PySpark

Ming-Hwa Wang, Ph.D.
COEN 242 Introduction to Big Data
Department of Computer Engineering
Santa Clara University

https://github.com/PacktPublishing/Learning-PySpark/

Load RDDs

- Read JSON file:
  - Read JSON file from sqlContext:
```
df = sqlContext.read \
    .format('json').load('py/test/sql/people.json')
```
  - Read JSON file from spark.read.format('json').load(f) or spark.read.json(f):
```
df = spark.read.format('json').load('py/test/sql/people.json')
df = spark.read.json('py/test/sql/people.json')
```
- Create RDDs:
  - .parallelize(...) a collection (tuple, list, dict, etc.):
```
data = sc.parallelize(
    [('Amber', 22), ('Alfred', 23), ('Skye',4), ('Albert', 12),
    ('Amber', 9)]
).collect()
```
  - reference a file (or zipped/unzipped files) with number of partitions:
```
data_from_file = sc.\
textFile(
    '/Users/drabast/Documents/PySpark_Data/VS14MORT.txt.gz', 4
).take(1)
```
- Extract information with Python method (can slow down your application):
```
def extractInformation(row):
    import re
    import numpy as np
    selected_indices = [
        2,4,5,6,7,9,10,11,12,13,14,15,16,17,18,...
        77,78,79,81,82,83,84,85,87,89
    ]
    record_split = re\
    .compile(
        r'([\s]{19})([0-9]{1})([\s]{40})...
        ([\s]{33})([0-9\s]{3})([0-9\s]{1})([0-9\s]{1})')
    try:
        rs = np.array(record_split.split(row))[selected_indices]
    except:
        rs = np.array(['-99'] * len(selected_indices))
    return rs
```
  - Filter out the malformed records:
```
data_from_file_conv = data_from_file.map(extractInformation)
```

Transformations

- The .map() transformation applies to each element of the dataset:
```
data_2014 = data_from_file_conv.map(lambda row: int(row[16]))
```
- The .filter() transformation select elements that fit specified criteria:
```
data_filtered = data_from_file_conv.filter(
    lambda row: row[16] == '2014' and row[21] == '0')
```
- The .flatMap() transformation returns a flattened result instead of a list:
```
data_2014_flat = data_from_file_conv.flatMap(
    lambda row: (row[16], int(row[16]) + 1))
```
- The .distinct() transformation returns a list of distinct values in a column (an expensive transformation):
```
distinct_gender = data_from_file_conv.map(
    lambda row: row[5]).distinct()
```
- The .sample() transformation returns a randomized sample from the dataset with replacement:
```
fraction = 0.1
seed = 666
data_sample = data_from_file_conv.sample(False, fraction, seed)
```
- The .join() transformation returns records when the two RDDs match (an expensive transformation):
```
rdd1 = sc.parallelize([('a', 1), ('b', 4), ('c',10)])
rdd2 = sc.parallelize([('a', 4), ('a', 1), ('b', '6'), ('d', 15)])
rdd3 = rdd1.join(rdd2)
```
  produce the following:
```
[ ('b', (4, '6')), ('a', (1, 4)), ('a', (1, 1))]
```
- The .leftOuterJoin() transformation returns records from the left RDD with records from the right one appended in places where the two RDDs match (an expensive transformation):
```
rdd1 = sc.parallelize([('a', 1), ('b', 4), ('c',10)])
rdd2 = sc.parallelize([('a', 4), ('a', 1), ('b', '6'), ('d', 15)])
rdd3 = rdd1.leftOuterJoin(rdd2)
```
  produce the following:
```
[('c', (10, None)), ('b', (4, '6')), ('a', (1, 4)), ('a', (1, 1))]
```
- The .intersection() transformation returns records that are equal in both RDDs:
```
rdd5 = rdd1.intersection(rdd2)
rdd5.collect()
```
  produce the following:
```
[('a', 1)]
```
- The .repartition() transformation changes the number of partitions (an expensive transformation):
```
rdd1 = rdd1.repartition(4)
len(rdd1.glom().collect())
```

The .glom() method produces a list where each element is another list of all elements of the dataset present in a specified partition.

Actions

Actions execute the scheduled task on the dataset. This might contain no transformations.

- The .take() method returns the *n* top rows from a single data partition:
```
data_first = data_from_file_conv.take(1)
```
- The .takeSample() method returns the *n* random sample rows from a single data partition:
```
replacement = False
n = 1
seed = 667
data_take_sampled = data_from_file_conv.takeSample(
    replacement, n, seed)
```
- The .collect() method returns all the elements of the RDD to the driver:
- The .reduce() method reduces the elements of an RDD using a specified method:
```
rdd1.map(lambda row: row[1]).reduce(lambda x, y: x + y)
```
The functions passed as a reducer need to be associative and commutative.
- The .reduceByKey() method performs a reduction on a key-by-key basis:
```
data_key = sc.parallelize(
    [('a', 4), ('b', 3), ('c', 2), ('a', 8), ('d', 2), ('b', 1),
    ('d', 3)], 4)
data_key.reduceByKey(lambda x, y: x + y).collect()
```
- The .count() method counts the number of elements in the RDD.
```
data_reduce.count()
```
- The .countByKey() method gets the counts of distinct keys:
```
data_key.countByKey().items()
```
- The .saveAsTextFile() method saves to text files, each partition to a separate file:
```
data_key.saveAsTextFile(
'/Users/drabast/Documents/PySpark_Data/data_key.txt')
```
    - To read it back, parse it back as all the rows are treated as strings:
```
def parseInput(row):
    import re
    pattern = re.compile(r'\(\'([a-z])\', ([0-9])\)')
    row_split = pattern.split(row)
    return (row_split[1], int(row_split[2]))
data_key_reread = sc.textFile(
    '/Users/drabast/Documents/PySpark_Data/data_key.txt') \
.map(parseInput)
data_key_reread.collect()
```
- The .foreach() method applies the same function to each element of the RDD:
```
def f(x):
    print(x)
data_key.foreach(f)
```

DataFrames

- Generating JSON data:
```
stringJSONRDD = sc.parallelize((
    """{
        "id": "123",
        "name": "Katie",
        "age": 19,
        "eyeColor": "brown"
    }""",
    """{
        "id": "234",
        "name": "Michael",
        "age": 22,
        "eyeColor": "green"
    }""",
    """{
        "id": "345",
        "name": "Simone",
        "age": 23,
        "eyeColor": "blue"
    }"""
))
```
- Creating a DataFrame:
```
swimmersJSON = spark.read.json(stringJSONRDD)
```
- Creating a temporary table:
```
swimmersJSON.createOrReplaceTempView("swimmersJSON")
```
- DataFrame API query using the show(n=10) method:
```
swimmersJSON.show()
```
- SQL query
```
spark.sql("select * from swimmersJSON").collect()
```
    - Databricks uses the %sql command and run your SQL statement directly within a notebook cell.
- Interoperating with RDDs: programmatically specifying the schema:
```
# Import types
from pyspark.sql.types import *
# Generate comma delimited data
stringCSVRDD = sc.parallelize([
    (123, 'Katie', 19, 'brown'),
    (234, 'Michael', 22, 'green'),
    (345, 'Simone', 23, 'blue')
])
# Specify schema
schema = StructType([
    StructField("id", LongType(), True),
    StructField("name", StringType(), True),
    StructField("age", LongType(), True),
```

```
    StructField("eyeColor", StringType(), True)
])
# Apply the schema to the RDD and Create DataFrame
swimmers = spark.createDataFrame(stringCSVRDD, schema)
# Creates a temporary view using the DataFrame
swimmers.createOrReplaceTempView("swimmers")
# find schema from DataFrame
swimmers.printSchema()
```

- Querying with the DataFrame

```
# get the number of rows
swimmers.count()
# Running filter statements. Get the id, age where age = 22
swimmers.select("id", "age").filter("age = 22").show()
# Another way to write the above query is below
swimmers.select(swimmers.id, swimmers.age) \
    .filter(swimmers.age == 22).show()
# Get the name, eyeColor where eyeColor like 'b%'
swimmers.select("name", "eyeColor") \
    .filter("eyeColor like 'b%'").show()
```

- Querying with SQL

```
#number of rows
spark.sql("select count(1) from swimmers").show()
# running filter statements using the where Clauses
# Get the id, age where age = 22 in SQL
spark.sql("select id, age from swimmers where age = 22").show()
spark.sql(
    "select name, eyeColor from swimmers where eyeColor like 'b%'"
).show()
```

- DataFrame scenario example – on-time flight performance.
  - Preparing the source datasets

```
# Set File Paths
flightPerfFilePath =
    "/databricks-datasets/flights/departuredelays.csv"
airportsFilePath =
    "/databricks-datasets/flights/airport-codes-na.txt"
# Obtain Airports dataset
airports = spark.read.csv(
    airportsFilePath, header='true', inferSchema='true',
    sep='\t'
)
airports.createOrReplaceTempView("airports")
# Obtain Departure Delays dataset
# csv file can be common-delimited (default) or tab-delimited
flightPerf = spark.read.csv(flightPerfFilePath, header='true')
flightPerf.createOrReplaceTempView("FlightPerformance")
# Cache the Departure Delays dataset
flightPerf.cache()
```

  - Joining flight performance and airports

```
# Query Sum of Flight Delays by City and Origin Code
# (for Washington State)
spark.sql("""
    select a.City, f.origin, sum(f.delay) as Delays
    from FlightPerformance f
    join airports a
    on a.IATA = f.origin
    where a.State = 'WA'
    group by a.City, f.origin
    order by sum(f.delay) desc"""
).show()
```

  - Databricks notebook can use the %sql function to execute SQL statements within that notebook cell to get the same result as previous query but easier to read:

```
%sql
-- Query Sum of Flight Delays by City and Origin Code (for
-- Washington State)
select a.City, f.origin, sum(f.delay) as Delays
from FlightPerformance f
join airports a
on a.IATA = f.origin
where a.State = 'WA'
group by a.City, f.origin
order by sum(f.delay) desc
```

  - Visualizing our flight-performance data

Getting familiar with your data

We can build a model without knowing data by taking longer time with suboptimal results. Thus, any serious data scientist or data modeler will become acquainted with the dataset before starting any modeling.

- Descriptive statistics tell you the basic information about your dataset: mean, standard deviation, min and max.

```
import pyspark.sql.types as typ
fraud = sc.textFile('ccFraud.csv.gz')
header = fraud.first()
fraud = fraud.filter(lambda row: row != header) \
    .map(lambda row: [int(elem) for elem in row.split(',')])
# create the schema for our DataFrame:
fields = [*[
    typ.StructField(h[1:-1], typ.IntegerType(), True) \
    for h in header.split(',')
]]
schema = typ.StructType(fields)
# create our DataFrame:
fraud_df = spark.createDataFrame(fraud, schema)
# get the schema of our DataFrame:
fraud_df.printSchema()
```

```
# for categorical features, count the frequencies of their values
fraud_df.groupby('gender').count().show()
# for the truly numerical features, use the .describe() method
numerical = ['balance', 'numTrans', 'numIntlTrans']
desc = fraud_df.describe(numerical)
desc.show()
# check the skeweness
fraud_df.agg({'balance': 'skewness'}).show()
```

A list of aggregation functions (the names are fairly self-explanatory) includes: avg(), count(), countDistinct(), first(), kurtosis(), max(), mean(), min(), skewness(), stddev(), stddev_pop(), stddev_samp(), sum(), sumDistinct(), var_pop(), var_samp() and variance().

- Correlations

```
# calculate pairwise correlations:
fraud_df.corr('balance', 'numTrans')
# create a correlations matrix:
n_numerical = len(numerical)
corr = []
for i in range(0, n_numerical):
    temp = [None] * i
    for j in range(i+1, n_numerical):
        temp.append(fraud_df.corr(numerical[i], numerical[j]))
    corr.append(temp)
```

- Visualization using visualization packages: matplotlib and Bokeh (preinstalled with Anaconda).

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('ggplot')
import bokeh.charts as chrt
from bokeh.io import output_notebook
output_notebook()
```

  - Histograms: visualize distribution of your features

```
# aggregate the data first:
hists = fraud_df.select('balance').rdd.flatMap(
    lambda row: row
).histogram(20)
# plot the histogram using matplotlib:
data = {
    'bins': hists[0][:-1], 'freq': hists[1]
}
plt.bar(data['bins'], data['freq'], width=2000)
plt.title('Histogram of \'balance\'')
# histogram created with Bokeh (which uses interactive D3.js):
b_hist = chrt.Bar(
    data, values='freq', label='bins',
    title='Histogram of \'balance\''
)
chrt.show(b_hist)
# small data, use matplotlib's .hist() or Bokeh's .Histogram()
data_driver = {
    'obs': fraud_df.select('balance').rdd.flatMap(
        lambda row: row
    ).collect()
}
plt.hist(data_driver['obs'], bins=20)
plt.title('Histogram of \'balance\' using .hist()')
b_hist_driver = chrt.Histogram(
    data_driver, values='obs',
    title='Histogram of \'balance\' using .Histogram()',
    bins=20
)
chrt.show(b_hist_driver)
```

  - Interactions between features: scatter charts for up to 3 variables, 3D visualizations with temporal data, and sample huge dataset.

```
# samples at a predefined sampling fraction
data_sample = fraud_df.sampleBy(
    'gender', {1: 0.0002, 2: 0.0002}
).select(numerical)
# put multiple 2D charts in one:
data_multi = dict([
    (elem, data_sample.select(elem).rdd \
    .flatMap(lambda row: row).collect()) for elem in numerical
])
sctr = chrt.Scatter(data_multi, x='balance', y='numTrans')
chrt.show(sctr)
```

Prepare Data for Modeling

Checking for duplicates, missing observations, and outliers

- Duplicates removal with .distinct() method:

```
# example dataset:
df = spark.createDataFrame(
    [
        (1, 144.5, 5.9, 33, 'M'),
        (2, 167.2, 5.4, 45, 'M'),
        (3, 124.1, 5.2, 23, 'F'),
        (4, 144.5, 5.9, 33, 'M'),
        (5, 133.2, 5.7, 54, 'F'),
        (3, 124.1, 5.2, 23, 'F'),
        (5, 129.2, 5.3, 42, 'M'),
    ],
    ['id', 'weight', 'height', 'age', 'gender']
)
print('Count of rows: {0}'.format(df.count()))
print('Count of distinct rows: {0}'.format(df.distinct().count()))
# if these two numbers differ, drop these rows
```

```
df = df.dropDuplicates()
# check whether any duplicates in the data irrespective of ID
print('Count of ids: {0}'.format(df.count()))
print('Count of distinct ids: {0}'.format(df.select(
    [c for c in df.columns if c != 'id']
).distinct().count()))
# or use the .dropDuplicates(),
df = df.dropDuplicates(subset=[
    c for c in df.columns if c != 'id'
])
# to calculate the total and distinct number of IDs in one step
import pyspark.sql.functions as fn
df.agg(
    fn.count('id').alias('count'),
    fn.countDistinct('id').alias('distinct')
).show()
# give each row a unique ID:
df.withColumn('new_id',
fn.monotonically_increasing_id()).show()
```

- Missing observations

```
# example dataset:
df_miss = spark.createDataFrame(
[
    (1, 143.5, 5.6, 28, 'M', 100000),
    (2, 167.2, 5.4, 45, 'M', None),
    (3, None , 5.2, None, None, None),
    (4, 144.5, 5.9, 33, 'M', None),
    (5, 133.2, 5.7, 54, 'F', None),
    (6, 124.1, 5.2, None, 'F', None),
    (7, 129.2, 5.3, 42, 'M', 76000),
],
['id', 'weight', 'height', 'age', 'gender', 'income']
)
# find the number of missing observations
df_miss.rdd.map(
    lambda row: (row['id'], sum([c == None for c in row]))
).collect()
# produces [(1,0), (2,1), (3,4), (4,1), (5,1), (6,2), (7,0)]
df_miss.where('id == 3').show()
# percentage of missing, the * argument in .count() counts all rows
df_miss.agg(*[
    (1 - (fn.count(c) / fn.count('*'))).alias(c + '_missing') \
    for c in df_miss.columns
]).show()
# drop the 'income' feature, as most of its values are missing.
df_miss_no_income = df_miss.select([
    c for c in df_miss.columns if c != 'income'
])
df_miss_no_income.dropna(thresh=3).show()
# If you want to impute a mean, median, or other calculated value,
# you need to first calculate the value, create a dictionary with
# such values, and then pass it to the .fillna() method.
# records parameter in .to_dict() instructs to create a dictionary
means = df_miss_no_income.agg(
    *[fn.mean(c).alias(c)
    for c in df_miss_no_income.columns if c != 'gender']
).toPandas().to_dict('records')[0]
means['gender'] = 'missing'
df_miss_no_income.fillna(means).show()
```

- Outliers: observations that deviate significantly from the distribution of the rest samples. There are no outliers if all the values are roughly within the Q1−1.5IQR and Q3+1.5IQR range, where IQR is the interquartile range; the IQR is defined as a difference between the upper- and lower-quartiles, that is, the 75th percentile (the Q3) and 25th percentile (the Q1), respectively.

```
# example dataset:
df_outliers = spark.createDataFrame(
    [
        (1, 143.5, 5.3, 28),
        (2, 154.2, 5.5, 45),
        (3, 342.3, 5.1, 99),
        (4, 144.5, 5.5, 33),
        (5, 133.2, 5.4, 54),
        (6, 124.1, 5.1, 21),
        (7, 129.2, 5.3, 42),
    ],
    ['id', 'weight', 'height', 'age']
)
# Calculate the lower and upper cut off points for each feature.
# The first parameter specified is the name of the column, the
# second parameter can be either a number between 0 or 1 (where
# 0.5 means to calculated median) or a list, and the third parameter
# specifies the acceptable level of an error for each metric.
cols = ['weight', 'height', 'age']
bounds = {}
for col in cols:
    quantiles = df_outliers.approxQuantile(col, [0.25, 0.75], 0.05)
    IQR = quantiles[1] - quantiles[0]
    bounds[col] = [
        quantiles[0] - 1.5 * IQR, quantiles[1] + 1.5 * IQR
    ]
# flag outliers:
outliers = df_outliers.select(
    *['id'] + [
        (
            (df_outliers[c] < bounds[c][0]) |
            (df_outliers[c] > bounds[c][1])
```

```
        ).alias(c + '_o') for c in cols
    ]
)
outliers.show()
# lists the values significantly differing from the rest of the
# distribution:
df_outliers = df_outliers.join(outliers, on='id')
df_outliers.filter('weight_o').select('id', 'weight').show()
df_outliers.filter('age_o').select('id', 'age').show()
```

MLlib on RDDs

For example, to predict whether the 'INFANT_ALIVE_AT_REPORT' is either 1 or 0.

- Loading and transforming the data

```
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())
]
schema = typ.StructType([
    typ.StructField(e[0], e[1], False) for e in labels
])
births = spark.read.csv(
    'births_train.csv.gz', header=True, schema=schema
)
# specify recode dictionary:
recode_dictionary = {
    'YNU': {
        'Y': 1,
        'N': 0,
        'U': 0
    }
}
```

- Manual feature selection

```
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)
# specify 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)
# the recode function needs to be converted to a UDF on a DataFrame
rec_integer = func.udf(recode, typ.IntegerType())
# 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'))
# correcting the Yes/No/Unknown features
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])
# DataFrames can transform the features in bulk while
# selecting features.
births.select([
    'INFANT_NICU_ADMISSION',
    rec_integer(
        'INFANT_NICU_ADMISSION', func.lit('YNU')
    ).alias('INFANT_NICU_ADMISSION_RECODE')
]).take(5)
# transform all the YNU_cols in one go
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)
# check if we got it correctly:
births_transformed.select(YNU_cols[-5:]).show(5)
```

- Understanding the data

```
# calculate 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'
]
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))
    )
# For categorical variables, calculate the frequencies of 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])
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)
    )
```

- Correlations: identify collinear numeric 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])
            )
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

```
# for categorical features, run a Chi-square test to determine
# if there are significant differences.
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()
    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))
print(ln.Matrices.dense(3,2, [1,2,3,4,5,6]))
```

- Creating an RDD of LabeledPoints

```
# 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 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:]))
    )
```

- Splitting into training and testing

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

- Predicting infant survival by logistic regression with SGD

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

Here's what we got:

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

- Selecting only the most predictable features

```
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]))
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]))
```

- Predicting infant survival by logistic regression with SGD

```
from pyspark.mllib.tree import RandomForest
RF_model = RandomForest.trainClassifier(
    data=topFeatures_train, numClasses=2,
    categoricalFeaturesInfo={}, numTrees=6,
    featureSubsetStrategy='all', seed=666
)
# check how well or how bad our model performed:
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))
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
```

- Selecting only the most predictable 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))
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
```

The ML Package on DataFrames

- 3 main abstract classes: a Transformer, an Estimator, and a Pipeline.
- Predicting the chances of infant survival with ML

```
# load the data
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())
]
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
)
# convert to numeric values
import pyspark.ml.feature as ft
births = births \
    .withColumn('BIRTH_PLACE_INT', births['BIRTH_PLACE'] \
    .cast(typ.IntegerType()))
#  create Transformer:
encoder = ft.OneHotEncoder(
    inputCol='BIRTH_PLACE_INT', outputCol='BIRTH_PLACE_VEC'
)
#  create a single column with all the features collated together.
featuresCreator = ft.VectorAssembler(
    inputCols=[
        col[0] for col in labels[2:]
    ] + [encoder.getOutputCol()],
    outputCol='features'
)
# Creating an estimator
import pyspark.ml.classification as cl
logistic = cl.LogisticRegression(
    maxIter=10, regParam=0.01, labelCol='INFANT_ALIVE_AT_REPORT'
)
# Creating a pipeline
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[
    encoder, featuresCreator, logistic
])
# split dataset into training, validation, and testing datasets.
train, test, val = births.randomSplit([0.7, 0.2, 0.1], seed=666)
# Fitting the model
model = pipeline.fit(births_train)
# estimation
test_model = model.transform(births_test)
test_model.take(1)
# Evaluating the performance of the model
import pyspark.ml.evaluation as ev
evaluator = ev.BinaryClassificationEvaluator(
    rawPredictionCol='probability',
    labelCol='INFANT_ALIVE_AT_REPORT'
)
print(evaluator.evaluate(
    test_model, {evaluator.metricName: 'areaUnderROC'}
))
print(evaluator.evaluate(
    test_model, {evaluator.metricName: 'areaUnderPR'}
))
# save the Pipeline definition for later use
pipelinePath = './infant_oneHotEncoder_Logistic_Pipeline'
pipeline.write().overwrite().save(pipelinePath)
# load it up and use it straight away to .fit(...) and predict:
loadedPipeline = Pipeline.load(pipelinePath)
loadedPipeline.fit(births_train).transform(births_test).take(1)
# Saving the model
from pyspark.ml import PipelineModel
modelPath = './infant_oneHotEncoder_Logistic_PipelineModel'
model.write().overwrite().save(modelPath)
# load it up and use it straight away to predict:
loadedPipelineModel = PipelineModel.load(modelPath)
test_reloadedModel = loadedPipelineModel.transform(births_test)
```

- Parameter hyper-tuning
    - Grid search

```
import pyspark.ml.tuning as tune
# specify the list of parameters we want to loop through:
logistic = cl.LogisticRegression(
    labelCol='INFANT_ALIVE_AT_REPORT'
)
grid = tune.ParamGridBuilder().addGrid(
    logistic.maxIter, [2, 10, 50]
).addGrid(
    logistic.regParam, [0.01, 0.05, 0.3]
).build()
# comparing the models:
evaluator = ev.BinaryClassificationEvaluator(
    rawPredictionCol='probability',
    labelCol='INFANT_ALIVE_AT_REPORT'
)
# validation
cv = tune.CrossValidator(
    estimator=logistic, estimatorParamMaps=grid,
    evaluator=evaluator
)
```

```
# create a purely transforming Pipeline:
pipeline = Pipeline(stages=[encoder ,featuresCreator])
data_transformer = pipeline.fit(births_train)
# find the optimal combination of parameters for our model:
cvModel = cv.fit(data_transformer.transform(births_train))
# check if it performed better than 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'}
))
# What parameters does the best model have?
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]
```

- Train-validation splitting

```
selector = ft.ChiSqSelector(
    numTopFeatures=5,
    featuresCol=featuresCreator.getOutputCol(),
    outputCol='selectedFeatures',
    labelCol='INFANT_ALIVE_AT_REPORT'
)
logistic = cl.LogisticRegression(
    labelCol='INFANT_ALIVE_AT_REPORT',
    featuresCol='selectedFeatures'
)
pipeline = Pipeline(stages=[encoder, featuresCreator, selector])
data_transformer = pipeline.fit(births_train)
tvs = tune.TrainValidationSplit(
    estimator=logistic,
    estimatorParamMaps=grid,
    evaluator=evaluator
)
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'}
))
```

- NLP - related feature extractors

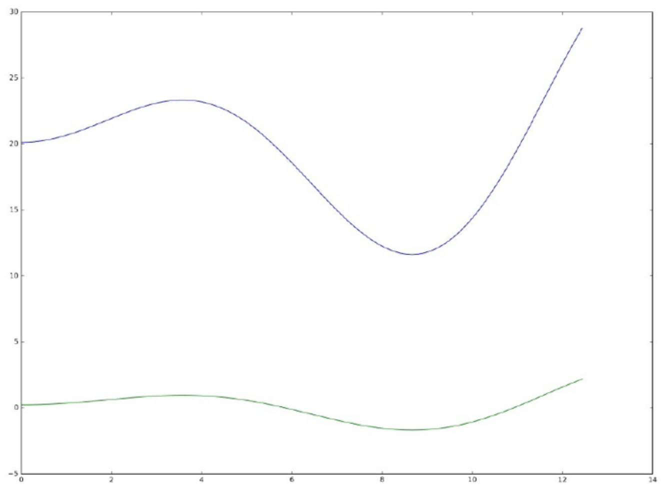
```
# text dataset
text_data = spark.createDataFrame(
    [
        ['''Machine learning can be applied to a wide variety of
        data types, such as vectors, text, images, and structured
        data. This API adopts the DataFrame from Spark SQL in order
        to support a variety of data types.'''],
        ...
        ['''Columns in a DataFrame are named. The code examples
        below use names such as "text," "features," and "label."''']
    ],
    ['input']
)
# tokenization
tokenizer = ft.RegexTokenizer(
    inputCol='input', outputCol='input_arr', pattern='\s+|[,.\"]'
)
# remove stopwords
stopwords = ft.StopWordsRemover(
    inputCol=tokenizer.getOutputCol(), outputCol='input_stop'
)
# build NGram model and the Pipeline:
ngram = ft.NGram(
    n=2, inputCol=stopwords.getOutputCol(), outputCol="nGrams"
)
pipeline = Pipeline(stages=[tokenizer, stopwords, ngram])
data_ngram = pipeline.fit(text_data).transform(text_data)
data_ngram.select('nGrams').take(1)
```

- Discretizing highly non-linear continuous variables into discrete buckets

```
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
# create a DataFrame
schema = typ.StructType([
    typ.StructField('continuous_var', typ.DoubleType(), False)
])
data = spark.createDataFrame(
```

```
    [[float(e), ] for e in y], schema=schema
)
# split continuous variable
discretizer = ft.QuantileDiscretizer(
    numBuckets=5, inputCol='continuous_var',
    outputCol='discretized'
)
# fit the data
data_discretized = discretizer.fit(data).transform(data)
# Standardizing continuous variables
vectorizer = ft.VectorAssembler(
    inputCols=['continuous_var'], outputCol= 'continuous_vec'
)
# normalization
normalizer = ft.StandardScaler(
    inputCol=vectorizer.getOutputCol(), outputCol='normalized',
    withMean=True, withStd=True
)
pipeline = Pipeline(stages=[vectorizer, normalizer])
data_standardized = pipeline.fit(data).transform(data)
```

Here's what the transformed data would look like:



GraphFrames

```
# Preparing your flights dataset
# Airline On-Time Performance and Causes of Flight Delays:
#   contains scheduled and actual departure and arrival times, and
#   delay causes as reported by US air carriers.
# Open Flights: Airports and airline data:
#   including the IATA code, airport name, and airport location.
# Set File Paths
tripdelaysFilePath =
    "/databricksdatasets/flights/departuredelays.csv"
airportsnaFilePath =
    "/databricks-datasets/flights/airportcodes-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")
# 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()
# 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")
# Only include airports with at least 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()
# 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_ge o"
)
departureDelays_geo.cache()
# Review the top 10 rows of the `departureDelays_geo` DataFrame
departureDelays_geo.show(10)
# Building the graph
# Note, ensure you have already installed 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()
display(tripEdges)
#  create a GraphFrame using the GraphFrame command:
tripGraph = GraphFrame(tripVertices, tripEdges)
# Executing simple queries
print "Airports: %d" % tripGraph.vertices.count()
print "Trips: %d" % tripGraph.edges.count()
# Determining the longest delay in this dataset
tripGraph.edges.groupBy().max("delay")
# Determining the number of delayed versus on-time/early flights
print "On-time / Early Flights: %d" % tripGraph.edges.filter("delay
<= 0").count()
print "Delayed Flights: %d" % tripGraph.edges.filter("delay >
0").count()
# What flights departing Seattle likely to have significant delays?
tripGraph.edges\
    .filter("src = 'SEA' and delay > 0")\
    .groupBy("src", "dst")\
    .avg("delay")\
    .sort(desc("avg(delay)"))\
    .show(5)
# What states have significant delays departing from Seattle?
# States with the longest cumulative delays (with individual
# delays > 100 minutes) (origin: Seattle)
display(tripGraph.edges.filter("src = 'SEA' and delay > 100"))
# Understanding vertex degrees
display(tripGraph.degrees.sort(desc("degree")).limit(20))
display(tripGraph.inDegrees.sort(desc("inDegree")).limit(20))
display(tripGraph.outDegrees.sort(desc("outDegree")).limit(20))
# Determining the top transfer airports by the ratio
# Calculate the inDeg (flights into the airport) and
```

```
# outDeg (flights leaving the airport)
inDeg = tripGraph.inDegrees
outDeg = tripGraph.outDegrees
# 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")
# List out the top 10 transfer city airports
display(transferAirports.orderBy("degreeRatio").limit(10))
# Understanding motifs
# 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)
# Determining airport ranking using PageRank
# 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)
)
# Determining the most popular non-stop flights
# 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))
# Using Breadth-First Search
# 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)
# 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)
# 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)
# Display most popular layover cities by descending count
display(
    filteredPaths.groupBy("v1.id", "v1.City") \
        .count().orderBy(desc("count")).limit(10)
)
# Visualizing flights using D3
%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]
)
```

TensorFrames

- Matrix multiplication using constants

```
# Import TensorFlow
import tensorflow as tf
# Setup the matrix
# c1: 1x3 matrix
# c2: 3x1 matrix
c1 = tf.constant([[3., 2., 1.]])
c2 = tf.constant([[-1.], [2.], [1.]])
# tensors in the form of numpy ndarray or
# tensorflow::Tensor interfaces in C/C++
# m3: matrix multiplication (m1 x m3)
mp = tf.matmul(c1, c2)
# Launch the default graph
s = tf.Session()
# run: Execute the ops in graph
r = s.run(mp)
print(r)
# Close the Session when completed
s.close()
```

- Matrix multiplication using placeholders for tensors of different sizes/shape

```
# Setup placeholder for your model
# t1: placeholder tensor
# t2: placeholder tensor
t1 = tf.placeholder(tf.float32)
t2 = tf.placeholder(tf.float32)
# t3: matrix multiplication (m1 x m3)
tp = tf.matmul(t1, t2)
# Running the model
# Define input matrices
m1 = [[3., 2., 1.]]
m2 = [[-1.], [2.], [1.]]
# Execute the graph within a session
with tf.Session() as s:
    print(s.run([tp], feed_dict={t1:m1, t2:m2}))
# setup input matrices
m1 = [[3., 2., 1., 0.]]
m2 = [[-5.], [-4.], [-3.], [-2.]]
# Execute the graph within a session
with tf.Session() as s:
    print(s.run([tp], feed_dict={t1:m1, t2:m2}))
```

- TensorFrames – quick start

```
# The version we're using in this notebook
$SPARK_HOME/bin/pyspark --packages \
tjhunter:tensorframes:0.2.2-s_2.10
# Or use the latest version
$SPARK_HOME/bin/pyspark –packages \
databricks:tensorframes:0.2.3-s_2.10
```

- Using TensorFlow to add a constant to an existing column

```
# Import TensorFlow, TensorFrames, and Row
import tensorflow as tf
import tensorframes as tfs
from pyspark.sql import Row
# Create RDD of floats and convert into DataFrame `df`
rdd = [Row(x=float(x)) for x in range(10)]
df = sqlContext.createDataFrame(rdd)
df.show()
# Executing the Tensor graph
# Run TensorFlow program executes:
# The 'op' performs the addition (i.e. 'x' + '3')
# Place the data back into a DataFrame
with tf.Graph().as_default() as g:
    # The placeholder that corresponds to column 'x'. The shape of
```

```
    # the placeholder is automatically inferred from the DataFrame.
    x = tfs.block(df, "x")
    # The output that adds 3 to x
    z = tf.add(x, 3, name='z')
    # The resulting `df2` DataFrame
    df2 = tfs.map_blocks(z, df)
# Note that 'z' is the tensor output from the 'tf.add' operation
print z
## Output
Tensor("z:0", shape=(?,), dtype=float64)
```

- Blockwise reducing operations example

```
# Build a DataFrame of vectors
data = [Row(y=[float(y), float(-y)]) for y in range(10)]
df = sqlContext.createDataFrame(data)
df.show()
# Analysing the DataFrame
# Print the information gathered by TensorFlow
tfs.print_schema(df)
## Output
root
|-- y: array (nullable = true) double[?,?]
# Because the dataframe contains vectors, we need to analyze it
# first to find the dimensions of the vectors.
df2 = tfs.analyze(df)
# The information gathered by TF can be printed to check the content
tfs.print_schema(df2)
## Output
root
|-- y: array (nullable = true) double[?,2]
# Computing elementwise sum (.reduce_sum) and min (.reduce_min)
# Note: First, let's make a copy of the 'y' column.
# This is an inexpensive operation in Spark 2.0+
df3 = df2.select(df2.y, df2.y.alias("z"))
# Execute the Tensor Graph
with tf.Graph().as_default() as g:
    # The placeholders.
    # Note the special name that end with '_input':
    y_input = tfs.block(df3, 'y', tf_name="y_input")
    z_input = tfs.block(df3, 'z', tf_name="z_input")
    # Perform elementwise sum and minimum
    y = tf.reduce_sum(y_input, [0], name='y')
    z = tf.reduce_min(z_input, [0], name='z')
# The resulting dataframe
(data_sum, data_min) = tfs.reduce_blocks([y, z], df3)
# The finalresults are numpy arrays:
print "Elementwise sum: %s and min: %s " % (data_sum, data_min)
## Output
Elementwise sum: [ 45. -45.] and minimum: [ 0. -9.]
```

Polyglot Persistence with Blaze

- Working with NumPy arrays

```
import numpy as np
simpleArray = np.array([[1,2,3], [4,5,6]])
simpleData_np = bl.Data(simpleArray)
simpleData_np.peek()
simpleData_np[0]
# transpose your DataShape: but the name of the column becomes None
simpleData_np.T[0]
# named columns => calling the column by its name
simpleData_np = bl.Data(simpleArray, fields=['a', 'b', 'c'])
simpleData_np['b']
```

- Working with pandas' DataFrame

```
import pandas as pd
simpleDf = pd.DataFrame([[1,2,3], [4,5,6]], columns=['a','b','c'])
# transform it into a DataShape:
simpleData_df = bl.Data(simpleDf)
# retrieve data
simpleData_df['a']
```

- Working with files

```
import odo
traffic = bl.Data('../Data/TrafficViolations.csv')
print(traffic.fields)
traffic_gz = bl.Data('../Data/TrafficViolations.csv.gz')
# produces the same results, it takes more time to retrieve from
the archived file
traffic.head(2)
traffic_gz.head(2)
# saves the data into a GZip archive
for year in traffic.Stop_year.distinct().sort():
    odo.odo(
        traffic[traffic.Stop_year == year],
        '../Data/Years/TrafficViolations_{0}.csv.gz'.format(year)
    )
# read from multiple files using the asterisk character *:
traffic_multiple = bl.Data(
    '../Data/Years/TrafficViolations_*.csv.gz'
)
traffic_multiple.head(2)
# you can read data from JSON, Excel files, HDFS, or bcolz files.
```

- Interacting with relational databases

```
traffic_psql = bl.Data(
    'postgresql://{0}:{1}@localhost:5432/drabast::traffic'\
    .format('<your_username>', '<your_password>')
)
traffic_2016 = traffic_psql[traffic_psql['Year'] == 2016]
# Drop commands
```

```
# odo.drop('sqlite:///traffic_local.sqlite::traffic2016')
# odo.drop('postgresql://{0}:
{1}@localhost:5432/drabast::traffic'\
.format('<your_username>', '<your_password>'))
# Save to SQLite
odo.odo(
    traffic_2016, 'sqlite:///traffic_local.sqlite::traffic2016'
)
# Save to PostgreSQL
odo.odo(
    traffic_2016,
    'postgresql://{0}:{1}@localhost:5432/drabast::traffic'\
    .format('<your_username>', '<your_password>')
)
# read from SQLite
traffic_sqlt = bl.Data(
    'sqlite:///traffic_local.sqlite::traffic2016'
)
```

- Interacting with the MongoDB database

```
# read from MongoDB
traffic_mongo = bl.Data(
    'mongodb://localhost:27017/packt::traffic'
)
```

- Data operations

```
# Accessing a single column
traffic.Year.head(2)
# selection of more than one column at a time:
(traffic[['Location',      'Year',      'Accident',      'Fatal',
'Alcohol']].head(2))
# the equivalent SQL code would be:
SELECT *
FROM traffic
LIMIT 2z
```

- Symbolic transformations: Blaze can operate symbolically using the .symbol() method; the first argument specifies a symbolic name of the transformation, and the second argument specifies the schema

```
schema_example = bl.symbol(
    'schema_exampl', '{id: int, name: string}'
)
# reuse the schema by using traffic.dshape
traffic_s = bl.symbol('traffic', traffic.dshape)
traffic_2013 = traffic_s[traffic_s['Stop_year'] == 2013][
['Stop_year', 'Arrest_Type','Color', 'Charge']]
traffic_pd = pd.read_csv('../Data/TrafficViolations.csv')
# perform the computation using the .compute() method:
#   the first argument specifies the transformation object and
#   the second parameter is the data
bl.compute(traffic_2013, traffic_pd).head(2)
# You can also pass a list of lists or a list of NumPy arrays
# using the .values attribute
bl.compute(traffic_2013, traffic_pd.values)[0:2]
```

- Operations on columns

```
# mathematical operations to be done on numeric columns.
traffic['Stop_year'].distinct().sort()
# An equivalent syntax for pandas would be as follows:
traffic['Stop_year'].unique().sort()
# For SQL, use the following code:
SELECT DISTINCT Stop_year
FROM traffic
# mathematical transformations/arithmetic to the columns.
traffic['Stop_year'].head(2) - 2000
# For SQL, the equivalent would be:
SELECT Stop_year - 2000 AS Stop_year
FROM traffic
# more complex mathematical operations (for example, log or pow)
bl.log(traffic['Stop_year']).head(2)
```

- Reducing data

```
# reduction methods, such as .mean(), .std, or .max()
traffic['Stop_year'].max()
# for SQL the same could be done with the following code:
SELECT MAX(Stop_year) AS Stop_year_max
FROM traffic
# the .transform() method
traffic = bl.transform(
    traffic, Age_of_car = traffic.Stop_year - traffic.Year
)
traffic.head(2)
# An equivalent operation in pandas
traffic['Age_of_car'] = traffic.apply(
    lambda row: row.Stop_year - row.Year, axis = 1
)
# For SQL you can use the following code:
SELECT *, Stop_year - Year AS Age_of_car
FROM traffic
# perform a group by operation using the .by() operation:
bl.by(
    traffic['Fatal'],
    Fatal_AvgAge=traffic.Age_of_car.mean(),
    Fatal_Count =traffic.Age_of_car.count()
)
# For pandas, an equivalent would be as follows:
traffic.groupby('Fatal')['Age_of_car']\
    .agg({
        'Fatal_AvgAge': np.mean, 'Fatal_Count': np.count_nonzero
    })
# For SQL, it would be as follows:
```

```
SELECT Fatal
    , AVG(Age_of_car) AS Fatal_AvgAge
    , COUNT(Age_of_car) AS Fatal_Count
FROM traffic
GROUP BY Fatal
```

- Joins

```
# Joining two DataShapes using .join()
violation = traffic[[
    'Stop_month','Stop_day','Stop_year',
    'Stop_hr','Stop_min','Stop_sec','Violation_Type'
]]
belts = traffic[[
    'Stop_month','Stop_day','Stop_year',
    'Stop_hr','Stop_min','Stop_sec','Belts'
]]
violation_belts = bl.join(
    violation, belts, [
        'Stop_month','Stop_day','Stop_year',
        'Stop_hr','Stop_min','Stop_sec'
    ]
)
bl.by(
    violation_belts[['Violation_Type', 'Belts']],
    Violation_count=violation_belts.Belts.count()
).sort('Violation_count', ascending=False)
# The same could be achieved in pandas with the following code:
violation.merge(
    belts,
    on=[
        'Stop_month','Stop_day','Stop_year',
        'Stop_hr','Stop_min','Stop_sec'
    ]
).groupby(['Violation_type','Belts']).agg({
    'Violation_count': np.count_nonzero
}) \
.sort('Violation_count', ascending=False)
# With SQL, you would use the following snippet:
SELECT innerQuery.* FROM (
    SELECT a.Violation_type, b.Belts, COUNT() AS Violation_count
    FROM violation AS a
    INNER JOIN belts AS b
    ON a.Stop_month = b.Stop_month
        AND a.Stop_day = b.Stop_day
        AND a.Stop_year = b.Stop_year
        AND a.Stop_hr = b.Stop_hr
        AND a.Stop_min = b.Stop_min
        AND a.Stop_sec = b.Stop_sec
    GROUP BY Violation_type, Belts
) AS innerQuery
ORDER BY Violation_count DESC
```

Structured Streaming

- Spark Streaming word count application using DStreams and Unix/Linux nc command (read and write data across network connection.

```
# Create a local SparkContext and Streaming Contexts
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
# Create sc with two working threads
sc = SparkContext("local[2]", "NetworkWordCount")
# Create local StreamingContextwith batch interval of 1 second
ssc = StreamingContext(sc, 1)
# Create DStream that connects to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
# Split lines into words
words = lines.flatMap(lambda line: line.split(" "))
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
# Print the first ten elements of each RDD in this DStream
wordCounts.pprint()
# Start the computation
ssc.start()
# Wait for the computation to terminate with <Ctrl><C>
ssc.awaitTermination()
```

To start the nc command, from one of your terminals:

```
$ nc -lk 9999
```

- Global aggregations: calculating a stateful aggregation beyond batch interval with windowing

```
# inserting any new chunks of data led to slow streaming
# performance with the ever-increasing scheduling delays.
sqlContext.sql(
    "insert into meetup_stream select * from meetup_stream_json"
)
# creating global aggregations via UpdateStateByKey (Spark 1.5)
# the performance is proportional to the size of the state.
# Create a local SparkContext and Streaming Contexts
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
# Create sc with two working threads
sc = SparkContext("local[2]", "StatefulNetworkWordCount")
# Create local StreamingContext with batch interval of 1 sec
ssc = StreamingContext(sc, 1)
# Create checkpoint for local StreamingContext: ensure that Spark
# Streaming is fault tolerant
ssc.checkpoint("checkpoint")
```

```
# Define updateFunc via UpdateStateByKey: sum of (key, value) pairs
def updateFunc(new_values, last_sum):
    return sum(new_values) + (last_sum or 0)
# Create DStream that connects to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
# Calculate running counts
running_counts = lines.flatMap(lambda line: line.split(""))\
    .map(lambda word: (word, 1)).updateStateByKey(updateFunc)
# Print the first ten elements of each RDD generated in this
# stateful DStream to the console
running_counts.pprint()
# Start the computation
ssc.start()
# Wait for the computation to terminate
ssc.awaitTermination()
# creating global aggregations via mapWithState (Spark 1.6)
# The performance is proportional to the size of the batch.
```

- Structured Streaming (Spark 2.0)
    - batch aggregation

```
# reads a data stream from S3 and saves it to a MySQL database:
logs = spark.read.json('s3://logs')
logs.groupBy(logs.UserId).agg(sum(logs.Duration)) \
    .write.jdbc('jdbc:mysql//...')
# continous aggregation:
logs = spark.readStream.json('s3://logs').load()
sq = logs.groupBy(logs.UserId).agg(sum(logs.Duration)) \
    .writeStream.format('json').start()
# Will return true if the `sq` stream is active
sq.isActive
# Will terminate the `sq` stream
sq.stop()
```

- DataFrames code

```
# Import the necessary classes and create a local SparkSession
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
spark = SparkSession.builder \
    .appName("StructuredNetworkWordCount").getOrCreate()
# Create DataFrame representing the stream of input lines
# from connection to localhost:9999
lines = spark\
    .readStream\
    .format('socket')\
    .option('host', 'localhost')\
    .option('port', 9999)\
    .load()
# Split the lines into words
words = lines.select(
    explode(
        split(lines.value, ' ')
    ).alias('word')
)
# Generate running word count
wordCounts = words.groupBy('word').count()
```

- output this data to the console

```
# Start running the query that prints the
# running counts to the console
query = wordCounts\
    .writeStream\
    .outputMode('complete')\
    .format('console')\
    .start()
# Await Spark Streaming termination
query.awaitTermination()
```

- run nc job in the first terminal:

```
$ nc -lk 9999
```

Packaging Spark Applications

- submitting jobs to Spark using the spark-submit script.

```
$ spark-submit [options] <python file> [app arguments]
```

--master: Parameter used to set the URL of the master (head) node: Local, local[n], local[*], Spark standalone cluster spark://host:port, mesos://host:port, and Yarn.
--deploy-mode: client or cluster
--name: Name of your application.
--py-files: Comma-delimited list of .py, .egg or .zip files to include for Python apps.
--files: comma-delimited list of files.
--conf: configure of your app dynamically from the command line parameters, or specified in the conf/spark-defaults.conf file. The syntax is <Spark property>=<value for the property>.
--properties-file: File with a configuration having the same set of properties as the conf/spark-defaults.conf file.
--driver-memory: default is 1,024M.
--executor-memory: default is 1G.
--help:
--verbose:
--version:
--supervise:
--kill:
--status:

- Deploying the app programmatically

```python
# Configuring your SparkSession and creating SparkSession
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName('CalculatingGeoDistances') \
    .getOrCreate()
print('Session created')
```

- Modularizing code
    - Structure of the Python package:

```
additionalCode/
    setup.py
    utilities/
        __init__.py
        base.py
        converters/
            __init__.py
            distance.py
        geoCalc.py
```

    - The setup.py file in our case looks as follows:

```python
from setuptools import setup
setup(
    name='PySparkUtilities',
    version='0.1dev',
    packages=['utilities', 'utilities/converters'],
    license='''
        Creative Commons
        Attribution-Noncommercial-Share Alike license''',
    long_description='''
        An example of how to package code for PySpark'''
)
```

    - The __init__.py file in the utilities folder has the following code:

```python
from .geoCalc import geoCalc
__all__ = ['geoCalc','converters']
```

    - Calculating the distance in miles between two points (latitude and longitude) on a map (Cartesian coordinates) using the Haversine formula with calculateDistance() in the geoCalc.py file.
    - Converting distance units: any class implemented as a converter should expose the same interface and implement the convert() method. E.g.,

```python
from abc import ABCMeta, abstractmethod
class BaseConverter(metaclass=ABCMeta):
@staticmethod
@abstractmethod
def convert(f, t):
    raise NotImplementedError
```

    - Building an egg

```
$ python setup.py bdist_egg
```

    - User defined functions in Spark

```python
import utilities.geoCalc as geo
from utilities.converters import metricImperial
getDistance = func.udf(
    lambda lat1, long1, lat2, long2:
        geo.calculateDistance(
            (lat1, long1), (lat2, long2)
        )
    )
convertMiles = func.udf(lambda m:
metricImperial.convert(str(m) + ' mile', 'km'))
```

- calculate the distance and convert it to miles:

```python
# Using the .withColumn() method we create additional columns.
uber = uber.withColumn(
    'miles',
    getDistance(
        func.col('pickup_latitude'),
        func.col('pickup_longitude'),
        func.col('dropoff_latitude'),
        func.col('dropoff_longitude')
    )
)
uber = uber.withColumn(
    'kilometers',
    convertMiles(func.col('miles'))
)
```

- Submitting a job

```
$ ./launch_spark_submit.sh \
--master local[4] \
--py-files calculatingGeoDistance.py \
    additionalCode/dist/PySparkUtilities-0.1.dev0-py3.5.egg
```

- configured Spark instance to run Jupyter and automated it with the launch_spark_submit.sh script:

```bash
#!/bin/bash
unset PYSPARK_DRIVER_PYTHON
spark-submit $*
export PYSPARK_DRIVER_PYTHON=jupyter
```

- Monitoring execution: you can switch between the Jobs view or the Stages view to track all the stages that are executed.