# Cluster Analysis in Spark

### Problem Description

This activity guides you through the process of performing cluster analysis on a dataset using k-means. In this activity, we will perform cluster analysis on the *minute-weather.csv* dataset using the k-means algorithm. Recall that this dataset contains weather measurements such as temperature, relative humidity, etc., from a weather station in San Diego, California, collected at one-minute intervals. The goal of cluster analysis on this data is to identify different weather patterns for this weather station.

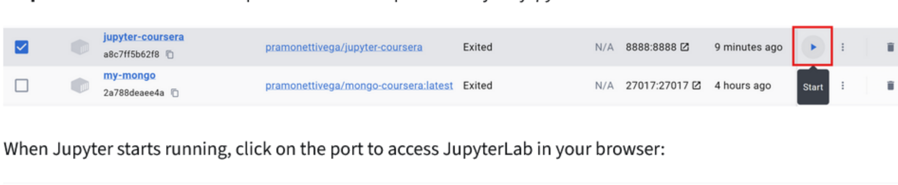
### Learning Objectives

By the end of this activity, you will be able to:

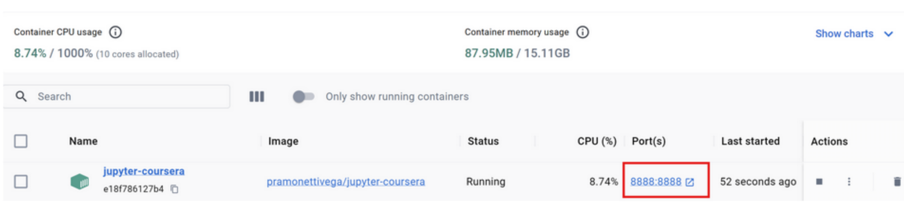
1. Scale all features so that each feature is zero-normalized
2. Create an "elbow" plot, the number of clusters vs. within-cluster sum-of-squared errors, to determine a value for k, the number of clusters in k-means
3. Perform cluster analysis on a dataset using k-means
4. Create parallel coordinates plots to analyze cluster centers

For this activity, you should have completed the creation of the JupyterLab container. If not follow, Steps 1-3 on the previous activity *Hand On: Data Exploration in Spark*, and then come back to Step 2 of this activity.

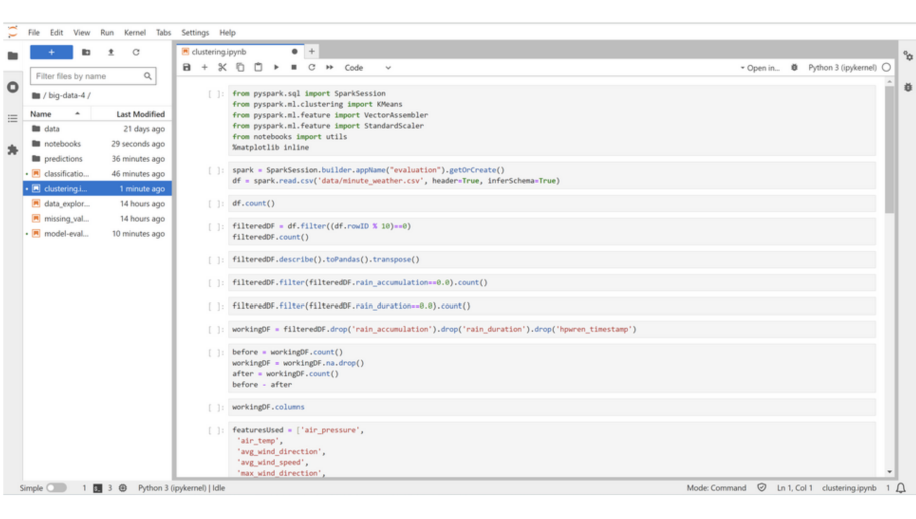
**Step 1. Start the container.** Open Docker Desktop and start your *jupyter-coursera* container.




When Jupyter starts running, click on the port to access JupyterLab in your browser:




**Step 2. Open your notebook.** Once you're in JupyterLab, go to the *big-data-4* folder and open the *clustering.ipynb* notebook.



**Step 3. Load minute weather data.** Execute the first cell to load the classes used in this activity:

```
[1]: from pyspark.sql import SparkSession
     from pyspark.ml.clustering import KMeans
     from pyspark.ml.feature import VectorAssembler
     from pyspark.ml.feature import StandardScaler
     from notebooks import utils
     %matplotlib inline
```

Execute the second cell to load the minute weather data in *minute_weather.csv:*

```
[2]: spark = SparkSession.builder.appName("evaluation").getOrCreate()
     df = spark.read.csv('data/minute_weather.csv', header=True, inferSchema=True)
```

**Step 4. Subset and remove unused data**. Let's count the number of rows in the DataFrame:

```
[3]: df.count()
[3]: 1587257
```

There are over 1.5 million rows in the DataFrame. Clustering this data on your computer in can take a long time, so let's use only one-tenth of the data. We can subset by calling *filter()* and using the *rowID* column:

```
[4]: filteredDF = df.filter((df.rowID % 10)==0)
     filteredDF.count()
[4]: 158726
```

Let's compute the summary statistics using *describe():*

```
[5]: filteredDF.describe().toPandas().transpose()
```

[5]:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **summary** | count | mean | stddev | min | max |
| **rowID** | 158726 | 793625.0 | 458203.9375103623 | 0 | 1587250 |
| **air_pressure** | 158726 | 916.8301614102267 | 3.0517165528306345 | 905.0 | 929.5 |
| **air_temp** | 158726 | 61.851589153636155 | 11.833569210641658 | 31.64 | 99.5 |
| **avg_wind_direction** | 158680 | 162.15610032770354 | 95.27820101905915 | 0.0 | 359.0 |
| **avg_wind_speed** | 158680 | 2.775214897907738 | 2.0576239697426457 | 0.0 | 31.9 |
| **max_wind_direction** | 158680 | 163.46214393748426 | 92.452138538387 | 0.0 | 359.0 |
| **max_wind_speed** | 158680 | 3.4005577262415136 | 2.4188016208098877 | 0.1 | 36.0 |
| **min_wind_direction** | 158680 | 166.77401688933702 | 97.44110914784562 | 0.0 | 359.0 |
| **min_wind_speed** | 158680 | 2.1346641038568066 | 1.7421125052424375 | 0.0 | 31.6 |
| **rain_accumulation** | 158725 | 3.178453299732832E-4 | 0.011235979086039793 | 0.0 | 3.12 |
| **rain_duration** | 158725 | 0.4096267128681682 | 8.665522693479785 | 0.0 | 2960.0 |
| **relative_humidity** | 158726 | 47.60946977810823 | 26.214408535062063 | 0.9 | 93.0 |

The weather measurements in this dataset were collected during a drought in San Diego. We can count the how many values of rain accumulation and duration are 0:

```
[6]: filteredDF.filter(filteredDF.rain_accumulation==0.0).count()
[6]: 157812
```

```
[7]: filteredDF.filter(filteredDF.rain_duration==0.0).count()
[7]: 157237
```

Since most the values for these columns are 0, let's drop them from the DataFrame to speed up our analyses. We can also drop the *hpwren_timestamp* column since we do not use it.

```
[8]: workingDF = filteredDF.drop('rain_accumulation').drop('rain_duration').drop('hpwren_timestamp')
```

Let's drop rows with missing values and count how many rows were dropped:

```
[9]: before = workingDF.count()
     workingDF = workingDF.na.drop()
     after = workingDF.count()
     before - after
[9]: 46
```

**Step 5. Scale the data**. Since the features are on different scales (e.g., air pressure values are in the 900's, while relative humidities range from 0 to 100), they need to be scaled. We will scale them so that each feature will have a value of 0 for the mean, and a value of 1 for the standard deviation.

First, we will combine the columns into a single vector column. Let's look at the columns in the DataFrame:

```
[10]: workingDF.columns
[10]: ['rowID',
       'air_pressure',
       'air_temp',
       'avg_wind_direction',
       'avg_wind_speed',
       'max_wind_direction',
       'max_wind_speed',
       'min_wind_direction',
       'min_wind_speed',
       'relative_humidity']
```

We do not want to include *rowID* since it is the row number. The minimum wind measurements have a high correlation to the average wind measurements, so we will not include them either. Let's create an array of the columns we want to combine, and use *VectorAssembler* to create the vector column:

```
[11]: featuresUsed = ['air_pressure',
       'air_temp',
       'avg_wind_direction',
       'avg_wind_speed',
       'max_wind_direction',
       'max_wind_speed', 'relative_humidity']
      assembler = VectorAssembler(inputCols=featuresUsed, outputCol='features_unscaled')
      assembled = assembler.transform(workingDF)
```

Next, let's use *StandardScaler* to scale the data:

```
[12]: scaler = StandardScaler(inputCol="features_unscaled", outputCol="features", withStd=True, withMean=True)
      scalerModel = scaler.fit(assembled)
      scaledData = scalerModel.transform(assembled)
```

The *withMean* argument specifies to center the data with the mean before scaling, and *withStd* specifies to scale the data to the unit standard deviation.

**Step 6. Create elbow plot.** The k-means algorithm requires that the value of k, the number of clusters, to be specified. To determine a good value for *k*, we will use the "elbow" method. This method involves applying k-means, using different values for *k*, and calculating the within-cluster sum-of-squared error (WSSE). Since this means applying k-means multiple times, this process can be very compute-intensive. To speed up the process, we will use only a subset of the dataset. We will take every third sample from the dataset to create this subset:

```
[13]: scaledData = scaledData.select("features","rowID")
      elbowset = scaledData.filter((scaledData.rowID % 3)==0).select("features")
      elbowset.persist()
```

The last line calls the *persist()* method to tell Spark to keep the data in memory (if possible), which will speed up the computations.

Let's compute the k-means clusters for *k* = 2 to 30 to create an elbow plot:

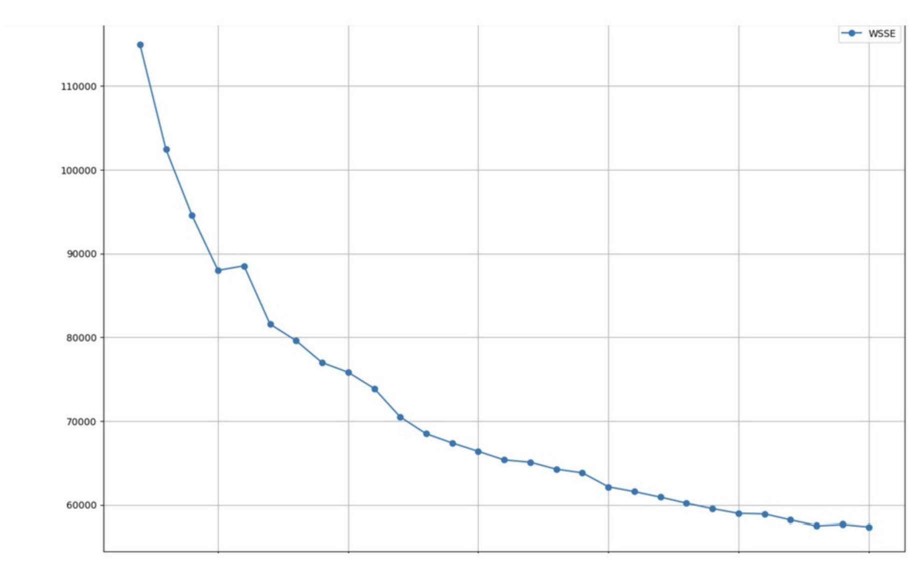
```
[14]: clusters = range(2,31)
      wsseList = utils.elbow(elbowset,clusters)

      Training for cluster size 2
      ......................WSSE = 114992.97915777276
      Training for cluster size 3
      ......................WSSE = 102466.18371986187
      Training for cluster size 4
      ......................WSSE = 94576.72056439659
      Training for cluster size 5
      ......................WSSE = 87993.46098415967
      Training for cluster size 6
      ......................WSSE = 88532.24321579303
      Training for cluster size 7
      ......................WSSE = 81578.21408692523
      Training for cluster size 8
      ......................WSSE = 79602.81644656365
```

The first line creates an array with the numbers 2 through 30, and the second line calls the *elbow()* function defined in the *utils.py* library to perform clustering. The first argument to *elbow()* is the dataset, and the second is the array of values for *k*. The *elbow()* function returns an array of the WSSE for each number of clusters.

Let's plot the results by calling *elbow_plot()* in the *utils.py* library:

```
[15]: utils.elbow_plot(wsseList,clusters)
```




The values for *k* are plotted against the WSSE values, and the elbow, or bend in the curve, provides a good estimate for the value for *k*. In this plot, we see that the elbow in the curve is between 10 and 15, so let's choose *k* = 12. We will use this value to set the number of clusters for k-means.

**Step 7. Cluster using selected k**. Let's select the data we want to cluster:

```
[16]: scaledDataFeat = scaledData.select("features")
      scaledDataFeat.persist()
[16]: DataFrame[features: vector]
```

Again, we call the *persist()* method to cache the data in memory for faster access.

We can perform clustering using *KMeans:*

```
[17]: kmeans = KMeans(k=12, seed=1)
      model = kmeans.fit(scaledDataFeat)
      transformed = model.transform(scaledDataFeat)
```

The first line creates a new *KMeans* instance with 12 clusters and a specific seed value. (As in previous hands-on activities, we use a specific seed value for reproducible results.) The second line fits the data to the model, and the third applies the model to the data.

Once the model is created, we can determine the center measurement of each cluster:

```
[18]: model.clusterCenters()
[18]: [array([ 0.13402354,  0.84479405,  1.89296205, -0.62876084, -1.54637993,
             -0.55508304, -0.75433809]),
       array([-0.17211005,  0.61532884,  0.40667   ,  0.66342113,  0.5162311 ,
              0.60561756, -0.17958117]),
       array([-1.16752155, -0.85790018,  0.44591112,  1.9725761 ,  0.53760185,
              1.93158639,  0.8997432 ]),
       array([ 0.25727752, -0.95775431,  0.6314963 , -0.54867129,  0.8479847 ,
             -0.53241394,  1.11284105]),
       array([-0.17689595,  0.86349233, -1.30999154, -0.57824377, -1.16753809,
             -0.59369794, -0.635747  ]),
       array([ 1.18818166, -0.25135089, -1.15449499,  2.08921565, -1.05215439,
              2.20534493, -1.13144893]),
       array([ 1.39522247, -0.09400324, -1.1156324 , -0.12295897, -0.97454891,
             -0.10506059, -0.97499866]),
       array([-0.25769268,  0.59401287,  0.19127265, -0.63001738,  0.36623541,
             -0.64414891, -0.38516216]),
       array([ 0.31053815,  0.77187662,  1.31035253, -0.63201795,  1.57332084,
             -0.58349867, -0.74769523]),
       array([-0.86001643, -1.16679464,  0.36413727,  0.34203153,  0.49202817,
              0.32783891,  1.34275222]),
       array([ 0.02242719, -0.76972384, -1.18982339, -0.56942298, -1.03438065,
             -0.5843519 ,  0.87713134]),
       array([ 0.37387152, -0.9786106 ,  1.85922661, -0.69276955, -1.53259165,
             -0.60261207,  0.88890541])]
```

It is difficult to compare the cluster centers by just looking at these numbers. So we will use plots in the next step to visualize them.

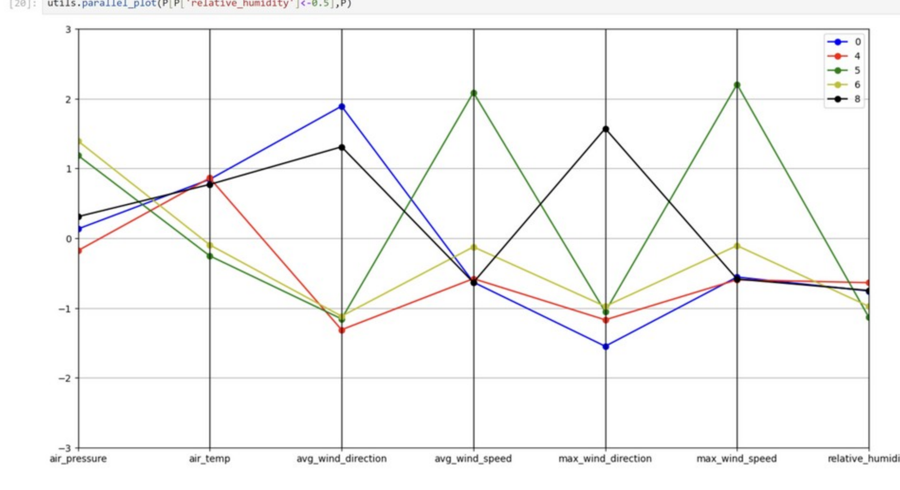
**Step 8. Create parallel plots of clusters and analysis.** A parallel coordinates plot is a great way to visualize multi-dimensional data. Each line plots the centroid of a cluster, and all of the features are plotted together. Recall that the feature values were scaled to have mean = 0 and standard deviation = 1. So the values on the y-axis of these parallel coordinates plots show the number of standard deviations from the mean. For example, +1 means one standard deviation higher than the mean of all samples, and -1 means one standard deviation lower than the mean of all samples.

We'll create the plots with *matplotlib* using a Pandas DataFrame each row contains the cluster center coordinates and cluster label. (Matplotlib can plot Pandas DataFrames, but not Spark DataFrames.) Let's use the *pd_centers()* function in the *utils.py* library to create the Pandas DataFrame:

```
[19]: P = utils.pd_centers(featuresUsed, model.clusterCenters())
```

Let's show clusters for "Dry Days", i.e., weather samples with low relative humidity:

```
[20]: utils.parallel_plot(P[P['relative_humidity']<-0.5],P)
```

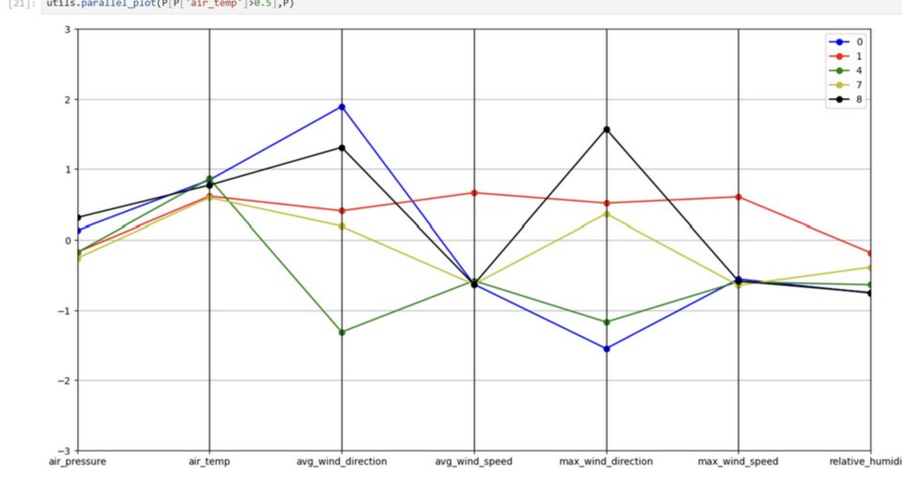



The first argument to *parallel_plot* selects the clusters whose relative humidities are centered less than 0.5 from the mean value. All clusters in this plot have *relative_humidity* < -0.5, but they differ in values for other features, meaning that there are several weather patterns that include low humidity.

Note in particular cluster 5. This cluster has samples with lower-than-average wind direction values. Recall that wind direction values are in degrees, and 0 means wind coming from the North and increasing clockwise. So samples in this cluster have wind coming from the N and NE directions, with very high wind speeds, and low relative humidity. These are characteristic weather patterns for Santa Ana conditions, which greatly increase the dangers of wildfires.

Let's show clusters for "Warm Days", i.e., weather samples with high air temperature:

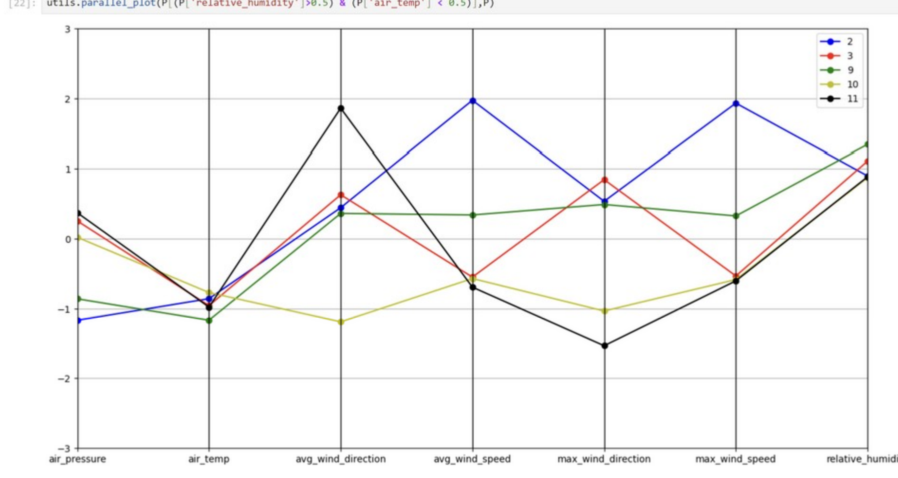
```
[21]: utils.parallel_plot(P[P['air_temp']>0.5],P)
```




All clusters in this plot have *air_temp* > 0.5, but they differ in values for other features.

Let's show clusters for "Cool Days", i.e., weather samples with high relative humidity and low air temperature:

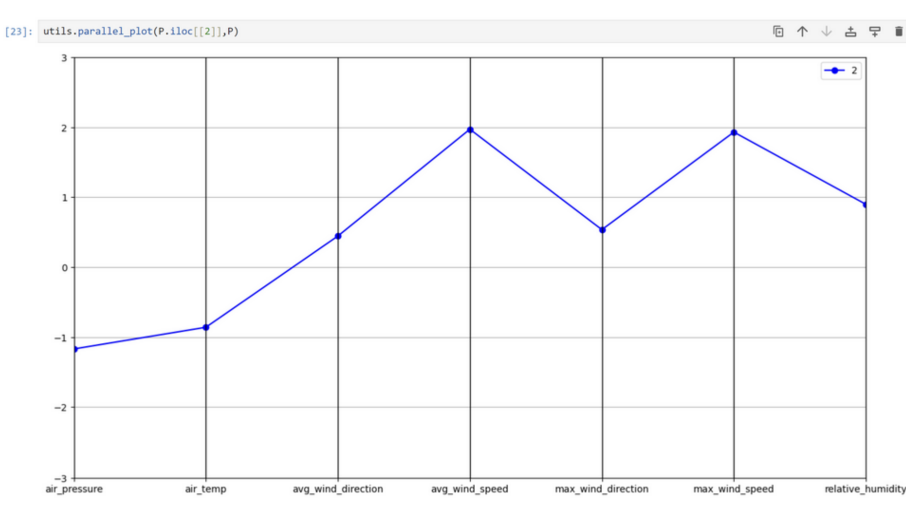
```
[22]: utils.parallel_plot(P[(P['relative_humidity']>0.5) & (P['air_temp'] < 0.5)],P)
```




All clusters in this plot have *relative_humidity* > 0.5 and *air_temp* < 0.5. These clusters represent cool temperature with high humidity and possibly rainy weather patterns. For cluster 5, note that the wind speed values are high, suggesting stormy weather patterns with rain and wind.

So far, we've seen all the clusters except 2 since it did not fall into any of the other categories. Let's plot this cluster:

```
[23]: utils.parallel_plot(P.iloc[[2]],P)
```




Cluster 2 captures days with mild weather.

**Step 9. Exiting the container.** To exit JupyterLab, simply close the tab in your browser. To stop the container, go to Docker Desktop and click on the *stop* button. We recommend not to delete the container, as this container will be used for multiple activities across this specialization.