Pandas and PySpark

Data Wrangling with PySpark for Data Scientists Who Know Pandas - ***Andrew Ray***

PRIMER

Distributed compute

- YARN, Mesos, Standalone cluster

Abstractions

- RDD—distributed collection of objects
- Dataframe—distributed dataset of tabular data.
  - Integrated SQL
  - ML Algorithms

IMPORTANT CONCEPTS

Immutable

- Changes create new object references
- Old versions are unchanged

Lazy

- Compute does not happen until output is requested

LOAD CSV

Pandas

```
df = pd.read_csv("mtcars.csv")
```

PySpark

```
df = spark.read.csv("mtcars.csv")
```

If we want to add options while loading csv then we can use options method. It's a bit different than pandas. Here we are chaining the methods to achieve the same results.

```
df = spark.read \
  .options(header=True, inferSchema=True) \
  .csv("mtcars.csv")
```

VIEW DATAFRAME

Pandas

```
df
df.head(10)
```

PySpark

```
df.show()
df.show(10)
```

To view the dataframe in Pandas we use variable name referencing to the dataframe object and its string representation shows us the well-structured dataframe data. In case of pyspark, we use the method show() for this; however, it is always better to first use limit (df.limit(5).toPandas()) and then convert to pandas to see the better string representation of data.

COLUMNS AND DATA TYPES

Pandas

```
df.columns
df.dtypes
```

PySpark

```
df.columns
df.dtypes
```

RENAME COLUMNS

Pandas

```
df.columns = ['a', 'b', 'c']
```

PySpark

```
df.toDF('a', 'b', 'c')
```

In python we can directly change the names of the columns of the same dataframe; however, since the objects in pyspark is immutable, thus the df.toDf('a', 'b', 'c') creates the new spark-dataframe object keeping the old spark-dataframe object as it is.

To change individual column names we can use method withColumnRenamed() method in pyspark.

Pandas

```
df.columns = ['a', 'b', 'c']
df.rename(columns = {'old': 'new'})
```

PySpark

```
df.toDF('a', 'b', 'c')
df.withColumnRenamed('old','new')
```

DROP COLUMN

Pandas

```
df.drop('mpg', axis=1)
```

PySpark

```
df.drop('mpg')
```

In pyspark the rows are not indexed, thus there is no concept of axis, therefore, we can only drop columns as above in pyspark.

FILTERING

Pandas

```
df[df.mpg < 20]
df[(df.mpg < 20) & (df.cyl == 6)]
```

PySpark

```
df[df.mpg < 20]
df[(df.mpg < 20) & (df.cyl == 6)]
```

Always remember to put the parenthesis in compound filtering based on several columns.

ADD COLUMN

Pandas

```
df['gpm'] = 1 / df.mpg
```

PySpark

```
df.withColumn('gpm', 1 / df.mpg)
```

Again, in pyspark the object is immutable so we use method withColumn('column name', 'function') to create a new dataframe object with the new column.

FILL NULLS

Pandas

```
df.fillna(0) ⟵ Many more options
```

PySpark

```
df.fillna(0)
```

Pandas have many more options of filling na values, however, in case of spark we can do that by using pyspark sql functions.

AGGREGATION

Pandas

```
df.groupby(['cyl', 'gear']) \
  .agg({'mpg': 'mean', 'disp': 'min'})
```

PySpark

```
df.groupby(['cyl', 'gear']) \
  .agg({'mpg': 'mean', 'disp': 'min'})
```

STANDARD TRANSFORMATIONS

Pandas

```
import numpy as np
df['logdisp'] = np.log(df.disp)
```

PySpark

```
import pyspark.sql.functions as F
df.withColumn('logdisp', F.log(df.disp))
```

We do not use numpy's log function in case of pyspark because pyspark is wrapper based on scala, which in turn is based on java, thus we avoid using any non-java functions here. Therefore, here we have imported sql functions from pyspark as 'F'. This has all those necessary transformation suitable for java and scala. 'F' has many transformation functions as mentioned below.

```
import pyspark.sql.functions as F
```

```
AutoBatchedSerializer collect_set expr length rank substring Column column ctorial
levenshtein regexp_extract substring_index Dataame concat rst lit regexp_replace sum
PickleSerializer concat_ws oor locate repeat sumDistinct SparkContext conv rmat_number log
reverse sys StringType corr rmat_string log10 rint tan UserDenednction cos om_json log1p
round tanh abs cosh om_unixtime log2 row_number toDegrees acos count om_utc_timestamp
lower rpad toRadians add_months countDistinct get_json_object lpad rtrim to_date
approxCountDistinct covar_pop greatest ltrim second to_json approx_count_distinct
covar_samp grouping map sha1 to_utc_timestamp array crc32 grouping_id math sha2 translate
array_contains create_map hash max shiLe trim asc cume_dist hex md5 shiRight trunc ascii
current_date hour mean shiRightUnsigned udasin current_timestamp hypot min signum unbase64
atan date_add ignore_unicode_prex minute sin unhex atan2 date_rmat initcap
monotonically_increasing_id since unix_timestamp avg date_sub input_le_name month sinh
upper base64 datedi instr months_between size v bin dayoonth isnan nanvl skewness var_pop
bitwiseNOT dayoear isnull next_day sort_array var_samp blacklist decode json_tuple ntile
soundex variance broadcast degrees k percent_rank spark_partition_id weekoear bround
dense_rank kurtosis posexplode split when cbrt desc lag pow sqrt window ceil encode last
quarter stddev year coalesce exp last_day radians stddev_pop col explode lead rand
stddev_samp collect_list expm1 least randn struct
```

ROW CONDITIONAL STATEMENTS

Pandas

```
df['cond']=df.apply(lambda r:
  1 if r.mpg > 20 else 2 if r.cyl == 6 else 3,
  axis=1)
```

PySpark

```
import pyspark.sql.functions as F
df.withColumn('cond', \
  F.when(df.mpg > 20, 1) \
   .when(df.cyl == 6, 2) \
   .otherwise(3))
```

PYTHON WHEN REQUIRED

Pandas

```
df['disp1'] = df.disp.apply(lambda x: x+1)
```

PySpark

```
import pyspark.sql.functions as F
from pyspark.sql.types import DoubleType
fn = F.udf(lambda x: x+1, DoubleType())
df.withColumn('disp1', fn(df.disp))
```

There are very few times when we use python in pyspark directly. This is one of those times when we have to use it. Here we are using the lambda function from python in pyspark. SO we have first registered lambda function using udf {F.udf(lambda function , return type)}, and then we have used it within withColumn() method. One thing to note here is that the DoubleType() has been exported (from pyspark.sql.types import DoubleType).

MERGE/JOIN DATAFRAMES

Pandas

```
left.merge(right, on='key')
left.merge(right, left_on='a', right_on='b')
```

PySpark

```
left.join(right, on='key')
left.join(right, left.a == right.b)
```

It has all the joins such as default inner joins, outer, inner etc.

PIVOT TABLE

Pandas

```
pd.pivot_table(df, values='D', \
  index=['A', 'B'], columns=['C'], \
  aggfunc=np.sum)
```

PySpark

```
df.groupBy("A", "B").pivot("C").sum("D")
```

SUMMARY STATISTICS

Pandas

```
df.describe()
```

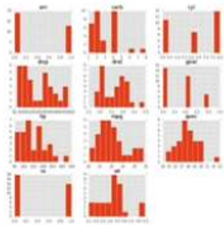
PySpark

```
df.describe().show()
```

(only count, mean, stddev, min, max)

```
df.selectExpr(
  "percentile_approx(mpg, array(.25, .5, .75)) as mpg"
  ).show()
```

HISTOGRAM



Pandas

```
df.hist()
```

PySpark

```
df.sample(False, 0.1).toPandas().hist()
```

We should always sample or Limit our data from pyspark before converting it into the pandas and then create a graph. It is because we pyspark dataframe is very huge and if you put that dataframe in memory then you will run out of memory. If we really want to do it then we can to a limit but if we can randomly take a good random sample of data then statistically we don't need to plot the whole data.

SQL Support in PySpark

We don't have much SQL support in Pandas; however we have a lot of SQL support in Pyspark. Also , we can always switch back and forth between dataframes and sql.

SQL

Pandas

```
n/a
```

PySpark

```
df.createOrReplaceTempView('foo')
df2 = spark.sql('select * from foo')
```

df.CreateOrReplaceTempView() is way of converting a pyspark dataframe into a SQL friendly table on which we can run any SQL query, and after running the query , the query result could be saved into the another pyspark dataframe.

BEST PRACTICES

MAKE SURE TO

- Use `pyspark.sql.functions` and other built in functions.
- Use the same version of python and packages on cluster as driver.
- Check out the UI at http://localhost:4040/
- Learn about SSH port forwarding
- Check out Spark MLlib
- RTFM: https://spark.apache.org/docs/latest/

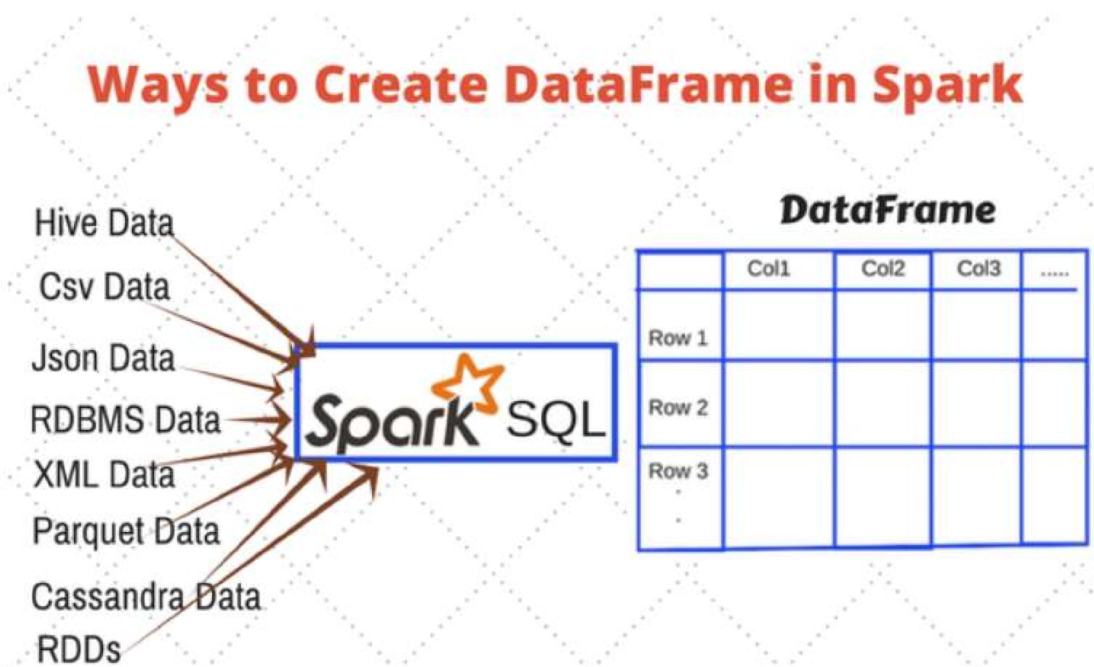
THINGS NOT TO DO

- Try to iterate through rows
- Hard code a master in your driver
  - Use `spark-submit` for that
- `df.toPandas().head()`
  - instead do: `df.limit(5).toPandas()`

IF THINGS GO WRONG

- Don't panic!
- Read the error
- Google it
- Search/Ask Stack Overflow (tag apache-spark)
- Search/Ask the user list: user@spark.apache.org
- Find a bug? Make a JIRA ticket: https://issues.apache.org/jira/browse/SPARK/

DATAFRAME OPERATIONS – ANALYTICS VIDHYA



- **How to see datatype of columns?**

To see the types of columns in DataFrame, we can use the printSchema, dtypes. Let's apply printSchema() on train which will Print the schema in a tree format.

```
train.printSchema()
Output:
root
 |-- User_ID: integer (nullable = true)
 |-- Product_ID: string (nullable = true)
 |-- Gender: string (nullable = true)
 |-- Age: string (nullable = true)
 |-- Occupation: integer (nullable = true)
 |-- City_Category: string (nullable = true)
 |-- Stay_In_Current_City_Years: string (nullable = true)
 |-- Marital_Status: integer (nullable = true)
 |-- Product_Category_1: integer (nullable = true)
 |-- Product_Category_2: integer (nullable = true)
 |-- Product_Category_3: integer (nullable = true)
 |-- Purchase: integer (nullable = true)
```

From above output, we can see that, we have perfectly captured the schema / data types of each columns while reading from csv.

- **How to Show first n observation?**

We can use **head** operation to see first n observation (say, 5 observation). Head operation in PySpark is similar to **head** operation in Pandas.

```
train.head(5)
Output:
[Row(User_ID=1000001, Product_ID=u'P00069042', Gender=u'F', Age=u'0-17', Occupation=10, City_Cate
gory=u'A', Stay_In_Current_City_Years=u'2', Marital_Status=0, Product_Category_1=3, Product_Categ
ory_2=None, Product_Category_3=None, Purchase=8370),
 Row(User_ID=1000001, Product_ID=u'P00248942', Gender=u'F', Age=u'0-17', Occupation=10, City_Cate
gory=u'A', Stay_In_Current_City_Years=u'2', Marital_Status=0, Product_Category_1=1, Product_Categ
ory_2=6, Product_Category_3=14, Purchase=15200),
 Row(User_ID=1000001, Product_ID=u'P00087842', Gender=u'F', Age=u'0-17', Occupation=10, City_Cate
gory=u'A', Stay_In_Current_City_Years=u'2', Marital_Status=0, Product_Category_1=12, Product_Cate
gory_2=None, Product_Category_3=None, Purchase=1422),
 Row(User_ID=1000001, Product_ID=u'P00085442', Gender=u'F', Age=u'0-17', Occupation=10, City_Cate
gory=u'A', Stay_In_Current_City_Years=u'2', Marital_Status=0, Product_Category_1=12, Product_Cate
gory_2=14, Product_Category_3=None, Purchase=1057),
 Row(User_ID=1000002, Product_ID=u'P00285442', Gender=u'M', Age=u'55+', Occupation=16, City_Categ
ory=u'C', Stay_In_Current_City_Years=u'4+', Marital_Status=0, Product_Category_1=8, Product_Categ
ory_2=None, Product_Category_3=None, Purchase=7969)]
```

Above results are comprised of row like format. To see the result in more interactive manner (rows under the columns), we can use the **show** operation. Let's apply show operation on train and take first 2 rows of it. We can pass the argument truncate = True to truncate the result.

```
train.show(2,truncate= True)
Output:
```

```
+-------+----------+------+----+----------+-------------+--------------------------+------------
-+------------------+------------------+------------------+--------+
|User_ID|Product_ID|Gender| Age|Occupation|City_Category|Stay_In_Current_City_Years|Marital_Statu
s|Product_Category_1|Product_Category_2|Product_Category_3|Purchase|
+-------+----------+------+----+----------+-------------+--------------------------+------------
-+------------------+------------------+------------------+--------+
|1000001| P00069042|     F|0-17|        10|            A|                         2|
0|                 3|              null|              null|    8370|
|1000001| P00248942|     F|0-17|        10|            A|                         2|
0|                 1|                 6|                14|   15200|
+-------+----------+------+----+----------+-------------+--------------------------+------------
-+------------------+------------------+------------------+--------+
only showing top 2 rows
```

- **How to Count the number of rows in DataFrame?**

We can use **count** operation to count the number of rows in DataFrame. Let's apply **count** operation on train & test files to count the number of rows.

```
train.count(),test.count()
Output:
(550068, 233599)
```

We have 550068, 233599 rows in train and test respectively.

- **How many columns do we have in train and test files along with their names?**

For getting the columns name we can use **columns** on DataFrame, similar to what we do for getting the columns in pandas DataFrame. Let's first print the number of columns and columns name in train file then in test file.

```
len(train.columns), train.columns
OutPut:
12 ['User_ID', 'Product_ID', 'Gender', 'Age', 'Occupation', 'City_Category', 'Stay_In_Current_Cit
y_Years', 'Marital_Status', 'Product_Category_1', 'Product_Category_2', 'Product_Category_3', 'Pu
rchase']
```

- **How to get the summary statistics (mean, standard deviance, min ,max, count) of numerical columns in a DataFrame?**

describe operation is use to calculate the summary statistics of numerical column(s) in DataFrame. If we don't specify the name of columns it will calculate summary statistics for all numerical columns present in DataFrame.

```
train.describe().show()
Output:
+-------+------------------+-----------------+-------------------+------------------+-----------
------+------------------+------------------+
|summary|           User_ID|       Occupation|     Marital_Status|Product_Category_1|Product_Cate
gory_2|Product_Category_3|          Purchase|
+-------+------------------+-----------------+-------------------+------------------+-----------
------+------------------+------------------+
|  count|            550068|           550068|             550068|            550068|
376430|            166821|            550068|
|   mean|1003028.8424013031|8.076706879876669|0.40965298835780306| 5.404270017525106| 9.842329251
122386|12.668243206790512| 9263.968712959126|
| stddev|1727.5915855308265|6.522660487341778| 0.4917701263173273|3.9362113692014082| 5.086589648
693526| 4.125337631575267|5023.0653938206015|
|    min|           1000001|                0|                  0|                 1|
2|                 3|                12|
```

Let's check what happens when we specify the name of a categorical / String columns in **describe** operation.

```
train.describe('Product_ID').show()
Output:
+-------+----------+
|summary|Product_ID|
+-------+----------+
|  count|    550068|
|   mean|      null|
| stddev|      null|
|    min| P00000142|
|    max|  P0099942|
+-------+----------+
```

As we can see that, **describe** operation is working for String type column but the output for mean, stddev are null and min & max values are calculated based on ASCII value of categories.

- **How to select column(s) from the DataFrame?**

To subset the columns, we need to use **select** operation on DataFrame and we need to pass the columns names separated by commas inside **select** Operation. Let's select first 5 rows of 'User_ID' and 'Age' from the train.

```
train.select('User_ID','Age').show(5)
 Output:
+-------+----+
|User_ID| Age|
+-------+----+
|1000001|0-17|
|1000001|0-17|
|1000001|0-17|
|1000001|0-17|
|1000002| 55+|
+-------+----+
```

- **How to find the number of distinct product in train and test files?**

The **distinct** operation can be used here, to calculate the number of distinct rows in a DataFrame. Let's apply **distinct** operation to calculate the number of distinct product in train and test file each.

```
train.select('Product_ID').distinct().count(),test.select('Product_ID').distinct().count()
Output:
(3631, 3491)
```

We have 3631 & 3491 distinct product in train & test file respectively. After counting the number of distinct values for train and test files, we can see the train file has more categories than test file. Let us check what are the categories for Product_ID, which are in test file but not in train file by applying **subtract** operation.We can do the same for all categorical features.

```
diff_cat_in_train_test=test.select('Product_ID').subtract(train.select('Product_ID'))
diff_cat_in_train_test.distinct().count()# For distict count
Output:
46
```

Above, you can see that 46 different categories are in test file but not in train. In this case, either we collect more data about them or skip the rows in test file for those categories (invalid category) which are not in train file.

- **What if I want to calculate pair wise frequency of categorical columns?**

We can use **crosstab** operation on DataFrame to calculate the pair wise frequency of columns. Let's apply **crosstab** operation on 'Age' and 'Gender' columns of train DataFrame.

```
train.crosstab('Age', 'Gender').show()
Output:
+----------+-----+------+
|Age_Gender|    F|     M|
+----------+-----+------+
|      0-17| 5083| 10019|
|     46-50|13199| 32502|
|     18-25|24628| 75032|
|     36-45|27170| 82843|
|       55+| 5083| 16421|
|     51-55| 9894| 28607|
|     26-35|50752|168835|
+----------+-----+------+
```

In the above output, the first column of each row will be the distinct values of Age and the column names will be the distinct values of Gender. The name of the first column will be Age_Gender. Pair with no occurrences will have zero count in contingency table.

- **What If I want to get the DataFrame which won't have duplicate rows of given DataFrame?**

We can use **dropDuplicates** operation to drop the duplicate rows of a DataFrame and get the DataFrame which won't have duplicate rows. To demonstrate that I am performing this on two columns Age and Gender of train and get the all unique rows for these columns.

```
train.select('Age','Gender').dropDuplicates().show()
Output:
+-----+------+
|  Age|Gender|
+-----+------+
|51-55|     F|
|51-55|     M|
|26-35|     F|
|26-35|     M|
|36-45|     F|
|36-45|     M|
|46-50|     F|
|46-50|     M|
|  55+|     F|
|  55+|     M|
|18-25|     F|
| 0-17|     F|
```

- **What if I want to drop the all rows with null value?**

The **dropna** operation can be use here. To drop row from the DataFrame it consider three options.

- how– 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null.
- thresh – int, default None If specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.
- subset – optional list of column names to consider.

Let't drop null rows in train with default parameters and count the rows in output DataFrame. Default options are any, None, None for how, thresh, subset respectively.

```
train.dropna().count()
Output:
166821
```

- **What if I want to fill the null values in DataFrame with constant number?**

Use **fillna** operation here. The **fillna** will take two parameters to fill the null values.

- value:
  - It will take a dictionary to specify which column will replace with which value.
  - A value (int , float, string) for all columns.

- subset: Specify some selected columns.

Let's fill '-1' inplace of null values in train DataFrame.

```
train.fillna(-1).show(2)
Output:
+-------+----------+------+----+----------+-------------+--------------------------+------------
-+------------------+------------------+------------------+--------+
|User_ID|Product_ID|Gender| Age|Occupation|City_Category|Stay_In_Current_City_Years|Marital_Statu
s|Product_Category_1|Product_Category_2|Product_Category_3|Purchase|
+-------+----------+------+----+----------+-------------+--------------------------+------------
-+------------------+------------------+------------------+--------+
|1000001| P00069042|     F|0-17|        10|            A|                         2|
0|                 3|                -1|                -1|    8370|
|1000001| P00248942|     F|0-17|        10|            A|                         2|
0|                 1|                 6|                14|   15200|
+-------+----------+------+----+----------+-------------+--------------------------+------------
-+------------------+------------------+------------------+--------+
```

- **If I want to filter the rows in train which has Purchase more than 15000?**

We can apply the **filter** operation on Purchase column in train DataFrame to filter out the rows with values more than 15000. We need to pass a condition. Let's apply filter on Purchase column in train DataFrame and print the number of rows which has more purchase than 15000.

```
train.filter(train.Purchase > 15000).count()
Output:
110523
```

- **How to find the mean of each age group in train?**

The **groupby** operation can be used here to find the mean of Purchase for each age group in train. Let's see how can we get the mean purchase for the 'Age' column train.

```
train.groupby('Age').agg({'Purchase': 'mean'}).show()
Output:
+-----+-----------------+
|  Age|    avg(Purchase)|
+-----+-----------------+
|51-55|9534.808030960236|
|46-50|9208.625697468327|
| 0-17|8933.464640444974|
|36-45|9331.350694917874|
```

We can also apply sum, min, max, count with **groupby** when we want to get different summary insight each group. Let's take one more example of **groupby** to count the number of rows in each Age group.

```
train.groupby('Age').count().show()
Output:
+-----+------+
|  Age| count|
+-----+------+
|51-55| 38501|
|46-50| 45701|
| 0-17| 15102|
|36-45|110013|
|26-35|219587|
|  55+| 21504|
|18-25| 99660|
+-----+------+
```

- ### How to create a sample DataFrame from the base DataFrame?

We can use **sample** operation to take sample of a DataFrame. The sample method on DataFrame will return a DataFrame containing the sample of base DataFrame. The sample method will take 3 parameters.

- withReplacement = True or False to select a observation with or without replacement.
- fraction = x, where x = .5 shows that we want to have 50% data in sample DataFrame.
- seed for reproduce the result

Let's create the two DataFrame t1 and t2 from train, both will have 20% sample of train and count the number of rows in each.

```
t1 = train.sample(False, 0.2, 42)
t2 = train.sample(False, 0.2, 43)
t1.count(),t2.count()
Output:
(109812, 109745)
```

- ### How to apply map operation on DataFrame columns?

We can apply a function on each row of DataFrame using map operation. After applying this function, we get the result in the form of RDD. Let's apply a map operation on User_ID column of train and print the first 5 elements of mapped RDD(x,1) after applying the function (I am applying lambda function).

```
train.select('User_ID').map(lambda x:(x,1)).take(5)
Output:
[(Row(User_ID=1000001), 1),
 (Row(User_ID=1000001), 1),
 (Row(User_ID=1000001), 1),
 (Row(User_ID=1000001), 1),
 (Row(User_ID=1000002), 1)]
```

In above code we have passed lambda function in the map operation which will take each row / element of 'User_ID' one by one and return pair for them ('User_ID',1).

- **How to sort the DataFrame based on column(s)?**

We can use **orderBy** operation on DataFrame to get sorted output based on some column. The **orderBy** operation take two arguments.

- List of columns.
- ascending = True or False for getting the results in ascending or descending order(list in case of more than two columns)

Let's sort the train DataFrame based on 'Purchase'.

```
train.orderBy(train.Purchase.desc()).show(5)
Output:
+-------+----------+------+-----+----------+-------------+--------------------------+------------
--+------------------+------------------+------------------+--------+
|User_ID|Product_ID|Gender|  Age|Occupation|City_Category|Stay_In_Current_City_Years|Marital_Stat
us|Product_Category_1|Product_Category_2|Product_Category_3|Purchase|
+-------+----------+------+-----+----------+-------------+--------------------------+------------
--+------------------+------------------+------------------+--------+
|1003160| P00052842|     M|26-35|        17|            C|                         3|
0|                10|                15|              null|   23961|
|1002272| P00052842|     M|26-35|         0|            C|                         1|
0|                10|                15|              null|   23961|
|1001474| P00052842|     M|26-35|         4|            A|                         2|
1|                10|                15|              null|   23961|
```

- **How to add the new column in DataFrame?**

We can use **withColumn** operation to add new column (we can also replace) in base DataFrame and return a new DataFrame. The **withColumn** operation will take 2 parameters.

- Column name which we want add /replace.
- Expression on column.

Let's see how **withColumn** works. I am calculating new column name 'Purchase_new' in train which is calculated by dviding Purchase column by 2.

```
train.withColumn('Purchase_new', train.Purchase /2.0).select('Purchase','Purchase_new').show(5)
Output:
+--------+------------+
|Purchase|Purchase_new|
+--------+------------+
|    8370|      4185.0|
|   15200|      7600.0|
|    1422|       711.0|
|    1057|       528.5|
|    7969|      3984.5|
+--------+------------+
only showing top 5 rows
```

- **How to drop a column in DataFrame?**

To drop a column from the DataFrame we can use **drop** operation. Let's drop the column called 'Comb' from the test and get the remaining columns in test.

```
test.drop('Comb').columns
Output:
['',
 'User_ID',
 'Product_ID',
 'Gender',
 'Age',
 'Occupation',
 'City_Category',
 'Stay_In_Current_City_Years',
 'Marital_Status',
 'Product_Category_1',
 'Product_Category_2',
 'Product_Category_3']
```

Intro to UDF

- **What if I want to remove some categories of Product_ID column in test that are not present in Product_ID column in train?**

Here, we can use a user defined function (**udf**) to remove the categories of a column which are in test but not in train. Let's again calculate the categories in Product_ID column which are in test but not in train.

```
diff_cat_in_train_test=test.select('Product_ID').subtract(train.select('Product_ID'))
diff_cat_in_train_test.distinct().count()# For distict count
Output:
46
```

We have got 46 different categories in test. For removing these categories from the test 'Product_ID' column. I am applying these steps.

- Create the distinct list of categories called 'not_found_cat' from the diff_cat_in_train_test using map operation.
- Register a udf(user define function).
- User defined function will take each element of test column and search this in not_found_cat list and it will put -1 if it finds in this list otherwise it will do nothing.

Let's see how it works. First create 'not_found_cat'

```
not_found_cat = diff_cat_in_train_test.distinct().rdd.map(lambda x: x[0]).collect()
len(not_found_cat)
Output:
46
```

Now resister the udf, we need to import **StringType** from the **pyspark.sql** and **udf** from the **pyspark.sql.functions**. The **udf** function takes 2 parameters as arguments:

- Function (I am using lambda function)
- Return type (in my case StringType())

```
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf
F1 = udf(lambda x: '-1' if x in not_found_cat else x, StringType())
```

In the above code function name is 'F1' and we are putting '-1' for not found catagories in test 'Product_ID'. Finally apply above 'F1' function on test 'Product_ID' and take result in k1 for new column calles "NEW_Product_ID".

```
k = test.withColumn("NEW_Product_ID",F1(test["Product_ID"])).select('NEW_Product_ID')
```

Now, let's see the results by again calculating the different categories in k and train **subtract** operation.

```
diff_cat_in_train_test=k.select('NEW_Product_ID').subtract(train.select('Product_ID'))
diff_cat_in_train_test.distinct().count()# For distinct count
Output:
1
```

The output 1 means we have now only 1 different category k and train.

```
diff_cat_in_train_test.distinct().collect()
Output:
Row(NEW_Product_ID=u'-1')
```

6. How to Apply SQL Queries on DataFrame?

We have already discussed in the above section that DataFrame has additional information about datatypes and names of columns associated with it. Unlike RDD, this additional information allows Spark to run SQL queries on DataFrame. To apply SQL queries on DataFrame first we need to register DataFrame as table. Let's first register train DataFrame as table.

```
train.registerAsTable('train_table')
```

In the above code, we have registered 'train' as table('train_table') with the help of **registerAsTable** operation. Let's apply SQL queries on 'train_table' to select Product_ID the result of SQL query will be a DataFrame. We need to apply a action to get the result.

```
sqlContext.sql('select Product_ID from train_table').show(5)
Output:
+----------+
|Product_ID|
+----------+
| P00069042|
| P00248942|
| P00087842|
| P00085442|
| P00285442|
+----------+
```

In the above code, I am using sqlContext.sql for specifying SQL query.

Let's get maximum purchase of each Age group in train_table.

```
sqlContext.sql('select Age, max(Purchase) from train_table group by Age').show()
Output:
+-----+-----+
|  Age|  _c1|
+-----+-----+
|51-55|23960|
|46-50|23960|
| 0-17|23955|
|36-45|23960|
|26-35|23961|
|  55+|23960|
|18-25|23958|
+-----+-----+
```

7. Pandas vs PySpark DataFrame

Pandas and Spark DataFrame are designed for structural and semistructral data processing. Both share some similar properties (which I have discussed above). The few differences between Pandas and PySpark DataFrame are:

- Operation on Pyspark DataFrame run parallel on different nodes in cluster but, in case of pandas it is not possible.
- Operations in PySpark DataFrame are lazy in nature but, in case of pandas we get the result as soon as we apply any operation.
- In PySpark DataFrame, we can't change the DataFrame due to it's immutable property, we need to transform it. But in pandas it is not the case.
- Pandas API support more operations than PySpark DataFrame. Still pandas API is more powerful than Spark.
- Complex operations in pandas are easier to perform than Pyspark DataFrame

In addition to above points, Pandas and Pyspark DataFrame have some basic differences like columns selection, filtering, adding the columns, etc. which I am not covering here.

The section not covered here in Analytics Vidhya article are covered above in Spark Summit article, to an extent.