Spark Preparation

We check if we are in Google Colab. If this is the case, install all necessary packages.

To run spark in Colab, we need to first install all the dependencies in Colab environment i.e. Apache Spark 3.3.2 with hadoop 3.2, Java 8 and Findspark to locate the spark in the system. The tools installation can be carried out inside the Jupyter Notebook of the Colab. Learn more from A Must-Read Guide on How to Work with PySpark on Google Colab for Data Scientists!

```
try:
  import google.colab
  IN_COLAB = True
except:
  IN_COLAB = False
```

```
if IN_COLAB:
    !apt-get install openjdk-8-jdk-headless -qq > /dev/null
    !wget -q https://dlcdn.apache.org/spark/spark-3.3.2/spark-3.3.2-bin-hadoop3.
    !tar xf spark-3.3.2-bin-hadoop3.tgz
    !mv spark-3.3.2-bin-hadoop3 spark
    !pip install -q findspark
    import os
    os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
    os.environ["SPARK_HOME"] = "/content/spark"
```

Start a Local Cluster

Use findspark.init() to start a local cluster. If you plan to use remote cluster, skip the findspark.init() and change the cluster_url according.

```
import findspark
findspark.init()
```

```
spark_url = 'local'
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
```

```
spark = SparkSession.builder\
        .master(spark_url)\
        .appName('Spark SQL')\
        .getOrCreate()
```

Spark SQL Data Preparation

First, we read a csv file. We can provide option such as delimiter and header. We then rename the colume names to remove dot ('.') in the names.

```
path = 'bank-additional-full.csv'
```

```
df = spark.read.option("delimiter", ";").option("header", True).csv(path)
```

```
df.show(3)
```

```
df.columns
```

```
cols = [c.replace('.', '_') for c in df.columns]
cols
```

```
df = df.toDF(*cols)
```

```
df.columns
```

Check out data and schema

```
df.show(5)
```

```
df.printSchema()
```

Spark SQL seems to not perform any guess on datatype. To convert to proper data type, we cast each column to proper type using '**cast**' and replace back to the same column using '**withColumn**'.

```
df = df.withColumn('age', df.age.cast('int'))
```

```
cols = ['age', 'duration', 'campaign', 'pdays', 'previous', 'nr_employed']
for c in cols:
    df = df.withColumn(c, col(c).cast('int'))
```

```
cols = ['emp_var_rate', 'cons_price_idx', 'cons_conf_idx', 'euribor3m']
for c in cols:
    df = df.withColumn(c, col(c).cast('double'))
```

Cast and also rename the column y to label

```
df = df.withColumn('label', df.y.cast('boolean'))
```

```
df.printSchema()
```

Basic Spark SQL Commands

We can select some columns using '**select**' and select some rows using '**filter**'. Note that we can perform basic math to columns.

```
df.select(df['job'], df['education'], df['housing']).show(5)
```

```
df.select(df['age'], df['duration'], df['pdays'], df['age']*2, df['duration']+c
```

```
df.filter(df.duration < 100).select('duration').show(3)
```

```
df.filter(df['job'] == 'housemaid').select('job').show(5)
```

```
df.filter((df['age'] > 60) & (df.age <= 65)).select('age', 'marital').show(5)
```

```
df.filter("marital == 'married'").select('job', 'marital').show(5)
```

```
df.filter('age < 40 and duration > 200').select('age', 'duration', 'marital').s
```

Aggregate and Groupby Functions

We can use several built-in aggegrate functions. We can also use groupby for group operations

```
from pyspark.sql.functions import avg, min, max, countDistinct
```

```
df.select(avg('age'), min('age'), max('duration')).show()
```

Groupby function allows us to work data in groups.

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

```
df.groupby('marital', 'education').agg({'age': 'min'}).show()
```

User-Defined Function

We can create user-defined function using udf.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
```

```
def agegroup_mapping(age):
    if age < 25:
        return 'young'
    if age < 55:
        return 'adult'
    return 'senior'

to_agegroup = udf(agegroup_mapping, StringType())
```

```
df.select('age', to_agegroup('age')).show(5)
```

```
new_df = df.withColumn('agegroup', to_agegroup(df.age))
new_df.select(new_df['age'], new_df['agegroup']).show(10)
```

```
spark.stop()
```

```

```