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 3.3, 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
```

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

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

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

```
df = spark.read.option("delimiter", ";").option("header", True).csv(path)
cols = [c.replace('.', '_') for c in df.columns]
df = df.toDF(*cols)
```

```
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'))

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

```
df.show(10)
```

Split data

We split data into 80% training and 20% testing data

```
train_df, test_df = df.randomSplit([0.8,0.2])
```

```
train_df.count()
```

```
test_df.count()
```

Spark ML Pipeline

Pipeline is a serie of data transformation to transform data for training and inferring. A column can contain categorical data or numerical data:

- For categorical data, we have to convert to unique numeric value using **'StringIndexer'** and perform feature encoding with **'OneHotEncoder'**.
- For numerical data, we do not have to do anything.

Once we transform all features, we vectorize them into a single column.

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler
from pyspark.ml import Pipeline
```

We first setup a pipeline of all data transformation.

- First, we transform all selected string columns
  - use a Transformer, *StringIndexer*, to encode labels in the column to indices (stored in columnnameIndex)
  - perform one hot encoder on the index to map the label index to a binary vector

```
stages = []
```

```
categoricalAttributes = ['job', 'marital', 'education', 'default',
                         'housing', 'loan', 'contact',
                         'month', 'day_of_week', 'poutcome']
for columnName in categoricalAttributes:
    stringIndexer = StringIndexer(inputCol=columnName, outputCol=columnName+ "I
    stages.append(stringIndexer)
    oneHotEncoder = OneHotEncoder(inputCol=columnName+ "Index", outputCol=colum
    stages.append(oneHotEncoder)

categoricalCols = [s + "Vec" for s in categoricalAttributes]
```

```
numericColumns = ['age', 'campaign', 'pdays', 'previous',
           'emp_var_rate', 'cons_price_idx', 'cons_conf_idx',
                  'euribor3m', 'nr_employed']
```

```
# Combine all the feature columns into a single column in the dataframe

allFeatureCols =  numericColumns + categoricalCols
vectorAssembler = VectorAssembler(
    inputCols=allFeatureCols,
    outputCol="features")
stages.append(vectorAssembler)
```

```
stages
```

Feature Extraction Pipeline

We build 2 pipelines, feature transformation pipeline and ML pipeline. This allows us to reuse the feature extraction pipeline with several ML algorithms. **'fit'** method is called to create a model and we can use **'transform'** to actual transform or infer data

```
# Build pipeline for feature extraction

featurePipeline = Pipeline(stages=stages)
featureOnlyModel = featurePipeline.fit(train_df)
```

When we 'fit' a pipeline to the data, we have the model. As we put only 'Transfomer' in the pipeline, the model is for feature extraction only.

We apply our feature extraction model with 'transform' operation to our training and testing data to create new DataFrames with 'features' column that can be used in the next pipeline.

```
trainingFeaturesDf = featureOnlyModel.transform(train_df)
testFeaturesDf = featureOnlyModel.transform(test_df)
```

trainingFeaturesDF and testFeaturesDF are training and testing DataFrames with feature columns

```
trainingFeaturesDf.show(1)
```

```
set(trainingFeaturesDf.columns) - set(train_df.columns)
```

```
# Peek into training features

trainingFeaturesDf.select("features", "label").rdd.take(5)
```

Machine Learning Pipeline

Spark ML supports several standard ML algorithm. In this example, we demonstrate how to use logistic regression and decision tree models.

```
from pyspark.ml.classification import LogisticRegression, DecisionTreeClassifie
```

Calculate accuracy

```
def calculateAccuracy(results):
    correct = results.filter(results['label'] == results['prediction']).count()
    total = results.count()
    return 1.0*correct/total
```

Logistic Regression Model

Configure an machine learning pipeline, which consists of only one stage containing an estimator (classification) (Logistic regression in this case)

```
lr = LogisticRegression(maxIter=10, regParam=0.01)
lrPipeline = Pipeline(stages=[lr])
```

Fit the pipeline to create a model from the training data. The logistic regression estimator looks for column 'features' and 'labels' to create an ML model.

```
lrPipelineModel = lrPipeline.fit(trainingFeaturesDf)
```

With our trained model, we transform testFeaturesDf to predict the results. The predicted results are stored in 'prediciton' column. We then use our calculateAccuracy function to calculate the results.

```
results = lrPipelineModel.transform(testFeaturesDf)
print('LogisticRegression Model test accuracy = ', calculateAccuracy(results))
```

```
results.select('label', 'prediction').rdd.take(5)
```

DecisionTree Model

Once again a ML pipeline is created with only an estimator in the pipeline. We then fit the pipeline with the trainingFeaturesDf to train a model. Then, we transform testFeaturesDf to predict the results.

```
dt = DecisionTreeClassifier(labelCol='label', featuresCol='features')
dtPipeline = Pipeline(stages=[dt])
```

```
dtPipelineModel = dtPipeline.fit(trainingFeaturesDf)
```

```
results = dtPipelineModel.transform(testFeaturesDf)
print('DecisionTree Model test accuracy = ', calculateAccuracy(results))
```

```
testFeaturesDf.columns
```

```
results.columns
```