Step by step guidance of project:

1. Familiarizing with Apache Flink and Understand Apache Beam which is a unified programming model for both batch and streaming data processing:
   * Going to [Apache Beam website](https://beam.apache.org/) to understand its programming model and how it can be used with Apache Flink.
2. Review the RIOBench Paper:
   * Reading the RIOBench [paper](https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.4257) to understand the benchmark streaming applications implemented in Apache Storm and the requirements for adapting them for Apache Flink.
3. Explore the RioBench GitHub Repository:
   * Visit the RioBench [GitHub repository](https://github.com/dream-lab/riot-bench) to access the code for the RioBench use case applications. Review the existing implementations and understand the logic and structure of the benchmark applications.
4. Set up Development Environment:
   * Setting up my development environment with Apache Flink and Apache Beam. Install the necessary tools and libraries required for building and running applications with Apache Flink and Apache Beam.
5. Adapt the Benchmark Applications for Flink:
   * Select a benchmark streaming application from the RioBench repository and begin adapting it for Apache Flink. This may involve making changes to the code to align with Flink’s programming model and APIs.
6. Implement the PRED for Apache Flink using Apache Beam:
   * Once we have adapted the benchmark application for Flink, start implementing the PRED (Predictive Analytics dataflow) using Apache Beam. This may involve writing new code to define the PRED and integrating it with the adapted benchmark application.
7. Test and Validate Implementation:
   * Testing ported application and the PRED implementation to ensure they function correctly with Apache Flink and Apache Beam. Validate that the applications produce the expected results and perform well in a streaming environment.
8. Document the Work:
   * Documenting porting process, the PRED implementation, and any challenges or insights.

---------------------------------------------+

PRED implementation:

The App class orchestrates a data processing pipeline designed to handle [taxi trip data](https://databank.illinois.edu/datasets/IDB-9610843). Upon execution, it reads input data from a specified CSV/ARFF file containing details of taxi trips. This input data is then split into smaller, manageable chunks for further processing. The class invokes the Decision Tree Classifier (DTC) to perform classification tasks on the processed data, aiding in tasks such as prediction or categorization based on given features. Additionally, the App class computes the block window average for the taxi trip data, which involves calculating the average total amount for a defined block size, providing insights into fare patterns over time or space.

package abolfazl.younesi;  
  
import abolfazl.younesi.beamutil.\*;  
  
import abolfazl.younesi.bolts.BlockWindowAverage;  
import abolfazl.younesi.bolts.DTC;  
import abolfazl.younesi.bolts.TaxiData;  
import org.apache.beam.sdk.Pipeline;  
import org.apache.beam.sdk.io.FileIO;  
import org.apache.beam.sdk.io.Compression;  
import org.apache.beam.sdk.io.TextIO;  
import org.apache.beam.sdk.options.PipelineOptionsFactory;  
import org.apache.beam.sdk.transforms.Contextful;  
import org.apache.beam.sdk.transforms.MapElements;  
import org.apache.beam.sdk.transforms.SerializableFunction;  
import org.apache.beam.sdk.values.PCollection;  
import org.apache.beam.sdk.values.TypeDescriptors;  
import org.apache.beam.sdk.options.Default;  
import org.apache.beam.sdk.options.Description;  
import org.apache.beam.sdk.options.StreamingOptions;  
import org.apache.beam.sdk.transforms.DoFn;  
import org.apache.beam.sdk.transforms.ParDo;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
import java.io.BufferedReader;  
import java.io.File;  
import java.io.FileReader;  
import java.io.IOException;  
import java.text.ParseException;  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
  
  
public class App {  
 private static final Logger *LOG* = LoggerFactory.*getLogger*(App.class);  
 public static String *dtc* = "F:\\utf-8-FOIL2013\\FOIL2013\\output";  
 public interface Options extends StreamingOptions {  
 @Description("Input text to print.")  
 @Default.String("My, input, text")  
 String getInputText();  
 void setInputText(String value);  
  
 @Description("Delimiter to separate input elements.")  
 @Default.String(",")  
 String getDelimiter();  
 void setDelimiter(String value);  
 }  
  
 private static PCollection<String> readInputData(Pipeline p, String inputFilePath) {  
 return p.apply("ReadData",  
 TextIO.*read*().from(inputFilePath));  
 }  
  
 private static PCollection<String> readCSVLines(Pipeline p, String csvInputFile) {  
 return p.apply("ReadCSVDataLine",  
 TextIO.*read*().from(csvInputFile));  
 }  
  
 private static PCollection<List<String>> splitIntoChunks(PCollection<String> lines) {  
 return lines.apply("SplitIntoChunks",  
 MapElements.*into*(TypeDescriptors.*lists*(TypeDescriptors.*strings*()))  
 .via((SerializableFunction<String, List<String>>) line -> {  
 assert line != null;  
 return Arrays.*asList*(line.split(","));  
 }));  
 }  
  
 private static void writeChunks(PCollection<List<String>> chunks, String outputFolder, int numberOfChunks) {  
 chunks.apply("WriteChunks", FileIO.<List<String>>*write*()  
 .via(Contextful.*fn*((List<String> chunk) -> {  
 assert chunk != null;  
 return chunk.stream().collect(Collectors.*joining*("\n"));  
 }), TextIO.*sink*())  
 .to(outputFolder)  
 .withPrefix("chunk\_")  
 .withSuffix(".csv")  
 .withNumShards(numberOfChunks));  
 }  
  
 // DoFn to invoke dtcclassify method  
 static class InvokeDTC extends DoFn<String, Void> {  
 @ProcessElement  
 public void processElement(@Element String line, OutputReceiver<Void> out) {  
 try {  
 // Call dtcclassify() method from DTC class  
 DTC.*dtcClassify*(*dtc*+"\\chunk\_1.csv",*dtc*);  
 } catch (Exception e) {  
 // Log error and continue processing  
 *LOG*.warn("Error invoking dtcclassify(): {}", e.getMessage());  
 }  
 }  
 }  
  
 public static void main(String[] args) throws IOException {  
 Pipeline p = Pipeline.*create*(PipelineOptionsFactory.*fromArgs*(args).withValidation().create());  
 System.*out*.println("Starting the pipeline...");  
 // Read input data  
// String inputFilePath = "path/to/input/data";  
// PCollection<String> inputData = readInputData(p, inputFilePath);  
  
 //-------------------------------------------  
  
 //--- Read input data  
 System.*out*.println("Reading input data...");  
 String csvInputFile = "F:\\utf-8-FOIL2013\\FOIL2013\\trip\_fare\_1\\trip\_fare\_1.csv"; //---- Replace with your CSV file path  
 String chunkOutputFolder = "F:\\utf-8-FOIL2013\\FOIL2013\\output"; //----- Replace with your output folder path  
 String arffOutputFile = "F:\\utf-8-FOIL2013\\FOIL2013\\arffoutput";  
 int numberOfChunks = 10; // Number of chunks  
  
// PCollection<String> csvInputData = readCSVLines(p, csvInputFile);  
  
 //---- Split CSV lines into chunks  
 System.*out*.println("Splitting CSV lines into chunks...");  
// PCollection<List<String>> chunks = splitIntoChunks(csvInputData);  
  
 //--- Write chunks to output  
 System.*out*.println("Writing chunks to output...");  
// writeChunks(chunks, outputFolder, numberOfChunks);  
  
// DTC.dtcClassify(dtc+"\\chunk\_1.csv",dtc);  
  
  
 try {  
 int numFilesWritten = CSVSplitter.*splitCSV*(csvInputFile, chunkOutputFolder, numberOfChunks);  
 System.*out*.println("Total number of files written: " + numFilesWritten);  
  
 CSVToARFF.*convertCSVsToARFFs*(chunkOutputFolder, arffOutputFile);  
 System.*out*.println("Conversion completed successfully.");  
  
 p.apply("ReadInputData", TextIO.*read*().from(chunkOutputFolder + "\\chunk\_1.csv"))  
 .apply("InvokeDTC", MapElements.*into*(TypeDescriptors.*strings*()).via((String line) -> {  
 try {  
 DTC.*dtcClassify*(*dtc*+"\\chunk\_1.csv",*dtc*);  
 } catch (Exception e) {  
 *LOG*.warn("Error invoking dtcclassify(): {}", e.getMessage());  
 }  
 return ""; // or any other value as per your requirement  
 }));  
  
 // Block window average  
 int blockSize = 5; // Define your block size  
 BlockWindowAverage blockWindowAverage = new BlockWindowAverage(blockSize);  
  
 String csvFile = App.*dtc* + "\\chunk\_1.csv"; // Provide the path to your CSV file  
 String outputDirectory = App.*dtc* + "\\BWA"; // Provide the path to the output directory  
 String line;  
 String cvsSplitBy = ",";  
  
 // Check if the input file exists  
 File inputFile = new File(csvFile);  
 if (!inputFile.exists()) {  
 System.*err*.println("Input file does not exist: " + csvFile);  
 return;  
 }  
  
 try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {  
 while ((line = br.readLine()) != null) {  
 String[] data = line.split(cvsSplitBy);  
 // Skipping header row  
 if (!data[0].equals("medallion")) {  
 try {  
 TaxiData taxiData = new TaxiData(data);  
 blockWindowAverage.addData(taxiData);  
 // Save processed data to new file for each block  
 blockWindowAverage.saveProcessedData(outputDirectory, blockSize);  
 System.*out*.println("Average total amount for block: " + blockWindowAverage.getAverage());  
 } catch (ParseException e) {  
 System.*err*.println("Error parsing data: " + e.getMessage());  
 }  
 }  
 }  
 } catch (IOException e) {  
 System.*err*.println("Error reading file: " + e.getMessage());  
 }  
  
 blockWindowAverage.saveAverageToFile(outputDirectory);  
 // Write accumulated average data to file  
// blockWindowAverage.writeAveragesToFile(outputDirectory);  
  
 } catch (IOException e) {  
 System.*err*.println("Error occurred while splitting CSV file: " + e.getMessage());  
 e.printStackTrace();  
 }  
  
 //--- Run the pipeline  
 System.*out*.println("Running the pipeline...");  
 p.run().waitUntilFinish();  
  
 System.*out*.println("CSV file has been split successfully.");  
  
 }  
  
}

The CSVSplitter class offers a convenient solution for breaking down large CSV files into smaller, more manageable chunks. Its primary method, splitCSV, accepts three parameters: the path to the input CSV file, the directory where the split chunks will be stored, and the desired number of resulting chunks. The method begins by checking if the specified output folder exists; if not, it creates the directory. Following this, it iterates through the input CSV file to determine the total number of rows, crucial for calculating the approximate size of each chunk. By dividing the total rows by the number of desired chunks, the method calculates an initial chunk size, taking into account any remainder rows that may not evenly distribute among chunks. Then, it proceeds to split the CSV file, creating individual chunk files with data distributed across them. These files are sequentially named, and the method ensures that existing files are not overwritten during the process. Throughout the operation, the method prints informative messages, indicating the generation of each output file. Ultimately, the CSVSplitter class serves as a valuable tool for efficiently handling large CSV datasets, enabling streamlined processing and analysis tasks.

package abolfazl.younesi.beamutil;  
  
import java.io.BufferedReader;  
import java.io.BufferedWriter;  
import java.io.FileReader;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
  
public class CSVSplitter {  
 public static int splitCSV(String inputFile, String outputFolder, int numberOfChunks) throws IOException {  
 if (!Files.*exists*(Paths.*get*(outputFolder))) {  
 Files.*createDirectories*(Paths.*get*(outputFolder));  
 System.*out*.println("Dataset files already exists!");  
 }  
  
 // Count the total number of rows in the CSV file  
 int totalRows = 0;  
 try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {  
 while (br.readLine() != null) {  
 totalRows++;  
 }  
 }  
 // Calculate the approximate chunk size  
 int chunkSize = totalRows / numberOfChunks;  
 int remainder = totalRows % numberOfChunks;  
  
 // Read and split the CSV file into chunks  
 int chunkNumber = 1;  
 int filesWritten = 0; // Counter for files written  
 try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {  
 for (int i = 0; i < numberOfChunks; i++) {  
 String outputFileName = outputFolder + "/Java\_chunk\_" + chunkNumber + ".csv";  
  
 // Check if the output file already exists, if not then write to it  
 if (!Files.*exists*(Paths.*get*(outputFileName))) {  
 try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFileName))) {  
 for (int j = 0; j < chunkSize; j++) {  
 String line = br.readLine();  
 if (line != null) {  
 writer.write(line);  
 writer.newLine();  
 }  
 }  
 chunkNumber++;  
 filesWritten++;  
 System.*out*.println("Output file generated: " + outputFileName); // Print output file generation  
 }  
 // If there is a remainder, distribute the remaining rows among the first few chunks  
 if (remainder > 0) {  
 String line = br.readLine();  
 if (line != null) {  
 try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFileName, true))) {  
 writer.write(line);  
 writer.newLine();  
 }  
 }  
 remainder--;  
 }  
 } else {  
 // Output file already exists, move to the next chunk  
 chunkNumber++;  
 }  
 }  
 }  
 return filesWritten;  
 }  
}

The CSVToARFF provides functionality to convert CSV files to ARFF (Attribute-Relation File Format), a common format used in data mining and machine learning. The class contains a single method convertCSVsToARFFs, which accepts the directory paths of CSV files and the target directory for storing the converted ARFF files.

The method begins by scanning the specified CSV directory for files with the .csv extension. For each CSV file found, it checks if a corresponding ARFF file already exists in the target directory. If so, it skips the conversion process for that file. Otherwise, it proceeds to convert the CSV file to ARFF format. During conversion, the method reads attribute names from the first row of the CSV file and writes corresponding attribute declarations to the ARFF file. It distinguishes between different types of attributes such as nominal (categorical) and numeric. Date attributes are formatted as per the specified pattern. Once attribute declarations are written, the method reads each row from the CSV file and writes it to the ARFF file.

package abolfazl.younesi.beamutil;  
  
import java.io.File;  
import java.io.BufferedReader;  
import java.io.BufferedWriter;  
import java.io.FileReader;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class CSVToARFF {  
 public static void convertCSVsToARFFs(String csvDirectory, String arffDirectory) {  
 File csvFolder = new File(csvDirectory);  
 File[] csvFiles = csvFolder.listFiles((dir, name) -> name.toLowerCase().endsWith(".csv"));  
  
 if (csvFiles == null) {  
 System.*err*.println("No CSV files found in the directory: " + csvDirectory);  
 return;  
 }  
  
 for (File csvFile : csvFiles) {  
 String arffFileName = arffDirectory + "/" + csvFile.getName().replace(".csv", ".arff");  
 File arffFile = new File(arffFileName);  
  
 if (arffFile.exists()) {  
 System.*out*.println("ARFF file already exists for: " + csvFile.getName());  
 continue; // Skip conversion  
 }  
  
 try {  
 System.*out*.println("Converting: " + csvFile.getName() + " to ARFF...");  
 *convertCSVtoARFF*(csvFile.getAbsolutePath(), arffFileName);  
 System.*out*.println("Converted: " + csvFile.getName() + " -> " + arffFileName);  
 } catch (IOException e) {  
 System.*err*.println("Error converting " + csvFile.getName() + " to ARFF: " + e.getMessage());  
 }  
 }  
 }  
  
 private static void convertCSVtoARFF(String csvFile, String arffFile) throws IOException {  
 // Open the CSV file for reading  
 System.*out*.println("Opening CSV file: " + csvFile);  
 BufferedReader csvReader = new BufferedReader(new FileReader(csvFile));  
 // Open the ARFF file for writing  
 System.*out*.println("Creating ARFF file: " + arffFile);  
 BufferedWriter arffWriter = new BufferedWriter(new FileWriter(arffFile));  
  
 // Write ARFF header  
 arffWriter.write("@relation data\n\n");  
  
 // Read the attribute names from the first row  
 String[] attributes = csvReader.readLine().split(",");  
 // Write attribute declarations  
 for (String attribute : attributes) {  
 // Remove leading/trailing spaces and single quotes  
 attribute = attribute.trim().replaceAll("^'+|'+$", "");  
 if (attribute.equalsIgnoreCase("vendor\_id")) {  
 arffWriter.write("@attribute " + attribute + " {VTS,CMT}\n");  
 } else if (attribute.equalsIgnoreCase("pickup\_datetime")) {  
 arffWriter.write("@attribute " + attribute + " date 'yyyy-MM-dd HH:mm:ss'\n");  
 } else if (attribute.equalsIgnoreCase("payment\_type")) {  
 arffWriter.write("@attribute " + attribute + " {CSH,CRD,NOC,DIS,UNK}\n");  
 } else {  
 arffWriter.write("@attribute " + attribute + " numeric\n");  
 }  
 }  
 arffWriter.write("\n@data\n");  
  
 // Read each line from CSV and write to ARFF  
 String row;  
 while ((row = csvReader.readLine()) != null) {  
 arffWriter.write(row + "\n");  
 }  
  
 // Close readers and writers  
 System.*out*.println("Closing CSV and ARFF files...");  
 csvReader.close();  
 arffWriter.close();  
 }  
}

The BlockWindowAverage class provides functionality for calculating the block window average of taxi trip data. Below is a breakdown of its key features:

Queue for Block Data: The class utilizes a queue (blockData) to store taxi trip data for the current block, ensuring efficient handling of incoming data.

Block Size Management: Upon instantiation, the class specifies the block size, determining the number of data entries to be considered within each block.

Data Accumulation: Taxi trip data is added to the current block using the addData method. The class maintains the sum of total amounts (blockSum) within the block for subsequent average calculations.

Average Calculation: The getAverage method computes the average total amount for the current block, facilitating analysis of fare trends.

Output Handling:

* The saveProcessedData method saves the processed data for the current block to individual files within a specified directory.
* The saveAverageToFile method writes the accumulated average values to a text file.
* The writeAveragesToFile method stores the accumulated average data in a single CSV file.

Data Conversion:

* The dataToString method converts a TaxiData object to a string format, facilitating storage and retrieval of taxi trip details.

The BlockWindowAverage class provides essential functionality for analyzing taxi trip data in a block-wise manner, enabling insights into fare patterns and trends over time. Its efficient data management and output handling make it a valuable tool for processing and analyzing large datasets.

package abolfazl.younesi.bolts;  
  
import java.io.\*;  
import java.text.SimpleDateFormat;  
import java.util.ArrayDeque;  
import java.util.Queue;  
// Class for calculating block window average  
public class BlockWindowAverage {  
 private final Queue<TaxiData> blockData; // Queue to store taxi data for the current block  
 private final int blockSize;  
 private double blockSum; // Sum of total amounts in the current block  
 private int fileIndex; // To keep track of processed files  
 private final StringBuilder averageData; // Accumulator for average values  
  
 // Constructor to initialize block window average calculator  
 public BlockWindowAverage(int blockSize) {  
 this.blockSize = blockSize;  
 this.blockData = new ArrayDeque<>(blockSize);  
 this.blockSum = 0;  
 this.fileIndex = 1; // Initialize file index  
 this.averageData = new StringBuilder();  
 }  
  
 // Method to add taxi data to the current block  
 public void addData(TaxiData data) {  
 blockData.add(data);  
 blockSum += data.getTotalAmount();  
 if (blockData.size() > blockSize) {  
 TaxiData removedData = blockData.poll();  
 blockSum -= removedData.getTotalAmount();  
 }  
 }  
  
 // Method to calculate the average total amount for the current block  
 public double getAverage() {  
 return blockSum / blockData.size();  
 }  
  
 public void saveAverageToFile(String directory) {  
 try (FileWriter writer = new FileWriter(directory + "/average\_values.txt")) {  
 writer.write(averageData.toString());  
 } catch (IOException e) {  
 System.*err*.println("Error writing average values to file: " + e.getMessage());  
 }  
 }  
 // Method to save processed data for the current block to individual files  
 public void saveProcessedData(String directory, int blockSize) {  
 // Create the output directory if it doesn't exist  
 File outputDir = new File(directory);  
 if (!outputDir.exists()) {  
 if (!outputDir.mkdirs()) {  
 System.*err*.println("Failed to create directory: " + directory);  
 return;  
 }  
 }  
  
 // Write block data to individual files  
 try (PrintWriter writer = new PrintWriter(new File(directory, "BWA\_chunk\_" + blockSize + "\_" + fileIndex + ".csv"))) {  
 for (TaxiData data : blockData) {  
 writer.println(dataToString(data));  
 }  
 } catch (FileNotFoundException e) {  
 System.*err*.println("Error saving processed data: " + e.getMessage());  
 }  
  
 // Accumulate average data  
 double average = getAverage();  
 averageData.append(average).append("\n");  
  
 fileIndex++; // Increment file index after processing each file  
 }  
  
 // Method to write accumulated average data to a single file  
 public void writeAveragesToFile(String directory) {  
 // Create the output directory if it doesn't exist  
 File outputDir = new File(directory);  
 if (!outputDir.exists()) {  
 if (!outputDir.mkdirs()) {  
 System.*err*.println("Failed to create directory: " + directory);  
 return;  
 }  
 }  
  
 // Write accumulated average data to a single file  
 try (PrintWriter writer = new PrintWriter(new File(directory, "BWA\_Averages.csv"))) {  
 writer.println(averageData);  
 } catch (FileNotFoundException e) {  
 System.*err*.println("Error writing average data to file: " + e.getMessage());  
 }  
 }  
  
 // Method to convert TaxiData object to string  
 private String dataToString(TaxiData data) {  
 SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
 return data.getMedallion() + "," +  
 data.getHackLicense() + "," +  
 data.getVendorId() + "," +  
 sdf.format(data.getPickupDatetime()) + "," +  
 data.getPaymentType() + "," +  
 data.getFareAmount() + "," +  
 data.getSurcharge() + "," +  
 data.getMtaTax() + "," +  
 data.getTipAmount() + "," +  
 data.getTollsAmount() + "," +  
 data.getTotalAmount();  
 }  
}

The DTC class, serves as a robust tool for executing Decision Tree Classification (DTC) tasks utilizing the Weka library. Central to its functionality is the dtcClassify method, which orchestrates the entire classification process. This method handles data loading from CSV files, initialization of the decision tree classifier, model training and evaluation through cross-validation, and serialization of the trained model for future use. Additionally, it generates visualizations to provide insights into model performance, including accuracy and loss charts and ROC curve visualizations. These visual representations are saved as PNG files for further analysis.

package abolfazl.younesi.bolts;  
import weka.classifiers.Evaluation;  
import weka.classifiers.trees.J48;  
import weka.core.Instances;  
import weka.core.converters.CSVLoader;  
import weka.gui.visualize.PlotData2D;  
import weka.gui.visualize.ThresholdVisualizePanel;  
import org.jfree.chart.ChartFactory;  
import org.jfree.chart.ChartUtils;  
import org.jfree.chart.JFreeChart;  
import org.jfree.chart.plot.PlotOrientation;  
import org.jfree.data.xy.XYSeries;  
import org.jfree.data.xy.XYSeriesCollection;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.image.BufferedImage;  
import java.io.File;  
import java.util.Random;  
  
public class DTC {  
 public static void dtcClassify(String csvFilePath, String outputFolder) {  
 try {  
 System.*out*.println("Loading CSV data...");  
 CSVLoader loader = new CSVLoader();  
 loader.setSource(new File(csvFilePath));  
 Instances data = loader.getDataSet();  
  
 System.*out*.println("Setting class attribute...");  
 data.setClassIndex(data.attribute(" payment\_type").index());  
  
 System.*out*.println("Initializing decision tree classifier...");  
 J48 tree = new J48();  
 tree.setUnpruned(false); // Unpruned tree  
  
 System.*out*.println("Training and evaluating the model...");  
 Evaluation eval = *trainAndEvaluateModel*(tree, data);  
  
 System.*out*.println("Saving the trained model...");  
 *saveModel*(tree);  
  
 System.*out*.println("Generating visualizations...");  
 *generateVisualizations*(eval, data, outputFolder);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 private static Evaluation trainAndEvaluateModel(J48 tree, Instances data) throws Exception {  
 int nFolds = 5; // Number of folds for cross-validation  
 Evaluation eval = new Evaluation(data);  
 eval.crossValidateModel(tree, data, nFolds, new Random(1));  
 return eval;  
 }  
  
 private static void saveModel(J48 tree) throws Exception {  
 weka.core.SerializationHelper.*write*("decision\_tree.model", tree);  
 }  
  
 private static void generateVisualizations(Evaluation eval, Instances data, String outputFolder) throws Exception {  
 // Generate accuracy and loss chart  
 *generateAccuracyLossChart*(eval, outputFolder);  
  
 // Generate ROC curve visualization  
 *generateROCCurveVisualization*(eval, data, outputFolder);  
 }  
  
 private static void generateAccuracyLossChart(Evaluation eval, String outputFolder) throws Exception {  
 // Create a chart for accuracy and loss  
 XYSeries accuracySeries = new XYSeries("Accuracy");  
 XYSeries lossSeries = new XYSeries("Loss");  
  
 for (int i = 0; i < eval.numInstances(); i++) {  
 accuracySeries.add(i, eval.pctCorrect());  
 lossSeries.add(i, eval.rootMeanSquaredError());  
 }  
  
 XYSeriesCollection dataset = new XYSeriesCollection();  
 dataset.addSeries(accuracySeries);  
 dataset.addSeries(lossSeries);  
  
 JFreeChart chart = ChartFactory.*createXYLineChart*(  
 "Accuracy and Loss",  
 "Instances",  
 "Value",  
 dataset,  
 PlotOrientation.*VERTICAL*,  
 true,  
 true,  
 false  
 );  
  
 // Save the chart as PNG file  
 String outputFileName = outputFolder + File.*separator* + "accuracy\_and\_loss\_chart.png";  
 ChartUtils.*saveChartAsPNG*(new File(outputFileName), chart, 800, 600);  
 System.*out*.println("Chart saved as PNG file: " + outputFileName);  
 }  
  
 private static void generateROCCurveVisualization(Evaluation eval, Instances data, String outputFolder) throws Exception {  
 // Create a chart for ROC curve  
 ThresholdVisualizePanel vmc = new ThresholdVisualizePanel();  
 vmc.setROCString("(Area under ROC) - Class 0: " + eval.areaUnderROC(0) + ", Class 1: " + eval.areaUnderROC(1));  
 vmc.setName(data.relationName());  
 PlotData2D tempPlot = new PlotData2D(data);  
 tempPlot.setPlotName(data.relationName());  
 tempPlot.addInstanceNumberAttribute();  
  
 // Specify which points are connected  
 boolean[] cp = new boolean[data.numInstances()];  
 for (int n = 1; n < cp.length; n++)  
 cp[n] = true;  
 tempPlot.setConnectPoints(cp);  
  
 // Add plot to the visualization panel  
 vmc.addPlot(tempPlot);  
  
 // Display the ROC curve  
 String plotName = vmc.getName();  
 JFrame jf = new JFrame("Decision Tree Visualizer: " + plotName);  
 jf.setSize(800, 600);  
 jf.getContentPane().setLayout(new BorderLayout());  
 jf.getContentPane().add(vmc, BorderLayout.*CENTER*);  
 jf.addWindowListener(new java.awt.event.WindowAdapter() {  
 public void windowClosing(java.awt.event.WindowEvent e) {  
 jf.dispose();  
 }  
 });  
 jf.setVisible(true);  
  
 // Save the ROC curve chart as PNG file  
 String outputFileName = outputFolder + File.*separator* + "decision\_tree\_visualization.png";  
 *saveChartAsPNG*(outputFileName, vmc);  
 System.*out*.println("ROC curve chart saved as PNG file: " + outputFileName);  
 }  
  
 private static void saveChartAsPNG(String outputFileName, Component component) {  
 try {  
 BufferedImage image = new BufferedImage(component.getWidth(), component.getHeight(), BufferedImage.*TYPE\_INT\_ARGB*);  
 component.paint(image.getGraphics());  
 javax.imageio.ImageIO.*write*(image, "PNG", new File(outputFileName));  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

Below (see the Fig. 1) we are going to see the accuracy and the loss of trained decision tree model on the Taxi dataset. moreover in Fig. 2 the result of the evaluation of the trained model on test data has been illustrated. This evaluation includes the confusion matrix.

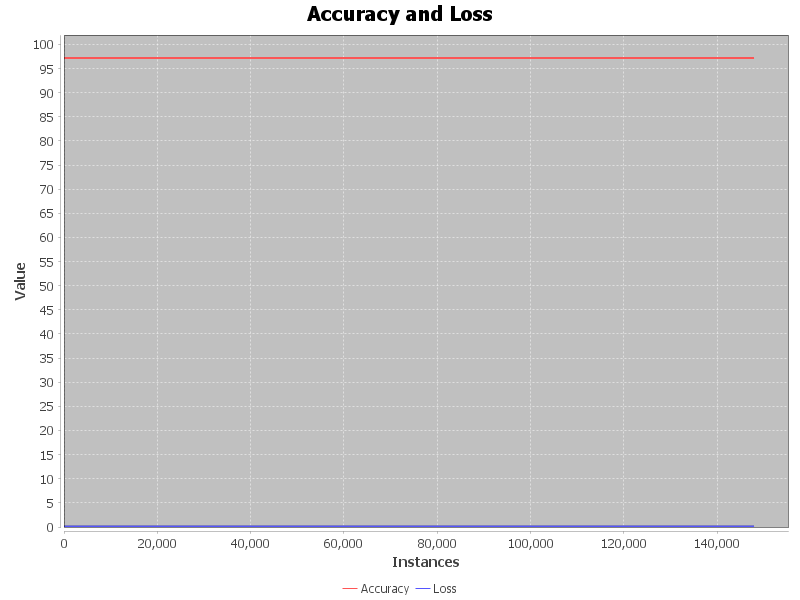


Figure 1 accuracy and loss of the trained decision tree

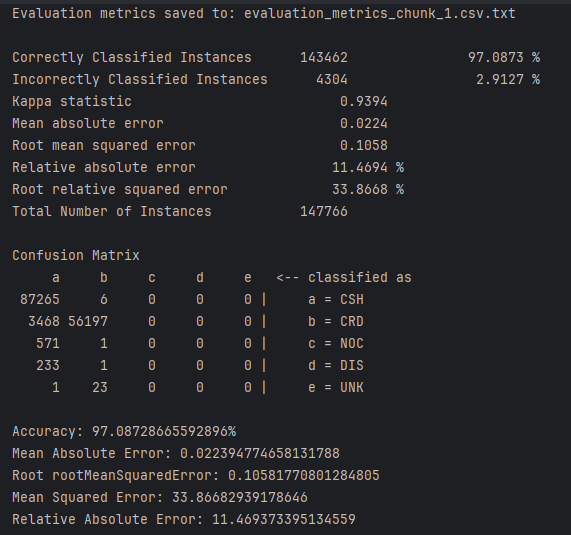


Figure 2 evaluation of the test data

The MLVR class orchestrates Machine Learning-driven Linear Regression tasks utilizing the Weka library. Initially, the class loads CSV data from a specified file location, ensuring it is readily available for subsequent processing. It then proceeds to train a Linear Regression model, employing a batch-based approach to handle potentially large datasets efficiently. Throughout the training process, the model is iteratively updated with batches of data until convergence is achieved. Once trained, the model undergoes rigorous evaluation against the loaded dataset, calculating a range of evaluation metrics to assess its performance. These metrics include Mean Absolute Error, Root Mean Squared Error, Relative Absolute Error, among others, providing insights into the model's accuracy and predictive capabilities. Furthermore, the trained model is serialized and stored as a file for future use, ensuring persistence and reusability. Additionally, the class generates predictions by applying the trained model to the dataset, enabling inference on unseen data instances. Finally, it produces a visualization depicting the predicted values plotted against the actual values, facilitating a visual understanding of the model's predictive accuracy. This plot is then saved as an image file for further analysis.

package abolfazl.younesi.bolts;  
  
import org.jfree.chart.ChartUtils;  
import weka.core.Instances;  
import weka.core.converters.CSVLoader;  
import weka.classifiers.functions.LinearRegression;  
import weka.classifiers.evaluation.NumericPrediction;  
import weka.classifiers.evaluation.Prediction;  
import weka.classifiers.evaluation.Evaluation;  
import org.jfree.chart.ChartFactory;  
import org.jfree.chart.JFreeChart;  
import org.jfree.data.xy.XYSeries;  
import org.jfree.data.xy.XYSeriesCollection;  
  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.List;  
  
public class MLVR {  
  
 public static void main(String[] args) {  
 try {  
 // Load CSV data  
 System.*out*.println("Loading CSV data...");  
 File csvFile = new File("F:\\utf-8-FOIL2013\\FOIL2013\\output\\chunk\_2.csv");  
 if (!csvFile.exists()) {  
 throw new IOException("CSV file not found.");  
 }  
 CSVLoader loader = new CSVLoader();  
 loader.setSource(csvFile);  
 Instances data = loader.getDataSet();  
  
 // Set the class attribute index  
 data.setClassIndex(data.attribute(" tip\_amount").index());  
  
 // Apply NominalToBinary filter  
// System.out.println("Converting nominal attributes to binary...");  
// NominalToBinary filter = new NominalToBinary();  
// filter.setInputFormat(data);  
// Instances filteredData = Filter.useFilter(data, filter);  
  
 // Train Linear Regression model  
 System.*out*.println("Training Linear Regression model...");  
 // Define batch size  
 int batchSize = 10;  
  
 // Train Linear Regression model in batches  
 LinearRegression model = new LinearRegression();  
 for (int i = 0; i < data.numInstances(); i += batchSize) {  
 System.*out*.println("Training Linear Regression model..."+i);  
 Instances batch = new Instances(data, i, Math.*min*(batchSize, data.numInstances() - i));  
 model.buildClassifier(batch);  
 }  
  
 // Save the model  
 System.*out*.println("Saving the model...");  
 weka.core.SerializationHelper.*write*("linear\_regression\_model.model", model);  
  
 // Evaluate the model  
 System.*out*.println("Evaluating the model...");  
 Evaluation evaluation = new Evaluation(data);  
 evaluation.evaluateModel(model, data);  
  
 // Print evaluation metrics  
 System.*out*.println("Evaluation Metrics:");  
 System.*out*.println("------------------------------");  
 System.*out*.println("Mean Absolute Error: " + evaluation.meanAbsoluteError());  
 System.*out*.println("Root Mean Squared Error: " + evaluation.rootMeanSquaredError());  
 System.*out*.println("Relative Absolute Error: " + evaluation.relativeAbsoluteError());  
 System.*out*.println("Root Relative Squared Error: " + evaluation.rootRelativeSquaredError());  
 System.*out*.println("Correlation Coefficient: " + evaluation.correlationCoefficient());  
 System.*out*.println("Coefficient of Determination: " + evaluation.correlationCoefficient());  
 System.*out*.println(evaluation.toSummaryString());  
  
 // Get predictions  
 System.*out*.println("Generating predictions...");  
 List<Prediction> predictions = new ArrayList<>();  
 for (int i = 0; i < data.numInstances(); i++) {  
 double actual = data.instance(i).classValue();  
 double predicted = model.classifyInstance(data.instance(i));  
 predictions.add(new NumericPrediction(actual, predicted));  
 }  
  
 // Plot accuracy and loss  
 System.*out*.println("Plotting accuracy and loss...");  
  
 XYSeries series = new XYSeries("Predicted vs. Actual");  
 for (int i = 0; i < predictions.size(); i++) {  
 double actual = predictions.get(i).actual();  
 double predicted = predictions.get(i).predicted();  
 series.add(i, predicted);  
 }  
  
 XYSeriesCollection dataset = new XYSeriesCollection(series);  
 JFreeChart chart = ChartFactory.*createXYLineChart*(  
 "Linear Regression Model Evaluation",  
 "Instance Number",  
 "Predicted",  
 dataset  
 );  
  
 // Save the chart as an image  
 System.*out*.println("Saving the plot as an image...");  
 File chartFile = new File("accuracy\_loss\_chart.png");  
 ChartUtils.*saveChartAsPNG*(chartFile,  
 chart,  
 800,  
 600);  
  
 System.*out*.println("Process completed successfully!");  
 } catch (IOException e) {  
 System.*err*.println("Error: " + e.getMessage());  
 } catch (Exception e) {  
 System.*err*.println("An unexpected error occurred: " + e.getMessage());  
 e.printStackTrace();  
 }  
 }  
}

The DTTest class encapsulates methods for loading a pre-trained decision tree model, loading test data from a CSV file, classifying test instances using the loaded model, and evaluating the model's performance. The loadModel method loads a serialized decision tree model from a file, while the loadTestData method reads test data from a CSV file and sets the class attribute appropriately. The classifyTestData method predicts the class of each test instance using the loaded model, printing the predicted class for each instance. Finally, the evaluateModel method evaluates the model's performance on the test data, calculating various metrics such as accuracy, mean absolute error, and generating a confusion matrix, which are then saved into a text file.

package abolfazl.younesi.spout;  
  
import abolfazl.younesi.App;   
import weka.classifiers.trees.J48;   
import weka.classifiers.Evaluation;   
import weka.core.SerializationHelper;   
import weka.core.Instances;   
import weka.core.converters.CSVLoader;   
  
import java.io.FileWriter;   
import java.io.PrintWriter;   
import java.io.File;   
import java.io.IOException;   
  
public class DTTest {  
 public static J48 loadModel(String modelFilePath) throws IOException, ClassNotFoundException { // Method for loading the saved model  
  
 try {  
 System.*out*.println("Loading model from file: " + modelFilePath);  
 J48 model = (J48) SerializationHelper.*read*(modelFilePath);  
 if (model == null) {  
 System.*err*.println("Error: Loaded model is null.");  
 throw new RuntimeException("Loaded model is null.");  
 }  
 System.*out*.println("Model loaded successfully." + model);  
 return model;  
 } catch (Exception e) {  
 System.*err*.println("Error loading model: " + e.getMessage());  
 e.printStackTrace();  
 throw new RuntimeException(e);  
 }  
 }  
  
 public static Instances loadTestData(String csvFilePath) throws IOException { // Method for loading test data from a CSV file  
  
 try {  
 System.*out*.println("Loading test data from file: " + csvFilePath);  
 CSVLoader loader = new CSVLoader();  
  
 loader.setSource(new File(csvFilePath));  
 Instances data = loader.getDataSet();  
  
 // Set class attribute  
 data.setClassIndex(data.attribute(" payment\_type").index());  
  
 System.*out*.println("Test data loaded successfully.");  
 return data;  
 } catch (IOException e) {  
 System.*err*.println("Error loading test data: " + e.getMessage());  
 throw e;  
 }  
 }  
  
 public static void classifyTestData(J48 model, Instances testData) { // Method for classifying test data using the loaded model  
 // Make predictions on test instances  
 try {  
 System.*out*.println("Classifying test data...");  
 for (int i = 0; i < testData.size(); i++) {  
// System.out.println("data: " + testData.get(i));  
 double predictedClass = model.classifyInstance(testData.get(i)); // Classifying test instance  
// System.out.println("Instance " + (i + 1) + ": Predicted class = " + testData.classAttribute().value((int) predictedClass));   
 }  
 System.*out*.println("Test data classified successfully.");   
 } catch (Exception e) {  
 System.*err*.println("Error classifying test data: " + e.getMessage()); // Printing error message  
 }  
 }  
  
 public static void evaluateModel(J48 model, Instances testData, String outputFilePath) { // Method for evaluating the model's performance  
 try {  
 System.*out*.println("Evaluating model on test data...");  
 Evaluation eval = new Evaluation(testData);  
 eval.evaluateModel(model, testData);  
  
 FileWriter fileWriter = new FileWriter(outputFilePath);  
 PrintWriter printWriter = new PrintWriter(fileWriter);  
  
 // Writing evaluation metrics to the output file  
 printWriter.println(eval.toSummaryString());  
 printWriter.println(eval.toMatrixString("Confusion Matrix"));  
 printWriter.println("Accuracy: " + eval.pctCorrect() + "%");  
 printWriter.println("Mean Absolute Error: " + eval.meanAbsoluteError());  
 printWriter.println("Root Mean Squared Error: " + eval.rootMeanSquaredError());  
 printWriter.println("Relative Absolute Error: " + eval.relativeAbsoluteError());  
 printWriter.println("root Relative Squared Error: " + eval.rootRelativeSquaredError());  
  
 printWriter.close();  
  
 System.*out*.println("Evaluation metrics saved to: " + outputFilePath);  
  
 // Printing evaluation metrics  
 System.*out*.println(eval.toSummaryString());  
 System.*out*.println(eval.toMatrixString("Confusion Matrix"));  
 System.*out*.println("Accuracy: " + eval.pctCorrect() + "%");  
 System.*out*.println("Mean Absolute Error: " + eval.meanAbsoluteError());  
 System.*out*.println("Root rootMeanSquaredError: " + eval.rootMeanSquaredError());  
 System.*out*.println("Mean Squared Error: " + eval.rootRelativeSquaredError());  
 System.*out*.println("Relative Absolute Error: " + eval.relativeAbsoluteError());  
 // You can also access other metrics such as precision, recall, F-measure, etc. using eval.precision(), eval.recall(), eval.fMeasure() etc.  
 } catch (Exception e) {  
 System.*err*.println("Error evaluating model: " + e.getMessage());  
 e.printStackTrace();  
 }  
 }  
   
}

Then we've developed a Java program to subscribe to an MQTT broker and monitor the messages received along with system metrics. Initially, we set up the MQTT connection parameters such as broker address, client ID, topic, and quality of service. Upon successful connection, we subscribe to the specified topic in a separate thread and define callbacks for message arrival, connection loss, and delivery completion. The messageArrived callback calculates and prints message details including topic, QoS, content, and latency. We concurrently start monitoring system metrics such as throughput, CPU utilization, and memory utilization in another thread. Through the startMonitoring method, we continuously calculate and display these metrics at regular intervals. CPU utilization is obtained using the OperatingSystemMXBean, while memory utilization is obtained through the MemoryMXBean.

package abolfazl.younesi.spout;  
  
import abolfazl.younesi.genevents.utils.GlobalConstants;  
import org.eclipse.paho.client.mqttv3.\*;  
  
import java.lang.management.ManagementFactory;  
import java.lang.management.MemoryMXBean;  
import java.lang.management.MemoryUsage;  
import java.lang.management.OperatingSystemMXBean;  
import java.util.ArrayDeque;  
import java.util.Queue;  
  
public class MQTTSub {  
  
 private static final long *INTERVAL* = 10000; // Monitoring interval in milliseconds  
 private static final long *WINDOW\_SIZE* = 10; // Size of the sliding window in seconds  
 private static long *startTime*;  
 private static long *messageCount* = 0;  
 private static Queue<Long> *messageTimestamps* = new ArrayDeque<>(); // Sliding window for message timestamps  
  
 public static void main(String[] args) {  
 String broker = GlobalConstants.*mqttBroker*;  
 String clientId = "Subscriber";  
 String subTopic = "topic/test\_pub";  
 int subQos = 1;  
  
 try {  
 System.*out*.println("Connecting to MQTT broker...");  
 MqttClient client = new MqttClient(broker, clientId);  
 MqttConnectOptions options = new MqttConnectOptions();  
 client.connect(options);  
  
 if (client.isConnected()) {  
 System.*out*.println("Connected to MQTT broker.");  
 // Subscribe in a separate thread  
 Thread subscribeThread = new Thread(() -> {  
 try {  
 client.setCallback(new MqttCallback() {  
 public void messageArrived(String topic, MqttMessage message) {  
 long currentTime = System.*currentTimeMillis*();  
 *messageCount*++;  
 *messageTimestamps*.offer(currentTime); // Add message timestamp to sliding window  
  
 // Calculate latency  
 long latency = currentTime - *startTime*;  
 long individualLatency = latency / *messageCount*;  
  
 System.*out*.println("Message arrived:");  
 System.*out*.println(" Topic: " + topic);  
 System.*out*.println(" QoS: " + message.getQos());  
 System.*out*.println(" Content: " + new String(message.getPayload()));  
 System.*out*.println(" Latency: " + individualLatency + " ms");  
 System.*out*.println(" Total Latency: " + latency + " ms");  
 }  
  
 public void connectionLost(Throwable cause) {  
 System.*out*.println("Connection lost: " + cause.getMessage());  
 }  
  
 public void deliveryComplete(IMqttDeliveryToken token) {  
 System.*out*.println("Delivery complete: " + token.isComplete());  
 }  
 });  
  
 client.subscribe(subTopic, subQos);  
 System.*out*.println("Subscribed to topic: " + subTopic);  
  
 // Start monitoring  
 *startTime* = System.*currentTimeMillis*();  
 *startMonitoring*();  
 } catch (MqttException e) {  
 e.printStackTrace();  
 }  
 });  
 subscribeThread.start();  
 }  
  
 } catch (MqttException e) {  
 e.printStackTrace();  
 }  
 }  
  
 // Method to start monitoring  
 private static void startMonitoring() {  
 new Thread(() -> {  
 while (true) {  
 try {  
 Thread.*sleep*(*INTERVAL*);  
 *updateMetrics*();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
 }  
  
 // Method to update metrics using sliding window approach  
 private static void updateMetrics() {  
 long currentTime = System.*currentTimeMillis*();  
  
 // Remove old message timestamps from the sliding window  
 while (!*messageTimestamps*.isEmpty() && *messageTimestamps*.peek() < currentTime - (*WINDOW\_SIZE* \* 1000)) {  
 *messageTimestamps*.poll();  
 *messageCount*--; // Decrease message count for removed messages  
 }  
  
 // Calculate throughput within the sliding window  
 double elapsedTime = (currentTime - *startTime*) / 1000.0; // Convert to seconds  
 double throughput = *messageCount* / elapsedTime;  
 System.*out*.println("Throughput: " + throughput + " messages/second");  
  
 // Calculate CPU utilization  
 double cpuUsage = *getCpuUsage*();  
 System.*out*.println("CPU Utilization: " + cpuUsage + "%");  
  
 // Calculate memory utilization  
 double memoryUsage = *getMemoryUsage*();  
 System.*out*.println("Memory Utilization: " + memoryUsage + "%");  
 }  
  
 // Method to get CPU utilization  
 private static double getCpuUsage() {  
 OperatingSystemMXBean osBean = ManagementFactory.*getOperatingSystemMXBean*();  
 double cpuUsage = osBean.getSystemLoadAverage();  
  
 if (cpuUsage == -1.0) {  
 cpuUsage = 0.0;  
 }  
 return cpuUsage;  
 }  
  
 // Method to get memory utilization  
 private static double getMemoryUsage() {  
 MemoryMXBean memBean = ManagementFactory.*getMemoryMXBean*();  
 MemoryUsage heapMemoryUsage = memBean.getHeapMemoryUsage();  
 return (double) heapMemoryUsage.getUsed() / heapMemoryUsage.getMax() \* 100.0;  
 }  
}

Then, we've implemented a program for publishing data to an MQTT broker. Initially, we set up the MQTT connection parameters such as broker address, client ID, topic, and quality of service. Upon establishing the connection successfully, we start a new thread for publishing messages. Within this thread, we read a CSV file line by line using a BufferedReader and publish each line as an MQTT message with a specified topic and quality of service. To introduce a delay between publishing each message, ensuring a controlled publishing rate, we pause the thread for a specified duration using Thread.sleep. This allows us to regulate the message flow being sent to the broker.

package abolfazl.younesi.spout;  
  
import abolfazl.younesi.App;  
import abolfazl.younesi.genevents.utils.GlobalConstants;  
import org.eclipse.paho.client.mqttv3.MqttClient;  
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;  
import org.eclipse.paho.client.mqttv3.MqttException;  
import org.eclipse.paho.client.mqttv3.MqttMessage;  
  
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class MQTTPub {  
  
 public static void main(String[] args) {  
 String broker = GlobalConstants.*mqttBroker*;  
 String clientId = "Publisher";  
 String pubTopic = "topic/test\_pub";  
 int pubQos = 1;  
 String msg = App.*dtc*+"\\chunk\_1.csv";  
  
 try {  
 System.*out*.println("Connecting to MQTT broker...");  
 MqttClient client = new MqttClient(broker, clientId);  
 MqttConnectOptions options = new MqttConnectOptions();  
 client.connect(options);  
  
 if (client.isConnected()) {  
 System.*out*.println("Connected to MQTT broker.");  
 Thread publishThread = new Thread(() -> {  
 try {  
 // Publish CSV file line by line with delay  
 try (BufferedReader br = new BufferedReader(new FileReader(msg))) {  
 String line;  
 while ((line = br.readLine()) != null) {  
 MqttMessage message = new MqttMessage(line.getBytes());  
 message.setQos(pubQos);  
 client.publish(pubTopic, message);  
  
 System.*out*.println("Published message: " + line);  
 Thread.*sleep*(100); // Delay for .1 second (100 milliseconds)  
 }  
 } catch (IOException | InterruptedException e) {  
 e.printStackTrace();  
 }  
 } catch (MqttException e) {  
 e.printStackTrace();  
 }  
 });  
 publishThread.start();  
 }  
  
 } catch (MqttException e) {  
 e.printStackTrace();  
 }  
 }  
}

moreover, we've developed a data processing pipeline using Apache Beam to calculate block averages from streaming taxi data. The pipeline first reads the input data from a CSV file, then applies a transformation to calculate the average of every fixed-size block of data. Within the CalculateBlockAverage class, we define a custom DoFn that processes each element of the input stream. This class maintains a sliding window of data points, accumulating them until a block size is reached, at which point it calculates the average and emits it with an associated block index. Additionally, we've implemented logic to skip the header line in the input data. Following the calculation of averages, the pipeline converts the results into strings and writes them to output files, each containing a header specifying the structure of the data.

package abolfazl.younesi.bolts;  
import abolfazl.younesi.App;  
import org.apache.beam.sdk.Pipeline;  
import org.apache.beam.sdk.io.TextIO;  
import org.apache.beam.sdk.transforms.DoFn;  
import org.apache.beam.sdk.transforms.ParDo;  
import org.apache.beam.sdk.transforms.windowing.FixedWindows;  
import org.apache.beam.sdk.transforms.windowing.Window;  
import org.apache.beam.sdk.values.KV;  
import org.apache.beam.sdk.values.PCollection;  
import org.joda.time.Duration;  
  
import java.util.ArrayDeque;  
import java.util.Queue;  
  
public class BWABeam {  
 private static final int *BLOCK\_SIZE* = 100; // Set the desired block size  
 public static final String *HEADER* = "BlockIndex, Average";  
  
 public static void main(String[] args) {  
 Pipeline pipeline = Pipeline.*create*();  
  
 PCollection<String> taxiData = pipeline.apply(TextIO.*read*().from(App.*dtc* + "\\chunk\_1.csv"));  
  
 PCollection<KV<Integer, Double>> averages = taxiData  
 .apply(ParDo.*of*(new CalculateBlockAverage()))  
 .apply(Window.<KV<Integer, Double>>*into*(FixedWindows.*of*(Duration.*standardMinutes*(1))));  
  
 PCollection<String> averagesAsStrings = averages.apply(ParDo.*of*(new KVToStringFn()));  
  
 averagesAsStrings  
 .apply(TextIO.*write*()  
 .to(App.*dtc* + "\\BWA")  
 .withHeader(*HEADER*)  
 .withNumShards(4));  
  
 pipeline.run().waitUntilFinish();  
 }  
  
 public static class CalculateBlockAverage extends DoFn<String, KV<Integer, Double>> {  
 private final Queue<Double> blockData;  
 private double blockSum;  
 private int blockIndex;  
 private boolean isHeaderSkipped; // Add this line  
  
 public CalculateBlockAverage() {  
 this.blockData = new ArrayDeque<>(*BLOCK\_SIZE*);  
 this.blockSum = 0;  
 this.blockIndex = 0;  
 this.isHeaderSkipped = false; // Add this line  
 }  
  
 @ProcessElement  
 public void processElement(@Element String line, OutputReceiver<KV<Integer, Double>> receiver) {  
 if (!isHeaderSkipped) { // Add this block  
 isHeaderSkipped = true;  
 return;  
 }  
  
 double totalAmount = parseTotalAmount(line);  
  
 blockData.add(totalAmount);  
 blockSum += totalAmount;  
  
 if (blockData.size() > *BLOCK\_SIZE*) {  
 double removedData = blockData.poll();  
 blockSum -= removedData;  
 }  
  
 if (blockData.size() == *BLOCK\_SIZE*) {  
 double average = blockSum / blockData.size();  
 receiver.output(KV.*of*(blockIndex, average));  
 blockIndex++;  
 blockData.clear();  
 blockSum = 0;  
 }  
 }  
  
 private double parseTotalAmount(String line) {  
 // Assuming the input line is in the format: "value1,value2,...,value10,totalAmount,value11"  
 String[] values = line.split(",");  
 return Double.*parseDouble*(values[10]);  
 }  
 }  
  
 public static class KVToStringFn extends DoFn<KV<Integer, Double>, String> {  
 @ProcessElement  
 public void processElement(@Element KV<Integer, Double> element, OutputReceiver<String> receiver) {  
 receiver.output(element.getKey() + "," + element.getValue());  
 }  
 }  
}

then we have also implemented a module for publishing data to an MQTT broker using Apache Beam. The PublishToMQTTFn class, which extends Apache Beam's DoFn, defines the logic for publishing messages to a specified MQTT topic. Upon setup, it establishes a connection to the broker with configurable parameters such as broker address, client ID, and quality of service (QoS). Connection attempts are retried with a backoff mechanism in case of failure. In the processElement method, incoming elements from the input stream are published to the MQTT topic as messages, provided the client is connected. Otherwise, an error message is displayed. Finally, in the teardown phase, the client disconnects from the MQTT broker if it's connected. This implementation ensures reliable and fault-tolerant publishing of data to MQTT, suitable for integration within Apache Beam pipelines for real-time data processing and IoT applications.

package abolfazl.younesi.spout;  
  
import abolfazl.younesi.App;  
import abolfazl.younesi.genevents.utils.GlobalConstants;  
import org.apache.beam.sdk.Pipeline;  
import org.apache.beam.sdk.io.TextIO;  
import org.apache.beam.sdk.options.PipelineOptionsFactory;  
import org.apache.beam.sdk.transforms.DoFn;  
import org.apache.beam.sdk.transforms.ParDo;  
import org.apache.beam.sdk.values.PCollection;  
import org.eclipse.paho.client.mqttv3.MqttClient;  
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;  
import org.eclipse.paho.client.mqttv3.MqttException;  
import org.eclipse.paho.client.mqttv3.MqttMessage;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class MQTTBeamPub {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(MQTTBeamPub.class);  
  
 public static class PublishToMQTTFn extends DoFn<String, Void> {  
 private final String broker;  
 private final String clientId;  
 private final String pubTopic;  
 private final int pubQos;  
 private MqttClient client;  
 private static final Logger *logger* = LoggerFactory.*getLogger*(PublishToMQTTFn.class);  
  
  
 public PublishToMQTTFn(String broker, String clientId, String pubTopic, int pubQos) {  
 this.broker = broker;  
 this.clientId = clientId;  
 this.pubTopic = pubTopic;  
 this.pubQos = pubQos;  
 }  
  
 @Setup  
 public void setup() throws MqttException, InterruptedException {  
 MqttConnectOptions options = new MqttConnectOptions();  
 int attempts = 0;  
 while (attempts < GlobalConstants.*MAX\_RECONNECT\_ATTEMPTS*) {  
 try {  
 client = new MqttClient(broker, clientId);  
 client.connect(options);  
 *logger*.info("Connected to MQTT broker");  
 break; // Break the loop if connection successful  
 } catch (MqttException e) {  
 *logger*.error("Failed to connect to MQTT broker. Retrying...");  
 attempts++;  
 Thread.*sleep*(10); // Wait for 1 second before retrying  
 }  
 }  
 if (attempts == GlobalConstants.*MAX\_RECONNECT\_ATTEMPTS*) {  
 throw new MqttException(MqttException.*REASON\_CODE\_CLIENT\_NOT\_CONNECTED*);  
 }  
 }  
  
 @ProcessElement  
 public void processElement(@Element String line, OutputReceiver<Void> out) throws MqttException {  
 System.*out*.println("Processing element: " + line);  
 if (client.isConnected()) {  
 MqttMessage message = new MqttMessage(line.getBytes());  
 message.setQos(pubQos);  
 client.publish(pubTopic, message);  
 System.*out*.println("Published message to topic: " + pubTopic);  
 } else {  
 System.*err*.println("MQTT client is not connected. Cannot publish message.");  
 }  
 }  
  
  
 @Teardown  
 public void teardown() throws MqttException {  
 if (client != null && client.isConnected()) {  
 client.disconnect();  
 *logger*.info("Disconnected from MQTT broker");  
 }  
 }  
 }  
}

at the end we will see the results we have got:

