import re

import itertools

import multiprocessing

from pyspark import StorageLevel

from pyspark.sql import functions as F

from pyspark.sql import SparkSession, types

from pyspark.sql.functions import col, isnan, when, count, udf

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

from pyspark.ml import Pipeline

from pyspark.ml.regression import GeneralizedLinearRegression

from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.ml.classification import LogisticRegression, LogisticRegressionModel

train\_df = spark.read.parquet('s3://myawsbucketfolder/train.parquet', header=True, inferSchema=True)

label\_df = spark.read.csv('s3://myawsbucketfolder/train\_labels.csv', header=True, inferSchema=True)

def add\_suffix(names, suffix):

         return [name + suffix for name in names]

***# Known Columns***

target\_df = ['target']

info\_df = ['customer\_ID', 'S\_2']

cat\_df = ['B\_30', 'B\_38','D\_114', 'D\_116', 'D\_117', 'D\_120', 'D\_126','D\_63','D\_64','D\_66','D\_68']

***# Define Numeric Columns***

 excluded = info\_df + cat\_df

 num\_df = [col for col **in** train\_df.columns if col **not** **in** excluded]

***# Define Feature Columns***

features = cat\_df + num\_df

train\_df = (train\_df.fillna(0, subset=num\_df)

...                     .fillna("null", subset=cat\_df))

***# Create columns aliases***

catindex = add\_suffix(cat\_df, "\_index")

***# Fit StringIndexer***

indexers = StringIndexer(inputCols=cat\_df, outputCols= catindex)

indexers\_model = indexers.fit(train\_df)

***# Transform to data***

train\_df\_indexed = indexers\_model.transform(train\_df)

***# Create columns aliases***

catVector = add\_suffix(cat\_df, "\_ohe")

***# Fit OneHotEncoder***

ohe = OneHotEncoder(inputCols=cat\_index\_cols, outputCols=cat\_ohe\_cols)

ohe\_model = ohe.fit(train\_df\_indexed)

***# Transform to data***

train\_df\_ohed = ohe\_model.transform(train\_df\_indexed)

***# Functions for each type***

***# each tuple consist of: (function, column's suffix)***

>>> num\_funcs = [

          (F.mean, "\_mean"),

          (F.stddev, "\_std"),

          (F.min, "\_min"),

          (F.max, "\_max"),

]

>>> cat\_funcs = [

          (F.count, "\_count"),

          (F.last, "\_last"),

          (F.countDistinct, "\_nunique"),

]

***# Arguments for .agg method***

***# each arg consist of: func(colname).alias(colname + suffix)***

agg\_num\_args = [

...     func(col).alias(col + suffix)

...     for col, (func, suffix) in itertools.product(num\_cols, num\_funcs)]

agg\_cols\_args = [

...     func(col).alias(col + suffix)

...     for col, (func, suffix) in itertools.product(cat\_ohe\_cols, cat\_funcs)]

***# Combine numeric and categoric agg arguments***

agg\_args = agg\_num\_args + agg\_cols\_args

agg\_args[0]

unused\_cols = cat\_cols + num\_cols + cat\_index\_cols + cat\_ohe\_cols

# Apply the agg while also dropping unused columns

train\_df\_grouped = (train\_df\_ohed.groupBy("customer\_ID")

...                                  .agg(\*agg\_cols\_args)

...                                  .drop(\*unused\_cols))

train\_df = train\_df.join(F.broadcast(label\_df), on="customer\_ID")

va = VectorAssembler(

...     inputCols=train\_joined\_df.drop("customer\_ID", "target").columns,

...     outputCol="features",

...     handleInvalid="error",

... )

train\_ready\_df = (va.transform(train\_joined\_df)

...                     .select(["customer\_ID", "features", "target"])

...                     .persist(StorageLevel.DISK\_ONLY))

**# Create a label. =1 if default, =0 otherwise**

train\_ready\_df= train\_ready\_df.withColumn("label", when(train\_ready\_df.target == 1, 1).otherwise(0))

**# Split the data into training and test sets**

trainingData, testData = train\_ready\_df.randomSplit([0.7, 0.3])

**# Create a LogisticRegression Estimator**

lr = LogisticRegression()

**# Create the pipeline logistic regression (lr) is stage 0**

  pipeline = Pipeline(stages=[lr])

**# Create a grid to hold hyperparameters**

  grid = ParamGridBuilder()

  grid = grid.addGrid(lr.regParam, [0.0, 0.2, 0.4, 0.6, 0.8, 1.0])

  grid = grid.addGrid(lr.elasticNetParam, [0, 0.5, 1])

**# Build the parameter grid**

  grid = grid.build()

**# How many models to be tested**

  print('Number of models to be tested: ', len(grid))

  Number of models to be tested:  18

**# Create a BinaryClassificationEvaluator to evaluate how well the model works**

  evaluator = BinaryClassificationEvaluator(metricName="areaUnderROC")

**# Create the CrossValidator using the hyperparameter grid**

  cv = CrossValidator(estimator=pipeline,

  ... estimatorParamMaps=grid,

  ... evaluator=evaluator,

  ... numFolds=3)

**# Train the models**

  cv = cv.fit(trainingData)

**# Test the predictions**

  predictions = cv.transform(testData)

**# Calculate AUC**

  auc = evaluator.evaluate(predictions)

  print('AUC:', auc)

**AUC: 0.8762406874932296**

**# Look at the parameters for the best model that was evaluated from the grid**

  parammap = cv.bestModel.stages[0].extractParamMap()

  for p, v in parammap.items():

  ...     print(p, v)

  LogisticRegression\_c8b92d598a08\_\_aggregationDepth 2

  LogisticRegression\_c8b92d598a08\_\_elasticNetParam 0.5

  LogisticRegression\_c8b92d598a08\_\_family auto

  LogisticRegression\_c8b92d598a08\_\_featuresCol features

  LogisticRegression\_c8b92d598a08\_\_fitIntercept True

  LogisticRegression\_c8b92d598a08\_\_labelCol label

  LogisticRegression\_c8b92d598a08\_\_maxBlockSizeInMB 0.0

  LogisticRegression\_c8b92d598a08\_\_maxIter 10

  LogisticRegression\_c8b92d598a08\_\_predictionCol prediction

  LogisticRegression\_c8b92d598a08\_\_probabilityCol probability

  LogisticRegression\_c8b92d598a08\_\_rawPredictionCol rawPrediction

  LogisticRegression\_c8b92d598a08\_\_regParam 0.0

  LogisticRegression\_c8b92d598a08\_\_standardization True

  LogisticRegression\_c8b92d598a08\_\_threshold 0.5

  LogisticRegression\_c8b92d598a08\_\_tol 1e-06

**# Look at the parameters for the best model that was evaluated from the grid**

  parammap = cv.bestModel.stages[0].extractParamMap()

  for p, v in parammap.items():

  ...     print(p, v)