

Big Data Analytics with pyspark

Module 3 | Chapter 2 | Notebook 4

In this notebook we'll focus on the syntax of `pyspark.sql` and use it to analyze the hard-disk data set. By the end of this exercise you will have done the following:

- Registered a `pyspark.sql.DataFrame` as an SQL table
 - Aggregated the data in your `pyspark.sql.DataFrame` with SQL queries
 - applied a Python function to the values within the `pyspark.sql.DataFrame`
-

The hard-disk data set

Scenario: You are an employee in a large data center and are tasked with investigating server hard drive failures. Thanks to the monitoring team's excellent work, you have the error data of the last quarter for all the hard drives that your data center has in operation - roughly 30000.

Your boss has two questions for you:

1. Which hard drive models are responsible for the failures?
2. Which hard driver manufacturer should the company order replacement hard disks from?

The monitoring team has stored the data in the *HDD_logs/* folder.

You are running out of time, your boss is still waiting for answers to their questions. Let's get to work and use `pyspark` to solve the task.

First import all the functions you need to establish a connection to Spark. We will also need the functions from the `pyspark.sql.functions`. Import this submodule with the alias `F` to reduce the amount of typing you have to do.

```
In [8]: from pyspark.sql import SparkSession
from pyspark.sql.functions import *
import os
```

Now create the connection to Spark.

Ignore warning messages if there are any.

```
In [6]: # Locate Data
data_dir = "HDD_logs/"

# connect to Spark
spark = (SparkSession.builder.appName("Python Spark SQL HDD Analysis").getOrCreate())
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/usr/local/spark-3.2.0-bin-hadoop3.2/jars/spark-unsafe_2.12-3.2.0.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/05/01 21:14:04 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
```

In the next step we have to import the log files into a `pyspark.sql.DataFrame`. When you handle large amounts of data, it always takes longer to process and a cell can take several minutes to execute. This makes it all the more annoying if you have to run the cell again because of a typing error. In big-data contexts, we recommend testing your code with a small part of the data set first, and then you only import the rest of the data once you're sure everything works.

Write a loop that iterates through the first three files in `data_dir` and merges the CSV files into a `pyspark.sql.DataFrame` named `fd`.

We suggest using the following steps:

1. Store all file names in the folder to a list and sort them.
2. Create a `DataFrame` out of the first file in the list.
3. Write a loop that creates DataFrames from the next two files in the list and adds them to the first `DataFrame`.

```
In [14]: file_list = sorted(os.listdir(data_dir))

df = spark.read.csv(data_dir + file_list[0], header=True);
print(df.count())

for file in file_list[1:3]:
    df_temp = spark.read.csv(data_dir + file, header=True);
    print(df_temp.count())
    df = df.union(df_temp)

print(df.count())
```

```
30597
30595
30623
91815
```

You've imported the test data. Now we have a `pyspark.sql.DataFrame` from three files with 91815 lines. First let's get an overview of the values. Just like in `pandas`, we can use `my_spark_df.describe()` for this. To make the display clearer, we'll first restrict ourselves to

the columns with the metadata and the 'failure' column. We've put these together for you in `metacols`. Run the following Code cell:

```
In [12]: metacols = ['date', 'serial_number', 'model', 'capacity_bytes', 'failure']
```

Return the summary of `df`.

Tip: Remember to combine your *transformation* with an *action* to display the data.

```
In [18]: df[metacols].describe().show()
```

```
+-----+-----+-----+-----+-----+
|summary|      date|serial_number|          model|    capacity_bytes|
failure|
+-----+-----+-----+-----+-----+
|  count|    91815|        91815|        91815|          91815|
91815|
|   mean|     null|         null|         null|3.725781544100114...|1.5248053
15035669...|
|  stddev|     null|         null|         null|9.215167404755322E11|0.0123474
26521042365|
|    min|2016-01-01|    13H2B97AS|HGST  HDS5C4040ALE630|    1500301910016|
0|
|    max|2016-01-03|    Z4D2B5EC|    WDC  WD800LB|    80026361856|
1|
+-----+-----+-----+-----+-----+
```

Each column contains 91815 values, according to `count`. Some values in the summary consist of `zero`. These are missing values, which are displayed in `pandas` as `NaN`. These are there because it's not possible to calculate a mean value or a standard deviation value from texts. Furthermore, the mean value (`mean`) in the 'failure' column has a strange value, because it's above the maximum.

Take a closer look at this value by first selecting the `failure` column from the overview. Use the `my_spark_df.collect()` *action* at the end of your query and store the output in the variable `failure_describe_values`.

Important: The `my_spark_df.collect()` *action* stores all the data from a `pyspark.sql.DataFrame` in the memory. So only use `my_spark_df.collect()` if you're requesting small amounts of data (<2GB), like in this exercise. Otherwise you will get a `MemoryError`.

```
In [24]: failure_describe_values = df[metacols].describe()["failure"].collect()
```

```
-----  
TypeError                                Traceback (most recent call last)  
Input In [24], in <module>  
----> 1 failure_describe_values = df[metacols].describe()["failure"].collect()  
  
TypeError: 'Column' object is not callable
```

We used the `failure_describe_values = df[metacols].describe()["failure"].collect()` command and got a `TypeError`.

If we take a closer look at the error message, we learn that `df[metacols].describe()["failure"]` has returned a `Column` object, but `my_spark_df.collect()` requires a `DataFrame`. The `Column` is not really similar to a `pandas.Series`, because it hardly offers any functionality.

To select a single (or multiple) column(s) as a `DataFrame`, we need the `my_spark_df.select()` method. We can give her one or more column names as strings or a `list`. You can also use `my_spark_df.select()` to make the selection with square brackets.

We repeat the command from the last code cell, but now we use `my_spark_df.select()` to select the `'failure'` column:

```
In [25]: failure_describe_values = df[metacols].describe().select("failure").collect()  
failure_describe_values
```

```
Out[25]: [Row(failure='91815'),  
          Row(failure='1.5248053150356697E-4'),  
          Row(failure='0.012347426521042365'),  
          Row(failure='0'),  
          Row(failure='1')]
```

As you can see, the mean value in the second row is less than 1 (the maximum). Only the exponent of the scientific notation was cut off in the display with `my_spark_df.show()`. So the data is all ok.

Congratulations: You have got to know the *action* `my_spark_df.collect()`, which allows you to store *transformations*. Also, you now know an alternative to making a selection with square brackets: `my_spark_df.select()`.

SQL queries with Spark

Next we'll focus on feature engineering. So far we haven't come across a column containing information about the hard disk manufacturers. So we should check if we can extract the manufacturer from the model number.

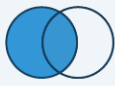


SQL - Selection from a table

Command description:	
Simple selection: Select the values of one or more columns from a table	SELECT column_name FROM table name
Selecting unique values: Eliminate duplicates in the result	SELECT DISTINCT column_name FROM table name
Limited selection: Select only the first elements.	SELECT .. FROM .. LIMIT number
Sorted selection: Sort the result according to the values of a column. Ascending (ASC) or descending (DESC).	SELECT .. FROM .. ORDER BY column name DESC
Conditional selection: Only select rows that fulfill a condition.	SELECT .. FROM .. WHERE condition
Aggregated selection: Group the data by a column and apply an aggregation function to the data.	SELECT aggregation (column name) FROM .. GROUP BY column name
Aggregated selection with condition: Group the data by a column, aggregate it, and select only those results that meet the condition.	SELECT aggregation (column name) FROM .. GROUP BY column name HAVING condition

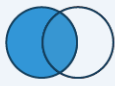


Aggregation functions	
Calculate the sum	SUM()
Determine number	COUNT()
Calculate average	AVG()
Determine maximum value	MAX()
Determine minimum value	MIN()

Conditions	
Equal to	= or IS
And – connector	AND
Or – connector	OR
Negation	NOT
Pattern matching	LIKE pattern
Included in Subset	IN (value_1, value_2, ..)

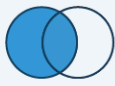


SQL - Merging two tables

Description	Command	Scheme
Left Join: Use only the rows from tablename_1 Typical Condition: table_name_1.key_1 = table_name_2.key_2	<pre>SELECT column_name_1, column_name_2 FROM table_name_1 LEFT JOIN table_name_2 ON condition</pre>	
Inner Join: Use only the lines that appear in both tables	<pre>SELECT column_name_1, column_name_2 FROM table_name_1 INNER JOIN table_name_2 ON condition</pre>	
Outer Join: Use all lines that occur in at least one of the tables	<pre>SELECT column_name_1, column_name_2 FROM table_name_1 FULL OUTER JOIN table_name_2 ON condition</pre>	

SQL - Merging two tables

Description	Command	Scheme
Left Join: Use only the rows from tablename_1 Typical Condition: table_name_1.key_1 = table_name_2.key_2	<pre>SELECT column_name_1, column_name_2 FROM table_name_1 LEFT JOIN table_name_2 ON condition</pre>	
Inner Join: Use only the lines that appear in both tables	<pre>SELECT column_name_1, column_name_2 FROM table_name_1 INNER JOIN table_name_2 ON condition</pre>	
Outer Join: Use all lines that occur in at least one of the tables	<pre>SELECT column_name_1, column_name_2 FROM table_name_1 FULL OUTER JOIN table_name_2 ON condition</pre>	

SQL - Merging two tables

Description	Command	Scheme
Left Join: Use only the rows from tablename_1 Typical Condition: table_name_1.key_1 = table_name_2.key_2	SELECT column_name_1, column_name_2 FROM table_name_1 LEFT JOIN table_name_2 ON condition	
Inner Join: Use only the lines that appear in both tables	SELECT column_name_1, column_name_2 FROM table_name_1 INNER JOIN table_name_2 ON condition	
Outer Join: Use all lines that occur in at least one of the tables	SELECT column_name_1, column_name_2 FROM table_name_1 FULL OUTER JOIN table_name_2 ON condition	

The `pyspark.sql` module provides us with a variety of building blocks that help us construct queries to output exactly the data from large datasets that we actually need. As the name of the module suggests, the syntax for using these modules is based on SQL. In fact, SQL queries can easily be applied to DataFrames from Spark.

Here is a selection of the different SQL commands you can use with `pyspark.sql`. This overview is also available as a download in this chapter:



Before we can use a `DataFrame` for an SQL query, we have to register it as an SQL table in our `SparkSession`. You can do this with the `my_spark_df.registerTempTable()` command.

We pass a `str` to this method containing the table name.

`my_spark_df.registerTempTable()` does not return an output value that we would have to then assign.

First create a `DataFrame` from `df`, which only contains `metacols` as columns. Call this `DataFrame` `df_meta`. Then register `df_meta` as an SQL table under the table name `'meta_data'`.

```
In [27]: df_meta = df[metacols]
df_meta.registerTempTable('meta_data')
```

```
/usr/local/spark/python/pyspark/sql/dataframe.py:138: FutureWarning: Deprecated in 2.0, use createOrReplaceTempView instead.
  warnings.warn(
```

We can now send SQL queries to the `DataFrame` using the `spark.sql()` command. Here again, you have to add an *action* to get the data as an output.

Now we want to select all the different disk models and store them sorted alphabetically in the variable `model_names`. So we need a query that only selects unique values and then sorts the result.

The best way is to write a query as a `str` with line breaks. This makes it easier to read and understand. We can do this by using three consecutive quotation marks. Our queries always begin with `'SELECT'`. This is then followed by the column name we want to query, so in this case it's `'model'`. To make sure we only get each value only once, we use `'DISTINCT'` on the column names. We give this selection the alias `'model_name'` by using the keyword `'AS'`. This is followed by `'FROM'` and the table name specification we defined previously. To sort the values, we use the `'ORDER BY'` in combination with the alias we want to sort by.

```
In [30]: query = """SELECT DISTINCT(model) AS model_name
FROM meta_data
ORDER BY model_name"""
```

Now select all the different disk models using `spark.sql()` and store them alphabetically in the variable `model_names`.

```
In [33]: model_names = spark.sql(query)
model_names.show()
```

```

+-----+
|          model_name|
+-----+
|HGST HDS5C4040ALE630|
|HGST HDS724040ALE640|
|HGST HMS5C4040ALE640|
|HGST HMS5C4040BLE640|
|HGST HUH728080ALE600|
|Hitachi HDS5C3030...|
|Hitachi HDS5C3030...|
|Hitachi HDS5C4040...|
|Hitachi HDS722020...|
|Hitachi HDS723020...|
|Hitachi HDS723030...|
|Hitachi HDS723030...|
|Hitachi HDS724040...|
|          ST2000VN000|
|          ST250LM004 HN|
|          ST250LT007|
|          ST31500541AS|
|          ST3160316AS|
|          ST3160318AS|
|          ST320LT007|
+-----+
only showing top 20 rows

```

If we look through the selected model numbers, it looks like we have five different hard drive manufacturers in our data set:

- HGST: Hitachi Global Storage Technology (bought by Western Digital)
- Hitachi: Hitachi
- ST: Seagate
- Toshiba: Toshiba
- WDC: Western Digital

On closer inspection, it becomes clear that the model names are systematically assigned to each manufacturer. We can take advantage of this to extract the manufacturer's name from it. We've prepared a function to take the model name as an input and prints the manufacturer name.

```

In [34]: #define rules for tagging the model names
def clean_name(name):
    """Extract HDD Brand from model_name.
    Args:
        name (str) : Model name of HDD
    Returns:
        str : Brand Tag of HDD
    """
    if name[:2] == "ST": # seagate has no space in model number
        brand = "Seagate"
    else:
        brand = name.split(" ")[0]

    # check if brand was saved
    if brand:
        return brand

```

```

else:
    print("Could not extract Brand from: {}".format(name))
    return "Other"

```

Now we have the `clean_name` function, which we only need to apply to each value in the `'model'` column to write the return values into a new column called `'brand'`. However, now we have the problem that we've written a Python function, but we can't use this directly with Spark. So we need to convert our function first, into a **user defined function** format, or `udf` for short.

The `udf()` function from the `pyspark.sql.functions` module will take care of the conversion for us. Remember that we have already imported this module with the alias `F`. Use `udf()` to convert `clean_name`. This returns a function object. Store it as `clean_name_udf`.

```
In [35]: clean_name_udf = udf(clean_name)
```

If we want to manipulate the column values of a Spark DataFrame and add the results as a new column or use them replace the existing ones, we need the `my_spark_df.withColumn()` method. It has the parameters `colName` and `col`. `colName` receives the column name as `str`. `col` receives a 'column' object or a function that manipulates a column.

Use `my_spark_df.withColumn()` to add a column named `'brand'` to `df_meta`, which is created by using the `clean_name_udf()` function on `'model'`. Remember to reassign the output back to `df_meta`. Then display `df_meta` again.

```
In [37]: df_meta = df_meta.withColumn(colName='brand', col=clean_name_udf('model'))
df_meta.show()
```

date	serial_number	model	capacity_bytes	failure	brand
2016-01-01	MJ0351YNG9Z0XA	Hitachi HDS5C3030...	3000592982016	0	Hitachi
2016-01-01	Z305B2QN	ST4000DM000	4000787030016	0	Seagate
2016-01-01	MJ0351YNG9Z7LA	Hitachi HDS5C3030...	3000592982016	0	Hitachi
2016-01-01	MJ0351YNGABYAA	Hitachi HDS5C3030...	3000592982016	0	Hitachi
2016-01-01	PL1321LAG34XWH	Hitachi HDS5C4040...	4000787030016	0	Hitachi
2016-01-01	S300Z6K9	ST4000DM000	4000787030016	0	Seagate
2016-01-01	PL1311LAG2205A	Hitachi HDS5C4040...	4000787030016	0	Hitachi
2016-01-01	MJ1311YNG36USA	Hitachi HDS5C3030...	3000592982016	0	Hitachi
2016-01-01	Z304K3TJ	ST4000DM000	4000787030016	0	Seagate
2016-01-01	Z300CMV1	ST4000DM000	4000787030016	0	Seagate
2016-01-01	PL1331LAGSPEUH	HGST HMS5C4040ALE640	4000787030016	0	HGST
2016-01-01	Z304JCBV	ST4000DM000	4000787030016	0	Seagate
2016-01-01	MK0313YHGI54WC	Hitachi HDS723030...	3000592982016	0	Hitachi
2016-01-01	MJ1311YNG733NA	Hitachi HDS5C3030...	3000592982016	0	Hitachi
2016-01-01	PL1321LAG33WSH	Hitachi HDS5C4040...	4000787030016	0	Hitachi
2016-01-01	PL2331LAGSSBMJ	HGST HMS5C4040ALE640	4000787030016	0	HGST
2016-01-01	PL1331LAGTU67H	HGST HMS5C4040ALE640	4000787030016	0	HGST
2016-01-01	Z4D06EFX	ST6000DX000	6001175126016	0	Seagate
2016-01-01	Z304JE2N	ST4000DM000	4000787030016	0	Seagate
2016-01-01	MJ1323YNG1U3YC	Hitachi HDS5C3030...	3000592982016	0	Hitachi

only showing top 20 rows

```
Traceback (most recent call last):
  File "/usr/local/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 186, in manager
  File "/usr/local/spark/python/lib/pyspark.zip/pyspark/daemon.py", line 74, in worker
  File "/usr/local/spark/python/lib/pyspark.zip/pyspark/worker.py", line 663, in main
    if read_int(infile) == SpecialLengths.END_OF_STREAM:
  File "/usr/local/spark/python/lib/pyspark.zip/pyspark/serializers.py", line 564, in read_int
    raise EOFError
EOFError
```

`df_meta` should now look like this:



Now you can also see the `'brand'` column in the `DataFrame`. However, this change has not yet been applied in the background in the SQL table `'meta_data'`. For this to happen, we need to register `'meta_data'` again. Then the new data is also available for SQL queries.

```
In [39]: df.registerTempTable('meta_data')
```

Congratulations: You've successfully applied an SQL query in Spark. Now you can aggregate the data using SQL queries and answer the question. Let's do that right away!

Which hard disk models are responsible for the failures?

Calculate the failure rate per disk model to answer the question. You can decide to answer it by constructing the two queries in stages, or with a single SQL query containing a subquery. In the first case, continue under **"For SQL beginners"** and in the second case skip ahead to **"For SQL pros"**.

For SQL beginners: Instead of one nested query, we'll use 2 queries. Now we'll sort out the first one. First we'd like to get the number of errors per model number as well as the total number of hard disks for each model number. Individual hard drives have different serial numbers.

So we'll select 3 columns with `'SELECT': 'model', 'failure' and 'serial_number'`. We need the sum from `'failure'` and we need the number of unique values for each model from `'serial_number'`. Then group this selection by the `'model'` column.

Use aliases for the columns to make your query and display clearer. You can assign an alias to a column by writing `AS my_alias` after the column selection, for example:

```
"""SELECT SUM(a) AS summe
FROM mytable"""
```

Run the query with `spark.sql(my_query).show()` and display it immediately.

```
In [46]: my_query = """SELECT model, SUM(failure) AS fails, COUNT(DISTINCT(serial_number)) as n
spark.sql(my_query).show()
```

```
+-----+-----+-----+
|          model|fails|no_model|
+-----+-----+-----+
|      ST9250315AS|  0.0|      30|
|      ST4000DM000| 10.0|    15595|
|      ST320LT007|  0.0|       59|
|      ST3500320AS|  0.0|        1|
|      WDC WD5002ABYS|  0.0|        4|
|      WDC WD2500BEVT|  0.0|        1|
|Hitachi HDS5C3030...|  0.0|    2409|
|      WDC WD60EFRX|  0.0|     245|
|      WDC WD800AAJB|  0.0|        4|
|Hitachi HDS722020...|  1.0|    2307|
|      TOSHIBA DT01ACA300|  0.0|       23|
|HGST HMS5C4040ALE640|  0.0|    3751|
|      WDC WD3200BEKT|  0.0|        1|
|      TOSHIBA MD04ABA500V|  0.0|       23|
|      TOSHIBA MD04ABA400V|  0.0|       73|
|      WDC WD1600BPVT|  0.0|        1|
|Hitachi HDS723020...|  0.0|        4|
|      WDC WD800BB|  0.0|        4|
|      WDC WD2500AAJS|  0.0|        2|
|      WDC WD5003ABYX|  0.0|        2|
+-----+-----+-----+
only showing top 20 rows
```

You should get the following result:



The query we've just written provides a table with values that we need in order to calculate the failure rate. By using a subquery, we can tell SQL to treat the result of this query like a table of its own. To do this, we copy the query code in round brackets into our main query instead of the table name. The code looks like this:

```
"""SELECT sum_a / num_a AS ratio
FROM (SELECT SUM(a) AS sum_a,
COUNT(a) AS num_a
FROM table)
ORDER BY ratio DESC"""
```

Now copy the code from your first query and use it as subquery.

The task for the main query is: Select all the columns (with `'*'`) from the subquery as well as the division of the sum and number from the subquery. Order the results by failure rate and display the ten most unreliable hard drive models.

```
In [48]: my_query = """
select * , fails/no_model as failure_rate
from
(SELECT model,
SUM(failure) AS fails,
```

```

COUNT(DISTINCT(serial_number)) as no_model
FROM meta_data group by model)
ORDER BY failure_rate DESC
"""

```

```
spark.sql(my_query).show()
```

```

+-----+-----+-----+-----+
|          model|fails|no_model|failure_rate|
+-----+-----+-----+-----+
|WDC WD20EFRX|2.0|67|0.029850746268656716|
|Hitachi HDS5C4040...|1.0|1413|7.077140835102619E-4|
|ST4000DM000|10.0|15595|6.412311638345624E-4|
|Hitachi HDS722020...|1.0|2307|4.334633723450368...|
|ST9250315AS|0.0|30|0.0|
|WDC WD5002ABYS|0.0|4|0.0|
|ST320LT007|0.0|59|0.0|
|WDC WD2500BEVT|0.0|1|0.0|
|Hitachi HDS5C3030...|0.0|2409|0.0|
|WDC WD800AAJB|0.0|4|0.0|
|ST3500320AS|0.0|1|0.0|
|TOSHIBA MD04ABA500V|0.0|23|0.0|
|WDC WD1600BPVT|0.0|1|0.0|
|WDC WD800BB|0.0|4|0.0|
|WDC WD60EFRX|0.0|245|0.0|
|TOSHIBA DT01ACA300|0.0|23|0.0|
|HGST HMS5C4040ALE640|0.0|3751|0.0|
|WDC WD3200BEKT|0.0|1|0.0|
|TOSHIBA MD04ABA400V|0.0|73|0.0|
|Hitachi HDS723020...|0.0|4|0.0|
+-----+-----+-----+-----+

```

only showing top 20 rows

You should get following result:



According to the first three files, the model 'WDC WD20EFRX' has the highest failure rate.

For SQL pros: Calculate the failure rate directly with a query containing a subquery. To do this, group the 'meta_data' table by the model name(s) ('model') and get the average failure rate (('failure') per number of the respective model numbers ('model')). Order the Events by failure rate and display the ten most unreliable hard drive models.

```

In [ ]: my_query = """

        select * , fails/no_model as failure_rate
        from
            (SELECT model,
              SUM(failure) AS fails,
              COUNT(DISTINCT(serial_number)) as no_model
              FROM meta_data group by model)
        ORDER BY failure_rate DESC
        """

spark.sql(my_query).show()

```

The result for the entire query should look like something like this:



According to the first three files, the model `'WDC WD20EFRX'` has the highest failure rate.

Congratulations: You created an SQL query and used it to find out what the failure rate of each hard disk model is. You've already answered the first question. Your boss will be pleased: They can now include the default rates in their cost considerations. Now let's look at the second question.

Which manufacturer should replacement hard drives be ordered from?

Use the `meta_data` table to find out which manufacturer is expected to have the lowest failure rate. The manufacturer is located in the `'brand'` column. Otherwise, the query is very similar to the previous one.

In []:

We find that, `'HGST'` and `'TOSHIBA'` are the brands with the lowest failure rate:



Your code seems to work. It's time to apply it to the entire data set.

Restart your kernel and go right back to where you import the data into a `DataFrame`. As you may remember, we only imported the first three files to start with. Now change the code so that all the files are imported, and run all the cells again. Do the worst hard drive models and the best hard drive manufacturers change?

Tip: If you comment out the code in cell six and select the option "Restart Kernel and Run all Cells" when restarting the kernel, you can go and get a coffee and look at the results once you get back. To do this, you also have to remove the contents of the code cell that create a `TypeError`, otherwise it will interrupt execution.

For us, `'HGST'` is still the manufacturer with the lowest failure rate. These brands have some models with high failure rates:



Now close the `SparkSession`.

In []:

Congratulations: Your boss is very happy with your analysis! Thanks to you, the data center can weigh the cost of the hard drives against their failure rate. In the next exercise we'll finally look

at how we can do machine learning with Spark.

Remember:

- The `my_spark_df.select()` method always returns a `DataFrame` even if you only select a single column.
- To apply custom functions to the `pyspark.sql.DataFrame`, you have to convert them to an `udf` object with `pyspark.sql.functions.udf()`.
- You can register a `pyspark.sql.DataFrame` as a table in Spark with `my_spark_df.registerTempTable('my_TableName')` and then apply SQL queries to it with `my_SparkSession.sql()`.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
