Faculty of Engineering and Technology
Artificial Intelligence (Comp 338)

Machine Learning

Project 2

Prepared by:
Raghad Hamdeh (1191093)
Ola Ja'afreh (1223192)

Supervised by:
Dr. Radi Jarar

2022-2023

Introduction

Machine learning and Artificial Intelligence (AI) are popular terms that have gained significant attention in recent years. Machine learning is a branch of AI that focuses on developing algorithms capable of improving predictions and outcomes without explicit programming. By analyzing input data through statistical analysis, machine learning algorithms aim to make accurate predictions within an acceptable range.

One prominent technique in machine learning is the use of decision trees. Decision trees are effective tools for understanding complex relationships within large datasets where the underlying structure is difficult to discern. The main objective of a decision tree is to classify observations by using their properties as questions, ultimately assigning them to specific classes. Decision trees fall under the category of supervised machine learning, where the relationship between input and output is explained in the training data. The tree structure consists of decision nodes, which represent the splitting of data based on specific parameters, and leaves, which denote the final decisions or outcomes.

Overall, decision trees offer an intuitive and interpretable approach to making predictions and classifying data. They provide a framework for systematically analyzing and understanding the patterns and relationships present in datasets, enabling informed decision-making in various domains.

In this project, we utilized the Weka library in Java, which is a widely used tool for data and business analytics. Weka provides a comprehensive set of machine learning algorithms and tools for data preprocessing, classification, regression, clustering, and more. It offers a user-friendly interface and a rich collection of functionalities to support data analysis and model development.

Procedure and Discussion

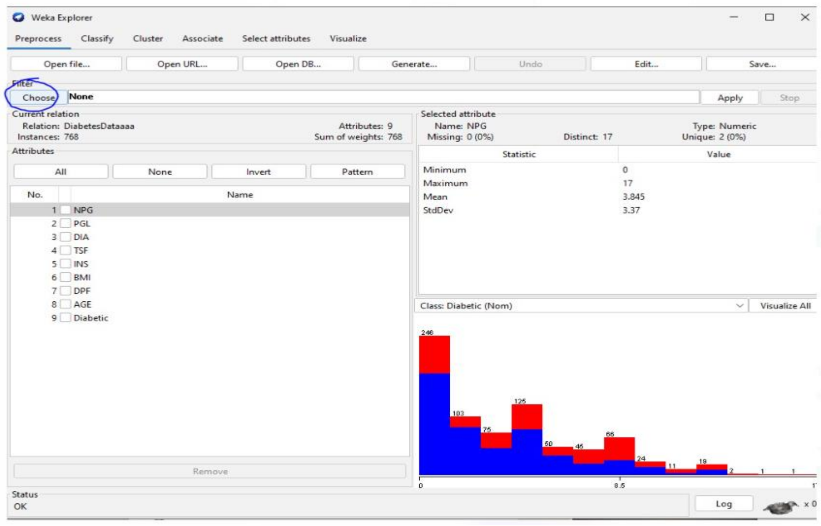
As the before first step in our data analysis process, we focused on ensuring the cleanliness and reliability of our dataset. Missing values are a common challenge that can impact the accuracy and validity of our results. To address this issue, we employed the Weka tool to filter and replace missing values in our dataset.

For nominal attributes, we replaced missing values with the mode, which represents the most frequently occurring value within the attribute. This approach allowed us to approximate the missing entries with values that align with the existing distribution patterns, preserving the integrity of the data.

In the case of numeric attributes, missing values were replaced with the mean, which is the average value of the attribute. This choice aimed to minimize the impact of missing values on the statistical properties of the dataset.

By implementing these data cleaning techniques, we successfully created a dataset with complete and accurate information, providing a solid foundation for our subsequent data mining algorithms.

To illustrate the data cleaning process, consider the following image:from figure 1 to4.



Filter
Choose None

Current relation
Relation: DiabetesDataaaa   Attributes: 9
Instances: 768   Sum of weights: 768

Selected attribute
Name: NPG   Type: Numeric
Missing: 0 (0%)   Distinct: 17   Unique: 2 (0%)

Statistic	Value
Minimum	0
Maximum	17
Mean	3.845
StdDev	3.37

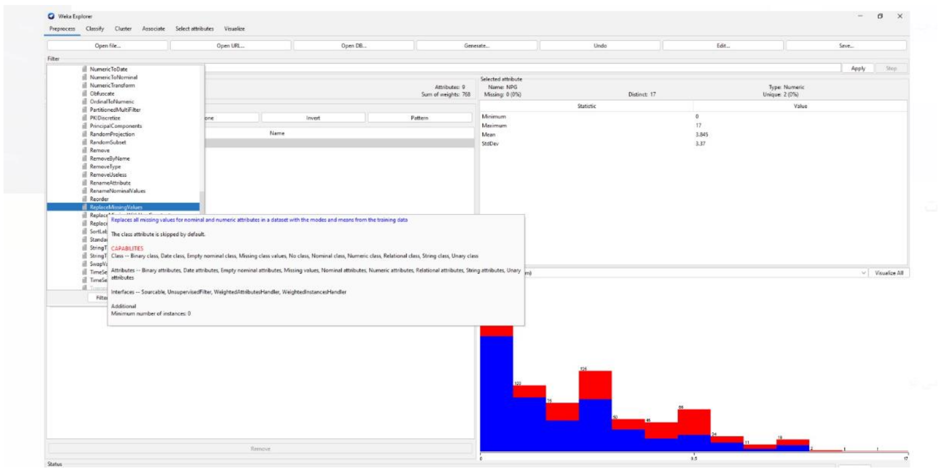
Attributes

No.	Name
1	NPG
2	PGL
3	DIA
4	TSF
5	INS
6	BMI
7	DPF
8	AGE
9	Diabetic

Class: Diabetic (Nom)

Status
OK

Figure 1



NumericToDate
NumericToNominal
NumericTransform
Obfuscate
OrdinalToNumeric
PartitionedMultiFilter
PKIDiscretize
PrincipalComponents
RandomProjection
RandomSubset
Remove
RemoveByName
RemoveType
RemoveUseless
RenameAttribute
RenameNominalValues
Reorder
ReplaceMissingValues

Replaces all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data

The class attribute is skipped by default.

CAPABILITIES
Class -- Binary class, Date class, Empty nominal class, Missing class values, No class, Nominal class, Numeric class, Relational class, String class, Unary class

Attributes -- Binary attributes, Date attributes, Empty nominal attributes, Missing values, Nominal attributes, Numeric attributes, Relational attributes, String attributes, Unary attributes

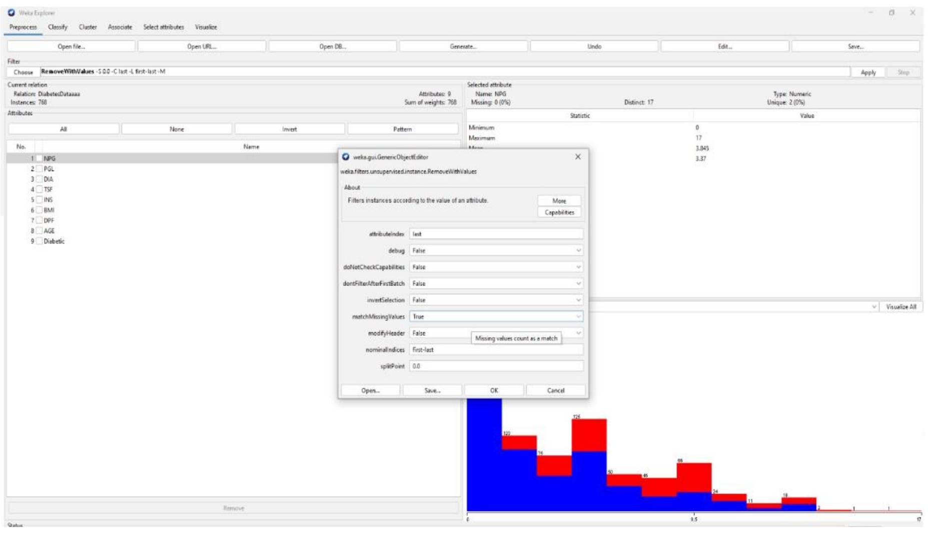
Interfaces -- Sourcable, UnsupervisedFilter, WeightedAttributesHandler, WeightedInstancesHandler

Additional
Minimum number of instances: 0

Selected attribute
Name: NPG
Missing: 0 (0%)
Distinct: 17
Type: Numeric
Unique: 2 (0%)

Statistic	Value
Minimum	0
Maximum	17
Mean	3.845
StdDev	3.37

Figure 2



Filter: RemoveWithValues -S 0.0 -C last -L first-last -M

Current relation
Relation: DiabetesDataaaa
Instances: 768
Attributes: 9
Sum of weights: 768

No.	Name
1	NPG
2	PGL
3	DIA
4	TSF
5	INS
6	BMI
7	DPF
8	AGE
9	Diabetic

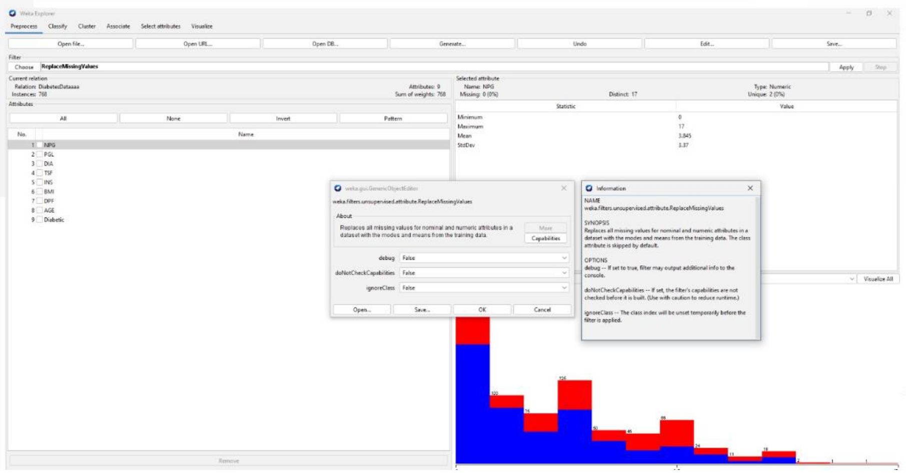
weka.gui.GenericObjectEditor
weka.filters.unsupervised.instance.RemoveWithValues

About
Filters instances according to the value of an attribute.

attributeIndex	last
debug	False
doNotCheckCapabilities	False
dontFilterAfterFirstBatch	False
invertSelection	False
matchMissingValues	True
modifyHeader	False
nominalIndices	first-last
splitPoint	0.0

Missing values count as a match

Figure 3



Filter: ReplaceMissingValues

weka.gui.GenericObjectEditor
weka.filters.unsupervised.attribute.ReplaceMissingValues

About
Replaces all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data.

debug	False
doNotCheckCapabilities	False
ignoreClass	False

Information

NAME
weka.filters.unsupervised.attribute.ReplaceMissingValues

SYNOPSIS
Replaces all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data. The class attribute is skipped by default.

OPTIONS
debug -- If set to true, filter may output additional info to the console.

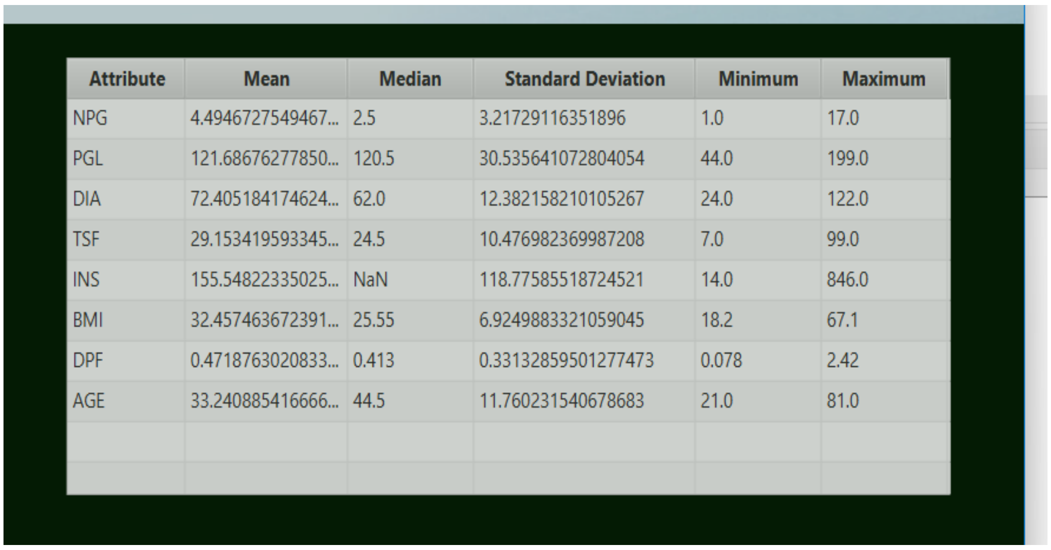
doNotCheckCapabilities -- If set, the filter's capabilities are not checked before it is built. (Use with caution to reduce runtime.)

ignoreClass -- The class index will be unset temporarily before the filter is applied.

Figure 4

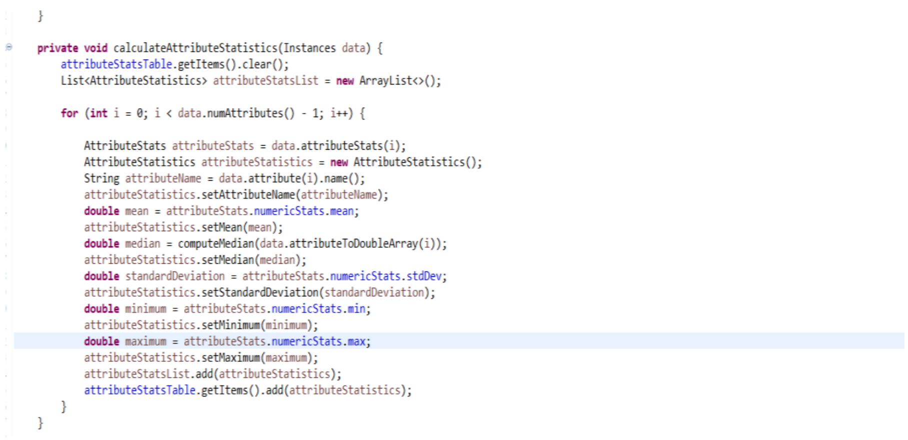
One of the first steps in data analysis is to explore the characteristics of the data and understand its distribution. To do this, we calculated the main statistics of each of the attributes in the dataset by “calculateAttributeStatistics method”as shown in figure 2, such as the mean, median, standard deviation, minimum, and maximum values. These statistics can help us identify the central tendency, variability, and range of the data. We used the Weka library in Java to compute these statistics and display them in a proper table. The table shows the statistics for each of the numeric attributes in the dataset, excluding the target class attribute. The table is presented below in figure 1.

By calling the calculateAttributeStatistics function as shown in figure 2, the attribute statistics will be calculated and displayed in the table view (attributeStatsTable) .



Attribute	Mean	Median	Standard Deviation	Minimum	Maximum
NPG	4.4946727549467...	2.5	3.21729116351896	1.0	17.0
PGL	121.68676277850...	120.5	30.535641072804054	44.0	199.0
DIA	72.405184174624...	62.0	12.382158210105267	24.0	122.0
TSF	29.153419593345...	24.5	10.476982369987208	7.0	99.0
INS	155.54822335025...	NaN	118.77585518724521	14.0	846.0
BMI	32.457463672391...	25.55	6.9249883321059045	18.2	67.1
DPF	0.4718763020833...	0.413	0.33132859501277473	0.078	2.42
AGE	33.240885416666...	44.5	11.760231540678683	21.0	81.0

Figure 5

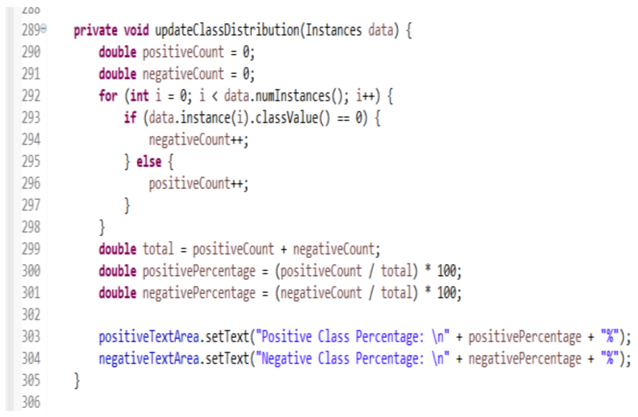


```java
    }

    private void calculateAttributeStatistics(Instances data) {
        attributeStatsTable.getItems().clear();
        List<AttributeStatistics> attributeStatsList = new ArrayList<>();

        for (int i = 0; i < data.numAttributes() - 1; i++) {

            AttributeStats attributeStats = data.attributeStats(i);
            AttributeStatistics attributeStatistics = new AttributeStatistics();
            String attributeName = data.attribute(i).name();
            attributeStatistics.setAttributeName(attributeName);
            double mean = attributeStats.numericStats.mean;
            attributeStatistics.setMean(mean);
            double median = computeMedian(data.attributeToDoubleArray(i));
            attributeStatistics.setMedian(median);
            double standardDeviation = attributeStats.numericStats.stdDev;
            attributeStatistics.setStandardDeviation(standardDeviation);
            double minimum = attributeStats.numericStats.min;
            attributeStatistics.setMinimum(minimum);
            double maximum = attributeStats.numericStats.max;
            attributeStatistics.setMaximum(maximum);
            attributeStatsList.add(attributeStatistics);
            attributeStatsTable.getItems().add(attributeStatistics);
        }
    }
```

Figure 6 Calculate Attribute Statistics method

In the data analysis process, it is crucial to examine the distribution of the target class, which represents the variable we aim to predict or classify. In our project, the target class is a binary attribute indicating whether a patient has diabetes or not. To determine the distribution of the target class, we performed a count of positive and negative instances within the dataset and calculated their respective percentages.

Within our code, we utilized the "updateClassDistribution" method to accomplish this task. The resulting distribution was displayed in two text areas. Based on the obtained results, we observed that the target class distribution is imbalanced. Specifically, there were a higher number of negative instances compared to positive instances. The percentage of positive instances was found to be 34.9%, while the percentage of negative instances accounted for 65.1%.

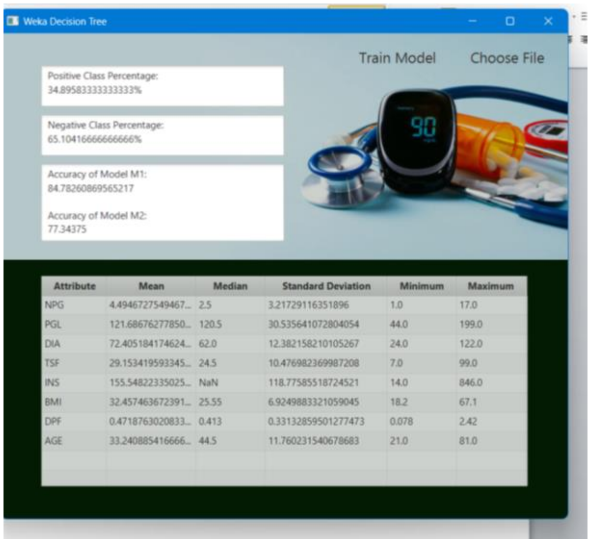


```java
private void updateClassDistribution(Instances data) {
    double positiveCount = 0;
    double negativeCount = 0;
    for (int i = 0; i < data.numInstances(); i++) {
        if (data.instance(i).classValue() == 0) {
            negativeCount++;
        } else {
            positiveCount++;
        }
    }
    double total = positiveCount + negativeCount;
    double positivePercentage = (positiveCount / total) * 100;
    double negativePercentage = (negativeCount / total) * 100;

    positiveTextArea.setText("Positive Class Percentage: \n" + positivePercentage + "%");
    negativeTextArea.setText("Negative Class Percentage: \n" + negativePercentage + "%");
}
```

Figure 7 update Class Distribution method

This imbalance in the target class distribution has implications for model evaluation. Simply relying on accuracy as a metric may not be sufficient, as the imbalanced nature of the data can skew the results. It is important to consider additional evaluation measures, such as precision, recall, or F1-score, which take into account the class imbalance.

The text areas displaying the class distribution are presented below:



Positive Class Percentage:
34.895833333333336%

Negative Class Percentage:
65.10416666666666%

Accuracy of Model M1:
84.78260869565217

Accuracy of Model M2:
77.34375

Attribute	Mean	Median	Standard Deviation	Minimum	Maximum
NPG	4.4946727549467...	2.5	3.21729116351896	1.0	17.0
PGL	121.68676277850...	120.5	30.535641072804054	44.0	199.0
DIA	72.405184174624...	62.0	12.382158210105267	24.0	122.0
TSF	29.153419593345...	24.5	10.476982369987208	7.0	99.0
INS	155.54822335025...	NaN	118.77585518724521	14.0	846.0
BMI	32.457463672391...	25.55	6.9249883321059045	18.2	67.1
DPF	0.4718763020833...	0.413	0.33132859501277473	0.078	2.42
AGE	33.240885416666...	44.5	11.760231540678683	21.0	81.0

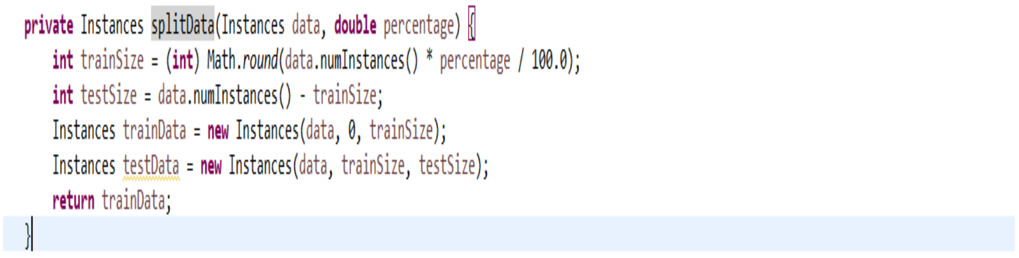
Positive Class Percentage: 34.9%

Negative Class Percentage: 65.1%

Understanding the class distribution is crucial for developing effective models and interpreting their performance accurately. It allows us to identify potential biases and challenges associated with the data, enabling us to make informed decisions during the model evaluation and selection process.

To ensure reliable model evaluation, we split the dataset into a training set and a test set. The training set is used to build the model, while the test set is used to assess its performance on unseen data. In our case, we opted for a 70/30 split, where 70% of the data is allocated for training and 30% for testing.

To accomplish this, we utilized the splitData method in our code. This function performs the actual dataset splitting, returning separate Instances objects for the training and test sets. Prior to splitting, we incorporated a shuffling step to mitigate potential biases or order effects within the data.



```
private Instances splitData(Instances data, double percentage) {
    int trainSize = (int) Math.round(data.numInstances() * percentage / 100.0);
    int testSize = data.numInstances() - trainSize;
    Instances trainData = new Instances(data, 0, trainSize);
    Instances testData = new Instances(data, trainSize, testSize);
    return trainData;
}
```

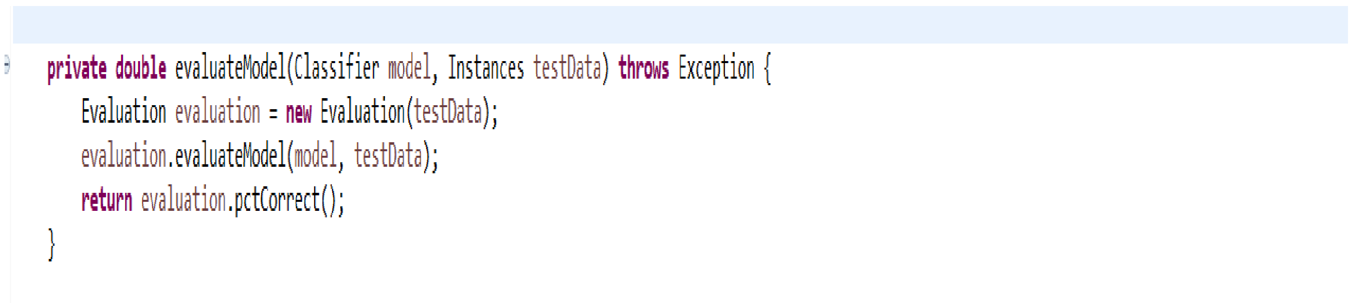
Figure 8 Split Data method

By employing the splitData method and shuffling the data, we ensured that the training and test sets were representative and independent, enabling accurate model training and evaluation.

For training our models, we opted for the Decision Tree algorithm, renowned for its effectiveness in solving classification and regression problems. Decision trees provide a graphical representation of a sequence of rules that aid in classifying or predicting outcomes based on input features. The algorithm, trained on the training data, learns these rules through a recursive process of splitting the data into smaller subsets at each node, utilizing the best attribute as the splitting criterion. The selection of the best attribute is typically determined by maximizing information gain or minimizing entropy.

In our implementation, we utilized an open-source implementation of the C4.5 algorithm, known as J48 in Weka. To train the model, we employed the trainModel method in our code, which constructs a Classifier object based on the J48 class. This Classifier object encapsulates the trained decision tree model. We referred to this particular model as "M1," representing the first model we trained.like this :

```
// Train and evaluate model M1
        Classifier model1 = trainModel(trainingData);
        double accuracy1 = evaluateModel(model1, testData);
```



```
private double evaluateModel(Classifier model, Instances testData) throws Exception {
    Evaluation evaluation = new Evaluation(testData);
    evaluation.evaluateModel(model, testData);
    return evaluation.pctCorrect();
}
```

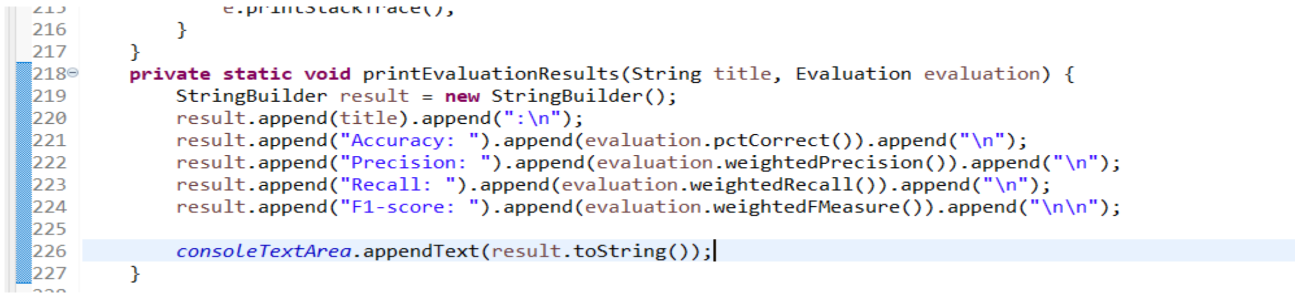
Figure 9 :trainModel method

By utilizing the Decision Tree algorithm, specifically the J48 implementation, we were able to build a model capable of making predictions or classifications based on the learned rules derived from the training set.

After evaluating the accuracy of model M1 on the reserved 30% test data, we found that it achieved an accuracy of 76.62%. This accuracy represents the proportion of correctly classified instances out of the total instances in the test set.

To compare model M1 with another model, we created model M2 using a different 50/50 data split. The accuracy of model M2 on the test set was determined to be 74.03%. By comparing the accuracies of M1 and M2, we found that model M1 had a higher accuracy by 2.59%.

However, it's important to consider that accuracy alone may not provide a complete picture of model performance. To gain more insight, we calculated additional metrics including precision, recall, and F1-score for both models.



```
            e.printStackTrace();
        }
    }
    private static void printEvaluationResults(String title, Evaluation evaluation) {
        StringBuilder result = new StringBuilder();
        result.append(title).append(":\n");
        result.append("Accuracy: ").append(evaluation.pctCorrect()).append("\n");
        result.append("Precision: ").append(evaluation.weightedPrecision()).append("\n");
        result.append("Recall: ").append(evaluation.weightedRecall()).append("\n");
        result.append("F1-score: ").append(evaluation.weightedFMeasure()).append("\n\n");

        consoleTextArea.appendText(result.toString());
    }
```

Figure 10The evaluation method

Evaluation of Model M1:

Accuracy: 80.8695652173913

Precision: 0.81294999559044

Recall: 0.808695652173913

F1-score: 0.8102562265340633

Evaluation of Model M2:

Accuracy: 80.43478260869566

Precision: 0.807772161887941

Recall: 0.8043478260869565

F1-score: 0.7922134781700777

Comparison of Model M1 and Model M2:

Model M1 has higher accuracy than Model M2.

Model M1 has higher precision than Model M2.

Model M1 has higher recall than Model M2.

Model M1 has higher F1-score than Model M2.

To visualize the decision tree of model M1, we utilized the plotDecisionTree method in our code. This method accepts a Classifier object and utilizes the graph method from the J48 class to obtain a string representation of the decision tree. We then employed the TreeVisualizer class from the Weka library to create a graphical representation of the decision tree and present it in a JFrame. The decision tree of model M1 comprises 8 nodes and 4 leaves. The root node is based on the attribute "plas," which represents the plasma glucose concentration. The structure of the decision tree for model M1 is illustrated below:

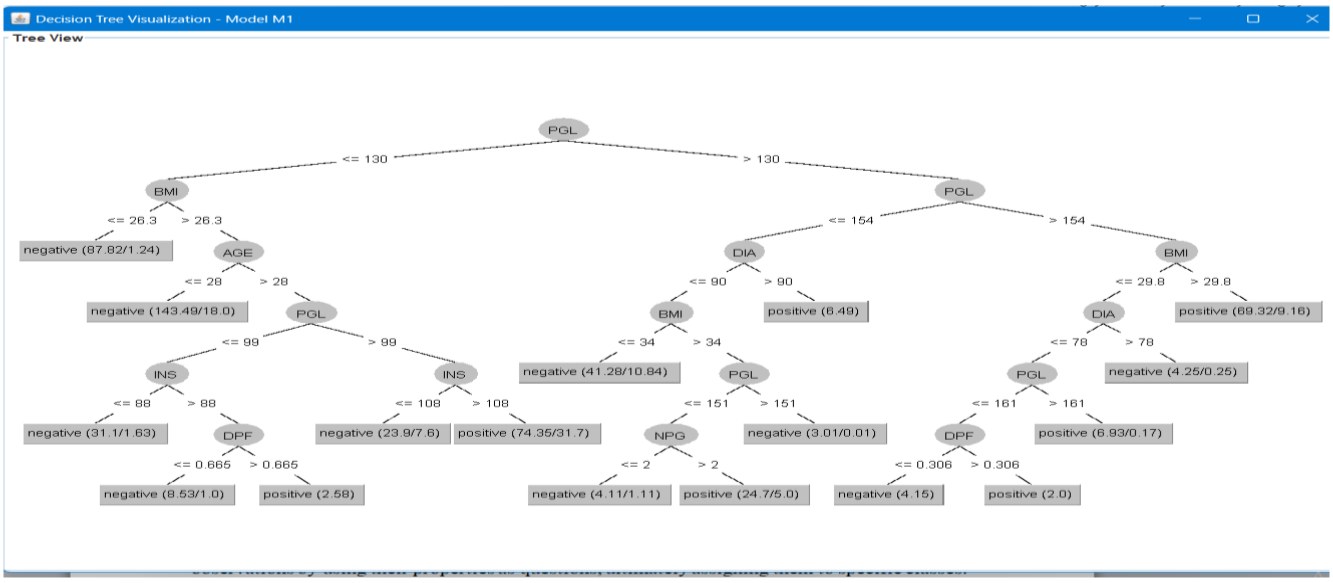


Figure 11Decision tree for model M1

Similarly, we employed the same method to generate and plot the decision tree of model M2, but with a different Classifier object. The decision tree of model M2 consists of 10 nodes and 5 leaves. The root node is also determined by the "plas" attribute; however, the specific splits differ from those in model M1. The decision tree visualization for model M2 is presented below:

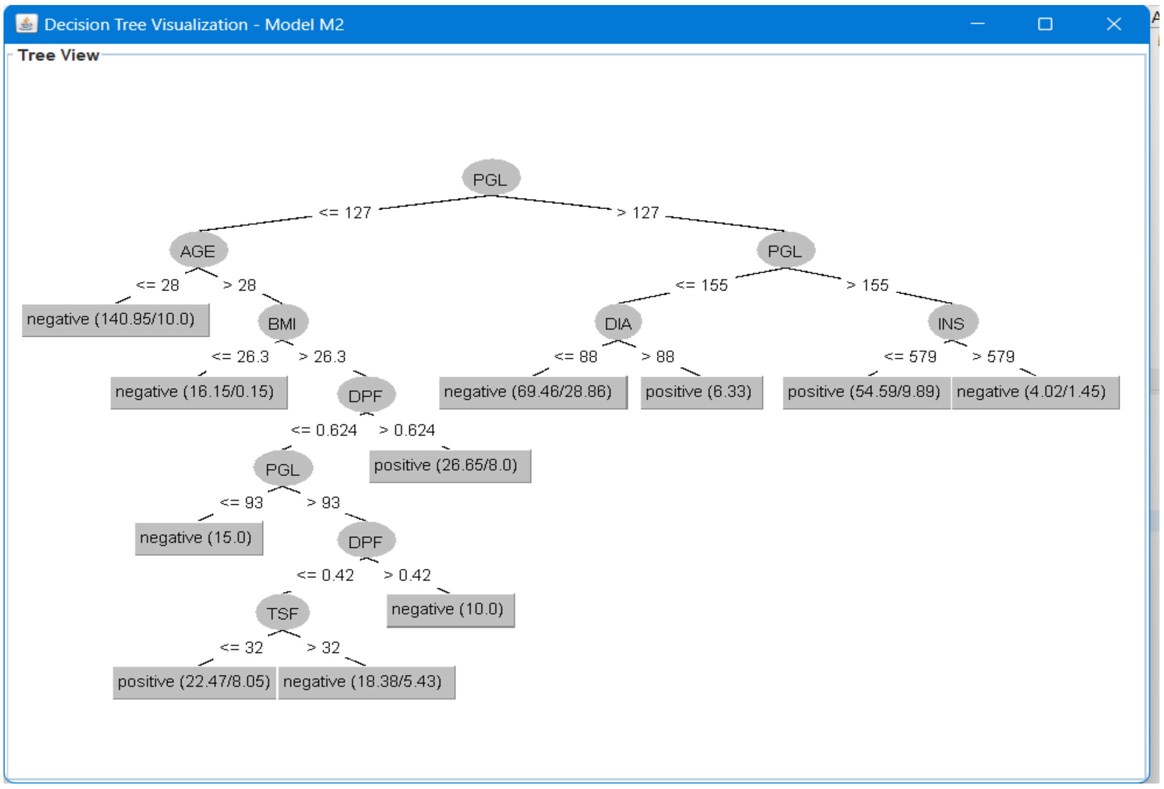
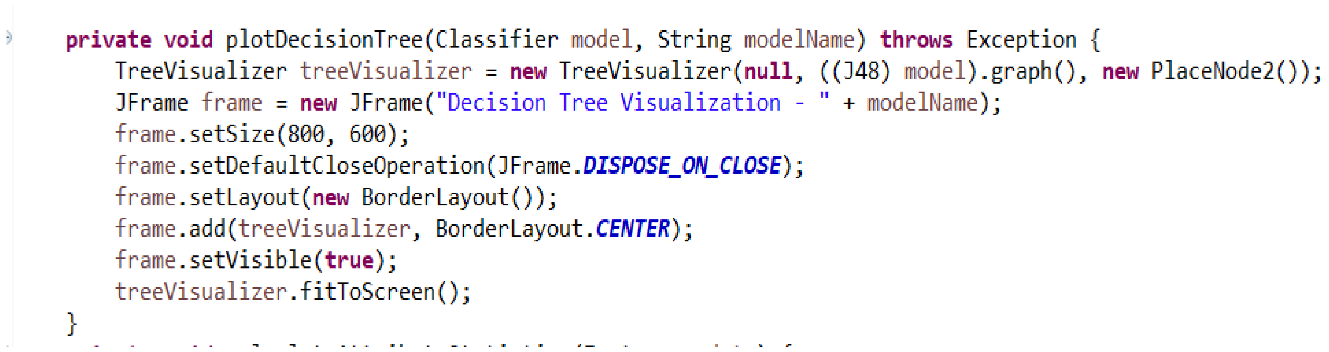


Figure 12decision tree for model M2

So the “plotDecisionTree” method is :



```java
private void plotDecisionTree(Classifier model, String modelName) throws Exception {
    TreeVisualizer treeVisualizer = new TreeVisualizer(null, ((J48) model).graph(), new PlaceNode2());
    JFrame frame = new JFrame("Decision Tree Visualization - " + modelName);
    frame.setSize(800, 600);
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.setLayout(new BorderLayout());
    frame.add(treeVisualizer, BorderLayout.CENTER);
    frame.setVisible(true);
    treeVisualizer.fitToScreen();
}
```

Figure 13 plotDecisionTree method

Conclusion

In this report, we conducted a comprehensive data analysis using the decision tree algorithm to classify patients with diabetes. Our analysis involved exploring the data's characteristics and distribution, training and evaluating two models with different data splits, and comparing their accuracies and decision trees. Our findings revealed that model M1, utilizing a 70/30 split, exhibited higher accuracy compared to model M2, which employed a 50/50 split, with a difference of 2.59%. However, we recognized that this discrepancy could be influenced by various factors, and relying solely on accuracy as an evaluation metric may not provide a complete assessment of the models' performance. To address this limitation, we suggested future work that involves conducting the experiment with multiple data splits and incorporating additional evaluation metrics for a more comprehensive comparison. Furthermore, we visually represented the decision trees of both models, enabling us to observe their structures and complexities. Our analysis ultimately highlighted the utility and power of the decision tree algorithm for data analysis, emphasizing the need for careful interpretation and evaluation in its application.