Department of Computer Science

Missouri State University

CSC735 - Data Analytics

Predicting NYC Yellow Taxi Fares Using Big Data Analytics

Project Proposal

Submitted by

Section: 01 Group number: 32
Shadman Sakib ID: M03543624
Khalid Hasan ID: M03543550

Submitted on December 23, 2023

1 Abstract

Due to the increase in ride-sharing app usage in New York City, the demand for yellow taxi cabs has been on the downside. This is due to the insightful information provided to customers by the app regarding the expected fare and time to reach their destination. To compete in this aspect we wish to provide a solution to this problem by building a model that can predict the fare of yellow taxis in New York City. In our experiment, we have used Apache Spark as a cluster computer to train our regression models. We have used Linear Regression, Decision Tree, and Random Forest models to predict the fare. From our evaluation of these three models, we have found Random Forest to be the best-performing.

2 Introduction

The heart of traffic in New York City revolves around taxi rides, which play a central role in the daily commute for many New Yorkers. These rides provide valuable insights into traffic patterns, potential roadblocks, and other relevant factors. Accurately predicting the duration of a taxi journey holds significant importance, as users consistently seek precise information on the time required to travel between locations. With the increasing prevalence of app-based taxi services offered by popular providers such as Ola and Uber, maintaining competitive pricing is essential to attract users and ensure their preference for these platforms. Anticipating the duration and cost of trips is beneficial for users in effectively planning their journeys. This information also assists drivers in selecting optimal routes, thereby reducing travel time. In this research study, real-time data provided by customers at the commencement or booking of a ride was utilized to predict fare. In our experiment, we have used an open-source distributed computing system that provides a fast and general-purpose cluster-computing framework for big data processing: Apache Spark, to predict the continuous value of taxi fare based on a handful of features from our dataset. We have trained and tested three different machine learning models to predict and evaluate them. The models used are Linear Regression, Decision Tree, and Random Forest. From our evaluation, we have found Random Forest outperforms the other 2 models by a small margin.

3 Objective

We wish to propose a fare prediction based on the location of pick-up and destination and other key factors such as the rate code and distance. We will be utilizing the powerful and popular open-source big data processing framework, Apache Spark. In this research, we have developed a fare prediction model for the New York City Yellow Taxi rides. We have used a few regression techniques to predict the continuous value of price based on a handful of features we have extracted from the raw data itself.

4 Problem Statement

Due to the increasing number of ride-sharing apps these days, taxi businesses are at a disadvantage. Ride-sharing apps often offer lower prices than traditional taxis due to lower overhead costs. This makes them more attractive to cost-conscious passengers, which can lead to a decline in taxi ridership. Ride-sharing apps use surge pricing during high-demand periods, which can incentivize more drivers to be on the road during those times. In contrast, traditional taxis typically charge fixed rates regardless of demand. Ride-sharing apps have leveraged technology to enhance the overall user experience, from seamless payment processing to predictive ride recommendations. If customers were transparent as to how much they would be charged beforehand maybe taxis would be in demand too. Ensuring price predictability is crucial, as it facilitates improved cost management, mitigates instances of unfair pricing, and empowers customers with the information needed to compare prices with those of competitors.

5 Literature Review

In a comparative study, the effectiveness of gradient boosting using XGBoost was evaluated against a deep learning technique known as multi-layer perceptron (MLP) for predicting trip duration [1]. The findings revealed that XGBoost, demonstrated superior performance over the MLP model when accounting for all variables. Nonetheless, the authors observed that the MLP model could potentially be enhanced through auto-tuning, albeit with the trade-off of requiring additional time.

6 Data Set

The trip data from the New York City Taxi and Limousine Commission, which includes observations on around 1 billion taxi journeys in New York City between 2009 and 2016, is the source of all the data used in this study. The data for yellow taxi rides in January 2020 were used for the primary analyses in this study. A random subset of 640,508 observations—of which 80% are used for training and 20% for testing—was used to construct the models. The dataset consists of 19 columns from which we have decided to use only those that show a high Correlation to taxi fare. We have decided to use the trip distance, pick-up point, destination, and rate code ID for different sections within New York City.

Table

	summary	trip_distance	PULocationID	DOLocationID	RateCodeID	fare_amount
1	count	6405008	6405008	6405008	6339567	6405008
2	mean	2.929643933309735	164.73225778952968	162.6626908194338	1.0599077192495954	12.694108119770615
3	stddev	83.1591059732502	65.54373944111758	69.91260629496094	0.811843207190649	12.127295340046553
4	min	-30.62	1.0	1.0	1.0	-1238.0
5	max	210240.07	265.0	265.0	99.0	4265.0

5 rows

Figure 1: Summary of features selected

7 Methodology

To carry out the project we have followed several data analytics steps including data cleaning, data pre-processing, model definition, model tuning, and data predictions. We have used Apache Spark's computing system which provides a fast and general-purpose cluster-computing framework for big data processing. Spark's ability to scale horizontally across a cluster of machines enables handling large datasets seamlessly, making it suitable for big data machine learning projects.

7.1 Data Preprocessing

Clean and well-processed data helps in building more accurate models. Removing inconsistencies, errors, or outliers ensures that the model is trained on reliable and representative data. In this step, we have removed invalid and redundant data generated due to error.

1. We have removed rows that have a negative trip distance and fare amount.
2. We have removed any rows that have null in the RateCode ID.
3. We have removed outliers using the IQR method where any values lower than or greater than 1.5 times the interquartile range below and above the first and third quartiles respectively are removed. You can see in figure 2 how we have reduced and normalized our data to a much more linear scale after removing the outliers.
4. We have split the data into train and test data sets in the ratio of 80:20.

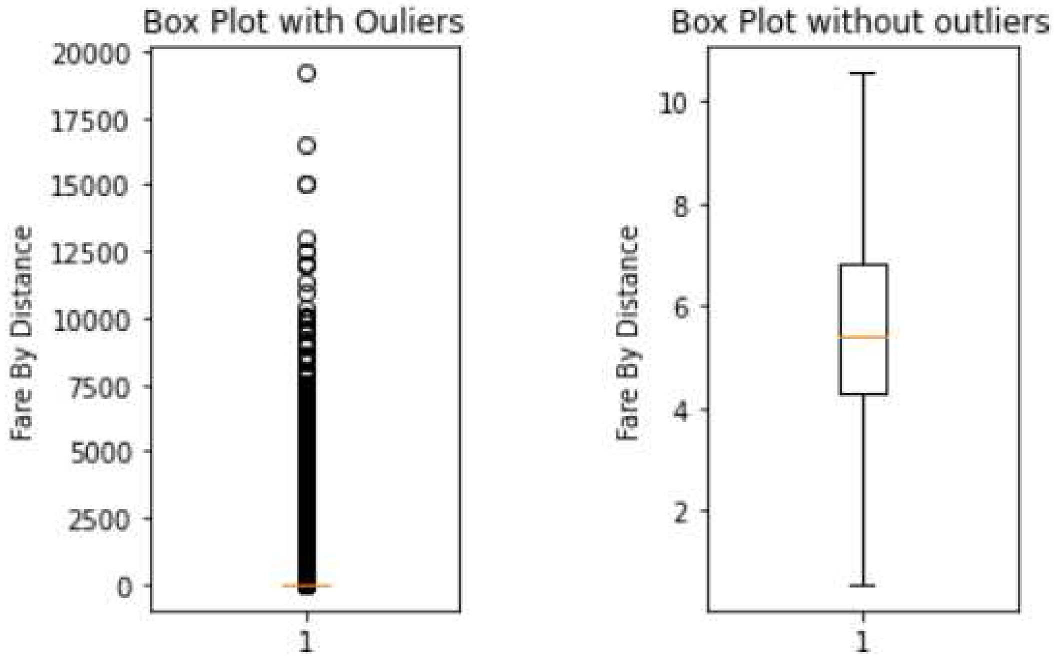

Figure 2: Outliers

7.2 Model Definition

We have applied the relevant regression models to the processed data to get the prediction. We have used 3 different models to compare and contrast between them. We have used Linear Regression, Decision Tree, and Random Forest Tree. Each model has been pushed through a pipeline to streamline and automate the process of building, training, and deploying models. Each pipeline includes three stages: an Assembler, a Standard scalar, and a Regressor as shown in figure 3.

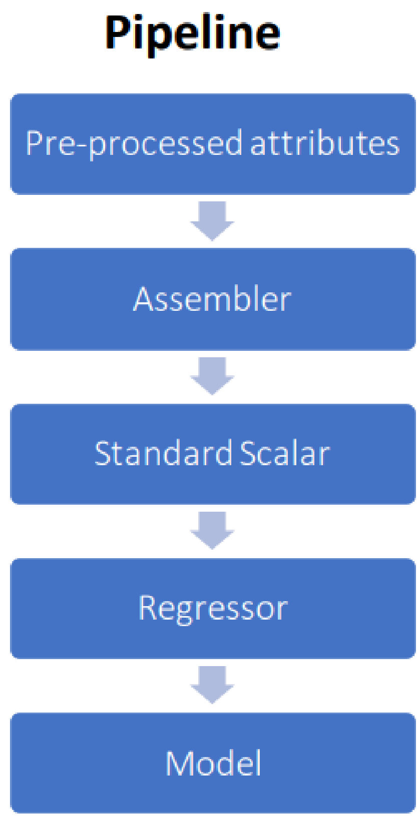

Figure 3: Pipeline

7.3 Model Tuning

To get the optimized model, we have tuned the models with different sets of hyper-parameters. Spark's ParmGridBuilder is used to construct parameter grids for hyperparameter tuning. Figure 4 demonstrates the hyperparameters we have tuned to get the best model.

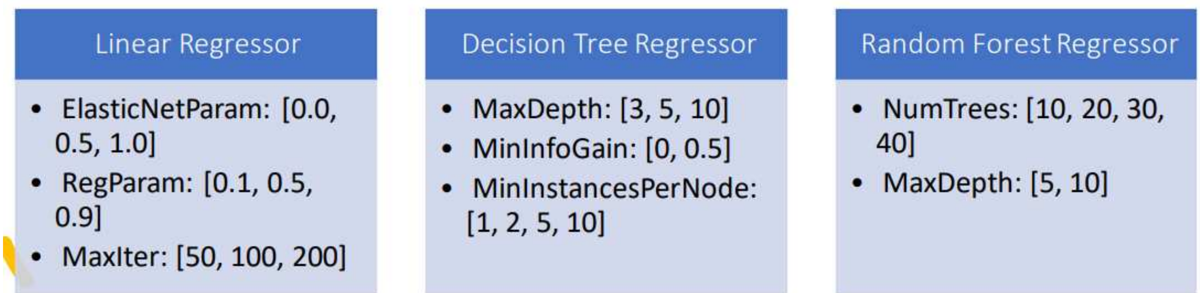

Figure 4: hyperparamters for 3 models

7.4 Model Training

Used Spark's TrainValidationSplit model to split our training dataset into a training set and a validation set, with a train ratio of 0.75. We set up one TrainValidationSplit model for each regressor with the pipeline, the parameter grid, and the evaluation process. Then we fitted the training data to the TrainValidationSplit model. After finishing the model training stage with the different combinations of hyperparameters, we find our best-trained models with optimized hyperparameters as shown in figure 5.

Model	Training Time	Parameters and hyper-parameters
Linear Regression	27.54 minutes	numIterations: 7 Co-efficients: [2.723277, 4.589777E-4, -2.8002004E-4, 0.825135] Intercept: 3.537987
Decision Tree	20.93 minutes	depth=10 numNodes=1663
Random Forest Tree	42.18 minutes	numTrees=30

Figure 5: results of training

7.5 Performance Evaluation

To evaluate the performance of our models we need proper evaluation metrics to find out the accuracy of the predictions and understand how close they are to the actual values. In our case, we have used 3 different metrics for a systematic comparison. We have used root mean squared error (RMSE), R-squared, and mean absolute error (MAE). The RMSE formula is given by:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

The R-squared formula is given by:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

The MAE formula is given by:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

8 Experiments and Results

We conducted an experimental study to evaluate our regression models, particularly the effects of corresponding features and regression techniques on the prediction of taxi fare. Our experiment consists of two analyses where each analysis visualizes the whole scenario of fare prediction based on a given test data set. These analyses are (1) Actual vs predicted fare analysis and (2) Residual analysis. We analyzed the prediction of a testing data set given by our defined three regression models: (1) Linear regression, (2) Decision tree regression, and (3) Random forest tree regression.

8.1 Actual vs Predicted Fare Analysis

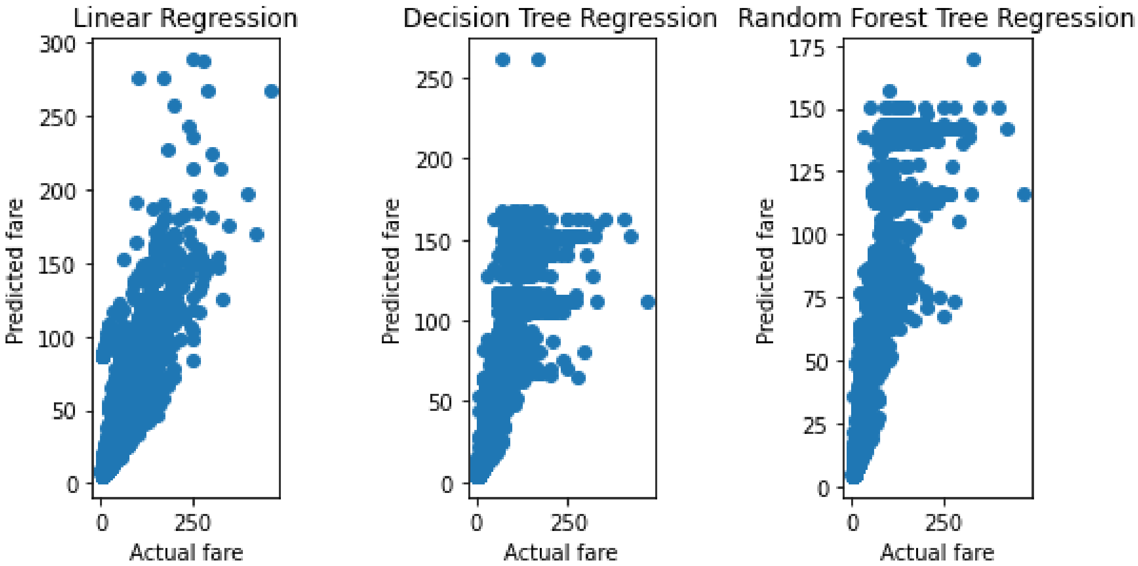

Figure 6: Actual vs Predicted Fare Analysis

In a regression analysis, an "actual vs predicted fare" analysis involves comparing the predicted values generated by a regression model to the actual observed values. This type of analysis helps evaluate how well the model performs in predicting the target variable (fare, in this case) based on input features. Each point on the scatter plot represents a data point from the test set, and the closer the points are to a diagonal line, the better the model's predictions match the actual values.

Figure 6 shows three scatter plots for each regression model where the x-axis represents the actual fares, and the y-axis represents the predicted fares. The plot for linear regression shows that the points deviate more than other models. For example, the model predicts above 250 for a considerable amount of less than 200. In contrast, we see the decision tree regression model deviates less and we get a reasonable amount of predicted value above 150 for 200 to 250 actual values. However, the random forest tree shows the most excellence. We can easily find out that this model presents a visible diagonal line and most of the points center that line which means the points do not deviate from the diagonal line. This analysis shows the random forest tree regression works better in the case of the taxi fare data set.

8.2 Residual Analysis

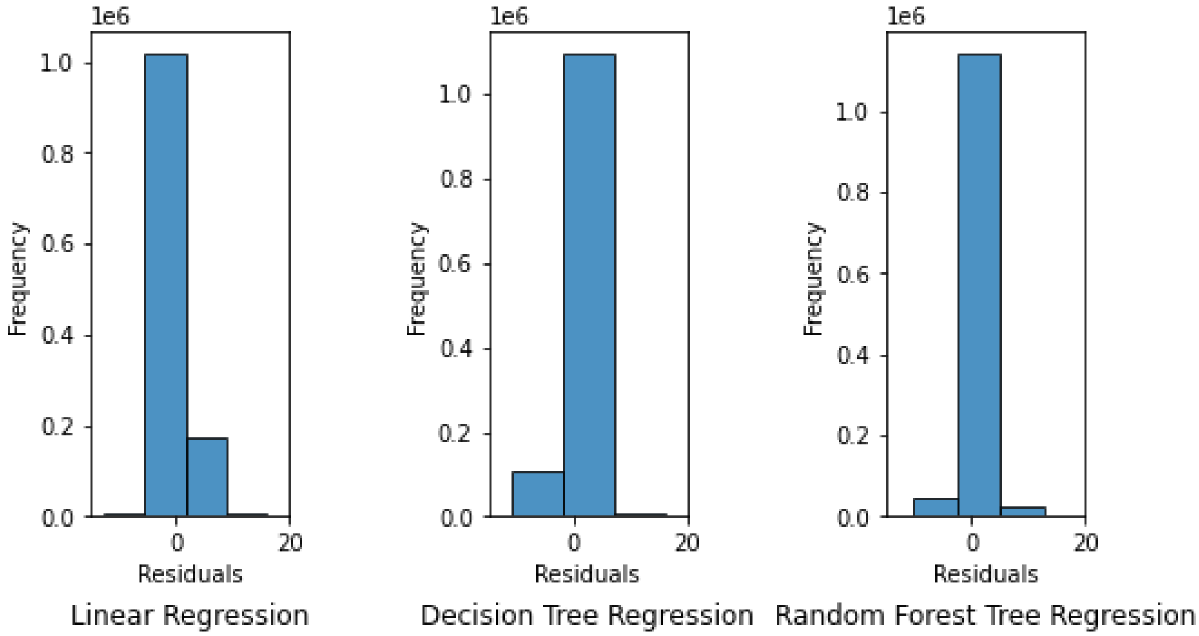

Figure 7: Residual analysis Analysis

Residual analysis is an essential step in evaluating the performance of a regression model. Residuals are the differences between the observed values (actual outcomes) and the predicted values generated by the regression model. Analyzing residuals helps to understand how well the model fits the data and identify areas for improvement. The histogram provides insights into the distribution of residuals. A normal distribution suggests that the model is performing well.

Figure 8 shows the histogram of residuals for the prediction values of each regression model. We can see four blocks in the case of the linear regression model which implies a greater standard deviation compared to other models. Furthermore, the frequency of the residuals at 0-mean is around $10\hat{6}$ whereas the decision tree model has above $10\hat{6}$ and the random forest tree regression model has a greater frequency at 0-mean with a smaller standard deviation. Moreover, the linear regression model is a bit right-skewed, the decision tree model is a bit left-skewed while the random forest model follows the normal distribution of models. These pieces of information suggest that the random forest tree regression model fits the given taxi fare data set better compared to other regression models.

9 Evaluation

Regression Model	RMSE	MAE	R2
Linear	2.898132	1.565766	0.929683
Decision Tree	2.416508	1.287255	0.951112
Random Forest Tree	2.399815	1.303674	0.951785

Figure 8: Model Evaluation

We have used the common metrics Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and R-squared (R2) which are used to evaluate the performance of regression models. RMSE is a metric that measures the average magnitude of the residuals (the differences between predicted and actual values), emphasizing larger errors. It is calculated as the square root of the mean squared error (MSE). Moreover, MAE is a metric that measures the average absolute difference between predicted and actual values, giving equal weight to all errors. It is less sensitive to outliers compared to RMSE. On top of that, R-squared is a metric that represents the proportion of the variance in the dependent variable (target) that is explained by the independent variables (features) in the model. It ranges from 0 to 1, where 0 indicates that the model does not explain any variability, and 1 indicates perfect prediction.

In the case of RMSE evaluation, the linear regression model gives more error compared to other models whereas the random forest tree model performs better. Furthermore, the decision tree regression model shows better performance in the scale of MAE followed by the random forest tree model. In addition to that, the random forest tree model has the maximum R2 score in comparison to others which has made it the best regression model. Thus, after comparing the evaluation metrics of our defined regression models, we can realize that each regression model shows excellence but the Random Forest Tree emerges as the best.

10 Conclusion

Here we have seen how to use Apache Spark and leverage its ability to handle big data with 3 different regression models. Overall we have found good results. In the future, we can compile all the recent data of these trips of the yellow taxi in New York, not just for a single month, and analyze trends and seasonality of the fare concerning the different features recorded. We can also extend the work by using neural networks or deep learning techniques.

References

[1] Poongodi, M., Malviya, M., Kumar, C. et al. New York City taxi trip duration prediction using MLP and XGBoost. Int J Syst Assur Eng Manag 13 (Suppl 1), 16–27 (2022). https://doi.org/10.1007/s13198-021-01130-x [Accessed 27/12/2022]

Appendices

A Code for the Implementation

```
// Databricks notebook source
// MAGIC %md
// MAGIC
// MAGIC ## Overview
// MAGIC
// MAGIC This notebook is generated for Data Analytics project. The project is to
    predict fare of yellow taxi in NewYork. Various regression model are used to
    build a model that is trained on the dataset received from Kaggle. The dataset
    is specically for January 2020 named yellow_tripdata_2020_01.csv. It is
    approximately 600 MB in size.
// MAGIC
// MAGIC Due to the big data size, this project has been conducted using Spark.
    Scala is mainly used as a programming language for data access and manipulating
    . Python is used particularly for plot graphs.



import org.apache.spark.sql.functions.col

// # File location and type
val file_location = "/FileStore/tables/yellow_tripdata_2020_01.csv"
val file_type = "csv"

// # CSV options
val infer_schema = "false"
val first_row_is_header = "true"
val delimiter = ","

// # The applied options are for CSV files. For other file types, these will be
    ignored.
val df = spark.read.format(file_type)
  .option("inferSchema", infer_schema)
  .option("header", first_row_is_header)
  .option("sep", delimiter)
  .load(file_location)

df.cache()

var selectedData = df.select(
  col("trip_distance").cast("double"),
```

```
  col("PULocationID").cast("double"),
  col("DOLocationID").cast("double"),
  col("RateCodeID").cast("double"),
  col("fare_amount").cast("double")
)

display(selectedData.describe())
println(selectedData.count())



// Count noisy data
println(selectedData.where(col("trip_distance")<=0 || col("fare_amount")<=0 || col
    ("RateCodeID").isNull).count)

// Removing noisy data observed from the statistics
selectedData = selectedData.selectExpr("*", "fare_amount / trip_distance as
    fareByDistance")
                          .where(col("trip_distance")>0 && col("fare_amount")>0 &&
                               col("RateCodeID").isNotNull)

// display(selectedData.orderBy('trip_distance.asc))
println(selectedData.count)

// Create a view or table
selectedData.createOrReplaceTempView("selected_data")



// MAGIC %python
// MAGIC
// MAGIC pyData = spark.sql(
// MAGIC   """
// MAGIC   select * from selected_data
// MAGIC   """
// MAGIC )



// MAGIC %python
// MAGIC
// MAGIC # Use Python to create plots
// MAGIC import matplotlib.pyplot as plt
// MAGIC
// MAGIC # Data
// MAGIC y_values = pyData.select("fareByDistance").rdd.flatMap(lambda x: x).
    collect()
// MAGIC
// MAGIC # Create a box plot with all values
// MAGIC plt.subplot(1, 2, 1)
// MAGIC plt.subplots_adjust(wspace=0.8)
// MAGIC plt.boxplot(y_values)
// MAGIC plt.title("Box Plot with Ouliers")
// MAGIC plt.ylabel("Fare By Distance")
// MAGIC
// MAGIC # Create a box plot without outliers
// MAGIC plt.subplot(1, 2, 2)
// MAGIC plt.boxplot(y_values, 0, '')
// MAGIC plt.title("Box Plot without outliers")
```

```
// MAGIC plt.ylabel("Fare By Distance")
// MAGIC
// MAGIC # Show the plot
// MAGIC plt.show()



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

def findOutliers(df: DataFrame, columns: Array[String]): DataFrame = {
  // Identifying the numerical columns in a Spark DataFrame
  // val numericColumns = df.dtypes.filter(_._2 == "Integer").map(_._1)
  val numericColumns = columns

  // Define the UDF to check if a value is an outlier
  val isOutlier = udf((value: Int, Q1: Double, Q3: Double) => {
    val IQR = Q3 - Q1
    val lowerThreshold = Q1 - 1.5 * IQR
    val upperThreshold = Q3 + 1.5 * IQR
    if (value < lowerThreshold || value > upperThreshold) 1 else 0
  })

  var updatedDF = df

  // Using the ‘for‘ loop to create new columns by identifying the outliers for each feature
  for (column <- numericColumns) {
    val Q1 = updatedDF.stat.approxQuantile(column, Array(0.25), 0)(0)
    val Q3 = updatedDF.stat.approxQuantile(column, Array(0.75), 0)(0)

    val isOutlierCol = s"is_outlier_$column"

    updatedDF = updatedDF
      .withColumn(isOutlierCol, isOutlier(col(column), lit(Q1), lit(Q3)))
  }

  // Selecting the specific columns which we have added above
  val selectedColumns = updatedDF.columns.filter(_.startsWith("is_outlier"))

  // Adding all the outlier columns into a new column "total_outliers"
  updatedDF = updatedDF.withColumn("total_outliers", selectedColumns.map(col).reduce(_ + _))

  // Dropping the extra columns created above
  updatedDF = updatedDF.drop(selectedColumns: _*)

  updatedDF
}

// Usage:
// val outliersDF = findOutliers(yourDataFrame)
// outliersDF.show()



// remove outliers
val outliersDF: DataFrame = findOutliers(selectedData, columns=Array("fareByDistance"))
```

```scala
println(outliersDF.where(col("total_outliers") > 0).count)

selectedData = outliersDF.where(col("total_outliers") === 0)
println(selectedData.count())



import org.apache.spark.ml.regression._
import org.apache.spark.ml.feature.{VectorAssembler, StandardScaler}
import org.apache.spark.sql.DataFrame
import org.apache.spark.ml.{Pipeline, PipelineModel}

// Step 4: Split the DataFrame into a training set and a test set (80% train, 20% test)
val Array(trainData, testData) = selectedData.randomSplit(Array(0.8, 0.2))

// val rFormula = new RFormula()

// Step 5: Prepare features and labels using a VectorAssembler
val assembler = new VectorAssembler()
  .setInputCols(Array("trip_distance", "PULocationID", "DOLocationID", "RateCodeID"))
  .setOutputCol("features")
  .setHandleInvalid("skip")

val sScalar = new StandardScaler().setInputCol("features")

// val regressionClasses = List("LinearRegression", "DecisionTreeRegressor", "RandomForestRegressor", "GBTRegressor")

// Step 6: Create a Linear Regression model
val lR = new LinearRegression()
  .setLabelCol("fare_amount")
  .setFeaturesCol("features")
  .setPredictionCol("prediction")

val dTR = new DecisionTreeRegressor()
  .setLabelCol("fare_amount")
  .setFeaturesCol("features")
  .setPredictionCol("prediction")

val rFR = new RandomForestRegressor()
  .setLabelCol("fare_amount")
  .setFeaturesCol("features")
  .setPredictionCol("prediction")


println(lR.explainParams())
println(dTR.explainParams())
println(rFR.explainParams())


val pipelineLR = new Pipeline().setStages(Array(assembler, sScalar, lR))
val pipelineDTR = new Pipeline().setStages(Array(assembler, sScalar, dTR))
val pipelineRFR = new Pipeline().setStages(Array(assembler, sScalar, rFR))



// specifying different combinations of hyperparameters to select the best model
```

```
    using an Evaluator, testing their predictions
import org.apache.spark.ml.tuning.ParamGridBuilder
import org.apache.spark.ml.evaluation.RegressionEvaluator

// Params for linear regressor
val paramsLR = new ParamGridBuilder()
  .addGrid(lR.elasticNetParam, Array(0.0, 0.5, 1.0))
  .addGrid(lR.regParam, Array(0.1, 0.5, 0.9))
  .addGrid(lR.maxIter, Array(50, 100, 200))
  .build()

// Params for Decision Tree regressors
val paramsDTR = new ParamGridBuilder()
  .addGrid(dTR.maxDepth, Array(3, 5, 10))
  .addGrid(dTR.minInfoGain, Array(0, 0.5))
  .addGrid(dTR.minInstancesPerNode, Array(1, 2, 5, 10))
  .build()

// Params for Random forests regressors
val paramsRFR = new ParamGridBuilder()
  .addGrid(rFR.numTrees, Array(10, 20, 30, 40))
  .addGrid(rFR.maxDepth, Array(5, 10))
  .build()



// Evaluator using RegressionEvaluator using rmse
val rmse_evaluator = new RegressionEvaluator()
  .setLabelCol("fare_amount")
  .setPredictionCol("prediction")
  .setMetricName("rmse")

// Evaluator using RegressionEvaluator using mae
val mae_evaluator = new RegressionEvaluator()
  .setLabelCol("fare_amount")
  .setPredictionCol("prediction")
  .setMetricName("mae")

// Evaluator using RegressionEvaluator using r2
val r2_evaluator = new RegressionEvaluator()
  .setLabelCol("fare_amount")
  .setPredictionCol("prediction")
  .setMetricName("r2")



// Define Train Validation Split
import org.apache.spark.ml.tuning.TrainValidationSplit

// Linear regressor
val tVS_LR = new TrainValidationSplit()
  .setTrainRatio(0.75) // also the default.
  .setEstimatorParamMaps(paramsLR).setEstimator(pipelineLR)
  .setEvaluator(rmse_evaluator)

// Decision Tree regressor
val tVS_DTR = new TrainValidationSplit()
  .setTrainRatio(0.75) // also the default.
  .setEstimatorParamMaps(paramsDTR).setEstimator(pipelineDTR)
```

```
  .setEvaluator(rmse_evaluator)

// Random Forests regressor
val tVS_RFR = new TrainValidationSplit()
  .setTrainRatio(0.75) // also the default.
  .setEstimatorParamMaps(paramsRFR).setEstimator(pipelineRFR)
  .setEvaluator(rmse_evaluator)


// Get TrainValidationSplitModel for linear regressor
val tVS_LR_Model = tVS_LR.fit(trainData)



// Get TrainValidationSplitModel for decision tree regressor
val tVS_DTR_Model = tVS_DTR.fit(trainData)



// Get TrainValidationSplitModel for random forests regressor
val tVS_RFR_Model = tVS_RFR.fit(trainData)



// Get best model and statistics
import org.apache.spark.ml.regression.{LinearRegressionModel,
    DecisionTreeRegressionModel, RandomForestRegressionModel, GBTRegressionModel}

val bestPipelineLRModel = tVS_LR_Model.bestModel.asInstanceOf[PipelineModel]
val bestLRModel = bestPipelineLRModel.stages.last.asInstanceOf[
    LinearRegressionModel]

val summary = bestLRModel.summary
println(summary)
println(s"numIterations: ${summary.totalIterations}")
println(s"Co-efficients: ${bestLRModel.coefficients} Intercept: ${bestLRModel.
    intercept}")
summary.residuals.show()
println(summary.objectiveHistory.toSeq.toDF.show())
println(summary.objectiveHistory)
println(summary.rootMeanSquaredError)
println(summary.r2)

val bestPipelineDTRModel = tVS_DTR_Model.bestModel.asInstanceOf[PipelineModel]
val bestDTRModel = bestPipelineDTRModel.stages.last.asInstanceOf[
    DecisionTreeRegressionModel]

val summaryLR = bestLRModel.summary
println(summaryLR)

val bestPipelineRFRModel = tVS_RFR_Model.bestModel.asInstanceOf[PipelineModel]
val bestRFRModel = bestPipelineRFRModel.stages.last.asInstanceOf[
    RandomForestRegressionModel]



// Step 7: Make predictions on the test data
// val testPreprocessed = assembler.transform(testData)
val predictionsLR = bestPipelineLRModel.transform(testData)
```

```scala
predictionsLR.show(false)

val predictionsDTR = bestPipelineDTRModel.transform(testData)
predictionsDTR.show()

val predictionsRFR = bestPipelineRFRModel.transform(testData)
predictionsRFR.show()



// RMSE
val rmseLR = rmse_evaluator.evaluate(predictionsLR)
println(s"Root Mean Squared Error (RMSE) for LR: $rmseLR")

val rmseDTR = rmse_evaluator.evaluate(predictionsDTR)
println(s"Root Mean Squared Error (RMSE) for DTR: $rmseDTR")

val rmseRFR = rmse_evaluator.evaluate(predictionsRFR)
println(s"Root Mean Squared Error (RMSE) for RFR: $rmseRFR")


// MAE
val maeLR = mae_evaluator.evaluate(predictionsLR)
println(s"Mean Absolute Error (MAE) for LR: $maeLR")

val maeDTR = mae_evaluator.evaluate(predictionsDTR)
println(s"Mean Absolute Error (MAE) for DTR: $maeDTR")

val maeRFR = mae_evaluator.evaluate(predictionsRFR)
println(s"Mean Absolute Error (MAE) for RFR: $maeRFR")


// R2
val r2LR = r2_evaluator.evaluate(predictionsLR)
println(s"R-squared (r2) Error for LR: $r2LR")

val r2DTR = r2_evaluator.evaluate(predictionsDTR)
println(s"R-squared (r2) Error for DTR: $r2DTR")

val r2RFR = r2_evaluator.evaluate(predictionsRFR)
println(s"R-squared (r2) Error for RFR: $r2RFR")



import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types._

val evaluatedDataMap = Seq(
  Map("Regression Model" -> "Linear", "RMSE" -> rmseLR, "MAE" -> maeLR, "R2" ->
      r2LR),
  Map("Regression Model" -> "Decision Tree", "RMSE" -> rmseDTR, "MAE" -> maeDTR, "
      R2" -> r2DTR),
  Map("Regression Model" -> "Random Forest Tree", "RMSE" -> rmseRFR, "MAE" ->
      maeRFR, "R2" -> r2RFR)
)

// Define the schema based on the keys and types of the first map
val schema = new StructType()
    .add("Regression Model",StringType)
```

```
    .add("RMSE", DoubleType)
    .add("MAE", DoubleType)
    .add("R2", DoubleType)

// Convert the sequence of maps to a sequence of Rows
val rows = evaluatedDataMap.map { rowMap =>
  Row.fromSeq(schema.map(field => rowMap.getOrElse(field.name, null)))
}

// Create a DataFrame
val evaluationMatrix = spark.createDataFrame(spark.sparkContext.parallelize(rows),
     schema)

display(evaluationMatrix.select(col("Regression Model"), round('RMSE, 6).as("RMSE
    "), round('MAE, 6).alias("MAE"), round('R2, 6).alias("R2")))



// create sql table view to access data while using python
predictionsLR.createOrReplaceTempView("predictions_lr")
predictionsDTR.createOrReplaceTempView("predictions_dtr")
predictionsRFR.createOrReplaceTempView("predictions_rfr")



// MAGIC %python
// MAGIC
// MAGIC import numpy as np
// MAGIC
// MAGIC # Create python dataframe
// MAGIC
// MAGIC predictionsLR = spark.sql(
// MAGIC     """
// MAGIC     select * from predictions_lr
// MAGIC     """
// MAGIC )
// MAGIC
// MAGIC predictionsDTR = spark.sql(
// MAGIC     """
// MAGIC     select * from predictions_dtr
// MAGIC     """
// MAGIC )
// MAGIC
// MAGIC predictionsRFR = spark.sql(
// MAGIC     """
// MAGIC     select * from predictions_rfr
// MAGIC     """
// MAGIC )
// MAGIC
// MAGIC
// MAGIC # Data Preparation of actual and predicted fares
// MAGIC models = {
// MAGIC     "lr": "Linear Regression",
// MAGIC     "dtr": "Decision Tree Regression",
// MAGIC     "rfr": "Random Forest Tree Regression"
// MAGIC }
// MAGIC
// MAGIC actual_fares = {
// MAGIC     "lr": np.array(predictionsLR.select("fare_amount").rdd.flatMap(lambda
```

```
    x: x).collect()),
// MAGIC     "dtr": np.array(predictionsDTR.select("fare_amount").rdd.flatMap(
    lambda x: x).collect()),
// MAGIC     "rfr": np.array(predictionsRFR.select("fare_amount").rdd.flatMap(
    lambda x: x).collect()),
// MAGIC }
// MAGIC predicted_fares = {
// MAGIC     "lr": np.array(predictionsLR.select("prediction").rdd.flatMap(lambda
    x: x).collect()),
// MAGIC     "dtr": np.array(predictionsDTR.select("prediction").rdd.flatMap(
    lambda x: x).collect()),
// MAGIC     "rfr": np.array(predictionsRFR.select("prediction").rdd.flatMap(
    lambda x: x).collect())
// MAGIC }
// MAGIC residuals = {
// MAGIC     "lr": actual_fares["lr"] - predicted_fares["lr"],
// MAGIC     "dtr": actual_fares["dtr"] - predicted_fares["dtr"],
// MAGIC     "rfr": actual_fares["rfr"] - predicted_fares["rfr"]
// MAGIC }



// MAGIC %python
// MAGIC # Use Python to create plots
// MAGIC import matplotlib.pyplot as plt
// MAGIC
// MAGIC # Create scatter plots
// MAGIC # Increase total size by setting figsize
// MAGIC plt.figure(figsize=(8, 4))
// MAGIC for i, k in enumerate(actual_fares.keys()):
// MAGIC     plt.subplot(1, 3, i+1)
// MAGIC     plt.subplots_adjust(wspace=1)
// MAGIC     plt.scatter(actual_fares[k], predicted_fares[k])
// MAGIC     plt.xlabel("Actual fare")
// MAGIC     plt.ylabel("Predicted fare")
// MAGIC     plt.title(models[k])
// MAGIC plt.show()
// MAGIC
// MAGIC # Create histogram of residuals
// MAGIC # Increase total size by setting figsize
// MAGIC plt.figure(figsize=(8, 4))
// MAGIC for i, k in enumerate(residuals.keys()):
// MAGIC     plt.subplot(1, 3, i+1)
// MAGIC     plt.subplots_adjust(wspace=1)
// MAGIC     plt.hist(residuals[k], bins=60, edgecolor='black', alpha=0.8)
// MAGIC     # Set x-axis range from -20 to 20
// MAGIC     plt.xlim(-15, 20)
// MAGIC     plt.xlabel('Residuals')
// MAGIC     plt.ylabel('Frequency')
// MAGIC     plt.title(models[k], y=-.25)
// MAGIC
// MAGIC plt.show()
```