

Spark SQL: CSV

1. Depending on your version of Scala, start the pyspark shell with a packages command line argument.

```
1
2 $SPARK_HOME/bin/pyspark --packages com.databricks:spark-csv_2.10:1.3.0
3
```

```
1
2 $SPARK_HOME/bin/pyspark --packages com.databricks:spark-csv_2.11:1.3.0
3
```

2. Using the available sqlContext from the shell load the CSV read, format, option and load functions

Spark SQL CSV Examples with Python

```
1 >>> df = sqlContext.read.format('com.databricks.spark.csv').options(header='true', inferschema='true')
2 >>> df.load('Uber-Jan-Feb-FOIL.csv')
```

In the above code, we are specifying the desire to use *com.databricks.spark.csv* format from the package we passed to the shell in step 1. “header” set to true signifies the first row has column names. “inferSchema” instructs Spark to attempt to infer the schema of the CSV and finally load function passes in the path and name of the CSV source file. In this example, we can tell the Uber-Jan-Feb-FOIL.csv file is in the same directory as where pyspark was launched.

3. Register a temp table

Python

```
1
2 >>> df.registerTempTable("uber")
3
```

4. We're now ready to query using SQL such as finding the distinct NYC Uber bases in the CSV

Python

```
1
2 >>> distinct_bases = sqlContext.sql("select distinct dispatching_base_number from uber")
3 >>> for b in distinct_bases.collect(): print b
4 Row(dispatching_base_number=u'B02598')
5 Row(dispatching_base_number=u'B02764')
6 Row(dispatching_base_number=u'B02765')
7 Row(dispatching_base_number=u'B02617')
8 Row(dispatching_base_number=u'B02682')
9 Row(dispatching_base_number=u'B02512')
10
```

5. It might be handy to know the schema

```
1
```

```

2 >>> df.printSchema()
3
4 root
5 |-- dispatching_base_number: string (nullable = true)
6 |-- date: string (nullable = true)
7 |-- active_vehicles: integer (nullable = true)
8 |-- trips: integer (nullable = true)
9

```

But, let's try some more advanced SQL, such as determining which Uber bases is the busiest based on number of trips

Python

```

1
2 >>> sqlContext.sql("""select distinct(`dispatching_base_number`),
3                               sum(`trips`) as cnt from uber group by `dispatching_base_number`
4                               order by cnt desc""").show()
5
6
7 +-----+-----+
8 |dispatching_base_number| cnt|
9 +-----+-----+
10 |          B02764|1914449|
11 |          B02617| 725025|
12 |          B02682| 662509|
13 |          B02598| 540791|
14 |          B02765| 193670|
15 |          B02512|  93786|
16 +-----+-----+
17

```

Or the 5 busiest days based on number of trips in the time range of the data:

```

>>> sqlContext.sql("""select distinct(`date`),
1                               sum(`trips`) as cnt from uber group by `date`
2                               order by cnt desc limit 5""").show()
3
4
5 +-----+-----+
6 |  date| cnt|
7 +-----+-----+
8 |2/20/2015|100915|
9 |2/14/2015|100345|
10 |2/21/2015| 98380|
11 |2/13/2015| 98024|
12 |1/31/2015| 92257|
13 +-----+-----+
14
15
16

```

Spark SQL: JSON

1. Start pyspark

```
$SPARK_HOME/bin/pyspark
```

2. Load a JSON file which comes with Apache Spark distributions by default. We do this by using the `jsonFile` function from the provided `sqlContext`.

```
1. Start pyspark
$SPARK_HOME/bin/pyspark
2. Load a JSON file which comes with Apache Spark distributions by default. We do this by
using the jsonFile function from the provided sqlContext.
```

2. Load a JSON file which comes with Apache Spark distributions by default. We do this by using the `jsonFile` function from the provided `sqlContext`.

2. Load a JSON file which comes with Apache Spark distributions by default. We do this by using the `jsonFile` function from the provided `sqlContext`.

```
Python pyspark Apache Spark JSON
Python
1
2 Welcome to
3
4  _ _ _ _ _
5  / \ / \ / \ / \ / \ / \
6  / \ / \ / \ / \ / \ / \ version 1.6.1
7  / \
8
9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16
```

[illegible]

```

1
2 Welcome to
3
4      _ _ _ _ _
5     / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \
6    / _ \| | | | | | | | | | | | | | | | | | | | | | | | | | version 1.6.1
7     \|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
8
9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16

```

```

2 Welcome to
3
4      / _ |   / _ |   / _ |   // 
5     \ V /   \ V /   \ V /   ' / 
6    / _ |   . ^ _ , / _ | / _ | \ version 1.6.1
7     \|_|
8
9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16
```

[illegible][illegible]

```

5      \V_V_\'/_/'\
6      /_/.^_/_/_/_/_ version 1.6.1
7      /\
8
9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16

```

```
6 \_/_/.\_\_/_/_\_ version 1.6.1
7 \_/_
8
9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16
```

```

7      /_/_
8
9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16

```

```

8
9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16

```

```

9 Using Python version 2.7.11
10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16

```

```

10 SparkContext available as sc, HiveContext available as sqlContext.
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16

```

```
11 >>> people = sqlContext.read.json("examples/src/main/resources/people.json")
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16
```

```
12 >>> people.printSchema()
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16
```

```
13 root
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16
```

```
14 |-- age: long (nullable = true)
15 |-- name: string (nullable = true)
16
```

```
15 |-- name: string (nullable = true)
16
```

16

3. Register the data as a temp table to ease our future SQL queries

```
Python Apache Spark registerTempTable
1
2 >>> people.registerTempTable("people")
3
```

```
Python Apache Spark registerTempTable
1
2 >>> people.registerTempTable("people")
3
```

```
1
2 >>> people.registerTempTable("people")
3
```

```
2 >>> people.registerTempTable("people")
3
```

4. Now, we can run some SQL

python json sql spark

```
>>> sqlContext.sql("select name from people").show()
1  +-----+
2  |  name|
3  +-----+
4  |Michael|
5  |  Andy|
6  | Justin|
7  +-----+
8
9
10
```

```

1 +-----+
2 |  name|
3 +-----+
4 |Michael|
5 |  Andy|
6 | Justin|
7 +-----+
8
9
10

```

```

2 | name|
3 +-----+
4 |Michael|
5 |  Andy|
6 | Justin|
7 +-----+
8
9
10

```

```

3 | +-----+
4 | Michael|
5 |   Andy|
6 |  Justin|
7 | +-----+
8
9
10

```

```

4 |Michael|
5 |  Andy|
6 |Justin|
7 +-----+
8
9
10

```

```

5 |   Andy|
6 | Justin|
7 | +-----+
8 |
9 |
10|

```

6 | Justin|
7 |-----+
8 |
9 |
10 |

7 +-----+

8

9

10

10

1. Start pyspark

2. Load the JSON using the Spark Context *wholeTextFiles* method which produces a tuple RDD whose 1st element is a filename and the 2nd element is the data with lines separated by whitespace. We use *map* to create the new RDD using the 2nd element of the tuple.

```
1
2 >>> jsonRDD = sc.wholeTextFiles("2014-world-cup.json").map(lambda x: x[1])
3
```

3. Then, we need to prepare this RDD so it can be parsed by sqlContext. Let's remove the whitespace

```
1
2 >>> import re
3 >>> js = jsonRDD.map(lambda x: re.sub(r"\s+", "", x, flags=re.UNICODE))
4
```

4. We're now able to consume the RDD using *jsonRDD* of *sqlContext*

```
1
2 >>> wc_players = sqlContext.jsonRDD(js)
3
```

5. Let's register the table and run a query

```
1
2 >>> wc_players.registerTempTable("players")
3 >>> sqlContext.sql("select distinct Team from players").show()
4 +-----+
5 |      Team|
6 +-----+
7 |    Mexico|
8 |   Portugal|
9 |   Colombia|
10 | SouthKorea|
11 | Netherlands|
12 |    Belgium|
13 |    Chile|
14 |    Brazil|
15 |BosniaandHerzegovina|
16 |   IvoryCoast|
17 |   Cameroon|
18 |   England|
19 |   Croatia|
20 | Argentina|
21 |   Algeria|
22 |    Ghana|
23 |    Iran|
24 |   Nigeria|
25 |    Russia|
26 |    France|
27 +-----+
28
```

Spark SQL: mysql JDBC

MySQL JDBC driver (download available <https://dev.mysql.com/downloads/connector/j/>)

1. Start the pyspark shell with `--jars` argument

```
$SPARK_HOME/bin/pyspark --jars mysql-connector-java-5.1.38-bin.jar
```

This example assumes the mysql connector jdbc jar file is located in the same directory as where you are calling spark-shell. If it is not, you can specify the path location such as:

```
$SPARK_HOME/bin/pyspark --jars /home/example/jars/mysql-connector-java-5.1.38-bin.jar
```

2. Once the shell is running, let's establish connection to mysql db and read the "trips" table:

Spark SQL Python with mySQL (JDBC)

```
1 Welcome to
2
3      _ _ _ _ _
4     / V _ V _ ' _ / ' _
5    / _ / . ^ _ , / / _ ^ \ version 1.6.1
6     / _/
7
8 Using Python version 2.7.11
9 SparkContext available as sc, HiveContext available as sqlContext.
10 >>> dataframe_mysql = sqlContext.read.format("jdbc").option("url", "jdbc:mysql://localhost/uber"
11 ).option("driver", "com.mysql.jdbc.Driver").option("dbtable", "trips").option("user", "root").option
12 ("password", "root").load()
```

Change the mysql url and user/password values in the above code appropriate to your environment.

3. Let's confirm the dataframe by show the schema of the table

Python Spark SQL example

Python

```
1
2 >>> dataframe_mysql.show()
3 +-----+-----+-----+-----+
4 |dispatching_base_number|  date|active_vehicles|trips|
5 +-----+-----+-----+-----+
6 |          B02512|1/1/2015|          190| 1132|
7 |          B02765|1/1/2015|          225| 1765|
8 |          B02764|1/1/2015|        3427|29421|
9 |          B02682|1/1/2015|          945| 7679|
10 |          B02617|1/1/2015|        1228| 9537|
11 |          B02598|1/1/2015|          870| 6903|
12 |          B02598|1/2/2015|          785| 4768|
13 |          B02617|1/2/2015|        1137| 7065|
14 |          B02512|1/2/2015|          175|   875|
15 |          B02682|1/2/2015|          890| 5506|
16 |          B02765|1/2/2015|          196| 1001|
17 |          B02764|1/2/2015|       3147|19974|
18 |          B02765|1/3/2015|          201| 1526|
```

```

19 |          B02617|1/3/2015|      1188|10664|
20 |          B02598|1/3/2015|      818| 7432|
21 |          B02682|1/3/2015|      915| 8010|
22 |          B02512|1/3/2015|      173| 1088|
23 |          B02764|1/3/2015|     3215|29729|
24 |          B02512|1/4/2015|      147|  791|
25 |          B02682|1/4/2015|      812| 5621|
26 +-----+-----+-----+-----+
27

```

3. Register the data as a temp table for future SQL queries

```

1
2 >>> dataframe_mysql.registerTempTable("trips")
3

```

4. We are now in position to run some SQL such as

Spark SQL mysql JDBC example query

```

1
2 >>> sqlContext.sql("select * from trips where dispatching_base_number like '%2512%'").show()
3 +-----+-----+-----+-----+
4 |dispatching_base_number|  date|active_vehicles|trips|
5 +-----+-----+-----+-----+
6 |          B02512|1/1/2015|      190| 1132|
7 |          B02512|1/2/2015|      175|  875|
8 |          B02512|1/3/2015|      173| 1088|
9 |          B02512|1/4/2015|      147|  791|
10 |          B02512|1/5/2015|      194|  984|
11 |          B02512|1/6/2015|      218| 1314|
12 |          B02512|1/7/2015|      217| 1446|
13 |          B02512|1/8/2015|      238| 1772|
14 |          B02512|1/9/2015|      224| 1560|
15 |          B02512|1/10/2015|     206| 1646|
16 |          B02512|1/11/2015|     162| 1104|
17 |          B02512|1/12/2015|     217| 1399|
18 |          B02512|1/13/2015|     234| 1652|
19 |          B02512|1/14/2015|     233| 1582|
20 |          B02512|1/15/2015|     237| 1636|
21 |          B02512|1/16/2015|     234| 1481|
22 |          B02512|1/17/2015|     201| 1281|
23 |          B02512|1/18/2015|     177| 1521|
24 |          B02512|1/19/2015|     168| 1025|
25 |          B02512|1/20/2015|     221| 1310|
26 +-----+-----+-----+-----+
27

```

Spark GraphX: PageRank algorithm implementation

Churn prediction using Spark MLlib

Churn prediction is big business. It minimizes customer defection by predicting which customers are likely to cancel a subscription to a service.

[churn-80](#) and [churn-20](#). The two sets are from the same batch, but have been split by an 80/20 ratio. We'll use the larger set for training and cross-validation purposes, and the smaller set for final testing and model performance evaluation.

You might need additional Python packages, such as **Pandas**.

```
CV_data = sqlContext.read.load('./data/churn-bigml-80.csv',format='com.databricks.spark.csv',header='true',inferSchema='true')
```

```
final_test_data = sqlContext.read.load('./data/churn-bigml-20.csv',format='com.databricks.spark.csv', header='true',inferSchema='true')
```

```
CV_data.cache()
```

```
CV_data.printSchema()
```

```
root
```

```
|-- State: string (nullable = true)
|-- Account length: integer (nullable = true)
|-- Area code: integer (nullable = true)
|-- International plan: string (nullable = true)
|-- Voice mail plan: string (nullable = true)
|-- Number vmail messages: integer (nullable = true)
|-- Total day minutes: double (nullable = true)
|-- Total day calls: integer (nullable = true)
|-- Total day charge: double (nullable = true)
|-- Total eve minutes: double (nullable = true)
|-- Total eve calls: integer (nullable = true)
|-- Total eve charge: double (nullable = true)
|-- Total night minutes: double (nullable = true)
|-- Total night calls: integer (nullable = true)
|-- Total night charge: double (nullable = true)
|-- Total intl minutes: double (nullable = true)
|-- Total intl calls: integer (nullable = true)
|-- Total intl charge: double (nullable = true)
|-- Customer service calls: integer (nullable = true)
|-- Churn: string (nullable = true)
```

```
pd.DataFrame(CV_data.take(5), columns=CV_data.columns).transpose()
```

0	1	2	3	4	
State	KS	OH	NJ	OH	OK
Account length	128	107	137	84	75
Area code	415	415	415	408	415
International plan	No	No	No	Yes	Yes
Voice mail plan	Yes	Yes	No	No	No
Number vmail messages	25	26	0	0	0
Total day minutes	265.1	161.6	243.4	299.4	166.7
Total day calls	110	123	114	71	113
Total day charge	45.07	27.47	41.38	50.9	28.34
Total eve minutes	197.4	195.5	121.2	61.9	148.3
Total eve calls	99	103	110	88	122
Total eve charge	16.78	16.62	10.3	5.26	12.61
Total night minutes	244.7	254.4	162.6	196.9	186.9
Total night calls	91	103	104	89	121
Total night charge	11.01	11.45	7.32	8.86	8.41
Total intl minutes	10	13.7	12.2	6.6	10.1
Total intl calls	3	3	5	7	3
Total intl charge	2.7	3.7	3.29	1.78	2.73
Customer service calls	1	1	0	2	3
Churn	False	False	False	False	False

```
CV_data.describe().toPandas().transpose()
```

0	1	2	3	4	
summary	count	mean	stddev	min	max
Account length	2666	100.62040510127532	39.56397365334986	1	243
Area code	2666	437.43885971492875	42.52101801942723	408	510
Number vmail messages	2666	8.021755438859715	13.612277018291945	0	50
Total day minutes	2666	179.48162040510107	54.21035022086984	0.0	350.8
Total day calls	2666	100.31020255063765	19.988162186059505	0	160
Total day charge	2666	30.512404351087763	9.215732907163499	0.0	59.64
Total eve minutes	2666	200.38615903975995	50.95151511764594	0.0	363.7
Total eve calls	2666	100.02363090772693	20.161445115318898	0	170
Total eve charge	2666	17.03307201800451	4.330864176799865	0.0	30.91
Total night minutes	2666	201.16894223555903	50.780323368725305	43.7	395.0
Total night calls	2666	100.10615153788447	19.418458551101708	33	166
Total night charge	2666	9.05268942235558	2.285119512915755	1.97	17.77
Total intl minutes	2666	10.237021755438855	2.788348577051261	0.0	20.0
Total intl calls	2666	4.467366841710428	2.456194903012949	0	20
Total intl charge	2666	2.7644898724681264	0.7528120531228485	0.0	5.4
Customer service calls	2666	1.5626406601650413	1.3112357589949097	0	9

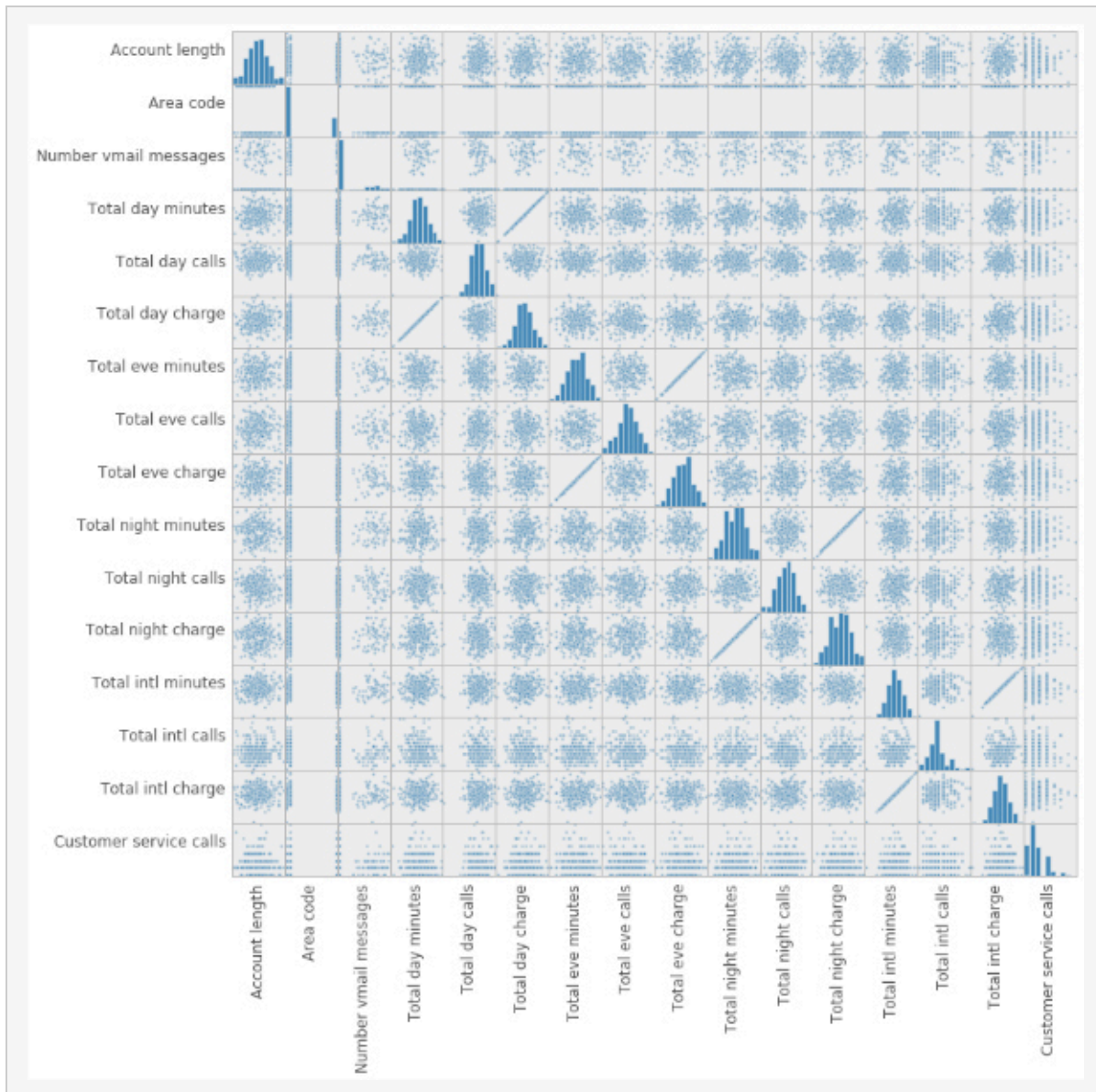
We can also perform our own statistical analyses, using the [MLlib statistics package](#) or other python packages. Here, we're use the Pandas library to examine correlations between the numeric columns by generating scatter plots of them.

For the Pandas workload, we don't want to pull the entire data set into the Spark driver, as that might exhaust the available RAM and throw an out-of-memory exception. Instead, we'll randomly sample a portion of the data (say 10%) to get a rough idea of how it looks.

```
numeric_features = [t[0] for t in CV_data.dtypes if t[1] == 'int' or t[1] == 'double']

sampled_data = CV_data.select(numeric_features).sample(False, 0.10).toPandas()

axs = pd.scatter_matrix(sampled_data, figsize=(12, 12));
# Rotate axis labels and remove axis ticks
n = len(sampled_data.columns)
for i in range(n):
    v = axs[i, 0]
    v.yaxis.label.set_rotation(0)
    v.yaxis.label.set_ha('right')
    v.set_yticks(())
    h = axs[n-1, i]
    h.xaxis.label.set_rotation(90)
    h.set_xticks(())
```



It's obvious that there are several highly correlated fields, ie *Total day minutes* and *Total day charge*. Such correlated data won't be very beneficial for our model training runs, so we're going to remove them. We'll do so by dropping one column of each pair of correlated fields, along with the *State* and *Area code* columns.

While we're in the process of manipulating the data sets, let's transform the categorical data into numeric as required by the machine learning routines, using a simple user-defined function that maps Yes/True and No/False to 1 and 0, respectively.

```
from pyspark.sql.types import DoubleType
```

```

from pyspark.sql.functions import UserDefinedFunction

binary_map = {'Yes':1.0, 'No':0.0, 'True':1.0, 'False':0.0}
toNum = UserDefinedFunction(lambda k: binary_map[k], DoubleType())

CV_data = CV_data.drop('State').drop('Area code') \
    .drop('Total day charge').drop('Total eve charge') \
    .drop('Total night charge').drop('Total intl charge') \
    .withColumn('Churn', toNum(CV_data['Churn'])) \
    .withColumn('International plan', toNum(CV_data['International p
lan'])) \
    .withColumn('Voice mail plan', toNum(CV_data['Voice mail plan']
)).cache()

final_test_data = final_test_data.drop('State').drop('Area code') \
    .drop('Total day charge').drop('Total eve charge') \
    .drop('Total night charge').drop('Total intl charge') \
    .withColumn('Churn', toNum(final_test_data['Churn'])) \
    .withColumn('International plan', toNum(final_test_data['Interna
tional plan'])) \
    .withColumn('Voice mail plan', toNum(final_test_data['Voice mail
plan'])).cache()

pd.DataFrame(CV_data.take(5), columns=CV_data.columns).transpose()

```

0	1	2	3	4	
Account length	128.0	107.0	137.0	84.0	75.0
International plan	0.0	0.0	0.0	1.0	1.0
Voice mail plan	1.0	1.0	0.0	0.0	0.0
Number vmail messages	25.0	26.0	0.0	0.0	0.0
Total day minutes	265.1	161.6	243.4	299.4	166.7
Total day calls	110.0	123.0	114.0	71.0	113.0
Total eve minutes	197.4	195.5	121.2	61.9	148.3
Total eve calls	99.0	103.0	110.0	88.0	122.0
Total night minutes	244.7	254.4	162.6	196.9	186.9
Total night calls	91.0	103.0	104.0	89.0	121.0
Total intl minutes	10.0	13.7	12.2	6.6	10.1
Total intl calls	3.0	3.0	5.0	7.0	3.0
Customer service calls	1.0	1.0	0.0	2.0	3.0
Churn	0.0	0.0	0.0	0.0	0.0

We will use decision tree model to do prediction.

Model training:

Mllib classifiers and regressors require data sets in a format of rows of type *LabeledPoint*, which separates row labels and feature lists, and names them accordingly. The custom *labelData()* function shown below performs the row parsing. We'll pass it the prepared data set (CV_data) and split it further into training and testing sets. A decision tree classifier model is then generated using the training data, using a *maxDepth* of 2, to build a "shallow" tree. The tree depth can be regarded as an indicator of model complexity.

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import DecisionTree

def labelData(data):
    # label: row[end], features: row[0:end-1]
    return data.map(lambda row: LabeledPoint(row[-1], row[:-1]))

training_data, testing_data = labelData(CV_data).randomSplit([0.8,0.2])

model = DecisionTree.trainClassifier(training_data, numClasses=2, maxDepth=2, categoricalFeaturesInfo={1:2, 2:2}, impurity='gini', maxBins=32)

print model.toDebugString()

DecisionTreeModel classifier of depth 2 with 7 nodes

If (feature 12 <= 3.0)
  If (feature 4 <= 262.8)
    Predict: 0.0
  Else (feature 4 > 262.8)
    Predict: 1.0
Else (feature 12 > 3.0)
  If (feature 4 <= 169.9)
    Predict: 1.0
  Else (feature 4 > 169.9)
    Predict: 0.0
```

Decision trees are often used for feature selection because they provide an automated mechanism for determining the most important features (those closest to the tree root).

```
print 'Feature 12:', CV_data.columns[12]
print 'Feature 4: ', CV_data.columns[4]

Feature 12: Customer service calls
Feature 4: Total day minutes
```

Model evaluation:

Predictions of the testing data's churn outcome are made with the model's *predict()* function and grouped together with the actual churn label of each customer data using *getPredictionsLabels()*.

We'll use MLlib's *MulticlassMetrics()* for the model evaluation, which takes rows of (prediction, label) tuples as input. It provides metrics such as precision, recall, F1 score and confusion matrix, which have been bundled for printing with the custom *printMetrics()* function.

```
from pyspark.mllib.evaluation import MulticlassMetrics

def getPredictionsLabels(model, test_data):
    predictions = model.predict(test_data.map(lambda r: r.features))

    return predictions.zip(test_data.map(lambda r: r.label))

def printMetrics(predictions_and_labels):
    metrics = MulticlassMetrics(predictions_and_labels)
    print 'Precision of True ', metrics.precision(1)
    print 'Precision of False', metrics.precision(0)
    print 'Recall of True    ', metrics.recall(1)
    print 'Recall of False   ', metrics.recall(0)
    print 'F-1 Score        ', metrics.fMeasure()
    print 'Confusion Matrix\n', metrics.confusionMatrix().toArray()

predictions_and_labels = getPredictionsLabels(model, testing_data)

printMetrics(predictions_and_labels)

Precision of True  0.697674418605
Precision of False 0.908722109533
Recall of True     0.4
Recall of False    0.971800433839
F-1 Score          0.891791044776
Confusion Matrix [[ 448\.   13.]
                  [  45\.   30.]
```