5.1 Introduction

The last lesson introduced us to one of the most popular distributed data processing platforms used to process big data—Spark.

In this lesson, we will learn more about how to work with Spark and Spark DataFrames using its Python API—**PySpark**. It gives us the capability to process petabyte-scale data, but also implements **machine learning** (**ML**) algorithms at petabyte scale in real time. This lesson will focus on the data processing part using Spark DataFrames in PySpark.

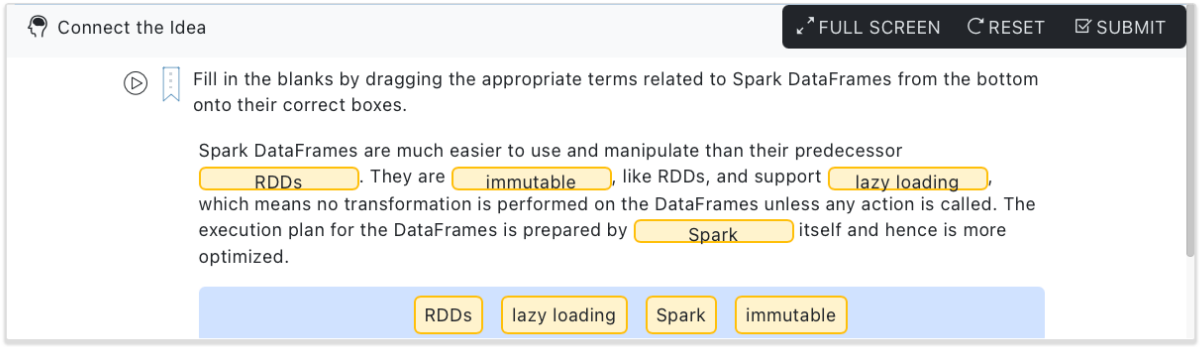
> **Note**
>
> We will be using the term DataFrame quite frequently during this lesson. This will explicitly refer to the Spark DataFrame, unless mentioned otherwise. Please do not confuse this with the pandas DataFrame.

Spark DataFrames are a distributed collection of data organized as named columns. They are inspired from R and Python DataFrames and have complex optimizations at the backend that make them fast, optimized, and scalable.

The DataFrame API was developed as part of **Project Tungsten** and is designed to improve the performance and scalability of Spark. It was first introduced with Spark 1.3.

Spark DataFrames are much easier to use and manipulate than their predecessor RDDs. They are *immutable,* like RDDs, and support lazy loading, which means no transformation is performed on the DataFrames unless an action is called. The execution plan for the DataFrames is prepared by Spark itself and hence is more optimized, making operations on DataFrames faster than those on RDDs.

Knowledge Check



5.2 Getting Started with Spark DataFrames

To get started with Spark DataFrames, we will have to create something called a SparkContext first. SparkContext configures the internal services under the hood and facilitates command execution from the Spark execution environment.

> **Note**
>
> We will be using Spark version 2.1.1, running on Python 3.7.1. Spark and Python are installed on a MacBook Pro, running macOS Mojave version 10.14.3, with a 2.7 GHz Intel Core i5 processor and 8 GB 1867 MHz DDR3 RAM.
>
> The following code snippet is used to create **SparkContext**:

```
from pyspark import SparkContext
sc = SparkContext()
```

> **Note**
>
> In case you are working in the PySpark shell, you should skip this step, as the shell automatically creates the **sc (SparkContext)** variable when it is started. However, be sure to create the **sc** variable while creating a PySpark script or working with Jupyter Notebook, or your code will throw an error.

We also need to create an **SQLContext** before we can start working with DataFrames. **SQLContext** in Spark is a class that provides SQL-like functionality within Spark. We can create **SQLContext** using **SparkContext**:

```
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)
```

There are three different ways of creating a DataFrame in Spark:

- We can programmatically specify the schema of the DataFrame and manually enter the data in it. However, since Spark is generally used to handle big data, this method is of little use, apart from creating data for small test/sample cases.
- Another method to create a DataFrame is from an existing RDD object in Spark. This is useful, because working on a DataFrame is way easier than working directly with RDDs.
- We can also read the data directly from a data source to create a Spark DataFrame. Spark supports a variety of external data sources, including CSV, JSON, parquet, RDBMS tables, and Hive tables.

Exercise 24: Specifying the Schema of a DataFrame

In this exercise, we will create a small sample DataFrame by manually specifying the schema and entering data in Spark. Even though this method has little application in a practical scenario, it will be a good starting point in getting started with Spark DataFrames:

1. Importing the necessary files:

```
from pyspark import SparkContext
sc = SparkContext()
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)
```

2. Import SQL utilities from the PySpark module and specify the schema of the sample DataFrame:

```
from pyspark.sql import *
na_schema = Row("Name","Age")
```

3. Create rows for the DataFrame as per the specified schema:

3. Create rows for the DataFrame as per the specified schema:

```
row1 = na_schema("Ankit", 23)
row2 = na_schema("Tyler", 26)
row3 = na_schema("Preity", 36)
```

4. Combine the rows together to create the DataFrame:

```
na_list = [row1, row2, row3]
df_na = sqlc.createDataFrame(na_list)
type(df_na)
```

5. Now, show the DataFrame using the following command:

```
df_na.show()
```

The output is as follows:

```
+------+---+
|  Name|Age|
+------+---+
| Ankit| 23|
| Tyler| 26|
|Preity| 36|
+------+---+
```

Figure 4.1: Sample PySpark DataFrame

Exercise 25: Creating a DataFrame from an Existing RDD

In this exercise, we will create a small sample DataFrame from an existing RDD object in Spark:

1. Create an RDD object that we will convert into DataFrame:

```
data = [("Ankit",23),("Tyler",26),("Preity",36)]
data_rdd = sc.parallelize(data)
type(data_rdd)
```

2. Convert the RDD object into a DataFrame:

```
data_sd = sqlc.createDataFrame(data_rdd)
```

3. Now, show the DataFrame using the following command:

```
data_sd.show()
```

```
+------+---+
|    _1| _2|
+------+---+
| Ankit| 23|
| Tyler| 26|
|Preity| 36|
+------+---+
```

Figure 4.2: DataFrame converted from the RDD object

Exercise 26: Creating a DataFrame Using a CSV File

A variety of different data sources can be used to create a DataFrame. In this exercise, we will use the open source Iris dataset, which can be found under datasets in the scikit-learn library. The Iris dataset is a multivariate dataset containing 150 records, with 50 records for each of the 3 species of Iris flower (Iris Setosa, Iris Virginica, and Iris Versicolor).

The dataset contains five attributes for each of the Iris species, namely, **`petal length`**, **`petal width`**, **`sepal length`**, **`sepal width`**, and **`species`**. We have stored this dataset in an external CSV file that we will read into Spark:

1. Download and install the PySpark CSV reader package from the Databricks website:

```
pyspark –packages com.databricks:spark-csv_2.10:1.4.0
```

2. Read the data from the CSV file into the Spark DataFrame:

```
df = sqlc.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load('iri
type(df)
```

3. Now, show the DataFrame using the following command:

```
df.show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.3: Iris DataFrame, first four rows

Instructor Note: Motivate the students to explore other data sources, such as tab-separated files, parquet files, and relational databases, as well.

5.3 Writing Output from Spark DataFrames

Spark gives us the ability to write the data stored in Spark DataFrames into a local pandas DataFrame, or write them into external structured file formats such as CSV. However, before converting a Spark DataFrame into a local pandas DataFrame, make sure that the data would fit in the local driver memory.

In the following exercise, we will explore how to convert the Spark DataFrame to a pandas DataFrame.

Exercise 27: Converting a Spark DataFrame to a Pandas DataFrame

In this exercise, we will use the pre-created Spark DataFrame of the Iris dataset in the previous exercise, and convert it into a local pandas DataFrame. We will then store this DataFrame into a CSV file. Perform the following steps:

1. Convert the Spark DataFrame into a pandas DataFrame using the following command:

```
import pandas as pd
df.toPandas()
```

2. Now use the following command to write the pandas DataFrame to a CSV file:

```
df.toPandas().to_csv('iris.csv')
```

Note

Writing the contents of a Spark DataFrame to a CSV file requires a one-liner using the **spark-csv** package:

```
df.write.csv('iris.csv')
```

5.4 Exploring Spark DataFrames

One of the major advantages that the Spark DataFrames offer over the traditional RDDs is the ease of data use and exploration. The data is stored in a more structured tabular format in the DataFrames and hence is easier to make sense of. We can compute basic statistics such as the number of rows and columns, look at the schema, and compute summary statistics such as mean and standard deviation.

Exercise 28: Displaying Basic DataFrame Statistics

In this exercise, we will show basic DataFrame statistics of the first few rows of the data, and summary statistics for all the numerical DataFrame columns and an individual DataFrame column:

1. Look at the DataFrame schema. The schema is displayed in a tree format on the console:

```
df.printSchema()
```

```
root
 |-- Sepallength: double (nullable = true)
 |-- Sepalwidth: double (nullable = true)
 |-- Petallength: double (nullable = true)
 |-- Petalwidth: double (nullable = true)
 |-- Species: string (nullable = true)
```

Figure 4.4: Iris DataFrame schema

2. Now, use the following command to print the column names of the Spark DataFrame:

```
df.schema.names
```

```
['Sepallength', 'Sepalwidth', 'Petallength', 'Petalwidth', 'Species']
```

Figure 4.5: Iris column names

3. To retrieve the number of rows and columns present in the Spark DataFrame, use the following command:

```
## Counting the number of rows in DataFrame
df.count()#134
## Counting the number of columns in DataFrame
len(df.columns)#5
```

4. Let's fetch the first *n* rows of the data. We can do this by using the **head()** method. However, we use the **show()** method as it displays the data in a better format:

```
df.show(4)
```

The output is as follows:

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.6: Iris DataFrame, first four rows

5. Now, compute the summary statistics, such as mean and standard deviation, for all the numerical columns in the DataFrame:

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

The output is as follows:

```
+-------+------------------+-------------------+------------------+------------------+---------+
|summary|       Sepallength|         Sepalwidth|       Petallength|        Petalwidth|  Species|
+-------+------------------+-------------------+------------------+------------------+---------+
|  count|               148|                150|               149|               150|      150|
|   mean| 5.854729729729732|  3.057333333333334|3.7744966442953043| 1.199333333333334|     null|
| stddev|0.8277774898579762|0.43586628493669793|1.7596127630823133|0.7622376689603467|     null|
|    min|               4.3|                2.0|               1.0|               0.1|   setosa|
|    max|               7.9|                4.4|               6.9|               2.5|virginica|
+-------+------------------+-------------------+------------------+------------------+---------+
```

Figure 4.7: Iris DataFrame, summary statistics

6. To compute the summary statistics for an individual numerical column of a Spark DataFrame, use the following command:

```
df.describe('Sepalwidth').show()
```

The output is as follows:

```
+-------+-------------------+
|summary|         Sepalwidth|
+-------+-------------------+
|  count|                150|
|   mean|  3.057333333333334|
| stddev|0.43586628493669793|
|    min|                2.0|
|    max|                4.4|
+-------+-------------------+
```

Figure 4.8: Iris DataFrame, summary statistics of Sepalwidth column

Activity 9: Getting Started with Spark DataFrames

In this activity, we will use the concepts learned in the previous sections and create a Spark DataFrame using all three methods. We will also compute DataFrame statistics, and finally, write the same data into a CSV file. Feel free to use any open source dataset for this activity:

1. Create a sample DataFrame by manually specifying the schema.
2. Create a sample DataFrame from an existing RDD.
3. Create a sample DataFrame by reading the data from a CSV file.
4. Print the first seven rows of the sample DataFrame read in step 3.
5. Print the schema of the sample DataFrame read in step 3.
6. Print the number of rows and columns in the sample DataFrame.
7. Print the summary statistics of the DataFrame and any 2 individual numerical columns.
8. Write the first 7 rows of the sample DataFrame to a CSV file using both methods mentioned in the exercises.

5.5 Data Manipulation with Spark DataFrames

Data manipulation is a prerequisite for any data analysis. To draw meaningful insights from the data, we first need to understand, process, and massage the data. Data manipulation operations are filtering, sampling, and aggregating. But this step becomes particularly hard with the increase in the size of data. Due to the scale of data, even simple operations such as filtering and sorting become complex coding problems. Spark DataFrames make data manipulation on big data a piece of cake.

Manipulating the data in Spark DataFrames is quite like working on regular pandas DataFrames. Most of the data manipulation operations on Spark DataFrames can be done using simple and intuitive one-liners. We will use the Spark DataFrame containing the Iris dataset that we created in previous exercises for these data manipulation exercises.

Exercise 29: Selecting and Renaming Columns from the DataFrame

In this exercise, we will first rename the column using the **withColumnRenamed** method and then select and print the schema using the **select** method. You can also use the **rename** method to rename columns in pandas.

Perform the following steps:

1. Rename the columns of a Spark DataFrame using the **withColumnRenamed()** method:

```
In Spark:
df = df.withColumnRenamed('Sepal.Width','Sepalwidth')

In Pandas:
df = df.rename(columns = {'Sepal.Width':'Sepalwidth'})
```

> **Note**
>
> Spark does not recognize column names containing a period(**.**). Make sure to rename them using this method.

2. Select a single column or multiple columns from a Spark DataFrame using the **select** method:

```
df.select('Sepalwidth','Sepallength').show(4)
```

```
+----------+-----------+
|Sepalwidth|Sepallength|
+----------+-----------+
|       3.5|        5.1|
|       3.0|        4.9|
|       3.2|        4.7|
|       3.1|        4.6|
+----------+-----------+
only showing top 4 rows
```

Figure 4.9: Iris DataFrame, Sepalwidth and Sepallength column

Exercise 30: Adding and Removing a Column from the DataFrame

In this exercise, we will add a new column in the dataset using the **withColumn** method, and later, using the **drop** function, will remove it. Now, let's perform the following steps:

1. Add a new column in a Spark DataFrame using the **withColumn** method:

```
In Spark:
df = df.withColumn('Half_sepal_width', df['Sepalwidth']/2.0)

In pandas:
df.insert(2, "Half_sepal_width",2.0)
```

2. Use the following command to show the dataset with the newly added column:

```
df.show(4)
```

```
+-----------+----------+-----------+----------+-------+---------------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|Half_sepal_width|
+-----------+----------+-----------+----------+-------+---------------+
|        5.1|       3.5|        1.4|       0.2| setosa|           1.75|
|        4.9|       3.0|        1.4|       0.2| setosa|            1.5|
|        4.7|       3.2|        1.3|       0.2| setosa|            1.6|
|        4.6|       3.1|        1.5|       0.2| setosa|           1.55|
+-----------+----------+-----------+----------+-------+---------------+
only showing top 4 rows
```

Figure 4.10: Introducing new column, Half_sepal_width

3. Now, to remove a column in a Spark DataFrame, use the **drop** method illustrated here:

```
In Spark:
df = df.drop('Half_sepal_width')

In pandas:
df = df.drop(['Sepal.Length'], axis = 1)
```

4. Let's show the dataset to verify that the column has been removed:

```
df.show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.11: Iris DataFrame after dropping the Half_sepal_width column

Exercise 31: Displaying and Counting Distinct Values in a DataFrame

To display the distinct values in a DataFrame, we use the **distinct().show()** method. Similarly, to count the distinct values, we will be using the **distinct().count()** method. Perform the following procedures to print the distinct values with the total count:

1. Select the distinct values in any column of a Spark DataFrame using the **distinct** method, in conjunction with the **select** method:

```
df.select('Species').distinct().show()
```

```
+----------+
|   Species|
+----------+
| virginica|
|versicolor|
|    setosa|
+----------+
```

Figure 4.12: Iris DataFrame, Species column

2. To count the distinct values in any column of a Spark DataFrame, use the **count** method, in conjunction with the **distinct** method:

```
df.select('Species').distinct().count()
```

Exercise 32: Removing Duplicate Rows and Filtering Rows of a DataFrame

In this exercise, we will learn how to remove the duplicate rows from the dataset, and later, perform filtering operations on the same column.

Perform these steps:

1. Remove the duplicate values from a DataFrame using the **dropDuplicates()** method:

```
df.select('Species').dropDuplicates().show()
```

```
+----------+
|   Species|
+----------+
| virginica|
|versicolor|
|    setosa|
+----------+
```

Figure 4.13: Iris DataFrame, Species column after removing the duplicate column

2. Filter the rows from a DataFrame using one or multiple conditions. These multiple conditions can be passed together to the DataFrame using Boolean operators such as and (**&**), or **|**, similar to how we do it for pandas DataFrames:

```
# Filtering using a single condition
df.filter(df.Species == 'setosa').show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.14: Iris DataFrame after filtering with single conditions

3. Now, to filter the column using multiple conditions, use the following command:

```
df.filter((df.Sepallength > 5) & (df.Species == 'setosa')).show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|       null|       3.4|        1.6|       0.4| setosa|
|       null|       3.0|        1.6|       0.2| setosa|
|        4.3|       3.0|        1.1|       0.1| setosa|
|        4.4|       3.0|       null|       0.2| setosa|
|        4.4|       2.9|        1.4|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 5 rows
```

Exercise 33: Ordering Rows in a DataFrame

In this exercise, we will explore how to sort the rows in a DataFrame in ascending and descending order. Let's perform these steps:

1. Sort the rows in a DataFrame, using one or multiple conditions, in ascending or descending order:

```
df.orderBy(df.Sepallength).show(5)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|       null|       3.4|        1.6|       0.4| setosa|
|       null|       3.0|        1.6|       0.2| setosa|
|        4.3|       3.0|        1.1|       0.1| setosa|
|        4.4|       3.0|       null|       0.2| setosa|
|        4.4|       2.9|        1.4|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 5 rows
```

Figure 4.16: Filtered Iris DataFrame

2. To sort the rows in descending order, use the following command:

```
df.orderBy(df.Sepallength.desc()).show(5)
```

```
+-----------+----------+-----------+----------+---------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|  Species|
+-----------+----------+-----------+----------+---------+
|        7.9|       3.8|        6.4|       2.0|virginica|
|        7.7|       2.6|        6.9|       2.3|virginica|
|        7.7|       3.8|        6.7|       2.2|virginica|
|        7.7|       3.0|        6.1|       2.3|virginica|
|        7.7|       2.8|        6.7|       2.0|virginica|
+-----------+----------+-----------+----------+---------+
only showing top 5 rows
```

Figure 4.17: Iris DataFrame after sorting it in the descending order

Exercise 34: Aggregating Values in a DataFrame

We can group the values in a DataFrame by one or more variables, and calculate aggregated metrics such as **`mean`**, **`sum`**, **`count`**, and many more. In this exercise, we will calculate the mean sepal width for each of the flower species in the Iris dataset. We will also calculate the count of the rows for each species:

1. To calculate the mean sepal width for each species, use the following command:

```
df.groupby('Species').agg({'Sepalwidth' : 'mean'}).show()
```

```
+----------+------------------+
|   Species|   avg(Sepalwidth)|
+----------+------------------+
| virginica|2.9739999999999998|
|versicolor|2.7700000000000005|
|    setosa| 3.428000000000001|
+----------+------------------+
```

Figure 4.18: Iris DataFrame, calculating mean sepal width

2. Now, let's calculate the number of rows for each species by using the following command:

2. Now, let's calculate the number of rows for each species by using the following command:

```
df.groupby('Species').count().show()
```

```
+----------+-----+
|   Species|count|
+----------+-----+
| virginica|   50|
|versicolor|   50|
|    setosa|   50|
+----------+-----+
```

Figure 4.19: Iris DataFrame, calculating the number of rows for each species

> **Note**
>
> In the second code snippet, the count can also be used with the `.agg` function; however, the method we used is more popular.

Activity 10: Data Manipulation with Spark DataFrames

In this activity, we will use the concepts learned in the previous sections to manipulate the data in the Spark DataFrame created using the Iris dataset. We will perform basic data manipulation steps to test our ability to work with data in a Spark DataFrame. Feel free to use any open-source dataset for this activity. Make sure the dataset you use has both numerical and categorical variables:

1. Rename any five columns of the DataFrame. If the DataFrame has more than columns, rename all the columns.
2. Select two numeric and one categorical column from the DataFrame.
3. Count the number of distinct categories in the categorical variable.
4. Create two new columns in the DataFrame by summing up and multiplying together the two numerical columns.
5. Drop both the original numerical columns.
6. Sort the data by the categorical column.
7. Calculate the mean of the summation column for each distinct category in the categorical variable.
8. Filter the rows with values greater than the mean of all the mean values calculated in step 7.
9. De-duplicate the resultant DataFrame to make sure it has only unique records.

Activity 10 Solution Download

5.6 Graphs in Spark

The ability to effectively visualize data is of paramount importance. Visual representations of data help the user develop a better understanding of data and uncover trends that might go unnoticed in text form. There are numerous types of plots available in Python, each with its own context.

We will be exploring some of these plots, including bar charts, density plots, boxplots, and linear plots for Spark DataFrames, using the widely used Python plotting packages of Matplotlib and Seaborn. The point to note here is that Spark deals with big data. So, make sure that your data size is reasonable enough (that is, it fits in your computer's RAM) before plotting it. This can be achieved by filtering, aggregating, or sampling the data before plotting it.

We are using the Iris dataset, which is small, hence we do not need to do any such pre-processing steps to reduce the data size.

> **Note**
>
> The user should install and load the Matplotlib and Seaborn packages beforehand, in the development environment, before getting started with the exercises in this section. If you are unfamiliar with installing and loading these packages, visit the official websites of Matplotlib and Seaborn.

Exercise 35: Creating a Bar Chart

In this exercise, we will try to plot the number of records available for each species using a bar chart. We will have to first aggregate the data and count the number of records for each species. We can then convert this aggregated data into a regular pandas DataFrame and use Matplotlib and Seaborn packages to create any kind of plots of it that we wish:

1. First, calculate the number of rows for each flower species and convert the result to a pandas DataFrame:

```
data = df.groupby('Species').count().toPandas()
```

2. Now, create a bar plot from the resulting pandas DataFrame:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.barplot( x = data['Species'], y = data['count'])
plt.xlabel('Species')
plt.ylabel('count')
plt.title('Number of rows per species')
```

The plot is as follows:

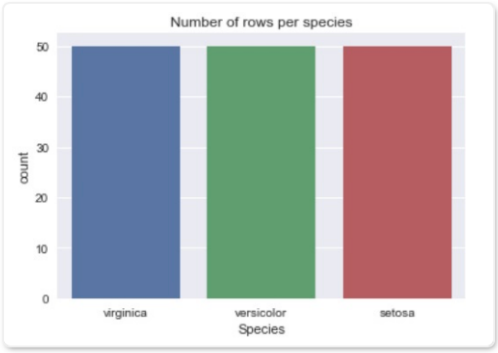


Figure 4.20: Bar plot for Iris DataFrame after calculating the number of rows for each flower species

Exercise 36: Creating a Linear Model Plot

In this exercise, we will plot the data points of two different variables and fit a straight line on them. This is similar to fitting a linear model on two variables and can help identify correlations between the two variables:

1. Create a **data** object from the pandas DataFrame:

```
data = df.toPandas()
sns.lmplot(x = "Sepallength", y = "Sepalwidth", data = data)
```

2. Plot the DataFrame using the following command:

```
plt.show()
```

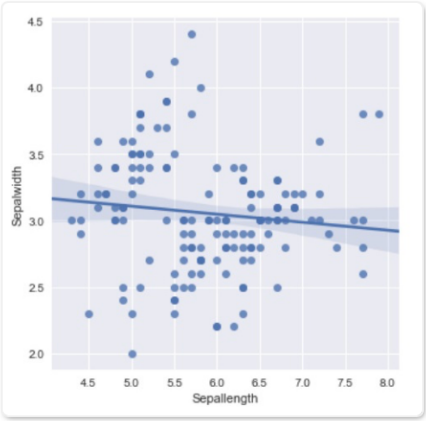


Figure 4.21: Linear model plot for Iris DataFrame

Exercise 37: Creating a KDE Plot and a Boxplot

In this exercise, we will create a **kernel density estimation** (**KDE**) plot, followed by a **boxplot**. Follow these instructions:

1. First, plot a KDE plot that shows us the distribution of a variable. Make sure it gives us an idea of the skewness and the kurtosis of a variable:

```
import seaborn as sns
data = df.toPandas()
sns.kdeplot(data.Sepalwidth, shade = True)
plt.show()
```

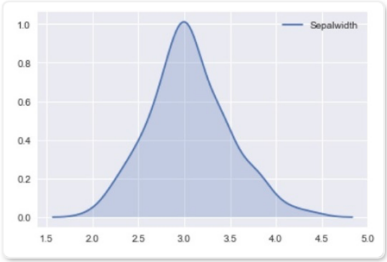


Figure 4.22: KDE plot for the Iris DataFrame

2. Now, plot the boxplots for the Iris dataset using the following command:

```
sns.boxplot(x = "Sepallength", y = "Sepalwidth", data = data)
plt.show()
```

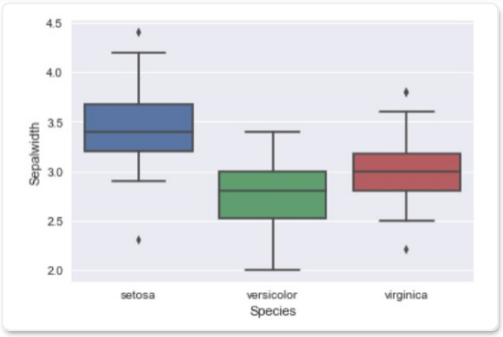


Figure 4.23: Boxplot for the Iris DataFrame

Boxplots are a good way to look at the data distribution and locate outliers. They represent the distribution using the 1st quartile, the median, the 3rd quartile, and the interquartile range (25th to 75th percentile).

Activity 11: Graphs in Spark

In this activity, we will use the plotting libraries of Python to visually explore our data using different kind of plots. For this activity, we are using the **`mtcars`** dataset from Kaggle:

1. Import all the required packages and libraries in the Jupyter Notebook.
2. Read the data into Spark object from the **`mtcars`** dataset.
3. Visualize the discrete frequency distribution of any continuous numeric variable from your dataset using a histogram:

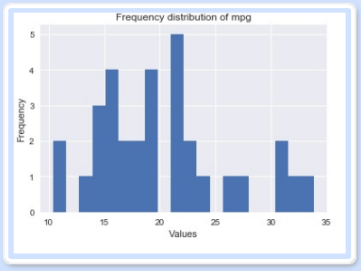


Figure 4.24: Histogram for the Iris DataFrame

4. Visualize the percentage share of the categories in the dataset using a pie chart:

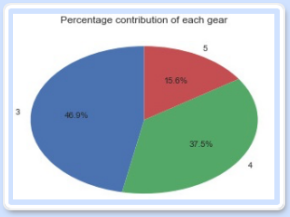


Figure 4.25: Pie chart for the Iris DataFrame

5. Plot the distribution of a continuous variable across the categories of a categorical variable using a boxplot:

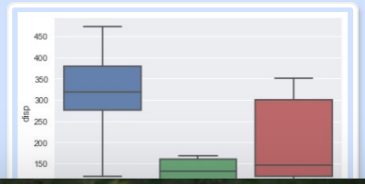


Figure 4.25: Pie chart for the Iris DataFrame

5. Plot the distribution of a continuous variable across the categories of a categorical variable using a boxplot:

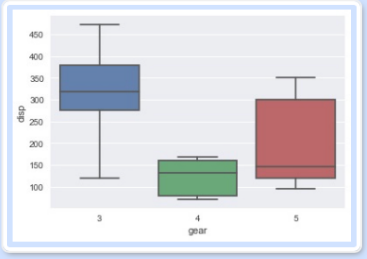


Figure 4.26: Boxplot for the Iris DataFrame

6. Visualize the values of a continuous numeric variable using a line chart:

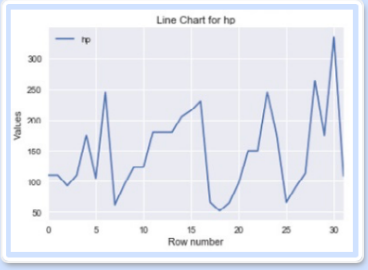


Figure 4.27: Line chart for the Iris DataFrame

7. Plot the values of multiple continuous numeric variables on the same line chart:

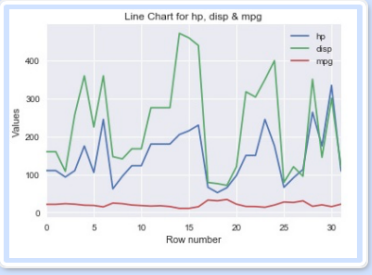


Figure 4.28: Line chart for the Iris DataFrame plotting multiple continuous numeric variables

Activity 11 Solution Download

Activity 11 Solution Download

5.7 Summary

In this lesson, we saw a basic introduction of Spark DataFrames and how they are better than RDDs. We explored different ways of creating Spark DataFrames and writing the contents of Spark DataFrames to regular pandas DataFrames and output files.

We tried out hands-on data exploration in PySpark by computing basic statistics and metrics for Spark DataFrames. We played around with the data in Spark DataFrames and performed data manipulation operations such as filtering, selection, and aggregation. We tried our hands at plotting the data to generate insightful visualizations.

Furthermore, we consolidated our understanding of various concepts by practicing hands-on exercises and activities.

In the next lesson, we will explore how to handle missing values and compute correlation between variables in PySpark.

Glossary

SparkContext

Configures the internal services under the hood and facilitates command execution from the Spark execution environment.

PySpark

An interface for Apache Spark in Python.

SQLContext

A class for initializing the functionalities of Spark SQL.

drop

A function that removes an object from a database.

Click Start to take the Post Assessment.
Your current score of Post Assessment is 0/100%. START

Next Steps

1. Study flashcards to ensure your understanding of the material. Open
2. Quiz yourself to check your understanding of fundamental facts. Open
3. Gain experience using hands-on lab. Open
4. Proceed to the next lesson. Open