Airline Delay Prediction Code

```
//Requirements:Zeppelin, Spark, Scala interpreter



//Importing all the required libraries

import org.apache.spark.ml.feature.VectorIndexer

import org.apache.spark.mllib.tree.configuration.BoostingStrategy

import org.apache.spark.mllib.tree.{GradientBoostedTrees, RandomForest}

import org.apache.spark.ml.feature.StringIndexer

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.linalg.{Vector, Vectors}

import org.apache.spark.sql.{SaveMode, DataFrame, SQLContext, Row}


//Creating SparkSession to Load csv
val spark = org.apache.spark.sql.SparkSession.builder
        .master("local")
        .appName("Spark CSV Reader")
        .getOrCreate;

// Function to load the Dataset
def loadData(filePath:String, bool:String): DataFrame={
        spark.read
        .format("com.databricks.spark.csv")
        .option("header", bool) //reading the headers
        .option("mode", "DROPMALFORMED")
        .load(filePath); //.csv("csv/file/path") //spark 2.0 api

// Function to combine Flight Data
def combineFlight(Dataset1 : DataFrame, Dataset2 : DataFrame) : DataFrame={

        Dataset1.registerTempTable("flightDataset1");
        Dataset2.registerTempTable("flightDataset2");

        spark.sql("Select * from flightDataset1 UNION Select * from flightDataset2");
```

Airline Delay Prediction Code

```
}



// Function to combine Weather Data
def combineWeather(Dataset1: DataFrame, Dataset2: DataFrame, Dataset3: DataFrame):
DataFrame = {

        Dataset1.registerTempTable("weatherDataset1");
        Dataset2.registerTempTable("weatherDataset2");
        Dataset3.registerTempTable("weatherDataset3");

        val weather_data_temp = spark.sql("select * from weatherDataset1 union select * from
weatherDataset2 union select * from weatherDataset3");

        weather_data_temp.registerTempTable("combinedWeatherData");

        spark.sql("select * from combinedWeatherData where _c1=2006 OR _c1=2007");

}

//Test Dataset
// Function to combine Weather Data
def combineWeatherTest(Dataset1: DataFrame, Dataset2: DataFrame, Dataset3: DataFrame):
DataFrame = {

        Dataset1.registerTempTable("weatherDataset1");
        Dataset2.registerTempTable("weatherDataset2");
        Dataset3.registerTempTable("weatherDataset3");

        val weather_data_temp = spark.sql("select * from weatherDataset1 union select * from
weatherDataset2 union select * from weatherDataset3");

        weather_data_temp.registerTempTable("combinedWeatherData");

        spark.sql("select * from combinedWeatherData where _c1=2008");

}
```

Airline Delay Prediction Code

```
//Function to process FlightData
//Only Considering JFK,LAX and SFO airports as the origin
def processFlightData(Dataset: DataFrame): DataFrame = {

        Dataset.registerTempTable("flightData");

        spark.sql("select Year,Month, DayofMonth, DayofWeek, CAST((CRSDepTime/100) AS
INT) as HourofDay, UniqueCarrier, Origin, Dest, Distance, CASE  WHEN DepDelay>15 THEN 1
WHEN DepDelay<=15 THEN 0 END AS LATE FROM flightData where Origin='JFK' OR
Origin='LAX' OR Origin='SFO'");
}

// Function to process WeatherData

def processWeatherData(Dataset: DataFrame): DataFrame = {

        Dataset.registerTempTable("weatherDataset1");

        spark.sql("select _c0, CAST(_c1 AS INT) as _c1, CAST(_c2 AS INT) as _c2, CAST(_c3
AS INT) as _c3, CAST(_c4 AS INT) as _c4, case when _c5 between 0 and 10 then 1 when _c5
between 10 and 20 then 2 when _c5 between 20 and 30 then 3 when  _c5 between 30 and 40
then 4 when _c5 between 40 and 50 then 5 when _c5 between 50 and 60 then 6 else 7 end as
_c5, _c7, _c8 from weatherDataset1 where _c8 not in ('-DZ:01 FG:2 |FG:30 DZ:51 |FG:44
DZ:51','TWF.') and _c7 not in('0.00V', '2.00V', 'TWF.')")

}


//Combining Flight and weather data
def combineFlightWeatherData(flightDataset: DataFrame, weatherDataset: DataFrame) :
DataFrame = {

        flightDataset.registerTempTable("flightDataset");
        weatherDataset.registerTempTable("weatherDataset");

        spark.sql("select * from flightDataset inner join weatherDataset on Year=_c1 and
Month=_c2 and DayofMonth=_c3 and Origin=_c0 and HourOfDay=_c4 ");

}
```

Airline Delay Prediction Code

```
/**After selecting the desired parameter */
//Function to index categorical variables

def categoricalIndexing(Dataset: DataFrame): DataFrame = {

        val indexer = new StringIndexer()
        .setInputCol("Origin")
        .setOutputCol("OriginIndex")
        .setHandleInvalid("skip");

        val inputData1 = indexer.fit(Dataset).transform(Dataset);

        val indexer2 = new StringIndexer()
        .setInputCol("Dest")
        .setOutputCol("DestIndex")
        .setHandleInvalid("skip");

        val inputData2 = indexer2.fit(inputData1).transform(inputData1);
        indexer2.fit(inputData1).transform(inputData1);

        val indexer3 = new StringIndexer()
        .setInputCol("HourlyVisibility")
        .setOutputCol("HourlyVisibilityIndex")
        .setHandleInvalid("skip");

        val inputData3 = indexer3.fit(inputData2).transform(inputData2);

        val indexer4 = new StringIndexer()
        .setInputCol("HourlyPrecip")
        .setOutputCol("HourlyPrecipIndex")
        .setHandleInvalid("skip");

        indexer4.fit(inputData3).transform(inputData3);
}

//Converting columns to Doubles
def convertToDouble(Dataset : DataFrame): DataFrame={
```

Airline Delay Prediction Code

```
        val toDouble = udf[Double, Int]( _.toDouble);

        Dataset.withColumn("Month", toDouble(Dataset("Month")))
                .withColumn("DayOfMonth", toDouble(Dataset("DayOfMonth")))
                .withColumn("DayOfWeek", toDouble(Dataset("DayOfWeek")))
                .withColumn("HourOfDay", toDouble(Dataset("HourOfDay")))
                .withColumn("Distance", toDouble(Dataset("Distance")))
                .withColumn("OriginIndex", toDouble(Dataset("OriginIndex")))
                .withColumn("DestIndex", toDouble(Dataset("DestIndex")))
                .withColumn("HourlyWindSpeed", toDouble(Dataset("HourlyWindSpeed")))
                .withColumn("HourlyVisibilityIndex", toDouble(Dataset("HourlyVisibilityIndex")))
                .withColumn("HourlyPrecipIndex", toDouble(Dataset("HourlyPrecipIndex")))
                .withColumn("LATE", toDouble(Dataset("LATE")))
                .select("Month", "DayOfMonth", "DayOfWeek", "HourOfDay", "Distance",
"OriginIndex", "DestIndex", "HourlyWindSpeed","HourlyVisibilityIndex",
"HourlyPrecipIndex","LATE");
}

// Loading the Flight Data
val flight_2006 = loadData("/home/prateek/Downloads/2006.csv", "true");

val flight_2007 = loadData("/home/prateek/Downloads/2007.csv", "true");

val flight_2008 = loadData("/home/prateek/Downloads/2008.csv", "true");

// Load Weather Data
val weather_jfk = loadData("/home/prateek/Downloads/JFK_weather2", "false");

val weather_lax = loadData("/home/prateek/Downloads/LAX_weather2", "false");

val weather_sf = loadData("/home/prateek/Downloads/SF_weather2", "false");

//Combining Flight Data
val combinedFlightData = combineFlight(flight_2006, flight_2007);

val flightData = processFlightData(combinedFlightData);

//Combine Weather Data
val combinedWeatherData = combineWeather(weather_jfk, weather_lax, weather_sf);

val weatherData = processWeatherData(combinedWeatherData);
```

Airline Delay Prediction Code

```
//Combine flightData with weatherData
val flightWeatherData = combineFlightWeatherData(flightData, weatherData);

flightWeatherData.registerTempTable("flightWeatherData");

//Selecting the desired Parameters
val inputData = spark.sql("select Month, DayOfMonth, DayOfWeek, HourOfDay, Distance,
Origin, Dest, UniqueCarrier, LATE, _c7 as HourlyVisibility, _c5 as HourlyWindSpeed, _c8 as
HourlyPrecip from flightWeatherData where (_c7 is NOT NULL) and (_c5 is NOT NULL) and
(_c8 is NOT NULL)");

// Dealing with categorical variables
val finalInputData = categoricalIndexing(inputData);

finalInputData.registerTempTable("finalInputData");

finalInputData.registerTempTable("finalInputData");

var finalProcessedData = spark.sql("select CAST(Month as INT), CAST(DayOfMonth AS INT),
CAST(DayOfWeek AS INT), CAST(HourOfDay AS INT), CAST(Distance AS INT),
CAST(OriginIndex AS INT), CAST(DestIndex AS INT), CAST(HourlyWindSpeed AS INT),
CAST(HourlyVisibilityIndex AS INT), CAST(HourlyPrecipIndex AS INT), CAST(LATE AS INT)
from finalInputData where (Month is NOT NULL) AND (DayOfMonth is NOT NULL) AND
(DayOfWeek is NOT NULL) AND (HourOfDay is NOT NULL) AND (Distance is NOT NULL)
AND (OriginIndex is NOT NULL) AND (DestIndex is NOT NULL) AND (HourlyWindSpeed is
NOT NULL) AND (HourlyVisibilityIndex is NOT NULL) AND (HourlyPrecipIndex is NOT NULL)
AND (LATE is NOT NULL)");


// Converting training data to double value
val trainData = convertToDouble(finalProcessedData);

//Training Features from trainData

val trainCategoricalFeatures = List("Month", "DayOfMonth", "DayOfWeek", "HourOfDay",
"Distance", "OriginIndex", "DestIndex", "HourlyWindSpeed", "HourlyVisibilityIndex",
"HourlyPrecipIndex").map(trainData.columns.indexOf(_));


val trainCategoricalTarget = List("LATE").map(trainData.columns.indexOf(_));

//Creation of Labeled Points
```

Airline Delay Prediction Code

```
val trainLabeledPoints = trainData.rdd.map(r =>
LabeledPoint(r.getDouble(trainCategoricalTarget(0).toInt),
Vectors.dense(trainCategoricalFeatures.map(r.getDouble(_)).toArray)));


//Boosting Strategy for GBT
val boostingStrategy = BoostingStrategy.defaultParams("Classification");

boostingStrategy.numIterations = 20;

boostingStrategy.treeStrategy.numClasses = 2;

boostingStrategy.treeStrategy.maxDepth = 5;

boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]();

//Training the gbt model
val gbtModel = GradientBoostedTrees.train(trainLabeledPoints, boostingStrategy);


// Hyper parameters for RandomForest
import org.apache.spark.mllib.tree.RandomForest

val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val numTrees = 3
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val impurity = "gini"
val maxDepth = 4
val maxBins = 32

val RFmodel = RandomForest.trainClassifier(trainLabeledPoints, numClasses,
categoricalFeaturesInfo, numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins);

// Preparing testDataset
val flightDataTest = processFlightData(flight_2008);

val combinedWeatherDataTest = combineWeatherTest(weather_jfk, weather_lax, weather_sf);

val weatherDataTest = processWeatherData(combinedWeatherDataTest);

val flightWeatherDataTest = combineFlightWeatherData(flightDataTest, weatherDataTest);
```

Airline Delay Prediction Code

```
flightWeatherDataTest.registerTempTable("flightWeatherDataTest");

val inputDataTest = spark.sql("select Month, DayOfMonth, DayOfWeek, HourOfDay, Distance,
Origin, Dest, UniqueCarrier, _c5 as HourlyWindSpeed,_c7 as HourlyVisibility, _c8 as
HourlyPrecip, LATE from flightWeatherDataTest where (_c5 is NOT NULL) AND (_c7 is NOT
NULL) AND (_c8 is NOT NULL)");

val finalInputDataTest = categoricalIndexing(inputDataTest);

finalInputDataTest.registerTempTable("finalInputDataTest");

var finalProcessedDataTest = spark.sql("select CAST(Month as INT), CAST(DayOfMonth AS
INT), CAST(DayOfWeek AS INT), CAST(HourOfDay AS INT), CAST(Distance AS INT),
CAST(OriginIndex AS INT), CAST(DestIndex AS INT), CAST(HourlyWindSpeed AS INT),
CAST(HourlyVisibilityIndex AS INT), CAST(HourlyPrecipIndex AS INT), CAST(LATE AS INT)
from finalInputDataTest where (Month is NOT NULL) AND (DayOfMonth is NOT NULL) AND
(DayOfWeek is NOT NULL) AND (HourOfDay is NOT NULL) AND (Distance is NOT NULL)
AND (OriginIndex is NOT NULL) AND (DestIndex is NOT NULL) AND (HourlyWindSpeed is
NOT NULL) AND (HourlyVisibilityIndex is NOT NULL) AND (HourlyPrecipIndex is NOT NULL)
AND (LATE is NOT NULL)");


val testData = convertToDouble(finalProcessedDataTest);

val testCategoricalFeatures = List("Month", "DayOfMonth", "DayOfWeek", "HourOfDay",
"Distance", "OriginIndex", "DestIndex", "HourlyWindSpeed", "HourlyVisibilityIndex",
"HourlyPrecipIndex").map(testData.columns.indexOf(_));

val testCategoricalTarget = List("LATE").map(testData.columns.indexOf(_));

val testLabeledPoints = testData.rdd.map(r =>
LabeledPoint(r.getDouble(testCategoricalTarget(0).toInt),
Vectors.dense(testCategoricalFeatures.map(r.getDouble(_)).toArray)));


//TestError in GBT
val labelAndPreds = testLabeledPoints.map { point =>
  val prediction = gbtModel.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testLabeledPoints.count()
//println("Learned classification GBT model:\n" + gbtModel.toDebugString)
println("Test Error = " + testErr)
```

Airline Delay Prediction Code

```
//TestError in Random Forest
val labelAndPreds = testLabeledPoints.map { point =>
  val prediction = RFmodel.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testLabeledPoints.count()
//println("Learned classification GBT model:\n" + gbtModel.toDebugString)
println("Test Error = " + testErr)
```