ID3 Implementation

PRACTICAL DATA MINING PROJECT

Riaan Van Onselen- 97124275

31005 Assignment 2

Table of Contents

Introduction:

This report will present an implementation of the ID3 classification algorithm developed by Ross Quinlan in Java to create a decision tree classifier for a mushroom data set (University of Florida). The objective of the classifier is to determine if a given mushroom is either “edible” or “poisonous” through the comparison of categorical data describing the mushrooms attributes. The implementation was completed using five Java classes and an external command line argument parsing library called Docopt. This report will cover the programs usage and design as well as the output of the final implementation.

It is recommended to view the code attached in the appendix through the provided website link as code formatting is important when assess readability and GitHub provides the basic structure to syntax highlighting and indenting.

The final code for this project can be found at the following link:
https://github.com/Skyray626/ADA_ID3

The video presentation for this project can be found at either of the following links:
https://drive.google.com/open?id=0BzpUsWVqqGDRRXYxbHhzekFqM00
https://www.dropbox.com/s/zax8f9gw2othsv2/RiaanVanOnselen_97124275.m4v?dl=0

Program design:

The following sections will describe the design of the implementation of the ID3 algorithm. The implementation was completed using five public class files and an inner class to provide the core functionality of the program. An additional library called Docopt was used to create the command line interaction allowing for easy use of the program. Further information on each class can be found in the extensive code comments.

Java Object Explanations:

Main Class:

This is the main class of the program and provides the starting method for execution. When the program starts, the main method is called and the arguments are passed in from the command line interface. This class makes use of the external library called Docopt to categorise the arguments that are passed in as well as to provide the user with usage instructions when an incorrect launch attempt is detected.

Once the arguments are passed in and stored in a Map data structure, the main method determines what functionality of the program is desired and store the conditions in Booleans. As the training data set is the base requirement of running the program, the main method extracts the data set from the file passed in using the data parser object. If the options for a test data set and prediction data set are active, the program will extract the data set from the corresponding csv files. All data sets are stored in Arraylists for quick access to each individual element. If the flag for binarising the data is set, the program then converts all the data sets into binarised data sets using the data preprocessor class. Once all the data has been converted, a decision tree is created using the ID3 class by passing in a data set list. This data set can be either the original data set extracted from the CSV files or the binarised data set. Using the ID3 object, the main class will pass in data to be tested or predicted. Finally, depending on the options provided files can be created for the decision tree structure, test analysis data or a list of predicted class values.

DataParser Class:

This class provides the methods for extracting a data set from a CSV file. Once the object is created the public method “parseData” is used to create a list of data elements and a data descriptor from the given file. The training file is assumed to have every possible unique value for each attribute. To Increase efficacy the program, the data descriptor is built only once and used to encode all extra CSV files. There are getter methods for both the data descriptor and data set list to allow access outside of the class.

DataDescriptor Class:

This class acts as a description for the stored data set. It contains two lists; One for all the attribute names and one for all the unique values for each attribute. These lists are used to encode the categorical data, from the CSV files, into integers instead of strings. This allows for faster comparisons between values as well as a faster means of counting each data element with the same value. This class provides also methods to convert an array of integer values to an array of string values for printing out information.

DataElement Class:

This class acts as an individual element of data from the data set. Each row from the CSV files is converted into a data element which store an integer array of values that correspond to the categorical data. The data element class also provides methods to print out an individual element in either integer values or string categorical data like the CSV.

DataPreprocessor Class:

This class provides the methods to convert a categorical data set with multiple values for each category into a data set with only binary attributes. It takes in the data set to be converted and the original data descriptor for that data set. It then creates a new data descriptor with attributes such as “habitat = grass”

that only contain a Boolean condition, true or false. Once this converted data descriptor is built, each of the data elements in the set are converted to a new data element with the required attributes set to true. Finally, the new data set and data descriptor can be extracted using getter methods. A converted data descriptor can be passed in to reduce the number of times it must be created.

ID3 Class:

This class creates and holds the decision tree built using the ID3 algorithm and as such it contains the methods to test and print the decision tree. Firstly, a data set and max node depth must be passed in which get handed off to the node class to build the tree. The root node is store for testing and printing the constructed tree. The three main methods are createTreeDiagramScript, testModel and predictClasses. Create tree diagram script outputs a string in the diagram format that graphviz can render. Test model takes in a data set and compares the predicted class against the actual class to perform analysis. A description of the analysis is shown in the program output section. Finally, the predict classes method takes in a data set and returns a list of predicted classes for each data value as a string.

Node Inner Class:

This class is a java inner class of the ID3 class as it is the only class that needs a reference. This class contains the implementation of the ID3 algorithm and particularly uses recursion to construct all the sub nodes. Each node has a set of data, a string identifying which attributes have been used and the current node depth passed into it. These three values change for each node as the tree gets built. Once a node has a single class or no data samples passed into it the recursion stops. The max tree depth passed into the ID3 class provides another method of stopping recursion at a certain depth.

Program Usage:

To use this program there is a minimum requirement of supplying a training data set as a CSV file. The command “java –jar Decision_Tree_Builder_ID3.jar <path to data set>” will run the program with the base operation of build and display a decision tree. The following is a description of additional options that can be added to alter the programs behaviour:

- --oTreeFile=<Path to output file and file name>
  - The file to output the decision tree structure to
- --testFile=<Path to the test data set>
  - The path to the test data set that will evaluate the model
- --oAnalysisFile=<Path to the output file for the test analysis>
  - The path to the output file for the test evaluation data
- --predictFile=<Path to the dataset to predict values for>
  - The path to the data set you wish to predict values for
- --oPredictFile=<Path to the output file for predicted classes>
  - The path to the output file for the predicted classes
- --binarise
  - Converts all attributes to binary attributes for each value of an attribute.
- --treeDepth=<Integer number of decisions before the tree stops>
  - The total number of decisions that the tree can make.
- --showEmptyLeaves
  - Used to display the leaves that contain no samples. For the multiple value attribute trees
- --debug
  - Show the data descriptor and the data elements for debugging

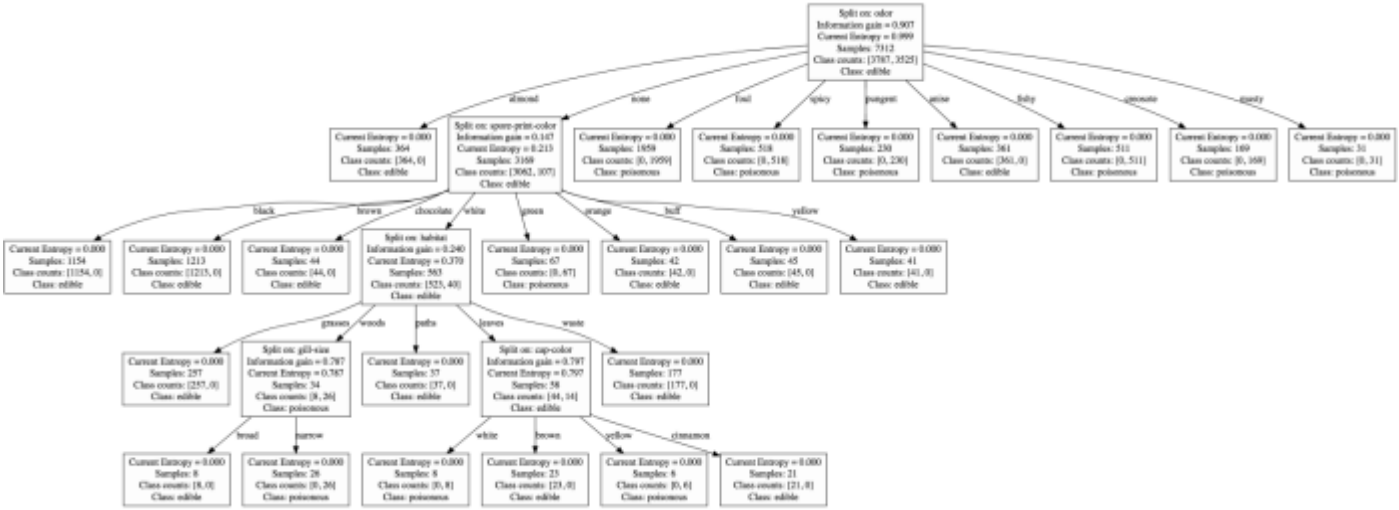
Program Output:

The following section will show four different graphs and outputted accuracy information. A breakdown of the accuracy statistics is shown in the table.

Statistic label	Function
Accuracy	The ratio of correct predictions for both classes against the total number of samples. A higher percentage is desired.
Error Rate	The ratio of incorrect predictions for both classes against the total number of samples. A lower percentage is desired
F1 score	A measure of the tests accuracy. It combines the precision and recall values to gain a more accurate accuracy of the classifier. A higher percentage is desired.

Test 1:

Full tree with multiple values for classes:



Number of samples: 812

True Class \ Predicted Class

\	H0	H1
H0	421	0
H1	0	391

Accuracy Statistics:

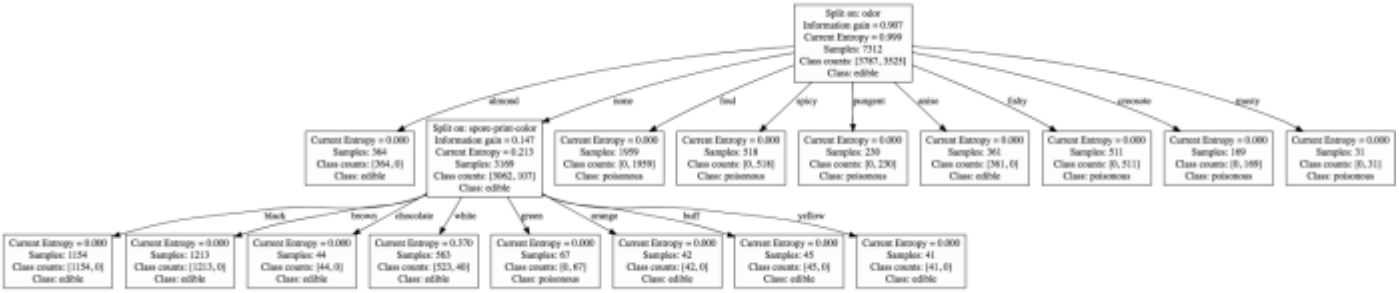
Accuracy: 100.000%

Error Rate: 0.000%

F1 score: 100.000%

Test 2:

Tree with only two decisions and multiple values for classes:



Number of samples: 812

True Class \ Predicted Class

\	H0	H1
H0	421	0
H1	8	383

Accuracy Statistics:

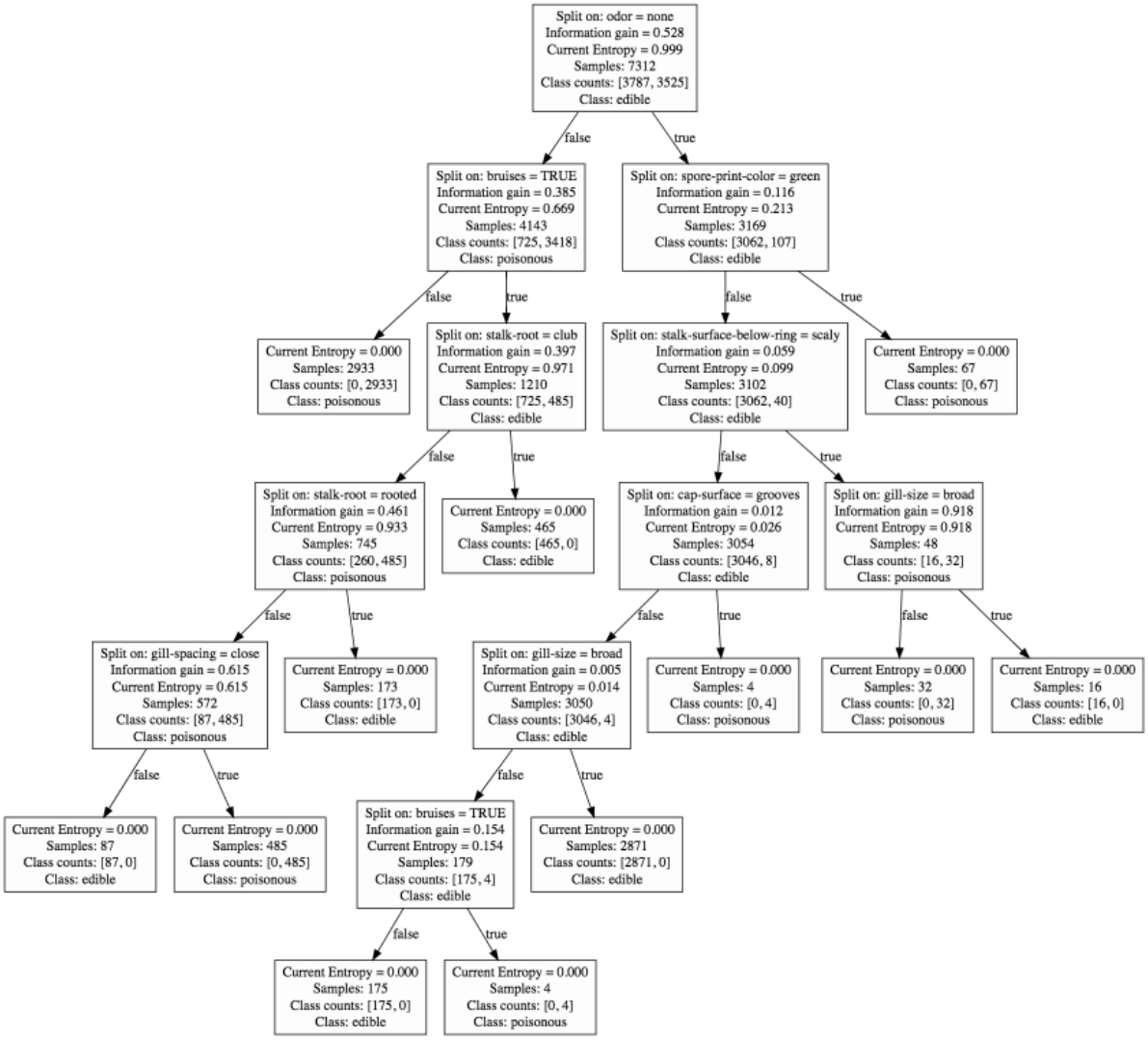
Accuracy: 99.015%

Error Rate: 0.985%

F1 score: 98.966%

Test 3:

Full tree with binarised attributes:



Number of samples: 812

True Class \ Predicted Class

\	H0	H1
H0	421	0
H1	0	391

Accuracy Statistics:

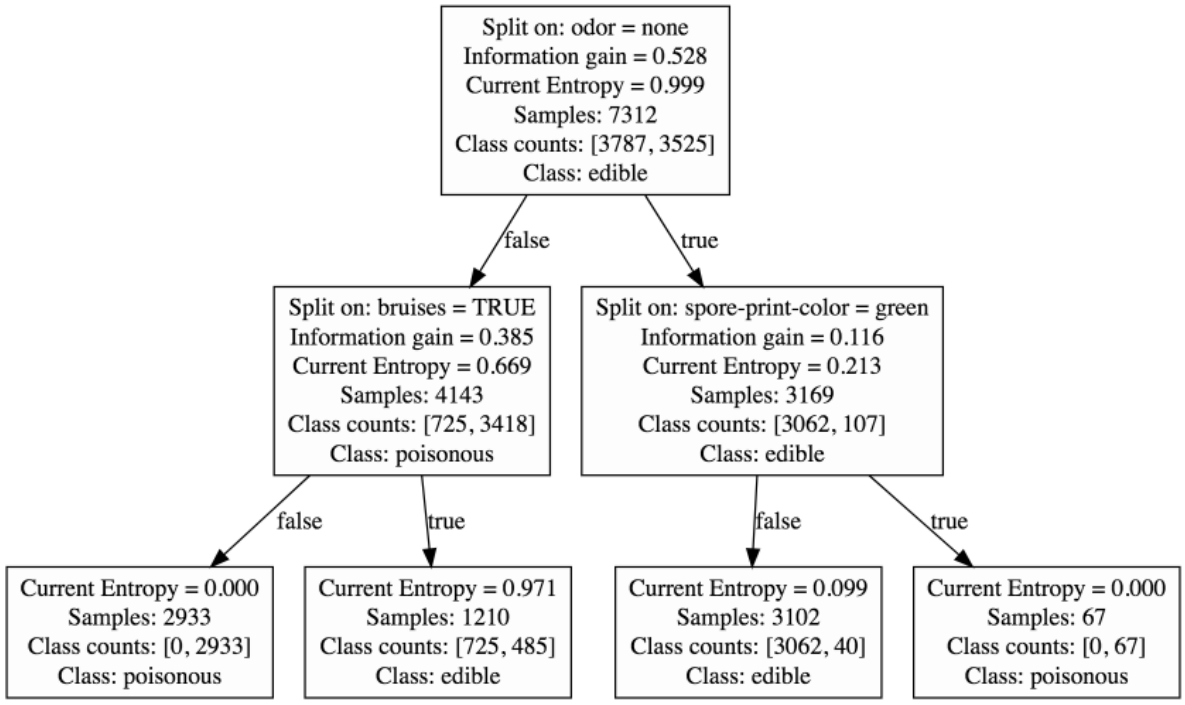
Accuracy: 100.000%

Error Rate: 0.000%

F1 score: 100.000%

Test 4:

Tree with two decisions and binarised attributes:



Number of samples: 812

True Class \ Predicted Class

\	H0	H1
H0	421	0
H1	67	324

Accuracy Statistics:

Accuracy: 91.749%

Error Rate: 8.251%

F1 score: 90.629%

References:

Command line argument parsing library made by Docopt organisation:
https://github.com/docopt/docopt.java

Graphviz Online graph builder made by Dreampuf:
https://dreampuf.github.io/GraphvizOnline/

University of Florida, *The ID3 Algorithm*, viewed 2 October 2017,
<https://www.cise.ufl.edu/~ddd/cap6635/Fall-97/Short-papers/2.htm>.

Wikipedia 2017, *ID3 Algorithm*, viewed 2 October 2017, <https://en.wikipedia.org/wiki/ID3_algorithm>.

Appendix:

Appendix 1: Code

Main:

```
package com.riaanvo;

import org.docopt.Docopt;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.ArrayList;
import java.util.Map;

/**
 * This program takes in a file containing a categorical data set and can create an ID3 model. Different arguments will
 * change the behaviour of the program to include model testing and data classification based on the model.
 */
public class Main {

   private static final String doc = "ID3 Builder\n\n"
         + "Usage:\n"
         + "  ID3_Builder <trainFile> [--oTreeFile=OTREEFILE --binarise --treeDepth=TREEDEPTH --showEmptyLeaves --
debug]\n"

         + "  ID3_Builder <trainFile> [--oTreeFile=OTREEFILE] [--testFile=TESTFILE] [--oAnalysisFile=OANALYSISFILE] "
         + "[--predictFile=PREDICTFILE] [--oPredictFile=OPREDICTFILE] "
         + "[--binarise] [--treeDepth=TREEDEPTH] [--showEmptyLeaves] [--debug]\n"

         + "  ID3_Builder (-h | --help)\n"
         + "  ID3_Builder --version\n"
         + "\n"

         + "Options:\n"
         + "  -h --help                 Show this screen.\n"
         + "  --version                 Show version.\n"
         + "  --testFile=TESTFILE          Test data set file. \n"
         + "  --predictFile=PREDICTFILE      Data set file to be predicted. \n"
         + "  --oTreeFile=OTREEFILE         Filename for the tree output. \n"
         + "  --oAnalysisFile=OANALYSISFILE   Filename for the analysis output. \n"
         + "  --oPredictFile=OPREDICTFILE     Filename for the prediction output. \n"
         + "  --binarise                Converts all categorical data to binary attributes. \n"
         + "  --treeDepth=TREEDEPTH         The number of decisions the tree is restricted to. [default: -1]\n"
         + "  --showEmptyLeaves            Includes the empty leaves in the model. \n"
         + "  --debug                   Prints out the data sets for debugging \n"
         + "\n";

   private static DataDescriptor dataDescriptor;
   private static ArrayList<DataElement> trainDataSet;
   private static ArrayList<DataElement> testDataSet;
   private static ArrayList<DataElement> predictDataSet;
   private static boolean debugMode;

   /**
    * This is the starting point of the ID3 Builder program. It takes in the options and arguments for building and
    * outputting the decision tree.
    * @param args The options and arguments for the ID3 builder
    */
```

```
public static void main(final String[] args) {

   // Extract the arguments from the commandline into the disired tokens
   final Map<String, Object> opts = new Docopt(doc).withVersion("ID3 Builder V1.0").parse(args);

   // Extract the file path, node depth and whether to binarise from the arguments
   String trainFile = opts.get("<trainFile>").toString();
   int nodeDepth = Integer.parseInt(opts.get("--treeDepth").toString());
   boolean binarise = (opts.get("--binarise").toString().equals("true"));
   boolean showEmptyLeaves = (opts.get("--showEmptyLeaves").toString().equals("true"));
   debugMode = (opts.get("--debug").toString().equals("true"));

   // Check if the aditional options have been included and store the result in booleans
   boolean hasOutputStructureFile = opts.get("--oTreeFile") != null;
   boolean hasTestData = opts.get("--testFile") != null;
   boolean hasOutputAnalysisFile = opts.get("--oAnalysisFile") != null;
   boolean hasPredictionData = opts.get("--predictFile") != null;
   boolean hasOutputPredictFile = opts.get("--oPredictFile") != null;

   // Create a data parser to convert the csv file into data objects
   DataParser dataParser = new DataParser();

   // Extract the training data set
   dataParser.parseData(trainFile, null);
   dataDescriptor = dataParser.getDataDescriptor();
   trainDataSet = dataParser.getDataSet();

   if(debugMode){
      displayDataSet(trainDataSet, dataDescriptor);
   }

   // If there is a test data set extract the contents
   if (hasTestData) {

      dataParser.parseData(opts.get("--testFile").toString(), dataDescriptor);
      testDataSet = dataParser.getDataSet();
   }

   // If there is a data set to predict classes for extract the contents
   if (hasPredictionData) {

      dataParser.parseData(opts.get("--predictFile").toString(), dataDescriptor);
      predictDataSet = dataParser.getDataSet();
   }

   // If the binarise option was included, convert all categorical data into binarised attributes
   if (binarise) {

      binariseDataSets(hasTestData, hasPredictionData);
   }

   // Build the ID3 decision tree model
   ID3 id3Tree = new ID3(trainDataSet, nodeDepth);

   // Create a text layout of the model
   String diagramScript = id3Tree.createTreeDiagramScript(showEmptyLeaves);
```

```
    // If there is an output file, write out the text model, else print it to the CLI
    if (hasOutputStructureFile) {

        writeToFile(opts.get("--oTreeFile").toString(), diagramScript);
    } else {

        System.out.println(diagramScript);
    }

    // If there is a test data set, test the performance of the model
    if (hasTestData) {

        String testInformation = id3Tree.testModel(testDataSet);

        // If an output file was provided write the analysis to it, else print it to the CLI
        if (hasOutputAnalysisFile) {

            writeToFile(opts.get("--oAnalysisFile").toString(), testInformation);
        } else {

            System.out.println(testInformation + "\n");
        }
    }

    // If there is a data set to predict classes for, use the model to predict the classes
    if (hasPredictionData) {

        String predictionInformation = id3Tree.predictClasses(predictDataSet);

        // If there is an output file, write the predictions to that file, else display to the CLI
        if (hasOutputPredictFile) {

            writeToFile(opts.get("--oPredictFile").toString(), predictionInformation);
        } else {

            System.out.println(predictionInformation + "\n");
        }
    }
}

/**
 * This method converts all the stored data sets to binarised attribute data sets. This can be used to create
 * decision trees with only true/false decisions and not multiple route decisions
 *
 * @param hasTestData      Does the test set exist
 * @param hasPredictionData Does the prediction set exist
 */
private static void binariseDataSets(boolean hasTestData, boolean hasPredictionData) {

    // Create a data preprocessor to convert to binarised data
    DataPreprocessor preprocessor = new DataPreprocessor();

    // Convert the training data set to the new binarised data set
    preprocessor.binariseDataSet(trainDataSet, dataDescriptor);
    trainDataSet = preprocessor.getDataSet();
```

```
    if(debugMode) {
        displayDataSet(trainDataSet, preprocessor.getDataDescriptor());
    }

    // If a test set was provided, binarise the values and store the new data set
    if (hasTestData) {

        preprocessor.binariseDataSet(testDataSet, dataDescriptor);
        testDataSet = preprocessor.getDataSet();
    }

    // If a prediction set was provided, binarise the values and store the new data set
    if (hasPredictionData) {

        preprocessor.binariseDataSet(predictDataSet, dataDescriptor);
        predictDataSet = preprocessor.getDataSet();
    }
}

/**
 * Writes the passed in string to the desired file name.
 *
 * @param fileName     Name of the output file
 * @param fileContents Contents to be placed in the file
 */
private static void writeToFile(String fileName, String fileContents) {
    System.out.print("Writing to file: " + fileName);
    try (FileWriter fileWriter = new FileWriter(fileName); BufferedWriter bw = new BufferedWriter(fileWriter)) {
        bw.write(fileContents);
        System.out.println(" | COMPLETE");
    } catch (Exception e) {
        System.out.println("\nWriting Failed");
        e.printStackTrace();
    }
}


/**
 * Displays the first desired number of data elements from the data set and the data descriptor.
 *
 * @param dataSet Data set to be displayed
 * @param dataDescriptor Data descriptor to be displayed
 */
private static void displayDataSet(ArrayList<DataElement> dataSet, DataDescriptor dataDescriptor) {

    // Set the number of elements to show to 10 or less depending on the data set
    int numberOfElementsToShow = 10;
    if(numberOfElementsToShow > dataSet.size()) {

        numberOfElementsToShow = dataSet.size();
    }

    StringBuilder s = new StringBuilder();

    //Check if the data descriptor is null and add the output if it is not
```

```
        if (dataDescriptor != null) {

            s.append(dataDescriptor.toString());
        }

        // Print out the desired number of rows of data
        s.append("\n\nData output Actual:\n");
        for (int i = 0; i < numberOfElementsToShow; i++) {

            s.append(dataSet.get(i).toStringInts()).append("\n");
        }

        // Print out the desired number of rows of data
        s.append("\n\nData output String conversion:\n");
        for (int i = 0; i < numberOfElementsToShow; i++) {

            s.append(dataSet.get(i).toString()).append("\n");
        }

        System.out.println(s.toString());
    }

}
```

DataParser:

```
package com.riaanvo;

import java.util.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

/**
 * This class is used to read in data sets in the comma separated value (CSV) file type and create a data descriptor
 * and a list of data elements. It can be used to extract data and generate a data descriptor or use a provided
 * data descriptor.
 */
public class DataParser {

   private DataDescriptor dataDescriptor;
   private ArrayList<DataElement> dataSet;


   public DataParser() {}

   /**
    * Takes in the file path of the data set and the data descriptor used
    * to convert the string values to integers. If no data descriptor is provided, one will be created.
    *
    * @param filePath       The file path of the data set
    * @param dataDescriptor The data descriptor to decode the data sets values
    */
   public void parseData(String filePath, DataDescriptor dataDescriptor){

      this.dataDescriptor = dataDescriptor;
      long previousTime = System.currentTimeMillis();

      System.out.println("File to load: " + filePath);
      System.out.print("Loading data set:");

      // Extract out the contents of the file
      String trainingCSVText = extractFileContents(filePath);

      System.out.println("\t| TIME TAKEN: " + (System.currentTimeMillis() - previousTime) + "ms");
      previousTime = System.currentTimeMillis();
      System.out.print("Extracting data:");

      //Split into rows and extract the data elements
      String[] rows = trainingCSVText.split("\n");
      extractDataSet(rows, dataDescriptor);

      System.out.println("\t| TIME TAKEN: " + (System.currentTimeMillis() - previousTime) + "ms\n");

   }


   /**
    * Extracts the contents of a file into a single string which is returned.
    *
```

```
 * @param filepath File path of the CSV file to extract
 * @return A string containing all the CSV files contents
 */
private String extractFileContents(String filepath) {

   StringBuilder output = new StringBuilder();

   // Attempt to open the file and read the data
   try (FileReader fr = new FileReader(filepath); BufferedReader br = new BufferedReader(fr)) {

      String nextLine;
      // Add the next line to the output string if it
      while ((nextLine = br.readLine()) != null) {
         output.append(nextLine).append("\n");
      }

   } catch (IOException e) {
      e.printStackTrace();
   }

   return output.toString();
}

/**
 * Converts an array of string rows to a list of data elements which can be used for data mining.
 *
 * @param rows          An array of data values in string CSV form
 * @param dataDescriptor The data descriptor used to convert the data values
 */
private void extractDataSet(String[] rows, DataDescriptor dataDescriptor) {

   dataSet = new ArrayList<>();

   // Check if the data descriptor was given and create a new one if not
   boolean dataDescriptorPredefined = true;
   if (dataDescriptor == null) {

      this.dataDescriptor = new DataDescriptor(rows[0].split(","));
      dataDescriptorPredefined = false;
   }

   //Create all the data elements
   for (int r = 1; r < rows.length; r++) {

      //Split into the individual values
      String[] values = rows[r].split(",");

      // Don't try to add the value to the attributes unique values if the descriptor exists
      if (!dataDescriptorPredefined) {

         //Create the unique values  Array List
         for (int c = 0; c < values.length; c++) {
            this.dataDescriptor.tryAddUniqueValue(c, values[c]);
         }
      }
```

```
            //Create a new dataElement
            dataSet.add(new DataElement(this.dataDescriptor.convertStringValuesToInt(values)));
        }

        //Set all elements data descriptors to this
        DataElement.setDataDescriptor(this.dataDescriptor);
    }

    /**
     * Returns the data descriptor used for this data parser
     *
     * @return The data descriptor
     */
    public DataDescriptor getDataDescriptor() {

        return dataDescriptor;
    }

    /**
     * Returns the list of data elements extracted from the CSV.
     *
     * @return Arraylist of data elements
     */
    public ArrayList<DataElement> getDataSet() {

        return dataSet;
    }
}
```

DataPreprocessor:

```
package com.riaanvo;

import java.util.ArrayList;

/**
 * This class is used to pre-process the data sets uses to build and test the model. The current functionality is to
 * convert categorical data into binary attributes for simpler decisions with only true and false values.
 */
public class DataPreprocessor {

   private DataDescriptor dataDescriptor;
   private ArrayList<DataElement> dataSet;

   public DataPreprocessor() {
   }

   /**
    * Takes in an old data set and data descriptor and creates a new binarised data descriptor and corresponding data
    * set. These new objects can be accessed using getters.
    *
    * @param oldDataSet       The original data set
    * @param oldDataDescriptor The original data descriptor
    */
   public void binariseDataSet(ArrayList<DataElement> oldDataSet, DataDescriptor oldDataDescriptor) {

      // Store the current time for duration calculations
      long previousTime = System.currentTimeMillis();

      // If the data descriptor does not exist, create and define a new binarised data descriptor
      if (dataDescriptor == null) {

         System.out.print("Defining new data descriptor:");

         dataDescriptor = new DataDescriptor(createNewHeaders(oldDataDescriptor));
         enterUniqueValues(oldDataDescriptor);

         System.out.println("\t| TIME TAKEN: " + (System.currentTimeMillis() - previousTime) + "ms");
      }

      previousTime = System.currentTimeMillis();
      System.out.print("Converting data set:");

      // Convert the old data set and store it
      dataSet = convertToNewDataSet(oldDataDescriptor, oldDataSet);

      System.out.println("\t| TIME TAKEN: " + (System.currentTimeMillis() - previousTime) + "ms\n");
   }

   /**
    * Takes in a data descriptor and for every categorical attribute it creates binarised headers for every unique
    * value.
    *
    * @param baseDataDescriptor Original data descriptor
    * @return Attribute names for a new data descriptor
```

```
   */
  private String[] createNewHeaders(DataDescriptor baseDataDescriptor) {

    // Count the new number of attributes the data descriptor will have
    int totalNumAttributes = 0;
    for (int i = 0; i < baseDataDescriptor.getNumberOfAttributes(); i++) {

      // If the current attribute is the class, only at a single attribute to the headers
      if (i == baseDataDescriptor.getClassAttributeIndex()) {

        totalNumAttributes++;
        continue;
      }

      // Add the number of unique values as each will become an attribute itself
      totalNumAttributes += baseDataDescriptor.getUniqueAttributeValues(i).size();
    }

    String[] newHeaders = new String[totalNumAttributes];

    // Loop through each base attribute and create new headers for the new binarised attributes
    int currentIndex = 0;
    for (int a = 0; a < baseDataDescriptor.getNumberOfAttributes(); a++) {

      // If the attribute is the class, leave the same name, else add all values as new attributes
      if (a == baseDataDescriptor.getClassAttributeIndex()) {

        newHeaders[currentIndex] = baseDataDescriptor.getAttribute(a);
        currentIndex++;
      } else {

        // Loop through all the base attributes values and create new specific headers
        for (int ua = 0; ua < baseDataDescriptor.getUniqueAttributeValues(a).size(); ua++) {

          newHeaders[currentIndex] = baseDataDescriptor.getAttribute(a) + " = " +
baseDataDescriptor.getUniqueAttributeValues(a).get(ua);
          currentIndex++;
        }
      }
    }

    return newHeaders;
  }

  /**
   * Sets up the unique data values for a binarised data set in the data descriptor
   *
   * @param oldDataDescriptor The previous data descriptor
   */
  private void enterUniqueValues(DataDescriptor oldDataDescriptor) {

    // Loop through all the attributes in the new data descriptor
    for (int a = 0; a < dataDescriptor.getNumberOfAttributes(); a++) {

      // If the attribute index is the same as the old data descriptors class index
```

```
        if
(dataDescriptor.getAttribute(a).equals(oldDataDescriptor.getAttribute(oldDataDescriptor.getClassAttributeIndex())))
{

            // Add all the class values to the attribute
            for (String value :
oldDataDescriptor.getUniqueAttributeValues(oldDataDescriptor.getClassAttributeIndex())) {

                dataDescriptor.tryAddUniqueValue(a, value);
            }
        } else {

            // Add false and true as the unique values
            dataDescriptor.tryAddUniqueValue(a, "false");
            dataDescriptor.tryAddUniqueValue(a, "true");
        }
    }
  }

  /**
   * Takes in a data descriptor and the old data set and uses the new data descriptor to convert the data elements
   * into values for the new data descriptor.
   *
   * @param baseDataDescriptor The original data descriptor
   * @param oldDataElements    The original data set
   * @return The binarised data set
   */
  private ArrayList<DataElement> convertToNewDataSet(DataDescriptor baseDataDescriptor,
ArrayList<DataElement> oldDataElements) {

    // Loop through the old data descriptor and count the number of values in each attribute
    int[] attributeUniqueNumbers = new int[baseDataDescriptor.getNumberOfAttributes()];
    for (int i = 0; i < baseDataDescriptor.getNumberOfAttributes(); i++) {

      // If it is the class attribute
      if (i == baseDataDescriptor.getClassAttributeIndex()) {

        attributeUniqueNumbers[i] = 1;
        continue;
      }
      attributeUniqueNumbers[i] = baseDataDescriptor.getUniqueAttributeValues(i).size();
    }

    // Create a list to hold the new data set
    ArrayList<DataElement> newDataSet = new ArrayList<DataElement>();

    // Store local values to reduce outer calls
    int newNumberOfAttributes = dataDescriptor.getNumberOfAttributes();
    int oldNumberOfAttributes = baseDataDescriptor.getNumberOfAttributes();
    int oldClassIndex = baseDataDescriptor.getClassAttributeIndex();

    // Loop through each data element and convert the values to the new data descriptor
    for (DataElement dataElement : oldDataElements) {

      // Create an array to store the new values of this data element
      int[] newValues = new int[newNumberOfAttributes];
```

```
        // Loop through the older data values and insert them into the new data values
        int currentIndex = 0;
        for (int a = 0; a < oldNumberOfAttributes; a++) {

          // If the current value is the class value, store that value
          if (a == oldClassIndex) {

            newValues[currentIndex] = dataElement.getValue(a);
          } else {

            // Set the value in the correct spot to 1 to signify true
            newValues[currentIndex + dataElement.getValue(a)] = 1;
          }

          // Increment by the number of vales in the old attribute
          currentIndex += attributeUniqueNumbers[a];
        }

        // Create a new data element and store it in the new data set
        newDataSet.add(new DataElement(newValues));
      }

      // Set the data descriptor for the data elements
      DataElement.setDataDescriptor(dataDescriptor);

      return newDataSet;
    }

    /**
     * Returns the data descriptor used for this data parser
     *
     * @return The data descriptor
     */
    public DataDescriptor getDataDescriptor() {

      return dataDescriptor;
    }

    /**
     * Returns the list of data elements extracted from the CSV.
     *
     * @return Arraylist of data elements
     */
    public ArrayList<DataElement> getDataSet() {

      return dataSet;
    }
}
```

DataDescriptor:

```
package com.riaanvo;

import java.util.ArrayList;
import java.util.Collections;

/**
 * Data Descriptor defines the current structure of the inputted data set. It allows for the conversion of string values
 * into integer values for faster comparisons and smaller storage requirements.
 */
public class DataDescriptor {

   private final ArrayList<String> attributes;
   private final ArrayList<ArrayList<String>> uniqueAttributeValues;

   private int classAttributeIndex = 0;
   private int numberOfClasses = 0;

   /**
    * Constructor for creating a data descriptor object. The data descriptor will define the structure of the supplied
    * data set.
    *
    * @param attributes Data sets attributes
    */
   public DataDescriptor(String[] attributes) {

      // Create a list of all the attribute headers and store the passed attribute name values
      this.attributes = new ArrayList<>();
      Collections.addAll(this.attributes, attributes);

      // Create arrays to store the unique data values found in the data set
      uniqueAttributeValues = new ArrayList<>();
      for (int i = 0; i < attributes.length; i++) {

         uniqueAttributeValues.add(new ArrayList<>());

         // Assign the attribute index if it is denoted with a '#'
         if (attributes[i].contains("#")) {
            classAttributeIndex = i;
         }
      }
   }

   /**
    * Attempts to add the value to the respective attribute values list to build a unique list of attribute values.
    *
    * @param attributeIndex Column index of the attribute in the data set
    * @param value          Value for that attribute
    */
   public void tryAddUniqueValue(int attributeIndex, String value) {

      // Check if the unique attribute list contains the value and add if not
      if (!uniqueAttributeValues.get(attributeIndex).contains(value)) {
         uniqueAttributeValues.get(attributeIndex).add(value);
      }
```

```
}

/**
 * Converts the string values into a list of integer indexes for faster comparisons.
 *
 * @param attributeValues The list of attribute values for the data element
 * @return A integer list of attribute values based of value indexes
 */
public int[] convertStringValuesToInt(String[] attributeValues) {

   // Create an integer array to store all the value indexes
   int[] values = new int[attributeValues.length];

   // Determine the index of the value based on the unique attribute values
   for (int i = 0; i < values.length; i++) {
      values[i] = uniqueAttributeValues.get(i).indexOf(attributeValues[i]);
   }

   return values;
}

/**
 * Converts an attribute index and attribute value index into the unique string value for that attribute.
 *
 * @param attributeIndex      The attribute index from the data set
 * @param attributeValueIndex The attribute value as an index
 * @return The attribute value as a string
 */
public String indexToValue(int attributeIndex, int attributeValueIndex) {

   return uniqueAttributeValues.get(attributeIndex).get(attributeValueIndex);
}

/**
 * Returns the unique list of attribute values for the desired attribute
 *
 * @param attributeIndex The index of the attribute
 * @return The list of unique attribute values
 */
public ArrayList<String> getUniqueAttributeValues(int attributeIndex) {

   return uniqueAttributeValues.get(attributeIndex);
}

/**
 * Used to get the string value of an attribute based on index. Returns a blank string if there is no
 * attribute index (Value '-1').
 *
 * @param attributeIndex The index of the desired attribute
 * @return The string name of the attribute
 */
public String getAttribute(int attributeIndex) {

   // Check if the index is for no attribute and return a blank string if it is
   if (attributeIndex == -1) return "";
   return attributes.get(attributeIndex);
```

```
}

/**
 * Returns the number of attributes in the data set. Includes the class attribute.
 *
 * @return The number of attributes contained in the data set
 */
public int getNumberOfAttributes() {

   return attributes.size();
}

/**
 * Returns the index of the data sets class attribute
 *
 * @return The index of the class attribute
 */
public int getClassAttributeIndex() {

   return classAttributeIndex;
}

/**
 * Determines the number of classes in the data set and returns the value.
 *
 * @return The number of classes in the data set
 */
public int getNumberOfClasses() {

   // Check if the number of classes has been defined and count the values if not
   if (numberOfClasses == 0) {

      numberOfClasses = uniqueAttributeValues.get(classAttributeIndex).size();
   }

   return numberOfClasses;
}

/**
 * Conversion of the data descriptor values to a string of information
 *
 * @return A string of the data descriptor values
 */
public String toString() {

   StringBuilder s = new StringBuilder();
   for (int c = 0; c < attributes.size(); c++) {

      // Added that attribute name to the output
      s.append(attributes.get(c)).append(":\n");

      // Add each unique value for that attribute to the output
      ArrayList<String> uniqueValues = uniqueAttributeValues.get(c);
      for (int v = 0; v < uniqueValues.size(); v++) {

         s.append("\t").append(v).append(": ").append(uniqueValues.get(v)).append("\n");
```

```
            }
            s.append("\n");
        }
        return s.toString();
    }
}
```

DataElement:

```
package com.riaanvo;

import java.util.ArrayList;

/**
 * Java object for storing a data elements values.
 */
public class DataElement {

   private static DataDescriptor dataDescriptor;
   private final ArrayList<Integer> values;

   /**
    * Constructor for creating a data element object. Stores all the values as indexes for a single row in
    * the data set.
    *
    * @param values List of value indexes for this data element
    */
   public DataElement(int[] values) {

      //Create the values list and store all the value indexes
      this.values = new ArrayList<>();
      for (int value : values) {

         this.values.add(value);
      }
   }

   /**
    * Sets the global Data Descriptor for all Data elements
    *
    * @param newDataDescriptor Data descriptor that is used for all data elements
    */
   public static void setDataDescriptor(DataDescriptor newDataDescriptor) {

      dataDescriptor = newDataDescriptor;
   }

   /**
    * This returns the data descriptor used for all data elements
    * @return The data descriptor used for data elements
    */
   public static DataDescriptor getDataDescriptor() {

      return dataDescriptor;
   }

   /**
    * Gets the index value for the desired attribute index
    * @param attributeIndex Attribute index of the desired value
    * @return The index value for that attribute
    */
   public int getValue(int attributeIndex){

      return values.get(attributeIndex);
```

```
    }

    /**
     * Returns a string list of all the value indexes stored in this data element.
     *
     * @return String of data value indexes
     */
    public String toStringInts() {

        StringBuilder s = new StringBuilder();

        // Determine the number of values and loop through them
        int numValues = values.size();
        for (int i = 0; i < numValues; i++) {

            s.append(values.get(i));

            // Add a comma if it is not the last value
            if (i != numValues - 1) {
                s.append(", ");
            }
        }

        return s.toString();
    }

    /**
     * Returns the data elements values as a string of attribute values separated by commas
     *
     * @return A data element as a string of values
     */
    public String toString() {

        StringBuilder s = new StringBuilder();

        // Determine the number of values and loop through them
        int numValues = values.size();
        for (int i = 0; i < numValues; i++) {

            s.append(dataDescriptor.indexToValue(i, values.get(i)));

            // Add a comma if it is not the last value
            if (i != numValues - 1) {
                s.append(", ");
            }
        }

        return s.toString();
    }

}
```

ID3:

```
package com.riaanvo;

import java.util.ArrayList;

/**
 * This class contains the ID3 model and allows the testing and printing of the model.
 */
public class ID3 {

   //Used for determining the decision tree structure
   private int currentNodeIndex = 0;
   private final int maxNodeDepth;

   private final DataDescriptor dataDescriptor;
   private Node rootNode;
   private final String decFormat = "%.3f";

   /**
    * Constructor for the ID3 model. It takes in the data descriptor for the data set as well as the training data,
    * it then creates the model and stores the structure.
    * @param trainingData Data set to build the model with
    * @param maxNodeDepth The max node depth of the tree
    */
   public ID3(ArrayList<DataElement> trainingData, int maxNodeDepth) {

      this.dataDescriptor = DataElement.getDataDescriptor();
      this.maxNodeDepth = maxNodeDepth;
      buildModel(trainingData);
   }

   /**
    * This method builds the ID3 model using the provided training data.
    *
    * @param trainingData List of data elements in the training data set
    */
   private void buildModel(ArrayList<DataElement> trainingData) {

      // Store the starting time of model construction
      long previousTime = System.currentTimeMillis();
      System.out.print("Building ID3 Tree:");

      // Reset node indexing if a new model is built
      currentNodeIndex = 0;

      // Create a string to contain indicators of which attributes have been used (Not needed for categorical data)
      StringBuilder attributesLeft = new StringBuilder();
      for (int i = 0; i < dataDescriptor.getNumberOfAttributes(); i++) {

         // If the index is the class attribute set it to one
         if (dataDescriptor.getClassAttributeIndex() == i) {
            attributesLeft.append("1");
         } else {
            attributesLeft.append("0");
         }
      }
```

```
    // Create and store the root node of the model. This will recursively construct the decision tree
    rootNode = new Node(trainingData, attributesLeft.toString(), 0);

    System.out.println("\t| TIME TAKEN: " + (System.currentTimeMillis() - previousTime) + "ms\n");
}

/**
 * Creates a string containing the script used to display a decision tree model using graphviz.
 *
 * @return The ID3 model as a script
 */
public String createTreeDiagramScript(boolean showEmptyLeaves) {

    String s = "";
    s += "digraph Tree {\nnode [shape=box, style=\"filled\", color=\"black\"];\n";
    s += rootNode.toString(showEmptyLeaves);
    s += "}\n";
    return s;
}

/**
 * Tests the model with the provided data set and returns a string containing the confusion matrix and
 * accuracy statistics based on the test data set.
 *
 * @param testDataSet List of data elements
 * @return A string containing the test results
 */
public String testModel(ArrayList<DataElement> testDataSet) {

    // Store the start time of testing
    long previousTime = System.currentTimeMillis();
    System.out.print("Testing model:");

    int numberOfClasses = dataDescriptor.getNumberOfClasses();
    int classIndex = dataDescriptor.getClassAttributeIndex();

    // Loop through all the data elements and increment counters in the confusion matrix
    int[][] confusionMatrix = new int[numberOfClasses][numberOfClasses];
    for (DataElement dataElement : testDataSet) {
        confusionMatrix[dataElement.getValue(classIndex)][rootNode.determineClass(dataElement)]++;
    }

    // Display the time taken to test the data set
    System.out.println("\t| TIME TAKEN: " + (System.currentTimeMillis() - previousTime) + "ms");

    StringBuilder s = new StringBuilder();
    s.append("Number of samples: ").append(testDataSet.size()).append("\n");

    // Display the confusion matrix
    s.append("True Class \\ Predicted Class\n \\\t");

    // Display hypothesis cases
    for (int h = 0; h < confusionMatrix.length; h++) {
        s.append("H").append(h).append("\t");
    }
```

```
        s.append("\n");

        // Display the confusion matrix values
        for (int r = 0; r < confusionMatrix.length; r++) {
            s.append("H").append(r).append("\t");

            for (int c = 0; c < confusionMatrix[0].length; c++) {
                s.append(confusionMatrix[r][c]).append("\t");
            }

            if (r < confusionMatrix.length - 1) {
                s.append("\n");
            }
        }

        // If there are only two classes display the accuracy statistics
        if (dataDescriptor.getNumberOfClasses() == 2) {
            s.append("\n\nAccuracy Statistics:");

            double accuracy = (double) (confusionMatrix[0][0] + confusionMatrix[1][1]) / (double) testDataSet.size() *
100;
            s.append("\nAccuracy: ").append(String.format(decFormat, accuracy)).append("%");

            double errorRate = (double) (confusionMatrix[0][1] + confusionMatrix[1][0]) / (double) testDataSet.size() *
100;
            s.append("\nError Rate: ").append(String.format(decFormat, errorRate)).append("%");

            double FAR = (double) (confusionMatrix[0][1]) / (double) (confusionMatrix[0][1] + confusionMatrix[0][0]) *
100;
            s.append("\nFalse Alarm Rate: ").append(String.format(decFormat, FAR)).append("%");

            double DR = (double) (confusionMatrix[1][1]) / (double) (confusionMatrix[1][1] + confusionMatrix[1][0]) *
100;
            s.append("\nDetection Rate: ").append(String.format(decFormat, DR)).append("%");

            double precision = (double) (confusionMatrix[1][1]) / (double) (confusionMatrix[0][1] + confusionMatrix[1][1])
* 100;
            s.append("\nPrecision: ").append(String.format(decFormat, precision)).append("%");

            double recall = DR;
            s.append("\nRecall: ").append(String.format(decFormat, recall)).append("%");

            double F1 = 2 * precision * recall / (precision + recall);
            s.append("\nF1 score: ").append(String.format(decFormat, F1)).append("%");
        }

        return s.toString();
    }

    /**
     * Takes in a data set and predicts the class for each data value
     *
     * @param dataSet List of data elements
     * @return A string containing all the predicted values
     */
    public String predictClasses(ArrayList<DataElement> dataSet) {
```

```
        StringBuilder s = new StringBuilder();
        s.append("Predicted Classes:\n");

        long previousTime = System.currentTimeMillis();
        System.out.print("Predicting Classes:");

        for (DataElement dataElement : dataSet) {
            String value = dataDescriptor.indexToValue(dataDescriptor.getClassAttributeIndex(),
rootNode.determineClass(dataElement));
            s.append(value).append("\n");
        }

        // Display the time taken to predict the data set
        System.out.println("\t| TIME TAKEN: " + (System.currentTimeMillis() - previousTime) + "ms");

        return s.toString();
    }
```

Node:

```
  /**
   * The java object that defines the nodes of a decision tree. When a node is created it will try to recursively add
   * sub nodes to build a decision tree until there are no more samples left undefined or there are no more
attributes
   * to break the data up into.
   * <p>
   * This class provides the implementation of the following algorithm
   * ID3 (Examples, Target_Attribute, Attributes)
   * Create a root node for the tree
   * If all examples are positive, Return the single-node tree Root, with label = +.
   * If all examples are negative, Return the single-node tree Root, with label = -.
   * If number of predicting attributes is empty, then Return the single node tree Root,
   * with label = most common value of the target attribute in the examples.
   * Otherwise Begin
   * A ← The Attribute that best classifies examples.
   * Decision Tree attribute for Root = A.
   * For each possible value, vi, of A,
   * Add a new tree branch below Root, corresponding to the test A = vi.
   * Let Examples(vi) be the subset of examples that have the value vi for A
   * If Examples(vi) is empty
   * Then below this new branch add a leaf node with label = most common target value in the examples
   * Else below this new branch add the subtree ID3 (Examples(vi), Target_Attribute, Attributes – {A})
   * End
   * Return Root
   */
  private class Node {

    private final int nodeIndex;
    private int attributeSplitIndex = -1;
    private double informationGain = 0;
    private int mostCommonClass;

    private double currentEntropy = 0;

    private int sampleCount = 0;
    private int[] classCounts;
    private ArrayList<Node> subNodes;

    /**
     * Constructor for creating a node in a decision tree. Functions recursively and will create sub nodes until all
     * data is split or there are no more attributes to split on.
     *
     * @param samples       List of data elements
     * @param attributesLeft String defining which attributes can be used to split the data
     */
    public Node(ArrayList<DataElement> samples, String attributesLeft, int nodeDepth) {

      // Self assign a node index and increment the value
      nodeIndex = currentNodeIndex++;

      // Get sample count and stop if there are no samples
      sampleCount = samples.size();
      if (sampleCount == 0) return;

      // Count the number of data elements for each class value
```

```
    classCounts = extractClassCounts(samples);

    // Determine the class with the most number of samples
    mostCommonClass = determineMostCommonClass();

    // Determine the current sample entropy for this node
    currentEntropy = calculateEntropy(samples);

    // Check if this node is a single class and stop recursion
    if (isSingleClass()) return;

    // Stop if the max node depth is reached
    if(nodeDepth >= maxNodeDepth && maxNodeDepth != -1) return;

    constructSubNodes(samples, attributesLeft, nodeDepth);
}

/**
 * Extracts the current number of each class that is present in the given sample.
 *
 * @param samples A list of data elements
 * @return A array of integers containing class value counts
 */
private int[] extractClassCounts(ArrayList<DataElement> samples) {

    int classAttributeIndex = dataDescriptor.getClassAttributeIndex();

    // Count the number of data elements for each class value
    int[] counts = new int[dataDescriptor.getNumberOfClasses()];
    for (DataElement dataElement : samples) {

        counts[dataElement.getValue(classAttributeIndex)]++;
    }

    return counts;
}

/**
 * Returns the class value with the largest number of data samples in it.
 *
 * @return The value index for the most common class
 */
private int determineMostCommonClass() {

    //Determine the current class with the most values
    int currentClassIndex = 0;
    for (int i = 0; i < classCounts.length; i++) {

        //Check if this class has a higher count
        if (classCounts[i] > classCounts[currentClassIndex]) {

            currentClassIndex = i;
        }
    }

    return currentClassIndex;
```

```
}

/**
 * Determines if all the sample data elements are of a single class.
 *
 * @return If the samples are all a single class
 */
private boolean isSingleClass() {

   boolean isSingleClass = false;
   for (int classCount : classCounts) {

      // Check if the number of data elements in this class is not zero
      if (classCount != 0) {

         // Check if there is already another class with more than one sample
         if (!isSingleClass) {
            isSingleClass = true;
         } else {
            isSingleClass = false;
            break;
         }
      }
   }

   return isSingleClass;
}

/**
 * Creates the sub nodes for this node in the tree. Takes in the current list of samples and the attributes that
 * can be used to split the data and determines which attribute has the highest information gain.
 *
 * @param samples       List of data elements to be split
 * @param attributesLeft A string describing the attributes that can be used to split the data
 */
private void constructSubNodes(ArrayList<DataElement> samples, String attributesLeft, int currentNodeDepth) {

   int largestInfoGainAttributeIndex = 0;
   double currentLargestInfoGain = Double.MIN_VALUE;

   // If there are no more attributes to split on, stop
   if (!attributesLeft.contains("0")) return;

   for (int i = 0; i < dataDescriptor.getNumberOfAttributes(); i++) {

      // Don't attempt to check information gain on a previously used attribute
      if (attributesLeft.charAt(i) == '1') continue;

      // Calculate information gain for this attribute index
      double infoGain = calculateInformationGain(samples, i);

      // Check if the new information gain is larger than before
      if (infoGain > currentLargestInfoGain) {
         currentLargestInfoGain = infoGain;
         largestInfoGainAttributeIndex = i;
      }
```

```
        }

        // Store the information gain for this attribute split
        informationGain = currentLargestInfoGain;

        // Use the attribute which gives the largest information gain to split the data
        attributeSplitIndex = largestInfoGainAttributeIndex;

        // Mark the split attribute as used
        StringBuilder newAttributesLeft = new StringBuilder(attributesLeft);
        newAttributesLeft.setCharAt(attributeSplitIndex, '1');

        // Break up the samples based on that attribute
        ArrayList<ArrayList<DataElement>> valueSubsets = getSubSets(samples, attributeSplitIndex);

        // For each subset of the samples create a new node
        subNodes = new ArrayList<>();
        for (ArrayList<DataElement> valueSubset : valueSubsets) {
            Node node = new Node(valueSubset, newAttributesLeft.toString(), currentNodeDepth + 1);

            // If there are no samples in the subset use the parents most common class value
            if (valueSubset.size() == 0) {
                node.setMostCommonClass(mostCommonClass);
            }

            subNodes.add(node);
        }
    }

    /**
     * This function calculates the information gain for splitting the samples using a specific attribute.
     *
     * @param samples       List of data elements
     * @param attributeIndex Attribute to split and calculate information gain
     * @return The information gain of this split
     */
    private double calculateInformationGain(ArrayList<DataElement> samples, int attributeIndex) {

        // Create subsets of the values based of the attribute to split on
        ArrayList<ArrayList<DataElement>> valueSubsets = getSubSets(samples, attributeIndex);

        // Sum up the weighted entropy for all the sub sets
        double subSetEntropySum = 0;
        for (ArrayList<DataElement> valueSubset : valueSubsets) {

            // If the sub set does not have any samples skip it
            if (valueSubset.size() == 0) continue;

            // Add the weighted entropy of the subset to the overall split entropy
            subSetEntropySum += ((double) valueSubset.size() / (double) sampleCount) *
calculateEntropy(valueSubset);
        }

        return currentEntropy - subSetEntropySum;
    }
```

```
/**
 * Calculates the current entropy of the provided data set based on the class attribute.
 *
 * @param samples List of data elements
 * @return The entropy of the data set
 */
private double calculateEntropy(ArrayList<DataElement> samples) {

   // Determine the number of samples in this data set
   int sampleSize = samples.size();

   // Create an array to store the class counts for this data set
   int[] classValueCounts = extractClassCounts(samples);

   // Sum up the entropy for this sample
   double ent = 0;
   for (int classValueCount : classValueCounts) {

      // Calculate the probability of this class
      double p_ = (double) classValueCount / (double) sampleSize;

      // If the probability is zero skip this value to prevent math errors
      if (p_ == 0) continue;

      // Calculate the entropy to be added (log base conversion is used to get log base 2)
      ent += -p_ * (Math.log(p_)/Math.log(2));
   }

   return ent;
}

/**
 * Returns a list of split data sets based on the provided attribute.
 *
 * @param samples       List of data samples to split
 * @param attributeIndex Attribute to split the data set on
 * @return A split list of data samples
 */
private ArrayList<ArrayList<DataElement>> getSubSets(ArrayList<DataElement> samples, int attributeIndex) {

   // Create a list to hold all the sub lists
   ArrayList<ArrayList<DataElement>> valueSubsets = new ArrayList<>();
   for (int i = 0; i < dataDescriptor.getUniqueAttributeValues(attributeIndex).size(); i++) {
      valueSubsets.add(new ArrayList<>());
   }

   // Populate the sub lists with the data elements
   for (DataElement dataElement : samples) {
      valueSubsets.get(dataElement.getValue(attributeIndex)).add(dataElement);
   }

   return valueSubsets;
}

/**
 * Determines the class value for the provided data element based on the tree structure. Functions recursively
```

```
     * and will go through the tree until there are no more sub nodes.
     *
     * @param dataElement The data element to be classified
     * @return The class classification for this data element
     */
    private int determineClass(DataElement dataElement) {

        // If this is a leaf node return the most common class value
        if (subNodes == null) return mostCommonClass;

        // Use the sub nodes to determine the most common class value
        return subNodes.get(dataElement.getValue(attributeSplitIndex)).determineClass(dataElement);
    }

    /**
     * Sets the most common class value. Used for sub nodes with no samples to determine themselves.
     *
     * @param mostCommonClass The most common class value
     */
    private void setMostCommonClass(int mostCommonClass) {
        this.mostCommonClass = mostCommonClass;
    }

    /**
     * Gets the node index used for defining the tree structure.
     *
     * @return Node index
     */
    private int getNodeIndex() {

        return nodeIndex;
    }

    /**
     * Gets the sample count at this node.
     *
     * @return Sample count for this node
     */
    private int getSampleCount() {

        return sampleCount;
    }

    /**
     * Returns the string describing the node and its connections to the sub nodes.
     *
     * @return A string structure of the node and sub nodes
     */
    public String toString(boolean showEmptyLeaves) {

        StringBuilder s = new StringBuilder();

        // Add all the information describing this node to the string
        s.append(nodeIndex).append(" [label=\"");

        // If this node is a leaf node don't display split and information gain
```

```
        if (attributeSplitIndex != -1) {

            s.append("Split on: ").append(dataDescriptor.getAttribute(attributeSplitIndex)).append("\\n");
            s.append("Information gain = ").append(String.format(decFormat, informationGain)).append("\\n");
        }
        s.append("Current Entropy = ").append(String.format(decFormat, currentEntropy)).append("\\n");
        s.append("Samples: ").append(sampleCount).append("\\n");
        s.append("Class counts: [");

        if(classCounts != null) {
            for (int i = 0; i < classCounts.length; i++) {

                s.append(classCounts[i]);

                // If this is not the last class count separate it by a ','
                if (i < classCounts.length - 1) {
                    s.append(", ");
                }
            }
        }
        s.append("]\\n");
        s.append("Class:
").append(dataDescriptor.getUniqueAttributeValues(dataDescriptor.getClassAttributeIndex()).get(mostCommonClas
s)).append("\"");
        s.append(", fillcolor=\"#" + 11111103 + "\"];\n");

        // Add all sub nodes to the string including their node connections
        if (subNodes != null) {
            for (int i = 0; i < subNodes.size(); i++) {

                // If the sub node sample count is 0 do not include it in the print out
                if (subNodes.get(i).getSampleCount() == 0 && !showEmptyLeaves) continue;

                // Add the node linking description
                s.append(getNodeIndex()).append(" -> ").append(subNodes.get(i).getNodeIndex());

s.append("[label=\"").append(dataDescriptor.getUniqueAttributeValues(attributeSplitIndex).get(i)).append("\"]").ap
pend(";");
                s.append("\n");

                // Add the sub nodes text
                s.append(subNodes.get(i).toString(showEmptyLeaves));
            }
        }

        return s.toString();
    }
  }
}
```

Appendix 2: Program output

File to load: mushroom-train(1).csv
Loading data set: | TIME TAKEN: 40ms
Extracting data: | TIME TAKEN: 96ms

File to load: mushroom-test(1).csv
Loading data set: | TIME TAKEN: 6ms
Extracting data: | TIME TAKEN: 9ms

File to load: mushroom-test(1).csv
Loading data set: | TIME TAKEN: 2ms
Extracting data: | TIME TAKEN: 2ms

Building ID3 Tree: | TIME TAKEN: 68ms

```
digraph Tree {
node [shape=box, style="filled", color="black"];
0 [label="Split on: odor\nInformation gain = 0.907\nCurrent Entropy = 0.999\nSamples: 7312\nClass counts: [3787, 3525]\nClass: edible", fillcolor="#11111103"];
0 -> 1[label="almond"];
1 [label="Current Entropy = 0.000\nSamples: 364\nClass counts: [364, 0]\nClass: edible", fillcolor="#11111103"];
0 -> 2[label="none"];
2 [label="Split on: spore-print-color\nInformation gain = 0.147\nCurrent Entropy = 0.213\nSamples: 3169\nClass counts: [3062, 107]\nClass: edible", fillcolor="#11111103"];
2 -> 3[label="black"];
3 [label="Current Entropy = 0.000\nSamples: 1154\nClass counts: [1154, 0]\nClass: edible", fillcolor="#11111103"];
2 -> 4[label="brown"];
4 [label="Current Entropy = 0.000\nSamples: 1213\nClass counts: [1213, 0]\nClass: edible", fillcolor="#11111103"];
2 -> 5[label="chocolate"];
5 [label="Current Entropy = 0.000\nSamples: 44\nClass counts: [44, 0]\nClass: edible", fillcolor="#11111103"];
2 -> 6[label="white"];
6 [label="Split on: habitat\nInformation gain = 0.240\nCurrent Entropy = 0.370\nSamples: 563\nClass counts: [523, 40]\nClass: edible", fillcolor="#11111103"];
6 -> 7[label="grasses"];
7 [label="Current Entropy = 0.000\nSamples: 257\nClass counts: [257, 0]\nClass: edible", fillcolor="#11111103"];
6 -> 8[label="woods"];
8 [label="Split on: gill-size\nInformation gain = 0.787\nCurrent Entropy = 0.787\nSamples: 34\nClass counts: [8, 26]\nClass: poisonous", fillcolor="#11111103"];
8 -> 9[label="broad"];
9 [label="Current Entropy = 0.000\nSamples: 8\nClass counts: [8, 0]\nClass: edible", fillcolor="#11111103"];
8 -> 10[label="narrow"];
10 [label="Current Entropy = 0.000\nSamples: 26\nClass counts: [0, 26]\nClass: poisonous", fillcolor="#11111103"];
6 -> 11[label="paths"];
11 [label="Current Entropy = 0.000\nSamples: 37\nClass counts: [37, 0]\nClass: edible", fillcolor="#11111103"];
6 -> 12[label="leaves"];
12 [label="Split on: cap-color\nInformation gain = 0.797\nCurrent Entropy = 0.797\nSamples: 58\nClass counts: [44, 14]\nClass: edible", fillcolor="#11111103"];
```

```
12 -> 13[label="white"];
13 [label="Current Entropy = 0.000\nSamples: 8\nClass counts: [0, 8]\nClass: poisonous",
fillcolor="#11111103"];
12 -> 14[label="brown"];
14 [label="Current Entropy = 0.000\nSamples: 23\nClass counts: [23, 0]\nClass: edible",
fillcolor="#11111103"];
12 -> 17[label="yellow"];
17 [label="Current Entropy = 0.000\nSamples: 6\nClass counts: [0, 6]\nClass: poisonous",
fillcolor="#11111103"];
12 -> 21[label="cinnamon"];
21 [label="Current Entropy = 0.000\nSamples: 21\nClass counts: [21, 0]\nClass: edible",
fillcolor="#11111103"];
6 -> 23[label="waste"];
23 [label="Current Entropy = 0.000\nSamples: 177\nClass counts: [177, 0]\nClass: edible",
fillcolor="#11111103"];
2 -> 26[label="green"];
26 [label="Current Entropy = 0.000\nSamples: 67\nClass counts: [0, 67]\nClass: poisonous",
fillcolor="#11111103"];
2 -> 27[label="orange"];
27 [label="Current Entropy = 0.000\nSamples: 42\nClass counts: [42, 0]\nClass: edible",
fillcolor="#11111103"];
2 -> 29[label="buff"];
29 [label="Current Entropy = 0.000\nSamples: 45\nClass counts: [45, 0]\nClass: edible",
fillcolor="#11111103"];
2 -> 30[label="yellow"];
30 [label="Current Entropy = 0.000\nSamples: 41\nClass counts: [41, 0]\nClass: edible",
fillcolor="#11111103"];
0 -> 31[label="foul"];
31 [label="Current Entropy = 0.000\nSamples: 1959\nClass counts: [0, 1959]\nClass: poisonous",
fillcolor="#11111103"];
0 -> 32[label="spicy"];
32 [label="Current Entropy = 0.000\nSamples: 518\nClass counts: [0, 518]\nClass: poisonous",
fillcolor="#11111103"];
0 -> 33[label="pungent"];
33 [label="Current Entropy = 0.000\nSamples: 230\nClass counts: [0, 230]\nClass: poisonous",
fillcolor="#11111103"];
0 -> 34[label="anise"];
34 [label="Current Entropy = 0.000\nSamples: 361\nClass counts: [361, 0]\nClass: edible",
fillcolor="#11111103"];
0 -> 35[label="fishy"];
35 [label="Current Entropy = 0.000\nSamples: 511\nClass counts: [0, 511]\nClass: poisonous",
fillcolor="#11111103"];
0 -> 36[label="creosote"];
36 [label="Current Entropy = 0.000\nSamples: 169\nClass counts: [0, 169]\nClass: poisonous",
fillcolor="#11111103"];
0 -> 37[label="musty"];
37 [label="Current Entropy = 0.000\nSamples: 31\nClass counts: [0, 31]\nClass: poisonous",
fillcolor="#11111103"];
}
```

Testing model: | TIME TAKEN: 1ms

Number of samples: 812
True Class \ Predicted Class

\	H0	H1
H0	421	0
H1	0	391

Accuracy Statistics:
Accuracy: 100.000%
Error Rate: 0.000%
False Alarm Rate: 0.000%
Detection Rate: 100.000%
Precision: 100.000%
Recall: 100.000%
F1 score: 100.000%

Predicting Classes: | TIME TAKEN: 1ms
Predicted Classes:
poisonous
poisonous
poisonous
poisonous
poisonous
poisonous
poisonous
poisonous
poisonous
poisonous
edible
poisonous
edible
poisonous
edible
poisonous
poisonous
edible
edible
edible
poisonous
etc…