databricks LYT-701 Build One Model

This Notebook Contains Util Functions Used in the Machine Learning Prototype Notebooks for the LTV 2.0 Epic

Part 0) Documentation, Packages, Parameters

Documentation

Epic JIRA Ticket

- https://jira.wishabi.com/browse/LYT-611 (https://jira.wishabi.com/browse/LYT-611)

Configs

```
spark.conf.set("spark.databricks.io.cache.enabled", "true")
```

Packages

```
import util.Try
import java.time.LocalDate
import org.apache.spark.sql.{DataFrame, Dataset, Row}
import org.apache.spark.sql.functions.{col, udf, lit, datediff, when}
import org.apache.spark.sql.types.{IntegerType, DoubleType, LongType,
StringType, DateType, StructType}
import org.apache.spark.ml.regression._
import org.apache.spark.ml.classification.{DecisionTreeClassificationModel,
DecisionTreeClassifier, RandomForestClassificationModel,
RandomForestClassifier, GBTClassificationModel, GBTClassifier}
import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer,
VectorIndexer, OneHotEncoderEstimator, IndexToString, PCA, StandardScaler}
import org.apache.spark.ml.{Pipeline, PipelineModel, PipelineStage, Predictor}
import org.apache.spark.ml.tuning.{CrossValidator, CrossValidatorModel,
TrainValidationSplit, TrainValidationSplitModel, ParamGridBuilder}
import org.apache.spark.ml.param.{ParamMap, Param}
import org.apache.spark.ml.evaluation.{Evaluator, RegressionEvaluator,
MulticlassClassificationEvaluator}
import org.apache.spark.ml.util.{DefaultParamsWritable, Identifiable}
import org.apache.spark.ml.param.shared.HasLabelCol
```

```
import util.Try
import java.time.LocalDate
import org.apache.spark.sql.{DataFrame, Dataset, Row}
import org.apache.spark.sql.functions.{col, udf, lit, datediff, when}
import org.apache.spark.sql.types.{IntegerType, DoubleType, LongType, StringTy
pe, DateType, StructType}
import org.apache.spark.ml.regression._
import org.apache.spark.ml.classification.{DecisionTreeClassificationModel, De
cisionTreeClassifier, RandomForestClassificationModel, RandomForestClassifier,
GBTClassificationModel, GBTClassifier}
import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer, VectorInde
xer, OneHotEncoderEstimator, IndexToString, PCA, StandardScaler}
import org.apache.spark.ml.{Pipeline, PipelineModel, PipelineStage, Predictor}
import org.apache.spark.ml.tuning.{CrossValidator, CrossValidatorModel, TrainV
alidationSplit, TrainValidationSplitModel, ParamGridBuilder}
import org.apache.spark.ml.param.{ParamMap, Param}
import org.apache.spark.ml.evaluation.{Evaluator, RegressionEvaluator, Multicl
assClassificationEvaluator}
import org.apache.spark.ml.util.{DefaultParamsWritable, Identifiable}
import org.apache.spark.ml.param.shared.HasLabelCol
```

Part 1) Create Functions

Util Functions

```
 /** vector of all numeric (int, double, long columns) in the imported df **/
def createNumericColumnNamesVector(df: DataFrame, filterCols: Array[String] =
null): Array[String] = {

  val cols = df.columns.filter(colString => Array(IntegerType, DoubleType,
LongType).contains(df.select(col(colString)).schema.head.dataType))

  if (filterCols == null)
    cols
  else
    cols.filter(colString => !filterCols.contains(colString))
}

/** vector of all non-numeric (string or date columns) in the imported df **/
def createNonNumericColumnNamesVector(df: DataFrame, filterCols: Array[String]
= null): Array[String] = {

  val cols = df.columns.filter(colString => Array(StringType,
DateType).contains(df.select(col(colString)).schema.head.dataType))

  if (filterCols == null)
    cols
  else
    cols.filter(colString => !filterCols.contains(colString))
}

/** process to turn **/
def createCategoricalFeatureEngPipeline(stringFeatureColNames: Array[String]):
Pipeline = {

  val stringFeaturesIndexedColNames = stringFeatureColNames.map(stringFeature
=> s"${stringFeature}Index")
  val stringFeaturesVectorColNames = stringFeatureColNames.map(stringFeature =>
s"${stringFeature}Vector")
  val stringFeaturesIndexed = stringFeatureColNames.map(stringFeature => new
StringIndexer().setHandleInvalid("keep").setInputCol(stringFeature).setOutputCo
l(s"${stringFeature}Index"))
  val encoder =
    new OneHotEncoderEstimator()
      .setInputCols(stringFeaturesIndexedColNames)
      .setOutputCols(stringFeaturesVectorColNames)
  val assemblerCAT = new
VectorAssembler().setInputCols(stringFeaturesVectorColNames).setOutputCol("feat
ures_CAT")
  val stagesCAT = stringFeaturesIndexed :+ encoder :+ assemblerCAT

  new Pipeline().setStages(stagesCAT)
```

```
}

def createNumericFeatureEngPipeline(numericFeatureColNames: Array[String], pca:
PCA): Pipeline = {

  require(numericFeatureColNames.nonEmpty, "numericFeatureColNames must not be
empty")

  val scalerFunc = (featureVectorColName: String) => (new
StandardScaler().setInputCol(featureVectorColName).setOutputCol(s"${featureVect
orColName}_scaled").setWithStd(true).setWithMean(true))
  val assemblerNUM = new
VectorAssembler().setInputCols(numericFeatureColNames).setOutputCol("features_n
umeric")

  val stagesNUM =
    if (pca != null)
      Array(assemblerNUM, scalerFunc("features_numeric"), new
VectorAssembler().setInputCols(Array("features_numeric_scaled")).setOutputCol("
features_NUM_temp"), pca)
    else
      Array(assemblerNUM, scalerFunc("features_numeric"), new
VectorAssembler().setInputCols(Array("features_numeric_scaled")).setOutputCol("
features_NUM"))

  new Pipeline().setStages(stagesNUM)
}

// if have param map --> input and it returns that or else it makes one for me.
- in one model the case expression fires off
def buildBestModelParamMap(
                          paramMap: ParamMap,
                          // monad - can be any 1 of those types
                          tuner: Either[CrossValidator, TrainValidationSplit],
                          dfValidCountryFiltered: DataFrame,
                          parallelism: Int,
                          pipelineML: Pipeline,
                          evaluator: Evaluator with HasLabelCol,
                          paramGrid: Array[ParamMap],
                          samplePercentageTuning: Double): ParamMap = {

  // list of 1 element - if paramMap is null other option to fire in this case
  Option(paramMap).getOrElse {
    val sampledDF = if (samplePercentageTuning == 1.0) dfValidCountryFiltered
else dfValidCountryFiltered.sample(samplePercentageTuning)

    // case dependent processing - eventually fit, get best model from CV and
```

```
from it get param map
    tuner match {
      case Left(cv) =>
cv.setParallelism(parallelism).setEstimator(pipelineML).setEvaluator(evaluator)
.setEstimatorParamMaps(paramGrid).fit(sampledDF).bestModel.asInstanceOf[Pipelin
eModel].stages.last.extractParamMap()
      case Right(tvs) =>
tvs.setParallelism(parallelism).setEstimator(pipelineML).setEvaluator(evaluator
).setEstimatorParamMaps(paramGrid).fit(sampledDF).bestModel.asInstanceOf[Pipeli
neModel].stages.last.extractParamMap()
    }
  }
}

// actual model end to end
// sampling the train
// actual model output (the if then statement)
// hyperparms and dataset --> fitted
def buildPipelineModelML(
                          bestModelParamMap: ParamMap,
                          dfValidCountryFiltered: DataFrame,
                          pipelineML: Pipeline,
                          samplePercentageFitting: Double): PipelineModel = {

  if (samplePercentageFitting == 1.0)
   // pipelineML - our feature engineering. Pipeline object fit to data
  // input best param map during the fitting - ex depth 5 param map; fit model
with that param map
  // pipelineML set of instructions - use them to fit the data
    pipelineML.fit(dfValidCountryFiltered, bestModelParamMap)
  else
    pipelineML.fit(dfValidCountryFiltered.sample(samplePercentageFitting),
bestModelParamMap)
}




// strings, numerics, creates separate pipelines and assmbles them together
def buildFeatureEngineeringPipeline(train: DataFrame, labelCol: String, idCols:
Array[String], dropCols: Array[String], pca: PCA): Pipeline = {
  // identify nonnumberic
  val stringCols = createNonNumericColumnNamesVector(train, (idCols.toSeq ++
dropCols.toSeq :+ labelCol).toArray)
  val numericCols = createNumericColumnNamesVector(train, (idCols.toSeq ++
dropCols.toSeq :+ labelCol).toArray)
```

```
  // separate pipelines created then assembled
  val stringFeaturesPipeline = createCategoricalFeatureEngPipeline(stringCols)
  val numericFeaturePipeline = createNumericFeatureEngPipeline(numericCols,
pca)
  val assembler = new VectorAssembler().setInputCols(Array("features_NUM",
"features_CAT")).setOutputCol("features")

  new Pipeline().setStages(Array(stringFeaturesPipeline,
numericFeaturePipeline, assembler))
}




sealed class LTVPipelineOutput(
                              val trainScored: DataFrame,
                              val testScored: DataFrame,
                              val paramMapReporting: ParamMap,
                              val paramMapModelBuilding: ParamMap,
                              val pipelineMLModelReporting: PipelineModel,
                              val pipelineMLModelBuilding: PipelineModel)

//
// expects a df that has columns and rows (e.g. country, hurdle) already
filtered
def buildLTVPipelineOutput(
                            train: DataFrame,
                            test: DataFrame,
                            modelBuilder: DataFrame,
                            labelCol: String,
                            idCols: Array[String],
                            dropCols: Array[String],
                            pca: PCA,
                            predictor: Predictor[_,_,_],
                            paramGrid: Array[ParamMap],
                            metricName: String,
                            evaluator: Evaluator with HasLabelCol,
                            samplePercentageTuning: Double,
                            parallelism: Int,
                            tuner: Either[CrossValidator,
TrainValidationSplit],
                            paramMapReporting: ParamMap,
                            paramMapModelBuilding: ParamMap,
                            samplePercentageFitting: Double,
                            buildOutputModel: Boolean, // how and if to build
final model - true if want it. Old
                            tuneOutputModel: Boolean): LTVPipelineOutput = {
```

```
  // all feat engineering - pre-regressor (pipe as spark object)
  val allFeaturePipeline = buildFeatureEngineeringPipeline(train, labelCol,
idCols, dropCols, pca)

  // make list as intermediary; add predictor to list as last transformation;
turn the new list into a new PipleLine object (can fit off it)
  val pipelineML = new Pipeline().setStages(allFeaturePipeline.getStages :+
predictor)

  // reporting error --> param map (not model but hyperparameters)
  // use crossvalidation to build best model alternatively return param map
that has already been generated
  val bestModelParamMapReporting = buildBestModelParamMap(paramMapReporting,
tuner, train, parallelism, pipelineML, evaluator, paramGrid,
samplePercentageTuning)

  // where the model is created (pipelinemodel object)
  val pipelineMLModelReporting =
buildPipelineModelML(bestModelParamMapReporting, train, pipelineML,
samplePercentageFitting)

  // have 2 variables --> adds prediction. Scored dataframes (what we usually
outpu before)
  val Vector(trainScored, testScored) = Vector(train, test).map(df =>
pipelineMLModelReporting.transform(df))
  //val Vector(metricTrain, metricTest) = Vector(trainScored,
testScored).map(df => evaluator.evaluate(df))


  // final-model/modelbuilder
  // generate parm map for best model and final model
  // regenerated for production models
  val (bestModelParamMapModelBuilder, pipelineMLModelBuilding) =


    // if building final
    if (buildOutputModel) {
      val paramMap =
        // if true
        if (tuneOutputModel)
          // same as bestModelParamMapReporting - just with new input dataset.
if want to recreate param map
          buildBestModelParamMap(paramMapModelBuilding, tuner, modelBuilder,
parallelism, pipelineML, evaluator, paramGrid, samplePercentageTuning)

        // takes param map we already generated above
```

```
        else
          bestModelParamMapReporting

      // build new model - option fo which param map I want to use new or old
      // returning tuple for  "val (bestModelParamMapModelBuilder,
pipelineMLModelBuilding)""
      (paramMap, buildPipelineModelML(paramMap, modelBuilder, pipelineML,
samplePercentageFitting))
    }

    // if not building final
    else
      (null, null)

  // instantiate with all vars creted
  new LTVPipelineOutput(trainScored, testScored, bestModelParamMapReporting,
bestModelParamMapModelBuilder, pipelineMLModelReporting,
pipelineMLModelBuilding)
}

createNumericColumnNamesVector: (df: org.apache.spark.sql.DataFrame, filterCol
s: Array[String])Array[String]
createNonNumericColumnNamesVector: (df: org.apache.spark.sql.DataFrame, filter
Cols: Array[String])Array[String]
createCategoricalFeatureEngPipeline: (stringFeatureColNames: Array[String])or
g.apache.spark.ml.Pipeline
createNumericFeatureEngPipeline: (numericFeatureColNames: Array[String], pca:
org.apache.spark.ml.feature.PCA)org.apache.spark.ml.Pipeline
buildBestModelParamMap: (paramMap: org.apache.spark.ml.param.ParamMap, tuner:
Either[org.apache.spark.ml.tuning.CrossValidator,org.apache.spark.ml.tuning.Tr
ainValidationSplit], dfValidCountryFiltered: org.apache.spark.sql.DataFrame, p
arallelism: Int, pipelineML: org.apache.spark.ml.Pipeline, evaluator: org.apac
he.spark.ml.evaluation.Evaluator with org.apache.spark.ml.param.shared.HasLabe
lCol, paramGrid: Array[org.apache.spark.ml.param.ParamMap], samplePercentageTu
ning: Double)org.apache.spark.ml.param.ParamMap
buildPipelineModelML: (bestModelParamMap: org.apache.spark.ml.param.ParamMap,
dfValidCountryFiltered: org.apache.spark.sql.DataFrame, pipelineML: org.apach
e.spark.ml.Pipeline, samplePercentageFitting: Double)org.apache.spark.ml.Pipel
ineModel
buildFeatureEngineeringPipeline: (train: org.apache.spark.sql.DataFrame, label
Col: String, idCols: Array[String], dropCols: Array[String], pca: org.apache.s
park.ml.feature.PCA)org.apache.spark.ml.Pipeline
defined class LTVPipelineOutput
buildLTVPipelineOutput: (train: org.apache.spark.sql.DataFrame, test: org.apac
he.spark.sql.DataFrame, modelBuilder: org.apache.spark.sql.DataFrame, labelCo
l: String, idCols: Array[String], dropCols: Array[String], pca: org.apache.spa
rk.ml.feature.PCA, predictor: org.apache.spark.ml.Predictor[_, _, _], paramGri
```

```
d: Array[org.apache.spark.ml.param.ParamMap], metricName: String, evaluator: o
rg.apache.spark.ml.evaluation.Evaluator with org.apache.spark.ml.param.shared.
HasLabelCol, samplePercentageTuning: Double, parallelism: Int, tuner: Either[o
rg.apache.spark.ml.tuning.CrossValidator,org.apache.spark.ml.tuning.TrainValid
ationSplit], paramMapReporting: org.apache.spark.ml.param.ParamMap, paramMapMo
delBuilding: org.apache.spark.ml.param.ParamMap, samplePercentageFitting: Doub
le, buildOutputModel: Boolean, tuneOutputModel: Boolean)LTVPipelineOutput
```

```
val pathData = "s3a://flipp-datalake-development/public/radu/ltv/2019-09-
12_model_one"
val pathReporting = s"${pathData}/reporting_error"
val pathModelBuilder = s"${pathData}/model_builder"
val df =
  spark
    .read
    .format("delta")
    .load(pathReporting)

/**** params ****/
val idCols = Array("user_id", "group", "birth_date", "birth_date_plus_horizon",
"window_end_date", "channel", "media_source", "media_source_clean",
"channel_clean", "campaign", "adset_name", "ad")
val Vector(train, test) =
  Vector("train", "test")
    .map(group => df.filter(col("group") === lit(group)).drop("group"))

val modelBuilder =
spark.read.format("delta").load(pathModelBuilder)//.sample(0.001)
val pca: PCA = null

// regressor
val labelCol = "label"
//val dropCols = Array("days_between_window_end_2015",
"days_between_birth_2015", "media_source_clean", "channel_clean", "birth_year",
"birth_month", "birth_day_of_year")
val dropCols = Array("days_between_window_end_2015", "days_between_birth_2015",
"birth_year", "birth_month", "birth_day_of_year")
val regressor = new
DecisionTreeRegressor().setLabelCol("label").setPredictionCol("prediction").set
FeaturesCol("features")
val paramGridRegressor =
  new ParamGridBuilder()
    .addGrid(regressor.minInstancesPerNode, (1000 to 1000 by 100).toArray)
    .addGrid(regressor.maxBins, (60 to 60 by 10).toArray)
    .addGrid(regressor.maxDepth, (17 to 17 by 1).toArray)
    .build()
val predictor = regressor
val evaluatorRegressor: Evaluator with HasLabelCol = new
RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMe
tricName("rmse")
val metricNameRegressor = "rmse"
val samplePercentageTuningRegressor = 0.3
val parallelismRegressor = 1
val tunerRegressor: Either[CrossValidator, TrainValidationSplit] = Left(new
CrossValidator().setNumFolds(3))
```

```
val samplePercentageFittingRegressor = 1.0

// hard code country
val Vector(trainValidCountry, testValidCountry, modelBuilderValidCountry) =
    Vector(train, test, modelBuilder)
      .map(df => df.filter(col("country").isin("CA", "US")))

val Vector(trainUnknownCountry, testUnknownCountry, modelBuilderUnknownCountry)
=
    Vector(train, test, modelBuilder)
      .map(df => df.filter(!col("country").isin("CA",
"US")).withColumn("prediction", lit(0.0)))

val output =
  buildLTVPipelineOutput(
    train = trainValidCountry,
    test = testValidCountry,
    modelBuilder = modelBuilderValidCountry,
    labelCol = "label",
    idCols = idCols,
    dropCols = dropCols,
    pca = pca,
    predictor = regressor,
    paramGrid = paramGridRegressor,
    metricName = "rmse",
    evaluator = evaluatorRegressor,
    samplePercentageTuning = 1.0,
    parallelism = 1,
    tuner = tunerRegressor,
    paramMapReporting = null,
    paramMapModelBuilding = null,
    samplePercentageFitting = 1.0,
    buildOutputModel = false,
    tuneOutputModel = false)


output.paramMapReporting
```

```
/*
DT, 0.2 sample, minInstancesPerNode 1000, maxBins 60, maxDepth 15, take out
apps_flyer_dim
0   8.880141586468579   11.574702489738154
10  7.520103849991049   9.904348745642762
20  6.864058520953757   9.037442523080745
30  6.521852576767447   8.076119119022403

DT, 0.2 sample, minInstancesPerNode 1000, maxBins 60, maxDepth 15, take out:
Array("days_between_window_end_2015", "days_between_birth_2015", "agg_source",
"media_source_clean", "channel_clean", "birth_year", "birth_month",
"birth_day_of_year")
1.21 hours
0   8.852789793215035   11.852129662695221
10  7.552351899175074   9.740299511154577
20  6.889430562859938   9.100480063875427
30  6.46265188074241    8.110554270477632

DT, 0.2 sample, minInstancesPerNode 1000, maxBins 60, maxDepth 15, take out:
Array("days_between_window_end_2015", "days_between_birth_2015",
"media_source_clean", "channel_clean", "birth_year", "birth_month",
"birth_day_of_year")
1.21 hours
0   8.884928145467462   11.552180595273295
10  7.581274556511702   9.863964143678377
20  6.931285380971786   9.049159129673527
30  6.522735613047001   8.131460062812007

DT, 0.3 sample, minInstancesPerNode 1000, maxBins 60, maxDepth 15, take out:
Array("days_between_window_end_2015", "days_between_birth_2015",
"media_source_clean", "channel_clean", "birth_year", "birth_month",
"birth_day_of_year")
0   8.953112785769834   11.81937472201737
10  7.529342869368072   9.864379198353369
20  6.890026792698095   8.810624583305744
30  6.484529689252659   8.336770435485779

DT, 0.05 sample, minInstancesPerNode 1000, maxBins 60, maxDepth 15, take out:
Array("days_between_window_end_2015", "days_between_birth_2015",
"media_source_clean", "channel_clean", "birth_year", "birth_month",
"birth_day_of_year")
0   8.953112785769834   11.81937472201737
10  7.529342869368072   9.864379198353369
20  6.890026792698095   8.810624583305744
30  6.484529689252659   8.336770435485779
```

```
*/


// loop and get error
val errorDF =
  Vector(0, 10, 20, 30)
    .map(daysSinceBirth => {

      val Vector(metricTrain, metricTest) =
        Vector((output.trainScored, trainUnknownCountry), (output.testScored,
testUnknownCountry))
          .map {
            case (dfScored, unknownCountryDF) =>
              evaluatorRegressor
                .evaluate(
                  dfScored
                    .filter(col("days_since_birth") ===
lit(daysSinceBirth)).select((train.columns :+ "prediction").map(col(_)): _*)
                    .union(unknownCountryDF.filter(col("days_since_birth") ===
lit(daysSinceBirth)))
                )
          }

      (daysSinceBirth, metricTrain, metricTest)
    })
    .toDF("days_since_birth", "metric_train", "metric_test")

display(errorDF)
```

```
import org.apache.spark.sql.functions._

display(
  output
    .testScored
    .select((train.columns :+ "prediction").map(col(_)): _*)
    .union(testUnknownCountry)
    .filter($"media_source_clean".isin("Facebook Ads", "googleadwords_int",
"Other"))
    .groupBy($"days_since_birth", $"media_source_clean")
    .agg(
      sum($"label").as("label"),
      sum($"prediction").as("prediction")
    )
    .withColumn("diff", $"prediction" - $"label")
    .withColumn("per_diff", $"diff" / $"label")
    .orderBy($"days_since_birth")
)
```