ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark Program

CHAPTER 3: DATAFRAMES

Chapter Objectives

In this chapter, we will:

- Introduce DataFrames
- Show how to create a structured object using DataFrames
- Apply transformations and actions on DataFrames

DataFrames

- Spark 2.0 introduced a more feature rich and easier to use version of RDD's known as a DataFrame
  - Modeled to be similar to Pandas DataFrame so it is easily familiar
  - Is an RDD but has column names and data types
  - Has transformations and actions that are easier to use than RDD versions
  - Attempts to be more SQL-like for even more familiarity
  - Can read and write many more file formats than basic RDD's could

Make a DataFrame

- Spark 2.0 introduced the `SparkSession`, simply called spark in PySpark
  - Provides easier access to the different spark contexts
  - `spark.sparkContext` is the same as the old `sc`
- Once we have the spark context, we can start using DataFrames

```
x = sc.parallelize([(1,'alpha'),(2,'beta')])
x0 = spark.createDataFrame(x)
x0.show()

+---+-----+
| _1|   _2|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+
```

Column Names

- To make the DataFrame more useful, column names can be applied

```
x1 = spark.createDataFrame(x, schema = ['ID', 'Name'])
x1.show()
x1.describe()

+---+-----+
| ID| Name|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+

DataFrame[summary: string, ID: string, Name: string]
```

Schemas

- To make the DataFrame even more useful, a schema with data types can be applied

```
x2 = spark.createDataFrame(x, 'ID:int, Name:string')
x2.show()
print(x2)
+---+-----+
| ID| Name|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+

DataFrame[ID: int, Name: string]
```

Convert RDD to DataFrame

- An existing RDD can also be turned into a DataFrame using the `toDF` method
  - Using the credit card csv file from before:

```
cc = sc.textFile ('/home/student/ROI/SparkProgram/
  datasets/finance/CreditCard.csv')
first = cc.first()
cc = cc.filter(lambda x : x != first)
import datetime
cc = cc.map(lambda x : x.split(','))
cc = cc.map(lambda x : (x[0][1:], x[1][1:-1],
  datetime.datetime.strptime(x[2], '%d-%b-%y').date(),
  x[3], x[4], x[5], float(x[6])))
df = cc.toDF()
df.show()
df = cc.toDF(['City', 'Country', 'Date', 'CardType',
  'TranType', 'Gender', 'Amount'])
df.show()
```

Selecting Columns

- DataFrames have methods with names similar to SQL commands

```
df.select('City', 'Country', 'Amount').show(10)

df.select('City', 'Country').distinct().show()

df.sort(df.Amount).show()

df.sort(df.Amount, ascending = False).show()

df.select('City', 'Amount').orderBy(df.City).show()
```

Calculated Columns

- New columns can be added to a DataFrame

```
df2 = df.withColumn('Discount', df.Amount * .03)
df2.show()
```

- Columns can be removed when not needed

```
df3 = df2.drop(df2.Country)
df3.show()
```

Filtering Data

- DataFrames can be filtered like a SQL table using either the `filter` or `where` method
  - They are the exact same method with different aliases

```
df3.filter(df3.Amount < 4000).show()
df3.filter('Amount < 4000').count()
df3.where('Amount < 4000').count()
df3.where(df3.Amount < 4000).count()
df3.where((df3.Amount > 3000) & (df3.Amount <
4000)).count()
df3.where('Amount > 3000 and Amount < 3000').count()
```

Sorting

- The `sort` and `orderBy` methods are different aliases for the same function

```
df.sort(df.Amount).show()
```

- They sort a DataFrame in ascending order or descending order if you pass the `ascending = False` parameter

```
df.sort(df.Amount, ascending = False).show()
```

- You can sort on multiple columns

```
df.select('City', 'Amount').orderBy(df.City,
df.Amount).show()
```

- Custom sort functions can use the `withColumn` method

JOIN

- DataFrames can be joined to other DataFrames just as you would in SQL and all the expected types are supported
  - INNER
  - LEFT
  - RIGHT
  - FULL

```
tab1 = sc.parallelize([(1, 'Alpha'), (2, 'Beta'), (3,
'Delta')]).toDF('ID:int, code:string')

tab2 = sc.parallelize([(100, 'One', 1), (101, 'Two', 2),
(102, 'Three', 1), (103, 'Four', 4)]) \
.toDF('ID:int, name:string, parentID:int')

tab1.join(tab2, tab1.ID == tab2.parentID).show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'left').show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'right').show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'full').show()
```

Grouping and Aggregating

- Grouping in Spark works a little differently—the `groupBy` method creates a grouped DataFrame which can then have aggregate methods called on it

```
tab3 = sc.parallelize([(1, 10), (1, 20), (1, 30), (2,
40), (2,50)]).toDF('groupID:int, amount:int')
x = tab3.groupby('groupID')
```

- There are various different syntaxes to accomplish the same results
  - Call the method after grouping

    ```
    x.max().show()
    ```
  - Use the `agg` method with a dictionary

    ```
    x.agg({'amount':'sum', 'amount':'max'}).show()
    ```
  - Use the `agg` method with the function names

    ```
    from pyspark.sql import functions as F
    x.agg(F.sum('amount'), F.max('amount')).show()
    ```

Reading Files

- There are many file formats directly supported for reading and writing
  - `csv`
  - `json`
  - `orc`
  - `parquet`
  - `jdbc`
- Other formats can be loaded using custom Java classes
  - Cassandra
  - Mongo
  - HBase
  - AVRO

Reading CSV Files

- There are many different syntaxes that you will see but they all do the same thing
- The `sep` parameter can also be used to indicate different separators like `\t` for tab

```
filename = '/home/student/ROI/SparkProgram/datasets/
           finance/CreditCard.csv'

df4 = spark.read.load(filename, format = 'csv',
  sep = ',', inferSchema = True, header = True)

df4 = spark.read.format('csv').option('header','true').
  option('inferSchema','true').load(filename)

df4 = spark.read.csv(filename, header = True,
  inferSchema = True)
```

Writing Files

- The write method on a DataFrame can be used just like the `read` function using many different options
- Some options are built-in such as:
  - `jdbc`
  - `json spark.read.json(filename)df.write.json(file)`
  - `orc  spark.read.orc(filename) df.write.orc(file)`
  - `parquet spark.read.parquet(file) df.write.parquet(file)`
  - `text  spark.read.text(file) df.write.text(file)`
- Other formats can use the option to supply a custom Java class that can be downloaded and installed on the computer
  - AVRO:
    ```
    spark.read.format("com.databricks.spark.avro").load("kv.avro")
    ```
  - Cassandra:
    ```
    sqlContext.read.format("org.apache.spark.sql.cassandra")
    .options(table = table_name, keyspace = keys_space_name).load()
    ```

Miscellaneous Useful Methods

- A lot of standard SQL is supported by Spark
- Using the `expr` function in combination with `withColumn` you can add calculated columns to a DataFrame if you can code the calculation as standard SQL

```
from pyspark.sql.functions import expr
x2.withColumn('uppername', expr('upper(name)')).show()
```

- Sometimes you just want to easily rename a column
  - `withColumnRenamed(oldname, newname)`
- Like `collect()` the `toLocalIterator()` method will return all the results to the driver node, but it does it as a generator instead of a `list`
- Just like SQL there are methods `union`, `unionAll`, `subtract`, and `intersect`

User Defined Functions

- Sometimes it is necessary to write complex functions using Python
- Import the helper functions in pyspark.sql

```
from pyspark.sql.functions import udf
from pyspark.sql.types import *
from pyspark.sql.functions import to_date
```

- Write whatever custom function you need:

```
def city(x):
    return x[:x.find(',')]
def country(x):
    return x[x.find(',') + 1 :]
```

- Call the built-in function or use the `udf` function to wrap and call your UDF:

```
df4.withColumn('City', udf(city, StringType())(df4.CityCountry)) \
.withColumn('Country', udf(country, StringType())(df4.CityCountry)) \
.withColumn('Date', to_date(df4.Date, 'dd-MMM-yy')) \
.drop(df4.CityCountry)
```

Chapter Summary

In this chapter, we have:

- Introduced DataFrames
- Shown how to create a structured object using DataFrames
- Applied transformations and actions on DataFrames