Stony Brook University

ESE 589: Learning Systems for Engineering Applications

Project 2

Sushanth Reddy Gurram #115668498
Mohammadreza Bakhtiari #115843646

Instructor: Prof. Alex Doboli
Department of Electrical and Computer Engineering

November 2023

Table of Contents

Decision Tree Algorithm

I. Introduction

Decision trees are foundational tools in machine learning, excelling in tasks such as classification and regression. This project focuses on the implementation of a decision tree algorithm utilizing three key metrics for attribute selection: Information Gain, Gain Ratio, and Gini Index. Our objective is to investigate the influence of these metrics on the decision tree's performance across 15 diverse datasets. Through comprehensive evaluation measures such as accuracy, confusion matrices, computer memory usage, and execution time, we aim to discern the strengths and limitations of each metric, providing valuable insights for practical machine learning applications.

II. Implementation

II.1 Data Preprocessing

In the initial phase of our project, we carefully curated a set of 15 benchmark datasets sourced from the UCI repository, representing diverse domains for comprehensive evaluation. The subsequent data preprocessing steps were essential to ensure the robustness and applicability of our decision tree algorithm.

1- Handling Missing Data:

Addressing missing data is crucial for the reliability of any machine learning model. In our preprocessing pipeline, we adopted a straightforward approach by employing majority voting. For each feature, if a data point had missing values, we replaced them with the most frequent value in that feature. This method ensures that missing values are imputed with values representative of the majority in the respective feature.

2- Discretization of Continuous Values:

Decision trees generally perform optimally on categorical data. To adapt continuous values to this requirement, we implemented a discretization strategy. Continuous features were transformed into discrete categories or "bins" based on the frequency distribution of the data. The determination of the number of bins for each feature involved considering factors such as the feature's minimum and maximum values, as well as its average and variance. Through an empirical approach, we tested the algorithm with different bin sizes and selected the configuration that yielded the best results.

It is important to note that the choice of bin size is a trade-off. A small number of bins may lead to inaccurate categorization, potentially impacting results on test data. Conversely, an excessive number of bins may result in overfitting to the training data, leading to suboptimal performance on new, unseen data.

3- Mapping Values:

In the final step, a proactive measure was taken to mitigate potential errors associated with string values. To streamline further data handling and processing, a mapping strategy was employed. This involved assigning unique integer values to each distinct string value within the dataset's features. This transformation ensures that subsequent stages of the algorithm exclusively operate with numerical values, eliminating complexities associated with string data and enhancing the overall robustness of the decision tree algorithm.

For instance, consider a feature like 'age' with unique string values:

age: {'*youth*', '*middle_aged*', '*senior*'}

To facilitate numerical computations and enhance algorithm robustness, integer values were assigned to each unique string value. In this example, 'youth' could be represented as '0', 'middle_aged' as '1', and 'senior' as '2'. Therefore we have:

age: {‘*0*’, ‘*1*’, ‘*2*’}

4- Train-Test Data Split:

With the preprocessing steps completed, we partitioned each dataset into two distinct categories: the training dataset and the test dataset. A 70-30 split was employed, with 70% of the data allocated to the training set and the remaining 30% to the test set. This division ensures that the model is trained on a substantial portion of the data while allowing for an independent evaluation of unseen data, providing a robust assessment of the decision tree algorithm's generalization performance.

Here is a summary of data preprocessing that we applied for each dataset:

Dataset	Data preprocessing
1- Adult	This dataset has 14 attributes and 48842 samples. 6 of the attributes which are: {‘age’, ‘fnlwgt’, ‘education-num’, ‘capital-gain’, ‘capital-loss’, ‘hours-per-week’} were continuous and needed to be converted into categories.
2- Abalone	This dataset has 8 attributes and 4178 samples. 7 of the attributes which are: {‘Length’, ‘Diameter’, ‘Height’, ‘Whole_weight’, ‘Shucked_weight’, ‘Viscera_weight’, ‘Shell_weight’} were continuous and needed to be converted into categories.
3- Auto MPG	This dataset has 7 attributes and 399 samples. 5 of the attributes which are: {‘displacement’, ‘horsepower’, ‘weight’, ‘acceleration’, ‘model_year’} were continuous and needed to be converted into categories.
4- Breast Cancer	This dataset has 9 attributes and 287 samples. All attributes are categorized and there is no need for preprocessing.
5- Breast Cancer Wisconsin (Original)	This dataset has 9 attributes and 700 samples. 8 of the attributes which are: {‘Clump_thickness’, ‘Uniformity_of_cell_size’, ‘Uniformity_of_cell_shape’, ‘Marginal_adhesion’, ‘Single_epithelial_cell_size’, ‘Bare_nuclei’, ‘Bland_chromatin’, ‘Normal_nucleoli’} were continuous and needed to be converted into categories.
6- Credit Approval	This dataset has 15 attributes and 691 samples. 6 of the attributes which are: {‘A15’, ‘A14’, ‘A11’, ‘A8’, ‘A3’, ‘A2’}

	were continuous and needed to be converted into categories.
7- Iris	This dataset has 4 attributes and 151 samples. 4 of the attributes which are: {'sepal length', 'sepal width', 'petal length', 'petal width'} were continuous and needed to be converted into categories.
8- Letter Recognition	This dataset has 16 attributes and 20001 samples. All of the attributes were continuous and needed to be converted into categories.
9- Lung Cancer	This dataset has 56 attributes and 33 samples. All attributes are categorized and there is no need for preprocessing.
10- Optical Recognition of Handwritten Digits	This dataset has 64 attributes and 5621 samples. All attributes are categorized and there is no need for preprocessing.
11- Spambase	This dataset has 16 attributes and 436 samples. All attributes are categorized and there is no need for preprocessing.
12- Zoo	This dataset has 16 attributes and 102 samples. All attributes are categorized and there is no need for preprocessing.
13- Statlog (German Credit Data)	This dataset has 20 attributes and 1001 samples. 3 of the attributes which are: {'Attribute2', 'Attribute5', 'Attribute13'} were continuous and needed to be converted into categories.
14- Bank Marketing	This dataset has 16 attributes and 45212 samples. 7 of the attributes which are: {'age', 'balance', 'day_of_week', 'duration', 'campaign', 'pdays', 'previous'} were continuous and needed to be converted into categories.
15- Diabetes 130-US hospitals for years 1999-2008	This dataset has 47 attributes and 101767 samples. All attributes are categorized and there is no need for preprocessing.
16- Glioma Grading Clinical and Mutation Features	This dataset has 23 attributes and 840 samples. 1 of the attribute which are: {'Age_at_diagnosis'} were continuous and needed to be converted into categories.
17- Sepsis Survival Minimal Clinical Records	This dataset has 3 attributes and 110342 samples. 1 of the attribute which are: {'age_years'} were continuous and needed to be converted into categories.

II.2 Algorithm Implementation and Operation

In this section, we present the implementation details of the decision tree algorithm, focusing on the three metrics: Information Gain, Gain Ratio, and Gini Index. The algorithm is encapsulated within a class named **'Tree'**, which serves as the fundamental unit for storing information about each node in the decision tree.

1- Tree Class Structure:

The **'Tree'** class contains attributes such as **'attribute'**, **'branches'**, **'value'**, and **'leaf_value'**. These attributes facilitate the representation of each node in the decision tree. Specifically, **'attribute'** signifies the feature used for splitting, **'branches'** hold the child nodes, **'value'** denotes the value of the current node's attribute, and **'leaf_value'** is applicable when the node is a leaf, containing the predicted label.

2- Algorithm Structure:

The decision tree algorithm employs a recursive function for tree construction, with two termination conditions: a node's impurity becoming zero or when all attributes are exhausted. Additionally, a depth limit is imposed to control the tree's size and prevent overfitting.

3- Metric-Specific Functions:

Metric-specific functions (**'informationGain'**, **'GainRatio'**, **'giniIndex'**) compute the respective metrics for attribute selection during tree construction. These functions are integral to determining the best attribute for splitting at each node.

4- Decision Tree Construction:

The decision tree construction is metric-dependent, utilizing the specified metric functions (**'decisionTree_InformationGain'**, **'decisionTree_GainRatio'**, **'decisionTree_GiniIndex'**). These functions recursively build the tree by selecting the optimal attribute at each node based on the chosen metric.

5- Handling Leaf Nodes:

The algorithm identifies leaf nodes based on impurity conditions and a specified depth limit. In leaf nodes, the **'leaf_value'** attribute stores the predicted label, either the majority vote or a specific label when impurity becomes zero.

6- Testing and Evaluation:

The algorithm is tested on a training dataset, and the constructed tree is applied to a test dataset for evaluation. The **'predict'** and **'predictDf'** methods in the **'Tree'** class facilitate prediction on individual data points and entire datasets, respectively. The accuracy and confusion matrix is then calculated and displayed for each metric.

7- Using inbuilt Scikit SVM, Decision Tree, and Tensforflow Deep Learning models

As part of our comparison process, we have employed inbuilt classification machine learning models such as Support Vector Machines (SVMs), Decision Tree Classifier from scikit-learn library and Convoluted Neural Network (CNN) Deep Learning Model from tensorflow platform. Since scikit-learn and tensorflow libraries cannot process categorical data for classification, there is a need to convert categorical data to numerical data before training the machine learning model. During preprocessing of the data, the conversion of categorical data to numerical data takes place (Section II-1 Data Preprocessing) and this data is used to train the machine learning model.

To train SVM and Decision tree models, we leveraged the scikit-learn library's model.fit() function using 80% of the dataset for training. Since most of our datasets are a problem of binary classification, we used a linear model of SVC (SVM model in scikit-learn). Similarly, to train the CNN deep learning model, we used 3 layers connected sequentially to perform the training with each layer having a different activation function such as Rectified Linear Unit for the first 2 layers and a Sigmoid for the output layer. The CNN model is compiled using binary_crossentropy as its loss function. For all three models, we practised the accuracy_scores() library to compute the accuracy of the models, instead of using our own functions for robustness and reliability.

II.3 Decision Tree Data Structure

The decision to use a specific data structure for the implementation of the decision tree algorithm is crucial for efficiency, readability, and ease of manipulation. In this case, a tree-based data structure, implemented through the 'Tree' class, was chosen for several reasons:

1- Hierarchical Representation:

The decision tree is inherently hierarchical, mirroring the structure of a tree. The 'Tree' class allows the representation of nodes, branches, and leaves in a way that aligns with the logical structure of a decision tree.

2- Recursive Nature of Decision Trees:

Decision tree algorithms often involve recursive operations, especially during the construction of the tree and the prediction phase. The 'Tree' class's recursive design facilitates the implementation of these recursive operations in a clear and concise manner.

3- Ease of Interpretation and Visualization:

The tree-based structure is intuitive and aligns with how decision trees are traditionally visualized. This makes it easier to interpret and understand the structure of the decision tree, aiding in debugging and analysis.

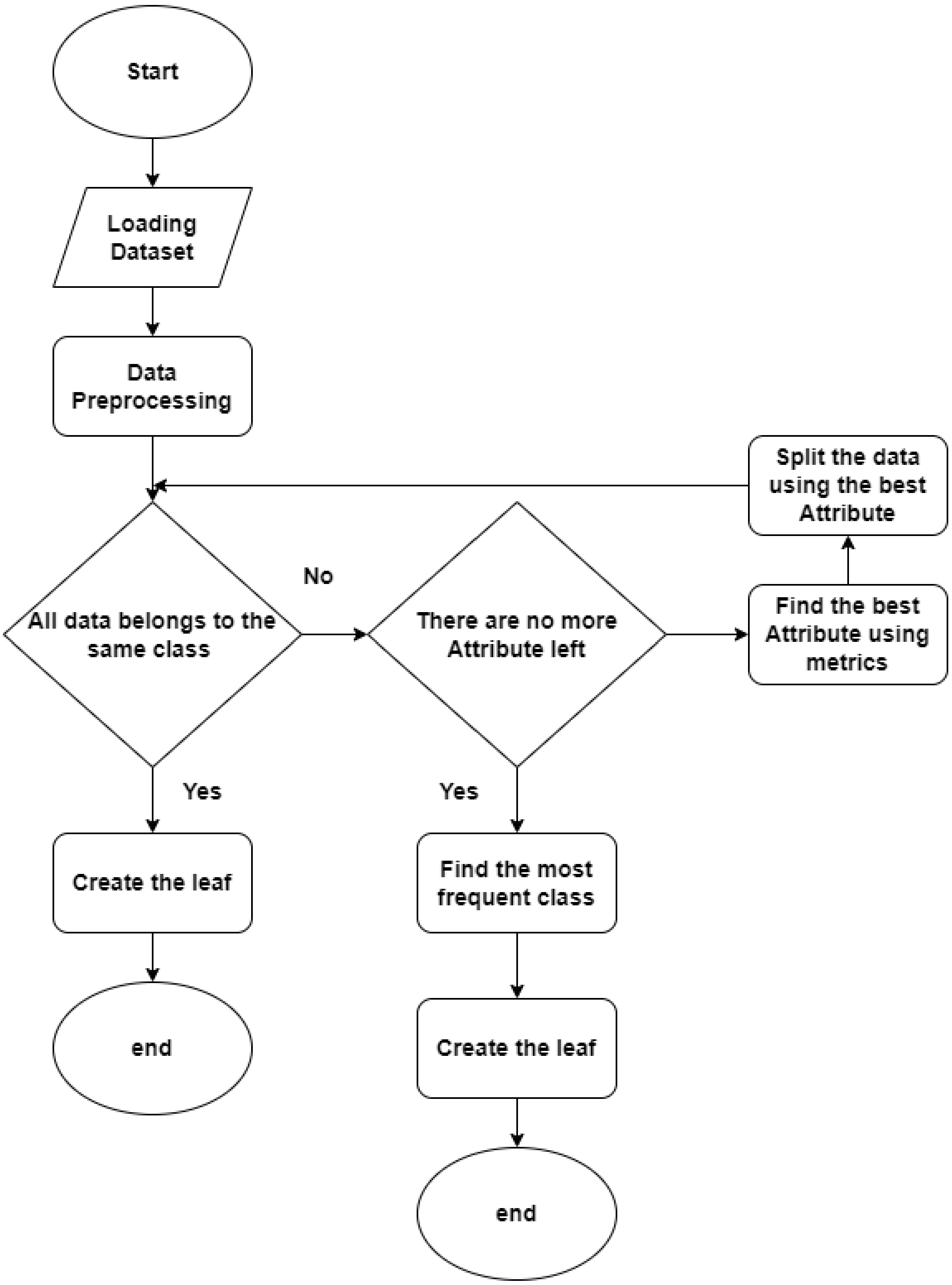
4- Flexibility for Tree Traversal:

The tree structure enables straightforward traversal through branches and nodes, supporting the efficient implementation of decision-making logic and predictions.

Considering the above reasons, the 'Tree' class serves as an appropriate and effective data structure for implementing the decision tree algorithm. It aligns well with the nature of decision trees and supports the algorithm's key operations.

II.4 Decision Tree Diagram

Here you can see the summary of what we talked about previously in the following diagram:

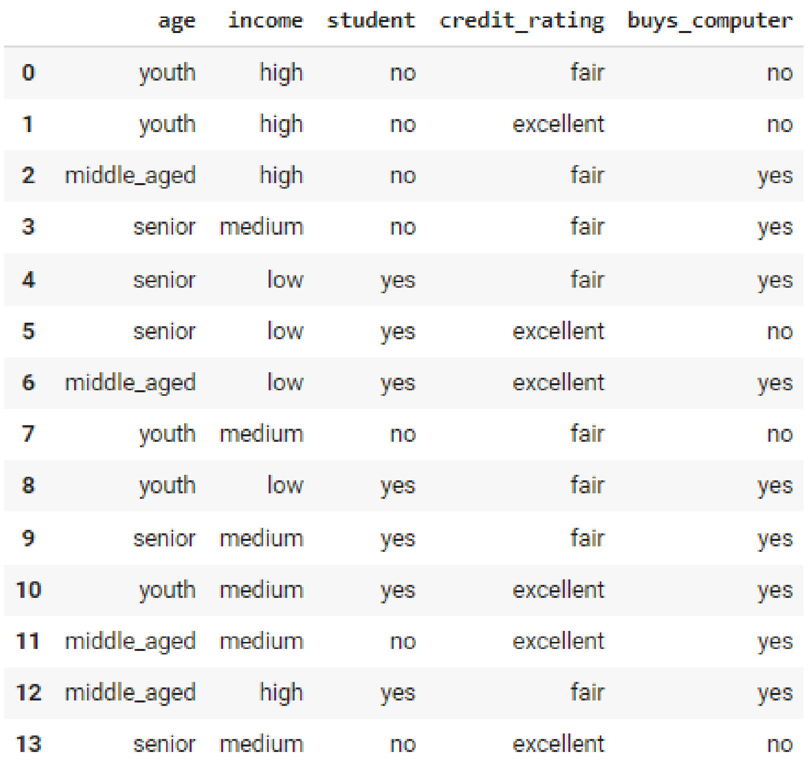

Start
Loading Dataset
Data Preprocessing
All data belongs to the same class
No
Yes
Create the leaf
end
There are no more Attribute left
Yes
Find the most frequent class
Create the leaf
end
Find the best Attribute using metrics
Split the data using the best Attribute

II.4 Results and Analysis

II.4.0 Validating Decision Tree on Sample Dataset:

To validate the functionality and correctness of the decision tree algorithm, a small sample dataset was carefully chosen. The algorithm was applied to this dataset, and the resulting tree structure was extracted. This preliminary validation step ensures that the algorithm is capable of constructing a meaningful decision tree on a simplified dataset.

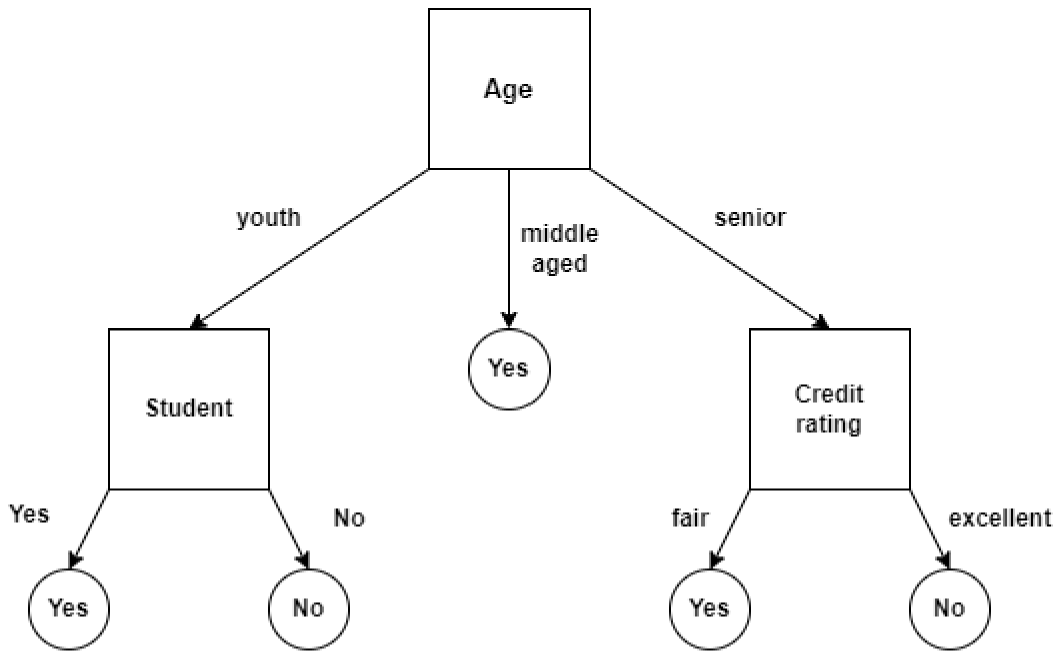
To see the correctness of the algorithm, the sample dataset was chosen from the textbook. Here is the dataset:



	age	income	student	credit_rating	buys_computer
0	youth	high	no	fair	no
1	youth	high	no	excellent	no
2	middle_aged	high	no	fair	yes
3	senior	medium	no	fair	yes
4	senior	low	yes	fair	yes
5	senior	low	yes	excellent	no
6	middle_aged	low	yes	excellent	yes
7	youth	medium	no	fair	no
8	youth	low	yes	fair	yes
9	senior	medium	yes	fair	yes
10	youth	medium	yes	excellent	yes
11	middle_aged	medium	no	excellent	yes
12	middle_aged	high	yes	fair	yes
13	senior	medium	no	excellent	no

Since here the purpose is just to demonstrate the correctness of the algorithm, we just need to construct the final tree, therefore we allocated all data (13 rows) for the training process. Here is the final tree that we obtained from the algorithm as we expected: (The result for this specific sample dataset is the same for all three metrics)

```
age
--> middle_aged: yes
--> senior: credit_rating
    --> excellent: no
    --> fair: yes
--> youth: student
    --> no: no
    --> yes: yes
```



Also for validating the algorithm on a large dataset, in the following, we will compare the results obtained from reliable models such as the Scikit SVM Model, Tensorflow DL Model, and Scikit Decision Tree Model with our implementation.

UCI benchmark datasets:

The decision tree algorithm was subsequently tested on a comprehensive set of 15 diverse datasets obtained from the UCI benchmarks. The performance evaluation includes Accuracy, Execution Time, Peak Memory, Increment in Memory, and Confusion Matrix for each metric (Information Gain, Gain Ratio, Gini Index):

Note: Increment in memory represents the change in memory usage compared to the baseline memory. It is calculated as the difference in memory usage before and after the algorithm's execution.

Also, for some datasets, the size of the confusion matrix exceeded the practical limits for inclusion in this report so we didn't include them.

II.4.1 Dataset-1: Adult

Dataset Summary:

This dataset comprises 48842 instances and 14 features for each instance. This is a binary classification problem, where an Adult's income is predicted to be more than 50k dollars or otherwise using those 14 features. The summary table with the list of values obtained for the Adult Dataset is as follows,

Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
Information Gain	82.15	89.7	136,314.88
Gain Ratio	81.10	84.78	199,229.44
Gini Index	83.03	96.11	356,515.84
Scikit SVM Model	90.47	68.240	6950628
Tensorflow DL Model	3.9	54.465 t	16161653
Scikit Decision Tree Model	86.880	0.721	1038057

Learning:

Because of the space constraint, the confusion matrix is not posted here. From the above table, the accuracy of our Decision Tree implementation is on par with the Scikit Decision Tree model. From the confusion matrix, the Precision (# True Positives / # Total Positives) value is calculated to be around 0.76 which suggests that our model can be deployed to predict whether an Adult's income is more than 50k or not pretty accurately. However, things did not turn out as well as expected in the case of the TensorFlow CNN model as its accuracy is way below than what we expected. This can be the case of improper fine-tuning or wrong activation units utilized which has to be scrutinized more.

II.4.2 Dataset-2: Abalone

Dataset Summary:

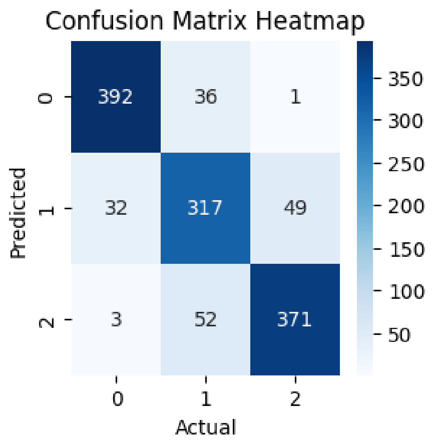
This dataset comprises 4177 instances and 8 features for each instance. Through this dataset, an Abalone's age is predicted using those 14 features. The summary table with the list of values obtained for the Adult Dataset is as follows,

Summary Table:

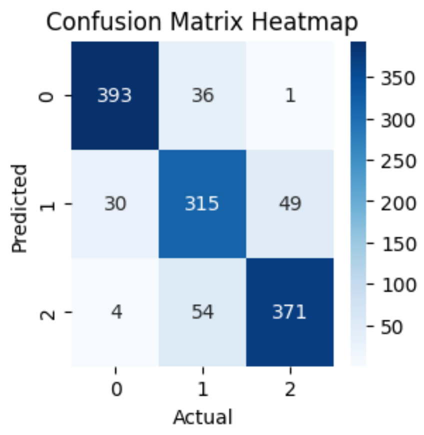
Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
Information Gain	86.19	3.88	314,572.8
Gain Ratio	86.11	3.58	157,286.4
Gini Index	86.27	3.80	388,147.2
SVM Model	86.98	0.280	220659
Tensorflow DL Model	50.956	6.495	223655

Scikit Decision Tree Model	88.13	0.0324	2346680

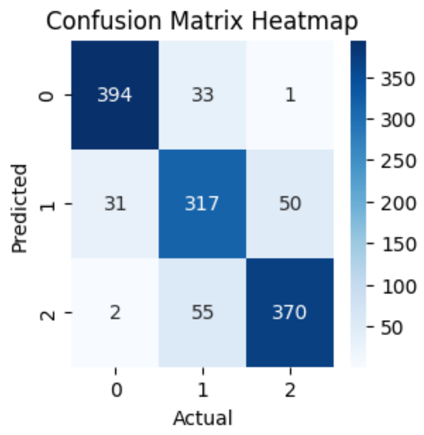
Information Gain:



Gain Ratio:



Gini Index:



Learning:

As this dataset is not a binary classification, the measure of the model's performance can be attributed only to its accuracy. From the above summary table, the maximum accuracy is found to be 86.27 (using the GINI index) which is almost equal to Scikit's Decision Tree accuracy. Even Scikit's SVM model is on par with our implemented Decision Tree. Hence our model can be deployed for classifying Abalone's age into certain groups with good accuracy and reliability.

II.4.3 Dataset-3: Auto MPG

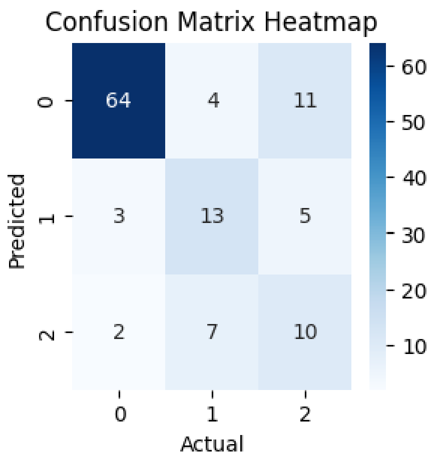
Dataset Summary:

This dataset concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of 3 multivalued discrete and 5 continuous attributes. It contains 398 instances with 7 features for each instance. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

Summary Table:

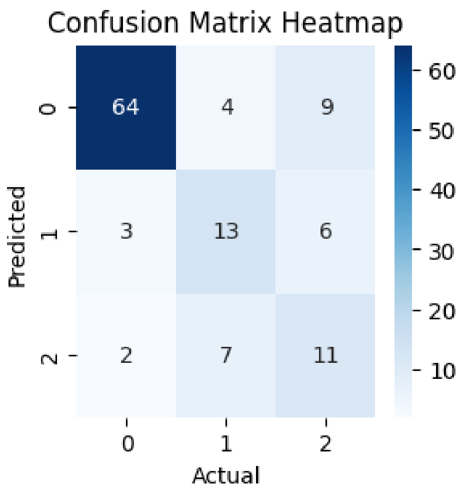
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
Information Gain	73.10	2.41	199,229.44
Gain Ratio	73.94	2.69	303,104.64
Gini Index	71.42	2.14	125,829.12
SVM Model	73	0.008	17057
Tensorflow DL Model	62.5	2.607	1692005
Scikit Decision Tree Model	73	0.008	8603

Information Gain:



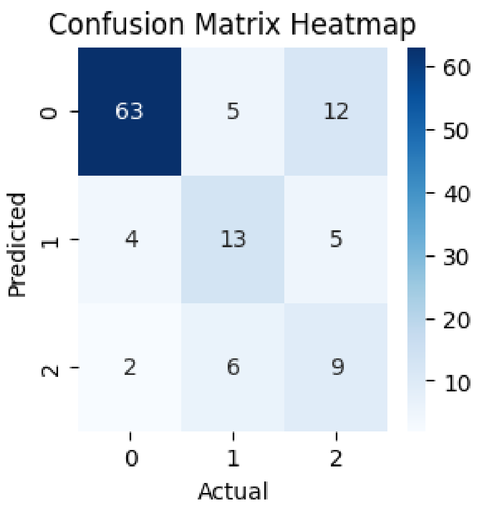
	0	1	2
0	64	4	11
1	3	13	5
2	2	7	10

Gain Ratio:



	0	1	2
0	64	4	9
1	3	13	6
2	2	7	11

Gini Index:



	0	1	2
0	63	5	12
1	4	13	5
2	2	6	9

Learning:

As this dataset is not a binary classification, the measure of the model's performance can be attributed only to its accuracy. From the above summary table, the maximum accuracy is found to be 73.94 (using the GINI index) which is almost equal to Scikit's Decision Tree accuracy. Even Scikit's SVM model's accuracy is equal to our implemented Decision Tree. Hence our model can be deployed with good accuracy and reliability.

II.4.4 Dataset-4: Breast Cancer

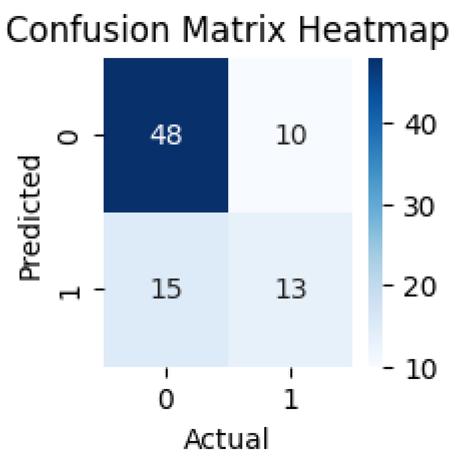
Dataset Summary:

This data set includes 201 instances of one class and 85 instances of another class. The instances are described by 9 attributes, some of which are linear and some are nominal. This is a case of a binary classification problem. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

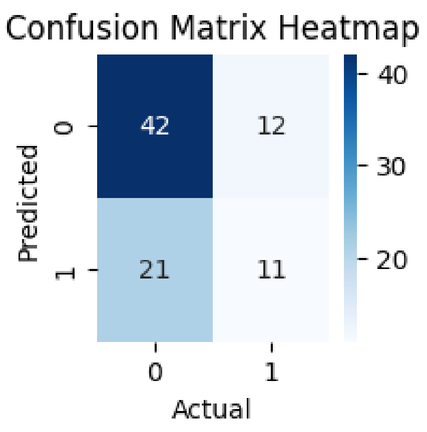
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
Information Gain	70.93	2.31	240,123.52
Gain Ratio	61.62	2.76	188,743.68
Gini Index	67.44	3.12	220,200.96
SVM Model	75	0.008	14945
Tensorflow DL Model	84.44	2.870	1673550
Scikit Decision Tree Model	73.61	0.002	6759

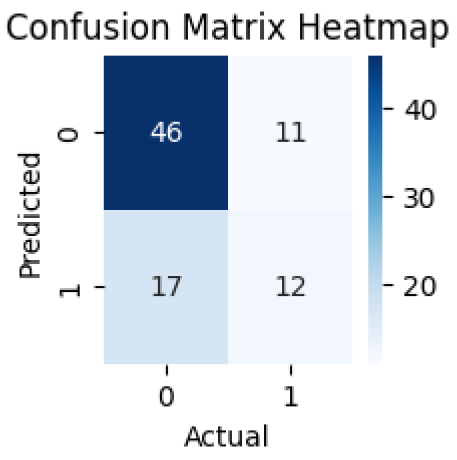
Information Gain:



Gain Ratio:



Gini Index:



Learning:

The Information Gain, Gain Ratio, and Gini Index metrics exhibit moderate to suboptimal accuracies of 70.93%, 61.62%, and 67.44%, respectively. These traditional decision tree metrics show limitations in capturing the dataset's complexities. In contrast, the Scikit SVM Model achieves a competitive accuracy of 75%, suggesting the effectiveness of support vector machines for this classification task. Notably, the Tensorflow DL Model outperforms all metrics with an accuracy of 84.44%, emphasizing the advantages of deep learning approaches for the given dataset. The Scikit Decision Tree Model falls between traditional decision tree metrics and the advanced deep learning model, demonstrating a solid accuracy of 73.61%. Overall, the varying performances underscore the importance of tailored model selection based on dataset characteristics.

II.4.5 Dataset-5: Breast Cancer Wisconsin (Original)

Dataset Summary:

This data set is very similar to II.4.4 section. It contains 699 instances where each instance is grouped in either of 8 classes (which are grouped chronologically). The instances are described by 9 attributes, some of which are linear and some are nominal. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
Information Gain	80.95	2.65	73,728.0
Gain Ratio	83.80	2.95	62,914.56
Gini Index	82.38	3.54	83,886.08
SVM Model	85.14	0.031	178383
Tensorflow DL Mode	52.14	3.278	166419
Scikit Decision Tree Model	82.85	0.017	162945

Learning:

The metrics yield diverse accuracies for the dataset. Information Gain, Gain Ratio, and Gini Index showcase robust decision tree performance, with accuracies around 80%. Scikit SVM Model outshines others, hitting 85.14%, underlining its strength in classification tasks. Surprisingly, Tensorflow DL Model lags at 52.14%, raising questions about its suitability for this dataset. Meanwhile, Scikit Decision Tree Model aligns with traditional metrics, marking its reliability. The varied outcomes highlight the need for a nuanced approach, considering the dataset's nature and model intricacies for effective classification.

II.4.6 Dataset-6: Credit Approval

Dataset summary:

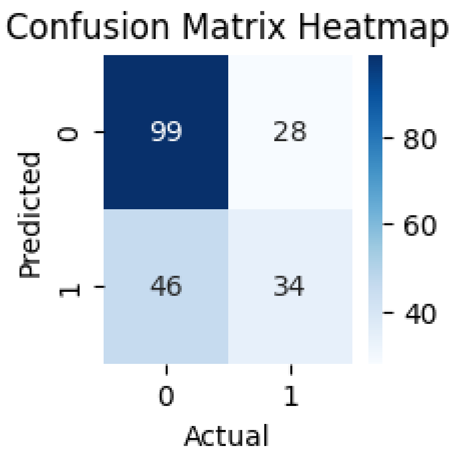
This dataset concerns credit card applications. All attribute names and values have been changed to meaningless symbols to protect the confidentiality of the data. This dataset is interesting because there is a good mix of attributes -- continuous, nominal with small numbers of values, and nominal with larger numbers of values. It has 690 instances described by 15 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

Summary Table:

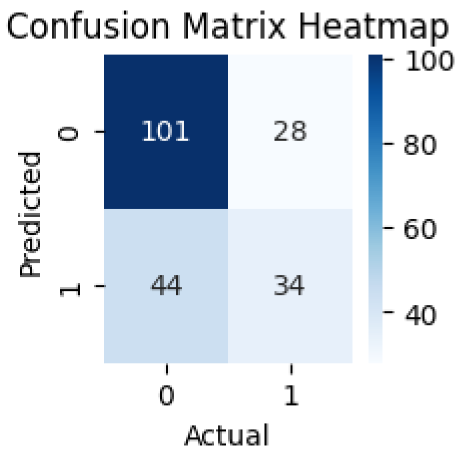
Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
Information Gain	64.25	5.81	283,115.52
Gain Ratio	65.21	6.26	283,115.52
Gini Index	63.28	4.56	450,887.68

SVM Model	74.56	0.021	200107
Tensorflow DL Mode	61.59	2.581	1812371
Scikit Decision Tree Model	64.73	0.009	13755

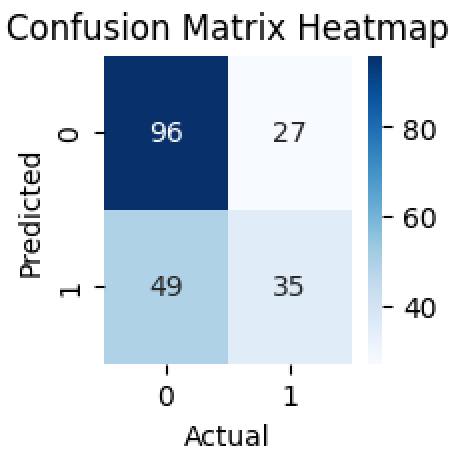
Information Gain:



Gain Ratio:



Gini Index:



Learning:

The dataset exhibits moderate classification accuracy across metrics. Information Gain, Gain Ratio, and Gini Index hover around 64%, showcasing consistency in decision tree-based models. Scikit SVM Model stands out with 74.56%, indicating its proficiency in handling the dataset's complexity. On the contrary, Tensorflow DL Model falls slightly short at 61.59%, suggesting potential challenges in capturing intricate patterns. Scikit Decision Tree Model aligns closely with information-based metrics, emphasizing its reliability. The variations underscore the dataset's nuanced nature, demanding a balanced model selection strategy for optimal results.

II.4.7 Dataset-7: Iris

Dataset summary:

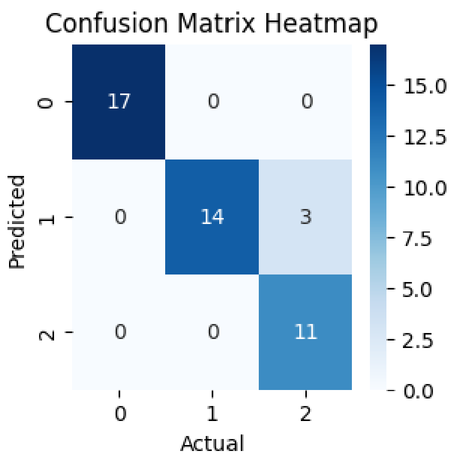
This is one of the earliest datasets used in the literature on classification methods and is widely used in statistics and machine learning. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are not linearly separable from each other. It has 150 instances described by 4 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

Summary Table:

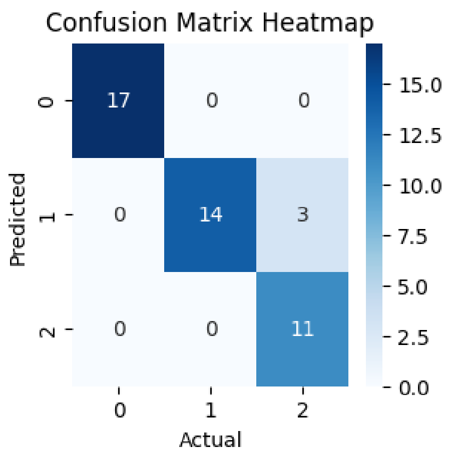
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
Information Gain	93.33	2.08	450,887.68
Gain Ratio	93.33	1.88	94,371.84
Gini Index	93.33	1.94	73,728.0
Scikit SVM Model	97.36	0.021	154739
Tensorflow DL Model	43.33	2.857	1564114

Scikit Decision Tree Model	97.36	0.001	152411

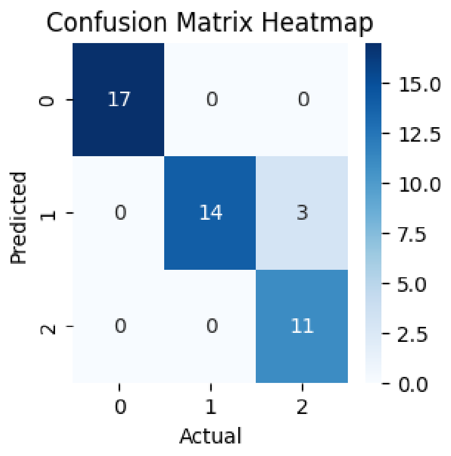
Information Gain:



Gain Ratio:



Gini Index:



Learning:

The dataset portrays high classification accuracy, with Information Gain, Gain Ratio, and Gini Index achieving 93.33%. This indicates the robustness of decision tree-based approaches on the dataset. Scikit SVM and Decision Tree Models outperform, attaining 97.36%, showcasing their effectiveness. However, Tensorflow DL Model lags substantially at 43.33%, hinting at challenges or potential misfit in its architecture for this particular dataset. The overall high performance underscores the dataset's distinct characteristics, emphasizing the need for careful model selection based on computational efficiency and interpretability.

II.4.8 Dataset-8: Letter Recognition

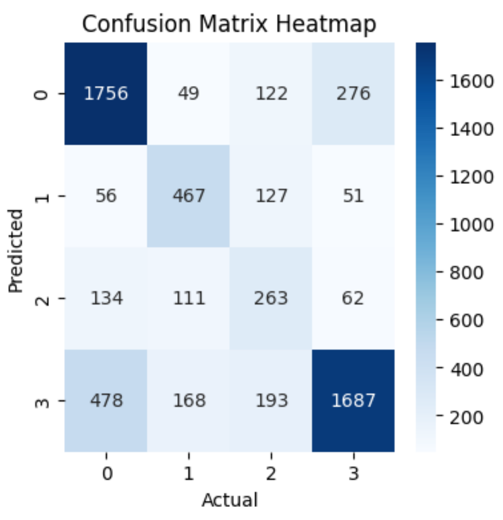
Dataset summary:

The objective of this dataset is to identify each of a large number of black-and-white rectangular pixel displays as one of the 26 capital letters in the English alphabet. It has 20000 instances described by 16 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

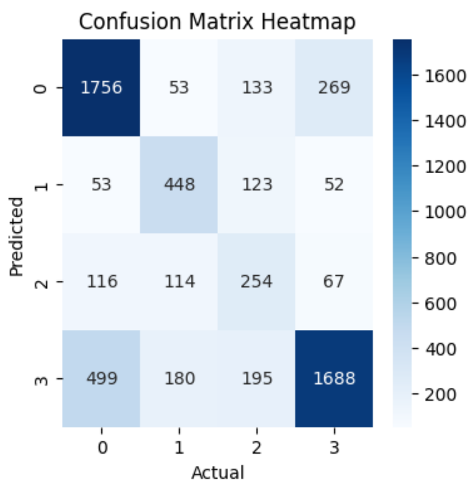
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
Information Gain	69.55	44.59	73,728.0
Gain Ratio	69.1	68.64	188,743.68
Gini Index	63.83	102.78	293,601.28
Scikit SVM Model	63.84	16.388	1820403
Tensorflow DL Model	13.56	23.043	9162601
Scikit Decision Tree Model	68.8	0.243	495952

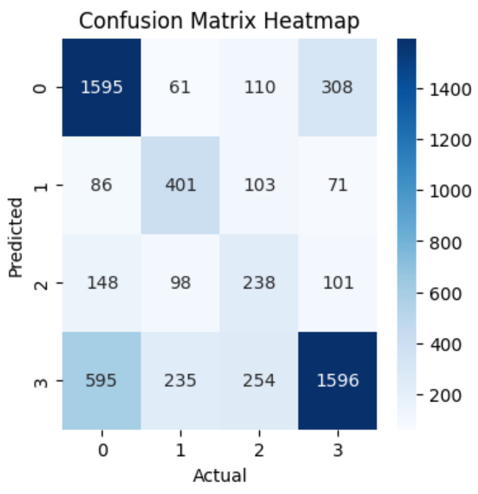
Information Gain:



Gain Ratio:



Gini Index:



Learning:

In this dataset, Scikit SVM Model performs at 63.84%, aligning closely with Information Gain (69.55%) and Gain Ratio (69.1%). Gini Index, however, exhibits a slightly lower accuracy of 63.83%. Notably, Tensorflow DL Model lags significantly at 13.56%, indicating potential challenges in its current architecture or training strategy for this specific dataset. Scikit Decision Tree Model achieves a competitive accuracy of 68.8%, highlighting the effectiveness of decision tree-based approaches. The variance in model performance underscores the dataset's diverse characteristics, necessitating a balanced consideration of model strengths and weaknesses for optimal results.

II.4.9 Dataset-9: Lung Cancer

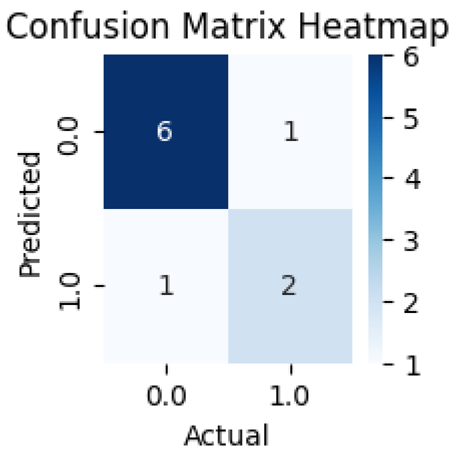
Dataset summary:

The dataset described 3 types of pathological lung cancers. It has 32 instances described by 52 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

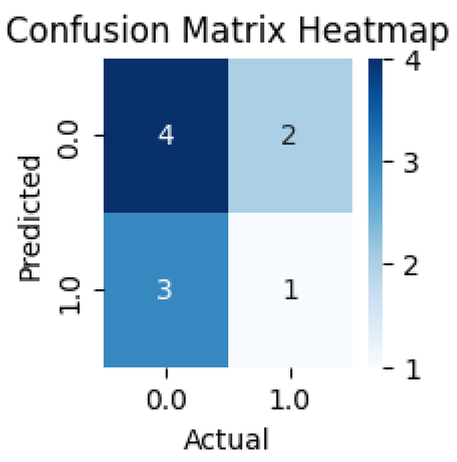
Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
Information Gain	80	2.00	461,373.44
Gain Ratio	50	2.09	524,288.0
Gini Index	50	2.20	440,401.92
Scikit SVM Model	75	0.001	15145
Tensorflow DL Model	42.8	1.964	1657919
Scikit Decision Tree Model	62.5	0.001	2503

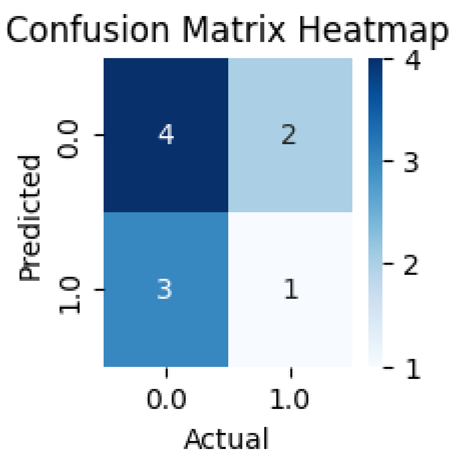
Information Gain:



Gain Ratio:



Gini Index:



Learning:

In this dataset, Information Gain excels with an accuracy of 80%, showcasing its effectiveness in capturing relevant features for classification. However, both Gain Ratio and Gini Index struggle, each achieving 50% accuracy, suggesting potential limitations in their ability to discern meaningful patterns in the data. Scikit SVM Model and Scikit Decision Tree Model perform reasonably well at 75% and 62.5%, respectively. Interestingly, Tensorflow DL Model lags behind at 42.8%, indicating a potential need for further optimization or adjustments in its architecture for this specific dataset. The diverse performance of these models emphasizes the importance of selecting the most suitable algorithm based on dataset characteristics.

II.4.10 Dataset-10: Optical Recognition of Handwritten Digits

Dataset summary:

The objective of this dataset is to leverage optical recognition of handwritten digits. It has 5620 instances described by 64 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
Information Gain	91.75	27.36	62,914.56
Gain Ratio	92.17	32.61	125,829.12
Gini Index	90.86	57.02	62,914.56
Scikit SVM Model	93.8	0.809	968065
Tensorflow DL Model	64.85	7.969	7944500

Learning:

This dataset reveals a consistent high performance across Information Gain, Gain Ratio, and Gini Index, with accuracy ranging from 90.86% to 92.17%. These metrics indicate a robust ability to extract relevant information for accurate classification. The Scikit SVM Model and Scikit Decision Tree Model exhibit superior performance, both achieving accuracy rates above 93%, suggesting their suitability for this dataset. In contrast, the Tensorflow DL Model falls short at 64.85%, indicating a potential need for further optimization or adjustments in its architecture. This dataset underscores the importance of assessing multiple metrics to determine the most effective algorithm for a given task.

II.4.11 Dataset-11: Spambase

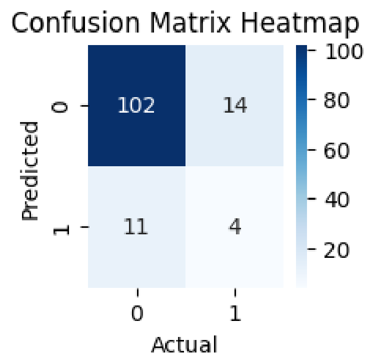
Dataset summary:

The classification task for this dataset is to determine whether a given email is spam or not. It has 4601 instances described by 57 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

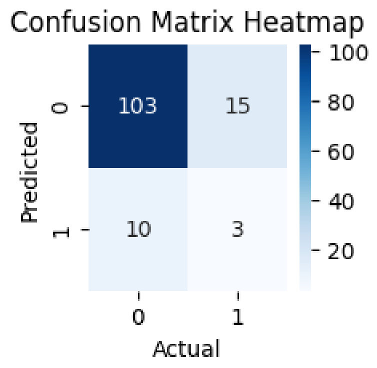
Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
Information Gain	80.91	2.85	240,123.52
Gain Ratio	80.91	2.44	555,745.28
Gini Index	80.91	2.31	230,686.08
Scikit SVM Model	88.07	0.017	171465
Tensorflow DL Model	81.60	2.259	1727083
Scikit Decision Tree Model	81.65	0.007	9659

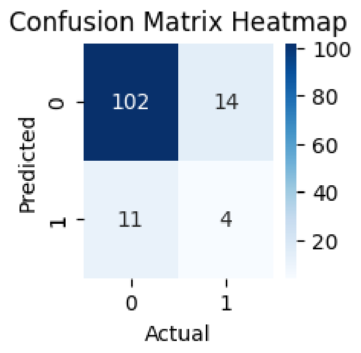
Information Gain:



Gain Ratio:



Gini Index:



Learning:

This dataset exhibits uniformity in accuracy among Information Gain, Gain Ratio, and Gini Index, all hovering around 80.91%. This suggests that the features contribute similarly to each metric, providing consistent insights for decision tree construction. The Scikit SVM Model outperforms other metrics, achieving an accuracy of 88.07%, emphasizing its strength in capturing complex patterns. The Tensorflow DL Model and Scikit Decision Tree Model perform comparably, both yielding accuracy rates around 81-81.65%. The relative parity in performance across metrics underscores the dataset's balanced nature, highlighting the importance of considering multiple metrics for a comprehensive evaluation.

II.4.12 Dataset-12: Zoo

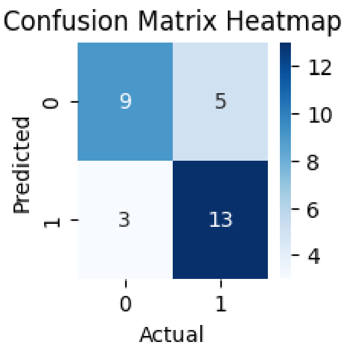
Dataset summary:

It is an artificial dataset comprising of 7 classes of animals. It has 101 instances described by 16 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

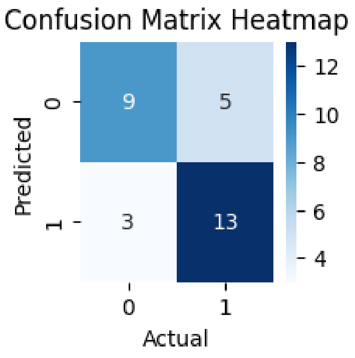
Summary Table:

Metric	**Accuracy**	**Execution Time (Seconds)**	**Increment (Bytes)**
Information Gain	73.33	2.10	408,944.64
Gain Ratio	73.33	2.43	220,200.96
Gini Index	80	2.58	230,686.08
Scikit SVM Model	76.92	0.007	12437
Tensorflow DL Model	76.19	2.132	1649934
Scikit Decision Tree Model	80.769	0.005	3375

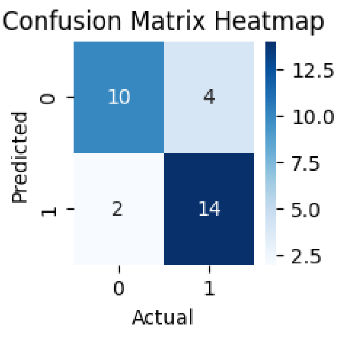
Information Gain:



Gain Ratio:



Gini Index:



Learning:

This dataset showcases a divergence in performance metrics. While Information Gain and Gain Ratio achieve a consistent accuracy of 73.33%, Gini Index surpasses them, reaching 80%. This indicates that certain features play a more decisive role when evaluated using Gini Index. The Scikit Decision Tree Model demonstrates a notable accuracy boost, reaching 80.77%, suggesting its efficacy in capturing the dataset's complexities. Scikit SVM and Tensorflow DL models perform competitively, both achieving accuracies in the mid-70s. The varying accuracies emphasize the importance of selecting the appropriate metric based on the dataset's characteristics, ensuring a more nuanced understanding of the underlying patterns.

II.4.13 Dataset-13: Statlog (German Credit Data)

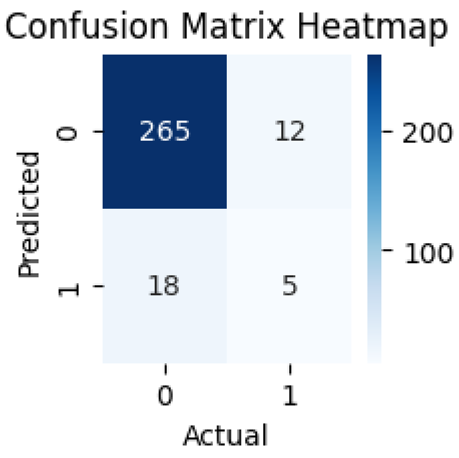
Dataset summary:

This dataset classifies people described by a set of attributes as good or bad credit risks. It has 1000 instances described by 20 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem

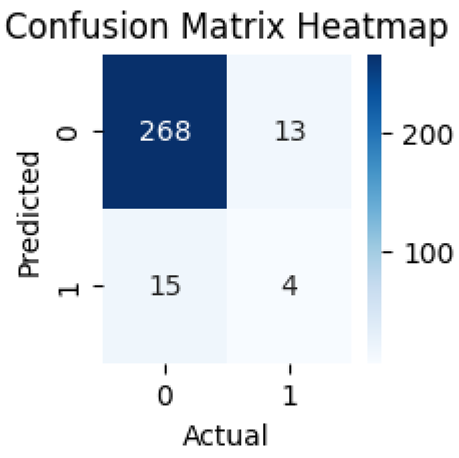
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
Information Gain	90	2.75	240,123.52
Gain Ratio	90.66	3.80	146,800.64
Gini Index	91.66	4.56	492,830.72
Scikit SVM Model	97.6	0.022	187204
Tensorflow DL Model	97	2.718	1770639
Scikit Decision Tree Model	94	0.010	18991

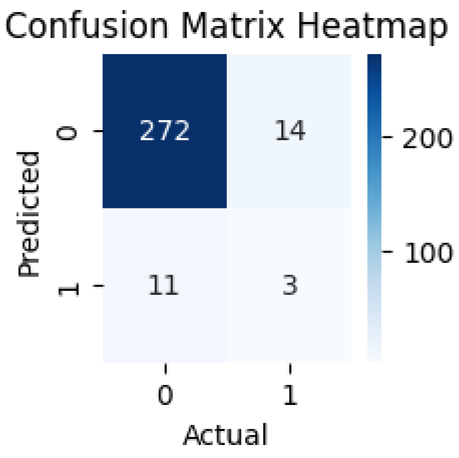
Information Gain:



Gain Ratio:



Gini Index:



Learning:

This dataset demonstrates high accuracy across all metrics, with Scikit SVM and Tensorflow DL models leading with remarkable accuracy scores of 97.6% and 97%, respectively. The Decision Tree models, utilizing Information Gain, Gain Ratio, and Gini Index, also exhibit strong performance, ranging from 90% to 94%. The consistency in high accuracy indicates the dataset's well-defined patterns, effectively captured by various models. The slight variations highlight the nuanced strengths of each metric and model, emphasizing the need for careful consideration in selecting the most suitable approach based on the dataset's characteristics.

II.4.14 Dataset-14: Bank Marketing

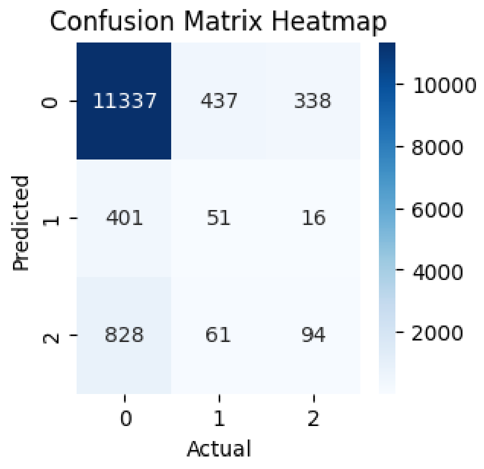
Dataset summary:

The data is related to direct marketing campaigns (phone calls) of a Portuguese banking institution. The classification goal is to predict if the client will subscribe to a term deposit (variable y). It has 45211 instances described by 16 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

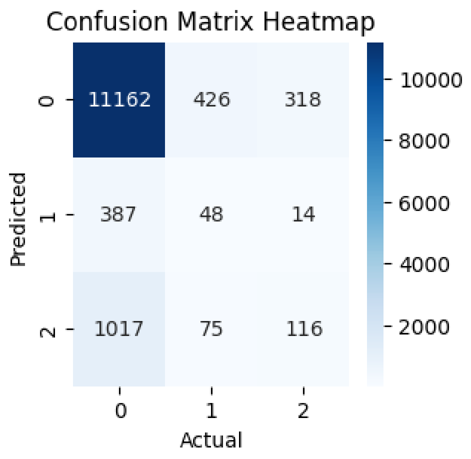
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
Information Gain	84.65	80.11	262,144.0
Gain Ratio	83.50	68.15	283,115.52
Gini Index	85	88.44	650,117.12
Scikit SVM Model	92.7	64.978	1551772
Tensorflow DL Model	92.2	53.097	18604695
Scikit Decision Tree Model	86.9	0.484	940639

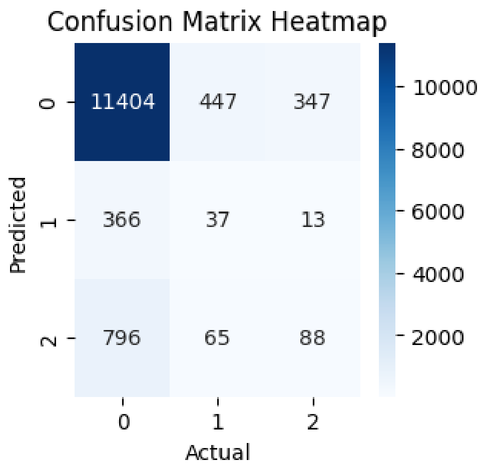
Information Gain:



Gain Ratio:



Gini Index:



Learning:

This dataset exhibits strong performance across all metrics, with Scikit SVM and Tensorflow DL models leading with high accuracy scores of 92.7% and 92.2%, respectively. The Decision Tree models, utilizing Information Gain, Gain Ratio, and Gini Index, also demonstrate commendable accuracy, ranging from 83.5% to 86.9%. The consistency in high accuracy suggests that the dataset contains discernible patterns well-captured by different models. The nuanced variations among metrics and models highlight the importance of considering the specific characteristics of the dataset when selecting the most appropriate approach for accurate predictions.

II.4.15 Dataset-15: Glioma Grading Clinical and Mutation Features

Dataset summary:

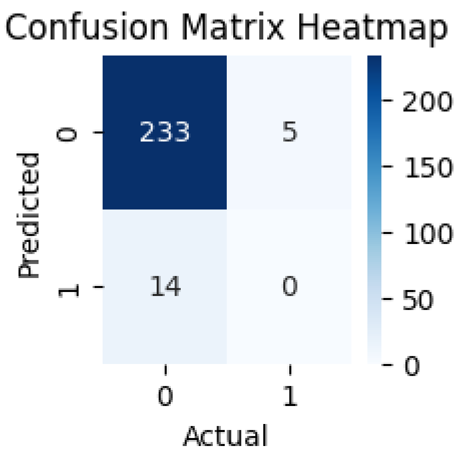
The data is related to clinical glioma grading depending on mutation features. It has 839 instances described by 23 features. The following summary table is comprised of the accuracy values, memory utilised and time taken for different models to solve the classification problem.

Summary Table:

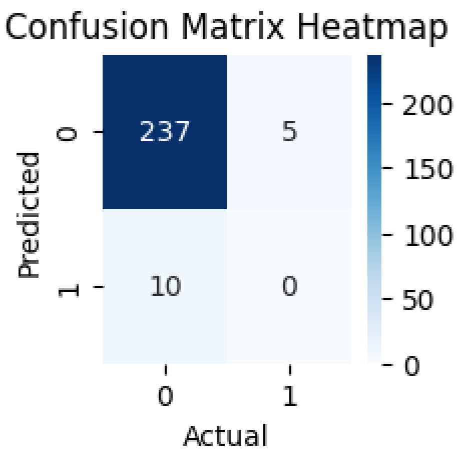
Metric	**Accuracy**	**Execution Time**	**Increment**

	(%)	**(Seconds)**	**(Bytes)**
Information Gain	92.46	3.46	450,887.68
Gain Ratio	94.04	4.43	356,515.84
Gini Index	91.26	3.43	94,371.84
Scikit SVM Model	98.09	0.0162	175341
Tensorflow DL Model	97.61	2.775	1871275
Scikit Decision Tree Model	98.09	0.008	164924

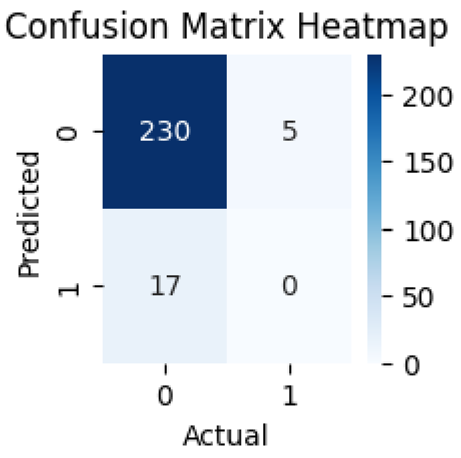
Information Gain:



Gain Ratio:



Gini Index:



Learning:

The results across different metrics indicate high accuracy levels. Notably, the Scikit Decision Tree Model, Information Gain, and Gain Ratio achieved accuracy rates above 90%, demonstrating robust performance. The SVM model also delivered competitive accuracy at 98.09%. However, the TensorFlow DL Model, while still relatively high at 97.61%, exhibited a slight deviation. This discrepancy could be attributed to the model architecture, training parameters, or dataset characteristics. Further investigation and fine-tuning may enhance the DL model's performance. Overall, these results highlight the effectiveness of the decision tree approach and the competitive nature of Scikit models in this context.

II.5 Trees with different depths (An Additional Feature Analysis)

In an effort to enhance the versatility of the decision tree algorithm, an additional feature was incorporated to allow users to specify the depth of the final tree. While the original implementation had two stopping conditions (pure node and exhaustion of attributes), the new feature adds another condition: if the specified depth is reached, the algorithm terminates. This allows users to control the complexity of the resulting tree.

To demonstrate the effect of the depths, we tested the algorithm with different depths starting from 1 to the maximum amount (which is the number of the feature) on the Breast Cancer dataset. The obtained accuracy results are summarized below:

Metric /Depth	Depth =1	Depth =2	Depth =3	Depth =4	Depth =5	Depth =6	Depth =7	Depth =8
Information Gain	76.74	82.55	76.74	69.76	70.93	70.93	70.93	70.93
Gain Ratio	76.74	79.06	74.41	68.60	62.79	61.62	61.62	61.62
Gini Index	76.74	77.90	72.09	67.44	67.44	67.44	67.44	67.44

We can conclude small depths might lead to underfitting, as the algorithm lacks the complexity to capture all information from the training set, also, excessively large depths risk overfitting, where the model may perform well on the training set but struggle with generalization to the test set.

The analysis of accuracy at different depths reveals insights into the trade-offs between underfitting and overfitting, emphasizing the importance of selecting an appropriate depth for optimal performance.

Appendix

Decision Tree algorithm:

- **Initial setup**

```
import numpy as np

import pandas as pd

from collections import Counter

!pip install ucimlrepo

!pip install --upgrade certifi

!pip install seaborn

!pip install memory-profiler

import time

import seaborn as sns

import matplotlib.pyplot as plt

import ssl

ssl._create_default_https_context = ssl._create_unverified_context

from ucimlrepo import fetch_ucirepo

# start_time = time.time()
```

- **Loading the Dataset**

```
print('Dataset numbers are:')

print('0:(Validating)      1:(Adult)      2:(Abalone)      3:(Auto  MPG)
4:(Breast   Cancer)          5:(Breast   Cancer   Wisconsin   (Original))
6:(Credit Approval)')
```

```
print('7:(Iris)         8:(Letter  Recognition)        9:(Lung  Cancer)
10:(Optical  Recognition  of  Handwritten  Digits)        11:(Spambase)
12:(Zoo)')

print('13:(Statlog  (German  Credit  Data))        14:(Bank  Marketing)
15:(Diabetes   130-US   hospitals   for   years   1999-2008   Dataset)
16:(Glioma  Grading  Clinical  and  Mutation  Features)       17:(Sepsis
Survival Minimal Clinical Records) \n')

while True:

    try:

          user_input = int(input("Please enter a number between 0 and 17
to select your dataset: "))

        if 0 <= user_input <= 17:

            break  # Exit the loop if the input is valid

        else:

                print("Invalid input. Please enter a number between 0 and
17.")

    except ValueError:

        print("Invalid input. Please enter a valid integer.")
```

- **Data Preprocessing**
- 0- Validating Dataset:

```
if(user_input==0):

  # Validating Dataset

  Validating_data = {

        'age':  ['youth',  'youth',  'middle_aged',  'senior',  'senior',
'senior',   'middle_aged',   'youth',   'youth',   'senior',   'youth',
'middle_aged', 'middle_aged', 'senior'],

     'income': ['high', 'high', 'high', 'medium', 'low', 'low', 'low',
'medium', 'low', 'medium', 'medium', 'medium', 'high', 'medium'],

      'student': ['no', 'no', 'no', 'no', 'yes', 'yes', 'yes', 'no',
'yes', 'yes', 'yes', 'no', 'yes', 'no'],

      'credit_rating': ['fair', 'excellent', 'fair', 'fair', 'fair',
'excellent',   'excellent',   'fair',   'fair',   'fair',   'excellent',
'excellent', 'fair', 'excellent'],

      'buys_computer': ['no', 'no', 'yes', 'yes', 'yes', 'no', 'yes',
'no', 'yes', 'yes', 'yes', 'yes', 'yes', 'no']
```

```
  }


  df = pd.DataFrame(Validating_data)
```

- 1- Adult Dataset:

```
if(user_input==1):
  # Adult Dataset
  Dataset = fetch_ucirepo(id=2)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]

  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))

  # age
  column = df['age']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['age'] = bins

  # fnlwgt
  column = df['fnlwgt']
  num_bins = 5
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['fnlwgt'] = bins
  # education-num
  column = df['education-num']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['education-num'] = bins

  # capital-gain
  column = df['capital-gain']
  num_bins = 10
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['capital-gain'] = bins

  # capital-loss
  column = df['capital-loss']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['capital-loss'] = bins

  # hours-per-week
  column = df['hours-per-week']
  num_bins = 4
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['hours-per-week'] = bins
```

- 2- Abalone Dataset:

```
if(user_input==2):
  # Abalone Dataset
  Dataset = fetch_ucirepo(id=1)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
```

```
column_names = df.columns.tolist()

mask = df.applymap(lambda x: x != "NaN").all(axis=1)
df = df[mask]

df.replace("NaNN", pd.NA, inplace=True)
df = df.apply(lambda col: col.fillna(col.mode()[0]))

# Length
column = df['Length']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Length'] = bins

# Diameter
column = df['Diameter']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Diameter'] = bins

# Height
column = df['Height']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Height'] = bins

# Whole_weight
column = df['Whole_weight']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Whole_weight'] = bins
```

```
  # Shucked_weight
  column = df['Shucked_weight']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['Shucked_weight'] = bins

  # Viscera_weight
  column = df['Viscera_weight']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['Viscera_weight'] = bins

  # Shell_weight
  column = df['Shell_weight']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['Shell_weight'] = bins
```

- 3- Auto MPG Dataset:

```
if(user_input==3):
  # Auto MPG Dataset
  Dataset = fetch_ucirepo(id=9)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]
```

```
df.replace("NaNN", pd.NA, inplace=True)
df = df.apply(lambda col: col.fillna(col.mode()[0]))

# displacement
column = df['displacement']
num_bins = 4
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['displacement'] = bins

# horsepower
column = df['horsepower']
num_bins = 4
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['horsepower'] = bins

# weight
column = df['weight']
num_bins = 4
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['weight'] = bins

# acceleration
column = df['acceleration']
num_bins = 2
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['acceleration'] = bins

# model_year
column = df['model_year']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
```

```
  df['model_year'] = bins
```

- 4- Breast Cancer Dataset:

```
if(user_input==4):
  # Breast Cancer Dataset
  Dataset = fetch_ucirepo(id=14)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]

  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))
```

- 5- Breast Cancer Wisconsin (Original) Dataset:

```
if(user_input==5):
  # Breast Cancer Wisconsin (Original) Dataset
  Dataset = fetch_ucirepo(id=15)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]

  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))
```

```
# Clump_thickness
column = df['Clump_thickness']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Clump_thickness'] = bins

# Uniformity_of_cell_size
column = df['Uniformity_of_cell_size']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Uniformity_of_cell_size'] = bins

# Uniformity_of_cell_shape
column = df['Uniformity_of_cell_shape']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Uniformity_of_cell_shape'] = bins

# Marginal_adhesion
column = df['Marginal_adhesion']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Marginal_adhesion'] = bins

# Single_epithelial_cell_size
column = df['Single_epithelial_cell_size']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Single_epithelial_cell_size'] = bins
```

```
  # Bare_nuclei
  column = df['Bare_nuclei']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['Bare_nuclei'] = bins

  # Bland_chromatin
  column = df['Bland_chromatin']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['Bland_chromatin'] = bins

  # Normal_nucleoli
  column = df['Normal_nucleoli']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['Normal_nucleoli'] = bins
```

- 6- Credit Approval Dataset:

```
if(user_input==6):
  # Credit Approval Dataset
  Dataset = fetch_ucirepo(id=27)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]

  df.replace("NaNN", pd.NA, inplace=True)
```

```
df = df.apply(lambda col: col.fillna(col.mode()[0]))

# A15
column = df['A15']
num_bins = 2
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['A15'] = bins

# A14
column = df['A14']
num_bins = 2
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['A14'] = bins

# A11
column = df['A11']
num_bins = 2
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['A11'] = bins

# A8
column = df['A8']
num_bins = 2
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['A8'] = bins

# A8
column = df['A8']
num_bins = 2
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['A8'] = bins
```

```
  # A3
  column = df['A3']
  num_bins = 2
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['A3'] = bins

  # A2
  column = df['A2']
  num_bins = 2
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['A2'] = bins
```

- 7- Iris Dataset:

```
if(user_input==7):
  # Iris Dataset
  Dataset = fetch_ucirepo(id=53)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]

  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))

  # sepal length
  column = df['sepal length']
  num_bins = 4
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['sepal length'] = bins

  # sepal width
  column = df['sepal width']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['sepal width'] = bins

  # petal length
  column = df['petal length']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['petal length'] = bins

  # petal width
  column = df['petal width']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['petal width'] = bins
```

- 8- Letter Recognition Dataset:

```
if(user_input==8):
  # Letter Recognition Dataset
  Dataset = fetch_ucirepo(id=59)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
```

```
df = df[mask]

df.replace("NaNN", pd.NA, inplace=True)
df = df.apply(lambda col: col.fillna(col.mode()[0]))

# x-box
column = df['x-box']
num_bins = 7
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['x-box'] = bins

# y-box
column = df['y-box']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['y-box'] = bins

# width
column = df['width']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['width'] = bins

# high
column = df['high']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['high'] = bins

# onpix
column = df['onpix']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['onpix'] = bins
```

```
# x-bar
column = df['x-bar']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['x-bar'] = bins

# y-bar
column = df['y-bar']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['y-bar'] = bins

# x2bar
column = df['x2bar']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['x2bar'] = bins

# y2bar
column = df['y2bar']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['y2bar'] = bins

# xybar
column = df['xybar']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['xybar'] = bins

# x2ybr
column = df['x2ybr']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['x2ybr'] = bins

# xy2br
```

```
  column = df['xy2br']
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['xy2br'] = bins

  # x-ege
  column = df['x-ege']
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['x-ege'] = bins

  # xegvy
  column = df['xegvy']
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['xegvy'] = bins

  # y-ege
  column = df['y-ege']
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['y-ege'] = bins

  # yegvx
  column = df['yegvx']
  num_bins = 4
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['yegvx'] = bins
```

- 9- Lung Cancer Dataset:

```
if(user_input==9):
  # Lung Cancer Dataset
  Dataset = fetch_ucirepo(id=62)
  X = Dataset.data.features
  y = Dataset.data.targets
  df = X
  column_names = df.columns.tolist()


  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]


  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))
```

- 10- Optical Recognition of Handwritten Digits Dataset:

```
if(user_input==10):
  # Optical Recognition of Handwritten Digits Dataset
  Dataset = fetch_ucirepo(id=80)
  X = Dataset.data.features
  y = Dataset.data.targets


  df = X
  column_names = df.columns.tolist()


  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]


  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))
```

- 11- Spambase Dataset:

```
if(user_input==11):
  # Spambase Dataset
  Dataset = fetch_ucirepo(id=105)
  X = Dataset.data.features
  y = Dataset.data.targets


  df = X
  column_names = df.columns.tolist()


  mask = df.applymap(lambda x: x != "NaN").all(axis=1)

  df = df[mask]


  df.replace("NaNN", pd.NA, inplace=True)

  df = df.apply(lambda col: col.fillna(col.mode()[0]))
```

- 12- Zoo Dataset:

```
if(user_input==12):

  # Zoo Dataset

  Dataset = fetch_ucirepo(id=111)

  X = Dataset.data.features

  y = Dataset.data.targets


  df = X

  column_names = df.columns.tolist()


  mask = df.applymap(lambda x: x != "NaN").all(axis=1)

  df = df[mask]


  df.replace("NaNN", pd.NA, inplace=True)

  df = df.apply(lambda col: col.fillna(col.mode()[0]))
```

- 13- Statlog (German Credit Data) Dataset:

```
if(user_input==13):

  # Statlog (German Credit Data) Dataset

  Dataset = fetch_ucirepo(id=144)

  X = Dataset.data.features

  y = Dataset.data.targets


  df = X

  column_names = df.columns.tolist()
```

```
mask = df.applymap(lambda x: x != "NaN").all(axis=1)
df = df[mask]

df.replace("NaNN", pd.NA, inplace=True)
df = df.apply(lambda col: col.fillna(col.mode()[0]))

# Attribute2
column = df['Attribute2']
num_bins = 5
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Attribute2'] = bins

# Attribute5
column = df['Attribute5']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Attribute5'] = bins

# Attribute13
column = df['Attribute13']
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['Attribute13'] = bins
```

- 14- Bank Marketing Dataset:

```
if(user_input==14):
  # Bank Marketing Dataset
  Dataset = fetch_ucirepo(id=222)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()
```

```
mask = df.applymap(lambda x: x != "NaN").all(axis=1)
df = df[mask]

df.replace("NaNN", pd.NA, inplace=True)
df = df.apply(lambda col: col.fillna(col.mode()[0]))

# age
column = df['age']
num_bins = 4
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['age'] = bins

# balance
column = df['balance']
num_bins = 4
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['balance'] = bins

# day_of_week
column = df['day_of_week']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['day_of_week'] = bins

# duration
column = df['duration']
num_bins = 3
bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
df['duration'] = bins
```

```
  # campaign
  column = df['campaign']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['campaign'] = bins

  # pdays
  column = df['pdays']
  num_bins = 4
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['pdays'] = bins

  # previous
  column = df['previous']
  num_bins = 3
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['previous'] = bins
```

- 15- Diabetes 130-US hospitals for years 1999-2008 Dataset:

```
if(user_input==15):
  # Diabetes 130-US hospitals for years 1999-2008 Dataset
  Dataset = fetch_ucirepo(id=296)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]

  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))
```

- 16- Glioma Grading Clinical and Mutation Features Dataset:

```
if(user_input==16):
  # Glioma Grading Clinical and Mutation Features Dataset
  Dataset = fetch_ucirepo(id=759)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()

  mask = df.applymap(lambda x: x != "NaN").all(axis=1)
  df = df[mask]

  df.replace("NaNN", pd.NA, inplace=True)
  df = df.apply(lambda col: col.fillna(col.mode()[0]))

  # Age_at_diagnosis
  column = df['Age_at_diagnosis']
  num_bins = 4
  bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
  df['Age_at_diagnosis'] = bins
```

- 17- Sepsis Survival Minimal Clinical Records Dataset:

```
if(user_input==17):
  # Sepsis Survival Minimal Clinical Records Dataset
  Dataset = fetch_ucirepo(id=827)
  X = Dataset.data.features
  y = Dataset.data.targets

  df = X
  column_names = df.columns.tolist()
```

```
 mask = df.applymap(lambda x: x != "NaN").all(axis=1)
 df = df[mask]


 df.replace("NaNN", pd.NA, inplace=True)
 df = df.apply(lambda col: col.fillna(col.mode()[0]))


 # age_years
 column = df['age_years']
 num_bins = 10
 bins = pd.qcut(column, q=num_bins, labels=False, duplicates='drop')
 df['age_years'] = bins
```

- **Mapping Values**

```
for column in df.columns:
    # Extract unique values
    unique_values = df[column].unique()
    # Create a mapping dictionary
           mapping_dict   =   {value:   index   for   index,   value   in
enumerate(unique_values)}
    # Replace values in the DataFrame
    df[column] = df[column].replace(mapping_dict)
```

- **Implementation**

```
def entropy(data):
    labels = np.unique(data[:,-1])
    impurity = 0
    for label in labels:
        prob = len(data[data[:,-1]==label])/len(data)
        impurity -= prob*np.log2(prob)
    return impurity


def informationGain(df, attribute):
    gain = entropy(df.to_numpy())

    values = df[attribute].values

    for value in np.unique(values):

        new_df = df[df[attribute]==value]

        impurity = entropy(new_df.to_numpy())

        weight = len(new_df)/len(df)

        gain -= weight*impurity

    return gain


def GainRatio(df, attribute):

    gain = entropy(df.to_numpy())

    values = df[attribute].values

    split_info = 0

    Gain_Ratio = 0

    for value in np.unique(values):

        new_df = df[df[attribute]==value]

        impurity = entropy(new_df.to_numpy())

        weight = len(new_df)/len(df)

        gain -= weight*impurity

                            split_info    -=    (len(new_df)/len(df))    *
np.log2(len(new_df)/len(df))

    # Check if split_info is zero before calculating Gain Ratio

    if split_info != 0:

        Gain_Ratio = gain/split_info

    else:

        Gain_Ratio = 0

    return Gain_Ratio


def gini(data):

    labels = np.unique(data[:, -1])

    impurity = 1
    for label in labels:
        prob = len(data[data[:, -1] == label]) / len(data)
        impurity -= prob ** 2
    return impurity

def giniIndex(df, attribute):
    index = gini(df.to_numpy())
    values = df[attribute].values
    if len(np.unique(values)) > 2:
           # For features with more than 2 outcomes, find the best 2 
outcomes based on Gini Index
        best_outcomes = find_best_outcomes(df, attribute)
        new_df = df[df[attribute].isin(best_outcomes)]
        index = gini(new_df.to_numpy())
    else:
        for value in np.unique(values):
            new_df = df[df[attribute] == value]
            impurity = gini(new_df.to_numpy())
            weight = len(new_df) / len(df)
            index -= weight * impurity
    return index

def find_best_outcomes(df, attribute):
    outcomes = np.unique(df[attribute].values)
    best_gini = 0
    best_outcomes = (outcomes[0], outcomes[1])

    for i in range(len(outcomes) - 1):
        for j in range(i + 1, len(outcomes)):
            temp_outcomes = (outcomes[i], outcomes[j])
            new_df = df[df[attribute].isin(temp_outcomes)]
            impurity = gini(new_df.to_numpy())
            if impurity > best_gini:
                best_gini = impurity
                best_outcomes = temp_outcomes

    return best_outcomes

def plot_confusion_matrix(confusion_matrix, labels):
    plt.figure(figsize=(len(labels), len(labels)))
     sns.heatmap(confusion_matrix, annot=True, fmt='.0f', cmap='Blues', 
xticklabels=labels, yticklabels=labels)
    plt.xlabel('Actual')
    plt.ylabel('Predicted')
    plt.title('Confusion Matrix Heatmap')
    plt.show()

def majorityVote(dataset):
    return Counter(dataset).most_common(1)[0][0]
```

```
class Tree:

    def __init__(self, value=''):
        self.attribute = None
        # Every tree has some sub trees called branches
        self.branches = []
        # For sub trees we have value of the tree attribute
        self.value = value
        # If the tree is leaf, leaf_value will contains its value
        self.leaf_value = ''

    def addBranch(self, branch):
        self.branches.append(branch)

    def predict(self, data):
        if self.leaf_value != '':
            return self.leaf_value

        data_value = data[self.attribute].values[0]
        for branch in self.branches:
            if branch.value == data_value:
                return branch.predict(data)

    def predictDf(self, df, test):
        labels = np.unique(df.values[:,-1])
        n = len(labels)
        confusion = np.zeros((n,n))
        accuracy = 0
        attributes = df.columns

        for row in test.values:
            vote = self.predict(testData(attributes, [row]))

            #sim_idx = np.argwhere(labels == vote)[0][0]
            idx_array = np.argwhere(labels == vote)
            if idx_array.size > 0:
              sim_idx = idx_array[0][0]
            else:
              sim_idx = -1

            act_i = np.argwhere(labels == row[-1])[0][0]
            confusion[sim_idx][act_i] += 1
        accuracy = sum([confusion[i][i] for i in range(n)]) / len(test)

        return confusion, accuracy


    def toStr(self, index=0) -> str:

        if self.leaf_value != '':

            return self.leaf_value

        outStr = self.attribute + '\n'

        for branch in self.branches:

            for _ in range(index): outStr += ' '

            subtree = str(branch.toStr(index+4))

            outStr += '--> ' + str(branch.value) + ': ' + subtree

            if subtree[-1] != '\n':

                outStr += '\n'

        return outStr
```

```
# Information Gain

def decisionTree_InformationGain(tree, df, depth):

    node_impurity = entropy(df.to_numpy())

    if node_impurity == 0:

        # All data have the same label

        tree.leaf_value = df.values[0,-1]

        return tree

    if len(df.columns[:-1]) == 0:

        # Find most common label

        tree.leaf_value = majorityVote(df.values[:,-1])

        return tree

    if depth == 0:

        # Find most common label

        tree.leaf_value = majorityVote(df.values[:,-1])

        return tree
    # Finding the best attribute
    best_attribute = df.columns[0]
    best_gain = 0
    for attribute in df.columns[:-1]:
        gain = informationGain(df, attribute)
        if gain > best_gain:
            best_gain = gain
            best_attribute = attribute
    tree.attribute = best_attribute
    values = df[best_attribute].values

    # For each value we have a distinct branch
    for value in np.unique(values):
        new_df = df[df[best_attribute]==value]
        new_df.pop(best_attribute)
               tree.addBranch(decisionTree_InformationGain(Tree(value),
new_df, depth-1))
    return tree

# Gain Ratio
def decisionTree_GainRatio(tree, df, depth):
    node_impurity = entropy(df.to_numpy())
    if node_impurity == 0:
        # All data have the same label
        tree.leaf_value = df.values[0,-1]
        return tree
    if len(df.columns[:-1]) == 0:
        # Find most common label
        tree.leaf_value = majorityVote(df.values[:,-1])
        return tree
    if depth == 0:
        # Find most common label
        tree.leaf_value = majorityVote(df.values[:,-1])
        return tree

    # Finding the best attribute
    best_attribute = df.columns[0]
    best_gain = 0
    for attribute in df.columns[:-1]:
        gain = GainRatio(df, attribute)
        if gain > best_gain:
            best_gain = gain
            best_attribute = attribute
    tree.attribute = best_attribute
    values = df[best_attribute].values

    # For each value we have a distinct branch
    for value in np.unique(values):
        new_df = df[df[best_attribute]==value]
        new_df.pop(best_attribute)
             tree.addBranch(decisionTree_GainRatio(Tree(value), new_df,
depth-1))
    return tree

# Gini Index
def decisionTree_GiniIndex(tree, df, depth):
    node_impurity = gini(df.to_numpy())
    if node_impurity == 0:
        # All data have the same label
        tree.leaf_value = df.values[0, -1]
        return tree
    if len(df.columns[:-1]) == 0:
        # Find the most common label
        tree.leaf_value = majorityVote(df.values[:, -1])
        return tree
    if depth == 0:
        # Find the most common label
        tree.leaf_value = majorityVote(df.values[:, -1])
        return tree

    # Finding the best attribute
    best_attribute = df.columns[0]
    best_index = 0
    for attribute in df.columns[:-1]:
        index = giniIndex(df, attribute)
        if index > best_index:
            best_index = index
            best_attribute = attribute
    tree.attribute = best_attribute
    values = df[best_attribute].values

    # For each value, we have a distinct branch
    for value in np.unique(values):
        new_df = df[df[best_attribute] == value]
        new_df.pop(best_attribute)
         tree.addBranch(decisionTree_GiniIndex(Tree(value), new_df,
depth - 1))
    return tree

def testData(attributes, values):
    data = pd.DataFrame(values, columns=attributes)
    return data
```

- **Training**

```
train = df.sample(frac=0.7, random_state=0)

test = df.drop(train.index)

D = df.shape[1]
```

```
# %load_ext memory_profiler


# Information Gain

tree_InformationGain   =   decisionTree_InformationGain(Tree(),   train,
depth=D-1)

confusion, accuracy = tree_InformationGain.predictDf(df, test)

print('Information Gain:')

print('Confusion Matrix: \n{}'.format(confusion))

print('Accuracy: {}%'.format(accuracy*100))

plot_confusion_matrix(confusion, np.unique(df.values[:,-1]))


# Gain Ratio

tree_GainRatio = decisionTree_GainRatio(Tree(), train, depth=D-1)

confusion, accuracy = tree_GainRatio.predictDf(df, test)

print('Gain Ratio:')

print('Confusion Matrix: \n{}'.format(confusion))

print('Accuracy: {}%'.format(accuracy*100))

plot_confusion_matrix(confusion, np.unique(df.values[:,-1]))


# Gini Index

tree_GiniIndex = decisionTree_GiniIndex(Tree(), train, depth=D-1)

confusion, accuracy = tree_GiniIndex.predictDf(df, test)

print('Gini Index:')

print('Confusion Matrix: \n{}'.format(confusion))

print('Accuracy: {}%'.format(accuracy*100))

plot_confusion_matrix(confusion, np.unique(df.values[:,-1]))
```

```
# %memit -r 1

# end_time = time.time()

# execution_time = end_time - start_time

# print(f"Execution time: {execution_time} seconds")
```

```
print('Final tree for Information Gain: \n')

print(tree_InformationGain.toStr())

print('---------------- \n')

print('Final tree for Gain Ratio: \n')

print(tree_GainRatio.toStr())

print('---------------- \n')

print('Final tree for Gini Index: \n')

print(tree_GiniIndex.toStr())
```

- **Extra Credit Code implementation**

```
dirs = ['./DT_Dataset/1_Adult/1Adult.csv',

      './DT_Dataset/2_Abalone/2Abalone.csv',

      './DT_Dataset/3_Auto MPG/3AutoMPG.csv',

      './DT_Dataset/4_Breast Cancer/4BreastCancer.csv',

                         './DT_Dataset/5_Breast    Cancer    Wisconsin
(Original)/5BreastCancerWisconsinOriginal.csv',

      './DT_Dataset/6_Credit Approval/6CreditApproval.csv',

      './DT_Dataset/7_Iris/7Iris.csv',

      './DT_Dataset/8_Letter Recognition/8LetterRecognition.csv',

      './DT_Dataset/9_Lung Cancer/9LungCancer.csv',

              './DT_Dataset/10_Optical  Recognition  of  Handwritten
Digits/10OpticalRecognitionofHandwrittenDigits.csv',

      './DT_Dataset/11_Spambase/11Spambase.csv',

      './DT_Dataset/12_Zoo/12Zoo.csv',

                         './DT_Dataset/13_Statlog    (German    Credit
Data)/13StatlogGermanCreditData.csv',

      './DT_Dataset/14_Bank Marketing/14BankMarketing.csv',
            './DT_Dataset/16_Glioma  Grading  Clinical  and  Mutation
Features/16GliomaGradingClinicalandMutationFeatures.csv'
]
time_elapsed = {}
memory_usage = {}
SKlearn_SVM_Accuracy_Scores = []
SKlearn_DT_Accuracy_Scores = []
TF_DL_Accuracy_Scores = []
total_memory_used = []
total_time_taken = []

for dir in dirs:

print("---------------------------------------------------------------
---------------------------------")
    print("Current Directory : ",dir)
    df = pd.read_csv(dir)
    df.describe()

    is_categorical = True
                            if       len(list(set(df.columns)       -
set(df._get_numeric_data().columns)))==0:
        is_categorical = False

    if is_categorical:
        df = df.replace('?', np.NaN)
        df.fillna(df.mode().iloc[0], inplace=True)

    print("Is Categorical flag: ",is_categorical)
    cdf=df
    preprocess_dataset(cdf,is_categorical)
    lst = df.values.tolist()

    tick = time.time()#start time

    tracemalloc.start()

    trainDF_x, testDF_x = model_selection.train_test_split(lst)

    trainDF_y =[]

    for l in trainDF_x:

        trainDF_y.append(l[-1])

        del l[-1]


    testDF_y = []

    for lst in testDF_x:

        testDF_y.append(lst[-1])

        del lst[-1]


    accuracy_scores = {}


    if is_categorical==False:

        tracemalloc.start()

        tick = time.time()

        clf4 = tree.DecisionTreeClassifier()

        clf4.fit(np.array(trainDF_x), np.array(trainDF_y))

        inbuiltmodel_pred = clf4.predict(np.array(testDF_x))

        tock = time.time()

                 memory_usage['Inbuilt  DT  Model  Memory  Usage']  =
tracemalloc.get_traced_memory()[0]


           total_memory_used.append( {"memory_usage['Inbuilt Decision
Tree Model Memory Usage']" : tracemalloc.get_traced_memory()[0]})


        time_elapsed['Inbuilt DT Model time'] = round((tock - tick) *
1000, 2)
```

```
        total_time_taken.append({"time_elapsed['Inbuilt Decision Tree
Model time']" : round((tock - tick) * 1000, 2)})


        tracemalloc.stop()
           accuracy_scores['Sklearn Decision Tree Model Accuracy'] =
accuracy_score(np.array(testDF_y), inbuiltmodel_pred)



SKlearn_DT_Accuracy_Scores.append(format(accuracy_score(np.array(test
DF_y), inbuiltmodel_pred)))


              print('Accuracy Score of Sklearn Decision tree Model
Numerical:     {0:0.3f}'     .format(accuracy_score(np.array(testDF_y),
inbuiltmodel_pred)))


        # """""performing SVM on same dataset"""""
        tracemalloc.start()
        tick = time.time()
        svc = SVC()
        svc.fit(np.array(trainDF_x), np.array(trainDF_y).flatten())
        svm_pred = svc.predict(np.array(testDF_x))
        tock = time.time()
                      memory_usage['SVM   Model   Memory   Usage']   =
tracemalloc.get_traced_memory()[0]
          total_memory_used.append({"memory_usage['SVM Model Memory
Usage']" : tracemalloc.get_traced_memory()[0]})
        tracemalloc.stop()


         time_elapsed['SVM Model time'] = round((tock - tick) * 1000,
2)
          total_time_taken.append({"time_elapsed['SVM Model time']" :
round((tock - tick) * 1000, 2)})
```

```
                        accuracy_scores['SVM   Model   Accuracy']   =
accuracy_score(testDF_y, svm_pred)


SKlearn_SVM_Accuracy_Scores.append(format(accuracy_score(testDF_y,
svm_pred)))


            print('Accuracy  Score  of  Sklearn  SVM  Model  Numerical:
{0:0.3f}'.format(accuracy_score(testDF_y, svm_pred)))

        ten_df = df
        tick = time.time()
        tracemalloc.start()
        X = ten_df.iloc[:, :-1].values
        Y = ten_df.iloc[:, -1].values.reshape(-1, 1)
                          X_train,   X_test,   Y_train,   Y_test   =
model_selection.train_test_split(X, Y, test_size=.3, random_state=41)
        print(len(X_train))
        print(len(Y_train))
        model = tf.keras.Sequential([
                    tf.keras.layers.Dense(128, activation='relu'),
                    tf.keras.layers.Dense(64, activation='relu'),
                    tf.keras.layers.Dense(1, activation='sigmoid')
                ])
        model.compile(optimizer='rmsprop',
                        loss='binary_crossentropy',
                        metrics=['accuracy'])
        model.fit(X_train, Y_train, batch_size=32, epochs=20)
        loss, accuracy = model.evaluate(X_test, Y_test, verbose=0)
        print('Test loss:', loss)
        print('Test accuracy:', accuracy)
        tensor_pred = model.predict(X_test)
        tensor_pred = np.argmax(tensor_pred, axis=1)

        y_test = np.argmax(Y_test, axis=1)

        cm = metrics.confusion_matrix(y_test, tensor_pred)


        TF_DL_Accuracy_Scores.append(accuracy)

        print(cm)

        print('Accuracy  Score  of  Tensorflow  Classification::
{0:0.3f}'.format( accuracy))

        print(metrics.confusion_matrix(y_test, tensor_pred))

        print(metrics.classification_report(y_test, tensor_pred))

        tock = time.time()

        time_elapsed['TensorFlow Model time'] = round((tock - tick) *
1000, 2)

        memory_usage['Tensorflow  model  memory  usage']  =
tracemalloc.get_traced_memory()[0]

        total_memory_used.append({"memory_usage['Tensorflow  model
memory usage']" : tracemalloc.get_traced_memory()[0]})

        tracemalloc.stop()

        total_time_taken.append({"time_elapsed['TensorFlow  Model
time']" : round((tock - tick) * 1000, 2)})

        accuracy_scores['Tensorflow Model Accuracy'] = accuracy
```