 databricks/cs110_lab1_power_plant_ml_pipeline



(<http://creativecommons.org/licenses/by-nc-nd/4.0/>)

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. (<http://creativecommons.org/licenses/by-nc-nd/4.0/>)



Power Plant Machine Learning Pipeline Application

This notebook is an end-to-end exercise of performing Extract-Transform-Load and Exploratory Data Analysis on a real-world dataset, and then applying several different machine learning algorithms to solve a supervised regression problem on the dataset.

This notebook covers:

- *Part 1: Business Understanding*
- *Part 2: Load Your Data*
- *Part 3: Explore Your Data*
- *Part 4: Visualize Your Data*
- *Part 5: Data Preparation*
- *Part 6: Data Modeling*
- *Part 7: Tuning and Evaluation*

Our goal is to accurately predict power output given a set of environmental readings from various sensors in a natural gas-fired power generation plant.

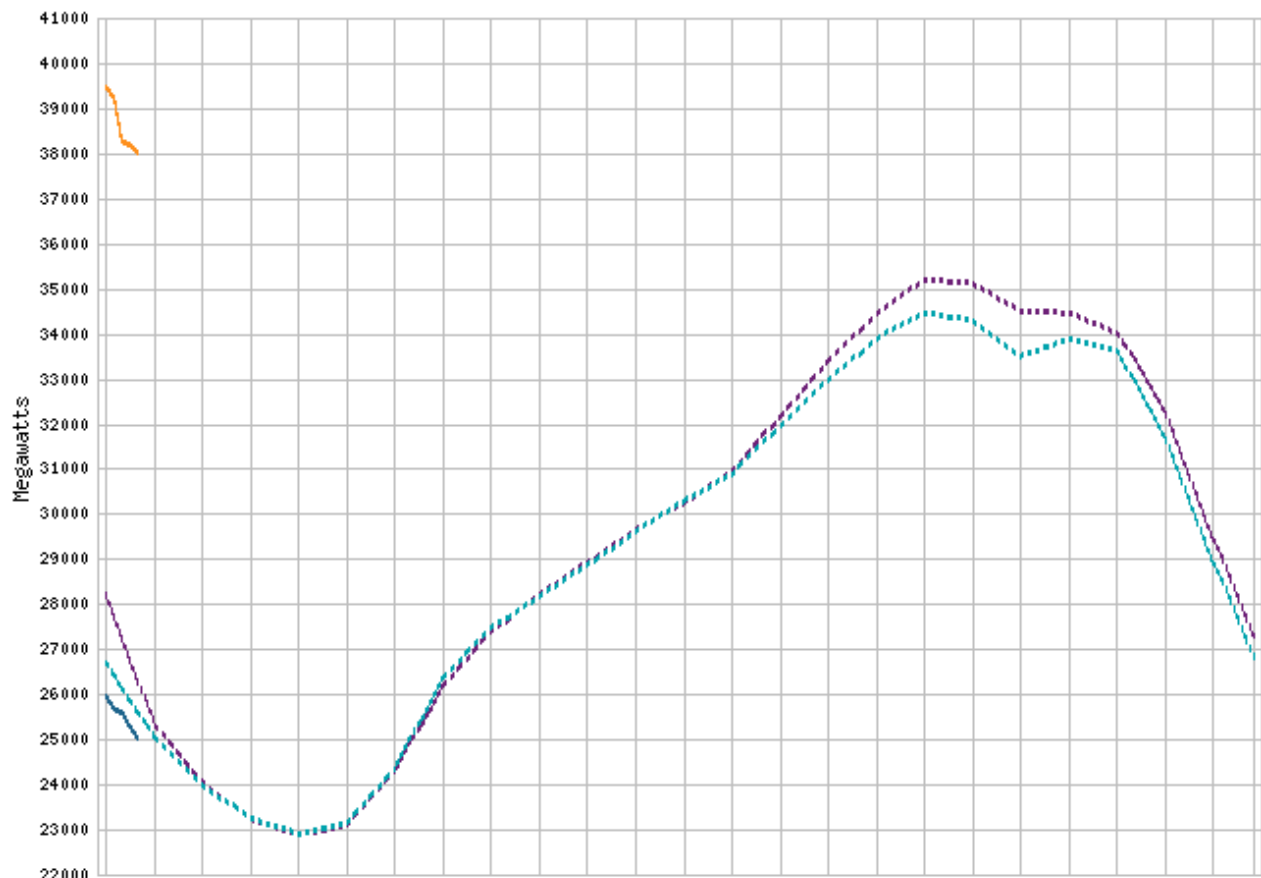
Background

Power generation is a complex process, and understanding and predicting power output is an important element in managing a plant and its connection to the power grid. The operators of a regional power grid create predictions of power demand based on historical information and environmental factors (e.g., temperature). They then compare the predictions against available resources (e.g., coal, natural gas, nuclear, solar, wind, hydro power plants). Power generation technologies such as solar and wind are highly dependent on environmental conditions, and all generation technologies are subject to planned and unplanned maintenance.

Here is an real-world example of predicted demand (on two time scales), actual demand, and available resources from the California power grid:

<http://www.caiso.com/Pages/TodaysOutlook.aspx>

(<http://www.caiso.com/Pages/TodaysOutlook.aspx>)



The challenge for a power grid operator is how to handle a shortfall in available resources versus actual demand. There are three solutions to a power shortfall: build more base load power plants (this process can take many years to decades of planning and construction), buy and import power from other regional power grids (this choice can

be very expensive and is limited by the power transmission interconnects between grids and the excess power available from other grids), or turn on small Peaker or Peaking Power Plants (https://en.wikipedia.org/wiki/Peaking_power_plant). Because grid operators need to respond quickly to a power shortfall to avoid a power outage, grid operators rely on a combination of the last two choices. In this exercise, we'll focus on the last choice.

The Business Problem

Because they supply power only occasionally, the power supplied by a peaker power plant commands a much higher price per kilowatt hour than power from a power grid's base power plants. A peaker plant may operate many hours a day, or it may operate only a few hours per year, depending on the condition of the region's electrical grid. Because of the cost of building an efficient power plant, if a peaker plant is only going to be run for a short or highly variable time it does not make economic sense to make it as efficient as a base load power plant. In addition, the equipment and fuels used in base load plants are often unsuitable for use in peaker plants because the fluctuating conditions would severely strain the equipment.

The power output of a peaker power plant varies depending on environmental conditions, so the business problem is *predicting the power output of a peaker power plant as a function of the environmental conditions* -- since this would enable the grid operator to make economic tradeoffs about the number of peaker plants to turn on (or whether to buy expensive power from another grid).

Given this business problem, we need to first perform Exploratory Data Analysis to understand the data and then translate the business problem (predicting power output as a function of environmental conditions) into a Machine Learning task. In this instance, the ML task is regression since the label (or target) we are trying to predict is numeric. We will use an Apache Spark ML Pipeline (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark-ml-package>) to perform the regression.

The real-world data we are using in this notebook consists of 9,568 data points, each with 4 environmental attributes collected from a Combined Cycle Power Plant over 6 years (2006-2011), and is provided by the University of California, Irvine at UCI Machine Learning Repository Combined Cycle Power Plant Data Set

(<https://archive.ics.uci.edu/ml/datasets/Combined+Cycle+Power+Plant>). You can find more details about the dataset on the UCI page, including the following background publications:

- Pinar T. Tokdemir, Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods (<http://www.journals.elsevier.com/international-journal-of-electrical-power-and-energy-systems/>), International Journal of Electrical Power & Energy Systems, Volume 60, September 2014, Pages 126-140, ISSN 0142-0615.
- Heysem Kaya, Pinar T. Tokdemir and Fikret S. Gökçe: Local and Global Learning Methods for Predicting Power of a Combined Gas & Steam Turbine (<http://www.cmpe.boun.edu.tr/~kaya/kaya2012gasturbine.pdf>), Proceedings of the International Conference on Emerging Trends in Computer and Electronics Engineering ICETCEE 2012, pp. 13-18 (Mar. 2012, Dubai).

To Do: Read the documentation and examples for Spark Machine Learning Pipeline (<https://spark.apache.org/docs/1.6.2/ml-guide.html#main-concepts-in-pipelines>).

```
> labVersion = 'cs110x-power-plant-1.0.0'
```

Part 1: Business Understanding

The first step in any machine learning task is to understand the business need.

As described in the overview we are trying to predict power output given a set of readings from various sensors in a gas-fired power generation plant.

The problem is a regression problem since the label (or target) we are trying to predict is numeric.

Part 2: Extract-Transform-Load (ETL) Your Data

Now that we understand what we are trying to do, the first step is to load our data into a format we can query and use. This is known as ETL or "Extract-Transform-Load". We will load our file from Amazon S3.

Note: Alternatively we could upload our data using "Databricks Menu > Tables > Create Table", assuming we had the raw files on our local computer.

Our data is available on Amazon s3 at the following path:

```
dbfs:/databricks-datasets/power-plant/data
```

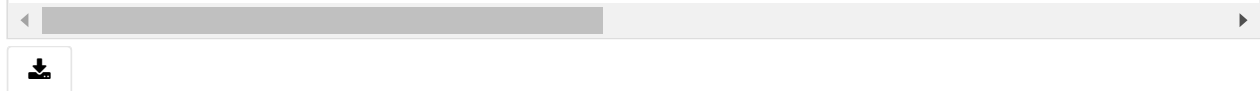
To Do: Let's start by printing a sample of the data.

We'll use the built-in Databricks functions for exploring the Databricks filesystem (DBFS)

Use `display(dbutils.fs.ls("/databricks-datasets/power-plant/data"))` to list the files in the directory

```
> display(dbutils.fs.ls("/databricks-datasets/power-plant/data"))
```

| path |
|---|
| dbfs:/databricks-datasets/power-plant/data/Sheet1.tsv |
| dbfs:/databricks-datasets/power-plant/data/Sheet2.tsv |
| dbfs:/databricks-datasets/power-plant/data/Sheet3.tsv |
| dbfs:/databricks-datasets/power-plant/data/Sheet4.tsv |
| dbfs:/databricks-datasets/power-plant/data/Sheet5.tsv |



Next, use the `dbutils.fs.head` command to look at the first 65,536 bytes of the first file in the directory.

Use `print dbutils.fs.head("/databricks-datasets/power-plant/data/Sheet1.tsv")` to list the files in the directory

```
> print dbutils.fs.head("/databricks-datasets/power-plant/data/Sheet1.tsv")
```

[Truncated to first 65536 bytes]

| AT | V | AP | RH | PE |
|-------|-------|---------|-------|--------|
| 14.96 | 41.76 | 1024.07 | 73.17 | 463.26 |
| 25.18 | 62.96 | 1020.04 | 59.08 | 444.37 |
| 5.11 | 39.4 | 1012.16 | 92.14 | 488.56 |
| 20.86 | 57.32 | 1010.24 | 76.64 | 446.48 |
| 10.82 | 37.5 | 1009.23 | 96.62 | 473.9 |
| 26.27 | 59.44 | 1012.23 | 58.77 | 443.67 |
| 15.89 | 43.96 | 1014.02 | 75.24 | 467.35 |
| 9.48 | 44.71 | 1019.12 | 66.43 | 478.42 |
| 14.64 | 45 | 1021.78 | 41.25 | 475.98 |
| 11.74 | 43.56 | 1015.14 | 70.72 | 477.5 |
| 17.99 | 43.72 | 1008.64 | 75.04 | 453.02 |
| 20.14 | 46.93 | 1014.66 | 64.22 | 453.99 |
| 24.34 | 73.5 | 1011.31 | 84.15 | 440.29 |
| 25.71 | 58.59 | 1012.77 | 61.83 | 451.28 |
| 26.19 | 69.34 | 1009.48 | 87.59 | 433.99 |
| 21.42 | 43.79 | 1015.76 | 43.08 | 462.19 |
| 18.21 | 45 | 1022.86 | 48.84 | 467.54 |
| 11.04 | 41.74 | 1022.6 | 77.51 | 477.2 |
| 14.45 | 52.75 | 1023.97 | 63.59 | 459.85 |

`dbutils.fs` has its own help facility, which we can use to see the various available functions.

```
> dbutils.fs.help()
```

Exercise 2(a)

Now, let's use PySpark instead to print the first 5 lines of the data.

Hint: First create an RDD from the data by using

```
sc.textFile("dbfs:/databricks-datasets/power-plant/data")
```

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.html#pyspark.SparkContext.textFile>) to read the data into an RDD.

Hint: Then figure out how to use the RDD `take()`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.html#pyspark.RDD.take>) method to extract the first 5 lines of the RDD and print each line.

```
> # TODO: Load the data and print the first five lines.
rawTextRdd = sc.textFile("dbfs:/databricks-datasets/power-plant/data")
rawTextRdd.take(5)
```

Out[5]:

```
[u'AT\tV\tAP\tRH\tPE',
 u'14.96\t41.76\t1024.07\t73.17\t463.26',
 u'25.18\t62.96\t1020.04\t59.08\t444.37',
 u'5.11\t39.4\t1012.16\t92.14\t488.56',
 u'20.86\t57.32\t1010.24\t76.64\t446.48']
```

From our initial exploration of a sample of the data, we can make several observations for the ETL process:

- The data is a set of .tsv (Tab Separated Values) files (i.e., each row of the data is separated using tabs)
- There is a header row, which is the name of the columns
- It looks like the type of the data in each column is consistent (i.e., each column is of type double)

Our schema definition from UCI appears below:

- AT = Atmospheric Temperature in C
- V = Exhaust Vacuum Speed
- AP = Atmospheric Pressure
- RH = Relative Humidity
- PE = Power Output. This is the value we are trying to predict given the measurements above.

We are ready to create a DataFrame from the TSV data. Spark does not have a native method for performing this operation, however we can use spark-csv (<https://spark-packages.org/package/databricks/spark-csv>), a third-party package from SparkPackages (<https://spark-packages.org/>). The documentation and source code for spark-csv (<https://spark-packages.org/package/databricks/spark-csv>) can be found on GitHub (<https://github.com/databricks/spark-csv>). The Python API can be found here (<https://github.com/databricks/spark-csv#python-api>).

(Note: In Spark 2.0, the CSV package is built into the DataFrame API.)

To use the spark-csv (<https://spark-packages.org/package/databricks/spark-csv>) package, we use the `sqlContext.read.format()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame> method to specify the input data source format: `'com.databricks.spark.csv'`

We can provide the spark-csv (<https://spark-packages.org/package/databricks/spark-csv>) package with options using the `options()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame> method. The available options are listed in the GitHub documentation here (<https://github.com/databricks/spark-csv#features>).

We will use the following three options:

- `delimiter='\t'` because our data is tab delimited
- `header='true'` because our data has a header row
- `inferSchema='true'` because we believe that all of the data is double values, so the package can dynamically infer the type of each column. *Note that this will require two pass over the data.*

The last component of creating the DataFrame is to specify the location of the data source using the `load()`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame> method: `"/databricks-datasets/power-plant/data"`

Putting everything together, we will use an operation of the following form:

```
sqlContext.read.format().options().load()
```

Exercise 2(b)

To Do: Create a DataFrame from the data.

Hint: Use the above template and fill in each of the methods.

```
> # TODO: Replace <FILL_IN> with the appropriate code.
powerPlantDF =
  sqlContext.read.format('com.databricks.spark.csv').options(delimiter='\t',
    header='true', inferSchema='true').load("/databricks-datasets/power-
    plant/data")
```



```
> # TEST
    from databricks_test_helper import *
    expected = set([(s, 'double') for s in ('AP', 'AT', 'PE', 'RH', 'V')])
    Test.assertEquals(expected, set(powerPlantDF.dtypes), "Incorrect schema for
    powerPlantDF")
```

1 test passed.

Check the names and types of the columns using the dtypes

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame> method.

```
> print powerPlantDF.dtypes
```

```
[('AT', 'double'), ('V', 'double'), ('AP', 'double'), ('RH', 'double'), ('PE',
'double')]
```

We can examine the data using the display() method.

```
> display(powerPlantDF)
```

| AT | V | AP |
|-------|-------|---------|
| 14.96 | 41.76 | 1024.07 |
| 25.18 | 62.96 | 1020.04 |
| 5.11 | 39.4 | 1012.16 |
| 20.86 | 57.32 | 1010.24 |
| 10.82 | 37.5 | 1009.23 |
| 26.27 | 59.44 | 1012.23 |
| 15.89 | 43.96 | 1014.02 |
| 9.48 | 44.71 | 1019.12 |
| 14.64 | 45 | 1021.78 |

Showing the first 1000 rows.



Part 2: Alternative Method to Load your Data

Instead of having spark-csv (<https://spark-packages.org/package/databricks/spark-csv>) infer the types of the columns, we can specify the schema as a `DataType` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.types.DataType> which is a list of `StructField` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.types.StructField>

You can find a list of types in the `pyspark.sql.types` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#module-pyspark.sql.types>) module. For our data, we will use `DoubleType()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.types.DoubleType>

For example, to specify that a column's name and type, we use: `StructField(name, type, True)`. (The third parameter, `True`, signifies that the column is nullable.)

Exercise 2(c)

Create a custom schema for the power plant data.

```
> # TO DO: Fill in the custom schema.
    from pyspark.sql.types import *

    # Custom Schema for Power Plant
    customSchema = StructType([ \
        StructField("AT", DoubleType(), True), \
        StructField("V", DoubleType(), True), \
        StructField("AP", DoubleType(), True), \
        StructField("RH", DoubleType(), True), \
        StructField("PE", DoubleType(), True) \
    ])

> # TEST
    Test.assertEquals(set([f.name for f in customSchema.fields]), set(['AT', 'V', 'AP', 'RH', 'PE']), 'Incorrect column names in schema.')
    Test.assertEquals(set([f.dataType for f in customSchema.fields]), set([DoubleType(), DoubleType(), DoubleType(), DoubleType(), DoubleType()]), 'Incorrect column types in schema.')

1 test passed.
1 test passed.
```

Exercise 2(d)

Now, let's use the schema to read the data. To do this, we will modify the earlier `sqlContext.read.format` step. We can specify the schema by:

- Adding `schema = customSchema` to the load method (use a comma and add it after the file name)
- Removing the `inferredSchema='true'` option because we are explicitly specifying the schema

```
> # TODO: Use the schema you created above to load the data again.
altPowerPlantDF =
sqlContext.read.format('com.databricks.spark.csv').options(delimiter='\t',
header='true').load("/databricks-datasets/power-plant/data", schema =
customSchema)

> # TEST
from databricks_test_helper import *
expected = set([(s, 'double') for s in ('AP', 'AT', 'PE', 'RH', 'V')])
Test.assertEquals(expected, set(altPowerPlantDF.dtypes), "Incorrect schema for
powerPlantDF")
```

1 test passed.

Note that no Spark jobs are launched this time. That is because we specified the schema, so the spark-csv (<https://spark-packages.org/package/databricks/spark-csv>) package does not have to read the data to infer the schema. We can use the `dtypes` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame>) method to examine the names and types of the columns. They should be identical to the names and types of the columns that were earlier inferred from the data.

When you run the following cell, data would not be read.

```
> print altPowerPlantDF.dtypes

[('AT', 'double'), ('V', 'double'), ('AP', 'double'), ('RH', 'double'), ('PE',
'double')]
```

Now we can examine the data using the `display()` method. *Note that this operation will cause the data to be read and the DataFrame will be created.*

```
> display(altPowerPlantDF)
```

| AT | V | AP |
|-------|-------|---------|
| 14.96 | 41.76 | 1024.07 |
| 25.18 | 62.96 | 1020.04 |
| 5.11 | 39.4 | 1012.16 |
| 20.86 | 57.32 | 1010.24 |
| 10.82 | 37.5 | 1009.23 |
| 26.27 | 59.44 | 1012.23 |
| 15.89 | 43.96 | 1014.02 |
| 9.48 | 44.71 | 1019.12 |
| 11.61 | 45.1 | 1021.78 |

Showing the first 1000 rows.



Part 3: Explore Your Data

Now that your data is loaded, the next step is to explore it and perform some basic analysis and visualizations.

This is a step that you should always perform **before** trying to fit a model to the data, as this step will often lead to important insights about your data.

First, let's register our DataFrame as an SQL table named `power_plant`. Because you may run this lab multiple times, we'll take the precaution of removing any existing tables first.

We can delete any existing `power_plant` SQL table using the SQL command:

`DROP TABLE IF EXISTS power_plant` (we also need to delete any Hive data associated with the table, which we can do with a Databricks file system operation).

Once any prior table is removed, we can register our DataFrame as a SQL table using `sqlContext.registerDataFrameAsTable()`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.SQLContext.registerDataFrameAsTable>)

3(a)

ToDo: Execute the prepared code in the following cell.

```
> sqlContext.sql("DROP TABLE IF EXISTS power_plant")
dbutils.fs.rm("dbfs:/user/hive/warehouse/power_plant", True)
sqlContext.registerDataFrameAsTable(powerPlantDF, "power_plant")
```

Now that our DataFrame exists as a SQL table, we can explore it using SQL commands.

To execute SQL in a cell, we use the `%sql` operator. The following cell is an example of using SQL to query the rows of the SQL table.

NOTE: `%sql` is a Databricks-only command. It calls `sqlContext.sql()` and passes the results to the Databricks-only `display()` function. These two statements are equivalent:

```
%sql SELECT * FROM power_plant

display(sqlContext.sql("SELECT * FROM power_plant"))
```

3(b)

ToDo: Execute the prepared code in the following cell.

```
> %sql
-- We can use %sql to query the rows
SELECT * FROM power_plant
```

| AT | V | AP |
|-------|-------|---------|
| 14.96 | 41.76 | 1024.07 |
| 25.18 | 62.96 | 1020.04 |
| 5.11 | 39.4 | 1012.16 |
| 20.86 | 57.32 | 1010.24 |
| 10.82 | 37.5 | 1009.23 |
| 26.27 | 59.44 | 1012.23 |
| 15.89 | 43.96 | 1014.02 |
| 9.48 | 44.71 | 1019.12 |
| 11.61 | 45 | 1021.78 |

Showing the first 1000 rows.




3(c)

Use the SQL `desc` command to describe the schema, by executing the following cell.

```
> %sql
  desc power_plant
```

| col_name | data_type |
|----------|-----------|
| AT | double |
| V | double |
| AP | double |
| RH | double |
| PE | double |



Schema Definition

Once again, here's our schema definition:

- AT = Atmospheric Temperature in C
- V = Exhaust Vacuum Speed
- AP = Atmospheric Pressure
- RH = Relative Humidity
- PE = Power Output

PE is our label or target. This is the value we are trying to predict given the measurements.

Reference UCI Machine Learning Repository Combined Cycle Power Plant Data Set
(<https://archive.ics.uci.edu/ml/datasets/Combined+Cycle+Power+Plant>)

Let's perform some basic statistical analyses of all the columns.

We can get the DataFrame associated with a SQL table by using the `sqlContext.table()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame>) method and passing in the name of the SQL table. Then, we can use the DataFrame `describe()`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame> method with no arguments to compute some basic statistics for each column like count, mean, max, min and standard deviation.

```
> df = sqlContext.table("power_plant")
    display(df.describe())
```

| summary | AT | V | AP |
|---------|--------------------|--------------------|--------------------|
| count | 47840 | 47840 | 47840 |
| mean | 19.651231187290996 | 54.30580372073594 | 101.30580372073594 |
| stddev | 7.452161658340004 | 12.707361709685806 | 5.907361709685806 |
| min | 1.81 | 25.36 | 99.2 |
| max | 37.11 | 81.56 | 103.56 |



Part 4: Visualize Your Data

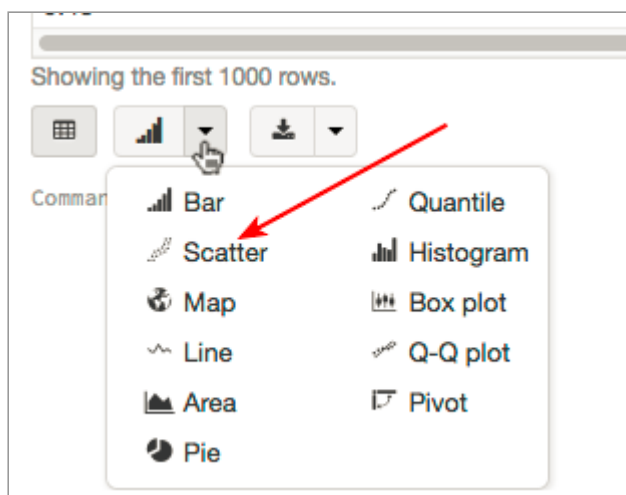
To understand our data, we will look for correlations between features and the label. This can be important when choosing a model. E.g., if features and a label are linearly correlated, a linear model like Linear Regression can do well; if the relationship is very non-linear, more complex models such as Decision Trees can be better. We can use Databrick's built in visualization to view each of our predictors in relation to the label column as a scatter plot to see the correlation between the predictors and the label.

Exercise 4(a)

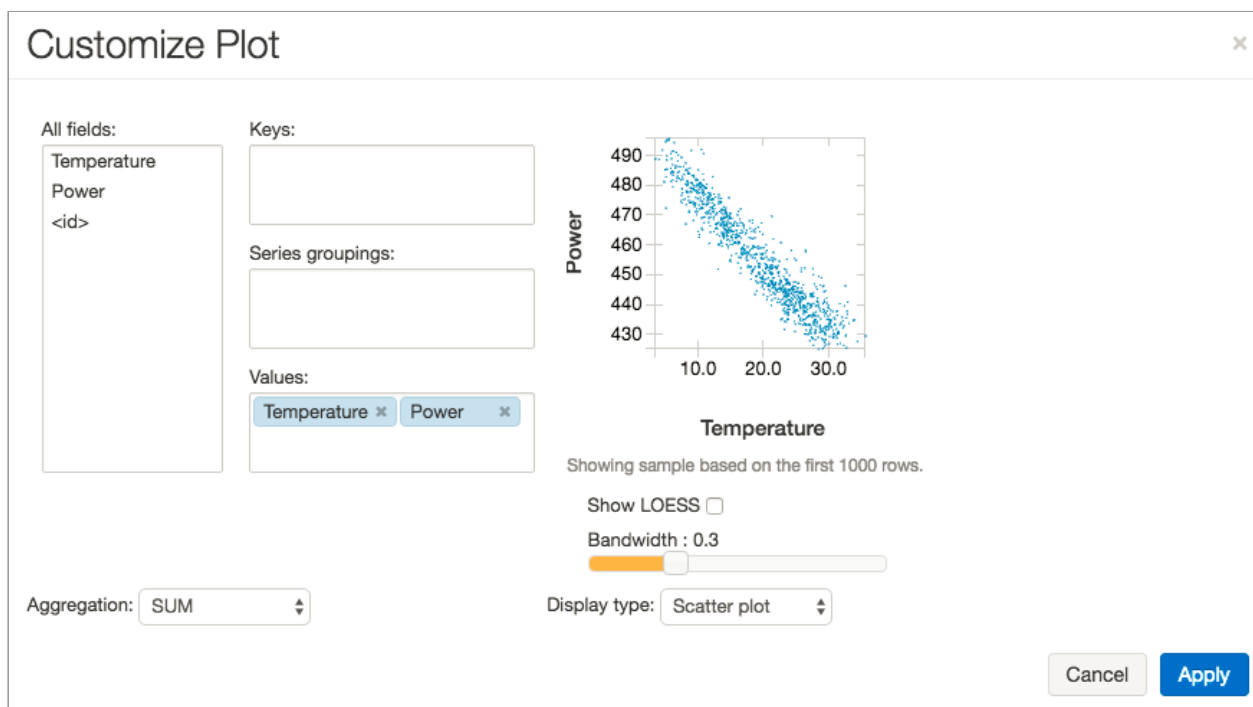
Add figures to the following: Let's see if there is a correlation between Temperature and Power Output. We can use a SQL query to create a new table consisting of only the Temperature (AT) and Power (PE) columns, and then use a scatter plot with Temperature on the X axis and Power on the Y axis to visualize the relationship (if any) between Temperature and Power.

Perform the following steps:

- Run the following cell
- Click on the drop down next to the "Bar chart" icon and select "Scatter" to turn the table into a Scatter plot



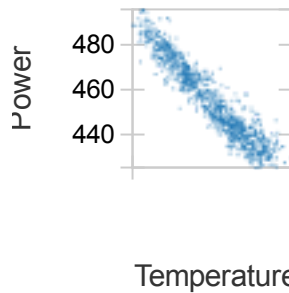
- Click on "Plot Options..."
- In the Values box, click on "Temperature" and drag it before "Power", as shown below:



- Apply your changes by clicking the Apply button
- Increase the size of the graph by clicking and dragging the size control

> %sql

```
select AT as Temperature, PE as Power from power_plant
```

Showing sample based on the first 1000 rows.



It looks like there is strong linear correlation between Temperature and Power Output.

ASIDE: A quick physics lesson: This correlation is to be expected as the second law of thermodynamics puts a fundamental limit on the thermal efficiency (https://en.wikipedia.org/wiki/Thermal_efficiency) of all heat-based engines. The limiting factors are:

- The temperature at which the heat enters the engine T_H
- The temperature of the environment into which the engine exhausts its waste heat T_C

Our temperature measurements are the temperature of the environment. From Carnot's theorem (https://en.wikipedia.org/wiki/Carnot%27s_theorem_%28thermodynamics%29), no heat engine working between these two temperatures can exceed the Carnot Cycle efficiency:

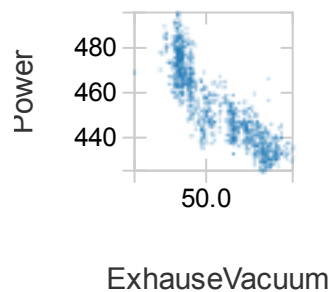
$$n_{th} \leq 1 - \frac{T_C}{T_H}$$

Note that as the environmental temperature increases, the efficiency decreases -- *this is the effect that we see in the above graph.*

Exercise 4(b)

Use SQL to create a scatter plot of Power(PE) as a function of ExhaustVacuum (V). Name the y-axis "Power" and the x-axis "ExhaustVacuum"

```
> %sql
select V as ExhausteVacuum, PE as Power from power_plant
```



Showing sample based on the first 1000 rows.

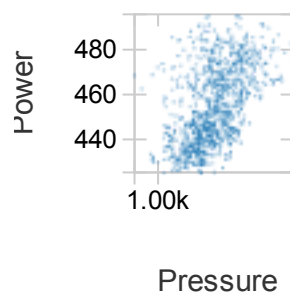


Let's continue exploring the relationships (if any) between the variables and Power Output.

Exercise 4(c)

Use SQL to create a scatter plot of Power(PE) as a function of Pressure (AP). Name the y-axis "Power" and the x-axis "Pressure"

```
> %sql
select AP as Pressure, PE as Power from power_plant
```



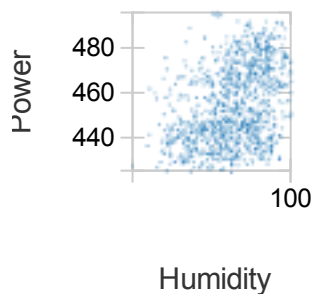
Showing sample based on the first 1000 rows.



Exercise 4(d)

Use SQL to create a scatter plot of Power(PE) as a function of Humidity (RH). Name the y-axis "Power" and the x-axis "Humidity"

```
> %sql
select RH as Humidity, PE as Power from power_plant
```



Showing sample based on the first 1000 rows.



Part 5: Data Preparation

The next step is to prepare the data for machine learning. Since all of this data is numeric and consistent this is a simple and straightforward task.

The goal is to use machine learning to determine a function that yields the output power as a function of a set of predictor features. The first step in building our ML pipeline is to convert the predictor features from DataFrame columns to Feature Vectors using the `pyspark.ml.feature.VectorAssembler()`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.feature.VectorAssembler> method).

The `VectorAssembler` is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees. `VectorAssembler` takes a list of input column names (each is a string) and the name of the output column (as a string).

Exercise 5(a)

- Read the Spark documentation and usage examples for `VectorAssembler` (<https://spark.apache.org/docs/1.6.2/ml-features.html#vectorassembler>)
- Convert the `power_plant` SQL table into a DataFrame named `dataset`
- Set the vectorizer's input columns to a list of the four columns of the input DataFrame: `["AT", "V", "AP", "RH"]`

- Set the vectorizer's output column name to "features"

```
> # TODO: Replace <FILL_IN> with the appropriate code
from pyspark.ml.feature import VectorAssembler

datasetDF = sqlContext.table("power_plant")

vectorizer = VectorAssembler()
vectorizer.setInputCols(["AT", "V", "AP", "RH"])
vectorizer.setOutputCol("features")

Out[18]: VectorAssembler_47aba5c3784a09789c02

> # TEST
Test.assertEquals(set(vectorizer.getInputCols()), {"AT", "V", "AP", "RH"},
"Incorrect vectorizer input columns")
Test.assertEquals(vectorizer.getOutputCol(), "features", "Incorrect vectorizer
output column")

1 test passed.
1 test passed.
```

Part 6: Data Modeling

Now let's model our data to predict what the power output will be given a set of sensor readings

Our first model will be based on simple linear regression since we saw some linear patterns in our data based on the scatter plots during the exploration stage.

We need a way of evaluating how well our linear regression model predicts power output as a function of input parameters. We can do this by splitting up our initial data set into a *Training Set* used to train our model and a *Test Set* used to evaluate the model's performance in giving predictions. We can use a DataFrame's `randomSplit()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame>) method to split our dataset. The method takes a list of weights and an optional random seed. The seed is used to initialize the random number generator used by the splitting function.

Exercise 6(a)

Use the randomSplit()

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame> method to divide up datasetDF into a trainingSetDF (80% of the input DataFrame) and a testSetDF (20% of the input DataFrame), and for reproducibility, use the seed 1800009193L. Then cache each DataFrame in memory to maximize performance.

```
> # TODO: Replace <FILL_IN> with the appropriate code.
# We'll hold out 20% of our data for testing and leave 80% for training
seed = 1800009193L
(split20DF, split80DF) = datasetDF.randomSplit([0.2,0.8], 1800009193L)

# Let's cache these datasets for performance
testSetDF = split20DF.cache()
trainingSetDF = split80DF.cache()

> # TEST
Test.assertEquals(trainingSetDF.count(), 38243, "Incorrect size for training
data set")
Test.assertEquals(testSetDF.count(), 9597, "Incorrect size for test data set")

1 test passed.
1 test passed.
```

Next we'll create a Linear Regression Model and use the built in help to identify how to train it. See API details for Linear Regression

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.L> in the ML guide.

Exercise 6(b)

- Read the documentation and examples for Linear Regression
(<https://spark.apache.org/docs/1.6.2/ml-classification-regression.html#linear-regression>)
- Run the next cell

```
> # ***** LINEAR REGRESSION MODEL *****
```

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.regression import LinearRegressionModel
from pyspark.ml import Pipeline
```

```
# Let's initialize our linear regression learner
lr = LinearRegression()
```

```
# We use explain params to dump the parameters we can use
print(lr.explainParams())
```

elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty. (default: 0.0)

featuresCol: features column name. (default: features)

fitIntercept: whether to fit an intercept term. (default: True)

labelCol: label column name. (default: label)

maxIter: max number of iterations (≥ 0). (default: 100)

predictionCol: prediction column name. (default: prediction)

regParam: regularization parameter (≥ 0). (default: 0.0)

solver: the solver algorithm for optimization. If this is not set or empty, default value is 'auto'. (default: auto)

standardization: whether to standardize the training features before fitting the model. (default: True)

tol: the convergence tolerance for iterative algorithms. (default: $1e-06$)

weightCol: weight column name. If this is not set or empty, we treat all instance weights as 1.0. (undefined)

The cell below is based on the Spark ML Pipeline API for Linear Regression

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.L>

The first step is to set the parameters for the method:

- Set the name of the prediction column to "Predicted_PE"
- Set the name of the label column to "PE"
- Set the maximum number of iterations to 100
- Set the regularization parameter to 0.1

Next, we create the ML Pipeline

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.Pipeline>)

and set the stages to the Vectorizer and Linear Regression learner we created earlier.

Finally, we create a model by training on trainingSetDF.

Exercise 6(c)

- Read the Linear Regression (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression> documentation)
- Run the next cell, and be sure you understand what's going on.

```
> # Now we set the parameters for the method
lr.setPredictionCol("Predicted_PE")\
  .setLabelCol("PE")\
  .setMaxIter(100)\
  .setRegParam(0.1)

# We will use the new spark.ml pipeline API. If you have worked with scikit-
learn this will be very familiar.
lrPipeline = Pipeline()

lrPipeline.setStages([vectorizer, lr])

# Let's first train on the entire dataset to see what we get
lrModel = lrPipeline.fit(trainingSetDF)
```

From the Wikipedia article on Linear Regression
(https://en.wikipedia.org/wiki/Linear_regression):

In statistics, linear regression is an approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X . In linear regression, the relationships are modeled using linear predictor functions whose unknown model parameters are estimated from the data. Such models are called linear models.

Linear regression has many practical uses. Most applications fall into one of the following two broad categories:

- If the goal is prediction, or forecasting, or error reduction, linear regression can be used to fit a predictive model to an observed data set of y and X values. After

developing such a model, if an additional value of X is then given without its accompanying value of y , the fitted model can be used to make a prediction of the value of y .

- Given a variable y and a number of variables X_1, \dots, X_p that may be related to y , linear regression analysis can be applied to quantify the strength of the relationship between y and the X_j , to assess which X_j may have no relationship with y at all, and to identify which subsets of the X_j contain redundant information about y .

We are interested in both uses, as we would like to predict power output as a function of the input variables, and we would like to know which input variables are weakly or strongly correlated with power output.

Since Linear Regression is simply a Line of best fit over the data that minimizes the square of the error, given multiple input dimensions we can express each predictor as a line function of the form:

$$y = a + bx_1 + bx_2 + bx_i \dots$$

where a is the intercept and the b are the coefficients.

To express the coefficients of that line we can retrieve the Estimator stage from the PipelineModel and express the weights and the intercept for the function.

Exercise 6(d)

Run the next cell. Ensure that you understand what's going on.


```

> # The intercept is as follows:
intercept = lrModel.stages[1].intercept

# The coefficients (i.e., weights) are as follows:
weights = lrModel.stages[1].coefficients

# Create a list of the column names (without PE)
featuresNoLabel = [col for col in datasetDF.columns if col != "PE"]

# Merge the weights and labels
coefficients = zip(weights, featuresNoLabel)

# Now let's sort the coefficients from greatest absolute weight most to the
least absolute weight
coefficients.sort(key=lambda tup: abs(tup[0]), reverse=True)

equation = "y = {intercept}".format(intercept=intercept)
variables = []
for x in coefficients:
    weight = abs(x[0])
    name = x[1]
    symbol = "+" if (x[0] > 0) else "-"
    equation += (" {} ({} * {})".format(symbol, weight, name))

# Finally here is our equation
print("Linear Regression Equation: " + equation)

Linear Regression Equation: y = 436.42267224 - (1.91608232805 * AT) - (0.2552301
67561 * V) - (0.147726340554 * RH) + (0.0792423513007 * AP)

```

Recall **Part 4: Visualize Your Data** when we visualized each predictor against Power Output using a Scatter Plot, does the final equation seems logical given those visualizations?

ToDo: Answer the quiz questions about correlations (on edX).

Exercise 6(e)

Now let's see what our predictions look like given this model. We apply our Linear Regression model to the 20% of the data that we split from the input dataset. The output of the model will be a predicted Power Output column named "Predicted_PE".

- Run the next cell

- Scroll through the resulting table and notice how the values in the Power Output (PE) column compare to the corresponding values in the predicted Power Output (Predicted_PE) column

```
> # Apply our LR model to the test data and predict power output
predictionsAndLabelsDF = lrModel.transform(testSetDF).select("AT", "V", "AP",
"RH", "PE", "Predicted_PE")

display(predictionsAndLabelsDF)
```

| AT | V | AP | RH |
|------|-------|---------|-------|
| 1.81 | 39.42 | 1026.92 | 76.97 |
| 3.2 | 41.31 | 997.67 | 98.84 |
| 3.38 | 41.31 | 998.79 | 97.76 |
| 3.4 | 39.64 | 1011.1 | 83.43 |
| 3.51 | 35.47 | 1017.53 | 86.56 |
| 3.63 | 38.44 | 1016.16 | 87.38 |
| 3.91 | 35.47 | 1016.92 | 86.03 |
| 3.94 | 39.9 | 1008.06 | 97.49 |
| 4 | 39.9 | 1008.06 | 97.49 |

Showing the first 1000 rows.



From a visual inspection of the predictions, we can see that they are close to the actual values.

However, we would like a scientific measure of how well the Linear Regression model is performing in accurately predicting values. To perform this measurement, we can use an evaluation metric such as Root Mean Squared Error (https://en.wikipedia.org/wiki/Root-mean-square_deviation) (RMSE) to validate our Linear Regression model.

RSME is defined as follows: $RMSE = \sqrt{\frac{\sum_{i=1}^n (x_i - y_i)^2}{n}}$ where y_i is the observed value and x_i is the predicted value

RMSE is a frequently used measure of the differences between values predicted by a model or an estimator and the values actually observed. The lower the RMSE, the better our model.

Spark ML Pipeline provides several regression analysis metrics, including

`RegressionEvaluator()`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.evaluation.R>

After we create an instance of `RegressionEvaluator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.evaluation.R>

we set the label column name to "PE" and set the prediction column name to "Predicted_PE". We then invoke the evaluator on the predictions.

Exercise 6(f)

Run the next cell and ensure that you understand what's going on.

```
> # Now let's compute an evaluation metric for our test dataset
  from pyspark.ml.evaluation import RegressionEvaluator

  # Create an RMSE evaluator using the label and predicted columns
  regEval = RegressionEvaluator(predictionCol="Predicted_PE", labelCol="PE",
                                metricName="rmse")

  # Run the evaluator on the DataFrame
  rmse = regEval.evaluate(predictionsAndLabelsDF)

  print("Root Mean Squared Error: %.2f" % rmse)
```

Root Mean Squared Error: 4.59

Another useful statistical evaluation metric is the coefficient of determination, denoted R^2 or r^2 and pronounced "R squared". It is a number that indicates the proportion of the variance in the dependent variable that is predictable from the independent variable and it provides a measure of how well observed outcomes are replicated by the model, based on the proportion of total variation of outcomes explained by the model. The coefficient of determination ranges from 0 to 1 (closer to 1), and the higher the value, the better our model.

To compute r^2 , we invoke the evaluator with `regEval.metricName: "r2"`

Exercise 6(g)

Run the next cell and ensure that you understand what's going on.

```
> # Now let's compute another evaluation metric for our test dataset
  r2 = regEval.evaluate(predictionsAndLabelsDF, {regEval.metricName: "r2"})

  print("r2: {0:.2f}".format(r2))

r2: 0.93
```

Generally, assuming a Gaussian distribution of errors, a good model will have 68% of predictions within 1 RMSE and 95% within 2 RMSE of the actual value (see <http://statweb.stanford.edu/~susan/courses/s60/split/node60.html> (<http://statweb.stanford.edu/~susan/courses/s60/split/node60.html>)).

Let's examine the predictions and see if a RMSE of 4.59 meets this criteria.

We create a new DataFrame using selectExpr()

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame> to project a set of SQL expressions, and register the DataFrame as a SQL table using registerTempTable()

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#pyspark.sql.DataFrame>

Exercise 6(h)

Run the next cell and ensure that you understand what's going on.

```
> # First we remove the table if it already exists
  sqlContext.sql("DROP TABLE IF EXISTS Power_Plant_RMSE_Evaluation")
  dbutils.fs.rm("dbfs:/user/hive/warehouse/Power_Plant_RMSE_Evaluation", True)

  # Next we calculate the residual error and divide it by the RMSE
  predictionsAndLabelsDF.selectExpr("PE", "Predicted_PE", "PE - Predicted_PE
  Residual_Error", "(PE - Predicted_PE) / {}
  Within_RSME".format(rmse)).registerTempTable("Power_Plant_RMSE_Evaluation")
```

We can use SQL to explore the Power_Plant_RMSE_Evaluation table. First let's look at at the table using a SQL SELECT statement.

Exercise 6(i)

Run the next cell and ensure that you understand what's going on.

```
> %sql
SELECT * from Power_Plant_RMSE_Evaluation
```

| PE | Predicted_PE | Residual_Error |
|--------|--------------------|---------------------|
| 490.55 | 492.8984489863317 | -2.3484489863316753 |
| 489.86 | 484.2040956902032 | 5.655904309796824 |
| 489.11 | 484.10749675240885 | 5.002503247591164 |
| 459.86 | 487.58780129031743 | -27.727801290317416 |
| 489.07 | 488.48848690589034 | 0.5815130941096527 |
| 487.87 | 487.2708258083337 | 0.5991741916662932 |
| 488.67 | 487.75201110087056 | 0.9179888991294547 |
| 488.81 | 484.1688278934683 | 4.641172106531712 |
| 490.70 | 484.22781556122305 | 6.562184428776067 |

Showing the first 1000 rows.

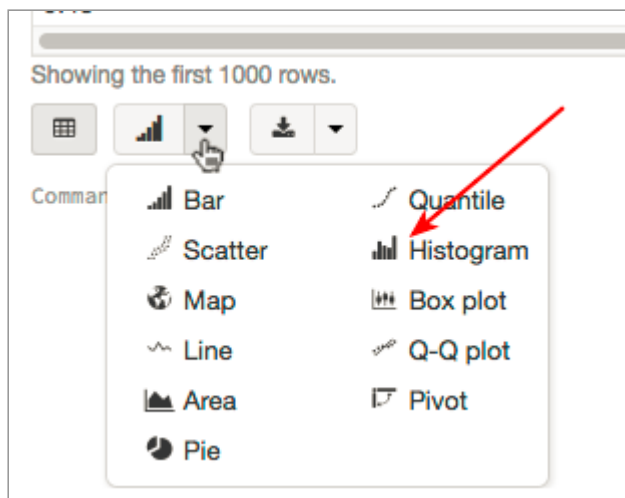


Now we can display the RMSE as a Histogram.

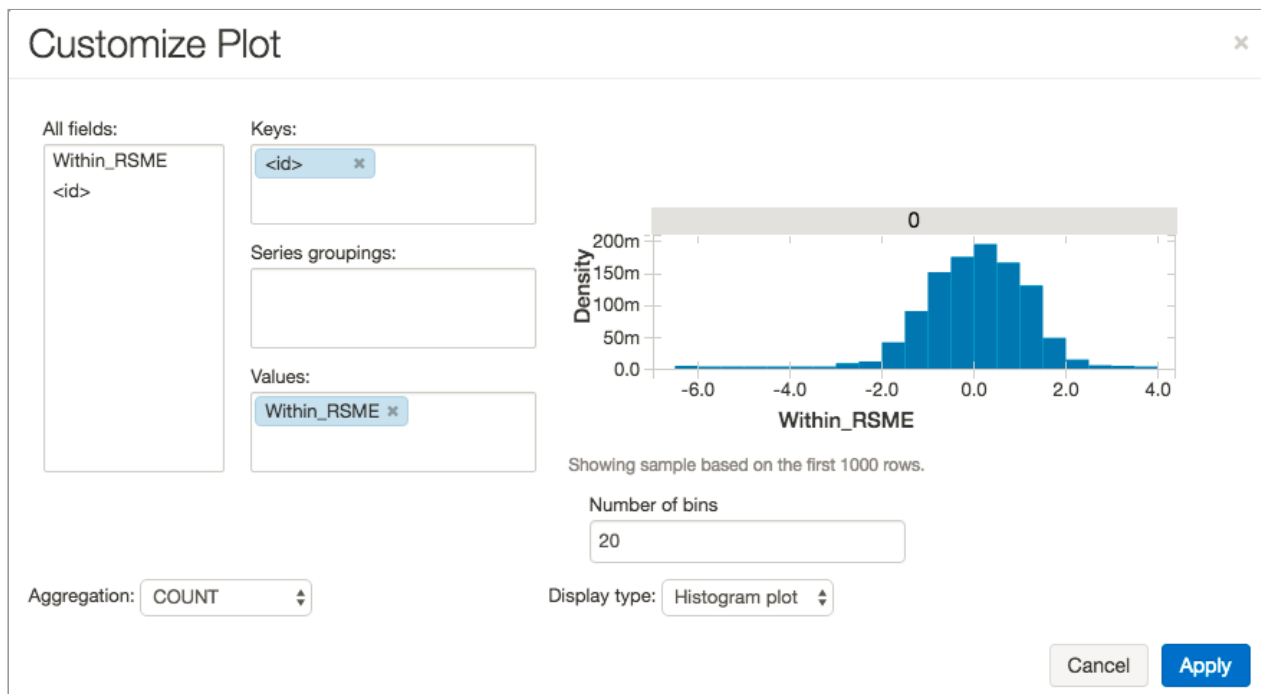
Exercise 6(j)

Perform the following steps:

- Run the following cell
- Click on the drop down next to the "Bar chart" icon a select "Histogram" to turn the table into a Histogram plot



- Click on "Plot Options..."
- In the "All Fields:" box, click on "<id>" and drag it into the "Keys:" box
- Change the "Aggregation" to "COUNT"



- Apply your changes by clicking the Apply button
- Increase the size of the graph by clicking and dragging the size control

Notice that the histogram clearly shows that the RMSE is centered around 0 with the vast majority of the error within 2 RMSEs.

> %sql

-- Now we can display the RMSE as a Histogram

SELECT Within_RSME **from** Power_Plant_RMSE_Evaluation

| Within_RSME |
|---------------------|
| -0.5111964452663913 |
| 1.2311437015505429 |
| 1.0889152340484416 |
| -6.035623314434327 |
| 0.12658032101819366 |
| 0.1304246839756194 |
| 0.19982237841239994 |
| 1.010262814537229 |
| 1.4284172842286727 |

Showing the first 1000 rows.



Using a complex SQL SELECT statement, we can count the number of predictions within + or - 1.0 and + or - 2.0 and then display the results as a pie chart.

Exercise 6(k)

Perform the following steps:

- Run the following cell
- Click on the drop down next to the "Bar chart" icon a select "Pie" to turn the table into a Pie Chart plot
- Increase the size of the graph by clicking and dragging the size control

```
> %sql
SELECT case when Within_RSME <= 1.0 AND Within_RSME >= -1.0 then 1
         when Within_RSME <= 2.0 AND Within_RSME >= -2.0 then 2 else 3
         end RSME_Multiple, COUNT(*) AS count
FROM Power_Plant_RMSE_Evaluation
GROUP BY case when Within_RSME <= 1.0 AND Within_RSME >= -1.0 then 1 when
Within_RSME <= 2.0 AND Within_RSME >= -2.0 then 2 else 3 end
```

| RSME_Multiple |
|---------------|
| 1 |
| 2 |
| 3 |



From the pie chart, we can see that 68% of our test data predictions are within 1 RMSE of the actual values, and 97% (68% + 29%) of our test data predictions are within 2 RMSE. So the model is pretty decent. Let's see if we can tune the model to improve it further.

Part 7: Tuning and Evaluation

Now that we have a model with all of the data let's try to make a better model by tuning over several parameters. The process of tuning a model is known as Model Selection (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#module-pyspark.ml.tuning>) or Hyperparameter Tuning (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#module-pyspark.ml.tuning>), and Spark ML Pipeline makes the tuning process very simple and easy.

An important task in ML is model selection, or using data to find the best model or parameters for a given task. This is also called tuning. Tuning may be done for individual Estimators such as LinearRegression (<https://spark.apache.org/docs/1.6.2/ml-classification-regression.html#linear-regression>), or for entire Pipelines which include multiple algorithms, featurization, and other steps. Users can tune an entire Pipeline at once, rather than tuning each element in the Pipeline separately.

Spark ML Pipeline supports model selection using tools such as CrossValidator (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Cross>) which requires the following items:

- Estimator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.Estimator>)
algorithm or Pipeline to tune
- Set of ParamMaps
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.ParamMaps>)
parameters to choose from, sometimes called a *parameter grid* to search over
- Evaluator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.evaluation>)
metric to measure how well a fitted Model does on held-out test data

At a high level, model selection tools such as `CrossValidator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) work as follows:

- They split the input data into separate training and test datasets.
- For each (training, test) pair, they iterate through the set of `ParamMaps`:
 - For each `ParamMap`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) they fit the `Estimator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.Estimator>) using those parameters, get the fitted `Model`, and evaluate the `Model`'s performance using the `Evaluator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.evaluation>)
- They select the `Model` produced by the best-performing set of parameters.

The `Evaluator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.evaluation>) can be a `RegressionEvaluator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.evaluation.RegressionEvaluator>) for regression problems. To help construct the parameter grid, users can use the `ParamGridBuilder`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.ParamGridBuilder>) utility.

Note that cross-validation over a grid of parameters is expensive. For example, in the next cell, the parameter grid has 10 values for `lr.regParam`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.LinearRegressionModel>) and `CrossValidator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) uses 3 folds. This multiplies out to $(10 \times 3) = 30$ different models being trained. In realistic settings, it can be common to try many more parameters (e.g., multiple values for multiple parameters) and use more folds ($k = 3$ and $k = 10$ are common). In other words, using `CrossValidator`

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) can be very expensive. However, it is also a well-established method for choosing parameters which is more statistically sound than heuristic hand-tuning.

We perform the following steps:

- Create a CrossValidator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Ci>)
using the Pipeline and RegressionEvaluator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.evaluatic>)
that we created earlier, and set the number of folds to 3
- Create a list of 10 regularization parameters
- Use ParamGridBuilder
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Pa>)
to build a parameter grid with the regularization parameters and add the grid to the
CrossValidator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Ci>)
- Run the CrossValidator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Ci>)
to find the parameters that yield the best model (i.e., lowest RMSE) and return the
best model.

Exercise 7(a)

Run the next cell. *Note that it will take some time to run the CrossValidator*

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Cross>)
as it will run almost 200 Spark jobs

```
> from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# We can reuse the RegressionEvaluator, regEval, to judge the model based on
the best Root Mean Squared Error
# Let's create our CrossValidator with 3 fold cross validation
crossval = CrossValidator(estimator=lrPipeline, evaluator=regEval, numFolds=3)

# Let's tune over our regularization parameter from 0.01 to 0.10
regParam = [x / 100.0 for x in range(1, 11)]

# We'll create a parameter grid using the ParamGridBuilder, and add the grid to
the CrossValidator
paramGrid = (ParamGridBuilder()
             .addGrid(lr.regParam, regParam)
             .build())
crossval.setEstimatorParamMaps(paramGrid)

# Now let's find and return the best model
cvModel = crossval.fit(trainingSetDF).bestModel
```

Now that we have tuned our Linear Regression model, let's see what the new RMSE and r^2 values are versus our initial model.

Exercise 7(b)

Complete and run the next cell.

```
> # TODO: Replace <FILL_IN> with the appropriate code.
# Now let's use cvModel to compute an evaluation metric for our test dataset:
testSetDF
predictionsAndLabelsDF = lrModel.transform(testSetDF).select("AT", "V", "AP",
"RH", "PE", "Predicted_PE")

# Run the previously created RMSE evaluator, regEval, on the
predictionsAndLabelsDF DataFrame
rmseNew = regEval.evaluate(predictionsAndLabelsDF)

# Now let's compute the r2 evaluation metric for our test dataset
r2New = regEval.evaluate(predictionsAndLabelsDF, {regEval.metricName: "r2"})

print("Original Root Mean Squared Error: {0:2.2f}".format(rmse))
print("New Root Mean Squared Error: {0:2.2f}".format(rmseNew))
print("Old r2: {0:2.2f}".format(r2))
print("New r2: {0:2.2f}".format(r2New))
```

Original Root Mean Squared Error: 4.59

New Root Mean Squared Error: 4.59

Old r2: 0.93

New r2: 0.93

```
> # TEST
Test.assertEquals(round(rmse, 2), 4.59, "Incorrect value for rmse")
Test.assertEquals(round(rmseNew, 2), 4.59, "Incorrect value for rmseNew")
Test.assertEquals(round(r2, 2), 0.93, "Incorrect value for r2")
Test.assertEquals(round(r2New, 2), 0.93, "Incorrect value for r2New")
```

1 test passed.

1 test passed.

1 test passed.

1 test passed.

So our initial untuned and tuned linear regression models are statistically identical. Let's look at the regularization parameter that the CrossValidator (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Cross>) has selected.

Recall that the original regularization parameter we used was 0.01.

NOTE: The ML Python API currently doesn't provide a way to query the regularization parameter, so we cheat, by "reaching through" to the JVM version of the API.

```
> print("Regularization parameter of the best model:
      {0:.2f}".format(cvModel.stages[-1]._java_obj.parent().getRegParam()))
```

Regularization parameter of the best model: 0.01

Given that the only linearly correlated variable is Temperature, it makes sense try another Machine Learning method such as Decision Tree (https://en.wikipedia.org/wiki/Decision_tree_learning) to handle non-linear data and see if we can improve our model.

Decision Tree Learning (https://en.wikipedia.org/wiki/Decision_tree_learning) uses a Decision Tree (https://en.wikipedia.org/wiki/Decision_tree) as a predictive model which maps observations about an item to conclusions about the item's target value. It is one of the predictive modelling approaches used in statistics, data mining and machine learning. Decision trees where the target variable can take continuous values (typically real numbers) are called regression trees.

Spark ML Pipeline provides `DecisionTreeRegressor()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.DecisionTreeRegressor>) as an implementation of Decision Tree Learning (https://en.wikipedia.org/wiki/Decision_tree_learning).

The cell below is based on the Spark ML Pipeline API for Decision Tree Regressor (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.DecisionTreeRegressor>).

Exercise 7(c)

- Read the Decision Tree Regressor (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.DecisionTreeRegressor>) documentation
- In the next cell, create a `DecisionTreeRegressor()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.DecisionTreeRegressor>)
- The next step is to set the parameters for the method (we do this for you):
 - Set the name of the prediction column to "Predicted_PE"
 - Set the name of the features column to "features"
 - Set the maximum number of bins to 100

- Create the ML Pipeline

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.Pipeline>) and set the stages to the Vectorizer we created earlier and DecisionTreeRegressor() (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.DecisionTreeRegressor>) learner we just created.

```
> # TODO: Replace <FILL_IN> with the appropriate code.
from pyspark.ml.regression import DecisionTreeRegressor
```

```
# Create a DecisionTreeRegressor
dt = DecisionTreeRegressor()
```

```
dt.setLabelCol("PE")\
    .setPredictionCol("Predicted_PE")\
    .setFeaturesCol("features")\
    .setMaxBins(100)
```

```
# Create a Pipeline
dtPipeline = Pipeline()
```

```
# Set the stages of the Pipeline
dtPipeline.setStages([vectorizer, dt])
```

```
Out[33]: Pipeline_4710aa89a148a3315fcd
```

```
> # TEST
```

```
Test.assertEquals(str(dtPipeline.getStages()[0].__class__.__name__),
'4617be70bcf475326c0b07400b97b13457cc4949', "Incorrect pipeline stage 0")
Test.assertEquals(str(dtPipeline.getStages()[1].__class__.__name__),
'46b18f257cf2f778d0d3b6e30ccc7b3398d7846a', "Incorrect pipeline stage 1")
```

```
1 test passed.
```

```
1 test passed.
```

Instead of guessing what parameters to use, we will use Model Selection (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#module-pyspark.ml.tuning>) or Hyperparameter Tuning (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#module-pyspark.ml.tuning>) to create the best model.

We can reuse the existing CrossValidator

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) by replacing the Estimator with our new dtPipeline (the number of folds remains 3).

Exercise 7(d)

- Use ParamGridBuilder

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.ParamGridBuilder>) to build a parameter grid with the parameter dt.maxDepth and a list of the values 2 and 3, and add the grid to the CrossValidator

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>)

- Run the CrossValidator

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) to find the parameters that yield the best model (i.e. lowest RMSE) and return the best model.

Note that it will take some time to run the CrossValidator

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) as it will run almost 50 Spark jobs

```
> # TODO: Replace <FILL_IN> with the appropriate code.
# Let's just reuse our CrossValidator with the new dtPipeline,
# RegressionEvaluator regEval, and 3 fold cross validation
crossval.setEstimator(dtPipeline)

# Let's tune over our dt.maxDepth parameter on the values 2 and 3, create a
# parameter grid using the ParamGridBuilder
paramGrid = (ParamGridBuilder()
              .addGrid(dt.maxDepth, [2,3])
              .build())

# Add the grid to the CrossValidator
crossval.setEstimatorParamMaps(paramGrid)

# Now let's find and return the best model
dtModel = crossval.fit(trainingSetDF).bestModel
```

```
> # TEST
```

```
Test.assertEquals(str(dtModel.stages[0].__class__.__name__),
'4617be70bcf475326c0b07400b97b13457cc4949', "Incorrect pipeline stage 0")
Test.assertEquals(str(dtModel.stages[1].__class__.__name__),
'a2bf7b0c1a0fb9ad35650d0478ad51a9b880befa', "Incorrect pipeline stage 1")
```

```
1 test passed.
```

```
1 test passed.
```

Exercise 7(e)

Now let's see how our tuned DecisionTreeRegressor model's RMSE and r^2 values compare to our tuned LinearRegression model.

Complete and run the next cell.

```
> # TODO: Replace <FILL_IN> with the appropriate code.
```

```
# Now let's use dtModel to compute an evaluation metric for our test dataset:
testSetDF
predictionsAndLabelsDF = dtModel.transform(testSetDF).select("AT", "V", "AP",
"RH", "PE", "Predicted_PE")
```

```
# Run the previously created RMSE evaluator, regEval, on the
predictionsAndLabelsDF DataFrame
rmseDT = regEval.evaluate(predictionsAndLabelsDF)
```

```
# Now let's compute the r2 evaluation metric for our test dataset
r2DT = regEval.evaluate(predictionsAndLabelsDF, {regEval.metricName: "r2"})
```

```
print("LR Root Mean Squared Error: {0:.2f}".format(rmseNew))
print("DT Root Mean Squared Error: {0:.2f}".format(rmseDT))
print("LR r2: {0:.2f}".format(r2New))
print("DT r2: {0:.2f}".format(r2DT))
```

```
LR Root Mean Squared Error: 4.59
```

```
DT Root Mean Squared Error: 5.19
```

```
LR r2: 0.93
```

```
DT r2: 0.91
```



```
> # TEST
Test.assertEquals(round(rmseDT, 2), 5.19, "Incorrect value for rmseDT")
Test.assertEquals(round(r2DT, 2), 0.91, "Incorrect value for r2DT")

1 test passed.
1 test passed.
```

The line below will pull the Decision Tree model from the Pipeline and display it as an if-then-else string. Again, we have to "reach through" to the JVM API to make this one work.

ToDo: Run the next cell

```
> print dtModel.stages[-1]._java_obj.toDebugString()
```

```
DecisionTreeRegressionModel (uid=DecisionTreeRegressor_4d1694081128b692a1ae) o
f depth 3 with 15 nodes
If (feature 0 <= 17.94)
  If (feature 0 <= 11.7)
    If (feature 0 <= 8.69)
      Predict: 483.764931685101
    Else (feature 0 > 8.69)
      Predict: 476.0754881541267
  Else (feature 0 > 11.7)
    If (feature 0 <= 14.38)
      Predict: 469.098125445474
    Else (feature 0 > 14.38)
      Predict: 462.0270406852249
Else (feature 0 > 17.94)
  If (feature 0 <= 23.01)
    If (feature 1 <= 56.65)
      Predict: 454.4627180406213
    Else (feature 1 > 56.65)
      Predict: 447.35858133971294
  Else (feature 0 > 23.01)
    If (feature 1 <= 65.75)
```

So our DecisionTree has slightly worse RMSE than our LinearRegression model (LR: 4.59 vs DT: 5.19). Maybe we can try an Ensemble Learning (https://en.wikipedia.org/wiki/Ensemble_learning) method such as Gradient-Boosted

Decision Trees (https://en.wikipedia.org/wiki/Gradient_boosting) to see if we can strengthen our model by using an ensemble of weaker trees with weighting to reduce the error in our model.

Random forests (https://en.wikipedia.org/wiki/Random_forest) or random decision tree forests are an ensemble learning method for regression that operate by constructing a multitude of decision trees at training time and outputting the class that is the mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

Spark ML Pipeline provides `RandomForestRegressor()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.F>) as an implementation of Random forests (https://en.wikipedia.org/wiki/Random_forest).

The cell below is based on the Spark ML Pipeline API for Random Forest Regressor (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.F>).

Exercise 7(f)

- Read the Random Forest Regressor (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.F>) documentation
- In the next cell, create a `RandomForestRegressor()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.F>)
- The next step is to set the parameters for the method (we do this for you):
 - Set the name of the prediction column to "Predicted_PE"
 - Set the name of the features column to "features"
 - Set the random number generator seed to 100088121L
 - Set the maximum depth to 8
 - Set the number of trees to 30
- Create the ML Pipeline (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.Pipeline>) and set the stages to the Vectorizer we created earlier and `RandomForestRegressor()` (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.regression.F>) learner we just created.

```
> # TODO: Replace <FILL_IN> with the appropriate code.
```

```
from pyspark.ml.regression import RandomForestRegressor
```

```
# Create a RandomForestRegressor
```

```
rf = RandomForestRegressor()
```

```
rf.setLabelCol("PE")\
  .setPredictionCol("Predicted_PE")\
  .setFeaturesCol("features")\
  .setSeed(100088121L)\
  .setMaxDepth(8)\
  .setNumTrees(30)
```

```
# Create a Pipeline
```

```
rfPipeline = Pipeline()
```

```
# Set the stages of the Pipeline
```

```
rfPipeline.setStages([vectorizer, rf])
```

```
Out[40]: Pipeline_4114b9e161c5ef098d8f
```

```
> # TEST
```

```
Test.assertEqualsHashed(rfPipeline.getStages()[0].__class__.__name__,
  '4617be70bcf475326c0b07400b97b13457cc4949', "Stage 0 of pipeline is not
  correct")
```

```
Test.assertEqualsHashed(rfPipeline.getStages()[1].__class__.__name__,
  'ecdccce2d075f00c97a6d2a2b8b1f66de322e57d2', "Stage 1 of pipeline is not
  correct")
```

```
1 test passed.
```

```
1 test passed.
```

As with Decision Trees, instead guessing what parameters to use, we will use Model Selection (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#module-pyspark.ml.tuning>) or Hyperparameter Tuning (<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#module-pyspark.ml.tuning>) to create the best model.

We can reuse the existing CrossValidator

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.Cross> by replacing the Estimator with our new rfPipeline (the number of folds remains 3).

Exercise 7(g)

- Use ParamGridBuilder
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.ParamGridBuilder>) to build a parameter grid with the parameter `rf.maxBins` and a list of the values 50 and 100, and add the grid to the CrossValidator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>)
- Run the CrossValidator
(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) to find the parameters that yield the best model (i.e., lowest RMSE) and return the best model.

Note that it will take some time to run the CrossValidator

(<https://spark.apache.org/docs/1.6.2/api/python/pyspark.ml.html#pyspark.ml.tuning.CrossValidator>) as it will run almost 100 Spark jobs, and each job takes longer to run than the prior CrossValidator runs.

```
> # TODO: Replace <FILL_IN> with the appropriate code.
# Let's just reuse our CrossValidator with the new rfPipeline,
# RegressionEvaluator regEval, and 3 fold cross validation
crossval.setEstimator(rfPipeline)

# Let's tune over our rf.maxBins parameter on the values 50 and 100, create a
# parameter grid using the ParamGridBuilder
paramGrid = (ParamGridBuilder()
              .addGrid(rf.maxBins, [50,100])
              .build())

# Add the grid to the CrossValidator
crossval.setEstimatorParamMaps(paramGrid)

# Now let's find and return the best model
rfModel = crossval.fit(trainingSetDF).bestModel

> # TEST
Test.assertEqualsHashed(rfModel.stages[0].__class__,
                        'f0c3b910468d87808e019409e7ae5e587d6aca3d', 'rfModel has incorrect stage 0')
Test.assertEqualsHashed(rfModel.stages[1].__class__,
                        '0ed43512ea7e35ebeebeed3ddac0186248999a87', 'rfModel has incorrect stage 1')
```

1 test passed.

1 test passed.

Exercise 7(h)

Now let's see how our tuned RandomForestRegressor model's RMSE and r^2 values compare to our tuned LinearRegression and tuned DecisionTreeRegressor models.

Complete and run the next cell.

```
> # TODO: Replace <FILL_IN> with the appropriate code.

# Now let's use rfModel to compute an evaluation metric for our test dataset:
testSetDF
predictionsAndLabelsDF = rfModel.transform(testSetDF).select("AT", "V", "AP",
"RH", "PE", "Predicted_PE")

# Run the previously created RMSE evaluator, regEval, on the
predictionsAndLabelsDF DataFrame
rmseRF = regEval.evaluate(predictionsAndLabelsDF)

# Now let's compute the r2 evaluation metric for our test dataset
r2RF = regEval.evaluate(predictionsAndLabelsDF, {regEval.metricName: "r2"})

print("LR Root Mean Squared Error: {0:.2f}".format(rmseNew))
print("DT Root Mean Squared Error: {0:.2f}".format(rmseDT))
print("RF Root Mean Squared Error: {0:.2f}".format(rmseRF))
print("LR r2: {0:.2f}".format(r2New))
print("DT r2: {0:.2f}".format(r2DT))
print("RF r2: {0:.2f}".format(r2RF))

LR Root Mean Squared Error: 4.59
DT Root Mean Squared Error: 5.19
RF Root Mean Squared Error: 3.55
LR r2: 0.93
DT r2: 0.91
RF r2: 0.96

> # TEST
Test.assertEquals(round(rmseRF, 2), 3.55, "Incorrect value for rmseRF")
Test.assertEquals(round(r2RF, 2), 0.96, "Incorrect value for r2RF")

1 test passed.
1 test passed.
```

Note that the r^2 values are similar for all three. However, the RMSE for the Random Forest model is better.

The line below will pull the Random Forest model from the Pipeline and display it as an if-then-else string.

ToDo: Run the next cell

```
> print rfModel.stages[-1]._java_obj.toDebugString()
```

```
RandomForestRegressionModel (uid=rfr_67ce4ddaed1a) with 30 trees
Tree 0 (weight 1.0):
  If (feature 2 <= 1014.68)
    If (feature 3 <= 76.75)
      If (feature 2 <= 1011.52)
        If (feature 0 <= 21.08)
          If (feature 1 <= 43.34)
            If (feature 2 <= 1003.35)
              If (feature 3 <= 65.61)
                If (feature 1 <= 39.82)
                  Predict: 456.652676056338
                Else (feature 1 > 39.82)
                  Predict: 458.93767441860473
              Else (feature 3 > 65.61)
                If (feature 0 <= 9.98)
                  Predict: 478.9557142857143
                Else (feature 0 > 9.98)
                  Predict: 461.34870129870126
            Else (feature 2 > 1003.35)
              If (feature 2 <= 1005.61)
                If (feature 1 <= 39.54)
```

Conclusion

Wow! So our best model is in fact our Random Forest tree model which uses an ensemble of 30 Trees with a depth of 8 to construct a better model than the single decision tree.

Appendix A: Submitting Your Exercises to the Autograder

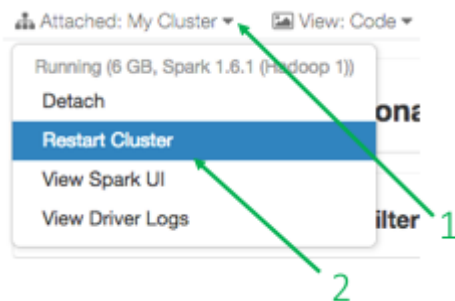
This section guides you through Step 2 of the grading process ("Submit to Autograder").

Once you confirm that your lab notebook is passing all tests, you can submit it first to the course autograder and then second to the edX website to receive a grade.

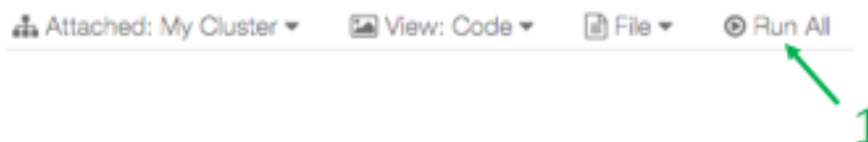
Note that you can only submit to the course autograder once every 1 minute.

Step 2(a): Restart your cluster by clicking on the dropdown next to your cluster name and selecting "Restart Cluster".

You can do this step in either notebook, since there is one cluster for your notebooks.



Step 2(b): *IN THIS NOTEBOOK*, click on "Run All" to run all of the cells.



This step will take some time.

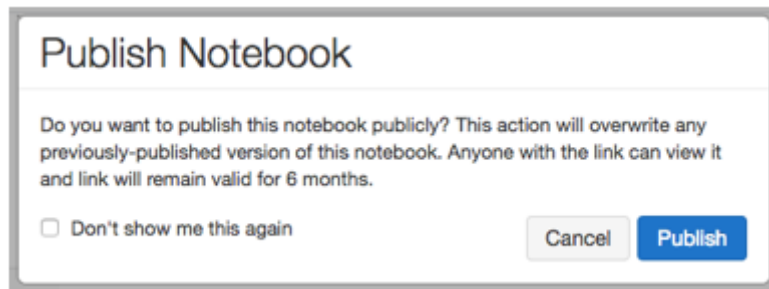
Wait for your cluster to finish running the cells in your lab notebook before proceeding.

Step 2(c): Publish this notebook

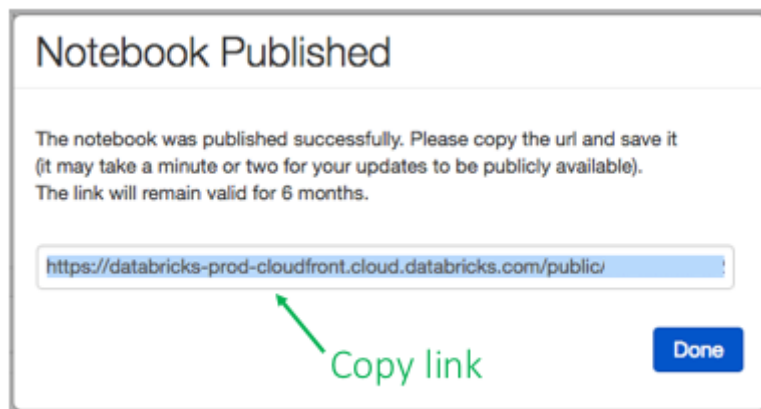
Publish *this* notebook by clicking on the "Publish" button at the top.



When you click on the button, you will see the following popup.



When you click on "Publish", you will see a popup with your notebook's public link. **Copy the link and set the notebook_URL variable in the AUTOGRADER notebook (not this notebook).**



Step 2(d): Set the notebook URL and Lab ID in the Autograder notebook, and run it

Go to the Autograder notebook and paste the link you just copied into it, so that it is assigned to the notebook_url variable.

```
notebook_url = "... " # put your URL here
```


Then, find the line that looks like this:

```
lab = <FILL IN>
```

and change <FILL IN> to "CS110x-lab1":

```
lab = "CS110x-lab1"
```

Then, run the Autograder notebook to submit your lab.



If things go wrong

It's possible that your notebook looks fine to you, but fails in the autograder. (This can happen when you run cells out of order, as you're working on your notebook.) If that happens, just try again, starting at the top of Appendix A.