Introduction to Big Data

ADVANCED ANALYTICS IN SPARK

Le Ngoc Thanh – Nguyen Ngoc Thao
{lnthanh, nnthao}@fit.hcmus.edu.vn

Outline

- Advanced analytics process
- Data analytics with MLlib
- Graph analytics with GraphFrames

Advanced analytics process

Classification

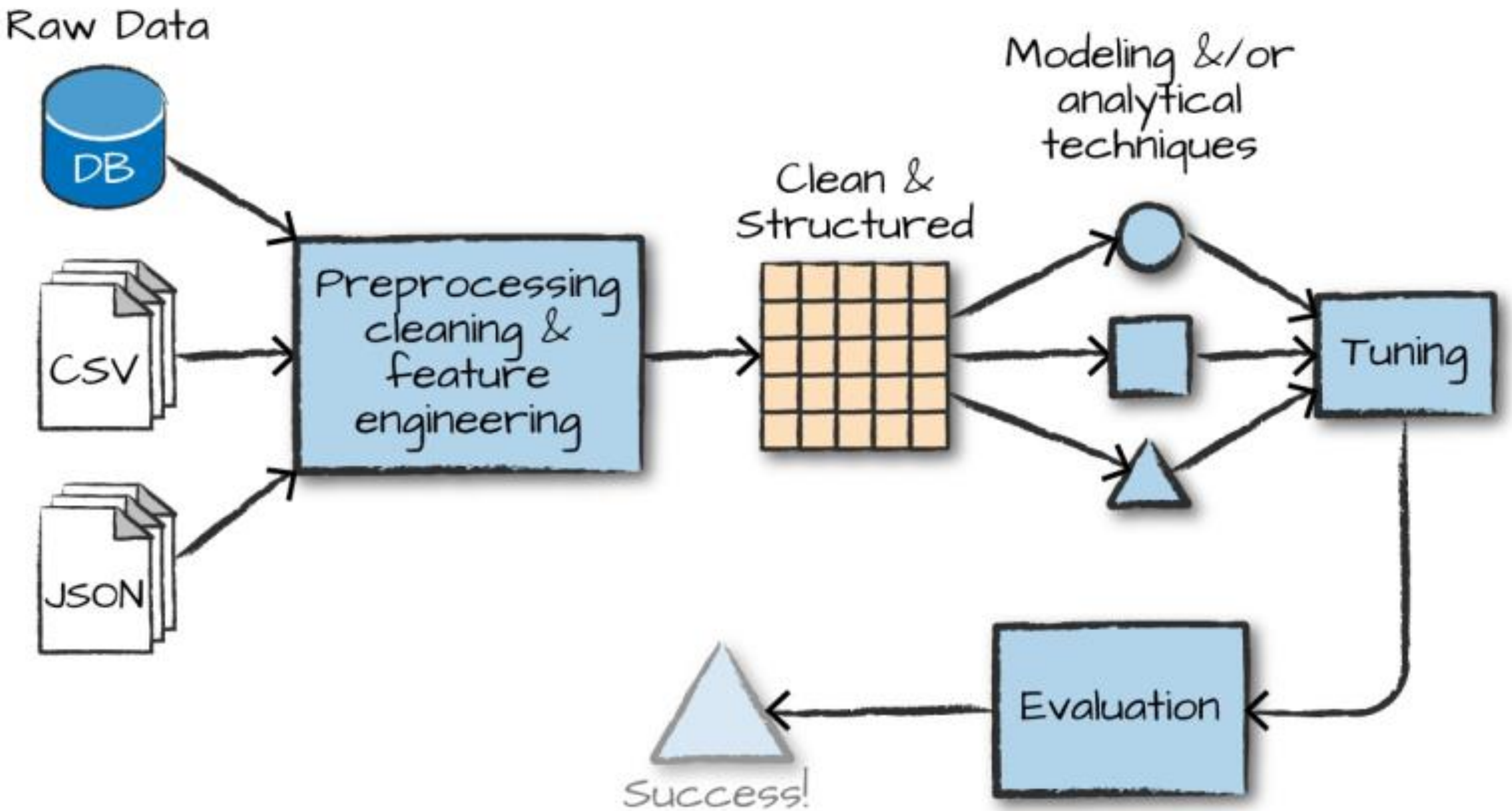
Recommendation

Data processing

Graph analytics

Clustering

Advanced analytics process



Advanced analytics process

- Data collection: gather the data needed to train an algorithm
  - Spark can handle a variety of data sources and data sizes.
- Data cleaning: resolve data ambiguity and inconsistencies
  - MLlib offers a wide variety of APIS for data preprocessing, e.g., lill missing entries, correct invalid values, and solve conflictions.
- Feature engineering: convert data to a form suitable for ML algorithms
  - Add/delete attributes, normalize / discretize values of an attribute, handle categorical variables, etc.
  - Variables in MLlib are usually **vectors of doubles**.

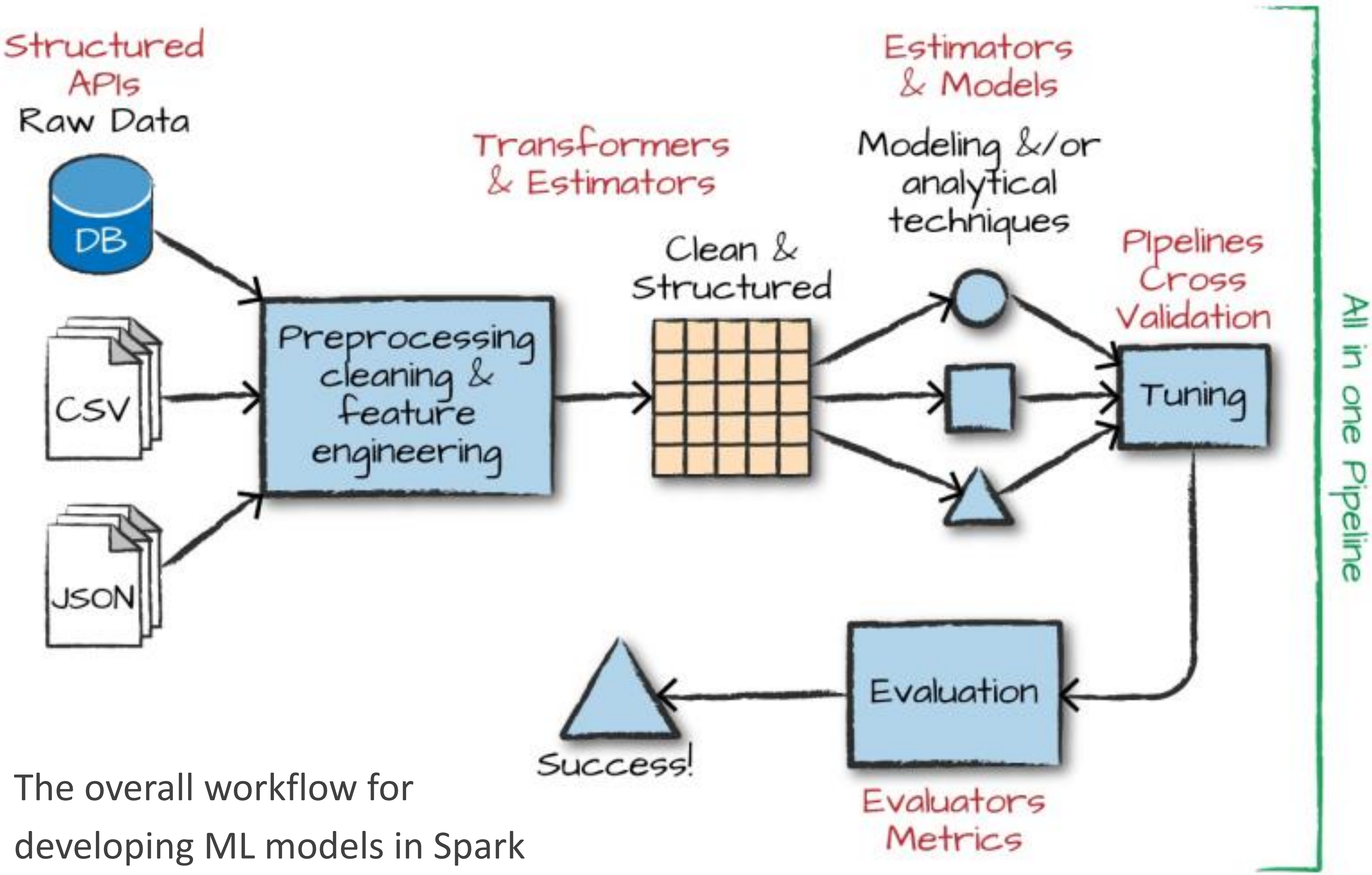
Advanced analytics process

- Model training: build a model to predict the correct output, given some input
  - Models can be used to gain insights or to make future predictions
  - E.g., spam emails classification $\rightarrow$ the model learns certain words should have more influence than others
- Model tuning and evaluation: try out different hyper-parameters and compare model variations without overfitting
  - Training set vs. Validation set vs. Test set
- Leveraging the model and/or insights

Advanced analytics with Spark

- **`MLlib`** provides interfaces for preprocessing data, training and tuning large-scale models, and using them in production
  - **`org.apache.spark.ml`** for **`DataFrames`** manipulation
  - `org.apache.spark.mllib` for Spark's low-level **`RDD`** APIs
- `MLlib` is mainly designed for scalability issues.
  - Single-machine tools are usually complementary to MLlib, e.g., scikit-learn, TensorFlow, R, etc.

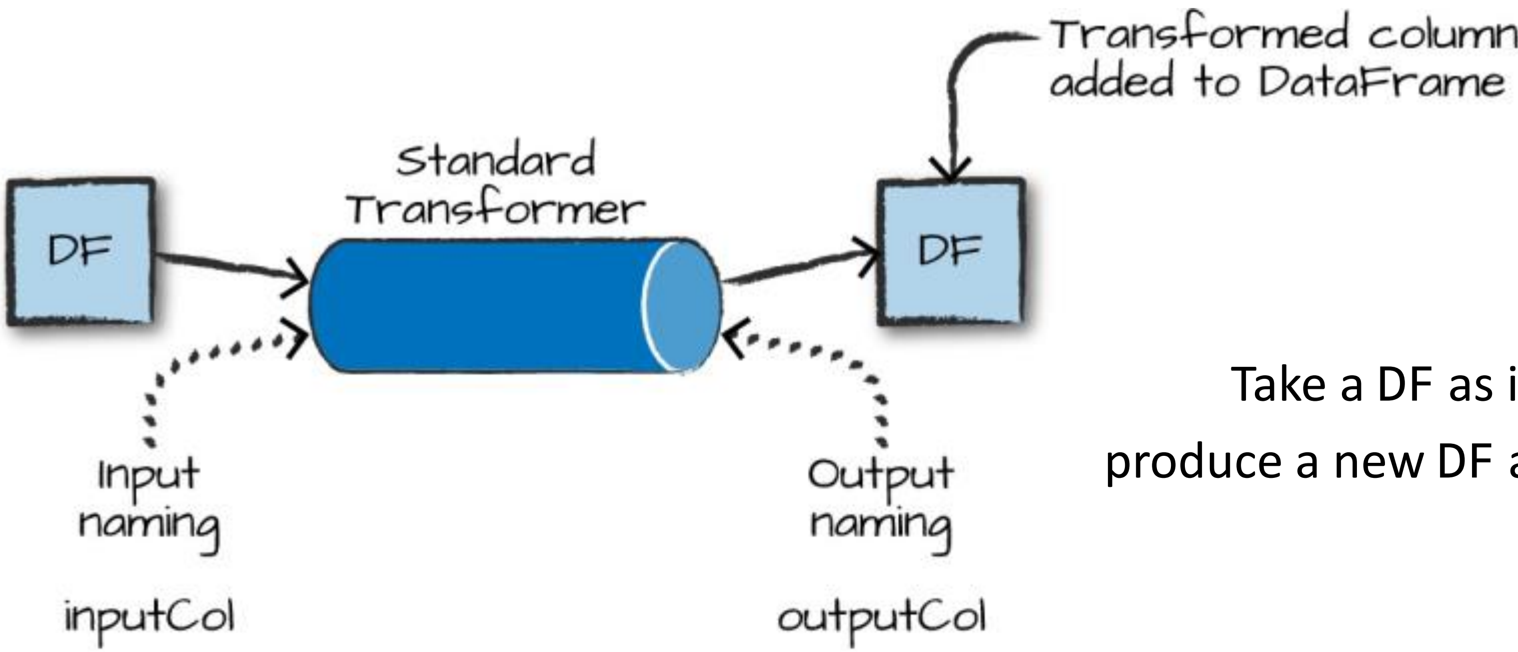
MLlib fundamental structural types



The overall workflow for developing ML models in Spark

MLlib Types: Transformers

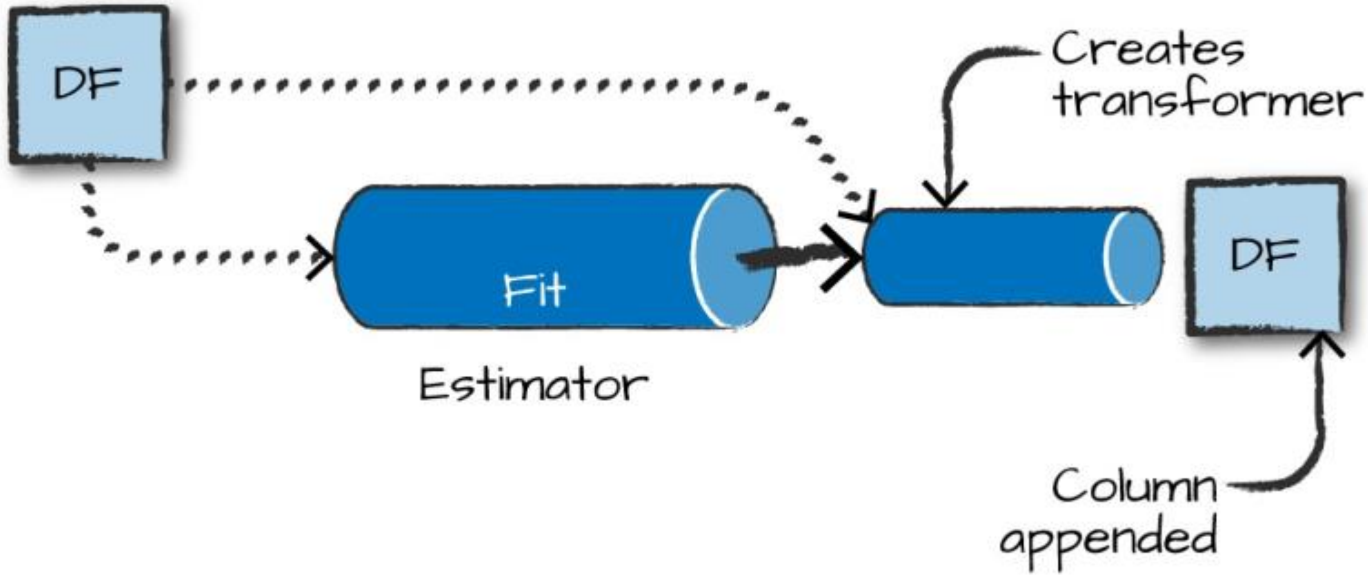
- Transformers are functions that convert raw data in some way
  - E.g., create a new interaction variable (from two other variables), normalize a column, or change an Integer into a Double type, etc.
- Primarily used in preprocessing and feature engineering



Take a DF as input and produce a new DF as output

MLlib Types: Estimators

- Estimators can be a kind of transformers initialized with data.
  - E.g., numerical normalization: initialize the transformation with some information about the current values



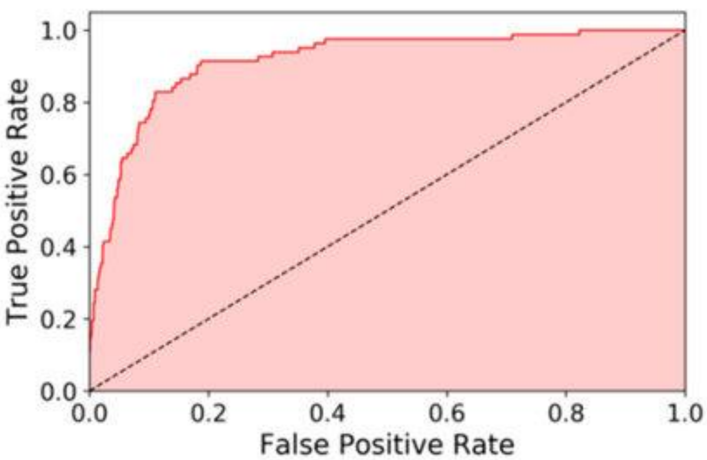
- Algorithms that allow users to train a model from data are also referred to as estimators.

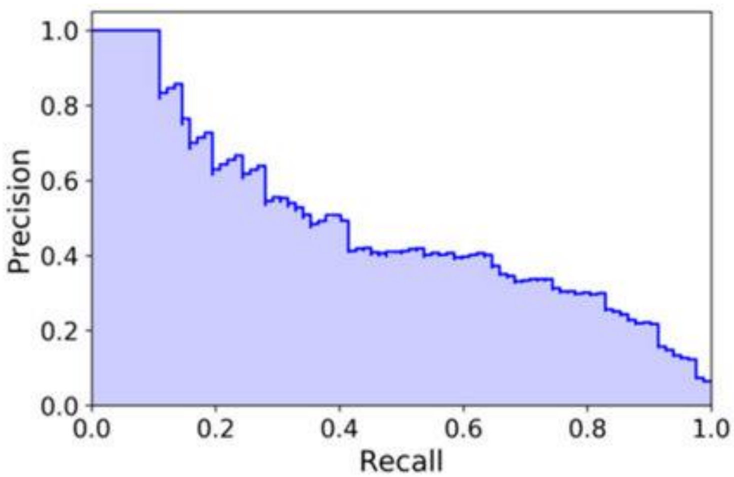
MLlib Types: Evaluators

- Evaluators determine how a given model performs following the criteria specified.

BinaryClassificationEvaluator RegressionEvaluator

RankingEvaluator MultilabelClassificationEvaluator





BinaryClassificationEvaluator: The metric can be areaUnderROC or areaUnderPR.

MLlib low-level data types

- There are also several lower-level data types to work with.
- `Vector` is the most common.
  - Features passed into a ML model should be vectors of `Doubles`.

```
# Vectors are either dense or sparse.
from pyspark.ml.linalg import Vectors
denseVec = Vectors.dense(1.0, 2.0, 3.0)
size = 3
idx = [1, 2]  # locations of non-zero elements in vector
values = [2.0, 3.0]
sparseVec = Vectors.sparse(size, idx, values)
```

```
print(denseVec)
```

```
[1.0,2.0,3.0]
```

```
print(sparseVec)
```

```
(3,[1,2],[2.0,3.0])
```

MLlib in action: Preparation

- Let’s read a synthetic data set and see a sample

```
df = spark.read.json("/data/simple-ml")
df.orderBy("value2")
```

```
+-----+----+------+------------------+
|color| lab|value1|            value2|
+-----+----+------+------------------+
|green|good|     1|14.386294994851129|
...
|  red| bad|    16|14.386294994851129|
|green|good|    12|14.386294994851129|
+-----+----+------+------------------+
```

110 lines (110 sloc) | 7.38 KB


```
{"lab":"good","color":"green","value1":1,"value2":14.386294994851129}
{"lab":"bad","color":"blue","value1":8,"value2":14.386294994851129}
{"lab":"bad","color":"blue","value1":12,"value2":14.386294994851129}
{"lab":"good","color":"green","value1":15,"value2":38.97187133755819}
{"lab":"good","color":"green","value1":12,"value2":14.386294994851129}
{"lab":"bad","color":"green","value1":16,"value2":14.386294994851129}
{"lab":"good","color":"red","value1":35,"value2":14.386294994851129}
{"lab":"bad","color":"red","value1":1,"value2":38.97187133755819}
{"lab":"bad","color":"red","value1":2,"value2":14.386294994851129}
{"lab":"bad","color":"red","value1":16,"value2":14.386294994851129}
```

MLlib in action: Preparation

- Feature engineering with transformers of RFormula
- RFormula implements the transforms required for fitting a dataset against an R model formula.
  - An explanation of the below formula can be found here.

```
from pyspark.ml.feature import RFormula
supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")
fittedRF = supervised.fit(df)
preparedDF = fittedRF.transform(df)
```

```
--------+----------------------------------------------------------------------+-----+
        |features                                                              |label|
--------+----------------------------------------------------------------------+-----+
94851129|(10,[1,2,3,5,8],[1.0,1.0,14.386294994851129,1.0,14.386294994851129])|1.0  |
94851129|(10,[2,3,6,9],[8.0,14.386294994851129,8.0,14.386294994851129])      |0.0  |
94851129|(10,[2,3,6,9],[12.0,14.386294994851129,12.0,14.386294994851129])    |0.0  |
3755819 |(10,[1,2,3,5,8],[1.0,15.0,38.97187133755819,15.0,38.97187133755819])|1.0  |
--------+----------------------------------------------------------------------+-----+
```

MLlib in action: Application

- Create a simple test set based on a random split of the data

```
# split the original dataset with ratio 7:3
train, test = preparedDF.randomSplit([0.7, 0.3])
```

.

- Apply some ML algorithm with specified hyperparameters

```
# apply logistic regression with default hyperparameters
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(labelCol="label", featuresCol="features")
print(lr.explainParams())

fittedLR = lr.fit(train)
fittedLR.transform(train).select("label", "prediction")
```

```
+-----+----------+
|label|prediction|
+-----+----------+
|  0.0|       0.0|
...
|  0.0|       0.0|
+-----+----------+
```

MLlib in action: Pipelining

- Set up a dataflow of relevant transformations that results an estimator automatically tuned following some specifications
  - Setting up another pipeline must make a new instance of the model.
- Let's create the base stages in our pipeline

```
# # split the original dataset with ratio 7:3
train, test = df.randomSplit([0.7, 0.3])
# define the transformer by RFormula, whose expression is still undefined
rForm = RFormula()
# apply logistic regression with default hyperparameters
lr = LogisticRegression().setLabelCol("label").setFeaturesCol("features")
# define the pipeline including two consecutive stages: transform and model
from pyspark.ml import Pipeline
stages = [rForm, lr]
pipeline = Pipeline().setStages(stages)
```

MLlib in action: Pipelining

- Define hyperparameter combinations to train model variants

```
# create 12 combinations of (RFormula, ElasticNet and regularization params)
from pyspark.ml.tuning import ParamGridBuilder
params = ParamGridBuilder().addGrid(rForm.formula, ["lab ~ . + color:value1",
                                                   "lab ~ . + color:value1 + color:value2"])\
            .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
            .addGrid(lr.regParam, [0.1, 2.0])\
            .build()
```

- Compare multiple models on the same evaluation metric

```
# use BinaryClassificationEvaluator with areaUnderROC metric on “label”
from pyspark.ml.evaluation import BinaryClassificationEvaluator
evaluator = BinaryClassificationEvaluator()
                  .setMetricName("areaUnderROC")\
                  .setRawPredictionCol("prediction")\
                  .setLabelCol("label")
```

MLlib in action: Pipelining

- Create a validation set to avoid overfitting

```
# create a validation set which accounts for 25% amount of original data
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit().setTrainRatio(0.75)
                            .setEstimatorParamMaps(params)\
                            .setEstimator(pipeline)\
                            .setEvaluator(evaluator)
```

- Run the pipeline to test each model variant on the validation set and finally evaluate how it performs on the test set

```
tvsFitted = tvs.fit(train)
evaluator.evaluate(tvsFitted.transform(test))
```

- It can also extract a training summary for some models from the pipeline, cast it to the proper type, and print out results.

MLlib in action: Persist and Apply

- Persist the trained model to disk for later prediction purposes

```
tvsFitted.write.overwrite().save("/tmp/modelLocation")
```

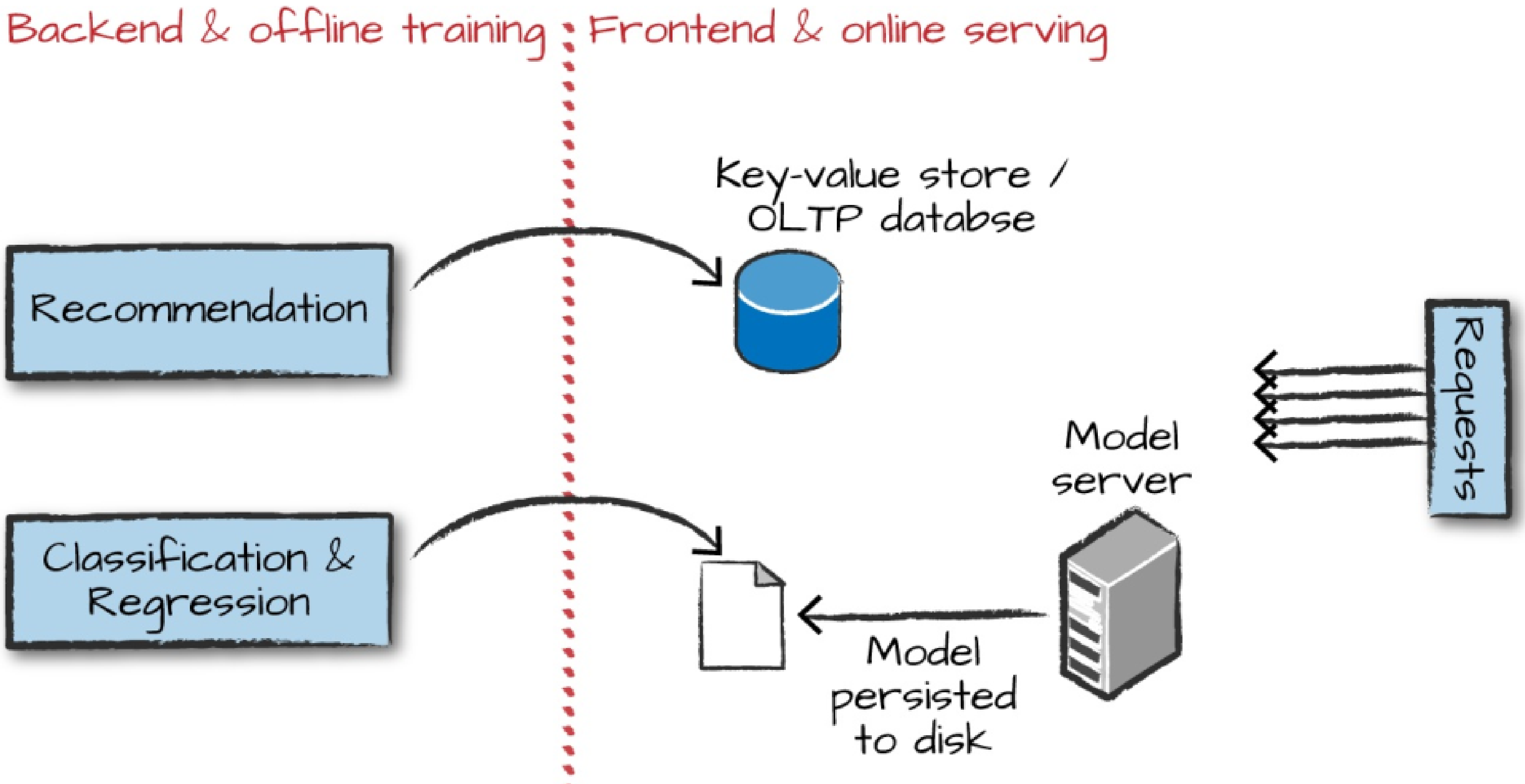
- Load the written model into another Spark program
  - Use a “model” version of the particular algorithm that built the model

```
CrossValidator → CrossValidatorModel
                  LogisticRegression → LogisticRegression
    TrainValidationSplit → TrainValidationSplitModel
```

```
from pyspark.ml. tuning import TrainValidationSplitModel
model = TrainValidationSplitModel.load("/tmp/modelLocation")
model.transform(test)
```

Deploying patterns in Spark

- There are several deployment patterns for putting ML models into production.



Deploying patterns in Spark

- Train the model offline and then supply it with offline data
  - Offline data is stored for analysis, not to get an answer from quickly
  - Spark is well suited to this sort of deployment
- Train the model offline, persist it to disk, and use it for serving
  - Spark cannot serve as a low-latency solution (the overhead of starting up a Spark job can be high, even if not running on a cluster)
  - Not parallelize well $\rightarrow$ manually put a load balancer in front of multiple model replicas and build out some REST API integration
  - No standards currently exist for this sort of model serving

Deploying patterns in Spark

- Train the model offline and put the results into a database (usually a key-value store)
  - Work well for recommendation but badly for classification/regression (i.e., not just look up a value for a given user but must calculate one based on the input)
- Manually (or via some other software) convert the distributed model to one that run more quickly on a single machine
  - Work well when there is not too much manipulation of the raw data in Spark, hard to maintain over time
  - MLlib exports models to PMML, a common model interchange format
- Train the ML algorithm online and use it online
  - Possible with **Structured Streaming**, complex for some models

Data analytics with MLlib

Features transformations

- Consider the following toy datasets.

id	single	tuple	label
1	18.0	[6.0,4.0]	4.0
2	19.0	[5.0,3.0]	3.0
3	8.0	[4.0,2.0]	2.0
4	5.0	[8.0,1.0]	1.0
5	2.0	[7.0,5.0]	2.0

```
# a numerical dataset
numDF = spark.createDataFrame([
        (1, 18.0, Vectors.dense([6.0, 4.0]), 4.0),
        (2, 19.0, Vectors.dense([5.0, 3.0]), 3.0),
        (3, 8.0, Vectors.dense([4.0, 2.0]), 2.0),
        (4, 5.0, Vectors.dense([9.0, 1.0]), 1.0),
        (5, 2.0, Vectors.dense([9.0, 5.0]), 2.0)], ["id", "single", "tuple", "label"])
```

```
# a collection of sentences, each of which is in a separate record
stringDF = spark.createDataFrame([(1, "i love dogs"),
        (2, "i hate dogs and knitting"),
        (3, "knitting is my hobby and my passion")],
        ["id", "sentence"])
```

id	sentence
1	i love dogs
2	i hate dogs and knitting
3	knitting is my hobby and my passion

Transformations: Tokenizer and NGram

- **`Tokenizer`**: turn a string into lowercase and split it by white spaces

```python
from pyspark.ml.feature import HashingTF, IDF, Tokenizer
tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsDF = tokenizer.transform(stringDF)
```

```
+---+--------
|id |words
+---+------------------------------------------+
|1  |[i, love, dogs]                           |
|2  |[i, hate, dogs, and, knitting]            |
|3  |[knitting, is, my, hobby, and, my, passion]|
+---+------------------------------------------+
```

- **`NGram`**: create a sequence of n-grams (consecutive words)

```python
from pyspark.ml.feature import NGram
ngram = NGram(n=2, inputCol="words", outputCol="ngrams")
ngramDF = ngram.transform(wordsDF)
```

```
+---+--------
|id |ngrams
+---+---------------------------------------------------------------+
|1  |[i love, love dogs]                                            |
|2  |[i hate, hate dogs, dogs and, and knitting]                    |
|3  |[knitting is, is my, my hobby, hobby and, and my, my passion]|
+---+---------------------------------------------------------------+
```

Transformations: QuantileDiscretizer

- **`QuantileDiscretizer`**: take a column with continuous features and output equivalent binned categorical features

```python
from pyspark.ml.feature import MinMaxScaler
# compute summary statistics to create a MixMaxScaler
scaler = MinMaxScaler(inputCol="tuple", outputCol="scaled")
scalerModel = scaler.fit(numDF)
# rescale each feature to range [min, max]
scaledDF = scalerModel.transform(numDF)

# similar code structure for QuantileDiscretizer
from pyspark.ml.feature import QuantileDiscretizer
discretizer = QuantileDiscretizer(numBuckets=3,\
                inputCol="single", outputCol="discrete")
resultDF =  discretizer.fit(scaledDF).transform(scaledDF)
```

```
+---+-----------+--------+
| id|     scaled|discrete|
+---+-----------+--------+
|  1|[0.5,0.75] |     2.0|
|  2|[0.25,0.5] |     2.0|
|  3|[0.0,0.25] |     1.0|
|  4| [1.0,0.0] |     1.0|
|  5|[0.75,1.0] |     0.0|
+---+-----------+--------+
```

- **`MinMaxScaler`**: rescaling each feature to a specific range

Extraction: CountVectorizer

- **CountVectorizer**: convert a collection of text documents to vectors of token counts

```
from pyspark.ml.feature import CountVectorizer
# fit a CountVectorizerModel from the corpus to create a CountVectorizerModel
cv = CountVectorizer(inputCol="words", outputCol="features",\
                                                          vocabSize=4, minDF=2.0)
model = cv.fit(wordsDF)

# apply the CountVectorizerModel to DF
result = model.transform(wordsDF)
```

```
+---+----------------------------+-------------------------------+
|id |words                       |features                       |
+---+----------------------------+-------------------------------+
|1  |[i, love, dogs]             |(4,[1,2],[1.0,1.0])            |
|2  |[i, hate, dogs, and, knitting]|(4,[0,1,2,3],[1.0,1.0,1.0,1.0])|
|3  |[knitting, is, my, hobby, and,|(4,[0,3],[1.0,1.0])            |
+---+----------------------------+-------------------------------+
```

Extraction: CountVectorizer

- **`OneHotEncoder`**: map a categorical feature, served as a label index, to a binary vector of at most a single one-value
  - This one-value indicates the presence of a specific feature value from among the set of all feature values.

```
from pyspark.ml.feature import OneHotEncoder
# fit an OneHotEncoder to the label indices
encoder = OneHotEncoder(inputCols=["label"], outputCols=["encoded"])
encodedDF = encoder.fit(numDF)
# encode the indices
result = encodedDF.transform(numDF)
```

```
+---+-----+-------------+
| id|label|      encoded|
+---+-----+-------------+
|  1|  4.0|    (4,[],[])|
|  2|  3.0|(4,[3],[1.0])|
|  3|  2.0|(4,[2],[1.0])|
|  4|  1.0|(4,[1],[1.0])|
|  5|  2.0|(4,[2],[1.0])|
+---+-----+-------------+
```

Features selection: RFormula

- `RFormula`: select columns by an R model formula
- Spark currently supports a limited subset of the R operators
  - ~ separate target and terms
  - + concat terms, “+ 0” means removing intercept
  - - remove a term, “- 1” means removing intercept
  - : multiplication for numeric values or binarized categorical values
  - . all columns except target
- For example, the formula `y ~ a + b + a:b – 1`

  means model `y ~ w1*a + w2*b + w3*a*b`

  where `w1, w2, w3` are coefficients.

Features selection: RFormula

- Consider a `DF` that contains the attributes `clicked`, `country`, and `hour`.
- The RFormula `clicked ~ country + hour` means to predict `clicked` based on `country` and `hour`.

```
+---+-------+----+-------+--------------+-----+
|id |country|hour|clicked|features      |label|
+---+-------+----+-------+--------------+-----+
|7  |US     |18  |1.0    |[0.0,0.0,18.0]|1.0  |
|8  |CA     |12  |0.0    |[1.0,0.0,12.0]|0.0  |
|9  |NZ     |15  |0.0    |[0.0,1.0,15.0]|0.0  |
+---+-------+----+-------+--------------+-----+
```

```
from pyspark.ml.feature import Rformula
dataset = spark.createDataFrame([(7, "US", 18, 1.0), (8, "CA", 12, 0.0),\
                                 (9, "NZ", 15, 0.0)],\
                                 ["id", "country", "hour", "clicked"])
rformula = RFormula(formula="clicked ~ country + hour",\
                    featuresCol="features", labelCol="label")
output = rformula.fit(dataset).transform(dataset)
```

Frequent pattern mining

- Spark supports **`FPGrowth`** for frequent pattern mining and **`PrefixSpan`** for sequential pattern mining.
- Consider the following transactional dataset.

```
df = spark.createDataFrame([(0, ['pasta', 'lemon', 'bread', 'orange']),
                            (1, ['pasta', 'lemon']),
                            (2, ['pasta', 'orange', 'cake']),
                            (3, ['pasta', 'lemon', 'cake', 'orange’])
                            ], ["id", "items"])
```

```
+---+----------------------------+
|id |items                       |
+---+----------------------------+
|0  |[pasta, lemon, bread, orange]|
|1  |[pasta, lemon]              |
|2  |[pasta, orange, cake]       |
|3  |[pasta, lemon, cake, orange] |
+---+----------------------------+
```

Frequent pattern mining: FPGrowth

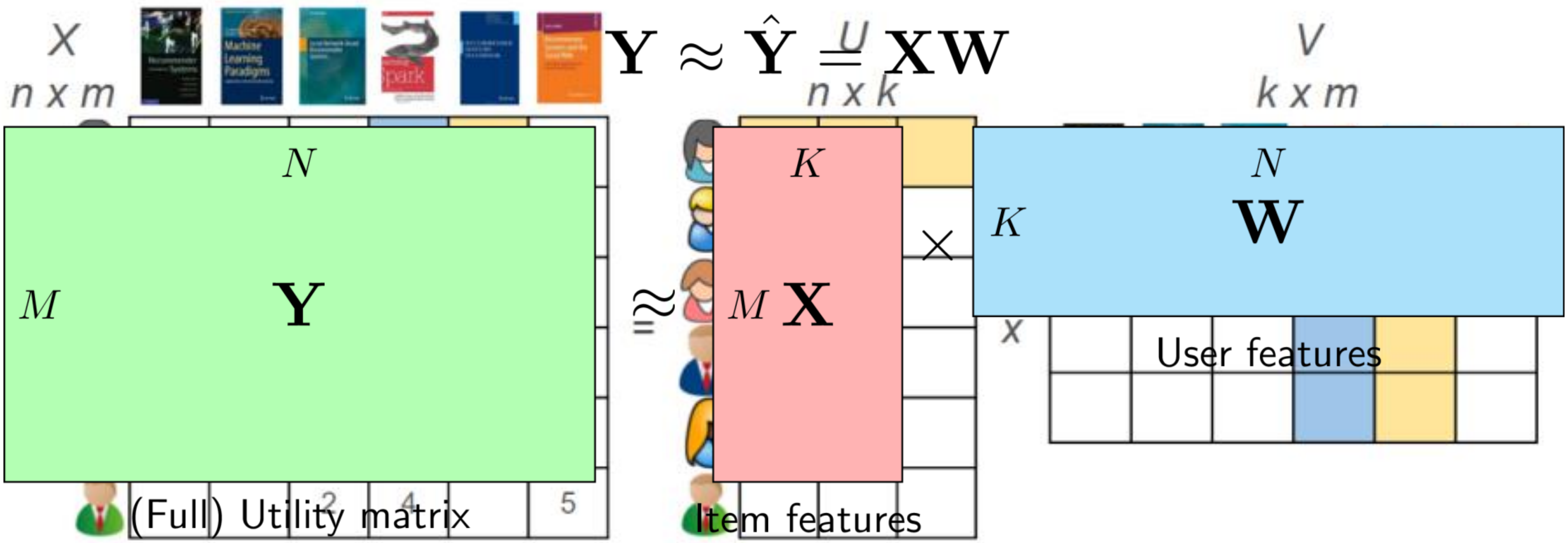
```python
from pyspark.ml.fpm import FPGrowth
fpGrowth = FPGrowth(itemsCol="items", minSupport=0.5, minConfidence=0.75)
model = fpGrowth.fit(df)
# get the frequent itemsets
patternsDF = model.freqItemsets
# get the association rules
rulesDF = model.associationRules
```

items	freq
[lemon]	3
[lemon, orange]	2
[lemon, orange, pasta]	2
[lemon, pasta]	3
[orange]	3
[orange, pasta]	3
[cake]	2
[cake, orange]	2
[cake, orange, pasta]	2
[cake, pasta]	2
[pasta]	4

antecedent	consequent	confidence	lift	
[cake]	[orange]	1.0	1.333333	
[cake]	[pasta]	1.0	1.0	
[cake, orange]	[pasta]	1.0	1.0	
[orange]	[pasta]	1.0	1.0	
[lemon]	[pasta]	1.0	1.0	
[cake, pasta]	[orange]	1.0	1.333333	[illegible]
[lemon, orange]	[pasta]	1.0	1.0	0.5
[pasta]	[lemon]	0.75	1.0	0.75
[pasta]	[orange]	0.75	1.0	0.75

Recommendation with Spark ALS

- Alternating Least Squares (ALS): perform the mapping from user and item features to corresponding item ratings.



Recommendation: MovieLens

- Data format in the text file sample_movielens_ratings.txt

```
userIDd (int)::movieId (int)::rating (float)::timestamp (long)
```

1501 lines (1501 sloc) | 31.6 KB

```
0::2::3::1424380312
0::3::1::1424380312
0::5::2::1424380312
0::9::4::1424380312
0::11::1::1424380312
0::12::2::1424380312
0::15::1::1424380312
```

- The dataset can be downloaded from here.

Recommendation with Spark ALS

```
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
# parse the input data to a DF
ratings = spark.read.text("/data/sample_movielens_ratings.txt").rdd.toDF()\
         .selectExpr("split(value , '::') as col")\
         .selectExpr("cast(col[0] as int) as userId", "cast(col[1] as int) as movieId",\
         "cast(col[2] as float) as rating","cast(col[3] as long) as timestamp")
als = ALS().setMaxIter(5).setRegParam(0.01).setUserCol("userId")\
                                          .setItemCol("movieId").setRatingCol("rating")

training, test = ratings.randomSplit([0.8, 0.2])
alsModel = als.fit(training)
predictions = alsModel.transform(test)

# generate top 10 movie recommendations for each user
userRecs = model.recommendForAllUsers(10)
# generate top 10 user recommendations for each movie
movieRecs = model.recommendForAllItems(10)
```

Recommendation with Spark ALS

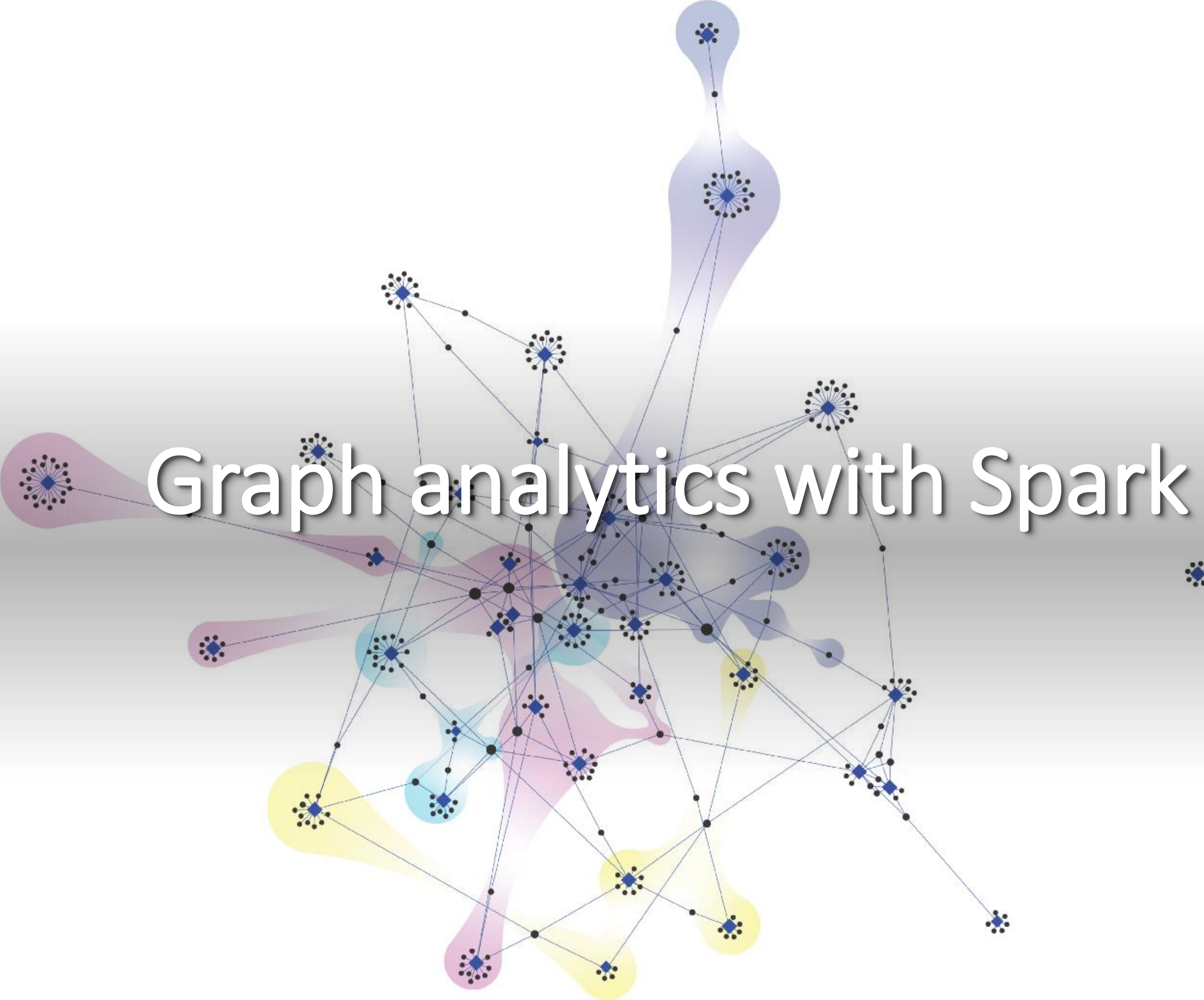
- Recommendation is a kind of regression problem.
  - Predict the rating for given users, minimizing the total difference between our users' ratings and the true values

`RegressionEvaluator` `RegressionMetrics` `RankingMetrics`

```
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator().setMetricName("rmse")\
                       .setLabelCol("rating").setPredictionCol("prediction")
rmse = evaluator.evaluate(predictions)
```

```
from pyspark.mllib.evaluation import RegressionMetrics
regComparison = predictions.select("rating", "prediction")\
                                      .rdd.map(lambda x: (x(0), x(1)))
metrics = RegressionMetrics(regComparison)
```

Graph analytics with Spark



From GraphX to GraphFrames

- `GraphX` provides an RDD-based interface only.
  - It is extremely powerful yet not easy to be used or optimized.
- **`GraphFrames`** extends `GraphX` to DataFrame APIs.
  - It is presently a Spark package, and it may be merged into Spark core in the future.
- There should be little difference in performance between the two, for the most part.
  - `GraphFrames` tries to call down to `GraphX` where appropriate
  - There is some small overhead, yet user experience gains greatly outweigh this.

GraphFrames: Build a graph

- Point to the proper package from the command line

```
./bin/spark-shell --packages graphframes:graphframes:
                                        0.5.0-spark2.2-s_2.11
```

- Read the bike data from the Bay Area Bike Share portal

```
bikeStations = spark.read.option("header","true")\
                    .csv("/data/bike-data/201508_station_data.csv")
tripData = spark.read.option("header","true")\
                    .csv("/data/bike-data/201508_trip_data.csv")
```

- Define the vertices and edges as two DFs

```
stationVertices = bikeStations.withColumnRenamed("name", "id").distinct()
tripEdges = tripData.withColumnRenamed("Start Station", "src")\
                    .withColumnRenamed("End Station", "dst")
```

GraphFrames: Build a graph

- Build a `GraphFrame` object from the vertex and edge
  - Data is frequently accessed in queries → leverage caching

```
from graphframes import GraphFrame
stationGraph = GraphFrame(stationVertices, tripEdges)
stationGraph.cache()
```

- We can see the basic statistics about graph

```
Total Number of Stations: 70
Total Number of Trips in Graph: 354152
Total Number of Trips in Original Data: 354152
```

```
print("Total Number of Stations: " + str(stationGraph.vertices.count()) )
print("Total Number of Trips in Graph: " + str(stationGraph.edges.count()) )
print("Total Number of Trips in Original Data: " + str(tripData.count()) )
```

GraphFrames: Query the graph

- `GraphFrame` offers access tc

```
+--------------------+--------------------+-----+
|                 src|                 dst|count|
+--------------------+--------------------+-----+
|San Francisco Cal...|     Townsend at 7th| 3748|
|Harry Bridges Pla...|Embarcadero at Sa...| 3145|
...
|     Townsend at 7th|San Francisco Cal...| 2192|
|Temporary Transba...|San Francisco Cal...| 2184|
+--------------------+--------------------+-----+
```

```
from pyspark.sql.functions import desc
stationGraph.edges.groupBy("src","dst").count().orderBy(desc("count"))
```

- It is also possible to filter by any valid DataFrame expression.

```
stationGraph.edges.where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")\
        .groupBy("src", "dst").count()\
        .orderBy(desc("count"))
```

```
+--------------------+--------------------+-----+
|                 src|                 dst|count|
+--------------------+--------------------+-----+
|San Francisco Cal...|     Townsend at 7th| 3748|
|     Townsend at 7th|San Francisco Cal...| 2734|
...
|   Steuart at Market|     Townsend at 7th|  746|
|     Townsend at 7th|Temporary Transba...|  740|
+--------------------+--------------------+-----+
```

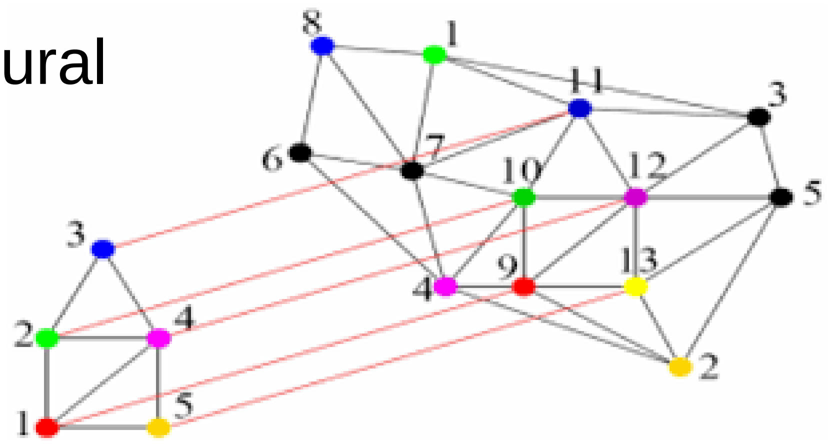
GraphFrames: Subgraphs

- Subgraphs are just smaller graphs within the larger one.

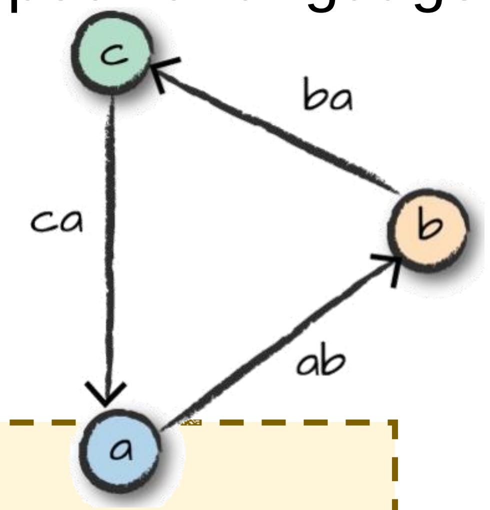
```
# create a subgraph that corresponds to the edges in townAnd7thEdges
townAnd7thEdges = stationGraph.edges\
                  .where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")
subgraph = GraphFrame(stationGraph.vertices, townAnd7thEdges)
```

GraphFrames: Finding motifs

- Motifs are expressions of structural patterns in a graph.



- `find`: specify a motif query in a domain-specific language like Neo4J's Cypher language.
  - E.g., `(a)-[ab]->(b)`: a vertex a connects to another vertex b through an edge ab.




```
# find motifs that resembles a triangle
motifs = stationGraph.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[ca]->(a)")
```

GraphFrames: Finding motifs

- Motif finding can combine with DF queries over the resulting tables to further narrow down, sort, or aggregate the

```
# find the trip with shortest duration
# in which three different people use the same bike
from pyspark.sql.functions import expr
motifs.selectExpr("*",
    "to_timestamp(ab.`Start Date`, 'MM/dd/yyyy HH:mm') as abStart",
    "to_timestamp(bc.`Start Date`, 'MM/dd/yyyy HH:mm') as bcStart",
    "to_timestamp(ca.`Start Date`, 'MM/dd/yyyy HH:mm') as caStart")\
        .where("ca.`Bike #` = bc.`Bike #`").where("ab.`Bike #` = bc.`Bike #`")\
        .where("a.id != b.id").where("b.id != c.id")\
        .where("abStart < bcStart").where("bcStart < caStart")\
        .orderBy(expr("cast(caStart as long) - cast(abStart as long)"))\
        .selectExpr("a.id", "b.id", "c.id", "ab.`Start Date`", "ca.`End Date`")
        .limit(1)
```

GraphFrames: PageRank

- PageRank is an algorithm to rank web pages

PageRank counts the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

- **pageRank**: we apply this to get a sense for important bike stations (i.e., those with high bike traffic).

```
+--------------------+------------------+
|                  id|          pagerank|
+--------------------+------------------+
|San Jose Diridon ...| 4.051504835989922|
|San Francisco Cal...|3.3511832964279518|
...
|     Townsend at 7th| 1.568456580534273|
|Embarcadero at Sa...|1.5414242087749768|
+--------------------+------------------+
```

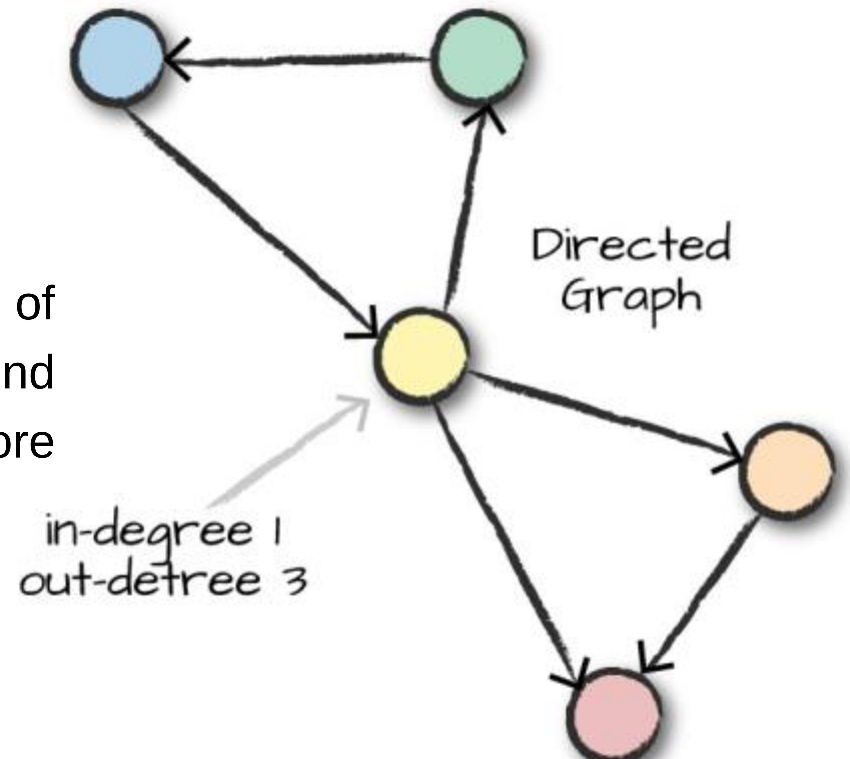
```
from pyspark.sql.functions import desc
ranks = stationGraph.pageRank(resetProbability=0.15, maxIter=10)
ranks.vertices.orderBy(desc("pagerank")).select("id", "pagerank")
```

In-degree and Out-degree metrics

- One common task is to count the number of inbound or outbound connections at a certain vertex.

Bike data: count the number of trips into or out of a given station

Social networks: count the number of followers and people they follow to find interesting people who might have more influence than others.



In-degree and Out-degree metrics

```
# sort by in-degree
stationGraph.inDegrees.orderBy(desc("inDegree"))
```

```
+----------------------------------------+--------+
|id                                      |inDegree|
+----------------------------------------+--------+
|San Francisco Caltrain (Townsend at 4th)|34810   |
|San Francisco Caltrain 2 (330 Townsend) |22523   |
|Harry Bridges Plaza (Ferry Building)    |17810   |
|2nd at Townsend                         |15463   |
|Townsend at 7th                         |15422   |
+----------------------------------------+--------+
```

```
+---------------------------------------------+---------+
|id                                           |outDegree|
+---------------------------------------------+---------+
|San Francisco Caltrain (Townsend at 4th)     |26304    |
|San Francisco Caltrain 2 (330 Townsend)      |21758    |
|Harry Bridges Plaza (Ferry Building)         |17255    |
|Temporary Transbay Terminal (Howard at Beale)|14436    |
|Embarcadero at Sansome                       |14158    |
+---------------------------------------------+---------+
```

```
# sort by in-degree
stationGraph.outDegrees.orderBy(desc("outDegree"))

# the ratio of two values is an interesting metric to look at.
degreeRatio = inDeg.join(outDeg, "id")\
            .selectExpr("id", "double(inDegree)/double(outDegree) as degreeRatio")
degreeRatio.orderBy(desc("degreeRatio"))
```

GraphFrames: Breadth-first search

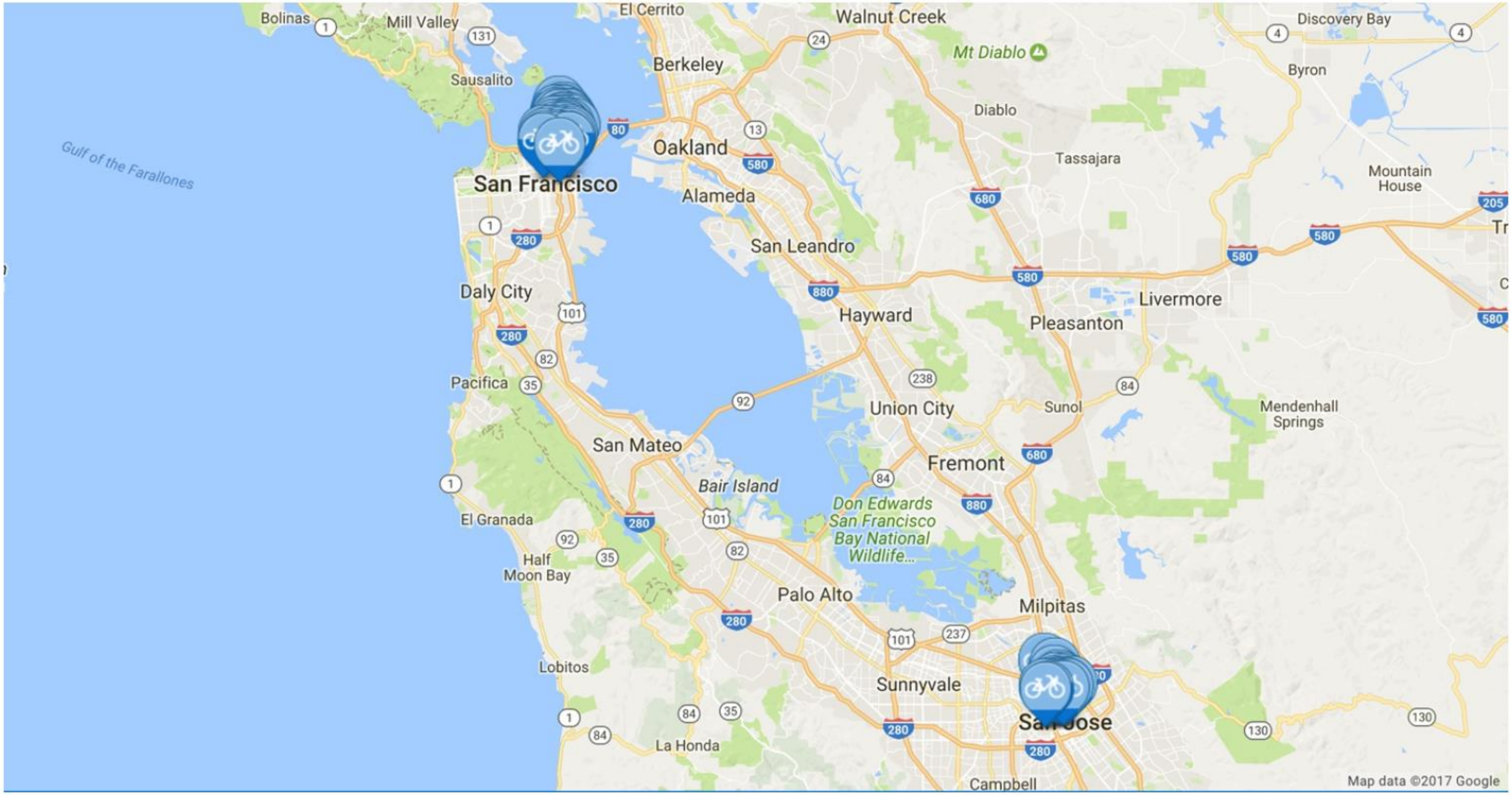
- **`bfs`**: searches the graph for how to connect two sets of nodes, based on the edges in the graph.
  - `maxPathLength`: maximum of edges to follow

```
+--------------------+--------------------+--------------------+
|                from|                  e0|                  to|
+--------------------+--------------------+--------------------+
|[65,Townsend at 7...|[913371,663,8/31/...|[49,Spear at Fols...|
|[65,Townsend at 7...|[913265,658,8/31/...|[49,Spear at Fols...|
...
|[65,Townsend at 7...|[903375,850,8/24/...|[49,Spear at Fols...|
|[65,Townsend at 7...|[899944,910,8/21/...|[49,Spear at Fols...|
+--------------------+--------------------+--------------------+
```

```
stationGraph.bfs(fromExpr="id = 'Townsend at 7th’”,\
                 toExpr="id = 'Spear at Folsom'", maxPathLength=2)
```

GraphFrames: Connect components

- A connected component is a (undirected) subgraph that has connections to itself but not to the greater graph



GraphFrames: Connect components

- **`connectedComponents`**: find the connected components and return a DF with new vertices column “component”.
  - Checkpoint and sampling are good huge data at local machine.

```
spark.sparkContext.setCheckpointDir("/tmp/checkpoints")
minGraph = GraphFrame(stationVertices, tripEdges.sample(False, 0.1))
minGraph.connectedComponents().where("component != 0")
```

- **`stronglyConnectedComponents`**: give a (directed) subgraph that has paths between all pairs of vertices inside it.

```
scc = minGraph.stronglyConnectedComponents(maxIter=3)
scc.groupBy("component").count().show()
```


...the end.