataFrame

weekDay	distanceCenter	rentals	weekDayIndex	features
Monday	1.5	358	1.0	[1.0,1.5]
Saturday	1.0	272	0.0	[0.0,1.0]
Saturday	0.5	390	0.0	[0.0,0.5]
Monday	3.0	120	1.0	[1.0,3.0]
Saturday	0.3	439	0.0	[0.0,0.3]
Monday	0.9	509	1.0	[1.0,0.9]
Saturday	1.9	102	0.0	[0.0,1.9]
Saturday	2.7	43	0.0	[0.0,2.7]
Monday	0.6	597	1.0	[1.0,0.6]

Linear regression: example

Input DataFrame

```
+--------+-------------+-------+
```

As we saw in the preprocessing part, scaling the attribute “distanceCenter” and encoding the “weekDay” to many binary features might produce better results

```
|Saturday|          1.9|    102|
|Saturday|          2.7|     43|
|  Monday|          0.6|    597|
+--------+-------------+-------+
```

Preprocessed DataFrame

```
+--------+-------------+-------+------------+---------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|
+--------+-------------+-------+------------+---------+
|  Monday|          1.5|    358|         1.0|[1.0,1.5]|
|Saturday|          1.0|    272|         0.0|[0.0,1.0]|
|Saturday|          0.5|    390|         0.0|[0.0,0.5]|
|  Monday|          3.0|    120|         1.0|[1.0,3.0]|
|Saturday|          0.3|    439|         0.0|[0.0,0.3]|
|  Monday|          0.9|    509|         1.0|[1.0,0.9]|
|Saturday|          1.9|    102|         0.0|[0.0,1.9]|
|Saturday|          2.7|     43|         0.0|[0.0,2.7]|
|  Monday|          0.6|    597|         1.0|[1.0,0.6]|
+--------+-------------+-------+------------+---------+
```

Linear regression: example

Training the linear regressor

```
from pyspark.ml.regression import LinearRegression

lr = LinearRegression(labelCol="rentals",featuresCol="features",maxIter=10)

# Fit the model
lrModel = lr.fit(assembledDF)

# Create the predictions
predictionDF=lrModel.transform(assembledDF)
```

Linear regression: example

Preprocessed dataframe

weekDay	distanceCenter	rentals	weekDayIndex	features
Monday	1.5	358	1.0	[1.0,1.5]
Saturday	1.0	272	0.0	[0.0,1.0]
Saturday	0.5	390	0.0	[0.0,0.5]
Monday	3.0	120	1.0	[1.0,3.0]
Saturday	0.3	439	0.0	[0.0,0.3]
Monday	0.9	509	1.0	[1.0,0.9]
Saturday	1.9	102	0.0	[0.0,1.9]
Saturday	2.7	43	0.0	[0.0,2.7]
Monday	0.6	597	1.0	[1.0,0.6]

Output dataframe with regression

weekDay	distanceCenter	rentals	weekDayIndex	features	prediction
Monday	1.5	358	1.0	[1.0,1.5]	395.82984014193653
Saturday	1.0	272	0.0	[0.0,1.0]	299.82396947108407
Saturday	0.5	390	0.0	[0.0,0.5]	389.98082944364353
Monday	3.0	120	1.0	[1.0,3.0]	125.3592602242581
Saturday	0.3	439	0.0	[0.0,0.3]	426.04357343266736
Monday	0.9	509	1.0	[1.0,0.9]	504.01807210900796
Saturday	1.9	102	0.0	[0.0,1.9]	137.541621520477
Saturday	2.7	43	0.0	[0.0,2.7]	-6.709354435618195
Monday	0.6	597	1.0	[1.0,0.6]	558.1121880925436

Linear regression: example

Information about the created regressor

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


# Summarize the model over the training set and print out some
    metrics
trainingSummary = lrModel.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
trainingSummary.residuals.show()
```

Linear regression: example

Output dataframe with regression

weekDay	distanceCenter	rentals	weekDayIndex	features	prediction
Monday	1.5	358	1.0	[1.0,1.5]	395.82984014193653
Saturday	1.0	272	0.0	[0.0,1.0]	299.82396947108407
Saturday	0.5	390	0.0	[0.0,0.5]	389.98082944364353
Monday	3.0	120	1.0	[1.0,3.0]	125.3592602242581
Saturday	0.3	439	0.0	[0.0,0.3]	426.04357343266736
Monday	0.9	509	1.0	[1.0,0.9]	504.01807210900796
Saturday	1.9	102	0.0	[0.0,1.9]	137.541621520477
Saturday	2.7	43	0.0	[0.0,2.7]	-6.709354435618195
Monday	0.6	597	1.0	[1.0,0.6]	558.1121880925436

Regressor information

```
Coefficients: [186.162730643412,-180.31371994511898]
Intercept: 480.13768941620305
RMSE: 29.198876
```

residuals
-37.82984014193653
-27.82396947108407
0.019170556356471025
-5.359260224258094
12.956426567332642
4.98192789099204
-35.54162152047701
49.709354435618195
38.88781190745635

Linear regression: example

Test DataFrame

```
+--------+--------------+-------+
| weekDay|distanceCenter|rentals|
+--------+--------------+-------+
|  Monday|           0.1|  641.0|
|Saturday|           2.1|  129.0|
|Saturday|           1.5|  199.0|
|  Monday|           2.0|  231.0|
|  Sunday|           0.5|  393.0|
+--------+--------------+-------+
```

Linear regression: example

Preprocess test

```
# Load test data
rentalsTestDF=spark.read.csv('rentalsTest.txt',header=True,inferSchema=True)


indexedTestDF=indexerModel.transform(rentalsTestDF)

assembledTestDF=va.transform(indexedTestDF)
```

Linear regression: example

Test DataFrame

```
+--------+-------------+-------+
| weekDay|distanceCenter|rentals|
+--------+-------------+-------+
|  Monday|          0.1|  641.0|
|Saturday|          2.1|  129.0|
|Saturday|          1.5|  199.0|
|  Monday|          2.0|  231.0|
|  Sunday|          0.5|  393.0|
+--------+-------------+-------+
```

Test DataFrame preprocessed

```
+--------+-------------+-------+-----------+---------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|
+--------+-------------+-------+-----------+---------+
|  Monday|          0.1|  641.0|        1.0|[1.0,0.1]|
|Saturday|          2.1|  129.0|        0.0|[0.0,2.1]|
|Saturday|          1.5|  199.0|        0.0|[0.0,1.5]|
|  Monday|          2.0|  231.0|        1.0|[1.0,2.0]|
|  Sunday|          0.5|  393.0|        2.0|[2.0,0.5]|
+--------+-------------+-------+-----------+---------+
```

Linear regression: example

Test DataFrame

```
+--------+--------------+-------+
| weekDay|distanceCenter|rentals|
+--------+--------------+-------+
|  Monday|           0.1|  641.0|
|Saturday|           2.1|  129.0|
|Saturday|           1.5|  199.0|
|  Monday|           2.0|  231.0|
|  Sunday|           0.5|  393.0|
+--------+--------------+-------+
```

"Sunday" does not produce an error since in the StringIndexer we selected to keep Iinvalid values. They are stored in another category (i.e., 2.0)

```
+--------+--------------+-------+-----------+---------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|
+--------+--------------+-------+-----------+---------+
|  Monday|           0.1|  641.0|        1.0|[1.0,0.1]|
|Saturday|           2.1|  129.0|        0.0|[0.0,2.1]|
|Saturday|           1.5|  199.0|        0.0|[0.0,1.5]|
|  Monday|           2.0|  231.0|        1.0|[1.0,2.0]|
|  Sunday|           0.5|  393.0|        2.0|[2.0,0.5]|
+--------+--------------+-------+-----------+---------+
```

Linear regression: example

Apply model to test data

```
from pyspark.ml.evaluation import RegressionEvaluator

predictionTestDF=lrModel.transform(assembledTestDF)


#  compute test error
evaluator = RegressionEvaluator(
    labelCol="rentals", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictionTestDF)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

Linear regression: example

Test DataFrame

```
+--------+--------------+-------+
| weekDay|distanceCenter|rentals|
+--------+--------------+-------+
|  Monday|           0.1|  641.0|
|Saturday|           2.1|  129.0|
|Saturday|           1.5|  199.0|
|  Monday|           2.0|  231.0|
|  Sunday|           0.5|  393.0|
+--------+--------------+-------+
```

Output test DataFrame with predictions

```
+--------+--------------+-------+------------+---------+------------------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|        prediction|
+--------+--------------+-------+------------+---------+------------------+
|  Monday|           0.1|  641.0|         1.0|[1.0,0.1]| 648.2690480651031|
|Saturday|           2.1|  129.0|         0.0|[0.0,2.1]|101.47887753145318|
|Saturday|           1.5|  199.0|         0.0|[0.0,1.5]|209.66710949852455|
|  Monday|           2.0|  231.0|         1.0|[1.0,2.0]|305.67298016937707|
|  Sunday|           0.5|  393.0|         2.0|[2.0,0.5]| 762.3062907304675|
+--------+--------------+-------+------------+---------+------------------+
```

```
Root Mean Squared Error (RMSE) on test data = 169.049
```

Decision tree regression

Decision tree regression

- Decision trees applied to regression work similarly to the ones for classification
- In regression, the trees output a single number per leaf node instead of a label
- A tree can predict a non linear function

Decision tree regression

- Use the **DecisionTreeRegressor** estimator from **pyspark.ml.regression** on a DataFrame
- Explicitly specify input columns **featuresCol** (vector) and **labelCol** (double)
- Output column:
  - **predictionCol** with the predicted double value

Decision tree regression

- Parameters are the same as for the classification
- Only difference is:
    - **impurity**: it represents the metric for wheter or not the model should split a particular leaf node with a particular value. The only currently supported metric is “variance”

Decision tree regression: example

- Consider the same previous example about predicting bike sharing usage
- Let’s start already from the preprocessed dataframe

```
+--------+-------------+-------+------------+---------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|
+--------+-------------+-------+------------+---------+
|  Monday|          1.5|    358|         1.0|[1.0,1.5]|
|Saturday|          1.0|    272|         0.0|[0.0,1.0]|
|Saturday|          0.5|    390|         0.0|[0.0,0.5]|
|  Monday|          3.0|    120|         1.0|[1.0,3.0]|
|Saturday|          0.3|    439|         0.0|[0.0,0.3]|
|  Monday|          0.9|    509|         1.0|[1.0,0.9]|
|Saturday|          1.9|    102|         0.0|[0.0,1.9]|
|Saturday|          2.7|     43|         0.0|[0.0,2.7]|
|  Monday|          0.6|    597|         1.0|[1.0,0.6]|
+--------+-------------+-------+------------+---------+
```

Decision tree regression: example

Training the decision tree

```
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.evaluation import RegressionEvaluator

# Train a DecisionTree model.
dt = DecisionTreeRegressor(labelCol="rentals",featuresCol="features",maxDepth=4)

# Fit the model
dtModel = dt.fit(assembledDF)

# Predict output
predictionDF=dtModel.transform(assembledDF)

# Compute training error
evaluator = RegressionEvaluator(
    labelCol="rentals", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictionDF)
print("Root Mean Squared Error (RMSE) on training data = %g" % rmse)
```

Decision tree regression: example

Preprocessed dataframe

weekDay	distanceCenter	rentals	weekDayIndex	features
Monday	1.5	358	1.0	[1.0,1.5]
Saturday	1.0	272	0.0	[0.0,1.0]
Saturday	0.5	390	0.0	[0.0,0.5]
Monday	3.0	120	1.0	[1.0,3.0]
Saturday	0.3	439	0.0	[0.0,0.3]
Monday	0.9	509	1.0	[1.0,0.9]
Saturday	1.9	102	0.0	[0.0,1.9]
Saturday	2.7	43	0.0	[0.0,2.7]
Monday	0.6	597	1.0	[1.0,0.6]

Output dataframe with regression

weekDay	distanceCenter	rentals	weekDayIndex	features	prediction
Monday	1.5	358	1.0	[1.0,1.5]	358.0
Saturday	1.0	272	0.0	[0.0,1.0]	272.0
Saturday	0.5	390	0.0	[0.0,0.5]	390.0
Monday	3.0	120	1.0	[1.0,3.0]	120.0
Saturday	0.3	439	0.0	[0.0,0.3]	439.0
Monday	0.9	509	1.0	[1.0,0.9]	509.0
Saturday	1.9	102	0.0	[0.0,1.9]	102.0
Saturday	2.7	43	0.0	[0.0,2.7]	43.0
Monday	0.6	597	1.0	[1.0,0.6]	597.0

```
Root Mean Squared Error (RMSE) on training data = 0
```

Decision tree regression: example

Preprocessed dataframe

```
+--------+--------------+-------+------------+---------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|
+--------+--------------+-------+------------+---------+
|  Monday|           1.5|    358|         1.0|[1.0,1.5]|
|Saturday|           1.0|    272|         0.0|[0.0,1.0]|
|Saturday|           0.5|    390|         0.0|[0.0,0.5]|
|  Monday|           3.0|    120|         1.0|[1.0,3.0]|
|Saturday|           0.3|    439|         0.0|[0.0,0.3]|
|  Monday|           0.9|    509|         1.0|[1.0,0.9]|
+--------+--------------+-------+------------+---------+
```

The model is perfect with respect to training data

Output dataframe with regression

```
+--------+--------------+-------+------------+---------+----------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|prediction|
+--------+--------------+-------+------------+---------+----------+
|  Monday|           1.5|    358|         1.0|[1.0,1.5]|     358.0|
|Saturday|           1.0|    272|         0.0|[0.0,1.0]|     272.0|
|Saturday|           0.5|    390|         0.0|[0.0,0.5]|     390.0|
|  Monday|           3.0|    120|         1.0|[1.0,3.0]|     120.0|
|Saturday|           0.3|    439|         0.0|[0.0,0.3]|     439.0|
|  Monday|           0.9|    509|         1.0|[1.0,0.9]|     509.0|
|Saturday|           1.9|    102|         0.0|[0.0,1.9]|     102.0|
|Saturday|           2.7|     43|         0.0|[0.0,2.7]|      43.0|
|  Monday|           0.6|    597|         1.0|[1.0,0.6]|     597.0|
+--------+--------------+-------+------------+---------+----------+
Root Mean Squared Error (RMSE) on training data = 0
```

Decision tree regression: example

Test DataFrame

```
+--------+--------------+-------+
| weekDay|distanceCenter|rentals|
+--------+--------------+-------+
|  Monday|           0.1|  641.0|
|Saturday|           2.1|  129.0|
|Saturday|           1.5|  199.0|
|  Monday|           2.0|  231.0|
|  Sunday|           0.5|  393.0|
+--------+--------------+-------+
```

Decision tree regression: example

Apply model to test data

```
from pyspark.ml.evaluation import RegressionEvaluator

predictionTestDF=lrModel.transform(assembledTestDF)


#  compute test error
evaluator = RegressionEvaluator(
   labelCol="rentals", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictionTestDF)

print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

Decision tree regression: example

Test DataFrame

```
+--------+--------------+-------+
| weekDay|distanceCenter|rentals|
+--------+--------------+-------+
|  Monday|           0.1|  641.0|
|Saturday|           2.1|  129.0|
|Saturday|           1.5|  199.0|
|  Monday|           2.0|  231.0|
|  Sunday|           0.5|  393.0|
+--------+--------------+-------+
```

Output test DataFrame with predictions

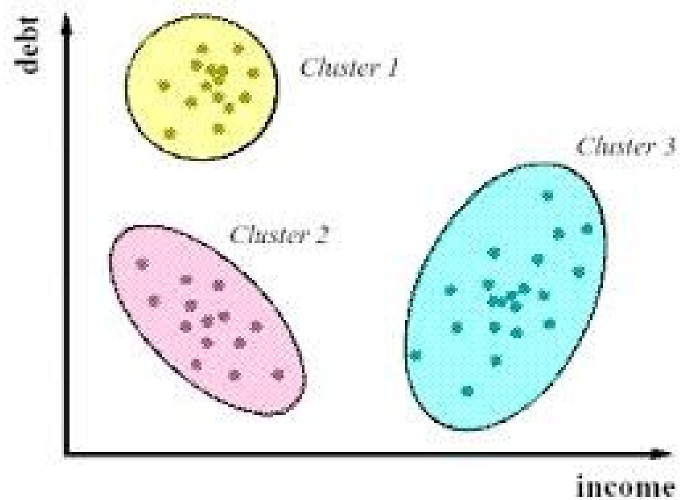
```
+--------+--------------+-------+------------+---------+----------+
| weekDay|distanceCenter|rentals|weekDayIndex| features|prediction|
+--------+--------------+-------+------------+---------+----------+
|  Monday|           0.1|  641.0|         1.0|[1.0,0.1]|     597.0|
|Saturday|           2.1|  129.0|         0.0|[0.0,2.1]|     102.0|
|Saturday|           1.5|  199.0|         0.0|[0.0,1.5]|     272.0|
|  Monday|           2.0|  231.0|         1.0|[1.0,2.0]|     120.0|
|  Sunday|           0.5|  393.0|         2.0|[2.0,0.5]|     597.0|
+--------+--------------+-------+------------+---------+----------+

Root Mean Squared Error (RMSE) on test data = 111.293
```

Unsupervised learning: clustering

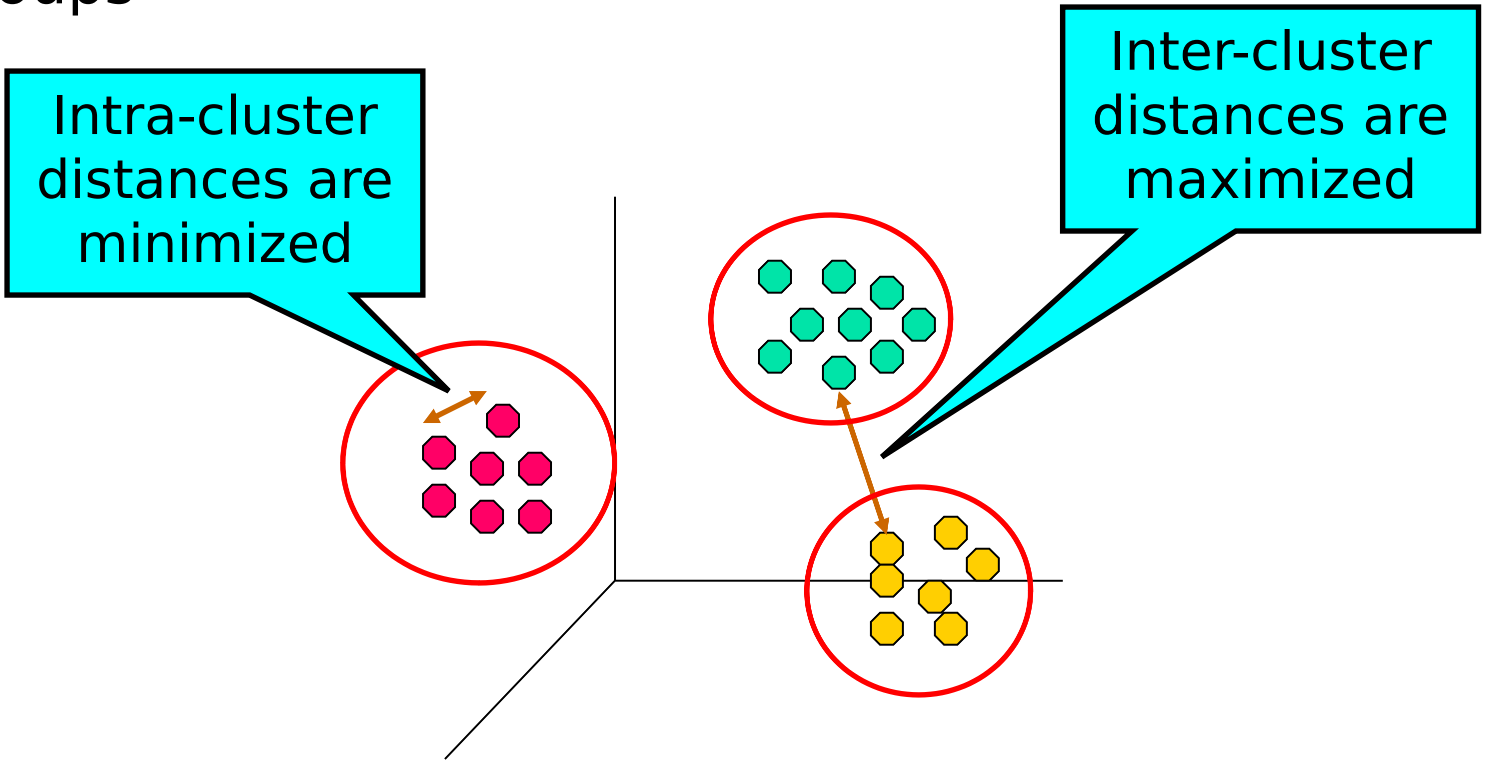
Unsupervised learning

- Learning “what normally happens”
- No output
- Clustering: Grouping similar instances
- Other applications: Summarization, Association Analysis



Clustering

- Finding groups of objects such that the objects in a group will be similar (or related) to one another and different from (or unrelated to) the objects in other groups

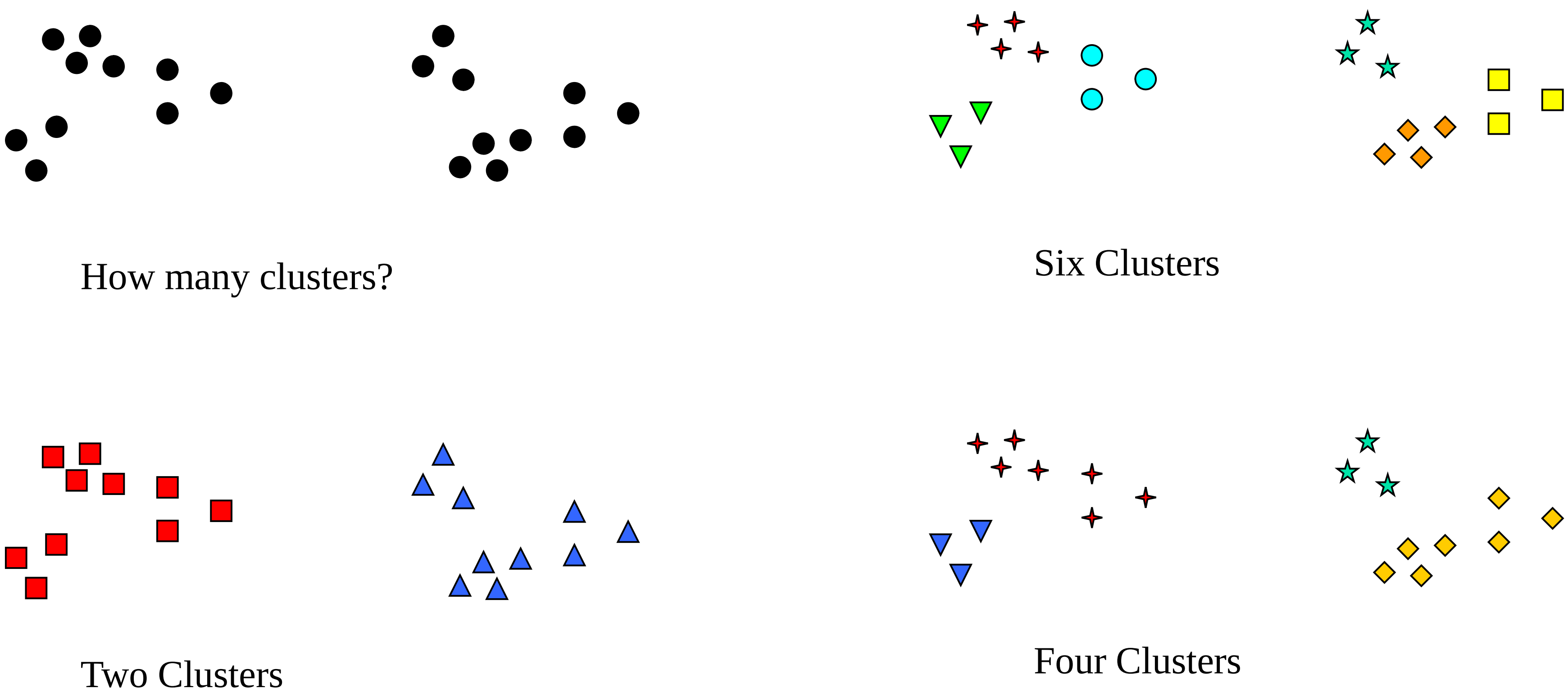


Clustering: use cases

- Finding users with similar behaviour without the need to set thresholds
- Finding anomalies in data
- Topic modelling
- ...

Clustering: ambiguity

- Notion of a cluster can be ambiguous



Clustering: weak points

- It is usually hard to define and measure success
- For big data, the problem is exacerbated
- Clustering in high dimensional space can create odd clusters
- *Curse of dimensionality*: as feature space expands in dimensionality, it become more sparse and not statistically significant

Clustering

"The validation of clustering structures is the most difficult and frustrating part of cluster analysis.

Without a strong effort in this direction, cluster analysis will remain a black art accessible only to those true believers who have experience and great courage."

From: Algorithms for Clustering Data, Jain and Dubes

Clustering algorithms

- Spark MLlib provides a (limited) set of clustering algorithms
  - K-means
  - Gaussian mixture models
  - Bisecting K-means
  - Latent Dirichlet Allocation
  - …

Clustering algorithms

- Spark MLlib provides a (limited) set of clustering algorithms
  - K-means
  - Gaussian mixture models
  - Bisecting K-means
  - Latent Dirichlet Allocation
  - …

These algos are shown in the slides

Clustering model scalability

Model	Statistical recommendation	Computation limits	Training size
k-means	features<100	Features x clusters < 10 million	No limit
Bisecting k-means	features<100	Features x clusters < 10 million	No limit
Gaussian Mixture models	features<100	Features x clusters< 10 million	No limit
Latent Dirichlet Allocation	An interpretable number	> 1000 clusters	No limit

Clustering

- Each clustering algorithm has its own parameters
- However, all the provided algorithms identify a set of groups of objects/clusters and assign each input object to one single cluster
- All the clustering algorithms available in Spark work only with numerical data
  - Categorical values must be mapped to integer values (i.e., numerical values)

Clustering

- The input of the MLlib clustering algorithms is a DataFrame containing a feature column
  - Data type: pyspark.ml.linalg.Vectors
- The clustering algorithm clusters the input records by considering only the content of features
  - The other columns, if any, are not considered

Clustering: Example

- Example: credit score
  - A set of customer profiles
  - For each customer we have savings and income
  - We want to automatically group customer in groups based on their characteristics

Savings	Income	User
15000	1000	Paolo
0	5000	Luca
20000	800	Martino
6000	1300	Mike
50000	2500	Francesca
2000	1100	Steve
700	1500	Maria
75000	0	Guido
4000	500	Roberta
7000	3000	Idilio
3000	900	Marco
6000	1200	Dena

Clustering: Example

Preprocess data for clustering

```
from pyspark.ml.feature import StandardScaler
from pyspark.ml.feature import VectorAssembler

data=spark.read.csv('credit_score_cluster.txt',header=True,inferSche
    ma=True)

va=VectorAssembler(inputCols=["Savings","Income"],
    outputCol="features")
assembledDF=va.transform(data)

scaler = StandardScaler(inputCol="features",
    outputCol="scaledFeatures", withStd=True, withMean=True)
scalerModel = scaler.fit(assembledDF)
scaledDF=scalerModel.transform(assembledDF)
```

Clustering: Example

Input DataFrame

Savings	Income	User
15000	1000	Paolo
0	5000	Luca
20000	800	Martino
6000	1300	Mike
50000	2500	Francesca
2000	1100	Steve
700	1500	Maria
75000	0	Guido
4000	500	Roberta
7000	3000	Idilio
3000	900	Marco
6000	1200	Dena

Preprocessed DataFrame

Savings	Income	User	features	scaledFeatures
15000	1000	Paolo	[15000.0,1000.0]	[-0.0312169519277...
0	5000	Luca	[0.0,5000.0]	[-0.6770849228476...
20000	800	Martino	[20000.0,800.0]	[0.18407237171215...
6000	1300	Mike	[6000.0,1300.0]	[-0.4187377344796...
50000	2500	Francesca	[50000.0,2500.0]	[1.47580831355180...
2000	1100	Steve	[2000.0,1100.0]	[-0.5909691933916...
700	1500	Maria	[700.0,1500.0]	[-0.6469444175380...
75000	0	Guido	[75000.0,0.0]	[2.55225493175152...
4000	500	Roberta	[4000.0,500.0]	[-0.5048534639356...
7000	3000	Idilio	[7000.0,3000.0]	[-0.3756798697517...
3000	900	Marco	[3000.0,900.0]	[-0.5479113286636...
6000	1200	Dena	[6000.0,1200.0]	[-0.4187377344796...

Clustering performance: silhouette

- Silhouette measures consistency within clusters of data.
- The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation).
- It ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters.
- If most objects have a high value, then the clustering configuration is appropriate.
- The silhouette can be calculated with any distance metric

See https://en.wikipedia.org/wiki/Silhouette_(clustering)

Clustering performance in MLlib

- As for classification and regression, in order to test the goodness of clusters there is an **evaluator**
- The evaluator is **ClusteringEvaluator** from **pyspark.ml.evaluation**
- The instantiated estimator has the method **evaluate()** that is applied to a dataframe
- It compares the clusters with the input data
- It computes the **silhouette measure**

Clustering performance in MLlib

- Parameters of **ClusteringEvaluator**:
  - **metricName:** string for metric name in evaluation. Only supports "silhouette"
  - **distanceMeasure**: param for distance measure to be used in evaluation. Supports "squaredEuclidean" (default) and "cosine"
  - **featuresCol**: input column with the features
  - **predictionCol**: input column with the cluster assignment

K-means clustering algorithm

K-means clustering algorithm

- K-means is one of the most popular clustering algorithms
- It is characterized by one important parameter
  - The number of clusters **K**
    - The choice of **K** is a complex operation
    - Often chosen by experimenting different values

K-means clustering algorithm

- It uses proximity by Euclidean clustering
- It is able to identify only spherical shaped clusters
- Each cluster is associated with a centroid (center point)
- Each point is assigned to the cluster with the closest centroid

K-means clustering algorithm

The basic algorithm is very simple:

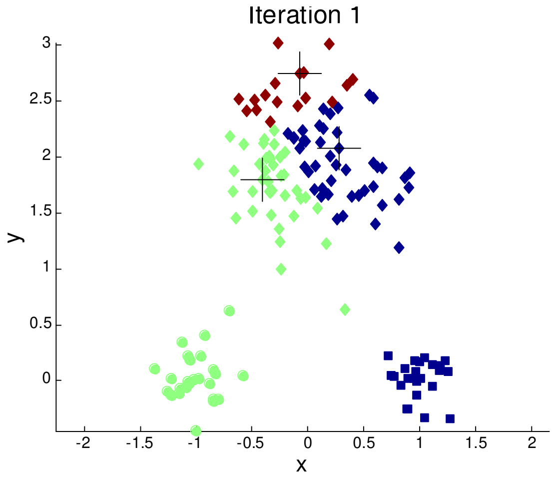
1: Select K points as the initial centroids.
2: **repeat**
3: Form K clusters by assigning all points to the closest centroid.
4: Recompute the centroid of each cluster.
5: **until** The centroids don't change

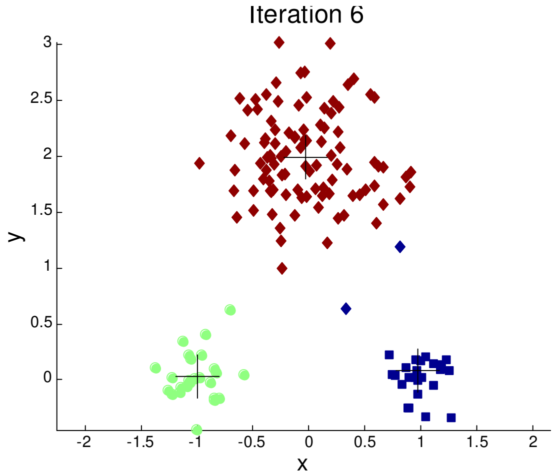
Most of the time, convergence happens in the first few iterations.

K-means clustering algorithm

Importance of choosing initial centroids

Case 1:



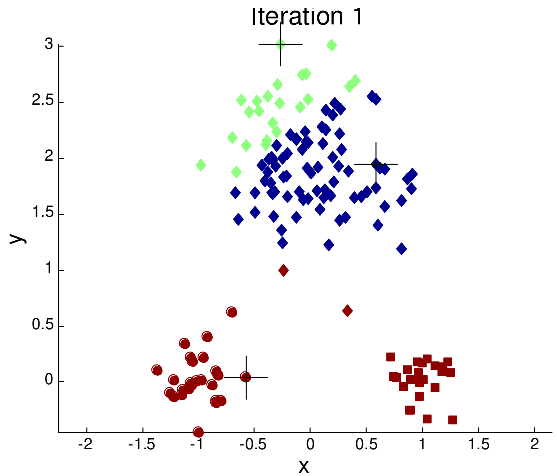


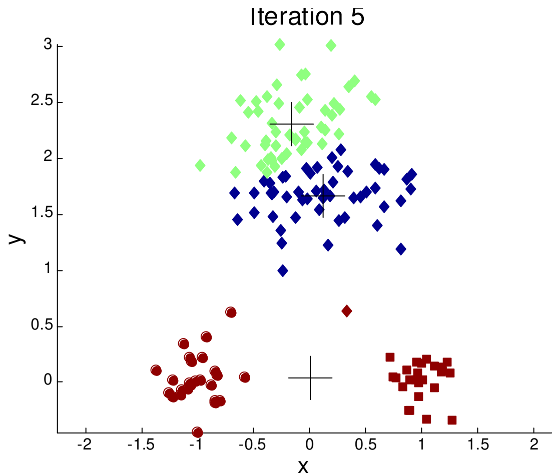
At convergence

K-means clustering algorithm

Importance of choosing initial centroids

Case 2:





At convergence

Specific performance measure for K-means

Evaluating K-means cluster: Sum of Squared Error (SSE)

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} dist^2(m_i, x)$$

- x is a data point in cluster C_i and m_i is the centroid of the cluster C_i
- Given two clusters, we can choose the one with the smallest error
- One easy way to reduce SSE is to increase K, the number of clusters

K-means clustering algorithm in MLlib

- The following slides show how to apply the **K-means algorithm** provided by Mllib
- The input dataset is a structured dataset with a fixed number of attributes
  - All the attributes are numerical attributes

K-means in MLlib

- Use the **Kmeans** estimator from **pyspark.ml.clustering** on a DataFrame
- Explicitly specify input columns **featuresCol** (vector)
- Output column:
  - **PredictionCol:** with its predicted cluster center

K-means in MLlib

- Parameters are:
  - **k**: number of clusters
  - **initMode**: initialization of the centroids. Supported are “random” and “k-means||”
  - **initSteps**: number of steps for “k-means||” initialization mode
  - **maxIter**: Total number of iterations over the data before stopping
  - **tol**: specifies threshold for keeping optimizing centroids position

K-means: summary of results

- K-means include a **summary** class to evaluate our model
- It includes information about clusters created and relative size (number of data for each cluster)
- We can also compute the sum of squared error within cluster which measure how close values are from their cluster centroid (**computeCost**)
- Given **k**, k-means try to minimize the **computeCost**

K-means: Example

- Example: credit score
  - Start from preprocessed dataframe

Preprocessed DataFrame

Savings	Income	User	features	scaledFeatures
15000	1000	Paolo	[15000.0,1000.0]	[-0.0312169519277...
0	5000	Luca	[0.0,5000.0]	[-0.6770849228476...
20000	800	Martino	[20000.0,800.0]	[0.18407237171215...
6000	1300	Mike	[6000.0,1300.0]	[-0.4187377344796...
50000	2500	Francesca	[50000.0,2500.0]	[1.47580831355180...
2000	1100	Steve	[2000.0,1100.0]	[-0.5909691933916...
700	1500	Maria	[700.0,1500.0]	[-0.6469444175380...
75000	0	Guido	[75000.0,0.0]	[2.55225493175152...
4000	500	Roberta	[4000.0,500.0]	[-0.5048534639356...
7000	3000	Idilio	[7000.0,3000.0]	[-0.3756798697517...
3000	900	Marco	[3000.0,900.0]	[-0.5479113286636...
6000	1200	Dena	[6000.0,1200.0]	[-0.4187377344796...

K-means: Example

```
from pyspark.ml.clustering import KMeans


# Trains a k-means model.
kmeans = KMeans(k=3,featuresCol="scaledFeatures",initMode="k-
    means||")
model = kmeans.fit(scaledDF)

# Make predictions
predictionsDF = model.transform(scaledDF)
```

K-means: Example

Preprocessed DataFrame

Savings	Income	User	features	scaledFeatures
15000	1000	Paolo	[15000.0,1000.0]	[-0.0312169519277...
0	5000	Luca	[0.0,5000.0]	[-0.6770849228476...
20000	800	Martino	[20000.0,800.0]	[0.18407237171215...
6000	1300	Mike	[6000.0,1300.0]	[-0.4187377344796...
50000	2500	Francesca	[50000.0,2500.0]	[1.47580831355180...
2000	1100	Steve	[2000.0,1100.0]	[-0.5909691933916...
700	1500	Maria	[700.0,1500.0]	[-0.6469444175380...
75000	0	Guido	[75000.0,0.0]	[2.55225493175152...
4000	500	Roberta	[4000.0,500.0]	[-0.5048534639356...
7000	3000	Idilio	[7000.0,3000.0]	[-0.3756798697517...
3000	900	Marco	[3000.0,900.0]	[-0.5479113286636...
6000	1200	Dena	[6000.0,1200.0]	[-0.4187377344796...

Output DataFrame with clusters

Savings	Income	User	features	scaledFeatures	prediction
15000	1000	Paolo	[15000.0,1000.0]	[-0.0312169519277...	0
0	5000	Luca	[0.0,5000.0]	[-0.6770849228476...	1
20000	800	Martino	[20000.0,800.0]	[0.18407237171215...	0
6000	1300	Mike	[6000.0,1300.0]	[-0.4187377344796...	0
50000	2500	Francesca	[50000.0,2500.0]	[1.47580831355180...	2
2000	1100	Steve	[2000.0,1100.0]	[-0.5909691933916...	0
700	1500	Maria	[700.0,1500.0]	[-0.6469444175380...	0
75000	0	Guido	[75000.0,0.0]	[2.55225493175152...	2
4000	500	Roberta	[4000.0,500.0]	[-0.5048534639356...	0
7000	3000	Idilio	[7000.0,3000.0]	[-0.3756798697517...	1
3000	900	Marco	[3000.0,900.0]	[-0.5479113286636...	0
6000	1200	Dena	[6000.0,1200.0]	[-0.4187377344796...	0

K-means: Example

```
from pyspark.ml.evaluation import ClusteringEvaluator

# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
print("Size of the clusters: ", model.summary.clusterSizes)

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

silhouette = evaluator.evaluate(predictionsDF)

print("Silhouette with squared euclidean distance = " + str(silhouette))

print("SSE: ",model.computeCost(predictionsDF))
```

K-means: Example

Output DataFrame with clusters

```
+-------+------+---------+---------------+--------------------+----------+
|Savings|Income|     User|       features|      scaledFeatures|prediction|
+-------+------+---------+---------------+--------------------+----------+
|  15000|  1000|    Paolo|[15000.0,1000.0]|[-0.0312169519277...|         0|
|      0|  5000|     Luca|    [0.0,5000.0]|[-0.6770849228476...|         1|
|  20000|   800|  Martino| [20000.0,800.0]|[0.18407237171215...|         0|
|   6000|  1300|     Mike| [6000.0,1300.0]|[-0.4187377344796...|         0|
|  50000|  2500|Francesca|[50000.0,2500.0]|[1.47580831355180...|         2|
|   2000|  1100|    Steve| [2000.0,1100.0]|[-0.5909691933916...|         0|
|    700|  1500|    Maria|  [700.0,1500.0]|[-0.6469444175380...|         0|
|  75000|     0|    Guido|   [75000.0,0.0]|[2.55225493175152...|         2|
|   4000|   500|  Roberta|  [4000.0,500.0]|[-0.5048534639356...|         0|
|   7000|  3000|   Idilio| [7000.0,3000.0]|[-0.3756798697517...|         1|
|   3000|   900|    Marco|  [3000.0,900.0]|[-0.5479113286636...|         0|
|   6000|  1200|     Dena| [6000.0,1200.0]|[-0.4187377344796...|         0|
+-------+------+---------+---------------+--------------------+----------+
```

Standard output

```
Cluster Centers:
[-0.37191231 -0.39159297]
[-0.5263824   1.80071098]
[ 2.01403162 -0.2343391 ]

Size of the clusters:  [8, 2, 2]

Silhouette with squared euclidean distance = -0.09641007707651222

SSE:  4.404936191705298
```

K-means: Example

Assign new data to existing clusters

Savings	Income	User
10000	1860	Mariana
4500	1100	Nicola
27000	1000	Davide

K-means: Example

```
# Load New data
dataNewDF=spark.read.csv('credit_score_cluster_Test.txt',header=True,inferSchema=True)

assembledNewDF=va.transform(dataNewDF)
scaledNewDF=scalerModel.transform(assembledNewDF)

# Make predictions
predictionsNewDF = model.transform(scaledNewDF)
```

K-means: Example

Assign new data to existing clusters

```
+-------+------+-------+
|Savings|Income|   User|
+-------+------+-------+
|  10000|  1860|Mariana|
|   4500|  1100| Nicola|
|  27000|  1000| Davide|
+-------+------+-------+
```

Output dataframe with associated clusters

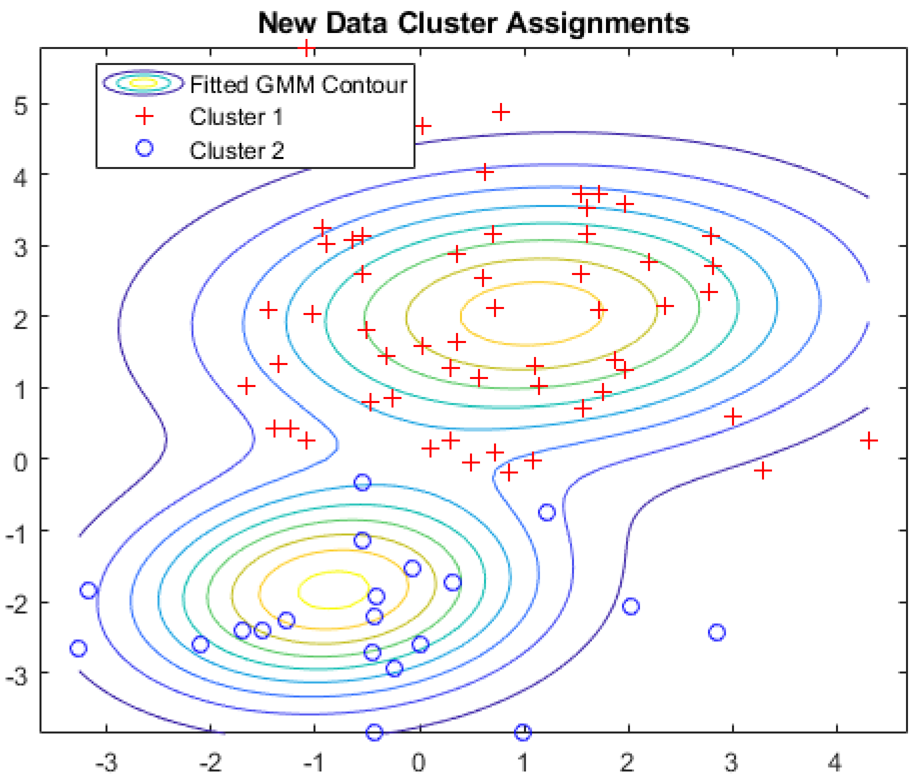
```
+-------+------+-------+----------------+--------------------+----------+
|Savings|Income|   User|        features|      scaledFeatures|prediction|
+-------+------+-------+----------------+--------------------+----------+
|  10000|  1860|Mariana|[10000.0,1860.0]|[-0.2465062755677...|         0|
|   4500|  1100| Nicola| [4500.0,1100.0]|[-0.4833245315716...|         0|
|  27000|  1000| Davide|[27000.0,1000.0]|[0.48547742480807...|         0|
+-------+------+-------+----------------+--------------------+----------+
```

Gaussian mixture model

Gaussian mixture model

- Gaussian mixture models (GMM) are another popular clustering algorithm
- It assumes each cluster produces data based upon random draws from a (multi-dimensional) Gaussian distribution
- Clusters should be less likely to have data at the edge
- Each Gaussian cluster has its own mean and standard deviation
- Allow for a more nuanced cluster associated with probability

Gaussian mixture model



GMM in MLlib

- Use the **GaussianMixture** estimator from **pyspark.ml.clustering** on a DataFrame
- Explicitly specify input columns **featuresCol** (vector)
- Output column:
  - **PredictionCol:** with its predicted cluster
  - **ProbabilityCol**: probability of each cluster

GMM in MLlib

- Parameters are:
  - **k**: number of clusters that you would like to end up with
  - **maxIter**: Total number of iterations over the data before stopping
  - **tol**: specifies threshold for keeping optimizing weights of Gaussian mixtures

GMM: summary of results

- Also GMM include a **summary** class to evaluate our model
- It includes information about weights of the Gaussian Mixtures and clusters created and relative size (number of data for each cluster)

GMM: Example

- Example: credit score
  - Start from preprocessed dataframe

Preprocessed DataFrame

```
+-------+------+---------+---------------+-------------------+
|Savings|Income|     User|       features|     scaledFeatures|
+-------+------+---------+---------------+-------------------+
|  15000|  1000|    Paolo|[15000.0,1000.0]|[-0.0312169519277...|
|      0|  5000|     Luca|    [0.0,5000.0]|[-0.6770849228476...|
|  20000|   800|  Martino| [20000.0,800.0]|[0.18407237171215...|
|   6000|  1300|     Mike| [6000.0,1300.0]|[-0.4187377344796...|
|  50000|  2500|Francesca|[50000.0,2500.0]|[1.47580831355180...|
|   2000|  1100|    Steve| [2000.0,1100.0]|[-0.5909691933916...|
|    700|  1500|    Maria|  [700.0,1500.0]|[-0.6469444175380...|
|  75000|     0|    Guido|   [75000.0,0.0]|[2.55225493175152...|
|   4000|   500|  Roberta|  [4000.0,500.0]|[-0.5048534639356...|
|   7000|  3000|   Idilio| [7000.0,3000.0]|[-0.3756798697517...|
|   3000|   900|    Marco|  [3000.0,900.0]|[-0.5479113286636...|
|   6000|  1200|     Dena| [6000.0,1200.0]|[-0.4187377344796...|
+-------+------+---------+---------------+-------------------+
```

GMM: Example

```
from pyspark.ml.clustering import GaussianMixture


# Trains a GMM model.
gmm = GaussianMixture(k=3,featuresCol="scaledFeatures")
model = gmm.fit(scaledDF)

# Make predictions
predictionsDF = model.transform(scaledDF)
predictionsDF.show(truncate=False)
```

GMM: Example

Preprocessed DataFrame

Savings	Income	User	features	scaledFeatures
15000	1000	Paolo	[15000.0,1000.0]	[-0.0312169519277...
0	5000	Luca	[0.0,5000.0]	[-0.6770849228476...
20000	800	Martino	[20000.0,800.0]	[0.18407237171215...
6000	1300	Mike	[6000.0,1300.0]	[-0.4187377344796...
50000	2500	Francesca	[50000.0,2500.0]	[1.47580831355180...
2000	1100	Steve	[2000.0,1100.0]	[-0.5909691933916...
700	1500	Maria	[700.0,1500.0]	[-0.6469444175380...
75000	0	Guido	[75000.0,0.0]	[2.55225493175152...
4000	500	Roberta	[4000.0,500.0]	[-0.5048534639356...
7000	3000	Idilio	[7000.0,3000.0]	[-0.3756798697517...
3000	900	Marco	[3000.0,900.0]	[-0.5479113286636...
6000	1200	Dena	[6000.0,1200.0]	[-0.4187377344796...

Output DataFrame with clusters and probabilities

Savings	Income	User	features	scaledFeatures	prediction	probability
15000	1000	Paolo	[15000.0,1000.0]	[-0.031216951927791736,-0.4193436518555962]	0	[0.9997461383843239,2.5339239436132E-4,4.6922131486960353E-7]
0	5000	Luca	[0.0,5000.0]	[-0.6770849228476208,2.5407291847721423]	1	[7.67384821545586E-15,0.9999999999999847,7.67384821545586E-15]
20000	800	Martino	[20000.0,800.0]	[0.18407237171215127,-0.5673472936869831]	0	[0.9997781944193962,2.2180558059577122E-4,8.08697666663197E-15]
6000	1300	Mike	[6000.0,1300.0]	[-0.41873773447968915,-0.1973381891085158]	2	[3.370858048561919E-5,5.087928478827556E-5,0.9999154121347261]
50000	2500	Francesca	[50000.0,2500.0]	[1.4758083135518092,0.690683618798058]	1	[7.671883651274481E-15,0.9999999999999847,7.671883651274481E-15]
2000	1100	Steve	[2000.0,1100.0]	[-0.5909691933916436,-0.3453418309399027]	2	[0.09756437460176307,2.851666521106474E-6,0.9024327737317159]
700	1500	Maria	[700.0,1500.0]	[-0.6469444175380287,-0.04933454727712887]	2	[1.414987519906597E-10,8.114221331490605E-5,0.9999188576451863]
75000	0	Guido	[75000.0,0.0]	[2.5522549317515244,-1.159361861012531]	0	[0.9987350372281458,0.0012649627718535413,6.71135687338088E-16]
4000	500	Roberta	[4000.0,500.0]	[-0.5048534639356663,-0.7893527564340636]	2	[2.9326429379156076E-16,4.3649700681163853E-7,0.9999995635029929]
7000	3000	Idilio	[7000.0,3000.0]	[-0.37567986975170053,1.060692766458273]	1	[7.66296622775793E-15,0.9999999226529782,7.734701413156046E-8]
3000	900	Marco	[3000.0,900.0]	[-0.547911328663655,-0.49334547277128965]	2	[4.378058313062203E-6,1.107903359412824E-6,0.9999945140383275]
6000	1200	Dena	[6000.0,1200.0]	[-0.41873773447968915,-0.27134001002420927]	2	[0.020490483344952695,2.355096422577629E-5,0.9794859656908216]

GMM: Example

```
print("Gaussians weights shown as a DataFrame: ")
model.gaussiansDF.show(truncate=False)

print("Size of the clusters: ", model.summary.clusterSizes)

from pyspark.ml.evaluation import ClusteringEvaluator

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

silhouette = evaluator.evaluate(predictionsDF)
print("Silhouette with squared euclidean distance = " +
    str(silhouette))
```

GMM: Example

Output DataFrame with clusters

Savings	Income	User	features	scaledFeatures	prediction	probability
15000	1000	Paolo	[15000.0,1000.0]	[-0.031216951927791736,-0.4193436518555962]	0	[0.9997461383843239,2.5339239436132E-4,4.6922131486960353E-7]
0	5000	Luca	[0.0,5000.0]	[-0.6770849228476208,2.5407291847721423]	1	[7.67384821545586E-15,0.9999999999999847,7.67384821545586E-15]
20000	800	Martino	[20000.0,800.0]	[0.18407237171215127,-0.5673472936869831]	0	[0.9997781944193962,2.2180558059577122E-4,8.08697666663197E-15]
6000	1300	Mike	[6000.0,1300.0]	[-0.41873773447968915,-0.1973381891085158]	2	[3.370858048561919E-5,5.087928478827556E-5,0.9999154121347261]
50000	2500	Francesca	[50000.0,2500.0]	[1.4758083135518092,0.6906836618798058]	1	[7.671883651274481E-15,0.9999999999999847,7.671883651274481E-15]
2000	1100	Steve	[2000.0,1100.0]	[-0.5909691933916436,-0.3453418309399027]	2	[0.09756437460176307,2.851666521106474E-6,0.9024327737317159]
700	1500	Maria	[700.0,1500.0]	[-0.6469444175380287,-0.04933454727712887]	2	[1.414987519906597E-10,8.114221331490605E-5,0.9999188576451863]
75000	0	Guido	[75000.0,0.0]	[2.5522549317515244,-1.159361861012531]	0	[0.9987350372281458,0.0012649627718535413,6.71135687338088E-16]
4000	500	Roberta	[4000.0,500.0]	[-0.5048534639356663,-0.7893527564340636]	2	[2.9326429379156076E-16,4.3649700681163853E-7,0.9999995635029929]
7000	3000	Idilio	[7000.0,3000.0]	[-0.37567986975170053,1.060692766458273]	1	[7.66296622775793E-15,0.999999226529782,7.734701413156046E-8]
3000	900	Marco	[3000.0,900.0]	[-0.547911328663655,-0.49334547277128965]	2	[4.378058313062203E-6,1.107903359412824E-6,0.9999945140383275]
6000	1200	Dena	[6000.0,1200.0]	[-0.41873773447968915,-0.27134001002420927]	2	[0.020490483344952695,2.355096422577629E-5,0.9794859656908216]

Standard output

Gaussians shown as a DataFrame:

mean	cov
[0.8454226569010553,-0.7006389191006298]	1.3958406580303437 -0.37788956959180653 0.37788956959180653 0.10368397252492009
[0.1419810453423357,1.4292249890390458]	0.907877948120921 -0.570423391679312 -0.570423391679312 0.6420293496870142
[-0.520539730446735,-0.3581825057605061]	0.007135030253629733 -0.003040032956584862 -0.003040032956584862 0.056615613731226165

Size of the clusters: [3, 3, 6]
Silhouette with squared euclidean distance = 0.14433979161213353

GMM: Example

Assign new data to existing clusters

Savings	Income	User
10000	1860	Mariana
4500	1100	Nicola
27000	1000	Davide

GMM: Example

```
# Load New data
dataNewDF=spark.read.csv('credit_score_cluster_Test.txt',header=True,inferSchema=True)

assembledNewDF=va.transform(dataNewDF)
scaledNewDF=scalerModel.transform(assembledNewDF)

# Make predictions
predictionsNewDF = model.transform(scaledNewDF)
```

GMM: Example

Assign new data to existing clusters

```
+-------+------+-------+
|Savings|Income|   User|
+-------+------+-------+
|  10000|  1860|Mariana|
|   4500|  1100| Nicola|
|  27000|  1000| Davide|
+-------+------+-------+
```

Output dataframe with associated clusters

```
+-------+------+-------+--------------+--------------------+----------+--------------------+
|Savings|Income|   User|      features|      scaledFeatures|prediction|         probability|
+-------+------+-------+--------------+--------------------+----------+--------------------+
|  10000|  1860|Mariana|[10000.0,1860.0]|[-0.2465062755677...|         2|[9.83009952163732...|
|   4500|  1100| Nicola| [4500.0,1100.0]|[-0.4833245315716...|         2|[1.32063772202194...|
|  27000|  1000| Davide|[27000.0,1000.0]|[0.48547742480807...|         0|[0.99999756357385...|
+-------+------+-------+--------------+--------------------+----------+--------------------+
```