PySpark Tutorial

What is Spark?

- Spark is one of the latest technologies being used to quickly and easily handle Big Data
- It is an open source project on **Apache**
- It was first released in February 2013 and has exploded in popularity due to it's ease of use and speed
- It was created at the AMPLab at UC Berkeley
- Spark is 100 times faster than **Hadoop MapReduce**
- Spark does not store anything unless any action is applied on the data

Libraries and Utilities

We need to install pyspark first. Install using command prompt

```
pip install pyspark
```

Or if you are using anaconda, using the anaconda prompt to using using the following

```
conda install -c conda-forge pyspark
```

Start your first pyspark program by import the appropriate libraries as below

```
import warnings
warnings.filterwarnings('ignore')
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, StringType, IntegerType, FloatType
from pyspark.sql.functions import split, count, when, isnan, col, regexp_replace
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import OneHotEncoder, StringIndexer
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
```

Then create a SparkSession

```
spark = SparkSession.builder.appName('First Session').getOrCreate()
print('Spark Version: {}'.format(spark.version))
```

Next, we create a **DataFrame** by reading data from an external source, such as a CSV file. Other example of external sources may include a JSON file, a Parquet file, a Hive table, or by converting an RDD to a **DataFrame**. We then create the schema and display the schema of the **DataFrame**: The schema of a **DataFrame** defines the names and types of the columns. You can display the schema of a **DataFrame** using the **printSchema()** method. Besides, we can display the first few rows of a **DataFrame** using the **show()** method.

```
#Defining a Schema
schema = StructType([StructField('mpg', FloatType(), nullable = True),
                     StructField('cylinders', IntegerType(), nullable = True),
                     StructField('displacement', FloatType(), nullable = True),
                     StructField('horsepower', StringType(), nullable = True),
                     StructField('weight', IntegerType(), nullable = True),
                     StructField('acceleration', FloatType(), nullable = True),
                     StructField('model year', IntegerType(), nullable = True),
                     StructField('origin', IntegerType(), nullable = True),
                     StructField('car name', StringType(), nullable = True)])

#Data is from UCI Machine Learning Repository
#See the detail in https://archive.ics.uci.edu/ml/datasets/auto+mpg
file_path = 'auto-mpg.csv'
df = spark.read.csv(file_path,
                    header = True,
                    inferSchema = True,
                    nanValue = '?')
df.show(5)
```

In the example, we only showing top 5 rows.

After that, we conduct data preprocessing by checking on the missing values so that we can handle the missing values by removing them. We can select one or more columns from a DataFrame using the select() method.

We also convert the necessary data from string to integer for processing later.

```
#Check Missing Values
def check_missing(dataframe):

    return dataframe.select([count(when(isnan(c) | col(c).isNull(), c)). \
                             alias(c) for c in dataframe.columns]).show()
check_missing(df)
```

```
#Handling Missing Values
df = df.na.drop()

#convert horsepower from string to int
df = df.withColumn("horsepower", df["horsepower"].cast(IntegerType()))
df.show(5)
```

PySpark DataFrame Basics

Spark DataFrame is a distributed collection of data organized into named columns. It is similar to a relational database table or a data frame in R/Python.

- Spark DataFrames hold data in a column and row format.
- Each column represents some feature or variable.
- Each row represents an individual data point.
- They are able to input and output data from a wide variety of sources.
- We can then use these DataFrames to apply various transformations on the data.
- At the end of the transformation calls, we can either show or collect the results to display or for some final processing.

```
#Check column names
df.columns
```

```
#Display data with pandas format
df.toPandas().head()
```

```
#Check the schema
df.printSchema()
```

```
#Renaming Columns
df = df.withColumnRenamed('model year', 'model_year')
df = df.withColumnRenamed('car name', 'car_name')
df.show(3)
```

```
#Get infos from first 4 rows
for car in df.head(4):
    print(car, '\n')
```

```
#statistical summary of dataframe
df.describe().show()
```

- `describe()` represents the statistical summary of dataframe but it also uses the string variables

```
#describe with specific variables
df.describe(['mpg', 'horsepower']).show()
```

```
#describe with numerical columns
def get_num_cols(dataframe):

    num_cols = [col for col in dataframe.columns if dataframe.select(col). \
                dtypes[0][1] in ['double', 'int']]

    return num_cols
num_cols = get_num_cols(df)

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

Spark DataFrame Basic Operations

The basic operations that can be performed on a Spark **DataFrame** also include

- Filtering rows: You can filter the rows of a **DataFrame** using the **filter()** method.
- Aggregating data: You can aggregate data in a DataFrame using methods such as **groupBy()**, **agg(), count(), sum(), avg(), min(), max()**, etc.
- Sorting data: You can sort the rows of a **DataFrame** using the **sort()** method.
- Joining **DataFrames**: You can join two or more **DataFrames** using methods such as **join(), left_outer_join(), right_outer_join(), full_outer_join(),** etc.
- Grouping and Aggregating data: You can group and aggregate data in a **DataFrame** using methods such as **groupBy(), agg(), count(), sum(), avg(), min(), max(),** etc.
- Saving data: You can save the contents of a **DataFrame** to an external storage system, such as HDFS, S3, or a database table, using methods such as **write().**

Filtering & Sorting

```
#Lets get the cars with mpg more than 23
df.filter(df['mpg'] > 23).show(5)
```

```
#Multiple Conditions
df.filter((df['horsepower'] > 80) &
          (df['weight'] > 2000)).select('car_name').show(5)
```

```
#Sorting
df.filter((df['mpg'] > 25) & (df['origin'] == 2)). \
orderBy('mpg', ascending = False).show(5)
```

```
#Get the cars with 'volkswagen' in their names, and sort them by model year and horsepower
df.filter(df['car_name'].contains('volkswagen')). \
orderBy(['model_year', 'horsepower'], ascending=[False, False]).show(5)
```

```
df.filter(df['car_name'].like('%volkswagen%')).show(3)
```

Filtering with SQL

```
#Get the cars with 'toyota' in their names
df.filter("car_name like '%toyota%'").show(5)
```

```
df.filter('mpg > 22').show(5)
```

```
#Multiple Conditions
df.filter('mpg > 22 and acceleration < 15').show(5)
```

```
df.filter('horsepower == 88 and weight between 2600 and 3000') \
.select(['horsepower', 'weight', 'car_name']).show()
```

GroupBy and Aggregate Operations

First,we use **createOrReplaceTempView** method in PySpark that allows us to create a temporary view of a **DataFrame**. This temporary view can then be used to perform SQL-like operations on the data in the **DataFrame**. This can be a powerful tool for working with large datasets and performing complex operations on them.

```
#Brands
df.createOrReplaceTempView('auto_mpg')
df = df.withColumn('brand', split(df['car_name'], ' ').getItem(0)).drop('car_name')
```

This line adds a new column called brand to the **DataFrame** by using the **withColumn** method. The split function is used to split the values in the **car_name** column by whitespace, and the **getItem** method is used to extract the first item (i.e. the brand name) from the resulting array. The drop method is then used to remove the **car_name** column from the **DataFrame**. The resulting **DataFrame** is assigned back to the variable **df**.

Overall, this code snippet demonstrates how to use the **withColumn** method to add a new column to a PySpark **DataFrame** based on the values in an existing column. It also shows how to drop a column from a DataFrame using the drop method. Finally, it shows how to create a temporary view from a PySpark DataFrame using the **createOrReplaceTempView** method.

```
#Replacing Misspelled Brands
auto_misspelled = {'chevroelt': 'chevrolet',
                   'chevy': 'chevrolet',
                   'vokswagen': 'volkswagen',
                   'vw': 'volkswagen',
                   'hi': 'harvester',
                   'maxda': 'mazda',
                   'toyouta': 'toyota',
                   'mercedes-benz': 'mercedes'}
for key in auto_misspelled.keys():
    df = df.withColumn('brand', regexp_replace('brand', key, auto_misspelled[key]))
df.show(5)
```

We define a dictionary **auto_misspelled** that maps misspelled brand names to their correct versions. Then, we use a for loop to iterate over each key-value pair in the **auto_misspelled** dictionary. For each pair, we use the **withColumn** method and the **regexp_replace** function to replace any occurrences of the misspelled brand name in the brand column with its correct version. Finally, we use the show method to display the first 5 rows of the **DataFrame**.

GroupBy and **aggregate** operations in PySpark allow you to group data by one or more columns and perform aggregate calculations on the grouped data. This can be useful for summarizing data and generating statistics or insights from large datasets. Here's an example of how to use **GroupBy** and **aggregate** operations in PySpark.

```
#Avg Acceleration by car brands
df.groupBy('brand').agg({'acceleration': 'mean'}).show(5)
```

```
#Max MPG by car brands
df.groupBy('brand').agg({'mpg': 'max'}).show(5)
```

Machine Learning

- Machine learning is a method of data analysis that automates analytical model building.
- Using algorithms that iteratively learn from data, machine learning allows computers to find hidden insights without being explicitly programmed where to look.

Supervised Learning

- Spark's MLlib is mainly designed for **Supervised** and **Unsupervised Learning** tasks, with most of its algorithms falling under those two categories.
- Supervised learning algorithms are trained using labeled examples, such as an input where the desired output is known.
- For example, a piece of equipment could have data points labeled either "F" (failed) or "R" (runs).
- The learning algorithm receives a set of inputs along with the corresponding correct outputs, and the algorithm learns by comparing its actual output with correct outputs to find errors.
- It then modifies the model accordingly.
- Through methods like classification, regression, prediction and gradient boosting, supervised learning uses patterns to predict the values of the label on additional unlabeled data.
- Supervised learning is commonly used in applications where historical data predicts likely future events.
- For example, it can anticipate when credit card transactions are likely to be fraudulent or which insurance customer is likely to file a claim.
- Or it can attempt to predict the price of a house based on different features for houses for which we have historical price data.

Unsupervised Learning

- Unsupervised learning is used against data that has no historical labels.
- The system is not told the "right answer." The algorithm must figure out what is being shown.
- The goal is to explore the data and find some structure within.
- For example, it can find the main attributes that separate customer segments from each other.
- Popular techniques include self-organizing maps, nearest-neighbor mapping, k-means clustering and singular value decomposition.
- One issue is that it can be difficult to evaluate results of an unsupervised model!

Machine Learning with PySpark

- Spark has its own MLlib for Machine Learning.
- The future of MLlib utilizes the Spark 2.0 DataFrame syntax.
- One of the main "quirks" of using MLlib is that you need to format your data so that eventually it just has one or two columns:
    - Features, Labels (Supervised)
    - Features (Unsupervised)
- This requires a little more data processing work than some other machine learning libraries, but the big upside is that this exact same syntax works with distributed data, which is no small feat for what is going on "under the hood"!

- When working with Python and Spark with MLlib, the documentation examples are always with nicely formatted data.
- A huge part of learning MLlib is getting comfortable with the documentation!
- Being able to master the skill of finding information (not memorization) is the key to becoming a great Spark and Python developer!
- Let's jump to it now!

Learn more in here: https://spark.apache.org/mllib/

Preprocessing

Encoding Brands

```
#Check brand frequencies first
df.groupby('brand').count().orderBy('count', ascending = False).show(5)
```

In PySpark, you can use the **StringIndexer** class to convert categorical values into category indices. This is a common pre-processing step for machine learning tasks, as many machine learning algorithms require numerical input data.

First, import the **StringIndexer** class from the **pyspark.ml.feature** module. Then, we create an instance of **StringIndexer** and specify the input and output columns by **setInputCol** and **setOutputCol**. After that, we Fit the **StringIndexer** to your **DataFrame** using the fit method:

We then use the **OneHotEncoder class to one hot encode your categorical data**. One-hot encoding is a proces s by which categorical data (such as nominal data) are converted into numerical features of a dataset. This is ofte n a required preprocessing step since machine learning models require numerical data.

```
def one_hot_encoder(dataframe, col):

    indexed = StringIndexer().setInputCol(col).setOutputCol(col + '_cat'). \
    #converting categorical values into category indices
    fit(dataframe).transform(dataframe)
    ohe = OneHotEncoder().setInputCol(col + '_cat').setOutputCol(col + '_OneHotEncoded
'). \
    fit(indexed).transform(indexed)

    ohe = ohe.drop(*[col, col + '_cat'])

     return ohe

df = one_hot_encoder(df, col = 'brand')
df.show(5)
```

VectorAssembler is a PySpark API that is used to transform a set of input columns into a single vector column. It is often used to prepare data for machine learning tasks, where a single feature vector column is required as input.

```
#Vector Assembler
def vector_assembler(dataframe, indep_cols):

    assembler = VectorAssembler(inputCols = indep_cols,
                                outputCol = 'features')
    output = assembler.transform(dataframe).drop(*indep_cols)

    return output


df = vector_assembler(df, indep_cols = df.drop('mpg').columns)
df.show(5)
```

Train-Test Split

We can use the **randomSplit** method of a **DataFrame** to split it into training and testing datasets. This is a common practice in machine learning tasks to evaluate the performance of a model on unseen data.

```
train_data, test_data = df.randomSplit([0.8, 0.2])
print('Train Shape: ({}, {})'.format(train_data.count(), len(train_data.columns)))
print('Test Shape: ({}, {})'.format(test_data.count(), len(test_data.columns)))
```

In this example, we split the **DataFrame** into a training **DataFrame** with 80% of the data and a testing **DataFrame** with 20% of the data. The **randomSplit** method takes a list of split weights that sum up to 1.0. The seed parameter is optional and can be used to set a random seed for reproducibility.

Perform your machine learning task on the training dataset and evaluate the performance on the testing dataset.

Multiple Linear Regression with PySpark

In PySpark, we can perform multiple linear regression using the **LinearRegression** class in the **pyspark.ml.regression** module. Multiple linear regression is a statistical method that models the linear relationship between multiple independent variables and a dependent variable.

In this example, we create a **LinearRegression** object with the feature column name and label column name specified. **regParam** is a regularization parameter in **LinearRegression** that controls the amount of regularization applied to the model. It is used to prevent overfitting by adding a penalty term to the loss function. The larger the value of **regParam**, the stronger the regularization applied to the model.

Fit the Model

```
lr = LinearRegression(labelCol = 'mpg',
                      featuresCol = 'features',
                      regParam = 0.3) #avoid overfitting
lr = lr.fit(train_data)
```

Model Evaluation

Now, we evaluate the performance of the model on the testing dataset.

```
def evaluate_reg_model(model, test_data):

    print(model.__class__.__name__.center(70, '-'))
    model_results = model.evaluate(test_data)
    print('R2: {}'.format(model_results.r2))
    print('MSE: {}'.format(model_results.meanSquaredError))
    print('RMSE: {}'.format(model_results.rootMeanSquaredError))
    print('MAE: {}'.format(model_results.meanAbsoluteError))
    print(70*'-')
evaluate_reg_model(lr, test_data)
```

In this example, we use **model.evaluate** to compute the root mean squared error (RMSE) between the predicted values and the actual values on the testing dataset.

Note that you can also tune the hyperparameters of the multiple linear regression model, such as the regularization parameter, using cross-validation or a validation dataset.

```
#End Session
spark.stop()
```

In the above example, **spark.stop()** is used to stop the **SparkSession** after the Spark code has been executed. This will release all the resources used by the Spark application and shut down all the Spark services. It is important to call this method at the end of the application to ensure that resources are not wasted. It shuts down all the Spark services and releases all the resources used by the Spark application.