Welcome to GKTCS Innovations Pvt Ltd

IT Training & Consultancy

Surendra Panpaliya

Director, GKTCS Innovations Pvt. Ltd, Pune.

18+ Years of Experience (MCA, PGDCS, BSc. [Electronics] , CCNA)

- Founder,GKTCS Innovations Pvt. Ltd. Pune [Nov 2009 – Till date]
- 500 + Corporate Training for HP, IBM, Cisco,Wipro, Samsung etc.
- Skills
  - **Hadoop, Pig, Hive, Sqoop, Oozie, Spark, PySpark**
  - **Ruby, Rails,Cucumber, Calabash, Capybara, Rspec, Appium**
  - **Python, Django, Data Science, Machine Learning, Jython, Selenium**
  - **UNIX /Linux Shell Scripting, Perl, PHP, CakePHP, System Programming**
  - **CA Siteminder, Autosys, SSO, Service Desk, Service Delivery**
- Author of 4 Books
- National Paper Presentation Awards at BARC Mumbai

Agenda

Day 1
Module 1
Introduction to Spark
What is Apache Spark?
Spark Jobs and APIs
Spark 2.0 architecture
Installation and Configuration

Module 2
Resilient Distributed Datasets
Internal workings of an RDD
Creating RDDs
Global versus local scope
Transformations
Actions
Hands on Session on RDD and Spark
Assignments 1
Best Practices 1

Agenda

Day 2

Module 3

DataFrames
Python to RDD communications
Catalyst Optimiser refresh
Speeding up PySpark with DataFrames
Creating DataFrames
Simple DataFrame queries
Interoperating with RDDs
Querying with the DataFrame API
Hands On Session on Pandas DataFrame and PySpark
Assignments 2

Module 4

Prepare Data for Modelling
Checking for duplicates, missing observations, and outliers
Getting familiar with your data Visualisation
Hands on Session Data Modelling
Assignments 3

Agenda

Day 3
Module 5
Introducing MLlib
Overview of the package
Loading and transforming the data
Getting to know your data
Creating the final dataset
Predicting infant survival
Hands on Session using PySpark MLib
Assignments 4

Module 6
Introducing the ML Package
Overview of the package
Predicting the chances of infant survival with ML
Parameter hyper-tuning
Other features of PySpark ML in action
Implementation of ML Algorithm
- Random Forest
- Regression
- K-means

Assignments 5

Agenda

Day 3
Module 7
GraphFrames
Introducing GraphFrames
Installing GraphFrames
Preparing your flights dataset
Building the graph
Executing simple queries
Understanding vertex degrees
Determining the top transfer airports
Understanding motifs
Determining airport ranking using PageRank
Determining the most popular non-stop flights
Using Breadth-First Search
Visualizing flights using D3
Assignment 6
Conclusion and Summary

SPARK Pyspark

Surendra R. Panpaliya
M:9975072320
surendra@gktcs.com
www.gktcs.com

Module 5

Introducing MLlib

Introducing MLlib

MLlib stands for Machine Learning Library.

Even though MLlib is now in a maintenance mode, that is, it is not actively being developed (and will most likely be deprecated later), it is warranted that we cover at least some of the features of the library.

In addition, MLlib is currently the only library that supports training models for streaming.

Note:

Starting with Spark 2.0, ML is the main machine learning library that operates on DataFrames instead of RDDs as is the case for MLlib.

The documentation for MLlib can be found here:

http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html.

Introducing MLlib

Objectives

You will learn how to do the following:

- Prepare the data for modeling with MLlib
- Perform statistical testing
- Predict survival chances of infants using logistic regression
- Select the most predictable features and train a random forest model

Overview of the package

At the high level, MLlib exposes three core machine learning functionalities:

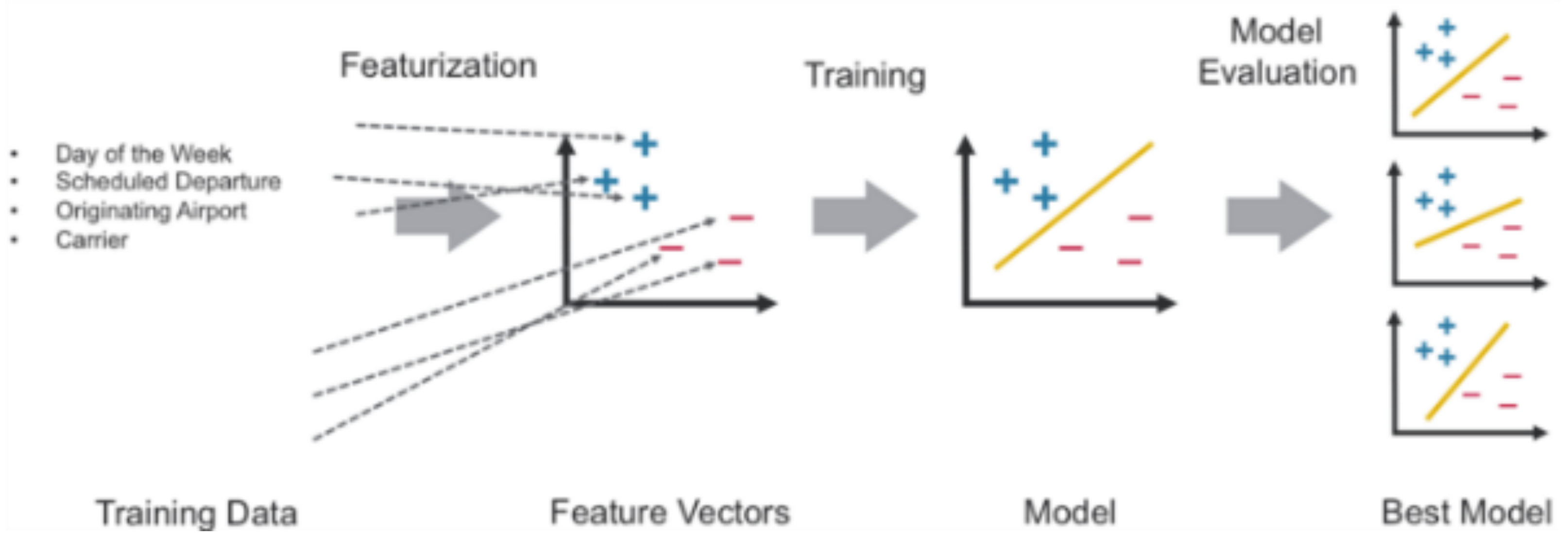
- **Data preparation**: Feature extraction, transformation, selection, hashing of categorical features, and some natural language processing methods
- **Machine learning algorithms:** Some popular and advanced regression, classification, and clustering algorithms are implemented
- **Utilities:** Statistical methods such as descriptive statistics, chi-square testing, linear algebra (sparse and dense matrices and vectors), and model evaluation methods.

Overview of the package

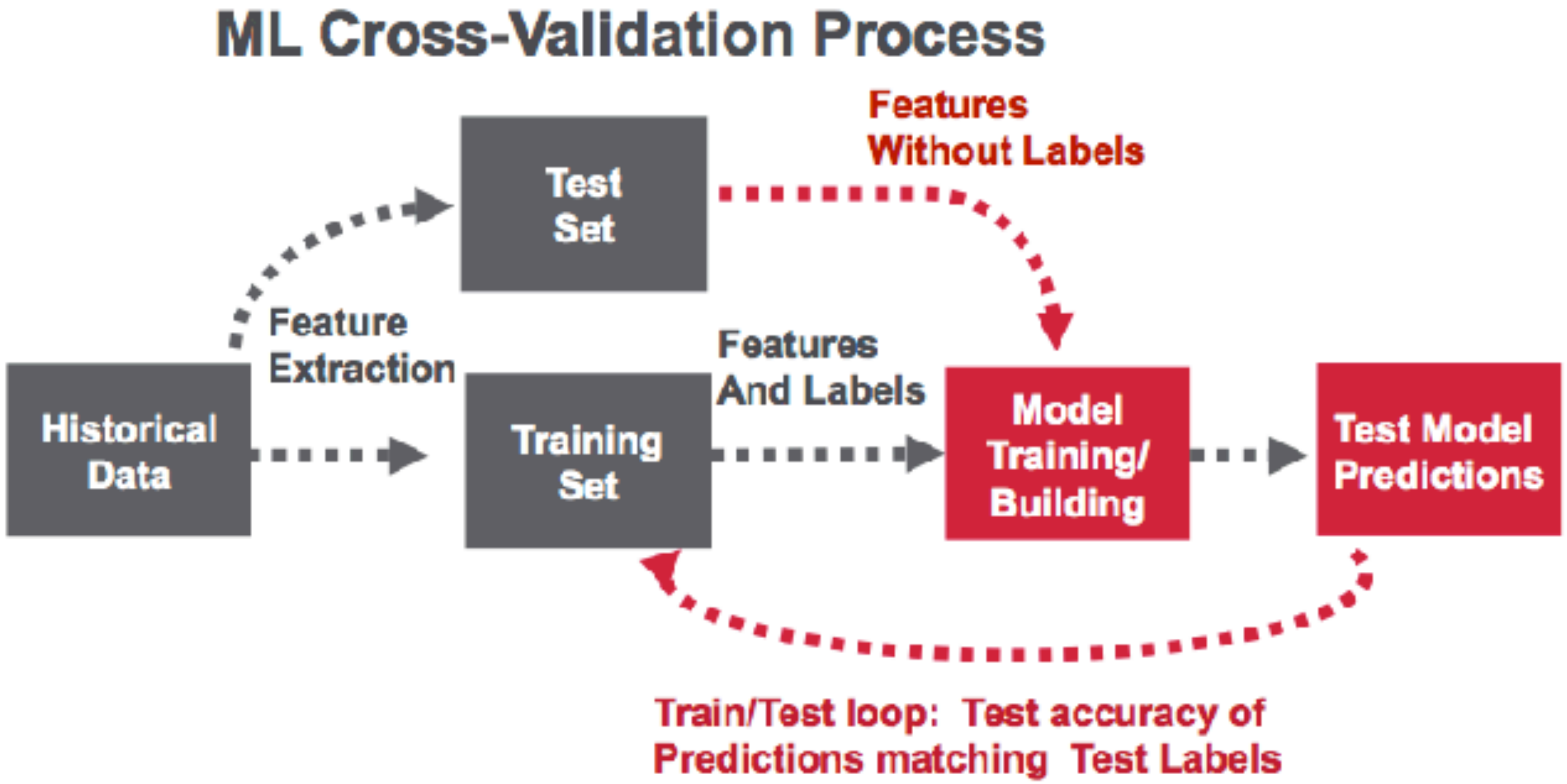
The main concepts in Spark ML are:

- DataFrame: The ML API uses DataFrames from Spark SQL as an ML dataset.
- Transformer: A Transformer is an algorithm which transforms one DataFrame into another DataFrame. For example, turning a DataFrame with features into a DataFrame with predictions.
- Estimator: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. For example, training/tuning on a DataFrame and producing a model.
- Pipeline: A Pipeline chains multiple Transformers and Estimators together to specify a ML workflow.
- ParamMaps: Parameters to choose from, sometimes called a “parameter grid” to search over.
- Evaluator: Metric to measure how well a fitted Model does on held-out test data.
- CrossValidator: Identifies the best ParamMap and re-fits the Estimator using the best ParamMap and the entire dataset.

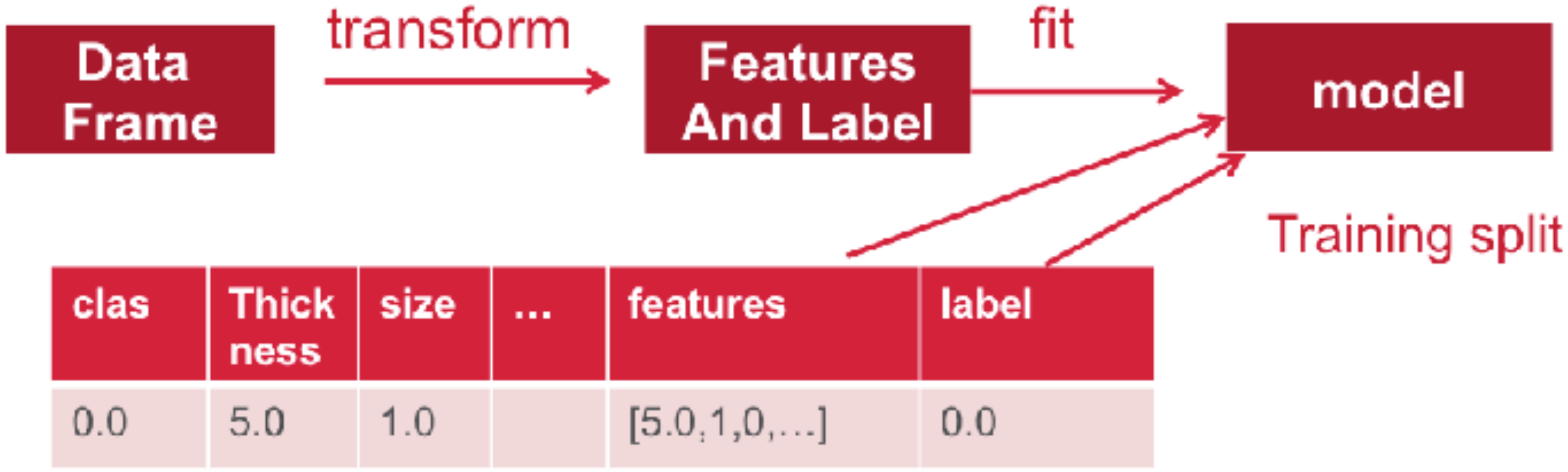
Overview of the package



Overview of the package

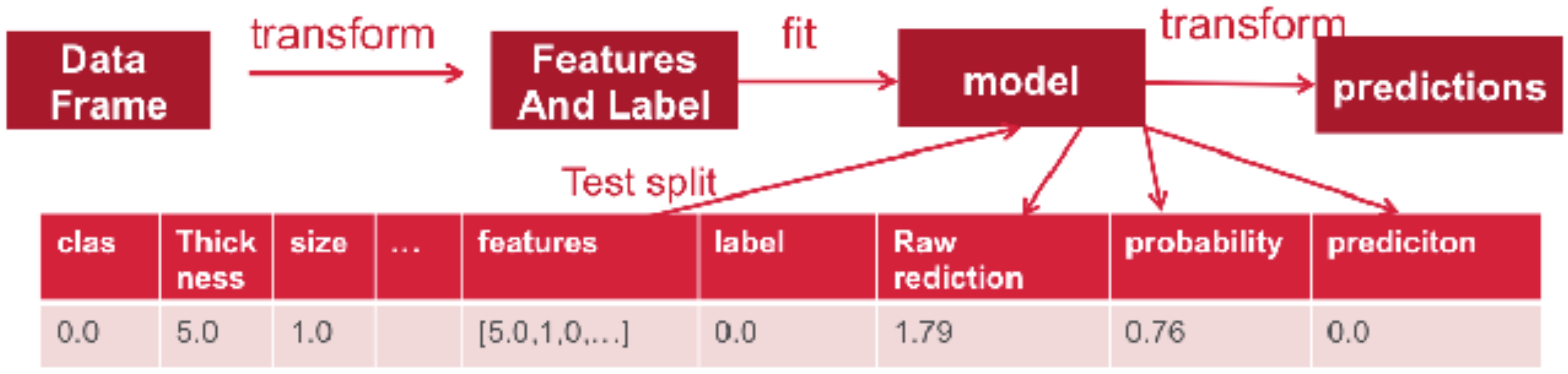


Overview of the package



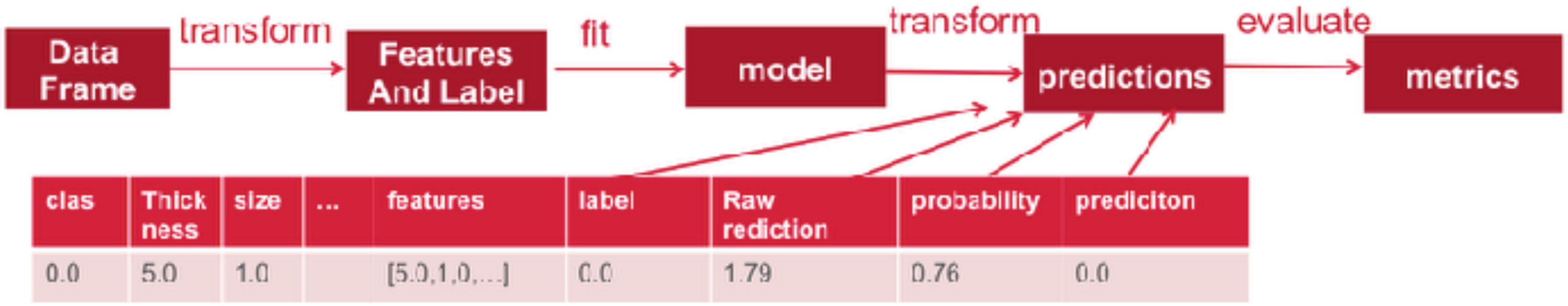
clas	Thick ness	size	...	features	label
0.0	5.0	1.0		[5.0,1,0,...]	0.0

Overview of the package



clas	Thick ness	size	…	features	label	Raw rediction	probability	prediciton
0.0	5.0	1.0		[5.0,1,0,…]	0.0	1.79	0.76	0.0

Overview of the package



clas	Thick ness	size	...	features	label	Raw rediction	probability	prediciton
0.0	5.0	1.0		[5.0,1,0,...]	0.0	1.79	0.76	0.0

Overview of the package

PySpark Mllib
http://spark.apache.org/docs/2.0.0/api/python/pyspark.mllib.html

PySpark ml

http://spark.apache.org/docs/2.0.0/api/python/pyspark.ml.html

PySpark streaming

http://spark.apache.org/docs/2.0.0/api/python/pyspark.streaming.html

PySpark sql

http://spark.apache.org/docs/2.0.0/api/python/pyspark.sql.html

Overview of the package

http://spark.apache.org/docs/2.0.0/api/python/pyspark

Overview of the package

pyspark.ml package

- ML Pipeline APIs
  - Transformer
  - Estimator
  - Model
  - Pipeline
  - PipelineModel
- pyspark.ml.param module
  - Param
  - Params
  - TypeConverters

Overview of the package

pyspark.ml.feature module

- Binarizer
- Bucketizer
- ChiSqSelector E
- ChiSqSelectorModel E
- CountVectorizer
- CountVectorizerModel
- DCT
- ElementwiseProduct
- HashingTF
- IDF
- IDFModel
- IndexToString
- MaxAbsScaler E
- MaxAbsScalerModel E
- MinMaxScaler
- MinMaxScalerModel

Overview of the package

pyspark.ml.feature module

- NGram
- Normalizer
- OneHotEncoder
- PCA
- PCAModel
- PolynomialExpansion
- QuantileDiscretizer E
- RegexTokenizer
- RFormula E
- RFormulaModel E
- SQLTransformer
- StandardScaler
- StandardScalerModel
- StopWordsRemover
- StringIndexer
- StringIndexerModel
- Tokenizer
- VectorAssembler
- VectorIndexer
- VectorIndexerModel
- VectorSlicer
- Word2Vec
- Word2VecModel

Overview of the package

pyspark.ml.classification module

- LogisticRegression
- LogisticRegressionModel
- LogisticRegressionSummaryE
- LogisticRegressionTrainingSummaryE
- BinaryLogisticRegressionSummaryE
- BinaryLogisticRegressionTrainingSummaryE
- DecisionTreeClassifier
- DecisionTreeClassificationModel
- GBTClassifier
- GBTClassificationModel
- RandomForestClassifier
- RandomForestClassificationModel
- NaiveBayes
- NaiveBayesModel
- MultilayerPerceptronClassifierE
- MultilayerPerceptronClassificationModelE
- OneVsRestE
- OneVsRestModelE

Overview of the package

pyspark.ml.clustering module

- BisectingKMeansE
- BisectingKMeansModelE
- KMeans
- KMeansModel
- GaussianMixtureE
- GaussianMixtureModelE
- LDAE
- LDAModelE
- LocalLDAModelE
- DistributedLDAModelE

Overview of the package

- pyspark.ml.linalg module
  - Vector
  - DenseVector
  - SparseVector
  - Vectors
  - Matrix
  - DenseMatrix
  - SparseMatrix
  - Matrices
- pyspark.ml.recommendation module
  - ALS
  - ALSModel

Overview of the package

pyspark.ml.regression module

- AFTSurvivalRegressionE
- AFTSurvivalRegressionModelE
- DecisionTreeRegressor
- DecisionTreeRegressionModel
- GBTRegressor
- GBTRegressionModel
- GeneralizedLinearRegressionE
- GeneralizedLinearRegressionModelE
- GeneralizedLinearRegressionSummaryE
- GeneralizedLinearRegressionTrainingSummaryE
- IsotonicRegression
- IsotonicRegressionModel
- LinearRegression
- LinearRegressionModel
- LinearRegressionSummaryE
- LinearRegressionTrainingSummaryE
- RandomForestRegressor
- RandomForestRegressionModel

Overview of the package

- pyspark.ml.tuning module
  - ParamGridBuilder
  - CrossValidator
  - CrossValidatorModel
  - TrainValidationSplit E
  - TrainValidationSplitModel E
- pyspark.ml.evaluation module
  - Evaluator
  - BinaryClassificationEvaluator E
  - RegressionEvaluator E
  - MulticlassClassificationEvaluator E

Overview of the package

- pyspark.streaming module
  - Module contents
    - StreamingContext
    - DStream
    - StreamingListener
      - Java
  - pyspark.streaming.kafka module
    - Broker
    - KafkaMessageAndMetadata
    - KafkaUtils
    - OffsetRange
    - TopicAndPartition
    - utf8_decoder
  - pyspark.streaming.kinesis module
    - KinesisUtils
    - InitialPositionInStream
    - utf8_decoder
  - pyspark.streaming.flume.module
    - FlumeUtils
    - utf8_decoder

Overview of the package

pyspark.mllib package

- pyspark.mllib.classification module
    - LogisticRegressionModel
    - LogisticRegressionWithSGD**D**
    - LogisticRegressionWithLBFGS
    - SVMModel
    - SVMWithSGD
    - NaiveBayesModel
    - NaiveBayes
    - StreamingLogisticRegressionWithSGD

Overview of the package

pyspark.mllib.clustering module

- BisectingKMeansModel
- BisectingKMeans
- KMeansModel
- KMeans
- GaussianMixtureModel
- GaussianMixture
- PowerIterationClusteringModel
- PowerIterationClustering
  - Assignment
- StreamingKMeans
- StreamingKMeansModel
- LDA
- LDAModel

Overview of the package

- pyspark.mllib.evaluation module
  - BinaryClassificationMetrics
  - RegressionMetrics
  - MulticlassMetrics
  - RankingMetrics
- pyspark.mllib.feature module
  - Normalizer
  - StandardScalerModel
  - StandardScaler
  - HashingTF
  - IDFModel
  - IDF
  - Word2Vec
  - Word2VecModel
  - ChiSqSelector
  - ChiSqSelectorModel
  - ElementwiseProduct

Overview of the package

- pyspark.mllib.fpm module
  - FPGrowth
    - FreqItemset
  - FPGrowthModel
  - PrefixSpan
    - FreqSequence
  - PrefixSpanModel
- pyspark.mllib.linalg module
  - Vector
  - DenseVector
  - SparseVector
  - Vectors
  - Matrix
  - DenseMatrix
  - SparseMatrix
  - Matrices
  - QRDecomposition

Overview of the package

- pyspark.mllib.linalg.distributed module
  - DistributedMatrix
  - RowMatrix
  - IndexedRow
  - IndexedRowMatrix
  - MatrixEntry
  - CoordinateMatrix
  - BlockMatrix
- pyspark.mllib.random module
  - RandomRDDs

Overview of the package

- pyspark.mllib.recommendation module
  - MatrixFactorizationModel
  - ALS
  - Rating
- pyspark.mllib.regression module
  - LabeledPoint
  - LinearModel
  - LinearRegressionModel
  - LinearRegressionWithSGD**D**
  - RidgeRegressionModel
  - RidgeRegressionWithSGD**D**
  - LassoModel
  - LassoWithSGD**D**
  - IsotonicRegressionModel
  - IsotonicRegression
  - StreamingLinearAlgorithm
  - StreamingLinearRegressionWithSGD

Overview of the package

- pyspark.mllib.stat module
  - Statistics
  - MultivariateStatisticalSummary
  - ChiSqTestResult
  - MultivariateGaussian
  - KernelDensity
- pyspark.mllib.tree module
  - DecisionTreeModel
  - DecisionTree
  - RandomForestModel
  - RandomForest
  - GradientBoostedTreesModel
  - GradientBoostedTrees

Overview of the package

- pyspark.mllib.util module
  - JavaLoader
  - JavaSaveable
  - LinearDataGenerator
  - Loader
  - MLUtils
  - Saveable

Overview of the package

We will build two classification models:

a linear regression and

a random forest.

We will use a portion of the US 2014 and 2015 birth data we downloaded from http://www.cdc.gov/nchs/data_access/vitalstatsonline.htm;

from the total of 300 variables we selected 85 features that we will use to build our models.

Also, out of the total of almost 7.99 million records, we selected a balanced sample of 45,429 records: 22,080 records where infants were reported dead and 23,349 records with infants alive.

The dataset we will use in this module can be downloaded from

http:// www.tomdrabas.com/data/LearningPySpark/births_train. csv.gz.

Loading and transforming the data

Even though MLlib is designed with RDDs and DStreams in focus, for ease of transforming the data we will read the data and convert it to a DataFrame.

The DStreams are the basic data abstraction for Spark Streaming (see http://bit.ly/2jIDT2A)

We first specify the schema of our dataset.

Note that here (for brevity), we only present a handful of features. You should always check our GitHub account for this book for the latest version of the code: https://github.com/drabastomek/ learningPySpark.

Loading and transforming the data

Here's the code:

```
import pyspark.sql.types as typ

labels = [

('INFANT_ALIVE_AT_REPORT', typ.StringType()),

('BIRTH_YEAR', typ.IntegerType()),

('BIRTH_MONTH', typ.IntegerType()),

('BIRTH_PLACE', typ.StringType()),

('MOTHER_AGE_YEARS', typ.IntegerType()),

('MOTHER_RACE_6CODE', typ.StringType()),

('MOTHER_EDUCATION', typ.StringType()),

('FATHER_COMBINED_AGE', typ.IntegerType()),

('FATHER_EDUCATION', typ.StringType()),

('MONTH_PRECARE_RECODE', typ.StringType()),

...

('INFANT_BREASTFED', typ.StringType())

]
```

Loading and transforming the data

```
...
('INFANT_BREASTFED', typ.StringType())
]
schema = typ.StructType([
typ.StructField(e[0], e[1], False) for e in labels
])
```

Next, we load the data. The .read.csv(...) method can read either uncompressed or (as in our case) GZipped comma-separated values. The header parameter set to True indicates that the first row contains the header, and we use the schema to specify the correct data types:

```
births = spark.read.csv('births_train.csv.gz',
header=True,
schema=schema)
```

Loading and transforming the data

There are plenty of features in our dataset that are strings. These are mostly categorical variables that we need to somehow convert to a numeric form.

You can glimpse over the original file schema specification here: ftp://ftp.cdc.gov/pub/Health_Statistics/NCHS/Dataset_ Documentation/DVS/natality/UserGuide2015.pdf.

We will first specify our recode dictionary:

```
recode_dictionary = {

'YNU': {

'Y': 1,

'N': 0,

'U': 0

}

}
```

Loading and transforming the data

Our goal in this Module is to predict whether the 'INFANT_ALIVE_AT_REPORT' is either 1 or 0. Thus, we will drop all of the features that relate to the infant and will try to predict the infant's chances of surviving only based on the features related to its mother, father, and the place of birth

selected_features = [

'INFANT_ALIVE_AT_REPORT', 'BIRTH_PLACE', 'MOTHER_AGE_YEARS', 'FATHER_COMBINED_AGE',

'CIG_BEFORE', 'CIG_1_TRI', 'CIG_2_TRI', 'CIG_3_TRI', 'MOTHER_HEIGHT_IN', 'MOTHER_PRE_WEIGHT',

'MOTHER_DELIVERY_WEIGHT', ’MOTHER_WEIGHT_GAIN', 'DIABETES_PRE', 'DIABETES_GEST',

'HYP_TENS_PRE', 'HYP_TENS_GEST', 'PREV_BIRTH_PRETERM'

]

births_trimmed = births.select(selected_features)

Loading and transforming the data

In our dataset, there are plenty of features with Yes/No/Unknown values; we will only code Yes to 1; everything else will be set to 0.

There is also a small problem with how the number of cigarettes smoked by the mother was coded:

as 0 means the mother smoked no cigarettes before or during the pregnancy,

between 1-97 states the actual number of cigarette smoked,

98 indicates either 98 or more,

whereas 99 identifies the unknown;

we will assume the unknown is 0 and recode accordingly.

Loading and transforming the data

So next we will specify our recoding methods:

```
import pyspark.sql.functions as func

def recode(col, key):

    return recode_dictionary[key][col]

def correct_cig(feat):

    return func \

        .when(func.col(feat) != 99, func.col(feat))\

        .otherwise(0)

rec_integer = func.udf(recode, typ.IntegerType())
```

Loading and transforming the data

The recode method looks up the correct key from the recode_dictionary (given the key) and returns the corrected value.

The correct_cig method checks when the value of the feature feat is not equal to 99 and (for that situation) returns the value of the feature; if the value is equal to 99, we get 0 otherwise.

We cannot use the recode function directly on a DataFrame; it needs to be converted to a UDF that Spark will understand.

The rec_integer is such a function: by passing our specified recode function and specifying the return value data type, we can use it then to encode our Yes/No/Unknown features.

Loading and transforming the data

So, let's get to it. First, we'll correct the features related to the number of cigarettes smoked:

```
births_transformed = births_trimmed \

.withColumn('CIG_BEFORE', correct_cig('CIG_BEFORE'))\

.withColumn('CIG_1_TRI', correct_cig('CIG_1_TRI'))\

.withColumn('CIG_2_TRI', correct_cig('CIG_2_TRI'))\

.withColumn('CIG_3_TRI', correct_cig('CIG_3_TRI'))
```

The .withColumn(...) method takes the name of the column as its first parameter and the transformation as the second one.

In the previous cases, we do not create new columns, but reuse the same ones instead.

Loading and transforming the data

Now we will focus on correcting the Yes/No/Unknown features. First, we will figure out which these are with the following snippet:

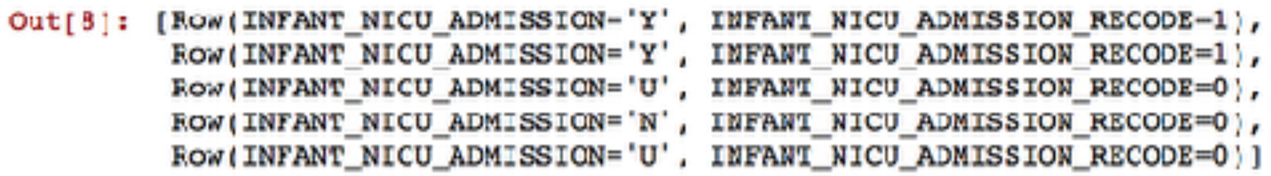
```
cols = [(col.name, col.dataType) for col in births_trimmed.schema]

YNU_cols = []

for i, s in enumerate(cols):

    if s[1] == typ.StringType():

    dis = births.select(s[0]).distinct().rdd.map(lambda row: row[0]) .collect()

    if 'Y' in dis:

        YNU_cols.append(s[0])
```

Loading and transforming the data

First, we created a list of tuples (cols) that hold column names and corresponding data types. Next, we loop through all of these and calculate distinct values of all string columns; if a 'Y' is within the returned list, we append the column name to the YNU_cols list.

DataFrames can transform the features in bulk while selecting features. To present the idea, consider the following example:

```
births.select([

'INFANT_NICU_ADMISSION',

rec_integer(

'INFANT_NICU_ADMISSION', func.lit('YNU')

) \

.alias('INFANT_NICU_ADMISSION_RECODE')]

).take(5)
```



```
Out[3]: [Row(INFANT_NICU_ADMISSION='Y', INFANT_NICU_ADMISSION_RECODE=1),
         Row(INFANT_NICU_ADMISSION='Y', INFANT_NICU_ADMISSION_RECODE=1),
         Row(INFANT_NICU_ADMISSION='U', INFANT_NICU_ADMISSION_RECODE=0),
         Row(INFANT_NICU_ADMISSION='N', INFANT_NICU_ADMISSION_RECODE=0),
         Row(INFANT_NICU_ADMISSION='U', INFANT_NICU_ADMISSION_RECODE=0)]
```

Loading and transforming the data

We select the 'INFANT_NICU_ADMISSION' column and we pass the name of the feature to the rec_integer method. We also alias the newly transformed column as 'INFANT_NICU_ADMISSION_RECODE'. This way we will also confirm that our UDF works as intended.

So, to transform all the YNU_cols in one go, we will create a list of such transformations, as shown here:

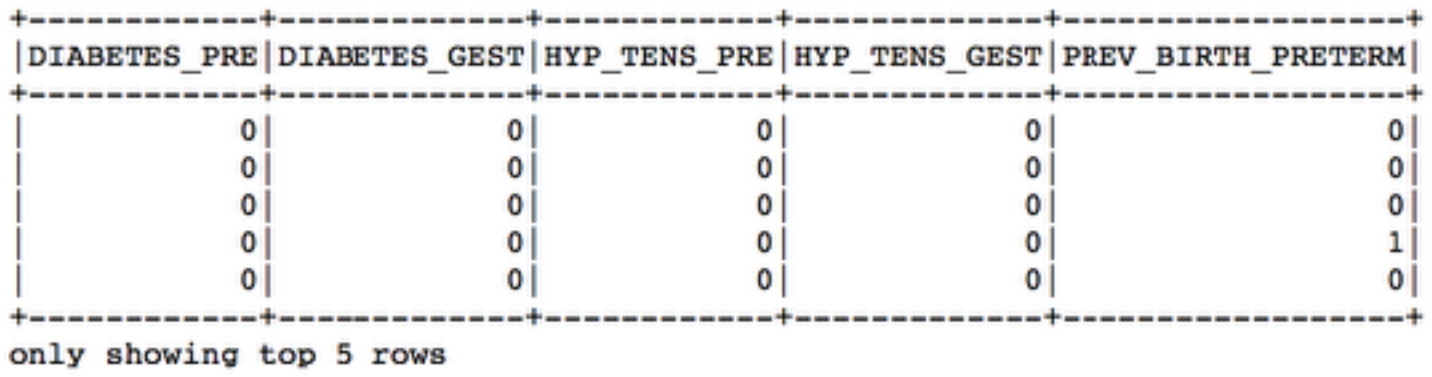
```
exprs_YNU = [

rec_integer(x, func.lit('YNU')).alias(x)

if x in YNU_cols

else x

for x in births_transformed.columns

]

births_transformed = births_transformed.select(exprs_YNU)
```

Loading and transforming the data

Let's check if we got it correctly:

births_transformed.select(YNU_cols[-5:]).show(5)

Here's what we get:



```
+------------+-------------+------------+-------------+-----------------+
|DIABETES_PRE|DIABETES_GEST|HYP_TENS_PRE|HYP_TENS_GEST|PREV_BIRTH_PRETERM|
+------------+-------------+------------+-------------+-----------------+
|           0|            0|           0|            0|                0|
|           0|            0|           0|            0|                0|
|           0|            0|           0|            0|                0|
|           0|            0|           0|            0|                1|
|           0|            0|           0|            0|                0|
+------------+-------------+------------+-------------+-----------------+
only showing top 5 rows
```

Looks like everything worked as we wanted it to work, so let's get to know our data better.

Getting to know your data

In order to build a statistical model in an informed way, an intimate knowledge of the dataset is necessary.

Without knowing the data it is possible to build a successful model, but it is then a much more arduous task, or it would require more technical resources to test all the possible combinations of features.

Therefore, after spending the required 80% of the time cleaning the data, we spend the next 15% getting to know it!

Descriptive statistics

I normally start with descriptive statistics. Even though the DataFrames expose the .describe() method, since we are working with MLlib, we will use the .colStats(...) method.

Note A word of warning: the .colStats(...) calculates the descriptive statistics based on a sample. For real world datasets this should not really matter but if your dataset has less than 100 observations you might get some strange results.

The method takes an RDD of data to calculate the descriptive statistics of and return a MultivariateStatisticalSummary object that contains the following descriptive statistics:

- count(): This holds a row count
- max(): This holds maximum value in the column
- mean(): This holds the value of the mean for the values in the column

Descriptive statistics

- min(): This holds the minimum value in the column
- normL1(): This holds the value of the L1-Norm for the values in the column
- normL2(): This holds the value of the L2-Norm for the values in the column
- numNonzeros(): This holds the number of nonzero values in the column
- variance(): This holds the value of the variance for the values in the column

You can read more about the L1- and L2-norms here http://bit.ly/2jJJPJ0

Descriptive statistics

We recommend checking the documentation of Spark to learn more about these. The following is a snippet that calculates the descriptive statistics of the numeric features:

```
import pyspark.mllib.stat as st

import numpy as np

numeric_cols = ['MOTHER_AGE_YEARS','FATHER_COMBINED_AGE',

'CIG_BEFORE','CIG_1_TRI','CIG_2_TRI','CIG_3_TRI',

'MOTHER_HEIGHT_IN','MOTHER_PRE_WEIGHT',

'MOTHER_DELIVERY_WEIGHT','MOTHER_WEIGHT_GAIN'

]
```

Descriptive statistics

```
numeric_rdd = births_transformed\
.select(numeric_cols)\
.rdd \
.map(lambda row: [e for e in row])
mllib_stats = st.Statistics.colStats(numeric_rdd)
for col, m, v in zip(numeric_cols,
mllib_stats.mean(),
mllib_stats.variance()):
print('{0}: \t{1:.2f} \t {2:.2f}'.format(col, m, np.sqrt(v)))
```

Descriptive statistics

The preceding code produces the following result:

```
MOTHER_AGE_YEARS:        28.30    6.08
FATHER_COMBINED_AGE:     44.55    27.55
CIG_BEFORE:        1.43    5.18
CIG_1_TRI:         0.91    3.83
CIG_2_TRI:         0.70    3.31
CIG_3_TRI:         0.58    3.11
MOTHER_HEIGHT_IN:        65.12    6.45
MOTHER_PRE_WEIGHT:       214.50   210.21
MOTHER_DELIVERY_WEIGHT:          223.63   180.01
MOTHER_WEIGHT_GAIN:      30.74    26.23
```

Descriptive statistics

As you can see, mothers, compared to fathers, are younger: the average age of mothers was 28 versus over 44 for fathers. A good indication (at least for some of the infants) was that many mothers quit smoking while being pregnant; it is horrifying, though, that there still were some that continued smoking.

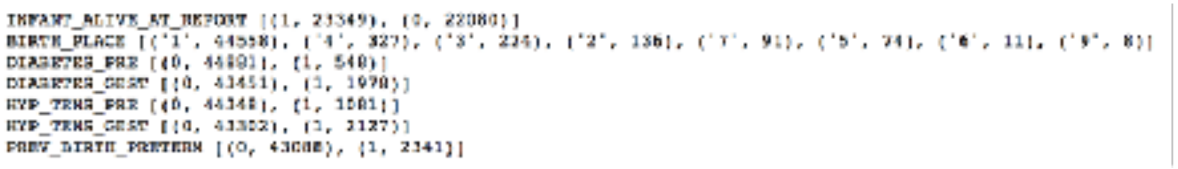
For the categorical variables, we will calculate the frequencies of their values:

```
categorical_cols = [e for e in births_transformed.columns

if e not in numeric_cols]

categorical_rdd = births_transformed\

.select(categorical_cols)\

.rdd \

.map(lambda row: [e for e in row])
```

Descriptive statistics

```
for i, col in enumerate(categorical_cols):
agg = categorical_rdd \
.groupBy(lambda row: row[i]) \
.map(lambda row: (row[0], len(row[1])))
print(col, sorted(agg.collect(),
key=lambda el: el[1],
reverse=True))
```

Here is what the results look like:



```
INFANT_ALIVE_AT_REPORT [(1, 23349), (0, 22080)]
BIRTH_PLACE [('1', 44558), ('4', 327), ('3', 224), ('2', 136), ('7', 91), ('5', 74), ('6', 11), ('9', 8)]
DIABETES_PRE [(0, 44881), (1, 548)]
DIABETES_GEST [(0, 43451), (1, 1978)]
HYP_TENS_PRE [(0, 44348), (1, 1081)]
HYP_TENS_GEST [(0, 43302), (1, 2127)]
PREV_BIRTH_PRETERM [(0, 43088), (1, 2341)]
```

Most of the deliveries happened in hospital (BIRTH_PLACE equal to 1). Around 550 deliveries happened at home: some intentionally ('BIRTH_PLACE' equal to 3), and some not ('BIRTH_PLACE' equal to 4).

Correlations

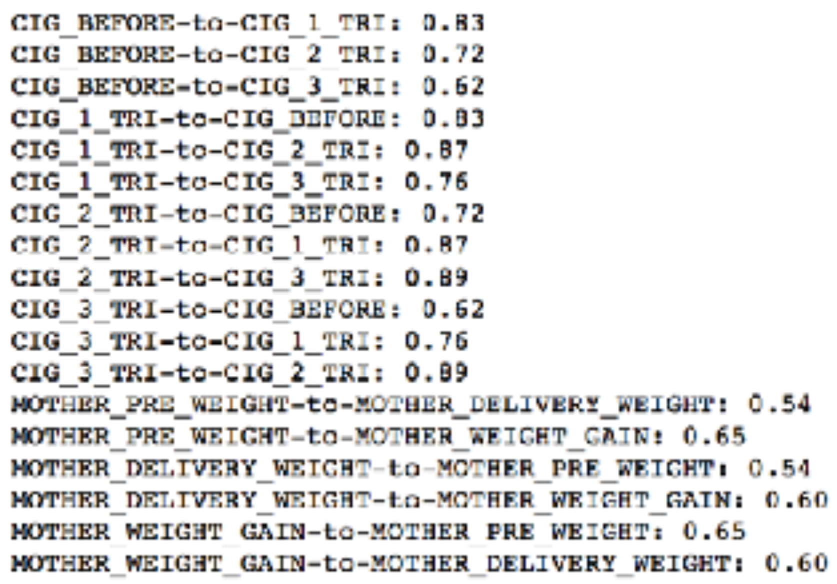
Correlations help to identify collinear numeric features and handle them appropriately. Let's check the correlations between our features:

```
corrs = st.Statistics.corr(numeric_rdd)

for i, el in enumerate(corrs > 0.5):

    correlated = [

    (numeric_cols[j], corrs[i][j])

    for j, e in enumerate(el)

        if e == 1.0 and j != i]

        if len(correlated) > 0:

            for e in correlated:

                print('{0}-to-{1}: {2:.2f}’.format(numeric_cols[i], e[0], e[1]))
```

Correlations

The preceding code will calculate the correlation matrix and will print only those features that have a correlation coefficient greater than 0.5: the corrs > 0.5 part takes care of that.

Here's what we get:



```
CIG_BEFORE-to-CIG_1_TRI: 0.83
CIG_BEFORE-to-CIG_2_TRI: 0.72
CIG_BEFORE-to-CIG_3_TRI: 0.62
CIG_1_TRI-to-CIG_BEFORE: 0.83
CIG_1_TRI-to-CIG_2_TRI: 0.87
CIG_1_TRI-to-CIG_3_TRI: 0.76
CIG_2_TRI-to-CIG_BEFORE: 0.72
CIG_2_TRI-to-CIG_1_TRI: 0.87
CIG_2_TRI-to-CIG_3_TRI: 0.89
CIG_3_TRI-to-CIG_BEFORE: 0.62
CIG_3_TRI-to-CIG_1_TRI: 0.76
CIG_3_TRI-to-CIG_2_TRI: 0.89
MOTHER_PRE_WEIGHT-to-MOTHER_DELIVERY_WEIGHT: 0.54
MOTHER_PRE_WEIGHT-to-MOTHER_WEIGHT_GAIN: 0.65
MOTHER_DELIVERY_WEIGHT-to-MOTHER_PRE_WEIGHT: 0.54
MOTHER_DELIVERY_WEIGHT-to-MOTHER_WEIGHT_GAIN: 0.60
MOTHER_WEIGHT_GAIN-to-MOTHER_PRE_WEIGHT: 0.65
MOTHER_WEIGHT_GAIN-to-MOTHER_DELIVERY_WEIGHT: 0.60
```

Correlations

As you can see, the 'CIG_...' features are highly correlated, so we can drop most of them. Since we want to predict the survival chances of an infant as soon as possible, we will keep only the 'CIG_1_TRI'. Also, as expected, the weight features are also highly correlated and we will only keep the 'MOTHER_PRE_WEIGHT':

```
features_to_keep = [

'INFANT_ALIVE_AT_REPORT', 'BIRTH_PLACE', 'MOTHER_AGE_YEARS',

'FATHER_COMBINED_AGE', 'CIG_1_TRI', 'MOTHER_HEIGHT_IN',
'MOTHER_PRE_WEIGHT',

'DIABETES_PRE', 'DIABETES_GEST', 'HYP_TENS_PRE',

'HYP_TENS_GEST', 'PREV_BIRTH_PRETERM'

]

births_transformed = births_transformed.select([e for e in features_ to_keep])
```

Statistical testing

We cannot calculate correlations for the categorical features. However, we can run a Chi-square test to determine if there are significant differences.

Here's how you can do it using the .chiSqTest(...) method of MLlib:

```
import pyspark.mllib.linalg as ln

for cat in categorical_cols[1:]:

    agg = births_transformed \

    .groupby('INFANT_ALIVE_AT_REPORT') .pivot(cat) .count()

agg_rdd = agg.rdd.map(lambda row: (row[1:]))\

.flatMap(lambda row: [0 if e == None else e for e in row]) .collect()
```

Statistical testing

```
row_length = len(agg.collect()[0]) - 1
agg = ln.Matrices.dense(row_length, 2, agg_rdd)
test = st.Statistics.chiSqTest(agg)
print(cat, round(test.pValue, 4))
```

We loop through all the categorical variables and pivot them by the 'INFANT_ALIVE_AT_REPORT' feature to get the counts.

Next, we transform them into an RDD, so we can then convert them into a matrix using the pyspark.mllib.linalg module.

The first parameter to the .Matrices.dense(...) method specifies the number of rows in the matrix; in our case, it is the length of distinct values of the categorical feature.

Statistical testing

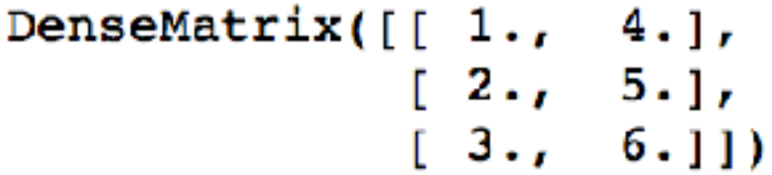
The second parameter specifies the number of columns: we have two as our 'INFANT_ALIVE_AT_REPORT' target variable has only two values.

The last parameter is a list of values to be transformed into a matrix.

Here's an example that shows this more clearly:

print(ln.Matrices.dense(3,2, [1,2,3,4,5,6]))

The preceding code produces the following matrix:



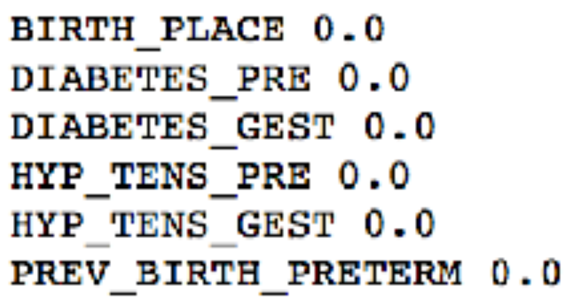
```
DenseMatrix([[ 1.,   4.],
             [ 2.,   5.],
             [ 3.,   6.]])
```

Statistical testing

Once we have our counts in a matrix form, we can use the .chiSqTest(...) to calculate our test.

Here's what we get in return:

Our tests reveal that all the features should be significantly different and should help us predict the chance of survival of an infant.



```
BIRTH_PLACE 0.0
DIABETES_PRE 0.0
DIABETES_GEST 0.0
HYP_TENS_PRE 0.0
HYP_TENS_GEST 0.0
PREV_BIRTH_PRETERM 0.0
```

Creating the final dataset

Therefore, it is time to create our final dataset that we will use to build our models. We will convert our DataFrame into an RDD of LabeledPoints.

A LabeledPoint is a MLlib structure that is used to train the machine learning models. It consists of two attributes: label and features.

The label is our target variable and features can be a NumPy array, list, pyspark.mllib.linalg.SparseVector, pyspark.mllib.linalg.DenseVector, or scipy.sparse column matrix.

Creating an RDD of LabeledPoints

Before we build our final dataset, we first need to deal with one final obstacle: our 'BIRTH_PLACE' feature is still a string. While any of the other categorical variables can be used as is (as they are now dummy variables), we will use a hashing trick to encode the 'BIRTH_PLACE' feature:

```
import pyspark.mllib.feature as ft

import pyspark.mllib.regression as reg

hashing = ft.HashingTF(7)

births_hashed = births_transformed.rdd .map(lambda row: [

list(hashing.transform(row[1]).toArray())

if col == 'BIRTH_PLACE'

    else row[i]

    for i, col
```

Creating an RDD of LabeledPoints

```
    for i, col in enumerate(features_to_keep)]) \
.map(lambda row: [[e] if type(e) == int else e
    for e in row]).map(lambda row: [item for sublist in row
    for item in sublist]).map(lambda row: reg.LabeledPoint( row[0],
ln.Vectors.dense(row[1:]))  )
```

First, we create the hashing model. Our feature has seven levels, so we use as many features as that for the hashing trick.

Next, we actually use the model to convert our 'BIRTH_PLACE' feature into a SparseVector; such a data structure is preferred if your dataset has many columns but in a row only a few of them have non-zero values. We then combine all the features together and finally create a LabeledPoint.

Splitting into training and testing

Before we move to the modeling stage, we need to split our dataset into two sets: one we'll use for training and the other for testing.

Luckily, RDDs have a handy method to do just that:

.randomSplit(...).

The method takes a list of proportions that are to be used to randomly split the dataset.

Here is how it is done:

```
births_train, births_test = births_hashed.randomSplit([0.6, 0.4])
```

Predicting infant survival

Finally, we can move to predicting the infants' survival chances.

In this section, we will build two models:

a linear classifier—the logistic regression, and

a non-linear one—a random forest.

For the former one, we will use all the features at our disposal,

whereas for the latter one, we will employ a ChiSqSelector(...) method to select the top four features.

Logistic regression in MLlib

Logistic regression is somewhat a benchmark to build any classification model.

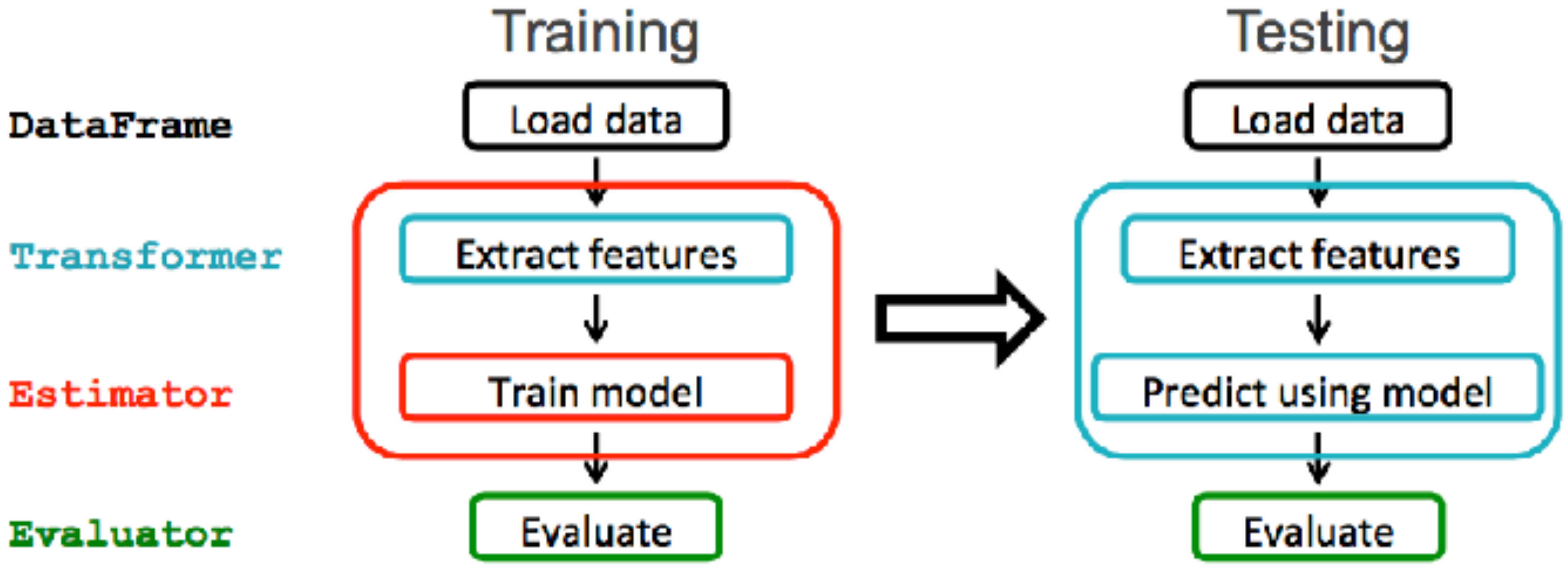
MLlib used to provide a logistic regression model estimated using a stochastic gradient descent (SGD) algorithm.

This model has been deprecated in Spark 2.0 in favor of the LogisticRegressionWithLBFGS model.

The LogisticRegressionWithLBFGS model uses the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (BFGS) optimization algorithm. It is a quasi-Newton method that approximates the BFGS algorithm.

Note : For those of you who are mathematically adept and interested in this, we suggest perusing this blog post that is a nice walk-through of the optimization algorithms: http://aria42.com/blog/2014/12/ understanding-lbfgs.

Logistic regression in MLlib



Logistic regression in MLlib

First, we train the model on our data:

```
from pyspark.mllib.classification \
import LogisticRegressionWithLBFGS
LR_Model = LogisticRegressionWithLBFGS \
.train(births_train, iterations=10)
```

Training the model is very simple: we just need to call the .train(...) method. The required parameters are the RDD with LabeledPoints; we also specified the number of iterations so it does not take too long to run.

Logistic regression in MLlib

Having trained the model using the births_train dataset, let's use the model to predict the classes for our testing set:

```
LR_results = (
births_test.map(lambda row: row.label) \
.zip(LR_Model \
.predict(births_test\
.map(lambda row: row.features)))
).map(lambda row: (row[0], row[1] * 1.0))
```

Logistic regression in MLlib

The preceding snippet creates an RDD where each element is a tuple, with the first element being the actual label and the second one, the model's prediction.

MLlib provides an evaluation metric for classification and regression. Let's check how well or how bad our model performed:

```
import pyspark.mllib.evaluation as ev

LR_evaluation = ev.BinaryClassificationMetrics(LR_results)

print('Area under PR: {0:.2f}' \

.format(LR_evaluation.areaUnderPR))

print('Area under ROC: {0:.2f}' \

.format(LR_evaluation.areaUnderROC))

LR_evaluation.unpersist()
```

Logistic regression in MLlib

Here's what we got:

The model performed reasonably well! The 85% area under the Precision-Recall curve indicates a good fit. In this case, we might be getting slightly more predicted deaths (true and false positives). In this case, this is actually a good thing as it would allow doctors to put the expectant mother and the infant under special care.

The area under Receiver-Operating Characteristic (ROC) can be understood as a probability of the model ranking higher than a randomly chosen positive instance compared to a randomly chosen negative one. A 63% value can be thought of as acceptable.

For more on these metrics, we point interested readers to http:// stats.stackexchange.com/questions/7207/ roc-vs-precision-and-recall-curves and http://gim. unmc.edu/dxtests/roc3.htm.

Selecting only the most predictable features

Any model that uses less features to predict a class accurately should always be preferred to a more complex one. MLlib allows us to select the most predictable features using a Chi-Square selector.

Here's how you do it:

```
selector = ft.ChiSqSelector(4).fit(births_train)

topFeatures_train = (

births_train.map(lambda row: row.label) \

.zip(selector \

.transform(births_train \

.map(lambda row: row.features)))

).map(lambda row: reg.LabeledPoint(row[0], row[1]))
```

Selecting only the most predictable features

```
topFeatures_test = (
births_test.map(lambda row: row.label) \
.zip(selector \
.transform(births_test \
.map(lambda row: row.features)))
).map(lambda row: reg.LabeledPoint(row[0], row[1]))
```

We asked the selector to return the four most predictive features from the dataset and train the selector using the births_train dataset. We then used the model to extract only those features from our training and testing datasets.

The .ChiSqSelector(...) method can only be used for numerical features; categorical variables need to be either hashed or dummy coded before the selector can be used.

Random forest in MLlib

We are now ready to build the random forest model.

The following code shows you how to do it:

```
from pyspark.mllib.tree import RandomForest
RF_model = RandomForest \
.trainClassifier(data=topFeatures_train,
numClasses=2,
categoricalFeaturesInfo={},
numTrees=6,
featureSubsetStrategy='all',
seed=666)
```

Random forest in MLlib

The first parameter to the .trainClassifier(...) method specifies the training dataset.

The numClasses one indicates how many classes our target variable has.

As the third parameter, you can pass a dictionary where the key is the index of a categorical feature in our RDD and the value for the key indicates the number of levels that the categorical feature has.

The numTrees specifies the number of trees to be in the forest.

The next parameter tells the model to use all the features in our dataset instead of keeping only the most descriptive ones, while the last one specifies the seed for the stochastic part of the model.

Random forest in MLlib

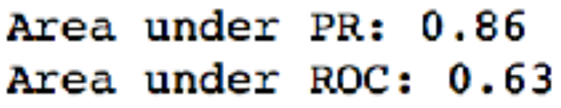
Let's see how well our model did:

```
RF_results = (
topFeatures_test.map(lambda row: row.label) \
.zip(RF_model \
.predict(topFeatures_test \
.map(lambda row: row.features)))
)
RF_evaluation = ev.BinaryClassificationMetrics(RF_results)
print('Area under PR: {0:.2f}' \
.format(RF_evaluation.areaUnderPR))
```

Random forest in MLlib

```
print('Area under ROC: {0:.2f}' \
.format(RF_evaluation.areaUnderROC))
model_evaluation.unpersist()
```

Here are the results:



```
Area under PR: 0.86
Area under ROC: 0.63
```

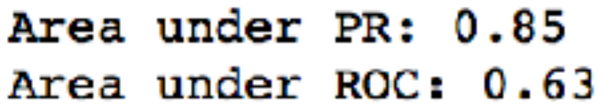
As you can see, the Random Forest model with fewer features performed even better than the logistic regression model. Let's see how the logistic regression would perform with a reduced number of features:

```
LR_Model_2 = LogisticRegressionWithLBFGS \
.train(topFeatures_train, iterations=10)
LR_results_2 = ( topFeatures_test.map(lambda row: row.label).zip(LR_Model_2 \
.predict(topFeatures_test.map(lambda row: row.features)))
).map(lambda row: (row[0], row[1] * 1.0))
```

Random forest in MLlib

```
LR_evaluation_2 = ev.BinaryClassificationMetrics(LR_results_2)
print('Area under PR: {0:.2f}' \
.format(LR_evaluation_2.areaUnderPR))
print('Area under ROC: {0:.2f}' \
.format(LR_evaluation_2.areaUnderROC))
LR_evaluation_2.unpersist()
```

The results might surprise you:



```
Area under PR: 0.85
Area under ROC: 0.63
```

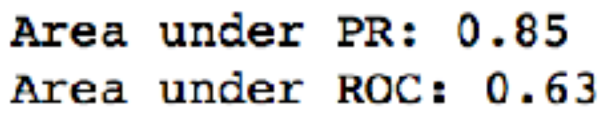
As you can see, both models can be simplified and still attain the same level of accuracy.

Having said that, you should always opt for a model with fewer variables.

Random forest in MLlib

```
LR_evaluation_2 = ev.BinaryClassificationMetrics(LR_results_2)
print('Area under PR: {0:.2f}' \
.format(LR_evaluation_2.areaUnderPR))
print('Area under ROC: {0:.2f}' \
.format(LR_evaluation_2.areaUnderROC))
LR_evaluation_2.unpersist()
```

The results might surprise you:



```
Area under PR: 0.85
Area under ROC: 0.63
```

As you can see, both models can be simplified and still attain the same level of accuracy.

Having said that, you should always opt for a model with fewer variables.

Summary

- In this Module we looked at the capabilities of the MLlib package of PySpark.
- Even though the package is currently in a maintenance mode and is not actively being worked on, it is still good to know how to use it.
- Also, for now it is the only package available to train models while streaming data.
- We used MLlib to clean up, transform, and get familiar with the dataset of infant deaths.
- Using that knowledge we then successfully built two models that aimed at predicting the chance of infant survival given the information about its mother, father, and place of birth.

Module 6

Introducing the ML Package

Agenda

Module 6
Introducing the ML Package
Overview of the package
Predicting the chances of infant survival with ML
Parameter hyper-tuning
Other features of PySpark ML in action
Implementation of ML Algorithm
- Random Forest
- Regression
- K-means

Assignments 5

Introducing the ML Package

In the previous module, we worked with the MLlib package in Spark that operated strictly on RDDs.

In this module, we move to the ML part of Spark that operates strictly on DataFrames.

Also, according to the Spark documentation, the primary machine learning API for Spark is now the DataFrame-based set of models contained in the spark.ml package.

So, let's get to it!

In this module, we will reuse a portion of the dataset we played within the previous module.

The data can be downloaded from http://www.tomdrabas.com/data/LearningPySpark/births_transformed.csv.gz.

Introducing the ML Package

Objectives

You will learn how to do the following:

- Prepare transformers, estimators, and pipelines
- Predict the chances of infant survival using models available in the ML package
- Evaluate the performance of the model
- Perform parameter hyper-tuning
- Use other machine-learning models available in the package

Overview of the package

At the top level, the package exposes three main abstract classes:

- a Transformer,
- an Estimator, and
- a Pipeline.

Transformer

The Transformer class, like the name suggests, transforms your data by (normally) appending a new column to your DataFrame.

At the high level, when deriving from the Transformer abstract class, each and every new Transformer needs to implement a .transform(...) method.

The method, as a first and normally the only obligatory parameter, requires passing a DataFrame to be transformed.

Transformer

This, of course, varies method-by-method in the ML package: other popular parameters are inputCol and outputCol; these, however, frequently default to some predefined values, such as, for example, 'features' for the inputCol parameter.

There are many Transformers offered in the spark.ml.feature and we will briefly describe them here:

Binarizer: Given a threshold, the method takes a continuous variable and transforms it into a binary one.

Bucketizer: Similar to the Binarizer, this method takes a list of thresholds (the splits parameter) and transforms a continuous variable into a multinomial one.

ChiSqSelector: For the categorical target variables (think classification models), this feature allows you to select a predefined number of features (parameterized by the numTopFeatures parameter) that explain the variance in the target the best.

Transformer

This, of course, varies method-by-method in the ML package: other popular parameters are inputCol and outputCol; these, however, frequently default to some predefined values, such as, for example, 'features' for the inputCol parameter.

There are many Transformers offered in the spark.ml.feature and we will briefly describe them here:

Binarizer: Given a threshold, the method takes a continuous variable and transforms it into a binary one.

Bucketizer: Similar to the Binarizer, this method takes a list of thresholds (the splits parameter) and transforms a continuous variable into a multinomial one.

ChiSqSelector: For the categorical target variables (think classification models), this feature allows you to select a predefined number of features (parameterized by the numTopFeatures parameter) that explain the variance in the target the best.

More information on Chi-squares can be found here:

http:// ccnmtl.columbia.edu/projects/qmss/the_chisquare_test/ about_the_chisquare_test.html.

Transformer

ChiSqSelector: The selection is done, as the name of the method suggests, using a Chi-Square test. It is one of the two-step methods:

first, you need to .fit(...) your data (so the method can calculate the Chi-square tests).

Calling the .fit(...) method (you pass your DataFrame as a parameter) returns a ChiSqSelectorModel object that you can then use to transform your DataFrame using the .transform(...) method.

CountVectorizer: This is useful for a tokenized text (such as [['Learning', 'PySpark', 'with', 'us'],['us', 'us', 'us']]). It is one of two-step methods:

first, you need to .fit(...), that is, learn the patterns from your dataset, before you can .transform(...) with the CountVectorizerModel returned by the .fit(...) method.

The output from this transformer, for the tokenized text presented previously, would look similar to this: [(4, [0, 1, 2, 3], [1.0, 1.0, 1.0, 1.0]),(4, [3], [3.0])].

Transformer

DCT: The Discrete Cosine Transform takes a vector of real values and returns a vector of the same length, but with the sum of cosine functions oscillating at different frequencies.

Such transformations are useful to extract some underlying frequencies in your data or in data compression.

• **ElementwiseProduct:** A method that returns a vector with elements that are products of the vector passed to the method, and a vector passed as the scalingVec parameter.

For example, if you had a [10.0, 3.0, 15.0] vector and your scalingVec was [0.99, 3.30, 0.66], then the vector you would get would look as follows: [9.9, 9.9, 9.9].

Transformer

HashingTF: A **hashing trick transformer** that takes a list of tokenized text and returns a vector (of predefined length) with counts.

From PySpark's documentation:

"Since a simple modulo is used to transform the hash function to a column index, it is advisable to use a power of two as the numFeatures parameter; otherwise the features will not be mapped evenly to the columns."

IDF: This method computes an **Inverse Document Frequency** for a list of documents. Note that the documents need to already be represented as a vector (for example, using either the HashingTF or CountVectorizer).

IndexToString: A complement to the StringIndexer method. It uses the encoding from the StringIndexerModel object to reverse the string index to original values. As an aside, please note that this sometimes does not work and you need to specify the values from the StringIndexer.

Transformer

MaxAbsScaler: Rescales the data to be within the [-1.0, 1.0] range (thus, it does not shift the center of the data).

• **MinMaxScaler**: This is similar to the MaxAbsScaler with the difference that it scales the data to be in the [0.0, 1.0] range.

• **NGram**: This method takes a list of tokenized text and returns n-grams: pairs, triples, or n-mores of subsequent words. For example, if you had a ['good', 'morning', 'Robin', 'Williams'] vector you would get the following output: ['good morning', 'morning Robin', 'Robin Williams'].

Normalizer: This method scales the data to be of unit norm using the p-norm value (by default, it is L2).

• **OneHotEncoder**: This method encodes a categorical column to a column of binary vectors.

• **PCA**: Performs the data reduction using principal component analysis.

Transformer

PolynomialExpansion: Performs a polynomial expansion of a vector. For example, if you had a vector symbolically written as [x, y, z], the method would produce the following expansion: [x, x*x, y, x*y, y*y, z, x*z, y*z, z*z].

• **QuantileDiscretizer**: Similar to the Bucketizer method, but instead of passing the splits parameter, you pass the numBuckets one. The method then decides, by calculating approximate quantiles over your data, what the splits should be.

• **RegexTokenizer**: This is a string tokenizer using regular expressions.

• **RFormula**: For those of you who are avid R users, you can pass a formula such as vec ~ alpha * 3 + beta (assuming your DataFrame has the alpha and beta columns) and it will produce the vec column given the expression.

SQLTransformer: Similar to the previous, but instead of R-like formulas, you can use SQL syntax.

Transformer

StandardScaler: Standardizes the column to have a 0 mean and standard deviation equal to 1.

StopWordsRemover: Removes stop words (such as 'the' or 'a') from a tokenized text.

StringIndexer: Given a list of all the words in a column, this will produce a vector of indices.

Tokenizer: This is the default tokenizer that converts the string to lower case and then splits on space(s).

VectorAssembler: This is a highly useful transformer that collates multiple numeric (vectors included) columns into a single column with a vector representation.

Transformer

VectorAssembler: For example, if you had three columns in your DataFrame:

```
df = spark.createDataFrame(
[(12, 10, 3), (1, 4, 2)],
['a', 'b', 'c'])
```

The output of calling:

```
ft.VectorAssembler(inputCols=['a', 'b', 'c'],
outputCol='features').transform(df).select(‘features').collect()
```

It would look as follows:

```
[Row(features=DenseVector([12.0, 10.0, 3.0])),
Row(features=DenseVector([1.0, 4.0, 2.0]))]
```

Transformer

VectorIndexer: This is a method for indexing categorical columns into a vector of indices. It works in a column-by-column fashion, selecting distinct values from the column, sorting and returning an index of the value from the map instead of the original value.

VectorSlicer: Works on a feature vector, either dense or sparse: given a list of indices, it extracts the values from the feature vector.

Word2Vec: This method takes a sentence (string) as an input and transforms it into a map of {string, vector} format, a representation that is useful in natural language processing.

Estimators

Estimators can be thought of as statistical models that need to be estimated to make predictions or classify your observations.

If deriving from the abstract Estimator class, the new model has to implement the .fit(...) method that fits the model given the data found in a DataFrame and some default or user-specified parameters.

There are a lot of estimators available in PySpark and we will now shortly discuss the models available in Spark 2.0.

Classification

The ML package provides a data scientist with seven classification models to choose from.

LogisticRegression: The benchmark model for classification. The logistic regression uses a logit function to calculate the probability of an observation belonging to a particular class.

DecisionTreeClassifier: A classifier that builds a decision tree to predict a class for an observation. Specifying the maxDepth parameter limits the depth the tree grows, the minInstancePerNode determines the minimum number of observations in the tree node required to further split, the maxBins parameter specifies the maximum number of bins the continuous variables will be split into, and the impurity specifies the metric to measure and calculate the information gain from the split.

GBTClassifier: A Gradient Boosted Trees model for classification. The model belongs to the family of ensemble models: models that combine multiple weak predictive models to form a strong one. At the moment, the GBTClassifier model supports binary labels, and continuous and categorical features.

Classification

RandomForestClassifier: This model produces multiple decision trees (hence the name—forest) and uses the mode output of those decision trees to classify observations. The RandomForestClassifier supports both binary and multinomial labels.

• **NaiveBayes**: Based on the Bayes' theorem, this model uses conditional probability theory to classify observations. The NaiveBayes model in PySpark ML supports both binary and multinomial labels.

• **MultilayerPerceptronClassifier**: A classifier that mimics the nature of a human brain. Deeply rooted in the Artificial Neural Networks theory, the model is a black-box, that is, it is not easy to interpret the internal parameters of the model.

The model consists, at a minimum, of three, fully connected layers (a parameter that needs to be specified when creating the model object) of artificial neurons:

- the input layer (that needs to be equal to the number of features in your dataset),
- a number of hidden layers (at least one), and
- an output layer with the number of neurons equal to the number of categories in your label. All the neurons in the input and hidden layers have a sigmoid activation function, whereas the activation function of the neurons in the output layer is softmax.

Classification

OneVsRest: A reduction of a multiclass classification to a binary one. For example, in the case of a multinomial label, the model can train multiple binary logistic regression models. For example, if label == 2, the model will build a logistic regression where it will convert the label == 2 to 1 (all remaining label values would be set to 0) and then train a binary model. All the models are then scored and the model with the highest probability wins.

Regression

There are seven models available for regression tasks in the PySpark ML package. As with classification, these range from some basic ones (such as the obligatory linear regression) to more complex ones:

AFTSurvivalRegression: Fits an Accelerated Failure Time regression model. It is a parametric model that assumes that a marginal effect of one of the features accelerates or decelerates a life expectancy (or process failure). It is highly applicable for the processes with well-defined stages.

DecisionTreeRegressor: Similar to the model for classification with an obvious distinction that the label is continuous instead of binary (or multinomial).

GBTRegressor: As with the DecisionTreeRegressor, the difference is the data type of the label.

Regression

GeneralizedLinearRegression: A family of linear models with differing kernel functions (link functions). In contrast to the linear regression that assumes normality of error terms, the GLM allows the label to have different error term distributions: the GeneralizedLinearRegression model from the PySpark ML package supports gaussian, binomial, gamma, and poisson families of error distributions with a host of different link functions.

IsotonicRegression: A type of regression that fits a free-form, non-decreasing line to your data. It is useful to fit the datasets with ordered and increasing observations.

LinearRegression: The most simple of regression models, it assumes a linear relationship between features and a continuous label, and normality of error terms.

RandomForestRegressor: Similar to either DecisionTreeRegressor or GBTRegressor, the RandomForestRegressor fits a continuous label instead of a discrete one.

Clustering

Clustering is a family of unsupervised models that are used to find underlying patterns in your data. The PySpark ML package provides the four most popular models at the moment:

BisectingKMeans: A combination of the k-means clustering method and hierarchical clustering. The algorithm begins with all observations in a single cluster and iteratively splits the data into k clusters.

Check out this website for more information on pseudo-algorithms:

http://minethedata.blogspot.com/2012/08/bisecting-k-means.html.

KMeans: This is the famous k-mean algorithm that separates data into k clusters, iteratively searching for centroids that minimize the sum of square distances between each observation and the centroid of the cluster it belongs to.

Clustering

GaussianMixture: This method uses k Gaussian distributions with unknown parameters to dissect the dataset. Using the Expectation-Maximization algorithm, the parameters for the Gaussians are found by maximizing the log-likelihood function.

Note: Beware that for datasets with many features this model might perform poorly due to the curse of dimensionality and numerical issues with Gaussian distributions.

LDA: This model is used for topic modeling in natural language processing applications.

Pipeline

A **Pipeline** in PySpark ML is a concept of an end-to-end transformation-estimation process (with distinct stages) that ingests some raw data (in a DataFrame form), performs the necessary data carpentry (transformations), and finally estimates a statistical model (estimator).

Note: A Pipeline can be purely transformative, that is, consisting of Transformers only.

A Pipeline can be thought of as a chain of multiple discrete stages. When a .fit(...) method is executed on a Pipeline object, all the stages are executed in the order they were specified in the stages parameter; the stages parameter is a list of Transformer and Estimator objects. The .fit(...) method of the Pipeline object executes the .transform(...) method for the Transformers and the .fit(...) method for the Estimators.

Normally, the output of a preceding stage becomes the input for the following stage: when deriving from either the Transformer or Estimator abstract classes, one needs to implement the .getOutputCol() method that returns the value of the outputCol parameter specified when creating an object.

Predicting the chances of infant survival with ML

Loading the data

First, we load the data with the help of the following code:

```
import pyspark.sql.types as typ
labels = [
('INFANT_ALIVE_AT_REPORT', typ.IntegerType()),
('BIRTH_PLACE', typ.StringType()),
('MOTHER_AGE_YEARS', typ.IntegerType()),
('FATHER_COMBINED_AGE', typ.IntegerType()),
('CIG_BEFORE', typ.IntegerType()),
('CIG_1_TRI', typ.IntegerType()),
('CIG_2_TRI', typ.IntegerType()),
('CIG_3_TRI', typ.IntegerType()),
('MOTHER_HEIGHT_IN', typ.IntegerType()),
('MOTHER_PRE_WEIGHT', typ.IntegerType()),
('MOTHER_DELIVERY_WEIGHT', typ.IntegerType()),
('MOTHER_WEIGHT_GAIN', typ.IntegerType()),
('DIABETES_PRE', typ.IntegerType()),
('DIABETES_GEST', typ.IntegerType()),
('HYP_TENS_PRE', typ.IntegerType()),
('HYP_TENS_GEST', typ.IntegerType()),
('PREV_BIRTH_PRETERM', typ.IntegerType()) ]
```

Predicting the chances of infant survival with ML

Loading the data …

```
schema = typ.StructType([
typ.StructField(e[0], e[1], False) for e in labels
])
births = spark.read.csv('births_transformed.csv.gz', header=True,
schema=schema)
```

We specify the schema of the DataFrame; our severely limited dataset now only has 17 columns.

Note: http://www.tomdrabas.com/data/ LearningPySpark/births_transformed.csv.gz.

Creating transformers

Before we can use the dataset to estimate a model, we need to do some transformations. Since statistical models can only operate on numeric data, we will have to encode the BIRTH_PLACE variable.

Before we do any of this, since we will use a number of different feature transformations ,let's import them all:

```
import pyspark.ml.feature as ft
```

To encode the BIRTH_PLACE column, we will use the OneHotEncoder method. However, the method cannot accept StringType columns; it can only deal with numeric types so first we will cast the column to an IntegerType:

```
births = births.withColumn('BIRTH_PLACE_INT', births['BIRTH_PLACE'] \

.cast(typ.IntegerType()))
```

Having done this, we can now create our first Transformer:

Creating transformers

encoder = ft.OneHotEncoder(inputCol='BIRTH_PLACE_INT', outputCol='BIRTH_PLACE_VEC')

Let's now create a single column with all the features collated together. We will use the VectorAssembler method:

featuresCreator = ft.VectorAssembler(

inputCols=[col[0]

for col in labels[2:]] + [encoder.getOutputCol()], outputCol='features'

)

The inputCols parameter passed to the VectorAssembler object is a list of all the columns to be combined together to form the outputCol—the 'features'.

Note that we use the output of the encoder object (by calling the .getOutputCol() method), so we do not have to remember to change this parameter's value should we change the name of the output column in the encoder object at any point.

Creating an estimator

import pyspark.ml.classification as cl

Once loaded, let's create the model by using the following code:

logistic = cl.LogisticRegression(maxIter=10, regParam=0.01,

labelCol='INFANT_ALIVE_AT_REPORT')

We would not have to specify the labelCol parameter if our target column had the name 'label'.

Also, if the output of our featuresCreator was not called 'features', we would have to specify the featuresCol by (most conveniently) calling the getOutputCol() method on the featuresCreator object.

Creating a pipeline

First, let's load the Pipeline from the ML package:

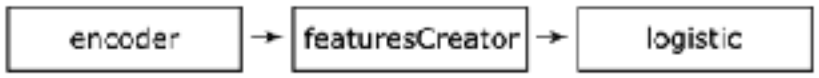
```
from pyspark.ml import Pipeline
```

Creating a Pipeline is really easy. Here's how our pipeline should look like conceptually:

Converting this structure into a Pipeline is a walk in the park:

```
pipeline = Pipeline(stages=[ encoder, featuresCreator, logistic ])
```

That's it! Our pipeline is now created so we can (finally!) estimate the model.



Fitting the model

Before you fit the model, we need to split our dataset into training and testing datasets. Conveniently, the DataFrame API has the .randomSplit(...) method:

births_train, births_test = births.randomSplit([0.7, 0.3], seed=666)

The first parameter is a list of dataset proportions that should end up in, respectively, births_train and births_test subsets.

The **seed** parameter provides a seed to the randomizer.

Note: You can also split the dataset into more than two subsets as long as the elements of the list sum up to 1, and you unpack the output into as many subsets.

For example, we could split the births dataset into three subsets like this:

train, test, val = births.randomSplit([0.7, 0.2, 0.1], seed=666)

The preceding code would put a random 70% of the births dataset into the train object, 20% would go to the test, and the val DataFrame would hold the remaining 10%.

Fitting the model

Now it is about time to finally run our pipeline and estimate our model:

```
model = pipeline.fit(births_train)
```

```
test_model = model.transform(births_test)
```

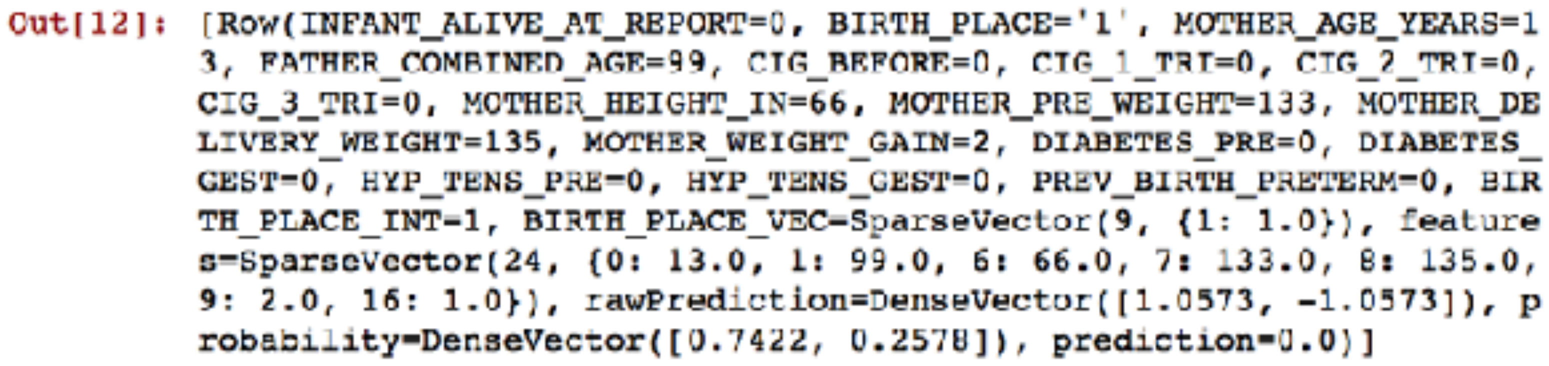
The .fit(...) method of the pipeline object takes our training dataset as an input. Under the hood, the births_train dataset is passed first to the encoder object. The DataFrame that is created at the encoder stage then gets passed to the featuresCreator that creates the 'features' column. Finally, the output from this stage is passed to the logistic object that estimates the final model.

Fitting the model

The .fit(...) method returns the PipelineModel object (the model object in the preceding snippet) that can then be used for prediction; we attain this by calling the .transform(...) method and passing the testing dataset created earlier. Here's what the test_model looks like in the following command:

test_model.take(1)

It generates the following output:



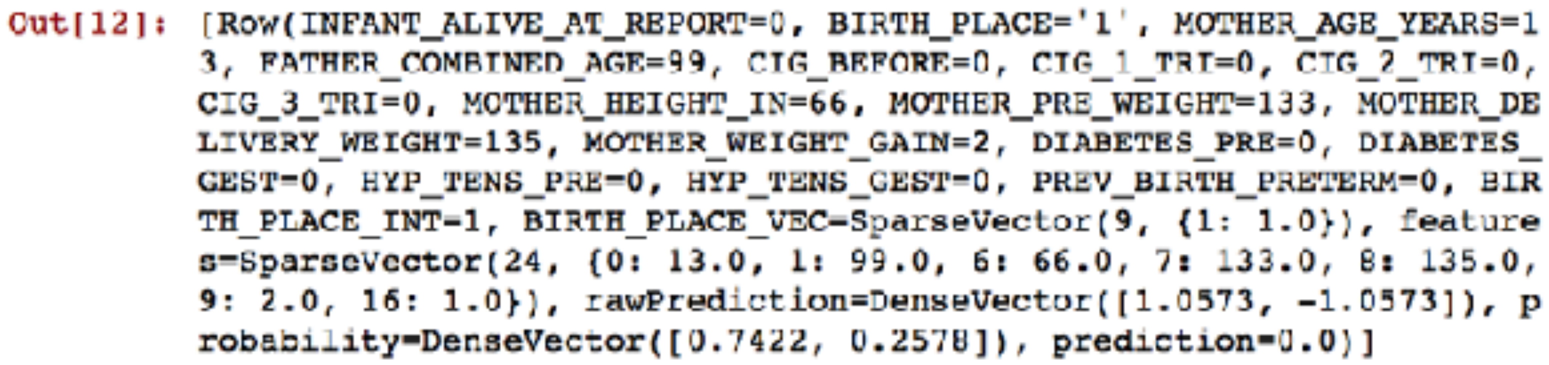
```
Out[12]: [Row(INFANT_ALIVE_AT_REPORT=0, BIRTH_PLACE='1', MOTHER_AGE_YEARS=1
         3, FATHER_COMBINED_AGE=99, CIG_BEFORE=0, CIG_1_TRI=0, CIG_2_TRI=0,
         CIG_3_TRI=0, MOTHER_HEIGHT_IN=66, MOTHER_PRE_WEIGHT=133, MOTHER_DE
         LIVERY_WEIGHT=135, MOTHER_WEIGHT_GAIN=2, DIABETES_PRE=0, DIABETES_
         GEST=0, HYP_TENS_PRE=0, HYP_TENS_GEST=0, PREV_BIRTH_PRETERM=0, BIR
         TH_PLACE_INT=1, BIRTH_PLACE_VEC=SparseVector(9, {1: 1.0}), feature
         s=SparseVector(24, {0: 13.0, 1: 99.0, 6: 66.0, 7: 133.0, 8: 135.0,
         9: 2.0, 16: 1.0}), rawPrediction=DenseVector([1.0573, -1.0573]), p
         robability=DenseVector([0.7422, 0.2578]), prediction=0.0)]
```

Fitting the model

The .fit(...) method returns the PipelineModel object (the model object in the preceding snippet) that can then be used for prediction; we attain this by calling the .transform(...) method and passing the testing dataset created earlier. Here's what the test_model looks like in the following command:

test_model.take(1)

It generates the following output:



```
Out[12]: [Row(INFANT_ALIVE_AT_REPORT=0, BIRTH_PLACE='1', MOTHER_AGE_YEARS=1
         3, FATHER_COMBINED_AGE=99, CIG_BEFORE=0, CIG_1_TRI=0, CIG_2_TRI=0,
         CIG_3_TRI=0, MOTHER_HEIGHT_IN=66, MOTHER_PRE_WEIGHT=133, MOTHER_DE
         LIVERY_WEIGHT=135, MOTHER_WEIGHT_GAIN=2, DIABETES_PRE=0, DIABETES_
         GEST=0, HYP_TENS_PRE=0, HYP_TENS_GEST=0, PREV_BIRTH_PRETERM=0, BIR
         TH_PLACE_INT=1, BIRTH_PLACE_VEC=SparseVector(9, {1: 1.0}), feature
         s=SparseVector(24, {0: 13.0, 1: 99.0, 6: 66.0, 7: 133.0, 8: 135.0,
         9: 2.0, 16: 1.0}), rawPrediction=DenseVector([1.0573, -1.0573]), p
         robability=DenseVector([0.7422, 0.2578]), prediction=0.0)]
```

Evaluating the performance of the model

Obviously, we would like to now test how well our model did.

PySpark exposes a number of evaluation methods for classification and regression in the .evaluation section of the package:

```
import pyspark.ml.evaluation as ev
```

We will use the BinaryClassficationEvaluator to test how well our model performed:

```
evaluator = ev.BinaryClassificationEvaluator(

rawPredictionCol='probability',

labelCol='INFANT_ALIVE_AT_REPORT')
```

The rawPredictionCol can either be the rawPrediction column produced by the estimator or the probability.

Evaluating the performance of the model

Let's see how well our model performed:

```
print(evaluator.evaluate(test_model,
{evaluator.metricName: 'areaUnderROC'}))
print(evaluator.evaluate(test_model,
{evaluator.metricName: 'areaUnderPR'}))
```

The preceding code produces the following result:

```
0.7401301847095617
0.713935434236567
```

The area under the ROC of 74% and area under PR of 71% shows a well-defined model, but nothing out of extraordinary; if we had other features, we could drive this up, but this is not the purpose of this chapter (nor the book, for that matter).

Saving the model

PySpark allows you to save the Pipeline definition for later use. It not only saves the pipeline structure, but also all the definitions of all the Transformers and Estimators:

pipelinePath = './infant_oneHotEncoder_Logistic_Pipeline'

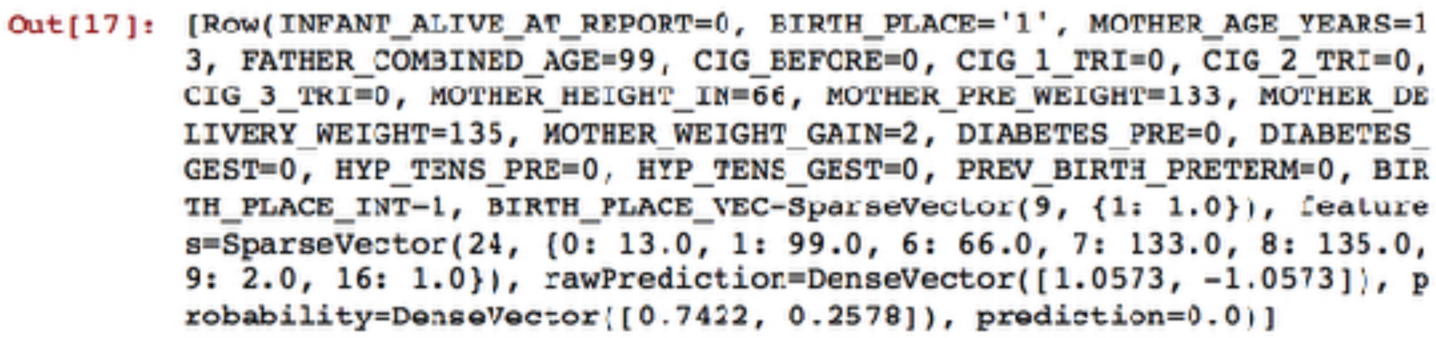
pipeline.write().overwrite().save(pipelinePath)

So, you can load it up later and use it straight away to .fit(...) and predict:

loadedPipeline = Pipeline.load(pipelinePath)

loadedPipeline.fit(births_train).transform(births_test).take(1)

The preceding code produces the same result (as expected):



```
Out[17]: [Row(INFANT_ALIVE_AT_REPORT=0, BIRTH_PLACE='1', MOTHER_AGE_YEARS=1
         3, FATHER_COMBINED_AGE=99, CIG_BEFORE=0, CIG_1_TRI=0, CIG_2_TRI=0,
         CIG_3_TRI=0, MOTHER_HEIGHT_IN=66, MOTHER_PRE_WEIGHT=133, MOTHER_DE
         LIVERY_WEIGHT=135, MOTHER_WEIGHT_GAIN=2, DIABETES_PRE=0, DIABETES_
         GEST=0, HYP_TENS_PRE=0, HYP_TENS_GEST=0, PREV_BIRTH_PRETERM=0, BIR
         TH_PLACE_INT=1, BIRTH_PLACE_VEC=SparseVector(9, {1: 1.0}), feature
         s=SparseVector(24, {0: 13.0, 1: 99.0, 6: 66.0, 7: 133.0, 8: 135.0,
         9: 2.0, 16: 1.0}), rawPrediction=DenseVector([1.0573, -1.0573]), p
         robability=DenseVector([0.7422, 0.2578]), prediction=0.0)]
```

Saving the model

To save your model, see the following the example:

```
from pyspark.ml import PipelineModel
modelPath = './infant_oneHotEncoder_Logistic_PipelineModel'
model.write().overwrite().save(modelPath)
loadedPipelineModel = PipelineModel.load(modelPath)
test_reloadedModel = loadedPipelineModel.transform(births_test)
```

The preceding script uses the .load(...) method, a class method of the PipelineModel class, to reload the estimated model.

You can compare the result of test_reloadedModel.take(1) with the output of test_model.take(1) we presented earlier.

Parameter hyper-tuning

Rarely, our first model would be the best we can do. By simply looking at our metrics and accepting the model because it passed our pre-conceived performance thresholds is hardly a scientific method for finding the best model.

A concept of parameter hyper-tuning is to find the best parameters of the model: for example, the maximum number of iterations needed to properly estimate the logistic regression model or maximum depth of a decision tree.

In this section, we will explore two concepts that allow us to find the best parameters for our models: grid search and train-validation splitting.

Grid search

Grid search is an exhaustive algorithm that loops through the list of defined parameter values, estimates separate models, and chooses the best one given some evaluation metric.

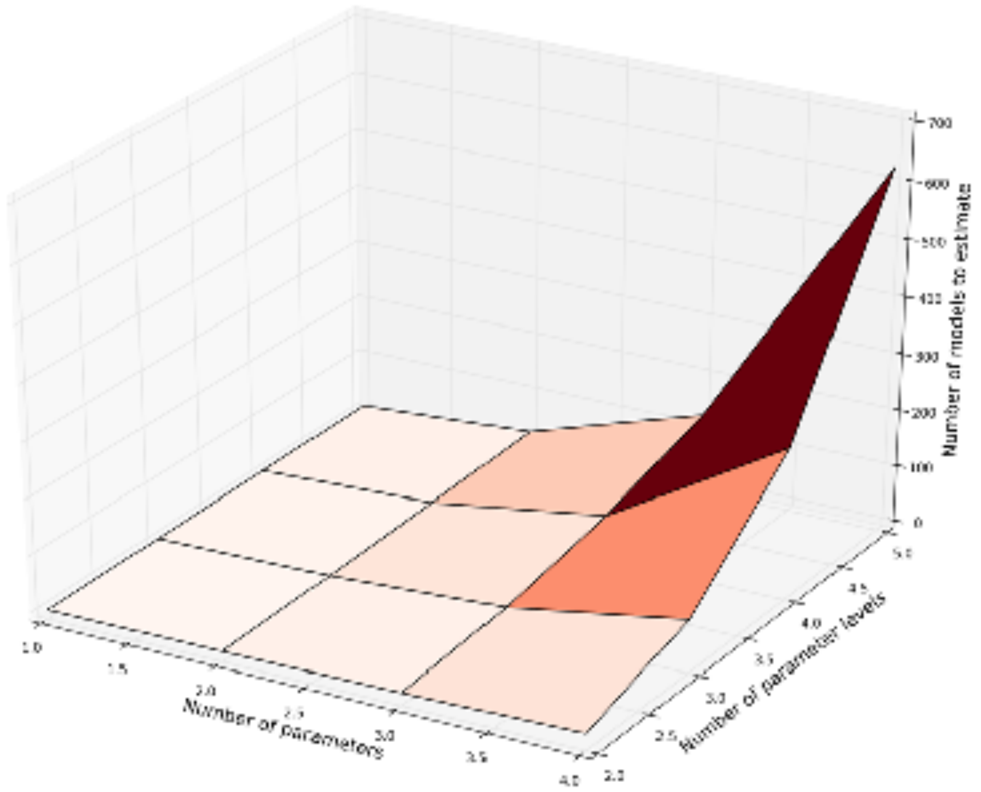
A note of caution should be stated here: if you define too many parameters you want to optimize over, or too many values of these parameters, it might take a lot of time to select the best model as the number of models to estimate would grow very quickly as the number of parameters and parameter values grow.

For example, if you want to fine-tune two parameters with two parameter values, you would have to fit four models. Adding one more parameter with two values would require estimating eight models, whereas adding one more additional value to our two parameters (bringing it to three values for each) would require estimating nine models.

As you can see, this can quickly get out of hand if you are not careful. See the following chart to inspect this visually:

Grid search

As you can see, this can quickly get out of hand if you are not careful. See the following chart to inspect this visually:



Grid search

Next, we need some way of comparing the models:

```
evaluator = ev.BinaryClassificationEvaluator(

rawPredictionCol='probability',

labelCol='INFANT_ALIVE_AT_REPORT')
```

So, once again, we'll use the BinaryClassificationEvaluator. It is time now to create the logic that will do the validation work for us:

```
cv = tune.CrossValidator( estimator=logistic, estimatorParamMaps=grid,

evaluator=evaluator

)
```

The CrossValidator needs the estimator, the estimatorParamMaps, and the evaluator to do its job. The model loops through the grid of values, estimates the models, and compares their performance using the evaluator.

Grid search

We cannot use the data straight away (as the births_train and births_test still have the BIRTHS_PLACE column not encoded) so we create a purely transforming Pipeline:

```
pipeline = Pipeline(stages=[encoder ,featuresCreator])

data_transformer = pipeline.fit(births_train)
```

Having done this, we are ready to find the optimal combination of parameters for our model:

```
cvModel = cv.fit(data_transformer.transform(births_train))
```

The cvModel will return the best model estimated. We can now use it to see if it performed better than our previous model:

```
data_train = data_transformer.transform(births_test)

results = cvModel.transform(data_train)

print(evaluator.evaluate(results,

{evaluator.metricName: 'areaUnderROC'}))

print(evaluator.evaluate(results,

{evaluator.metricName: 'areaUnderPR'}))
```

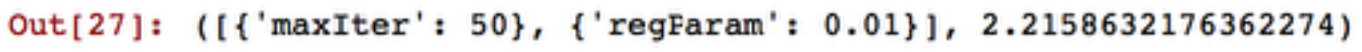
The preceding code will produce the following result:

```
0.7404304424804281
0.7156729757616691
```

Grid search

As you can see, we got a slightly better result. What parameters does the best model have? The answer is a little bit convoluted, but here's how you can extract it:

```
results = [ ( [ {key.name: paramValue}

for key, paramValue in zip(

params.keys(), params.values())

], metric )

for params, metric

in zip( cvModel.getEstimatorParamMaps(),

cvModel.avgMetrics

) ]

sorted(results,

key=lambda el: el[1],

reverse=True)[0]
```



```
Out[27]: ([{'maxIter': 50}, {'regParam': 0.01}], 2.2158632176362274)
```

Train-validation splitting

The TrainValidationSplit model, to select the best model, performs a random split of the input dataset (the training dataset) into two subsets: smaller training and validation subsets. The split is only performed once.

In this example, we will also use the ChiSqSelector to select only the top five features, thus limiting the complexity of our model:

```
selector = ft.ChiSqSelector( numTopFeatures=5,

featuresCol=featuresCreator.getOutputCol(),

outputCol='selectedFeatures',

labelCol='INFANT_ALIVE_AT_REPORT' )
```

The numTopFeatures specifies the number of features to return. We will put the selector after the featuresCreator, so we call the .getOutputCol() on the featuresCreator.

We covered creating the LogisticRegression and Pipeline earlier, so we will not explain how these are created again here:

```
logistic = cl.LogisticRegression( labelCol='INFANT_ALIVE_AT_REPORT', featuresCol='selectedFeatures' )

pipeline = Pipeline(stages=[encoder, featuresCreator, selector])

data_transformer = pipeline.fit(births_train)
```

Train-validation splitting

The TrainValidationSplit object gets created in the same fashion as the CrossValidator model:

tvs = tune.TrainValidationSplit(estimator=logistic, estimatorParamMaps=grid, evaluator=evaluator)

As before, we fit our data to the model, and calculate the results:

tvsModel = tvs.fit(data_transformer.transform(births_train))

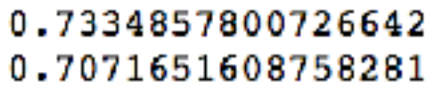
data_train = data_transformer.transform(births_test)

results = tvsModel.transform(data_train)

print(evaluator.evaluate(results, {evaluator.metricName: 'areaUnderROC'})) print(evaluator.evaluate(results,

{evaluator.metricName: 'areaUnderPR'}))

The preceding code prints out the following output:



```
0.7334857800726642
0.7071651608758281
```

Other features of PySpark ML

Feature extraction

NLP - related feature extractors

As described earlier, the NGram model takes a list of tokenized text and produces pairs (or n-grams) of words.

In this example, we will take an excerpt from PySpark's documentation and present how to clean up the text before passing it to the NGram model. Here's how our dataset looks like (abbreviated for brevity):

Note: Download the code from GitHub repository:

https://github. com/drabastomek/learningPySpark.
http://spark.apache.org/docs/latest/ ml-pipeline.html#dataframe.

Other features of PySpark ML

```
text_data = spark.createDataFrame([
['''Machine learning can be applied to a wide variety
of data types, such as vectors, text, images, and
structured data. This API adopts the DataFrame from
Spark SQL in order to support a variety of data
types.'''],
(...)
['''Columns in a DataFrame are named. The code examples
below use names such as "text," "features," and
"label."''']
], ['input'])
```

Other features of PySpark ML

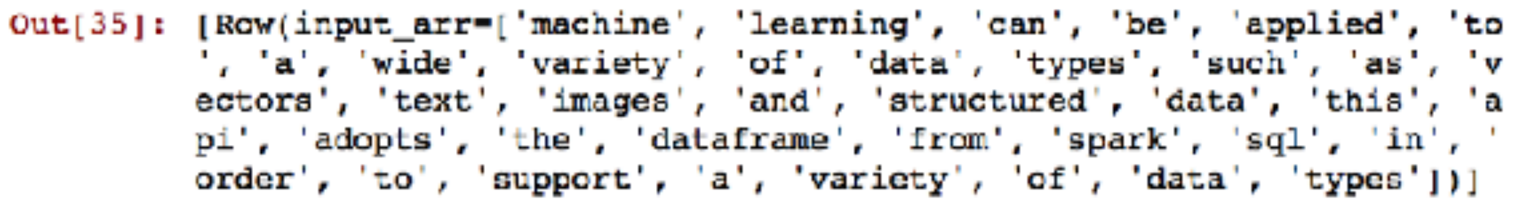
Each row in our single-column DataFrame is just a bunch of text.

First, we need to tokenize this text. To do so we will use the RegexTokenizer instead of just the Tokenizer as we can specify the pattern(s) we want the text to be broken at:

```
tokenizer = ft.RegexTokenizer(

inputCol='input',

outputCol='input_arr',

pattern='\s+|[,.\"]')
```

Other features of PySpark ML

The pattern here splits the text on any number of spaces, but also removes commas, full stops, backslashes, and quotation marks. A single row from the output of the tokenizer looks similar to this:



```
Out[35]: [Row(input_arr=['machine', 'learning', 'can', 'be', 'applied', 'to
         ', 'a', 'wide', 'variety', 'of', 'data', 'types', 'such', 'as', 'v
         ectors', 'text', 'images', 'and', 'structured', 'data', 'this', 'a
         pi', 'adopts', 'the', 'dataframe', 'from', 'spark', 'sql', 'in', '
         order', 'to', 'support', 'a', 'variety', 'of', 'data', 'types'])]
```

As you can see, the RegexTokenizer not only splits the sentences in to words, but also normalizes the text so each word is in small-caps.

Other features of PySpark ML

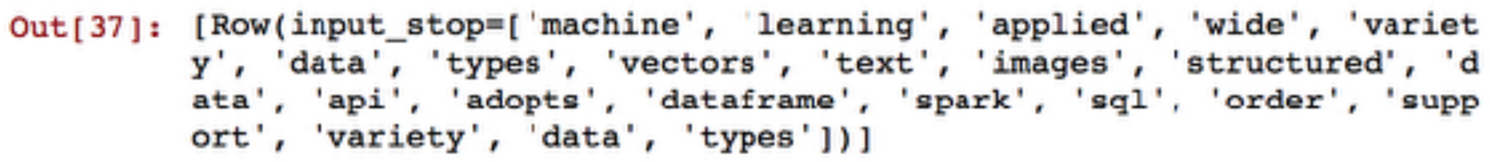
However, there is still plenty of junk in our text: words such as be, a, or to normally provide us with nothing useful when analyzing a text. Thus, we will remove these so called stopwords using nothing else other than the StopWordsRemover(...):

stopwords = ft.StopWordsRemover(

inputCol=tokenizer.getOutputCol(),

outputCol='input_stop')

The output of the method looks as follows:



```
Out[37]: [Row(input_stop=['machine', 'learning', 'applied', 'wide', 'variet
         y', 'data', 'types', 'vectors', 'text', 'images', 'structured', 'd
         ata', 'api', 'adopts', 'dataframe', 'spark', 'sql', 'order', 'supp
         ort', 'variety', 'data', 'types'])]
```

Other features of PySpark ML

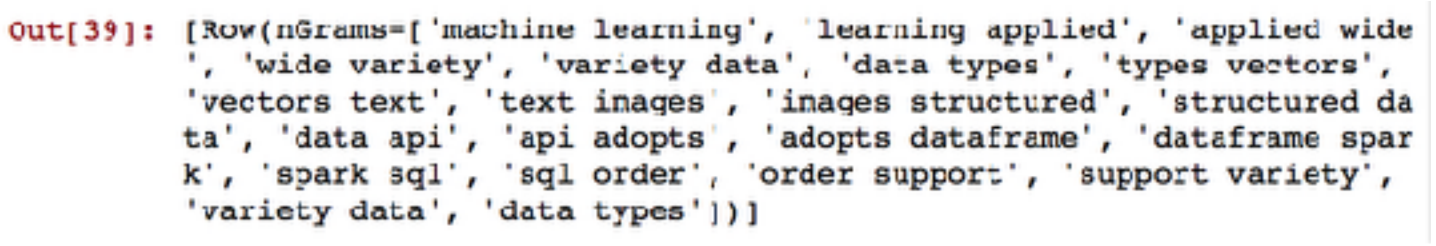
Now we only have the useful words. So, let's build our NGram model and the Pipeline:

```
ngram = ft.NGram(n=2,

inputCol=stopwords.getOutputCol(),

outputCol="nGrams")

pipeline = Pipeline(stages=[tokenizer, stopwords, ngram])
```

Now that we have the pipeline, we follow in a very similar fashion as before:

```
data_ngram = pipeline \

.fit(text_data) \

.transform(text_data)

data_ngram.select('nGrams').take(1)
```

The preceding code produces the following output:



```
Out[39]: [Row(nGrams=['machine learning', 'learning applied', 'applied wide
         ', 'wide variety', 'variety data', 'data types', 'types vectors',
         'vectors text', 'text images', 'images structured', 'structured da
         ta', 'data api', 'api adopts', 'adopts dataframe', 'dataframe spar
         k', 'spark sql', 'sql order', 'order support', 'support variety',
         'variety data', 'data types'])]
```

Discretizing continuous variables

Ever so often, we deal with a continuous feature that is highly non-linear and really hard to fit in our model with only one coefficient.

In such a situation, it might be hard to explain the relationship between such a feature and the target with just one coefficient. Sometimes, it is useful to band the values into discrete buckets.

First, let's create some fake data with the help of the following code:

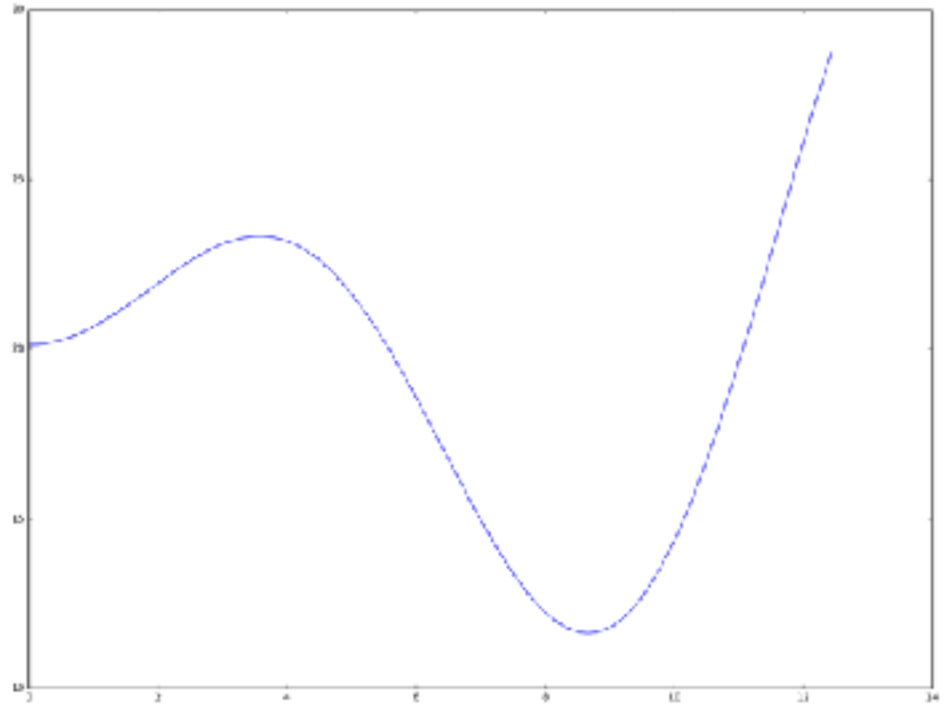
```
import numpy as np

x = np.arange(0, 100)

x = x / 100.0 * np.pi * 4

y = x * np.sin(x / 1.764) + 20.1234
```

Discretizing continuous variables

Now, we can create a DataFrame by using the following code:

```
schema = typ.StructType([
typ.StructField('continuous_var',
typ.DoubleType(),
False
)
])
data = spark.createDataFrame(
[[float(e), ] for e in y],
schema=schema)
```

Discretizing continuous variables



Discretizing continuous variables

Next, we will use the QuantileDiscretizer model to split our continuous variable into five buckets (the numBuckets parameter):

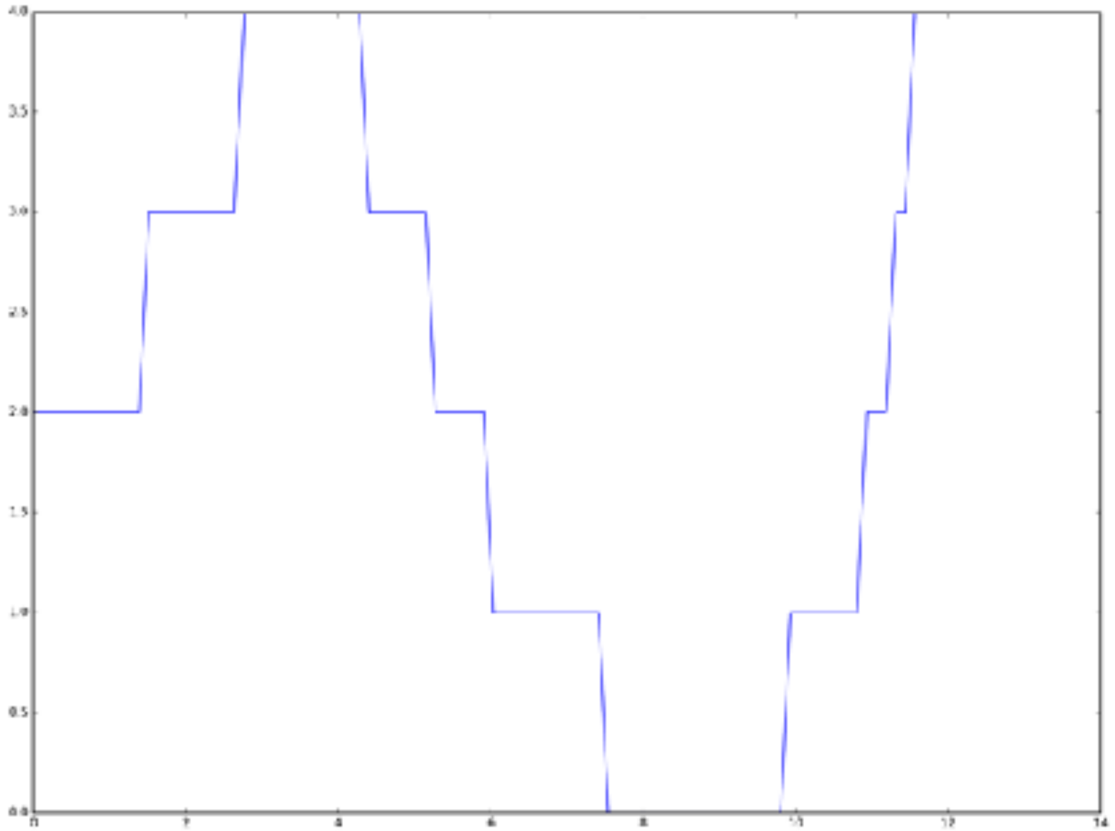
```
discretizer = ft.QuantileDiscretizer(
numBuckets=5,
inputCol='continuous_var',
outputCol='discretized')
```

Let's see what we have got:

```
data_discretized = discretizer.fit(data).transform(data)
```

Our function now looks as follows:

Discretizing continuous variables



Standardizing continuous variables

Standardizing continuous variables helps not only in better understanding the relationships between the features (as interpreting the coefficients becomes easier), but it also aids computational efficiency and protects from running into some numerical traps. Here's how you do it with PySpark ML.

First, we need to create a vector representation of our continuous variable (as it is only a single float):

```
vectorizer = ft.VectorAssembler(

inputCols=['continuous_var'],

outputCol= 'continuous_vec')
```

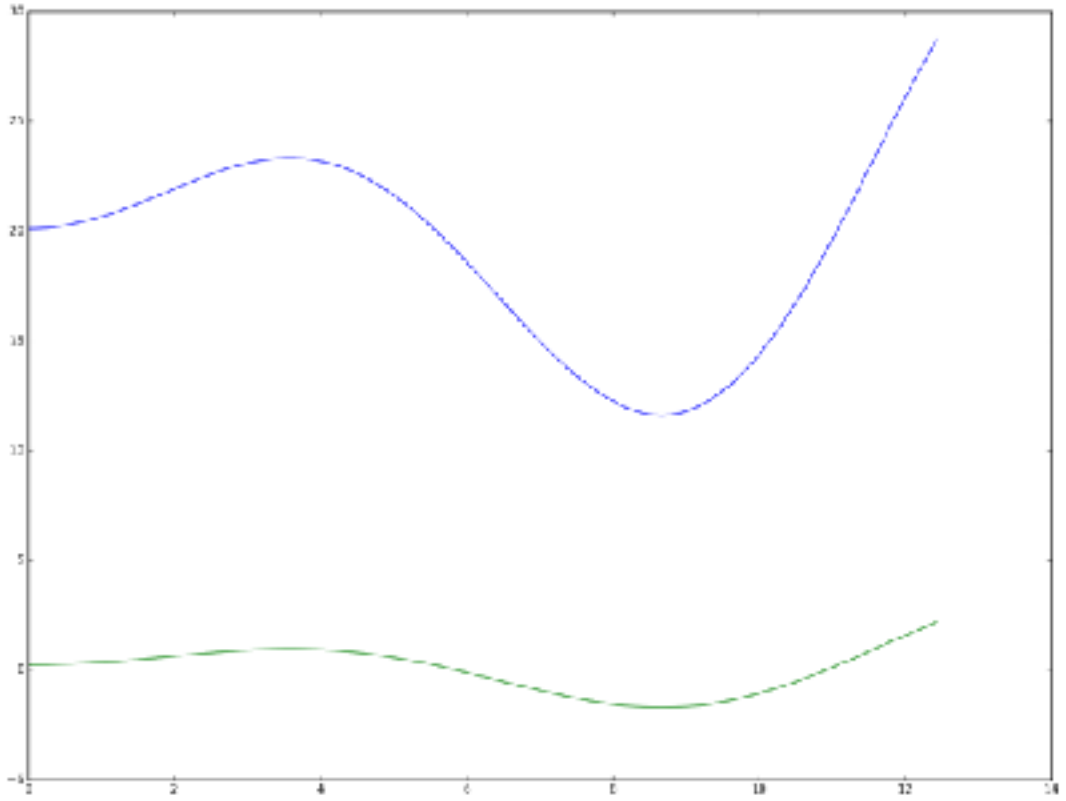
Standardizing continuous variables

Next, we build our normalizer and the pipeline. By setting the withMean and withStd to True, the method will remove the mean and scale the variance to be of unit length:

```
normalizer = ft.StandardScaler(

inputCol=vectorizer.getOutputCol(),

outputCol='normalized',

withMean=True,

withStd=True

)

pipeline = Pipeline(stages=[vectorizer, normalizer])

data_standardized = pipeline.fit(data).transform(data)
```

Standardizing continuous variables

Here's what the transformed data would look like:



Classification

So far we have only used the LogisticRegression model from PySpark ML.

In this section, we will use the RandomForestClassfier to, once again, model the chances of survival for an infant.

Before we can do that, though, we need to cast the label feature to DoubleType:

```
import pyspark.sql.functions as func

births = births.withColumn(

'INFANT_ALIVE_AT_REPORT',

func.col('INFANT_ALIVE_AT_REPORT').cast(typ.DoubleType())

)

births_train, births_test = births \

.randomSplit([0.7, 0.3], seed=666)
```

Classification

Now that we have the label converted to double, we are ready to build our model. We progress in a similar fashion as before with the distinction that we will reuse the encoder and featureCreator from earlier in the chapter. The numTrees parameter specifies how many decision trees should be in our random forest, and the maxDepth parameter limits the depth of the trees:

```
classifier = cl.RandomForestClassifier( numTrees=5, maxDepth=5,

labelCol='INFANT_ALIVE_AT_REPORT')

pipeline = Pipeline( stages=[ encoder, featuresCreator, classifier])

model = pipeline.fit(births_train)

test = model.transform(births_test)
```

Classification

Let's now see how the RandomForestClassifier model performs compared to the LogisticRegression:

```
evaluator = ev.BinaryClassificationEvaluator(
labelCol='INFANT_ALIVE_AT_REPORT')
print(evaluator.evaluate(test,
{evaluator.metricName: "areaUnderROC"}))
print(evaluator.evaluate(test,
{evaluator.metricName: “areaUnderPR"}))
```

We get the following results:

```
0.7736428008521183
0.7415879154340478
```

Classification

Well, as you can see, the results are better than the logistic regression model by roughly 3 percentage points. Let's test how well would a model with one tree do:

```
classifier = cl.DecisionTreeClassifier(
maxDepth=5,
labelCol='INFANT_ALIVE_AT_REPORT')
pipeline = Pipeline(stages=[ encoder, featuresCreator, classifier])
model = pipeline.fit(births_train)
test = model.transform(births_test)
evaluator = ev.BinaryClassificationEvaluator( labelCol='INFANT_ALIVE_AT_REPORT')
print(evaluator.evaluate(test,
{evaluator.metricName: "areaUnderROC"}))
print(evaluator.evaluate(test,
{evaluator.metricName: "areaUnderPR"}))
```

Clustering

Clustering is another big part of machine learning: quite often, in the real world, we do not have the luxury of having the target feature, so we need to revert to an unsupervised learning paradigm, where we try to uncover patterns in the data.

Finding clusters in the births dataset

In this example, we will use the k-means model to find similarities in the births data:

```
import pyspark.ml.clustering as clus

kmeans = clus.KMeans(k = 5, featuresCol='features')

pipeline = Pipeline(stages=[ assembler, featuresCreator, kmeans]

)

model = pipeline.fit(births_train)
```

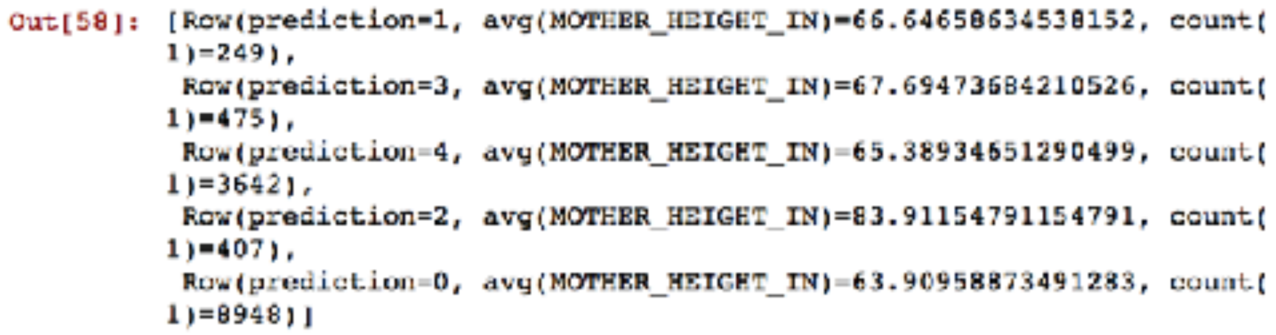
Clustering

Having estimated the model, let's see if we can find some differences between clusters:

test = model.transform(births_test)

test.groupBy('prediction').agg({ '*': 'count', 'MOTHER_HEIGHT_IN': 'avg' }).collect()

The preceding code produces the following output:



```
Out[58]: [Row(prediction=1, avg(MOTHER_HEIGHT_IN)=66.64658634538152, count(
         1)=249),
          Row(prediction=3, avg(MOTHER_HEIGHT_IN)=67.69473684210526, count(
         1)=475),
          Row(prediction=4, avg(MOTHER_HEIGHT_IN)=65.38934651290499, count(
         1)=3642),
          Row(prediction=2, avg(MOTHER_HEIGHT_IN)=83.91154791154791, count(
         1)=407),
          Row(prediction=0, avg(MOTHER_HEIGHT_IN)=63.90958873491283, count(
         1)=8948)]
```

Well, the MOTHER_HEIGHT_IN is significantly different in cluster 2. Going through the results (which we will not do here for obvious reasons) would most likely uncover more differences and allow us to understand the data better.

Topic mining

Clustering models are not limited to numeric data only. In the field of NLP, problems such as topic extraction rely on clustering to detect documents with similar topics. We will go through such an example.

First, let's create our dataset. The data is formed from randomly selected paragraphs found on the Internet: three of them deal with topics of nature and national parks, the remaining three cover technology.

```
text_data = spark.createDataFrame([
['''To make a computer do anything, you have to write a
computer program. To write a computer program, you have
to tell the computer, step by step, exactly what you want
it to do. The computer then "executes" the program,
following each step mechanically, to accomplish the end
goal. When you are telling the computer what to do, you
also get to choose how it's going to do it. That's where
computer algorithms come in. The algorithm is the basic
technique used to get the job done. Let's follow an
example to help get an understanding of the algorithm
concept.'''],
```

Topic mining

```
(...),
['"Australia has over 500 national parks. Over 28
million hectares of land is designated as national
parkland, accounting for almost four per cent of
Australia's land areas. In addition, a further six per
cent of Australia is protected and includes state
forests, nature parks and conservation reserves.National
parks are usually large areas of land that are protected
because they have unspoilt landscapes and a diverse
number of native plants and animals. This means that
commercial activities such as farming are prohibited and
human activity is strictly monitored."']
], ['documents'])
```

Topic mining

First, we will once again use the RegexTokenizer and the StopWordsRemover models:

```
tokenizer = ft.RegexTokenizer(

inputCol='documents',

outputCol='input_arr',

pattern='\s+|[,.\"]')

stopwords = ft.StopWordsRemover(

inputCol=tokenizer.getOutputCol(),

outputCol='input_stop')
```

Topic mining

Next in our pipeline is the CountVectorizer: a model that counts words in a document and returns a vector of counts. The length of the vector is equal to the total number of distinct words in all the documents, which can be seen in the following snippet:

```
stringIndexer = ft.CountVectorizer(

inputCol=stopwords.getOutputCol(),

outputCol="input_indexed")

tokenized = stopwords \

.transform( tokenizer.transform(text_data)

)
```

Topic mining

As you can see, there are 262 distinct words in the text, and each document is now represented by a count of each word occurrence.

It's now time to start predicting the topics. For that purpose we will use the LDA model—the Latent Dirichlet Allocation model:

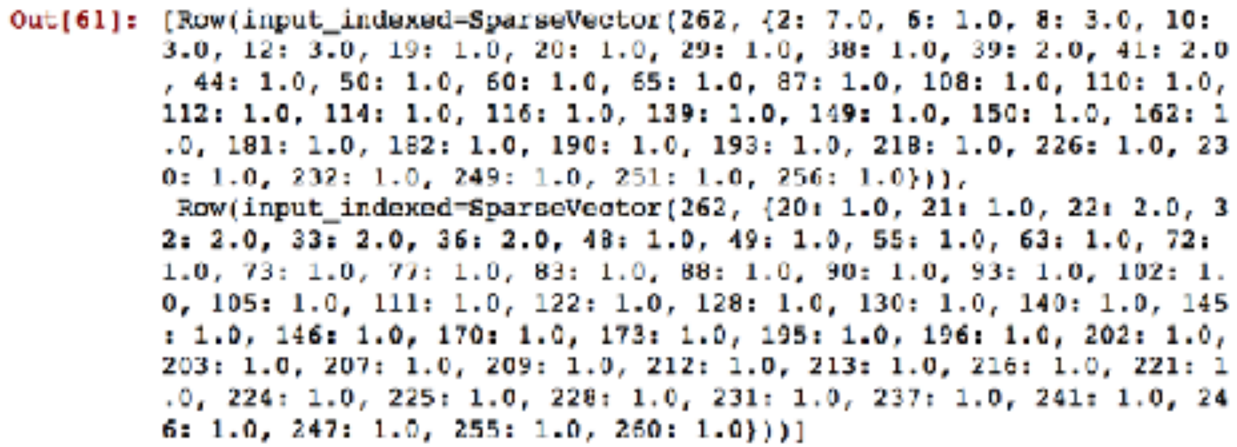
```
clustering = clus.LDA(k=2,

optimizer='online',

featuresCol=stringIndexer.getOutputCol())
```

The k parameter specifies how many topics we expect to see, the optimizer parameter can be either 'online' or 'em' (the latter standing for the Expectation Maximization algorithm).

Topic mining

```
stringIndexer.fit(tokenized).transform(tokenized).select('input_indexed').take(2)
```

The preceding code will produce the following output:



```
Out[61]: [Row(input_indexed=SparseVector(262, {2: 7.0, 6: 1.0, 8: 3.0, 10:
         3.0, 12: 3.0, 19: 1.0, 20: 1.0, 29: 1.0, 38: 1.0, 39: 2.0, 41: 2.0
         , 44: 1.0, 50: 1.0, 60: 1.0, 65: 1.0, 87: 1.0, 108: 1.0, 110: 1.0,
         112: 1.0, 114: 1.0, 116: 1.0, 139: 1.0, 149: 1.0, 150: 1.0, 162: 1
         .0, 181: 1.0, 182: 1.0, 190: 1.0, 193: 1.0, 218: 1.0, 226: 1.0, 23
         0: 1.0, 232: 1.0, 249: 1.0, 251: 1.0, 256: 1.0})),
          Row(input_indexed=SparseVector(262, {20: 1.0, 21: 1.0, 22: 2.0, 3
         2: 2.0, 33: 2.0, 36: 2.0, 48: 1.0, 49: 1.0, 55: 1.0, 63: 1.0, 72:
         1.0, 73: 1.0, 77: 1.0, 83: 1.0, 88: 1.0, 90: 1.0, 93: 1.0, 102: 1.
         0, 105: 1.0, 111: 1.0, 122: 1.0, 128: 1.0, 130: 1.0, 140: 1.0, 145
         : 1.0, 146: 1.0, 170: 1.0, 173: 1.0, 195: 1.0, 196: 1.0, 202: 1.0,
         203: 1.0, 207: 1.0, 209: 1.0, 212: 1.0, 213: 1.0, 216: 1.0, 221: 1
         .0, 224: 1.0, 225: 1.0, 228: 1.0, 231: 1.0, 237: 1.0, 241: 1.0, 24
         6: 1.0, 247: 1.0, 255: 1.0, 260: 1.0}))]
```

Topic mining

As you can see, there are 262 distinct words in the text, and each document is now represented by a count of each word occurrence.

It's now time to start predicting the topics. For that purpose we will use the LDA model—the Latent Dirichlet Allocation model:

```
clustering = clus.LDA(k=2, optimizer='online',

featuresCol=stringIndexer.getOutputCol())
```

The k parameter specifies how many topics we expect to see, the optimizer parameter can be either 'online' or 'em' (the latter standing for the Expectation Maximization algorithm).

Topic mining

Putting these puzzles together results in, so far, the longest of our pipelines:

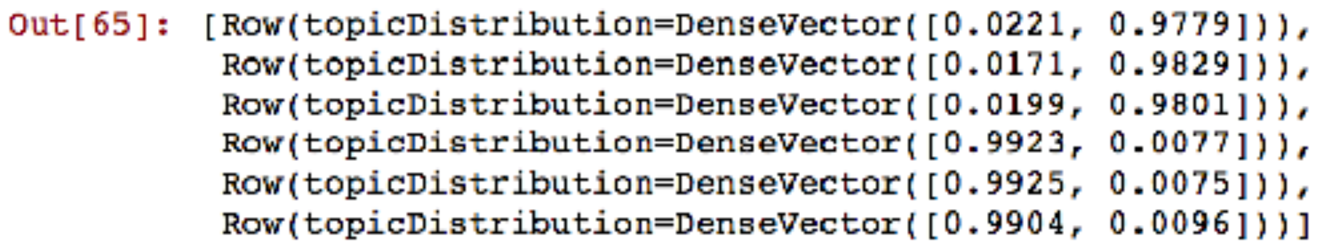
pipeline = ml.Pipeline(stages=[tokenizer, stopwords, stringIndexer, clustering])

Have we properly uncovered the topics? Well, let's see:

topics = pipeline.fit(text_data) .transform(text_data)

topics.select('topicDistribution').collect()

Here's what we get:



```
Out[65]: [Row(topicDistribution=DenseVector([0.0221, 0.9779])),
          Row(topicDistribution=DenseVector([0.0171, 0.9829])),
          Row(topicDistribution=DenseVector([0.0199, 0.9801])),
          Row(topicDistribution=DenseVector([0.9923, 0.0077])),
          Row(topicDistribution=DenseVector([0.9925, 0.0075])),
          Row(topicDistribution=DenseVector([0.9904, 0.0096]))]
```

Regression

We will try to predict the MOTHER_WEIGHT_GAIN given some of the features described here; these are contained in the features listed here:

```
features = ['MOTHER_AGE_YEARS','MOTHER_HEIGHT_IN',

'MOTHER_PRE_WEIGHT','DIABETES_PRE',

'DIABETES_GEST','HYP_TENS_PRE',

'HYP_TENS_GEST', 'PREV_BIRTH_PRETERM',

'CIG_BEFORE','CIG_1_TRI', 'CIG_2_TRI',

'CIG_3_TRI'

]
```

Regression

First, since all the features are numeric, we will collate them together and use the ChiSqSelector to select only the top six most important features:

```
featuresCreator = ft.VectorAssembler(
inputCols=[col for col in features[1:]],
outputCol='features'
)
selector = ft.ChiSqSelector(
numTopFeatures=6, outputCol="selectedFeatures",
labelCol='MOTHER_WEIGHT_GAIN'
)
```

Regression

In order to predict the weight gain, we will use the gradient boosted trees regressor:

```
import pyspark.ml.regression as reg
regressor = reg.GBTRegressor(
maxIter=15,
maxDepth=3,
labelCol='MOTHER_WEIGHT_GAIN')
```

Finally, again, we put it all together into a Pipeline:

```
pipeline = Pipeline(stages=[ featuresCreator, selector, regressor])
weightGain = pipeline.fit(births_train)
```

Regression

Having created the weightGain model, let's see if it performs well on our testing data:

```
evaluator = ev.RegressionEvaluator(
predictionCol="prediction",
labelCol='MOTHER_WEIGHT_GAIN')
print(evaluator.evaluate(
weightGain.transform(births_test),
{evaluator.metricName: 'r2'}))
```

We get the following output:

```
0.48862170400240335
```

Sadly, the model is no better than a flip of a coin. It looks that without additional independent features that are better correlated with the MOTHER_WEIGHT_GAIN label, we will not be able to explain its variance sufficiently.

Summary

- In This Module, We went into details of how to use PySpark ML: the official main machine learning library for PySpark.
- We explained what the Transformer and Estimator are, and showed their role in another concept introduced in the ML library: the Pipeline. Subsequently,
- we also presented how to use some of the methods to fine-tune the hyper parameters of models.
- Finally, we gave some examples of how to use some of the feature extractors and models from the library.
- In the next Module, we will delve into graph theory and GraphFrames that help in tackling machine learning problems better represented as graphs.

Module 7

GraphFrames

Agenda

Day 3
Module 7
GraphFrames
Introducing GraphFrames
Installing GraphFrames
Preparing your flights dataset
Building the graph
Executing simple queries
Understanding vertex degrees
Determining the top transfer airports
Understanding motifs
Determining airport ranking using PageRank
Determining the most popular non-stop flights
Using Breadth-First Search
Visualizing flights using D3
Assignment 6
Conclusion and Summary

Agenda

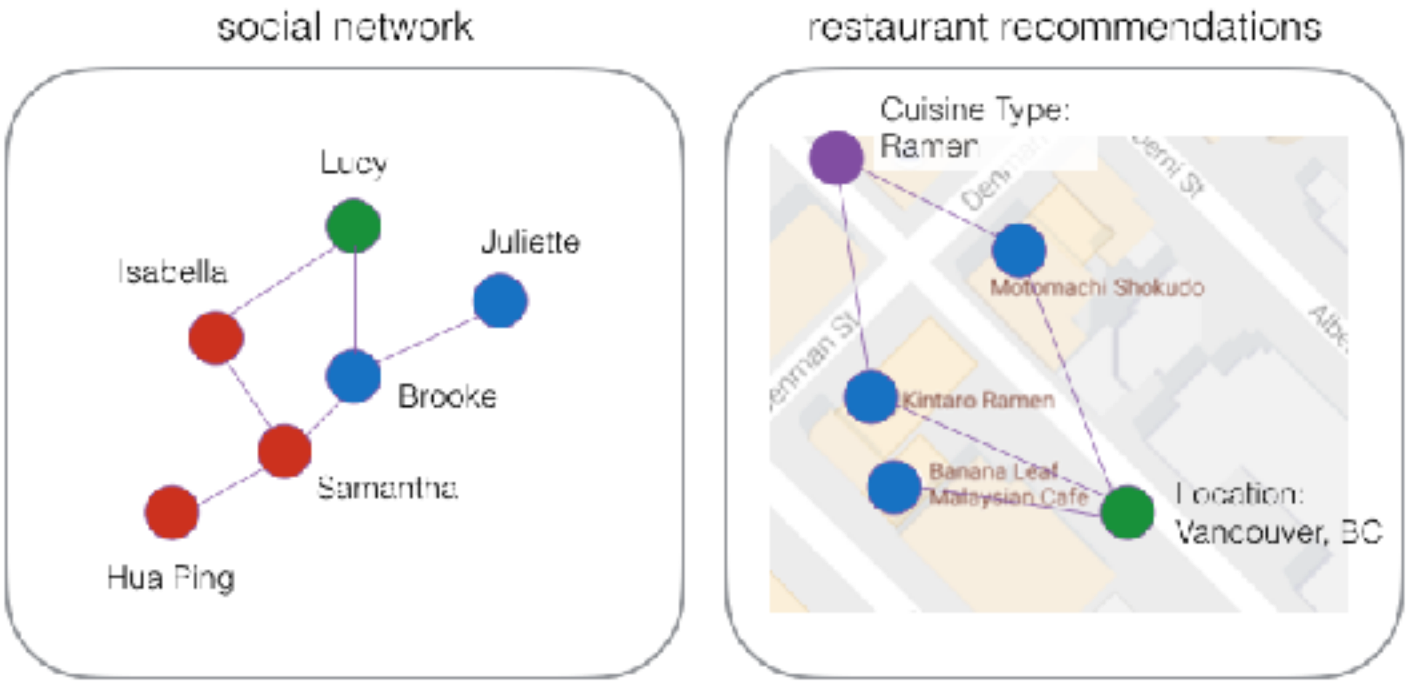
Objectives

You will learn how to do the following:

- Why use graphs?
- Understanding the classic graph problem: the flights dataset
- Understanding the graph vertices and edges
- Simple queries
- Using motif finding
- Using breadth first search
- Using PageRank
- Visualizing flights using D3

GraphFrames

Whether traversing social networks or restaurant recommendations, it is easier to understand these data problems within the context of graph structures: vertices, edges, and properties:

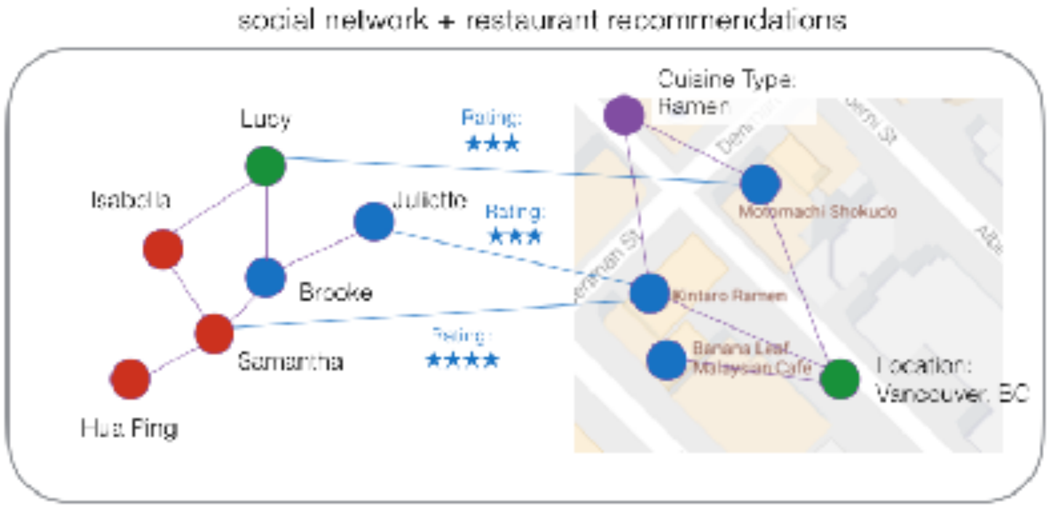


GraphFrames

For example, within the context of social networks, the vertices are the people while the edges are the connections between them.

Within the context of restaurant recommendations, the vertices (for example) involve the location, cuisine type, and restaurants while the edges are the connections between them (for example, these three restaurants are in Vancouver, BC, but only two of them serve ramen).

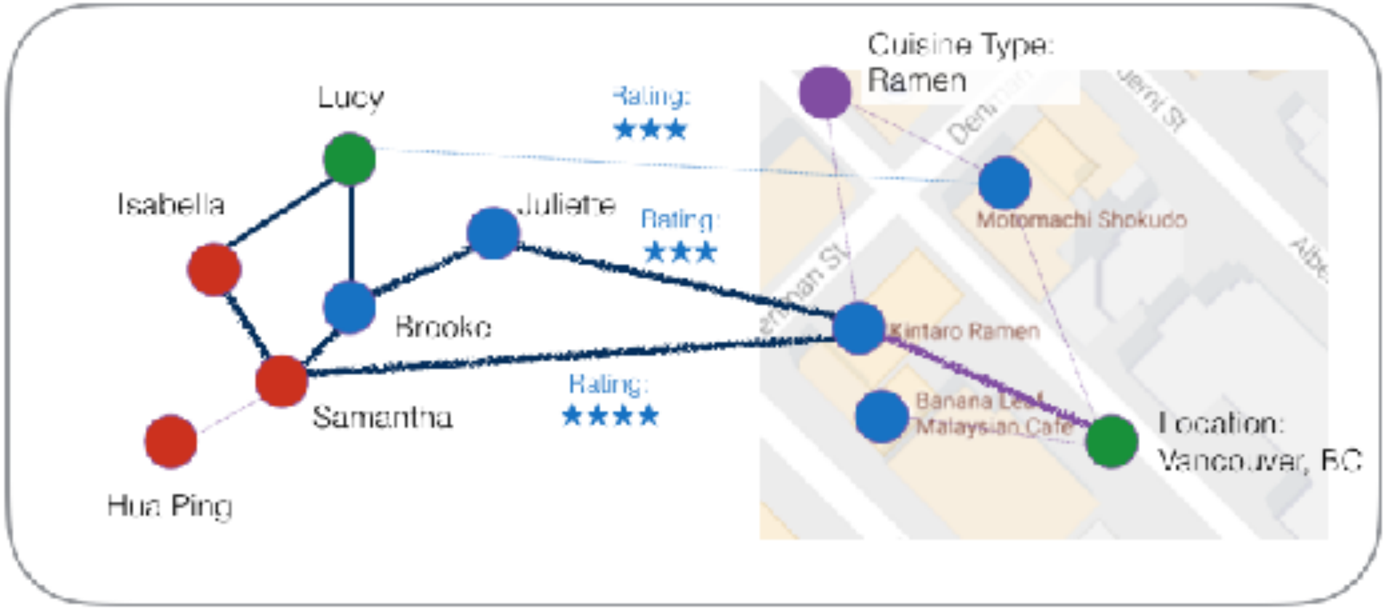
While the two graphs are seemingly disconnected, you can in fact create a social network + restaurant recommendation graph based on the reviews of friends within a social circle, as noted in the following figure:



GraphFrames

For example, if **Isabella** wants to find a great ramen restaurant in Vancouver, traversing her friends' reviews, she will most likely choose **Kintaro Ramen**, as both **Samantha** and **Juliette** have rated the restaurant favorably:

social network + restaurant recommendations

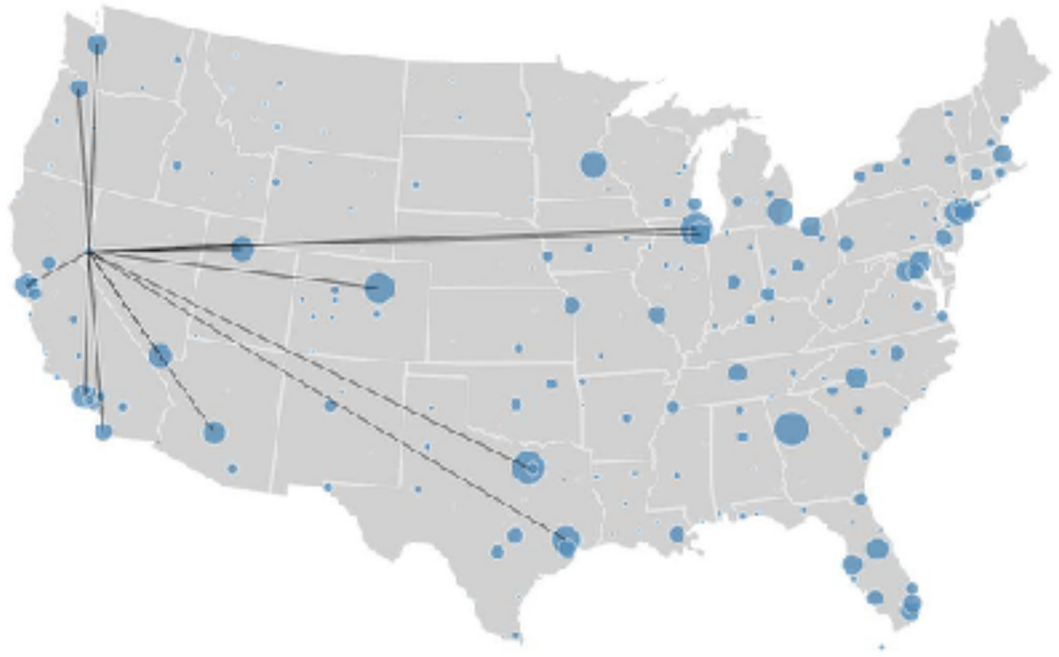


GraphFrames

Another classic graph problem is the analysis of flight data:

Airports are represented by vertices and flights between those airports are represented by edges.

Also, there are numerous properties associated with these flights, including, but not limited to, departure delays, plane type, and carrier:



GraphFrames

In this module, we will use GraphFrames to quickly and easily analyze flight performance data organized in graph structures.

Because we're using graph structures, we can easily ask many questions that are not as intuitive as tabular structures, such as finding structural motifs, airport ranking using PageRank, and shortest paths between cities.

GraphFrames leverages the distribution and expression capabilities of the DataFrame API to both simplify your queries and leverage the performance optimizations of the Apache Spark SQL engine.

In addition, with GraphFrames, graph analysis is available in Python, Scala, and Java.

Just as important, you can leverage your existing Apache Spark skills to solve graph problems (in addition to machine learning, streaming, and SQL) instead of making a paradigm shift to learn a new framework.

Introducing GraphFrames

GraphFrames utilizes the power of Apache Spark DataFrames to support general graph processing.

Specifically, the vertices and edges are represented by DataFrames allowing us to store arbitrary data with each vertex and edge.

While GraphFrames is similar to Spark's GraphX library, there are some key differences, including:

- GraphFrames leverage the performance optimizations and simplicity of the DataFrame API.
- By using the DataFrame API, GraphFrames now have Python, Java, and Scala APIs. GraphX is only accessible through Scala; now all its algorithms are available in Python and Java.
- Note, at the time of writing, there was a bug preventing GraphFrames from working with Python3.x, hence we will be using Python2.x.

Introducing GraphFrames

At the time of writing, GraphFrames is on version 0.3 and available as a Spark package (http://spark-packages.org) at

https://spark-packages.org/package/graphframes/graphframes.

For more information about GraphFrames, please refer to Introducing GraphFrames at

https://databricks.com/blog/2016/03/03/ introducing-graphframes.html.

Installing GraphFrames

If you are running your job from a Spark CLI (for example, spark-shell, pyspark, spark-sql, spark-submit), you can use the –-packages command, which will extract, compile, and execute the necessary code for you to use the GraphFrames package.

For example, to use the latest GraphFrames package (version 0.3) with Spark 2.0 and Scala 2.11 with spark-shell, the command is:

```
> $SPARK_HOME/bin/spark-shell --packages graphframes:graphframes:0.3.0- spark2.0-s_2.11
```

If you are using a notebook service, you may need to install the package first.

For example, the following section shows the steps to install the GraphFrames library within the free Databricks Community Edition (http://databricks.com/try-databricks).

Preparing your flights dataset

For this flights sample scenario, we will make use of two sets of data:

Airline On-Time Performance and Causes of Flight Delays: [http://bit. ly/2ccJPPM] This dataset contains scheduled and actual departure and arrival times, and delay causes as reported by US air carriers. The data is collected by the Office of Airline Information, Bureau of Transportation Statistics (BTS).

Open Flights: Airports and airline data: [http://openflights.org/data. html] This dataset contains the list of US airport data including the IATA code, airport name, and airport location.

Preparing your flights dataset

We will create two DataFrames – airports and departureDelays–which will make up our vertices and edges of our GraphFrame, respectively.

We will be creating this flights sample application using Python.

As we are using a Databricks notebook for our example, we can make use of the / databricks-datasets/location, which contains numerous sample datasets.

You can also download the data from:

departureDelays.csv: http://bit.ly/2ejPr8k

airportCodes: http://bit.ly/2ePAdKT

Preparing your flights dataset

We are creating two variables denoting the file paths for our Airports and Departure Delays data, respectively.

Then we will load these datasets and create the respective Spark DataFrames; note for both of these files, we can easily infer the schema:

```
# Set File Paths

tripdelaysFilePath = "/databricks-datasets/flights/departuredelays. csv"

airportsnaFilePath = "/databricks-datasets/flights/airport-codes-na. txt"

# Obtain airports dataset

# Note, this dataset is tab-delimited with a header

airportsna = spark.read.csv(airportsnaFilePath, header='true', inferSchema='true', sep='\t')

airportsna.createOrReplaceTempView("airports_na")
```

Preparing your flights dataset

```
# Obtain departure Delays data

# Note, this dataset is comma-delimited with a header

departureDelays = spark.read.csv(tripdelaysFilePath, header='true')

departureDelays.createOrReplaceTempView("departureDelays")

departureDelays.cache()
```

Once we loaded the departureDelays DataFrame, we also cache it so we can include some additional filtering of the data in a performant manner:

```
# Available IATA codes from the departuredelays sample dataset

tripIATA = spark.sql("select distinct iata from (select distinct origin as iata from departureDelays union all select distinct destination as iata from departureDelays) a")

tripIATA.createOrReplaceTempView("tripIATA")
```

Preparing your flights dataset

The preceding query allows us to build a distinct list with origin city IATA codes (for example, Seattle = 'SEA', San Francisco = 'SFO', New York JFK = 'JFK', and so on). Next, we only include airports that had a trip occur within the departureDelays DataFrame:

```
# Only include airports with atleast one trip from the

# `departureDelays` dataset

airports = spark.sql("select f.IATA, f.City, f.State, f.Country from airports_na f join tripIATA t on t.IATA = f.IATA")

airports.createOrReplaceTempView("airports")

airports.cache()
```

Preparing your flights dataset

Once we loaded the departureDelays DataFrame, we also cache it so we can include some additional filtering of the data in a performant manner:

```
# Available IATA codes from the departuredelays sample dataset

tripIATA = spark.sql("select distinct iata from (select distinct origin as iata from departureDelays union all select distinct destination as iata from departureDelays) a")

tripIATA.createOrReplaceTempView("tripIATA")
```

Preparing your flights dataset

The preceding query allows us to build a distinct list with origin city IATA codes (for example, Seattle = 'SEA', San Francisco = 'SFO', New York JFK = 'JFK', and so on). Next, we only include airports that had a trip occur within the departureDelays DataFrame:

```
# Only include airports with atleast one trip from the

# `departureDelays` dataset

airports = spark.sql("select f.IATA, f.City, f.State, f.Country from airports_na f join tripIATA t on t.IATA = f.IATA")

airports.createOrReplaceTempView("airports")

airports.cache()
```

Preparing your flights dataset

By building the distinct list of origin airport codes, we can build the airports DataFrame to contain only the airport codes that exist in the departureDelays dataset. The following code snippet generates a new DataFrame (departureDelays_geo) that is comprised of key attributes including date of flight, delays, distance, and airport information (origin, destination):

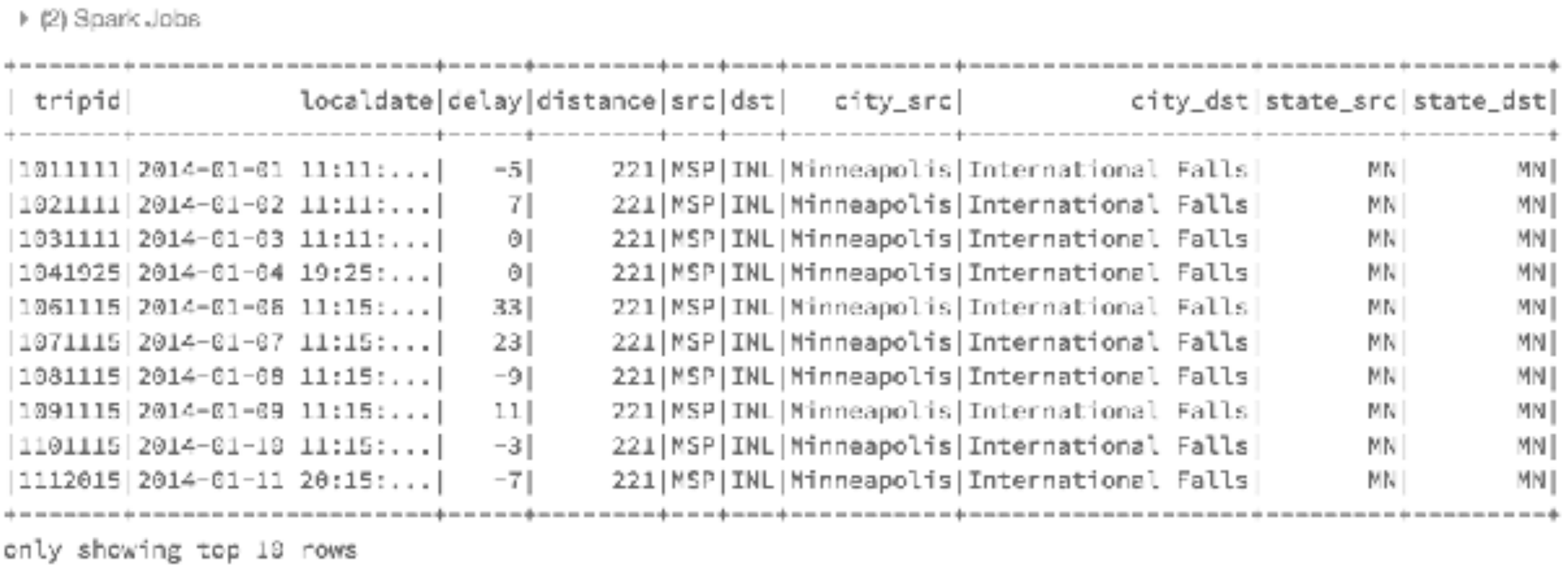
```
# Build `departureDelays_geo` DataFrame

# Obtain key attributes such as Date of flight, delays, distance,

# and airport information (Origin, Destination)

departureDelays_geo = spark.sql("select cast(f.date as int) as tripid,
cast(concat(concat(concat(concat(concat(concat('2014-', concat(concat(substr(cast(f.date as string), 1, 2), '-')),
substr(cast(f.date as string), 3, 2)), ''), substr(cast(f.date as string), 5, 2)), ':'), substr(cast(f.date as string), 7, 2)),
':00') as timestamp) as `localdate`, cast(f.delay as int), cast(f.distance as int), f.origin as src, f.destination as dst,
o.city as city_src, d.city as city_dst, o.state as state_src, d.state as state_dst from departuredelays f join
airports o on o.iata = f.origin join airports d on d.iata = f.destination")

# Create Temporary View and cache

departureDelays_geo.createOrReplaceTempView("departureDelays_geo")

departureDelays_geo.cache()
```

Preparing your flights dataset

To take a quick peek into this data, you can run the show method as shown here:

# Review the top 10 rows of the `departureDelays_geo` DataFrame

departureDelays_geo.show(10)



```
▸ (2) Spark Jobs

+-------+--------------------+-----+--------+---+---+-----------+--------------------+---------+---------+
| tripid|           localdate|delay|distance|src|dst|   city_src|            city_dst|state_src|state_dst|
+-------+--------------------+-----+--------+---+---+-----------+--------------------+---------+---------+
|1011111|2014-01-01 11:11:...|   -5|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1021111|2014-01-02 11:11:...|    7|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1031111|2014-01-03 11:11:...|    0|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1041925|2014-01-04 19:25:...|    0|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1061115|2014-01-06 11:15:...|   33|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1071115|2014-01-07 11:15:...|   23|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1081115|2014-01-08 11:15:...|   -9|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1091115|2014-01-09 11:15:...|   11|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1101115|2014-01-10 11:15:...|   -3|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
|1112015|2014-01-11 20:15:...|   -7|     221|MSP|INL|Minneapolis|International Falls|       MN|       MN|
+-------+--------------------+-----+--------+---+---+-----------+--------------------+---------+---------+
only showing top 10 rows
```

Building the graph

Now that we've imported our data, let's build our graph. To do this, we're going to build the structure for our vertices and edges. At the time of writing, GraphFrames requires a specific naming convention for vertices and edges:

The column representing the vertices needs to have the name ofid. In our case, the vertices of our flight data are the airports. Therefore, we will need to rename the IATA airport code to id in our airports DataFrame.

The columns representing the edges need to have a source (src) and destination (dst). For our flight data, the edges are the flights, therefore the src and dst are the origin and destination columns from the departureDelays_geo DataFrame.

Building the graph

To simplify the edges for our graph, we will create the tripEdges DataFrame with a subset of the columns available within the departureDelays_Geo DataFrame. As well, we created a tripVertices DataFrame that simply renames the IATA column to id to match the GraphFrame naming convention:

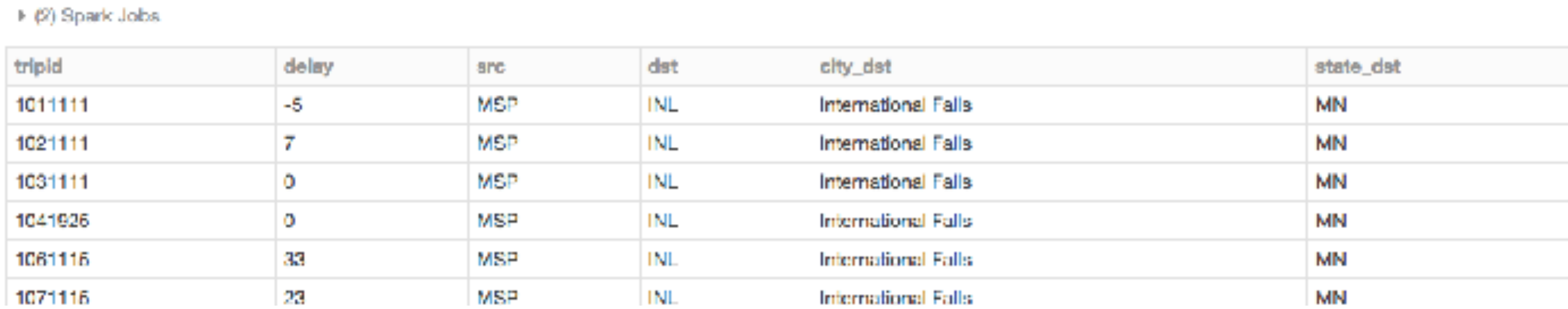
```
# Note, ensure you have already installed

# the GraphFrames spark-package

from pyspark.sql.functions import *

from graphframes import *

# Create Vertices (airports) and Edges (flights)

tripVertices = airports.withColumnRenamed("IATA", "id").distinct()

tripEdges = departureDelays_geo.select("tripid", "delay", "src", "dst", "city_dst", "state_dst")

# Cache Vertices and Edges

tripEdges.cache()

tripVertices.cache()
```

Building the graph

Within Databricks, you can query the data using the display command. For example, to view the tripEdges DataFrame, the command is as follows:

```
display(tripEdges)
```

The output is as follows:



▸ (2) Spark Jobs

tripid	delay	src	dst	city_dst	state_dst
1011111	-5	MSP	INL	International Falls	MN
1021111	7	MSP	INL	International Falls	MN
1031111	0	MSP	INL	International Falls	MN
1041926	0	MSP	INL	International Falls	MN
1061116	33	MSP	INL	International Falls	MN
1071116	23	MSP	INL	International Falls	MN

Now that we have the two DataFrames, we can create a GraphFrame using the GraphFrame command:

```
tripGraph = GraphFrame(tripVertices, tripEdges)
```

Executing simple queries

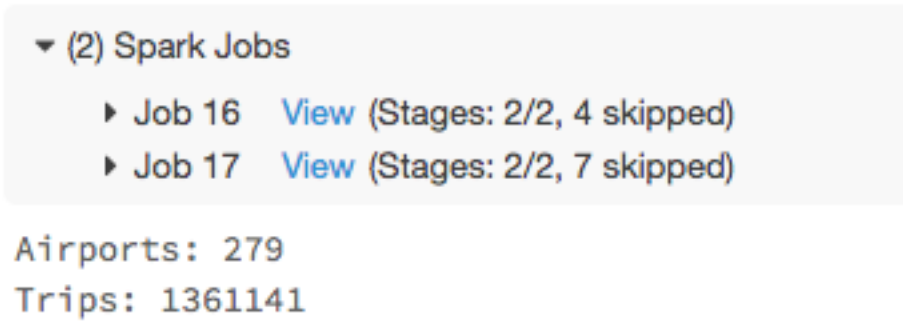
Let's start with a set of simple graph queries to understand flight performance and departure delays.

Determining the number of airports and trips

For example, to determine the number of airports and trips, you can run the following commands:

```
print "Airports: %d" % tripGraph.vertices.count()

print "Trips: %d" % tripGraph.edges.count()
```

As you can see from the results, there are 279 airports with 1.36 million trips:



```
▾ (2) Spark Jobs
    ▸ Job 16   View (Stages: 2/2, 4 skipped)
    ▸ Job 17   View (Stages: 2/2, 7 skipped)

Airports: 279
Trips: 1361141
```

Executing simple queries

Determining the longest delay in this dataset

To determine the longest delayed flight in the dataset, you can run the following query with the result of 1,642 minutes (that's more than 27 hours!):

```
tripGraph.edges.groupBy().max("delay")
```

```
# Output

+----------+

|max(delay)|

+----------+

| 1642|

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

Executing simple queries

Determining the number of delayed versus on-time/early flights

To determine the number of delayed versus on-time (or early) flights, you can run the following queries:

```
print "On-time / Early Flights: %d" % tripGraph.edges.filter("delay <= 0").count()

print "Delayed Flights: %d" % tripGraph.edges.filter("delay > 0"). count()
```

Executing simple queries

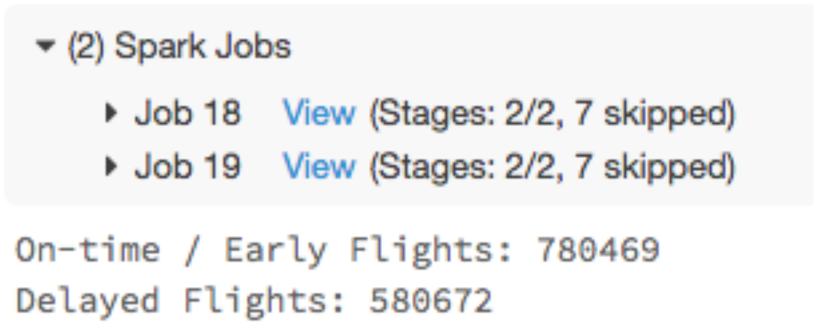
Determining the number of delayed versus on-time/early flights

To determine the number of delayed versus on-time (or early) flights, you can run the following queries:

print "On-time / Early Flights: %d" % tripGraph.edges.filter("delay <= 0").count()

print "Delayed Flights: %d" % tripGraph.edges.filter("delay > 0"). count()

with the results nothing that almost 43% of the flights were delayed!



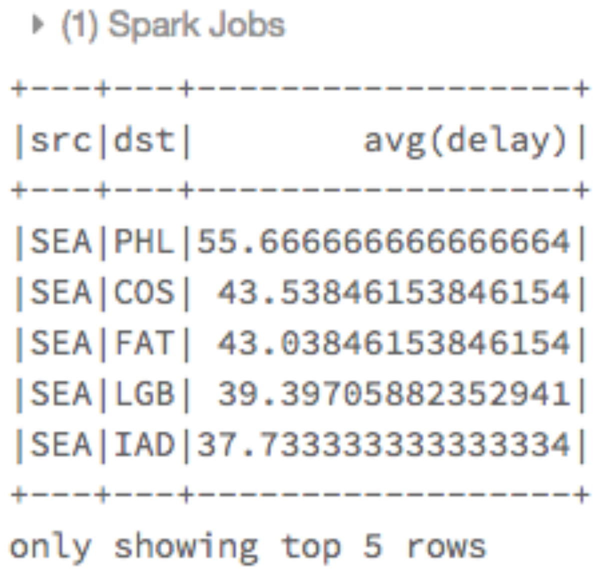
```
▾ (2) Spark Jobs
    ▸ Job 18   View (Stages: 2/2, 7 skipped)
    ▸ Job 19   View (Stages: 2/2, 7 skipped)

On-time / Early Flights: 780469
Delayed Flights: 580672
```

Executing simple queries

What flights departing Seattle are most likely to have significant delays?
Digging further in this data, let's find out the top five destinations for flights departing from Seattle that are most likely to have significant delays. This can be achieved through the following query:

```
tripGraph.edges.filter("src = 'SEA' and delay > 0").groupBy("src", "dst")\

.avg("delay").sort(desc("avg(delay)")).show(5)
```

As you can see in the following results: Philadelphia (PHL), Colorado Springs (COS), Fresno (FAT), Long Beach (LGB), and Washington D.C (IAD) are the top five cities with flights delayed originating from Seattle:



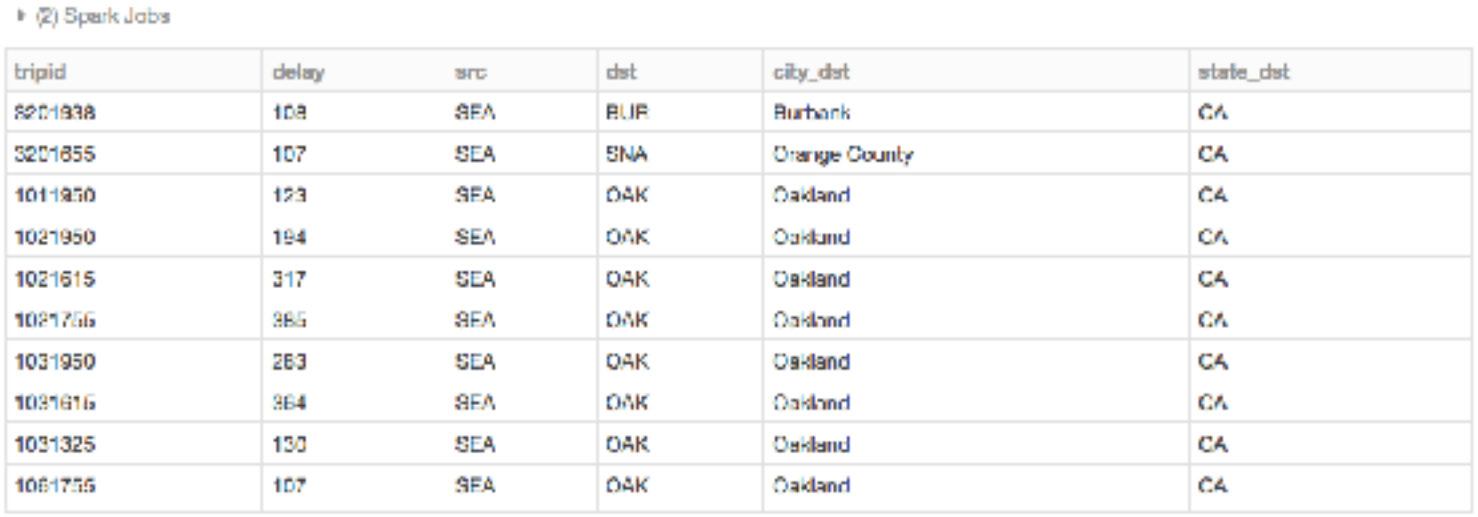
```
▸ (1) Spark Jobs

+---+---+------------------+
|src|dst|        avg(delay)|
+---+---+------------------+
|SEA|PHL|55.666666666666664|
|SEA|COS| 43.53846153846154|
|SEA|FAT| 43.03846153846154|
|SEA|LGB| 39.39705882352941|
|SEA|IAD|37.733333333333334|
+---+---+------------------+
only showing top 5 rows
```

Executing simple queries

What states tend to have significant delays departing from Seattle?

Let's find which states have the longest cumulative delays (with individual delays > 100 minutes) originating from Seattle. This time we will use the display command to review the data:

```
# States with the longest cumulative delays (with individual
# delays > 100 minutes) (origin: Seattle)
display(tripGraph.edges.filter("src = 'SEA' and delay > 100"))
```

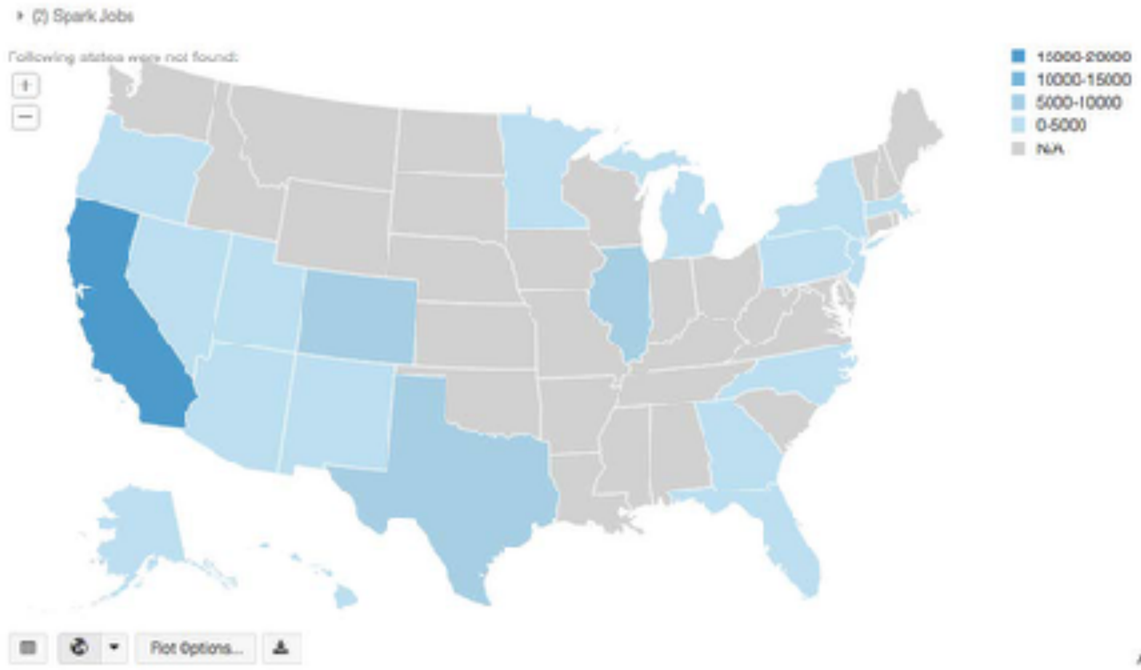


▸ (2) Spark Jobs

tripid	delay	src	dst	city_dst	state_dst
3201938	108	SEA	BUR	Burbank	CA
3201655	107	SEA	SNA	Orange County	CA
1011950	123	SEA	OAK	Oakland	CA
1021950	194	SEA	OAK	Oakland	CA
1021615	317	SEA	OAK	Oakland	CA
1021755	385	SEA	OAK	Oakland	CA
1031950	283	SEA	OAK	Oakland	CA
1031615	364	SEA	OAK	Oakland	CA
1031325	130	SEA	OAK	Oakland	CA
1061755	107	SEA	OAK	Oakland	CA

Executing simple queries

Using the Databricks display command, we can also quickly change from this table view to a map view of the data. As can be seen in the following figure, the state with the most cumulative delays originating from Seattle (in this dataset) is California:

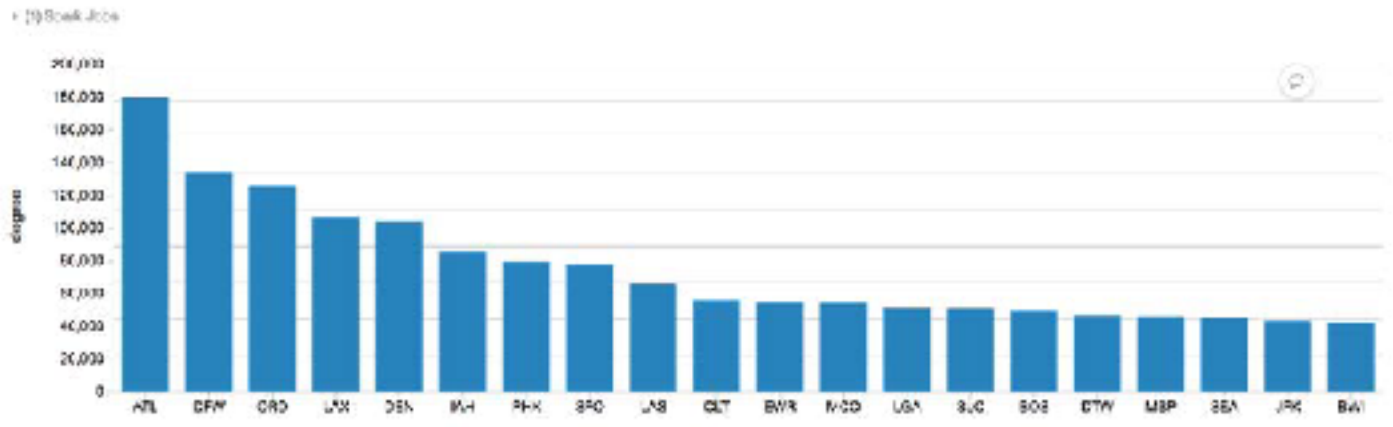


Understanding vertex degrees

Within the context of graph theory, the degrees around a vertex are the number of edges around the vertex. In our flights example, the degrees are then the total number of edges (that is, flights) to the vertex (that is, airports). Therefore, if we were to obtain the top 20 vertex degrees (in descending order) from our graph, then we would be asking for the top 20 busiest airports (most flights in and out) from our graph. This can be quickly determined using the following query:

```
display(tripGraph.degrees.sort(desc("degree")).limit(20))
```

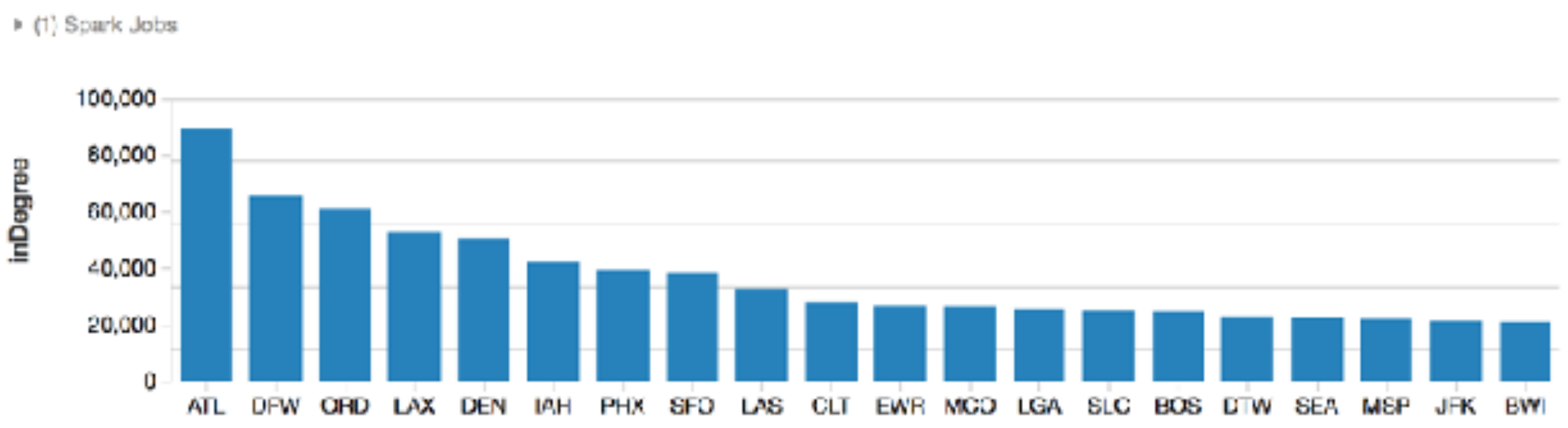
Because we're using the display command, we can quickly view a bar graph of this data:



Understanding vertex degrees

Diving into more details, here are the top 20 inDegrees (that is, incoming flights):

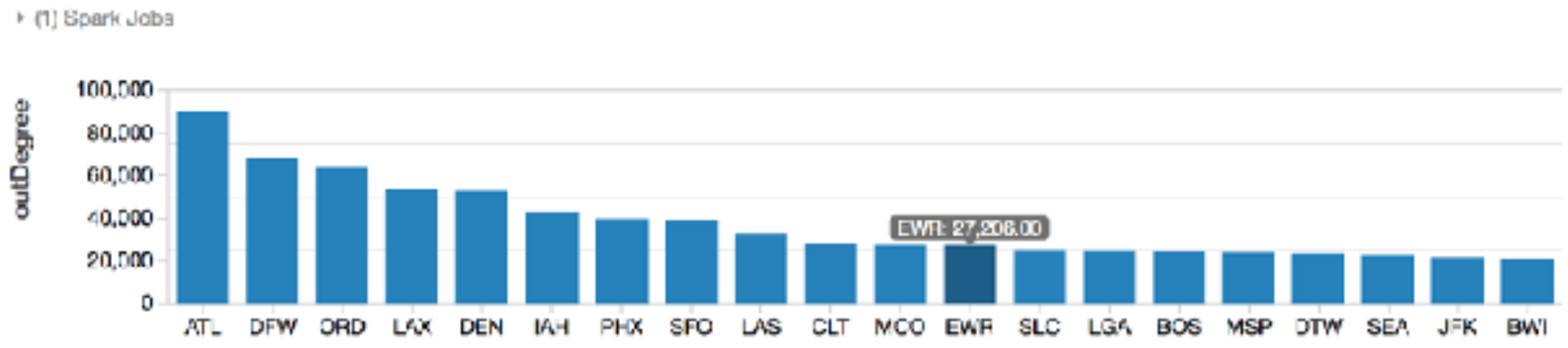
```
display(tripGraph.inDegrees.sort(desc("inDegree")).limit(20))
```



Understanding vertex degrees

While here are the top 20 outDegrees (that is, outgoing flights):

display(tripGraph.outDegrees.sort(desc("outDegree")).limit(20))



Interestingly, while the top 10 airports (Atlanta/ATL to Charlotte/CLT) are ranked the same for incoming and outgoing flights, the ranks of the next 10 airports change (for example, Seattle/SEA is 17th for incoming flights, but 18th for outgoing).

Understanding vertex degrees

Determining the top transfer airports

An extension of understanding vertex degrees for airports is to determine the top transfer airports. Many airports are used as transfer points instead of being the final destination. An easy way to calculate this is by calculating the ratio of inDegrees (the number of flights to the airport) and / outDegrees (the number of flights leaving the airport). Values close to 1 may indicate many transfers, whereas values <1 indicate many outgoing flights and values >1 indicate many incoming flights.

Note that this is a simple calculation that does not consider timing or scheduling of flights, just the overall aggregate number within the dataset:

```
# Calculate the inDeg (flights into the airport) and

# outDeg (flights leaving the airport)

inDeg = tripGraph.inDegrees

outDeg = tripGraph.outDegrees
```

Understanding vertex degrees

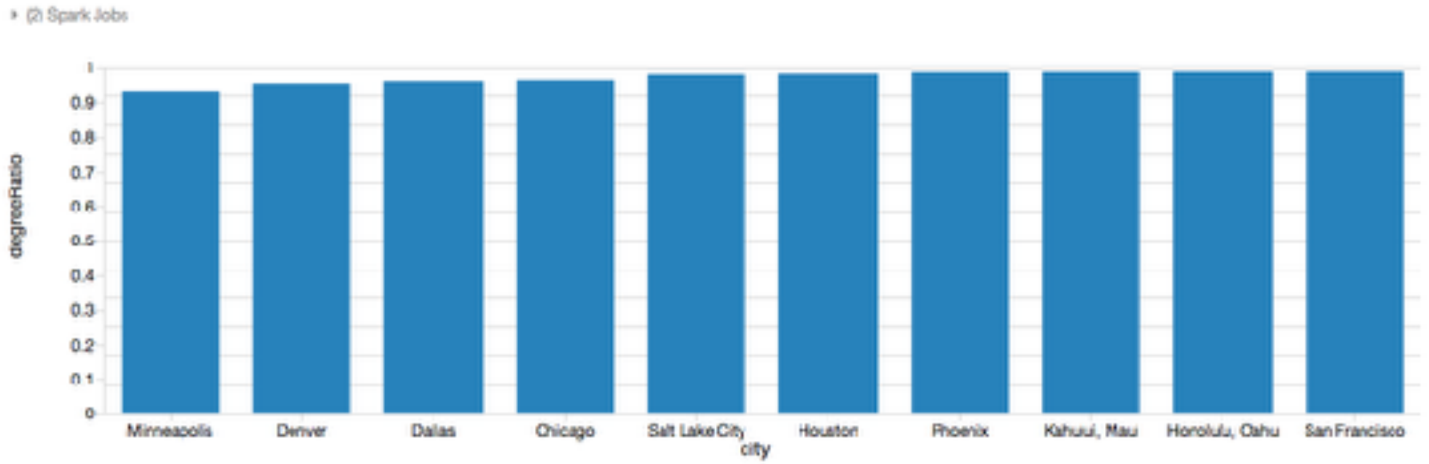
```
# Calculate the degreeRatio (inDeg/outDeg)
degreeRatio = inDeg.join(outDeg, inDeg.id == outDeg.id) \
.drop(outDeg.id) \
.selectExpr("id", "double(inDegree)/double(outDegree) as degreeRatio") \
.cache()
# Join back to the 'airports' DataFrame
# (instead of registering temp table as above)
transferAirports = degreeRatio.join(airports, degreeRatio.id == airports.IATA) \
.selectExpr("id", "city", "degreeRatio") \
.filter("degreeRatio between 0.9 and 1.1")
```

Understanding vertex degrees

# List out the top 10 transfer city airports

```
display(transferAirports.orderBy("degreeRatio").limit(10))
```

The output of this query is a bar chart of the top 10 transfer city airports (that is, hub airports):



This makes sense since these airports are major hubs for national airlines (for example, Delta uses Minneapolis and Salt Lake City as its hub, Frontier uses Denver, American uses Dallas and Phoenix, United uses Houston, Chicago, and San Francisco, and Hawaiian Airlines uses Kahului and Honolulu as its hubs).

Understanding motifs

To easily understand the complex relationship of city airports and the flights between each other, we can use motifs to find patterns of airports (for example, vertices) connected by flights (that is, edges). The result is a DataFrame in which the column names are given by the motif keys. Note that motif finding is one of the new graph algorithms supported as part of GraphFrames.

For example, let's determine the delays that are due to San Francisco International Airport (SFO):

```
# Generate motifs

motifs = tripGraphPrime.find("(a)-[ab]->(b); (b)-[bc]->(c)")\

.filter("(b.id = 'SFO') and (ab.delay > 500 or bc.delay > 500) and bc.tripid > ab.tripid and bc.tripid < ab.tripid + 10000”)

# Display motifs

display(motifs)
```

Breaking down the preceding query, the (x) represents the vertex (that is, airport) while the [xy] represents the edge (that is, flights between airports).

Understanding motifs

Therefore, to determine the delays that are due to SFO, use the following:

The vertex (b) represents the airport in the middle (that is, SFO)

The vertex(a)represents the origin airport (within the dataset)

The vertex (c) represents the destination airport (within the dataset)

The edge [ab] represents the flight between (a) (that is, origin) and (b) (that is, SFO)

The edge [bc] represents the flight between (b) (that is, SFO) and (c) (that is, destination)

Within the filter statement, we put in some rudimentary constraints (note that this is an over simplistic representation of flight paths):

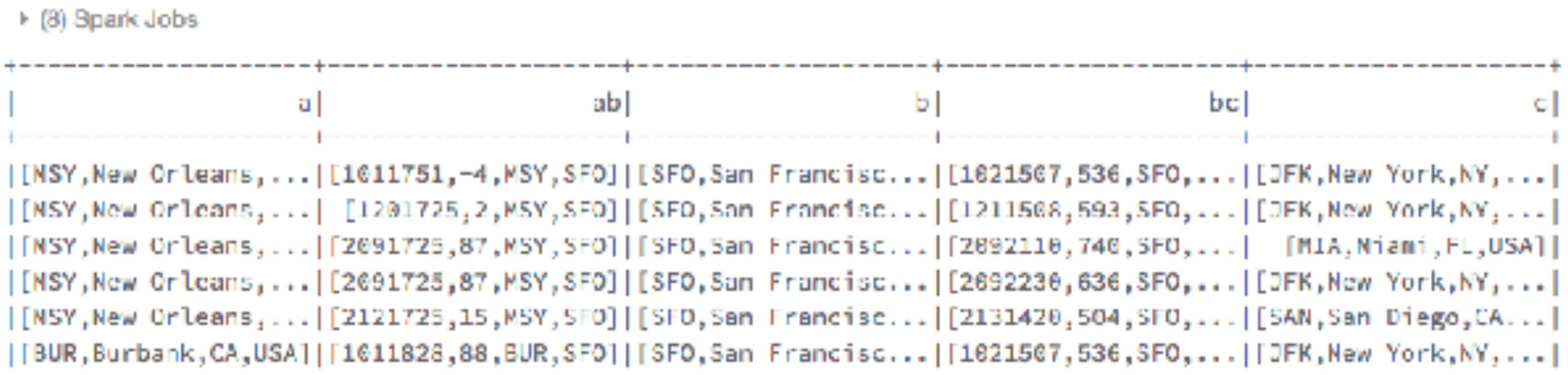
b.id = 'SFO' denotes that the middle vertex (b) is limited to just SFO airport

(ab.delay > 500 or bc.delay > 500) denotes that we are limited to flights that have delays greater than 500 minutes

Understanding motifs

(bc.tripid > ab.tripid and bc.tripid < ab.tripid + 10000) denotes that the (ab) flight must be before the (bc) trip and within the same day. The tripid was derived from the date time, thus explaining why it could be simplified this way

The output of this query is noted in the following figure:



```
▸ (8) Spark Jobs

+--------------------+--------------------+--------------------+--------------------+--------------------+
|                   a|                  ab|                   b|                  bc|                   c|
+--------------------+--------------------+--------------------+--------------------+--------------------+
|[NSY,New Orleans,...|[1011751,-4,MSY,SFO]|[SFO,San Francisc...|[1021507,536,SFO,...|[JFK,New York,NY,...|
|[NSY,New Orleans,...| [1201725,2,MSY,SFO]|[SFO,San Francisc...|[1211508,593,SFO,...|[JFK,New York,NY,...|
|[NSY,New Orleans,...|[2091725,87,MSY,SFO]|[SFO,San Francisc...|[2092110,740,SFO,...|  [MIA,Miami,FL,USA]|
|[NSY,New Orleans,...|[2091725,87,MSY,SFO]|[SFO,San Francisc...|[2092230,636,SFO,...|[JFK,New York,NY,...|
|[NSY,New Orleans,...|[2121725,15,MSY,SFO]|[SFO,San Francisc...|[2131420,504,SFO,...|[SAN,San Diego,CA...|
|[BUR,Burbank,CA,USA]|[1011828,88,BUR,SFO]|[SFO,San Francisc...|[1021507,536,SFO,...|[JFK,New York,NY,...|
```

Determining airport ranking using PageRank

Because GraphFrames is built on top of GraphX, there are several algorithms that we can immediately leverage.

PageRank was popularized by the Google Search Engine and created by Larry Page.

To quote Wikipedia:

"PageRank works by counting 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."

While the preceding example refers to web pages, this concept readily applies to any graph structure whether it is created from web pages, bike stations, or airports.

Yet the interface via GraphFrames is as simple as calling a method.

GraphFrames.PageRank will return the PageRank results as a new column appended to the vertices DataFrame to simplify our downstream analysis.

Determining airport ranking using PageRank

As there are many flights and connections through the various airports included in this dataset, we can use the PageRank algorithm to have Spark traverse the graph iteratively to compute a rough estimate of how important each airport is:

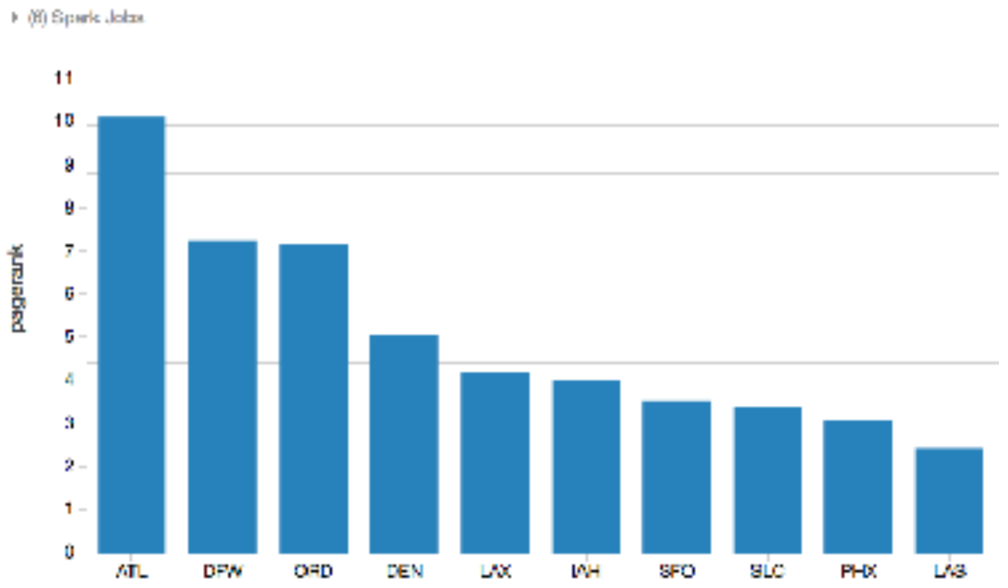
```
# Determining Airport ranking of importance using 'pageRank'

ranks = tripGraph.pageRank(resetProbability=0.15, maxIter=5)

# Display the pageRank output

display(ranks.vertices.orderBy(ranks.vertices.pagerank.desc()). limit(20))
```

Note that resetProbability = 0.15 represents the probability of resetting to a random vertex (this is the default value) while maxIter = 5 is a set number of iterations.

For more information on PageRank parameters, please refer to Wikipedia > Page Rank at https://en.wikipedia.org/wiki/PageRank.

Determining airport ranking using PageRank

The results of the PageRank are noted in the following bar graph:



In terms of airport ranking, the PageRank algorithm has determined that ATL (Hartsfield-Jackson Atlanta International Airport) is the most important airport in the United States.

This observation makes sense as ATL is not only the busiest airport in the United States (http://bit.ly/2eTGHs4), but it is also the busiest airport in the world (2000-2015) (http://bit.ly/2eTGDsy).

Determining airport ranking using PageRank

Determining the most popular non-stop flights

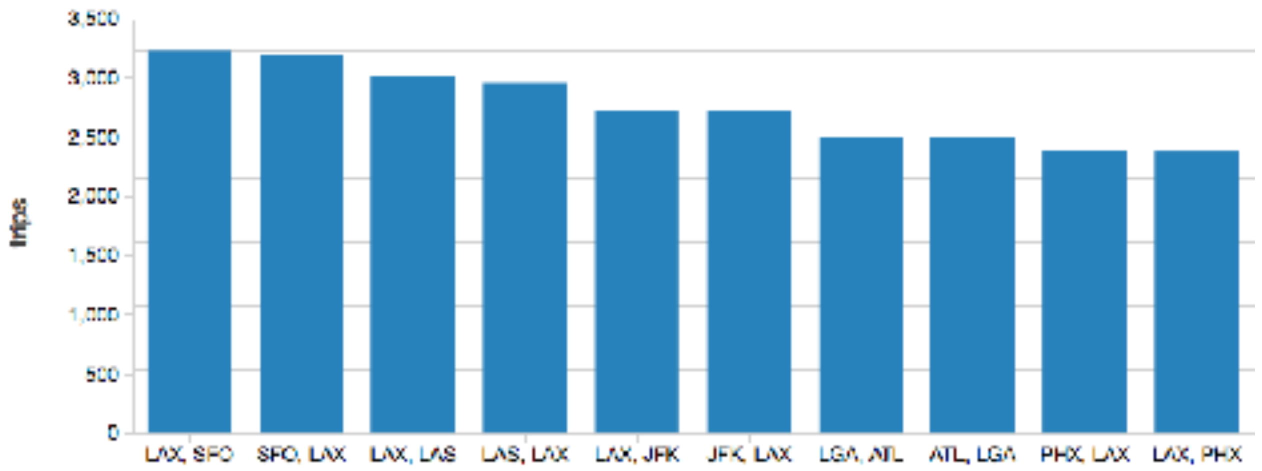
Expanding upon our tripGraph GraphFrame, the following query will allow us to find the most popular non-stop flights in the US (for this dataset):

```
# Determine the most popular non-stop flights

import pyspark.sql.functions as func

topTrips = tripGraph \

.edges \

.groupBy("src", "dst") \

.agg(func.count("delay").alias("trips"))

# Show the top 20 most popular flights (single city hops)

display(topTrips.orderBy(topTrips.trips.desc()).limit(20))
```

Determining airport ranking using PageRank

Note, while we are using the delay column, we're just actually doing a count of the number of trips. Here's the output:

▸ (1) Spark Jobs



As can be observed from this query, the two most frequent non-stop flights are between LAX (Los Angeles) and SFO (San Francisco). The fact that these flights are so frequent indicates their importance in the airline market. As noted in the New York Times article from April 4, 2016, *Alaska Air Sees Virgin America as Key to West Coast* (http://nyti.ms/2ea1uZR), acquiring slots at these two airports was one of the reasons why Alaska Airlines purchased Virgin Airlines. Graphs are not just fun but also contain potentially powerful business insight!

Using Breadth-First Search

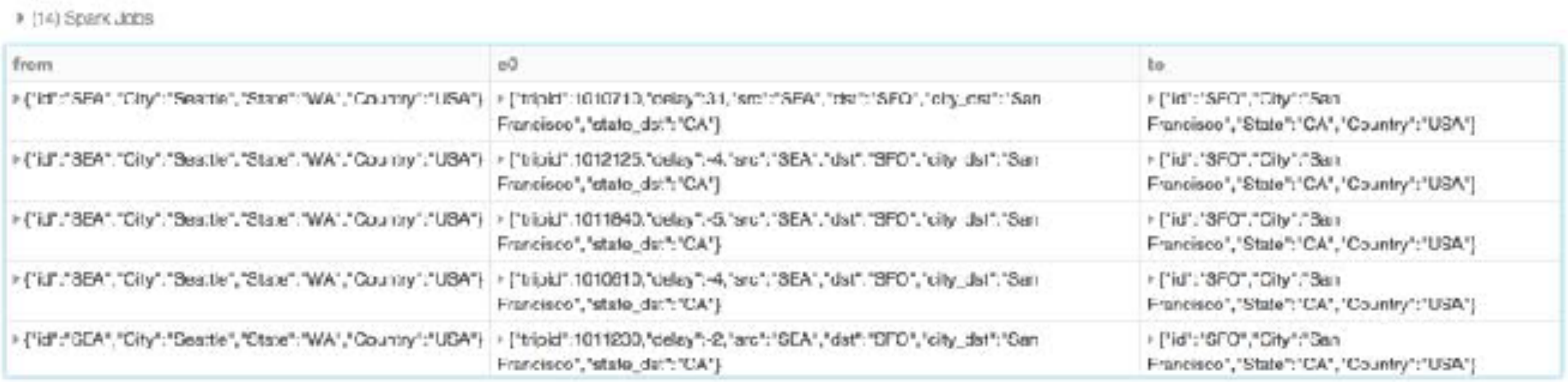
The **Breadth-first search (BFS)** is a new algorithm as part of GraphFrames that finds the shortest path from one set of vertices to another.

In this section, we will use BFS to traverse our tripGraph to quickly find the desired vertices (that is, airports) and edges (that is, flights). Let's try to find the shortest number of connections between cities based on the dataset. Note that these examples do not consider time or distance, just hops between cities. For example, to find the number of direct flights between Seattle and San Francisco, you can run the following query:

```
# Obtain list of direct flights between SEA and SFO

filteredPaths = tripGraph.bfs(

fromExpr = "id = 'SEA'",

toExpr = "id = 'SFO'",

maxPathLength = 1)

# display list of direct flights

display(filteredPaths)
```

Using Breadth-First Search

fromExpr and toExpr are the expressions indicating the origin and destination airports (that is, SEA and SFO, respectively). The maxPathLength = 1 indicates that we only want one edge between the two vertices, that is, a non-stop flight between Seattle and San Francisco. As noted in the following results, there are many direct flights between Seattle and San Francisco:



▸ (14) Spark Jobs

from	e0	to
▸ {"id":"SEA","City":"Seattle","State":"WA","Country":"USA"}	▸ ["tripid":1010710,"delay":31,"src":"SEA","dst":"SFO","city_dst":"San Francisco","state_dst":"CA"]	▸ ["id":"SFO","City":"San Francisco","State":"CA","Country":"USA"]
▸ {"id":"SEA","City":"Seattle","State":"WA","Country":"USA"}	▸ ["tripid":1012125,"delay":-4,"src":"SEA","dst":"SFO","city_dst":"San Francisco","state_dst":"CA"]	▸ ["id":"SFO","City":"San Francisco","State":"CA","Country":"USA"]
▸ {"id":"SEA","City":"Seattle","State":"WA","Country":"USA"}	▸ ["tripid":1011840,"delay":-5,"src":"SEA","dst":"SFO","city_dst":"San Francisco","state_dst":"CA"]	▸ ["id":"SFO","City":"San Francisco","State":"CA","Country":"USA"]
▸ {"id":"SEA","City":"Seattle","State":"WA","Country":"USA"}	▸ ["tripid":1010610,"delay":-4,"src":"SEA","dst":"SFO","city_dst":"San Francisco","state_dst":"CA"]	▸ ["id":"SFO","City":"San Francisco","State":"CA","Country":"USA"]
▸ {"id":"SEA","City":"Seattle","State":"WA","Country":"USA"}	▸ ["tripid":1011230,"delay":-2,"src":"SEA","dst":"SFO","city_dst":"San Francisco","state_dst":"CA"]	▸ ["id":"SFO","City":"San Francisco","State":"CA","Country":"USA"]

Using Breadth-First Search

But how about if we want to determine the number of direct flights between San Francisco and Buffalo? Running the following query will note that there are no results, that is, no direct flights between the two cities:

```
# Obtain list of direct flights between SFO and BUF

filteredPaths = tripGraph.bfs(

fromExpr = "id = 'SFO'",

toExpr = "id = 'BUF'",

maxPathLength = 1)

# display list of direct flights

display(filteredPaths)
```

Using Breadth-First Search

Once we modify the preceding query to maxPathLength = 2, that is, one layover, then you will see a lot more flight options:

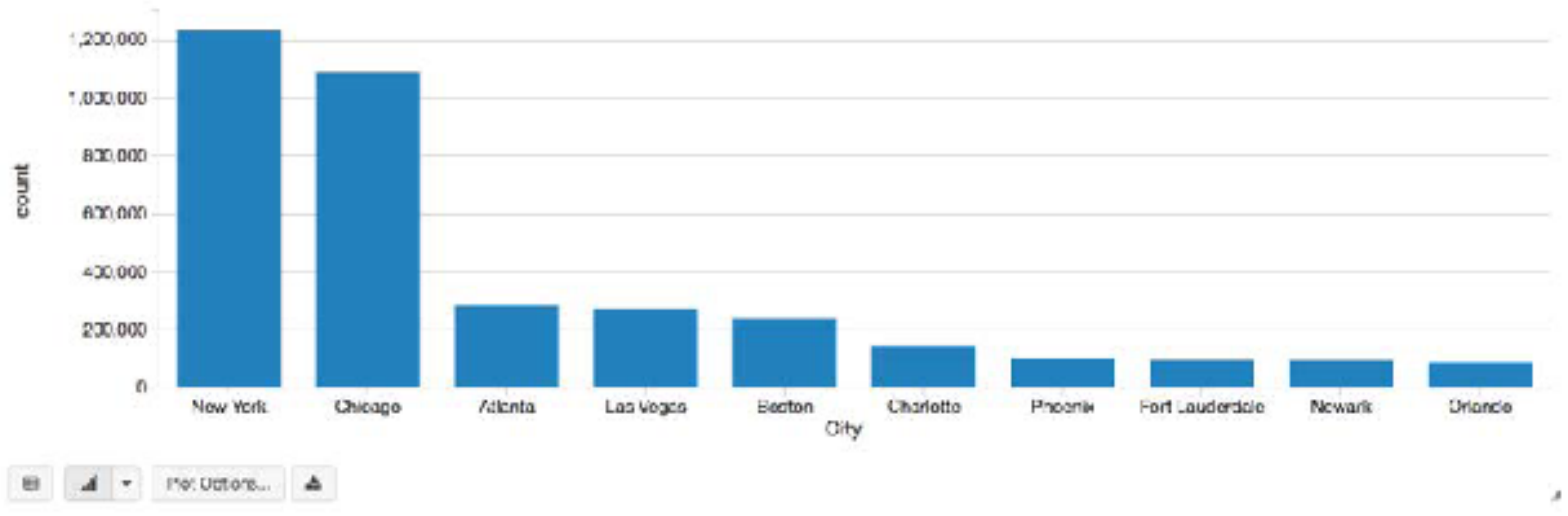
```
# display list of one-stop flights between SFO and BUF

filteredPaths = tripGraph.bfs(

fromExpr = "id = 'SFO'",

toExpr = "id = 'BUF'",

maxPathLength = 2)

# display list of flights

display(filteredPaths)
```

Using Breadth-First Search

But now that I have my list of airports, how can I determine which layover airports are more popular between SFO and BUF? To determine this, you can now run the following query:

```
# Display most popular layover cities by descending count

display(filteredPaths.groupBy("v1.id", "v1.City").count(). orderBy(desc("count")).limit(10))
```

The output is shown in the following bar chart:



Visualizing flights using D3

To get a powerful and fun visualization of the flight paths and connections in this dataset, we can leverage the Airports D3 visualization (https://mbostock.github. io/d3/ talk/20111116/airports.html) within our Databricks notebook. By connecting our GraphFrames, DataFrames, and D3 visualizations, we can visualize the scope of all the flight connections as noted for all on-time or early departing flights within this dataset.

The blue circles represent the vertices (that is, airports) where the size of the circle represents the number of edges (that is, flights) in and out of those airports. The black lines are the edges themselves (that is, flights) and their respective connections to the other vertices (that is, airports). Note for any edges that go offscreen, they are representing vertices (that is, airports) in the states of Hawaii and Alaska.

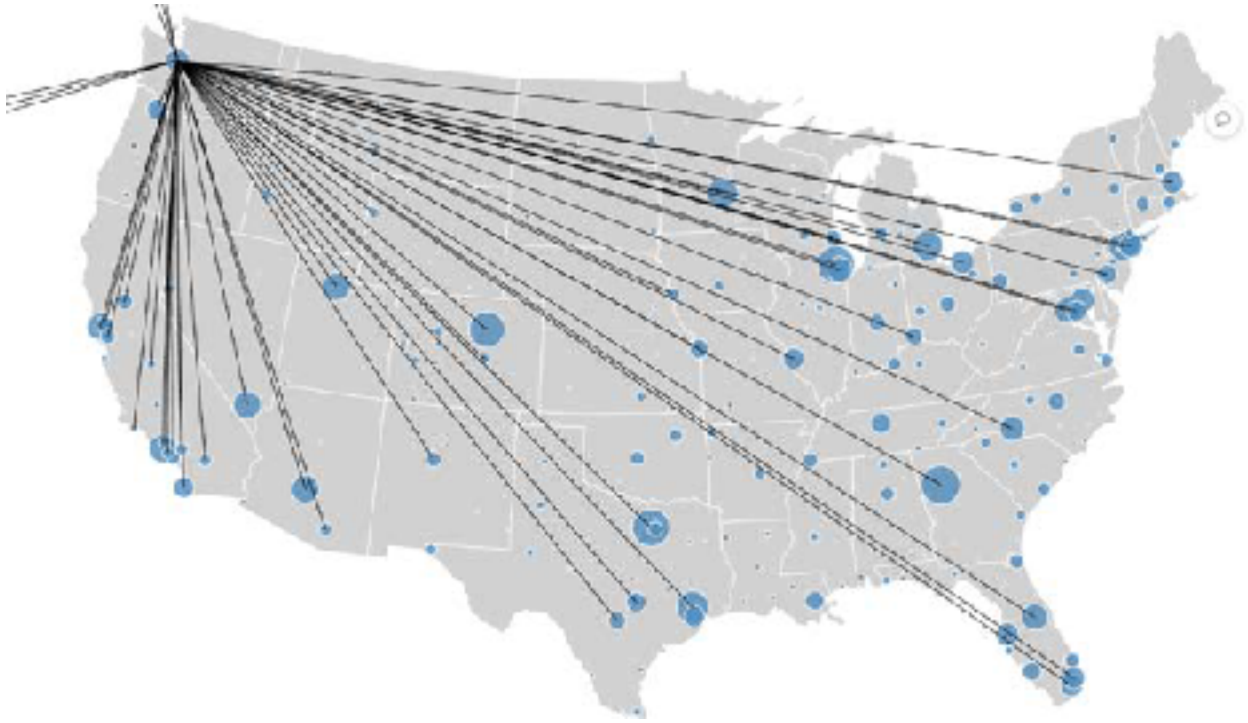
Visualizing flights using D3

For this to work, we first create a scala package called d3a that is embedded in our notebook (you can download it from here: http://bit.ly/2kPkXkc). Because we're using Databricks notebooks, we can make Scala calls within our PySpark notebook:

```
%scala

// On-time and Early Arrivals

import d3a._

graphs.force(

height = 800,

width = 1200,

clicks = sql("""select src, dst as dest, count(1) as count from departureDelays_geo where delay <= 0 group by src, dst""").as[Edge])
```

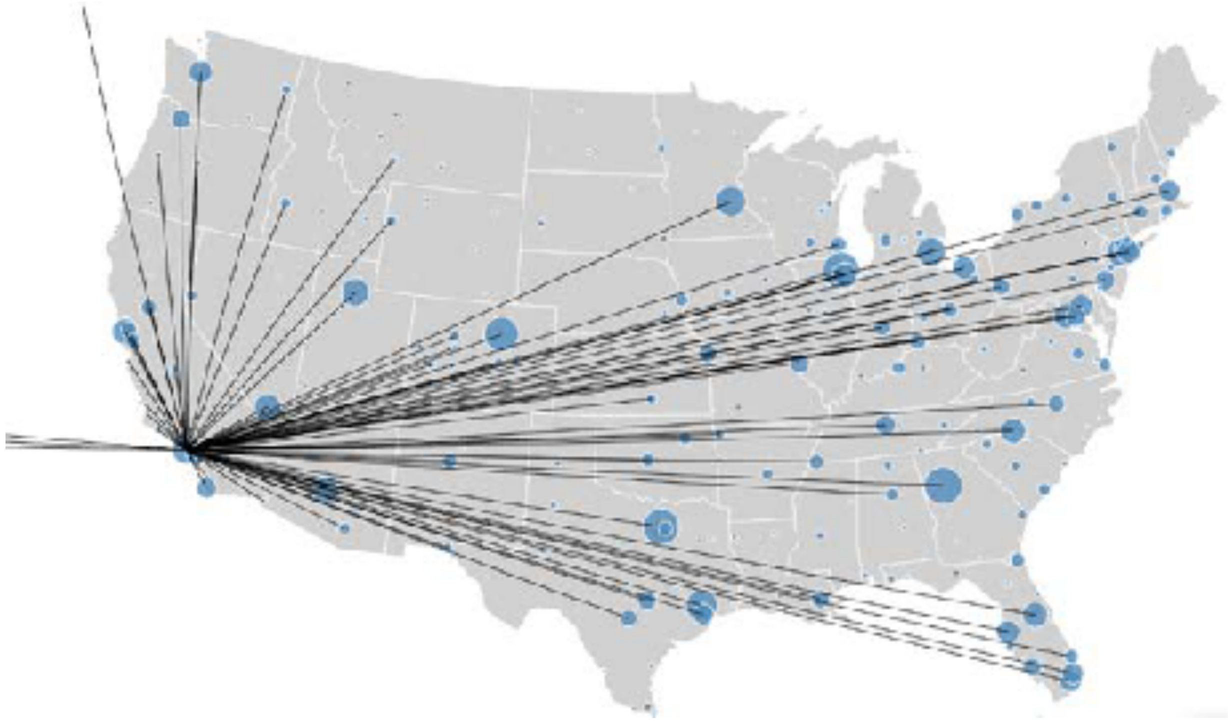
Visualizing flights using D3

The results of the preceding query for on-time and early arrivals flights are visualized in the following screenshot:



Visualizing flights using D3

You can hover over the airports (blue circle, vertex) in the airports D3 visualization where the lines are the edges (flights). The preceding visualization is a snapshot when hovering over Seattle (SEA) airport; while the following visualization is a snapshot when hovering over Los Angeles (LAX) airport:



Summary

As you can see in this module, you can easily perform a lot of powerful data analysis by executing queries against graph structures. With GraphFrames, you can leverage the power, simplicity, and performance of the DataFrame API against your graph problems.

For more information on GraphFrames, please refer to the following resources:

Introducing GraphFrames (http://bit.ly/2dBPhKn)

On-Time Flight Performance with GraphFrames for Apache Spark (http://bit.ly/2c804ZD)

On-Time Flight Performance with GraphFrames for Apache Spark (Spark 2.0) Notebook (http://bit.ly/2kPkXkc)

GraphFrames Overview (http://graphframes.github.io/)

Pygraphframes documentation (http://graphframes.github.io/api/ python/graphframes.html)

GraphX Programming Guide (http://spark.apache.org/docs/latest/ graphx-programming-guide.html)

Contact Us on:

G K T C S Innovations Pvt. Ltd.
IT Training & Consultancy,

Mobile: +91- 9975072320
Email : surendra@gktcs.com
Web: www.gktcs.com