**Import packages + Common Functions Definition**

%pyspark

# Ava FP Novel

sc.\_jsc.hadoopConfiguration().set("fs.s3a.endpoint", "es-si-s3-z2.eecloud.nsn-net.net")

sc.\_jsc.hadoopConfiguration().set("fs.s3a.access.key", "9LHONDBX17WLEA8SCG1S")

sc.\_jsc.hadoopConfiguration().set("fs.s3a.secret.key", "mE39w2rXZkfNCMDDYHqslRQlBDZM9lSNp82HMvZ5")

#IMPORT LIBRARIES

import sys

import \_\_builtin\_\_ as builtins

import time

import datetime

import json

import re, itertools

from pyspark.ml.linalg import Vectors

from pyspark.sql.functions import \*

from \_\_future\_\_ import print\_function

from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

from pyspark.ml.feature import StringIndexer, VectorIndexer, IndexToString, StringIndexerModel,VectorAssembler

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

from pyspark.ml.classification import DecisionTreeClassifier, DecisionTreeClassificationModel

from pyspark.mllib.evaluation import MulticlassMetrics

from pyspark.mllib.util import MLUtils

from pyspark.storagelevel import StorageLevel

from pyspark.sql.types import \*

from datetime import datetime

from pyspark.ml.classification import DecisionTreeClassifier, DecisionTreeClassificationModel

from pyspark.ml.feature import VectorAssembler, StringIndexer, IndexToString, StringIndexerModel

from pyspark.sql import HiveContext

sqlCtx = HiveContext(sc)

#FUNCTION PARSING CONFIG JSON

def read\_conf\_value(path\_conf, conf\_value):

return sqlCtx.read.json(path\_conf).collect()[0][conf\_value]

def getmaxproba(probability):

#from builtins import max, round

return builtins.round(float(builtins.max(probability)),3)

**Function to Generate the ML Model**

%pyspark

# Function to generate the ML model

def generate\_ml\_model(kpi\_name,input\_path\_train\_parquet\_file,pref\_train\_parquet\_file, input\_path\_test\_parquet\_file,pref\_test\_parquet\_file,save\_folder\_path,suffixe\_save\_folder,trees\_number\_rf,depth\_rf, n\_RF\_features\_to\_retain, cross\_val\_num\_folds,target\_col):

path = save\_folder\_path+kpi\_name+suffixe\_save\_folder+ "/batch1/cassandra\_based/"

# 1. DATA PREPARATION

print("import from parquet",kpi\_name)

TrainData = sqlCtx.read.parquet(input\_path\_train\_parquet\_file + pref\_train\_parquet\_file + kpi\_name)

TestData = sqlCtx.read.parquet(input\_path\_test\_parquet\_file + pref\_test\_parquet\_file + kpi\_name)

TrainData.groupby("actual\_class").count().show()

TestData.groupby("actual\_class").count().show()

#COLUMNS CALCULATION

print("columns calculation")

predictors\_list = [str(l.lower()).replace('.','\_') for l in TrainData.rdd.map(lambda line: sorted(line.predictedkpicollist.keys())).first()]

cassandra\_idcol = [column.lower() for column in TrainData.columns if column != "predictedkpicollist"]

cassandra\_col = cassandra\_idcol + predictors\_list

#TRAIN AND TEST SET

print("input train")

inputTrain = TrainData.select(cassandra\_idcol+["predictedkpicollist"]).rdd.map(lambda line:list(line[0:len(cassandra\_idcol)]) + [v for (k, v) in sorted(line.predictedkpicollist.items())])

train\_df = sqlCtx.createDataFrame(inputTrain, [str(l) for l in cassandra\_col], 0.1)

print("imput test for training df: ")

inputTest = TestData.select(cassandra\_idcol+["predictedkpicollist"]).rdd.map(lambda line: list(line[0:len(cassandra\_idcol)]) + [v for (k, v) in sorted(line.predictedkpicollist.items())])

test\_df = sqlCtx.createDataFrame(inputTest, [str(l) for l in cassandra\_col], 0.1)

#RECODING

print("recoding string indexer")

labelIndexer = StringIndexer(inputCol=target\_col, outputCol="label").fit(train\_df)

print("test delimiter")

print(path+"labelIndexer")

labelIndexer.write().overwrite().save(path+"labelIndexer")

label\_index = labelIndexer.transform(train\_df)

index\_labels\_df = label\_index.select(target\_col,"label").distinct().sort("label")

index\_labels = index\_labels\_df.collect()

labels = labelIndexer.labels

recoded\_train=label\_index.drop(target\_col).sort(unix\_timestamp("reading\_date", "yyyy-MM-dd")).sort("source\_id").select(predictors\_list + ['label'])

recoded\_test=labelIndexer.transform(test\_df)

#FEATURES LABELS

print("Features-label")

assembler = VectorAssembler(inputCols=predictors\_list, outputCol="features")

testing\_Data1 = assembler.transform(recoded\_test)

train\_Data1 = assembler.transform(recoded\_train).select(["features","label"])

#FEATURE EXTRACTION RF

print("FEATURE EXTRACTION RF")

from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(numTrees=trees\_number\_rf, maxDepth=depth\_rf, impurity="entropy",labelCol="label", seed=1234)

train\_Data1.cache()

rfmodel = rf.fit(train\_Data1)

fi = rfmodel.featureImportances

train\_Data1.unpersist()

#Top FEATURES Combination

import pandas as pd

indices\_list = fi.indices

values\_list = fi.values

features\_list = [predictors\_list[i] for i in fi.indices]

if type(n\_RF\_features\_to\_retain) != int:

predictors\_list\_rf = features\_list

else:

fi\_df = pd.DataFrame()

fi\_df['indices'] = indices\_list

fi\_df['values'] = values\_list

fi\_df['feature'] = features\_list

fi\_df = fi\_df.sort\_values('values', ascending= False)

predictors\_list\_rf=list(fi\_df.feature[:n\_RF\_features\_to\_retain])

fi\_df\_tosave = pd.DataFrame()

fi\_df\_tosave['features']=predictors\_list\_rf

fi\_df\_spark\_tosave= sqlCtx.createDataFrame(fi\_df\_tosave)

fi\_df\_spark\_tosave.coalesce(1).write.csv(path+"FeaturestoRetain/", sep=",", mode='overwrite', header='true')

#save important features

tuples\_rf = []

for i in predictors\_list\_rf:

tuples\_rf.append((i,))

#print(tuples\_rf)

#RECODING IMPORTANT FEATURES

print("RECODING NON CORRELATED AND IMPORTANT FEATURES")

assembler = VectorAssembler(inputCols=predictors\_list\_rf, outputCol="features")

assembler.write().overwrite().save(path+"vectorAssembler")

train\_Data = assembler.transform(recoded\_train.select(assembler.getInputCols()+ ['label']))

testing\_Data = assembler.transform(recoded\_test.select(cassandra\_idcol+assembler.getInputCols()+ ['label']))

# 2. DATA MINING

#MODEL DECLARATION

print("#MODEL DECLARATION")

dt = DecisionTreeClassifier(featuresCol="features", labelCol="label", impurity="entropy", seed=12345L, minInstancesPerNode=15)

paramGrid = ParamGridBuilder().addGrid(dt.maxDepth,[5,10,15]).build()

crossval = CrossValidator(estimator=dt,

estimatorParamMaps=paramGrid,

evaluator=MulticlassClassificationEvaluator(),

numFolds=cross\_val\_num\_folds,

seed=12345L)

#CROSSVALIDATION

print("persist data")

train\_Data.cache()

print("generate crossvalidated model")

cvModel = crossval.fit(train\_Data)

print("unpersist data")

train\_Data.unpersist()

cvModel.bestModel.write().overwrite().save(path+"model")

#except:

# print( kpi\_name, " has failed")

**Prediction Function**

%pyspark

# Produce Prediction

def producePrediction(kpi\_name, input\_path\_test\_parquet\_file,pref\_test\_parquet\_file,save\_folder\_path, suffixe\_save\_folder):

"""

Produce daily predictions from loaded KPI model on Cassandra input data.

"""

getproba = udf(lambda prob: builtins.round(builtins.max(prob), 2), DoubleType())

print("{} predictions starts ........".format(kpi\_name))

rawDailyDF = sqlCtx.read.parquet(input\_path\_test\_parquet\_file + pref\_test\_parquet\_file + kpi\_name).withColumnRenamed("actual\_class","actual\_class")

predList = [l.lower() for l in rawDailyDF.rdd.map(lambda line: sorted(line.predictedkpicollist.keys())).first()]

colList = ["kpi\_id", "reading\_date", "source\_id", "kpi\_value"] + [str(l) for l in predList] + ["actual\_class"]

outputList = colList[:4 - len(colList)]

#inputDaily = rawDailyDF.rdd.map(lambda line: [line.kpi\_id, line.reading\_date, line.source\_id, line.kpi\_name, line.kpi\_value] + [v for (k, v) in sorted(line.predictedkpicollist.items())] + [line.actual\_class])

inputDaily = rawDailyDF.rdd.map(lambda line: [line.kpi\_id, line.reading\_date, line.source\_id, line.kpi\_value] + [v for (k, v) in sorted(line.predictedkpicollist.items())] + [line.actual\_class])

predDF = sqlCtx.createDataFrame(inputDaily, colList, 0.1)

labelIndexer = StringIndexerModel.load("{}/batch1/cassandra\_based/labelIndexer".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

assembler = VectorAssembler.load("{}/batch1/cassandra\_based/vectorAssembler".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

cvModel = DecisionTreeClassificationModel.load("{}/batch1/cassandra\_based/model".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

pred = cvModel.transform(assembler.transform(predDF)).drop("features", "rawPrediction")

inverterLabel = IndexToString(inputCol="prediction", outputCol="predicted\_class", labels=labelIndexer.labels)

predOut = inverterLabel.transform(pred.withColumn("probability\_of\_prediction", getproba(pred.probability)).drop("probability"))

# Rules Computing

rules = sqlCtx.read.load("{}/batch1/cassandra\_based/rules\_output/temp\_rules/\*.csv".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder),

format="com.databricks.spark.csv",

header="true",

inferschema="true")

rules1 = rules.rdd.collect()

count\_rules = len(rules1)

tempList = predOut.columns + ["rules"]

predOutput = predOut.rdd.map(lambda x:prediction\_rule\_extraction(x,rules1, count\_rules)).toDF(tempList).select(outputList + ["actual\_class", "predicted\_class", "probability\_of\_prediction","rules"])

predOutput.select("reading\_date", "source\_id", "kpi\_id", "actual\_class", "predicted\_class", "probability\_of\_prediction", "rules")\

.coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').option("header","true").save("{}/batch1/cassandra\_based/predictions".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder), sep=",")

print("{} prediction completed! \n".format(kpi\_name))

### Start Computing metrics ####

#predicted\_cell\_asdegraded = predOutput.select("predicted\_class").where(predOutput.predicted\_class == "Degraded").count()

#degraded\_cells= predOutput.select("actual\_class").where(predOutput.actual\_class == "Degraded").count()

#relevant\_prediction =predOutput.where((predOutput.actual\_class == "Degraded") & (predOutput.predicted\_class == "Degraded")).count()

#vals=[str(float(relevant\_prediction)/float(predicted\_cell\_asdegraded)),str(float(relevant\_prediction)/float(degraded\_cells)),str(float(degraded\_cells)),str(float(predicted\_cell\_asdegraded)),str(float(relevant\_prediction))]

#print(vals)

#metr\_results = sqlCtx.createDataFrame(vals,["Precision","Recall","Degraded cell","Predicted cell as degraded", "Relevant prediction"])

#metr\_results.coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').option("header","true").save("{}/batch1/cassandra\_based/predictions".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder), sep=",")

#predicted\_cell\_asNondegraded = predOutput.select("predicted\_class").where(predOutput.predicted\_class == "Non-Degraded").count()

#degraded\_cells= predOutput.select("actual\_class").where(predOutput.actual\_class == "Non-Degraded").count()

#relevant\_prediction\_nondeg =predOutput.where((predOutput.actual\_class == "Non-Degraded") & (predOutput.predicted\_class == "Non-Degraded")).count() #vals=[str(float(relevant\_prediction)/float(predicted\_cell\_asNondegraded)),str(float(relevant\_predictio n)/float(degraded\_cells)),str(float(degraded\_cells)),str(float(predicted\_cell\_asNondegraded)),str(float(relevant\_prediction))]

#metr\_results = sqlCtx.createDataFrame(vals,["Precision","Recall","Non-Degraded cell","Predicted cell as Non-degraded", "Relevant prediction"])

#metr\_results.coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').option("header","true").save("{}/batch1/cassandra\_based/predictions".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder), sep=",")

**Functions to generate Rules**

%pyspark

#FUNCTION RULES EXTRACTION

def rules\_extraction(bestModelRules,col\_name\_list, path):

cModel = bestModelRules

important\_features = []

for i in range(len(col\_name\_list)):

if (len(re.findall("feature "+str(i)+" ",cModel))>1 & (col\_name\_list[i] not in important\_features)):

important\_features.append(col\_name\_list[i])

cModel = cModel.replace("feature "+str(i)+" ","line\_test\_rdd[\""+col\_name\_list[i]+"\"] ") #cModel = cModel.replace("feature "+str(i),col\_name\_list[i])

regles = cModel.split("\n")

predicted\_classes = []

rules= []

counter = 0

#regles = cModel.split("\n")

count\_rules=0

for j in range(len(regles)):

data = []

if regles[j].find("Predict")!=-1:

predict\_class = regles[j].strip()

if predict\_class not in predicted\_classes:

predicted\_classes.append(predict\_class)

n = j-1

while n != 0:

if (regles[n].find("Else")!=-1): #if find else

data.append(regles[n].strip().replace("Else", "&"))

count\_spaces = len(regles[n]) - len(regles[n].lstrip(' '))

for i in range(n-1,0,-1):

if (len(regles[i]) - len(regles[i].lstrip(' ')) == count\_spaces):

n = i-1

break

else:

data.append(regles[n].strip().replace("If", "&"))

n = n-1

if len(data)>0:

count\_rules = count\_rules +1

rule = str("rule "+str(count\_rules)+","+"\"\"\""+" ".join(reversed(data))).replace(",\"\"\"&", ",\"\"\"")+"\"\"\"" #rule = str("rule "+str(count\_rules)+","+" ".join(reversed(data))).replace(",&", ", if")

rule\_string = rule + ","+predict\_class.replace("Predict: ","")

rule\_str = rule\_string.split(",")

rules.append((rule\_str[0],rule\_str[1].replace("\"\"\"","").replace("\"]","").replace("line\_test\_rdd[\"",""),rule\_str[1],rule\_str[2]))

sc.parallelize([tuple(important\_features)]).toDF().coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').save(path+"rules\_output/important\_features")

rules\_df = sqlCtx.createDataFrame(rules, ["rule\_name","rule\_description","rule\_condition", "prediction\_class"]).repartition(3600)

return rules\_df

#FUNCTION RULE EXTRACTION FOR EACH PREDICTION

def prediction\_rule\_extraction(line\_test\_rdd, rules, count\_rules):

res = "not found"

for i in range(count\_rules):

c = str(rules[i]["rule\_condition"])

if eval(eval(c)):

res=rules[i]["rule\_name"]

break

else: continue

return ((list(line\_test\_rdd)+[res]))

#FUNCTION RANKING OF RULES

def rule\_ranking(r, clas, path):

c= r.filter(r.predicted\_class==clas)

c1 = c.rdd.map(lambda x: (1 if x.actual\_class == x.predicted\_class else 0, x.rule,1)).toDF(["well\_predicted","rule","count"]).groupBy("rule").sum()

res = c1.rdd.map(lambda x: (clas,x.rule, x["sum(well\_predicted)"],x["sum(count)"],builtins.round(float(x["sum(well\_predicted)"])/x["sum(count)"],5))).toDF(["actual\_class","rule","correct","total\_predicted\_with\_rule","precision"]).sort(desc("precision"))

res.coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').option("header","true").save(path+"rules\_output/ranking\_rules/"+clas)

# Generate Rules

def generate\_rules(kpi\_name,input\_path\_test\_parquet\_file,pref\_test\_parquet\_file,save\_folder\_path, suffixe\_save\_folder):

#DATA COLLECTION

print(kpi\_name, "import from cassandra")

TestData = sqlCtx.read.parquet(input\_path\_test\_parquet\_file + pref\_test\_parquet\_file + kpi\_name)

#TestData.groupby("class").count().show()

#COLUMNS CALCULATION

print("columns calculation")

predictors\_list = [l.lower() for l in TestData.rdd.map(lambda line: sorted(line.predictedkpicollist.keys())).first()]

cassandra\_idcol = [column.lower() for column in TestData.columns if column != "predictedkpicollist"]

cassandra\_col = cassandra\_idcol + predictors\_list

#TRAIN AND TEST SET

print("input train")

inputTest = TestData.select(cassandra\_idcol+["predictedkpicollist"]).rdd.map(lambda line:list(line[0:len(cassandra\_idcol)]) + [v for (k, v) in sorted(line.predictedkpicollist.items())])

test\_df = sqlCtx.createDataFrame(inputTest, [str(l) for l in cassandra\_col], 0.1).repartition(3600)

labelIndexer = StringIndexerModel.load("{}/batch1/cassandra\_based/labelIndexer".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

assembler = VectorAssembler.load("{}/batch1/cassandra\_based/vectorAssembler".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

recoded\_train = labelIndexer.transform(test\_df)

#assembler = VectorAssembler(inputCols=predictors\_list, outputCol="features")

cvModel = DecisionTreeClassificationModel.load("{}/batch1/cassandra\_based/model".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

pred = cvModel.transform(assembler.transform(recoded\_train))

udfGetMaxProba = udf(getmaxproba, DoubleType())

pred\_Data = pred.withColumn("probability\_of\_prediction", udfGetMaxProba("probability"))

#METRICS

print("calculate metrics")

metr\_res=[]

predictionsAndLabels = pred.select("prediction", "label").rdd

metrics = MulticlassMetrics(predictionsAndLabels)

confusion\_matrix = metrics.confusionMatrix().toArray()

#METRICS

labels = labelIndexer.labels

#RULES EXTRACTION

print("#rules extraction")

bestModelRules = cvModel.\_call\_java('toDebugString')

rules = rules\_extraction(bestModelRules,assembler.getInputCols(), "{}/batch1/cassandra\_based/".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

rules1 = rules.rdd.collect()

count\_rules = len(rules1)

rules.select(["rule\_name","rule\_description","rule\_condition", "prediction\_class"]).coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').option("header","true").save("{}/batch1/cassandra\_based/rules\_output/temp\_rules".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder), sep=",")

rules.select(["rule\_name","rule\_description","prediction\_class"]).coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').option("header","true").save("{}/batch1/cassandra\_based/rules\_output/rules".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder), sep=",")

#RULES FOR EACH PREDICTIONS

print("#rules for each prediction")

pred\_columns = pred\_Data.columns

predall = pred\_Data.rdd.map(lambda x:prediction\_rule\_extraction(x,rules1, count\_rules)).toDF(pred\_columns+["rule"])

#RULES MAPPING (Mamoutou's code)

kpi\_name\_col = re.sub(r"^(Row\(\w+=u')|('\))$",'',str(test\_df.select("kpi\_name").rdd.first()))

parsingRules = rules.select(["rule\_name","rule\_description"]).rdd.map(lambda line: [line.rule\_name, re.sub(r"""({})|(^\(|\)$)|(\(\\%\))|( [+-]?[0-9]+\.[0-9]+)|( <=| <| >=| >)|(\_\*meanlast\_\*|\_\*meanfull\_\*|\_\*meanfull\_\*|\_\*sdlast\_\*|\_\*sdlast\_e\-4\_\*|\_\*sdfull\_\*|\_\*sdfull\_\*|\_\*z1full\_\*|\_\*z1full\_\*|\_\*z1last\_\*|\_\*z1wdfull\_\*|\_\*z2full\_\*|\_\*z2last\_\*|\_\*z2wdfull\_\*|\_\*z3full\_\*|\_\*z3last\_\*|\_\*z3wdfull\_\*|\_\*z4last\_\*|\_\*z4full\_\*|\_\*z4wdfull\_\*)""".format(kpi\_name),"",line.rule\_description)]).map(lambda l: [l[0], str(list(set(re.sub(r"'|^( \()|(\) )$|(\))$","", l[1]).split(") & ("))))]).toDF(["rules","rule\_features"])

rulesMapping = parsingRules.rdd.map(lambda x: [[x.rules], eval(x.rule\_features)]).flatMap(lambda l: list(itertools.product(\*l))).toDF(["rule","feature"]).withColumn("kpi\_name", lit(kpi\_name\_col)).withColumn("reading\_date", lit(current\_timestamp())).filter(col("feature") != '')

rulesMapping.select(["reading\_date", "kpi\_name", "rule", "feature"]).coalesce(1).write.format('com.databricks.spark.csv').mode('overwrite').option("header","true").save("{}/batch1/cassandra\_based/rules\_output/mapping\_rules".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder), sep=",")

#rulesMapping.select(["reading\_date", "kpi\_name", "rule", "feature"]).write.format("org.apache.spark.sql.cassandra").mode('append').options(table="kpi\_rule\_features", keyspace=keyspace).save() #this part has to be run using next paragraph DONT DECOMMENT\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

print("Index to String")

#INDEX TO STRING

pred1 = IndexToString(inputCol="label", outputCol="class", labels= labelIndexer.labels).transform(predall)

predall\_df1 = IndexToString(inputCol="prediction", outputCol="predicted\_class", labels= labelIndexer.labels).transform(pred1)

predall\_df= predall\_df1.select([i for i in predall\_df1.columns if i not in predictors\_list])

predall\_df.cache()

# RULES RANKING

print("ranking rule")

for clas in labels:

c2 = rule\_ranking(predall\_df, clas, "{}/batch1/cassandra\_based/".format(save\_folder\_path + kpi\_name + suffixe\_save\_folder))

predall\_df.unpersist()

**Defining variables**

%pyspark

root\_path="s3a://csd-cell-site-degradation/development"

# Franklin Park

path\_conf\_files = root\_path+ "/csd-telefonica/config/data\_mining/"

#GET CONFIG VALUES

target\_col = read\_conf\_value(path\_conf\_files+"batch1TargetColumn.json", "target\_col").lower()

target\_col = "actual\_class"

# Kpi\_names refers to the list of KPis you are planning to generate their model.

kpi\_names = {'3g\_bpcr\_ps\_and\_hsdpa\_15days\_gteq2':901,'3g\_bpcr\_voice\_15days\_gteq2':902,'3g\_dpcr\_hsdpa\_15days\_gteq2':903,'3g\_dpcr\_voice\_15days\_gteq2':904,'3g\_estimated\_hsdpa\_user\_throughput\_kbps\_15days\_gteq2':905}

# input\_path\_train\_parquet\_file refers to the path where train data is uploaded

input\_path\_train\_parquet\_file=root\_path + "/csd-telefonica/data-lake/dm\_inputs/highly\_degraded/gteq2/15days/"

# pref\_train\_parquet\_file refers to the prefixe of parquet files where train data is uploaded

pref\_train\_parquet\_file = "input\_train\_tmp\_"

#input\_path\_test\_parquet\_file refers to the path where test data is uploaded

input\_path\_test\_parquet\_file=root\_path + "/csd-telefonica/data-lake/dm\_inputs/highly\_degraded/gteq2/15days/"

#pref\_test\_parquet\_file refers to the prefixe of parquet files where test data is uploaded

pref\_test\_parquet\_file = "input\_test\_tmp\_"

#save\_folder\_path refers to the path where generate ML model will be generated

save\_folder\_path = root\_path+ "/csd-telefonica/data-lake/data\_science/temp\_model/highly\_degraded/"

#suffixe\_save\_folder refers to the suffixe we are using to store

suffixe\_save\_folder ="\_1000\_Features"

#trees\_number\_rf refers to the number of trees we are using for Random Forest Model

trees\_number\_rf= 50

#depth\_rf refers to the depth we are using for Random Forest

depth\_rf= 15

#n\_RF\_features\_to\_retain refers to the number of Top combination of features will be used. two options are possible: put string value such as 'using\_all\_features' if you want to use all features or a number if you want to select only the TopN combination of features

n\_RF\_features\_to\_retain= 1000

# cross\_val\_num\_folds refers to the k\_folds we will use for cross validation

cross\_val\_num\_folds = 10

**Main Function: Generate ML Model + Prediction + Rules Extraction**

%pyspark

for k in range(len(kpi\_names.keys())):

#Generate the ML Model

generate\_ml\_model(kpi\_names.keys()[k],input\_path\_train\_parquet\_file ,pref\_train\_parquet\_file,input\_path\_test\_parquet\_file\

,pref\_test\_parquet\_file, save\_folder\_path, suffixe\_save\_folder,trees\_number\_rf, depth\_rf, n\_RF\_features\_to\_retain, cross\_val\_num\_folds,target\_col)

# Generate Rules

generate\_rules(kpi\_names.keys()[k],input\_path\_test\_parquet\_file,pref\_test\_parquet\_file,save\_folder\_path, suffixe\_save\_folder)

# Produce prediction

producePrediction(kpi\_names.keys()[k], input\_path\_test\_parquet\_file,pref\_test\_parquet\_file,save\_folder\_path, suffixe\_save\_folder)

**Populate Cassandra kpi\_rule\_features table**

%pyspark

keyspace = "ava\_ks\_350"

folder\_list = ["3g\_bpcr\_ps\_and\_hsdpa","3g\_bpcr\_voice","3g\_dpcr\_hsdpa","3g\_dpcr\_voice","3g\_estimated\_hsdpa\_user\_throughput\_kbps"]

for folder in folder\_list:

tbl = sqlCtx.read.format('com.databricks.spark.csv').option("header","true").option("inferSchema", "true").load("s3a://csd-cell-site-degradation/development/csd-telefonica/data-lake/models/"+folder+"/batch1/cassandra\_based/rules\_output/mapping\_rules", sep=",")

print folder, tbl.count()

rtbl = tbl.select(["reading\_date", "kpi\_name", "rule", "feature"])

rtbl.select(["reading\_date", "kpi\_name", "rule", "feature"]).write.format("org.apache.spark.sql.cassandra").mode('append').options(table="kpi\_rule\_features", keyspace=keyspace).save()

casstbl = sqlCtx.read.format("org.apache.spark.sql.cassandra").options(table="kpi\_rule\_features", keyspace=keyspace).load()

casstbl.select("kpi\_name").groupby("kpi\_name").count().show(20,False)

%cassandra

use ava\_ks\_350;

select \* from kpi\_rule\_features;

%cassandra

select count(\*) from ava\_ks\_350.kpi\_rule\_features;