Welcome to exercise one of week two of “Apache Spark for Scalable Machine Learning on BigData”. In this exercise you’ll read a DataFrame in order to perform a simple statistical analysis. Then you’ll rebalance the dataset. No worries, we’ll explain everything to you, let’s get started.

Let’s create a data frame from a remote file by downloading it:

In [17]:
```
# delete files from previous runs
!rm -f hmp.parquet*

# download the file containing the data in PARQUET format
!wget https://github.com/IBM/coursera/raw/master/hmp.parquet

# create a dataframe out of it
df = spark.read.parquet('hmp.parquet')

# register a corresponding query table
df.createOrReplaceTempView('df')
```

```
--2019-08-14 08:43:59--  https://github.com/IBM/coursera/raw/master/hm
p.parquet
Resolving github.com (github.com)... 192.30.253.113
Connecting to github.com (github.com)|192.30.253.113|:443... connecte
d.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/IBM/coursera/master/hmp.pa
rquet [following]
--2019-08-14 08:43:59--  https://raw.githubusercontent.com/IBM/courser
a/master/hmp.parquet
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 15
1.101.4.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|15
1.101.4.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 932997 (911K) [application/octet-stream]
Saving to: 'hmp.parquet'

hmp.parquet         100%[===================>] 911.13K  4.99MB/s    in
0.2s

2019-08-14 08:43:59 (4.99 MB/s) - 'hmp.parquet' saved [932997/932997]
```

Let’s have a look at the data set first. This dataset contains sensor recordings from different movement activities as we will see in the next week’s lectures. X, Y and Z contain accelerometer sensor values whereas the class field contains information about which movement has been recorded. The source field is optional and can be used for data lineage since it contains the file name of the original file where the particular row was importat from.

More details on the data set can be found here: https://github.com/wchill/HMP_Dataset

In [ ]:
```
df.show()
df.printSchema()
```

This is a classical classification data set. One thing we always do during data analysis is checking if the classes are balanced. In other words, if there are more or less the same number of example in each class. Let's find out by a simple aggregation using SQL.

```
In [ ]: spark.sql('select class,count(*) from df group by class').show()
```

As you can see there is quite an imbalance between classes. Before we dig into this, let's re-write the same query using the DataFrame API – just in case you are not familiar with SQL. As we've learned before, it doesn't matter if you express your queries with SQL or the DataFrame API – it all gets boiled down into the same execution plan optimized by Tungsten and accelerated by Catalyst. You can even mix and match SQL and DataFrame API code if you like.

Again, more details on the API can be found here:https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame

```
In [ ]: df.groupBy('class').count().show()
```

Let's create a bar plot from this data. We're using the pixidust library, which is Open Source, because of its simplicity. But any other library like matplotlib is fine as well.

```
In [ ]: import pixiedust
        from pyspark.sql.functions import col
        counts = df.groupBy('class').count().orderBy('count')
        display(counts)
```

This looks nice, but it would be nice if we can aggregate further to obtain some quantitative metrics on the imbalance like, min, max, mean and standard deviation. If we divide max by min we get a measure called minmax ration which tells us something about the relationship between the smallest and largest class. Again, let's first use SQL for those of you familiar with SQL. Don't be scared, we're used nested sub-selects, basically selecting from a result of a SQL query like it was a table. All within on SQL statement.

```
In [ ]: spark.sql('''
            select
                *,
                max/min as minmaxratio -- compute minmaxratio based on previou
        sly computed values
                from (
                    select
                        min(ct) as min, -- compute minimum value of all classe
        s
                        max(ct) as max, -- compute maximum value of all classe
        s
                        mean(ct) as mean, -- compute mean between all classes
                        stddev(ct) as stddev -- compute standard deviation bet
        ween all classes
                        from (
                            select
                                count(*) as ct -- count the number of rows per
        class and rename it to ct
                                from df -- access the temporary query table ca
        lled df backed by DataFrame df
                                group by class -- aggrecate over class
                        )
                )
        ''').show()
```

The same query can be expressed using the DataFrame API. Again, don't be scared. It's just a sequential expression of transformation steps. You now an choose which syntax you like better.

```
In [ ]: from pyspark.sql.functions import col, min, max, mean, stddev

        df \
            .groupBy('class') \
            .count() \
            .select([
                min(col("count")).alias('min'),
                max(col("count")).alias('max'),
                mean(col("count")).alias('mean'),
                stddev(col("count")).alias('stddev')
            ]) \
            .select([
                col('*'),
                (col("max") / col("min")).alias('minmaxratio')
            ]) \
            .show()
```

Now it's time for you to work on the data set. First, please create a table of all classes with the respective counts, but this time, please order the table by the count number, ascending.

```
In [ ]: $$$ your code goes here
```

Pixiedust is a very sophisticated library. It takes care of sorting as well. Please modify the bar chart so that it gets sorted by the number of elements per class, ascending. Hint: It's an option available in the UI once rendered using the display() function.

```
In [ ]: $$$ your code goes here
```

Imbalanced classes can cause pain in machine learning. Therefore let's rebalance. In the flowing we limit the number of elements per class to the amount of the least represented class. This is called undersampling. Other ways of rebalancing can be found here:

https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/

```
In [19]: from pyspark.sql.functions import min

         # create a lot of distinct classes from the dataset
         classes = [row[0] for row in df.select('class').distinct().collect()]

         # compute the number of elements of the smallest class in order to lim
         it the number of samples per calss
         min = df.groupBy('class').count().select(min('count')).first()[0]

         # define the result dataframe variable
         df_balanced = None

         # iterate over distinct classes
         for cls in classes:

             # only select examples for the specific class within this iteratio
         n
             # shuffle the order of the elements (by setting fraction to 1.0 sa
         mple works like shuffle)
             # return only the first n samples
             df_temp = df \
                 .filter("class = '"+cls+"'") \
                 .sample(False, 1.0) \
                 .limit(min)

             # on first iteration, assing df_temp to empty df_balanced
             if df_balanced == None:
                 df_balanced = df_temp
             # afterwards, append vertically
             else:
                 df_balanced=df_balanced.union(df_temp)
```

Please verify, by using the code cell below, if df_balanced has the same number of elements per class. You should get 6683 elements per class.

```
In [ ]: $$$
```