GraphFrames/MLlib in Spark

Lab 6
November 16th, 2017

Jun Heo(j.heo@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: Jonghyun Bae, Jun Heo, Jae W. Lee*

Index

GraphFrames

Previous Exercies's Answer

```
spark.sql("select a.City, f.origin, sum (f.delay) as Delays \
   from flightPerf f \
      join airports a on a.IATA = f.origin \
   group by a.City, f.origin \
   order by sum (f.delay) desc"
).show ()
```

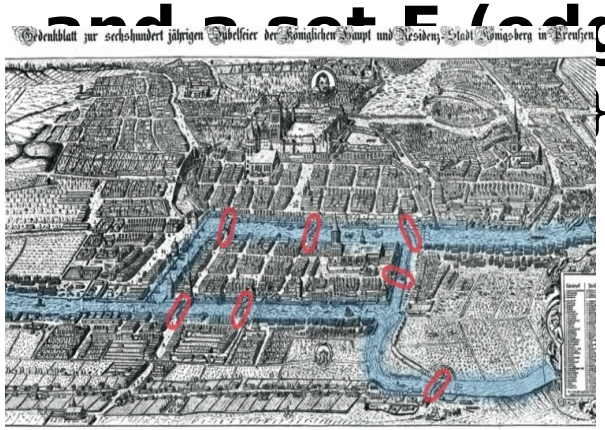
Before we start…

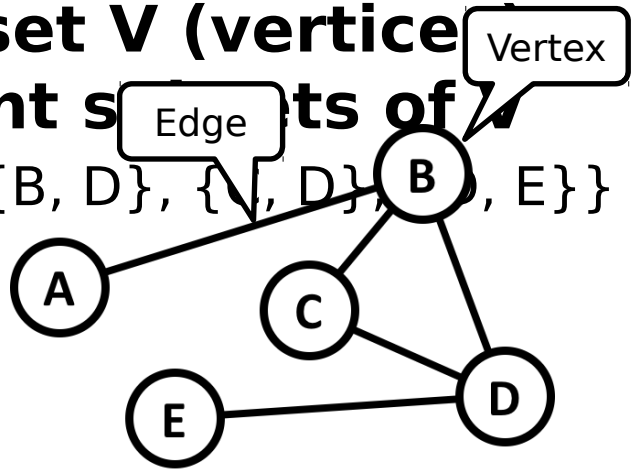
- **Please connect your VM using SSH**

```
# Please your public IP address in xxx.xxx.xxx.xxx
student@computer:~$ ssh -X -ibde3.pem ubuntu@xxx.xxx.xxx.xxx
Welcom e to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-125-generic x86_64)
[...snipp...]
ubuntu@ip-x-x-x:~$
```

Graphs in Real World

- **Abstract way of representing connectivity using vertices and edges**
- **Lots of problems formulated and solved in terms of graphs**
  - Eulerian path/circuit
  - Raking hyperlinks (PageRank)
  - Find shortest path in GPS, …
- **Graph is consists of a nonempty set V (vertice… and a set E (edges) of two element s…ts of V**

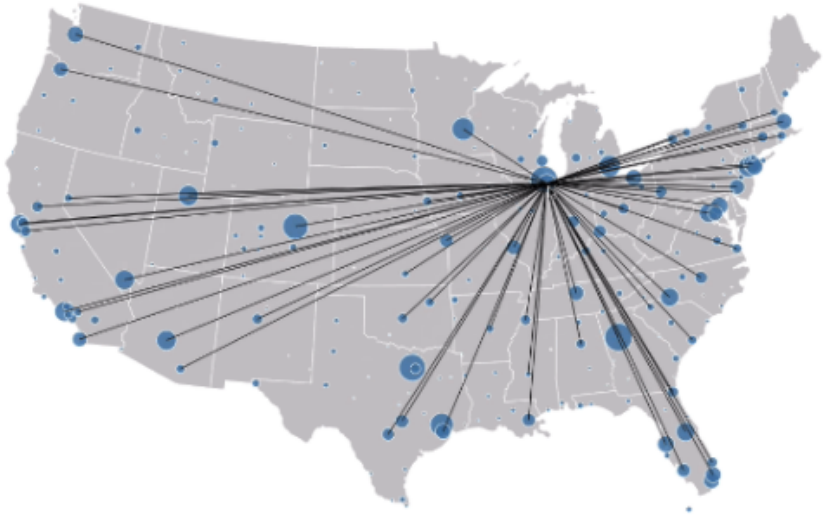




GraphFrames in Spark

- **Graph processing library for Apache Spark based on DataFrames**
  - Support general graph processing like Graph X library
    - Python, java, scala APIs
    - Powerful queries
    - Saving and loading graphs (based on DataFrame)

- **Example: analysis of flig**
  - Vertices: airports
  - Edges: flights between airport
  - Numerous properties associat with flights
    - Departure delays, plane ty



https://databricks.com/blog/2016/03/16/on-time-flight-performance-with-graphframes-for-

GraphFrames: Quick Start

- **Start PySpark with GraphFrames package and import it**

```
ubuntu@ip-x-x-x:~/spark-2.1.0$ bin/pyspark --packages
graphframes:graphframes:0.5.0-spark2.1-s_2.11
...
Using Python version 2.7.6 (default, Oct 26 2016 20:30:19)
SparkSession available as 'spark'.
>>> from graphframes import *
```

GraphFrames: Quick Start

- **Create vertex & edge DataFrames**

```
>>> v = sqlContext.createDataFrame([
... ("a", "Alice", 34), ("b", "Bob", 36), ("c", "Charlie", 30),
... ], ["id", "name", "age"])
>>> e = sqlContext.createDataFrame([
... ("a", "b", "friend"), ("b", "c", "follow"), ("c", "b", "follow"),
... ], ["src", "dst", "relationship"])
```

```
>>> v.show()
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
+---+-------+---+
```

```
>>> e.show()
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
+---+---+------------+
```

GraphFrames: Quick Start

- **Query GraphFrame and run the PageRank algorithm**

```
1 >>> g = GraphFrame(v, e)
2 >>> g.inDegrees.show()
3 >>> g.edges.filter("relationship = 'follow'").count()
  2
4 >>> results = g.pageRank(resetProbability=0.01, maxIter=5)
5 >>> results.vertices.select("id", "pagerank").show()
```

```
>>> g.inDegrees.show()
+---+--------+
| id|inDegree|
+---+--------+
|  c|       1|
|  b|       2|
+---+--------+
```

```
>>> results.vertices.select("id", "pagerank").show()
+---+------------------+
| id|          pagerank|
+---+------------------+
|  b|0.08763274109799998|
|  a|              0.01|
|  c|   0.077926810699|
+---+------------------+
```

GraphFrames

- **Vertex DataFrame**
  - Contain a special column named "id" which specifies unique IDs for each vertex in the graph

- **Edge DataFrame**
  - Contain two special columns: "src" (source vertex ID of edge) and
    "dst" (destination vertex ID of edge)

- **GraphFrame can be constructed from a DataFrame containing edges**
  - The vertices will be inferred from the sources and destinations of the edges

GraphFrames: Creating Graph-Frames

- Create a GraphFrame from vertex and edge DataFrames

```
>>> v = sqlContext.createDataFrame([
... ("a","Alice",34), ("b","Bob",36), ("c","Charlie",30), ("d","David",29),
... ("e","Esther", 32), ("f","Fanny", 36), ("g","Gabby", 60)
... ], ["id","name","age"])
>>> e = sqlContext.createDataFrame([
... ("a","b","friend"),("b","c","follow"),("c","b","follow"),("f","c","follow"),
... ("e","f","follow"), ("e","d","friend"), ("d","a","friend"),("a","e","friend")
>>> ], ["src","dst","relationship"])
>>> g = GraphFrame(v, e)
```

GraphFrames: Creating Graph-Frames

- Display the vertex and edge DataFrames

```
1 >>> g.vertices.show()
2 >>> g.edges.show()
```

```
>>> g.vertices.show()
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

```
>>> g.edges.show()
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
|  f|  c|      follow|
|  e|  f|      follow|
|  e|  d|      friend|
|  d|  a|      friend|
|  a|  e|      friend|
+---+---+------------+
```

GraphFrames: Simple Queries

- **Get the simple information in the graph**
  - Find the youngest user's age in the graph
  - Count the number of "follows" in the graph

```
>>> g.vertices.groupBy().min("age").show()
+--------+
|min(age)|
+--------+
|      29|
+--------+
>>> g.edges.filter("relationship = 'follow'").count()
>>> numFollows
4
```

GraphFrames: Motif Finding

- **Motif Finding**
  - Search for structural patterns in a graph
  - Ex) graph.find("(a)-[e]->(b); (b)-[e2]->(a)")
    - Search for pairs of vertices a, b connected by edges in both directions
  - Return a DataFrame of all such structures in the graph
  - Types of a structural pattern
    - (a)-[e]->(b)
    - (a)-[e]->(b); (b)-[e2]->(c)
  - It is acceptable to omit names for vertices and edges in motifs
    - (a)-[e]->()

GraphFrames: Motif Finding

- Search for pairs of vertices with edges in both directions
- More complex queries can be expressed by applying filters

```
1 >>> motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")
2 >>> motifs.show()
3 >>> motifs.filter("b.age > 30").show()
```

```
>>> motifs.show()
+--------------+-----------+-------------+-----------+
|             a|          e|            b|         e2|
+--------------+-----------+-------------+-----------+
|[c,Charlie,30]|[c,b,follow]|    [b,Bob,36]|[b,c,follow]|
|    [b,Bob,36]|[b,c,follow]|[c,Charlie,30]|[c,b,follow]|
+--------------+-----------+-------------+-----------+
```

```
>>> motifs.filter("b.age > 30").show()
+--------------+-----------+---------+-----------+
|             a|          e|        b|         e2|
+--------------+-----------+---------+-----------+
|[c,Charlie,30]|[c,b,follow]|[b,Bob,36]|[b,c,follow]|
+--------------+-----------+---------+-----------+
```

GraphFrames: Motif Finding

- **More complex motif queries**
- **Suppose one wishes to identify a chain of 4 vertices with some property defined by a sequence of functions**
  - a->b->c->d
  - Initialize state on path
  - Update state based on vertex a
  - Update state based on vertex b
  - Etc. for c and d
  - If final state matches some condition, then the chain is accepted by the filter
- **We identify chains of 4 vertices such that at least 2 of the 3 edges are "friend" relationships**

GraphFrames: Motif Finding

- **Code**

```
>>> from pyspark.sql.functions import col, lit, udf, when
>>> from pyspark.sql.types import IntegerType
>>> chain4 = g.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[cd]->(d)")
>>> sumFriends =\
... lambda cnt,relationship:\
... when(relationship == "friend", cnt+1).otherwise(cnt)
>>> condition =\
... reduce(lambda cnt,e: sumFriends(cnt, col(e).relationship),\
... ["ab", "bc", "cd"], lit(0))
>>> chainWith2Friends2 = chain4.where(condition >= 2)
>>> chainWith2Friends2.show()
```

GraphFrames: Motif Finding

- **Output**

```
>>> chainWith2Friends2.show()
+-------------+------------+-------------+------------+-------------+------------+--------------+
|            a|          ab|            b|          bc|            c|          cd|             d|
+-------------+------------+-------------+------------+-------------+------------+--------------+
| [d,David,29]|[d,a,friend]| [a,Alice,34]|[a,e,friend]|[e,Esther,32]|[e,f,follow]|  [f,Fanny,36]|
|[e,Esther,32]|[e,d,friend]| [d,David,29]|[d,a,friend]| [a,Alice,34]|[a,e,friend]| [e,Esther,32]|
| [d,David,29]|[d,a,friend]| [a,Alice,34]|[a,e,friend]|[e,Esther,32]|[e,d,friend]|  [d,David,29]|
| [d,David,29]|[d,a,friend]| [a,Alice,34]|[a,b,friend]|   [b,Bob,36]|[b,c,follow]|[c,Charlie,30]|
|[e,Esther,32]|[e,d,friend]| [d,David,29]|[d,a,friend]| [a,Alice,34]|[a,b,friend]|    [b,Bob,36]|
| [a,Alice,34]|[a,e,friend]|[e,Esther,32]|[e,d,friend]| [d,David,29]|[d,a,friend]|  [a,Alice,34]|
+-------------+------------+-------------+------------+-------------+------------+--------------+
```

GraphFrames: Subgraphs

- GraphFrames provide a powerful way to select subgraphs based on a combination of motif finding and DataFrame filters
- Simple subgraph: vertex and edge filters

```
1 >>> v2 = g.vertices.filter("age > 30")
2 >>> e2 = g.edges.filter("relationship = 'friend'")
3 >>> g2 = GraphFrame(v2, e2)
```

```
>>> g2.vertices.show()
+---+------+---+
| id|  name|age|
+---+------+---+
|  a| Alice| 34|
|  b|   Bob| 36|
|  e|Esther| 32|
|  f| Fanny| 36|
|  g| Gabby| 60|
+---+------+---+
```

```
>>> g2.edges.show()
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  e|  d|      friend|
|  d|  a|      friend|
|  a|  e|      friend|
+---+---+------------+
```

GraphFrames: Subgraphs

- **Complex subgraph: triplet filters**

```
>>> paths = g.find("(a)-[e]->(b)").filter("e.relationship = 'follow'")\
... .filter("a.age < b.age")
>>> e2 = paths.select("e.src", "e.dst", "e.relationship")
>>> g2 = GraphFrame(g.vertices, e2)
```

```
>>> g2.vertices.show()
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

```
>>> g2.edges.show()
[Stage 198:==========
[Stage 197:=========>
[Stage 198:==========

+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  e|  f|      follow|
|  c|  b|      follow|
+---+---+------------+
```

GraphFrames: Graph Algorithms

- **Breadth-first search (BFS)**
  - Breadth-first search (BFS) finds the shortest path(s) from one vertex to another vertex

```
>>> paths = g.bfs("name = 'Esther'", "age < 32")
>>> paths.show()
>>> relation = g.bfs("name = 'Esther'", "age < 32",\
... edgeFilter="relationship != 'friend'", maxPathLength=3)
>>> relation.show()
```

```
>>> paths.show()
+-------------+-----------+-----------+
|         from|         e0|         to|
+-------------+-----------+-----------+
|[e,Esther,32]|[e,d,friend]|[d,David,29]|
+-------------+-----------+-----------+
```

```
>>> relation.show()
+-------------+-----------+-----------+-----------+-------------+
|         from|         e0|         v1|         e1|           to|
+-------------+-----------+-----------+-----------+-------------+
|[e,Esther,32]|[e,f,follow]|[f,Fanny,36]|[f,c,follow]|[c,Charlie,30]|
+-------------+-----------+-----------+-----------+-------------+
```

GraphFrames: Graph Algorithms

- **PageRank**
  - Uses the standalone GraphFrame interface and runs PageRank for a fixed number of iterations. This can be run by setting maxIter.

```
>>> results = g.pageRank(resetProbability=0.15, tol=0.01)
>>> results.vertices.select("id", "pagerank").show()
>>> results.edges.select("src", "dst", "weight").show()
```

```
>>> results.vertices.select("id", "pagerank").show()
+---+------------------+
| id|          pagerank|
+---+------------------+
|  g|              0.15|
|  c|  2.240080617201845|
|  f|0.27366105468749996|
|  b| 2.2131428039184433|
|  a|0.37429242187499995|
|  e|  0.3090742792968 75|
|  d|0.27366105468749996|
+---+------------------+
```

```
>>> results.edges.select("src", "dst", "weight").show()
+---+---+------+
|src|dst|weight|
+---+---+------+
|  a|  b|   0.5|
|  b|  c|   1.0|
|  e|  f|   0.5|
|  e|  d|   0.5|
|  c|  b|   1.0|
|  a|  e|   0.5|
|  f|  c|   1.0|
|  d|  a|   1.0|
+---+---+------+
```

GraphFrames: Graph Algorithms

- **PageRank**

```
>>> results2 = g.pageRank(resetProbability=0.15, maxIter=5)
>>> results3 = g.pageRank(resetProbability=0.15, maxIter=5, sourceId="a")
>>> results2.vertices.show()
>>> results3.vertices.show()
>>> results2.edges.show()
>>> results3.edges.show()
```

GraphFrames: Graph Algorithms

- Output

result 2

```
>>> results2.vertices.show()
+---+-------+---+------------------+
| id|   name|age|          pagerank|
+---+-------+---+------------------+
|  g|  Gabby| 60|              0.15|
|  b|    Bob| 36| 1.2192788496093747|
|  e| Esther| 32|0.30907427929687503|
|  a|  Alice| 34|  0.382611896484375|
|  f|  Fanny| 36|0.27782079199218745|
|  d|  David| 29|0.27782079199218745|
|  c|Charlie| 30|    1.270496296875|
+---+-------+---+------------------+
```

```
>>> results2.edges.show()
+---+---+------------+------+
|src|dst|relationship|weight|
+---+---+------------+------+
|  a|  b|      friend|   0.5|
|  b|  c|      follow|   1.0|
|  e|  f|      follow|   0.5|
|  e|  d|      friend|   0.5|
|  c|  b|      follow|   1.0|
|  a|  e|      friend|   0.5|
|  f|  c|      follow|   1.0|
|  d|  a|      friend|   1.0|
+---+---+------------+------+
```

result 3

```
>>> results3.vertices.show()
+---+-------+---+------------------+
| id|   name|age|          pagerank|
+---+-------+---+------------------+
|  g|  Gabby| 60|               0.0|
|  b|    Bob| 36|       0.172450125|
|  e| Esther| 32|   0.0735376171875|
|  a|  Alice| 34|      0.1730296875|
|  f|  Fanny| 36|0.0312534873046875|
|  d|  David| 29|0.0312534873046875|
|  c|Charlie| 30| 0.141326080078125|
+---+-------+---+------------------+
```

```
>>> results3.edges.show()
+---+---+------------+------+
|src|dst|relationship|weight|
+---+---+------------+------+
|  a|  b|      friend|   0.5|
|  b|  c|      follow|   1.0|
|  e|  f|      follow|   0.5|
|  e|  d|      friend|   0.5|
|  c|  b|      follow|   1.0|
|  a|  e|      friend|   0.5|
|  f|  c|      follow|   1.0|
|  d|  a|      friend|   1.0|
+---+---+------------+------+
```

GraphFrames: Graph Algorithms

- **Shortest Paths**
  - Computes shortest paths from each vertex to the given set of landmark vertices, where landmarks are specified by vertex ID
  - Note that this takes edge direction into account

```
1 >>> results = g.shortestPaths(landmarks=["a", "d"])
2 >>> results.select("id", "distances").show()
```

```
>>> results.select("id", "distances").show()
+---+------------------+
| id|         distances|
+---+------------------+
|  g|              Map()|
|  c|              Map()|
|  f|              Map()|
|  b|              Map()|
|  a|Map(a -> 0, d -> 2)|
|  e|Map(d -> 1, a -> 2)|
|  d|Map(d -> 0, a -> 1)|
+---+------------------+
```

Saving and Loading Graph-Frames

- Since GraphFrames are built around DataFrames, they automatically support saving and loading to and from the same set of data sources

```
>>> g.vertices.write.parquet("hdfs://172.31.1.25:9000/graph/vertices")
>>> g.edges.write.parquet("hdfs://172.31.1.25:9000/graph/edges")
>>> sameV = sqlContext.read.parquet("hdfs://172.31.1.25:9000/graph/vertices")
>>> sameE = sqlContext.read.parquet("hdfs://172.31.1.25:9000/graph/edges")
>>> sameG = GraphFrame(sameV, sameE)
>>> sameG.vertices.show()
>>> sameG.edges.show()
```

MLlib

What's the MLlib

- **MLlib is Spark's machine learning (ML) library**
- **Mllib contains only parallel algorithms**
- **Text classification task**
  - Transform your messages to an **RDD** of strings
  - Run one of **feature extraction** algorithms
    - convert text into numerical features
    - return an RDD of vectors
  - Call a **classification algorithm**
    - return a model object that can be used to classify new points
  - Evaluate the model on a test dataset using one of evaluation functions

Data Types: Vector

- **Vector**
  - Integer-typed and 0-based indices and double-typed values
  - Dense vector: double array representing its entry values
    - NumPy's array, Python's list
  - Sparse vector: size of vector, indices and values
    - Mllib's SparseVector

```
>>> dv1 = np.array([1.0, 0.0, 3.0])
>>> dv2 = [1.0, 0.0, 3.0]
>>> sv1 = Vectors.sparse(3, [0, 2], [1.0, 3.0])
>>> dv1
array([ 1.,  0.,  3.])
>>> dv2
[1.0, 0.0, 3.0]
>>> sv1
SparseVector(3, {0: 1.0, 2: 3.0})
```

Data Types: Labeledpoint

- **Labeledpoint**
  - A vector associated with a label/response
  - Used in supervised learning
  - Ex) binary classification
    - Label: 0 (negative) or 1 (positive)

```
>>> from pyspark.mllib.linalg import SparseVector
>>> from pyspark.mllib.regression import LabeledPoint
>>> pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])
>>> neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
>>> pos
LabeledPoint(1.0, [1.0,0.0,3.0])
>>> neg
LabeledPoint(0.0, (3,[0,2],[1.0,3.0]))
```

Data Types: Matrix

- **Matrix**
  - Integer-typed row and column indices and double-typed values
  - DenseMatrix
    - Entry values are stored in a single double array in column-major order
  - SparseMatrix
    - Non-zero entry values are stored in the Compressed Sparse Column format

```
>>> from pyspark.mllib.linalg import Matrix, Matrices
>>> dm2 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])
>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
>>> dm2
DenseMatrix(3, 2, [1.0, 2.0, 3.0, 4.0, 5.0, 6.0], False)
>>> sm
SparseMatrix(3, 2, [0, 1, 3], [0, 2, 1], [9.0, 6.0, 8.0], False)
```

Algorithms

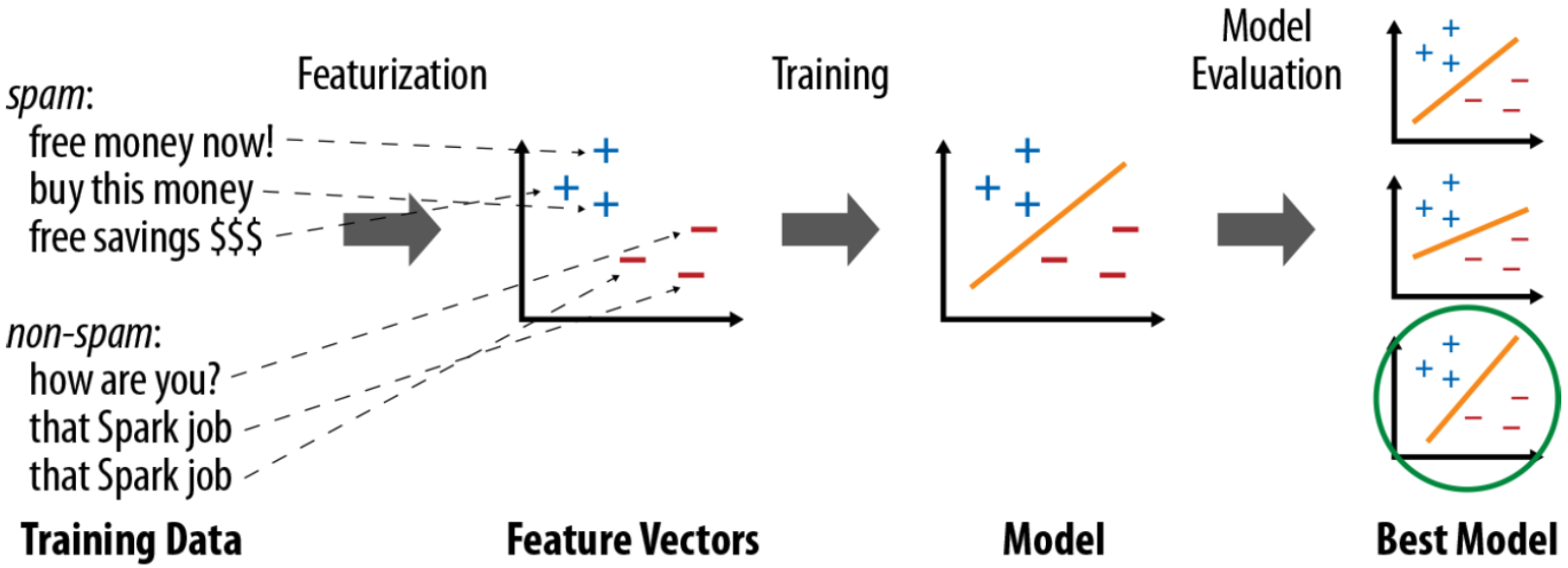
- Feature Extraction
  - Construct feature vectors from text, and ways to normalize and scale features
  - TF-IDF
  - TF-IDF
    - Generate feature vectors from text documents (e.g., web pages)
    - Term frequency (TF),
      - the number of times the term occurs in that document
    - Inverse document frequency (IDF)
      - how (in)frequently a term occurs across the whole document corpus
    - The product of these values (TF × IDF)
      - how relevant a term is to a specific document
    - HashingTF

Algorithms

- **Classification and Regression**
  - Common forms of supervised learning
  - Predict a variable from features of objects using labeled training data
  - Logistic Regression
    - Binary classification method
    - Identifies a linear separating plane between positive and negative examples
    - Use labeledpoints
    - Computes a score between 0 and 1 for each point
    - Returns either 0 or 1 based on a threshold (setThreshold())

Example: Spam Classification

- **Two MLlib algorithms**
  - HashingTF
    - Build term frequency feature vectors from text
  - LogisticRegressionWithSGD
    - Implement the logistic regression using stochastic gradient descent
- **S**



Example: Spam Classification

- **Code**

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> from pyspark.mllib.feature import HashingTF
>>> from pyspark.mllib.classification import LogisticRegressionWithSGD
>>> Spam = sc.textFile("spam.txt")
>>> Normal = sc.textFile("ham.txt")
>>> tf = HashingTF(numFeatures = 10000)
>>> spamFeatures = Spam.map(lambda email: tf.transform(email.split(" ")))
>>> normalFeatures = Normal.map(lambda email: tf.transform(email.split(" ")))
```

Example: Spam Classification

```
>>> positiveExamples = spamFeatures.map(lambda features: LabeledPoint(1, features))
>>> negativeExamples = normalFeatures.map(lambda features: LabeledPoint(0, features))
>>> trainingData = positiveExamples.union(negativeExamples)
>>> trainingData.cache()

>>> model = LogisticRegressionWithSGD.train(trainingData)

>>> posTest = tf.transform("O M G GET cheap stuff by sending money to ...".split(" "))
>>> negTest = tf.transform("Hi Dad, I started studying Spark the other ...".split(" "))
>>> print "Prediction for positive test example: %g" % model.predict(posTest)
>>> print "Prediction for negative test example: %g" % model.predict(negTest)
```

```
Prediction for negative test example: 0
Prediction for positive test example: 1
```

Exercise 1 (10pts)

- **GraphFrames**
  - departuredelays.csv & airports-codes-na.csv
  - https://drive.google.com/open?id=0B91DOcPTZ5DzWWplT1N3OEdDWVE
  - Answer the questions
  - Questions
    - The longest delay in this dataset? (2)
    - The number of delayed versus on-time/early flights (2)
    - What flights departing Seattle are most likely to have signifi-cant delays? (2)
      - Seattle == 'SEA'
    - Top 5 busiest airports (most flights in and out) (2)
      - Use vertex degrees
    - Airport ranking using PageRank (2)
      - Reset probability=0.15, max iteration = 5

Exercise: Code Snippet (1)

```
ubuntu@ip-x-x-x:~/spark-2.1.0$ bin/pyspark --packages graphframes:graphframes:0.5.0-spark2.1-s_2.11
>>> from pyspark.sql.functions import *
>>> from graphframes import *
>>> tripdelaysFilePath = "departuredelays.csv"
>>> airportsnaFilePath = "airports-codes-na.csv"
>>> airports = spark.read.csv(airportsnaFilePath, header='true')
>>> airports.createOrReplaceTempView("airports_na")
>>> departureDelays = spark.read.csv(tripdelaysFilePath, header='true')
>>> departureDelays.createOrReplaceTempView("departureDelays")
>>> tripIATA = spark.sql("select distinct iata from (select distinct src as iata from\
>>> departureDelays union all select distinct dst as iata from departureDelays) a")
>>> tripIATA.createOrReplaceTempView("tripIATA")
```

Exercise: Code Snippet (2)

```
1 >>> tripVertices = airports.withColumnRenamed("ID", "No").distinct()
2 >>> tripVertices = tripVertices.withColumnRenamed("IATA", "id").distinct()
3 >>> tripEdges = departureDelays.select("tripid", "delay", "src", "dst", "city_dst", "state_dst")
4 >>> tripEdges = departureDelays.select("tripid", departureDelays.delay.cast("int"), "src", "dst", "city_dst", "state_dst")
5 >>> tripEdges = tripEdges.withColumn("label",tripEdges["delay"].cast("int"))
6 >>> tripGraph=GraphFrame(tripVertices, tripEdges)
```

Appendix