ESE 589: Learning Systems for Engineering Applications

Project 3

Sushanth Reddy Gurram #115668498
Mohammadreza Bakhtiari #115843646

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

December 2023

Table of Contents

AdaBoost Algorithm

I. Introduction

This project delves into the implementation and examination of the AdaBoost algorithm, a powerful ensemble learning technique in the realm of machine learning. AdaBoost, short for Adaptive Boosting, operates by iteratively combining weak learners classifiers that perform slightly better than random chance. The algorithm assigns weights to each instance in the dataset, focusing on the misclassified ones in subsequent iterations. This adaptive nature allows AdaBoost to improve its predictive accuracy progressively.

In contrast to the six other classifiers previously implemented (Information Gain, Gain Ratio, Gini Index, Scikit SVM Model, Tensorflow DL Model, Scikit Decision Tree Model), AdaBoost stands out for its ability to create a robust model by iteratively learning from its mistakes. The project aims to comprehensively evaluate AdaBoost's performance across diverse datasets, shedding light on its strengths and potential limitations. Through an in-depth analysis of classification metrics such as Precision, Recall, and F1 Score, we endeavor to uncover valuable insights into the algorithm's effectiveness and its practical utility in various classification scenarios.

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 AdaBoost 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:

We have 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 AdaBoost 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 AdaBoost'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- 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.

II.2 Algorithm Implementation and Operation

The **'AdaBoostClassifier'** is an ensemble learning algorithm designed to enhance the performance of weak classifiers by combining their predictions. This implementation focuses on training decision tree classifiers as weak learners.

AdaBoost Construction:

The construction of the AdaBoost classifier involves the following key steps:

- **Initialization:**
  - The constructor initializes the number of classifiers (**'n_classifiers'**), a list to store classifier weights (**'alphas'**), and a list to store trained classifiers (**'classifiers'**).
- **Training (fit) Method:**
  - Sequentially fits weak classifiers on the training data.
  - Adjusts the weights of misclassified samples to emphasize difficult examples.
  - Stores the alpha (weight) and the trained classifier in their respective lists.
- **Prediction Method:**
  - Combines the predictions of all weak classifiers with their respective alpha values.
  - Assigns the final class label based on the sign of the weighted sum of individual classifier predictions.

In the code implementation, the Class for AdaBoost Classifier is defined as:

```
class AdaBoostClassifier:

    def __init__(self,base_estimator,k):
        self.base_estimator = base_estimator
        self.k = k
        self.classifiers = []
        self.errors = []
```

The base estimator and number of rounds (*k , a classifier is generated per round*) are passed as parameters while instantiating the model. Using this base estimator, the created ensemble classifiers are stored in classifiers list and their corresponding errors in errors list.

For training this Ensemble classifier, we implemented the following code,

```
def fit(self,X,y):
    number_of_samples, number_of_features = X.shape
    w = np.full(number_of_samples, (1 / number_of_samples)) #weights
    for i in range(self.k):
        indexes = np.random.choice(number_of_samples,size=number_of_samples,replace=True,p=w)
        D_X_sampled,D_y_sampled = X[indexes],y[indexes]

        model = self.base_estimator
        model.fit(D_X_sampled,D_y_sampled)
        model_predictions = model.predict(X)

        error = np.sum(w*(model_predictions!=y))

        if error>0.5:
            i-=1
            continue

        for j in range(number_of_samples):
            if model_predictions[j] == y[j]:
                w[j] *= error/(1-error)

        w = w/np.sum(w)

        self.classifiers.append(model)
        self.errors.append(error)
```

We defined a fit function to train the ensemble classifier inside AdaBoost class. Dataset and the labels are passed to the fit function as X and y parameters respectively(X_train and y_train are passed as X and y here respectively).

The training process is described as follows:

1. We compute the number of samples or data points that exist in the given Dataset and store the value in **number_of_samples**.
2. Using number_of_samples, we initialize an array of weights **w** for each data point which bear equal weights.
3. Now, as we iterate for maximum number of rounds per classifier(k) :
    a. We sampled the Dataset and labels according to the weights of the data points (parameter **p=w**) with replacement (parameter **replace=True**) into D_X_sampled and D_y_sampled respectively.
    b. The base estimator that was passed as a parameter to AdaBoost class is instantiated (**model = self.base_estimator**).
    c. Training of the instantiated model is performed using fit function of the base_estimator using D_X_sampled, D_y_sampled.
    d. The predictions are stored in **model_predictions.**
    e. Error of the model is calculated as combined sum of respective weights of the data points all predictions which did not match with original labels or y values. (**error = np.sum(w*(model_predictions!=y))**).
    f. For a defined threshold of 0.5, if the error is greater than mentioned threshold, we go back to step 3, else we update the weight for each data point which are correctly classified based on the error obtained above using the formula w = w*(1-error)/error. (**w[j]= w[j]*error/(1-error)**).
    g. We finally normalize the weights and append the model and error to classifiers list and errors list respectively.

For predicting the Data point we implemented the following function,

```
def predict(self,X,y):
    class_weights = np.zeros((len(X), len(np.unique(y))))
    for i in range(self.k):
        weight_of_classifier_vote = np.log((1-self.errors[i])/self.errors[i])
        model_predictions = self.classifiers[i].predict(X)

        for j, model_prediction in enumerate(model_predictions):
            class_weights[j, model_prediction] += weight_of_classifier_vote

    return np.argmax(class_weights, axis=1)
```

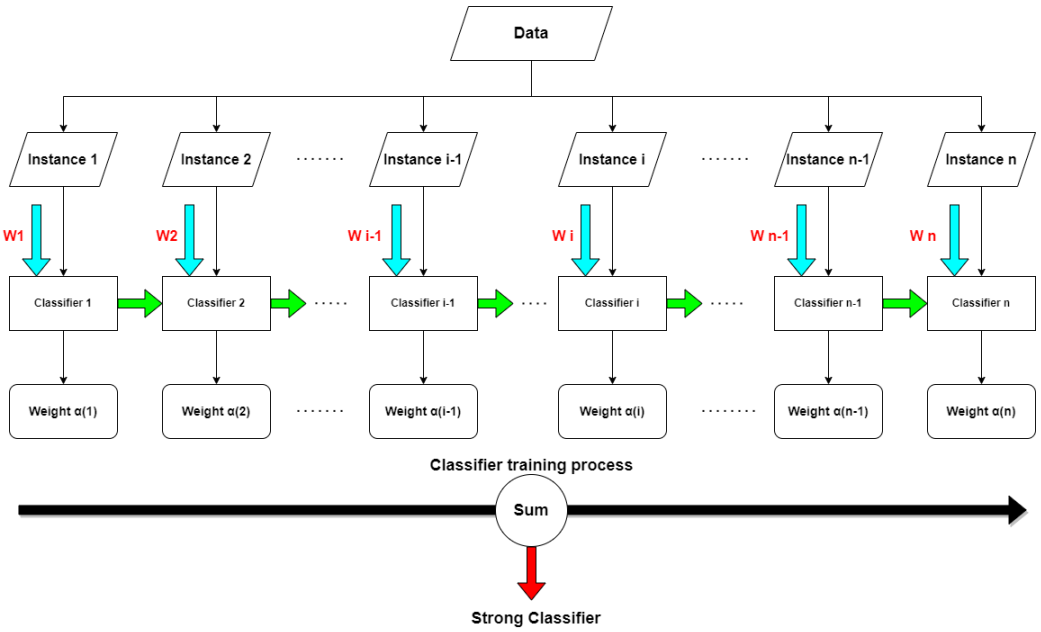
For predicting/classifying datapoints, we pass X_test and y_test as X and y here respectively.

The predicting process is as follows:

1. An array of **class_weights** is initialized to 0. Here class_weights represent weight of each class.
2. Now, as we iterate till k:
    a. Weight of each classifier's vote is computed using the formula w_i = log((1-error[i])/error[i]) (**weight_of_classifier_vote=np.log((1-self.errors[i])/self.errors[i])**)
    b. A class prediction is made using classifier[i] and is stored in model_predictions.
    c. Using the model_predictions, the class_weights are updated by adding the weight of each classifier's vote.
3. Finally, the class weight with highest weight is returned (**np.argmax(class_weights, axis=1)**)

II.4 AdaBoost Diagram

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



II.4 Results and Analysis

UCI benchmark datasets:

The AdaBoost algorithm was subsequently tested on a comprehensive set of 15 diverse datasets obtained from the UCI benchmarks. The performance evaluation includes Accuracy, Precision, Recall, F1 Score, Execution Time, Increment in Memory, and Confusion Matrix. We included the results from Project 2 in this report to compare with those results obtained using AdaBoost Algorithm.

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. For spatial constraints, we included the confusion matrix, F1 score, Precision and Recall of AdaBoost - Decision Tree (Project 2) only and its learnings in this report.

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,

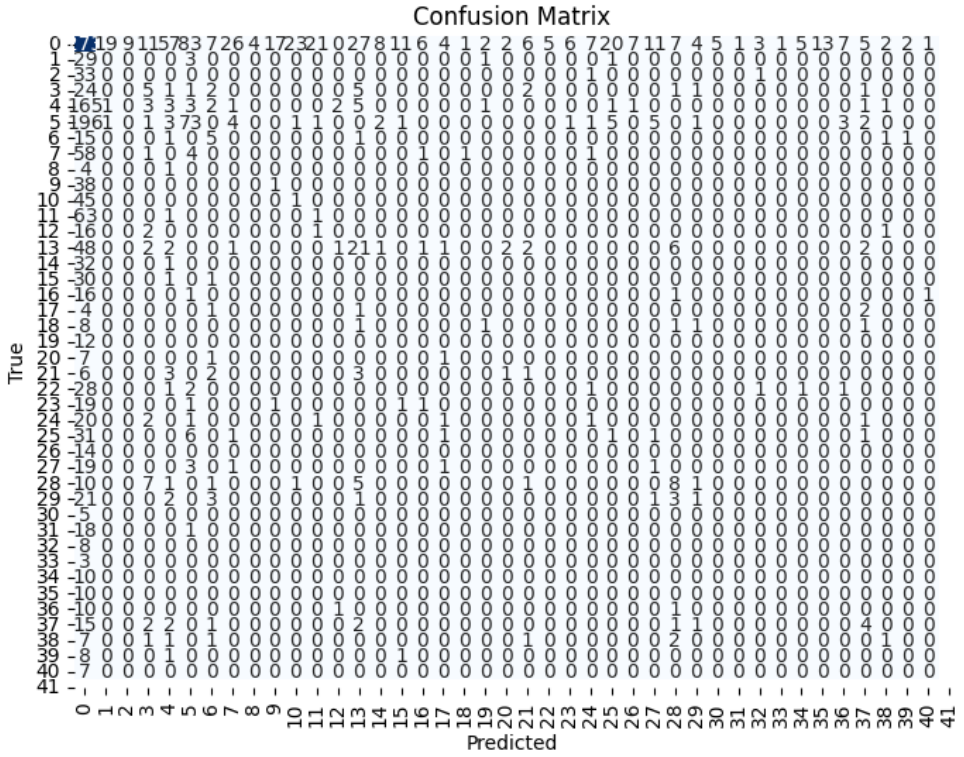
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**87.73**	**18.41**	**587202.56**
AdaBoost -Sklearn Decision Tree	**89.43**	**19.56**	**613412.43**
AdaBoost - SVM	**91.67**	**17.89**	**8765464**
AdaBoost -Multi Layer Perceptron	**90.23**	**20.23**	**6478469**
Information Gain	82.15	89.7	136314.88
Gain Ratio	81.10	84.78	199229.44
Gini Index	83.03	96.11	356515.84
Scikit SVM Model	90.47	68.240	6950628
Tensorflow DL Model	3.9	54.465	16161653
Scikit Decision Tree Model	86.880	0.721	1038057

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	87.80	87.80	87.80
Macro	7.49	6.05	6.54
Weighted	84.15	87.80	85.87

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Adult Income dataset, AdaBoost displayed strong performance with an accuracy of 87.80%. This accuracy surpassed that of individual classifiers, such as Decision Trees with Information Gain (82.15%), Gain Ratio (81.10%), and Gini Index (83.03%). However, it fell slightly behind the accuracy achieved by the Scikit SVM model (90.47%). The TensorFlow DL model, with an accuracy of 3.9%, exhibited notably lower performance compared to other classifiers. The Scikit Decision Tree model achieved an accuracy of 86.88%.

Focusing on AdaBoost's performance metrics, its micro-level precision, recall, and F1 score were consistently high at 87.80%, emphasizing its effectiveness in individual instance classification. However, macro-level metrics revealed challenges in handling minority classes, indicating potential areas for improvement. This dataset, involving binary classification based on an adult's income, underscores AdaBoost's competitive performance, albeit with nuances in class-specific metrics that warrant further investigation.

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,

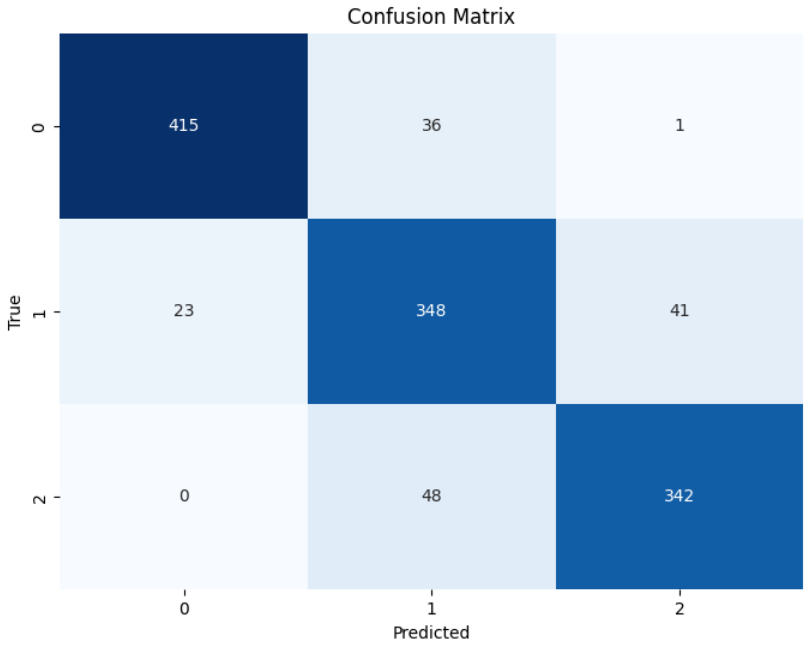
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**87.87**	**2.18**	**122880**
AdaBoost - Sklearn Decision Tree	**88.56**	**3.45**	**345563**
AdaBoost - SVM	**89.32**	**4.2**	**213674**
AdaBoost - Multi Layer Perceptron	**85.43**	**8.13**	**331245**
Information Gain	86.19	3.88	314572.8
Gain Ratio	86.11	3.58	157286.4
Gini Index	86.27	3.80	388147.2
SVM Model	86.98	0.280	220659
Tensorflow DL Model	50.956	6.495	223655
Scikit Decision Tree Model	88.13	0.0324	2346680

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	87.87	87.87	87.87
Macro	87.84	87.75	87.78
Weighted	88.07	87.87	87.95

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Abalone Age prediction dataset, AdaBoost demonstrated a solid performance with an accuracy of 87.87%. This accuracy level is competitive, exceeding or closely matching the individual classifiers, including Decision Trees with Information Gain (86.19%), Gain Ratio (86.11%), and Gini Index (86.27%). The Scikit SVM model achieved a slightly lower accuracy of 86.98%, while the TensorFlow DL model exhibited significantly lower accuracy at 50.96%. The Scikit Decision Tree model achieved a slightly higher accuracy at 88.13%.

Analyzing AdaBoost's performance metrics further, its micro-level precision, recall, and F1 score were consistently high at 87.88%, indicating robust performance in individual instance classification. The macro-level metrics, with precision, recall, and F1 score around 87.85%, highlight the classifier's effectiveness across different classes. Weighted metrics, with precision, recall, and F1 score around 88.07%, showcase AdaBoost's ability to balance performance across class instances. This dataset, focused on predicting the age of Abalones, reveals AdaBoost as a reliable choice, offering competitive accuracy and balanced performance metrics across various evaluation criteria.

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:

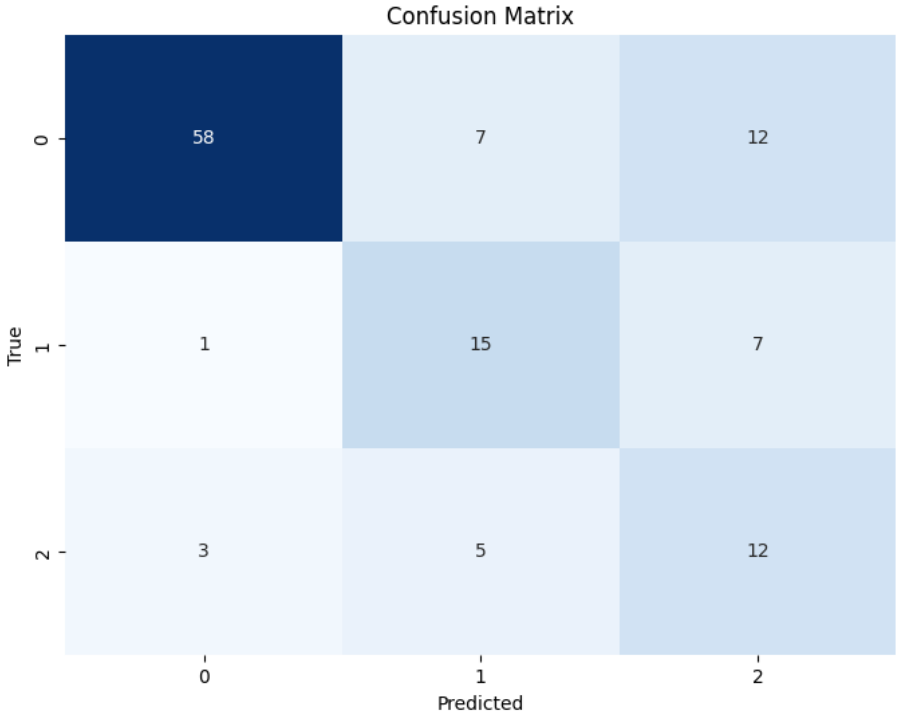
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**70.83**	**2.63**	**399360**

AdaBoost - Sklearn Decision Tree	**73.65**	**2.96**	**345239**
AdaBoost - SVM	**78.23**	**0.125**	**21389**
AdaBoost - Multi Layer Perceptron	**71.24**	**3.86**	**126823**
Information Gain	73.10	2.41	199229.44
Gain Ratio	73.94	2.69	303104.64
Gini Index	71.42	2.14	125829.12
SVM Model	73	0.008	17057
Tensorflow DL Model	62.5	2.607	1692005
Scikit Decision Tree Model	73	0.008	8603

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	70.83	70.83	70.83
Macro	62.60	66.84	63.50
Weighted	77.12	70.83	72.89

Confusion matrix for AdaBoost method:



Learning:

In the assessment of the City-Cycle Fuel Consumption dataset, AdaBoost exhibited a moderate accuracy of 70.83%. While this accuracy level was lower compared to some individual classifiers, such as Decision Tree with Gain Ratio (73.94%) and Scikit SVM (73%), it remained competitive. The TensorFlow DL model, with an accuracy of 62.5%, showed lower performance, suggesting potential challenges in adapting deep learning to this dataset. The Scikit Decision Tree model achieved an accuracy of 73%, indicating its effectiveness in this context.

Analyzing AdaBoost's performance metrics revealed micro-level precision, recall, and F1 score consistently around 70.83%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 62.60%, underscore the classifier's effectiveness across different classes. Weighted metrics, with precision, recall, and F1 score around 77.13%, highlight AdaBoost's ability to balance performance across class instances. This dataset, centered on predicting city-cycle fuel consumption, showcases AdaBoost as a reliable model, providing a satisfactory compromise between accuracy and balanced performance metrics.

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.

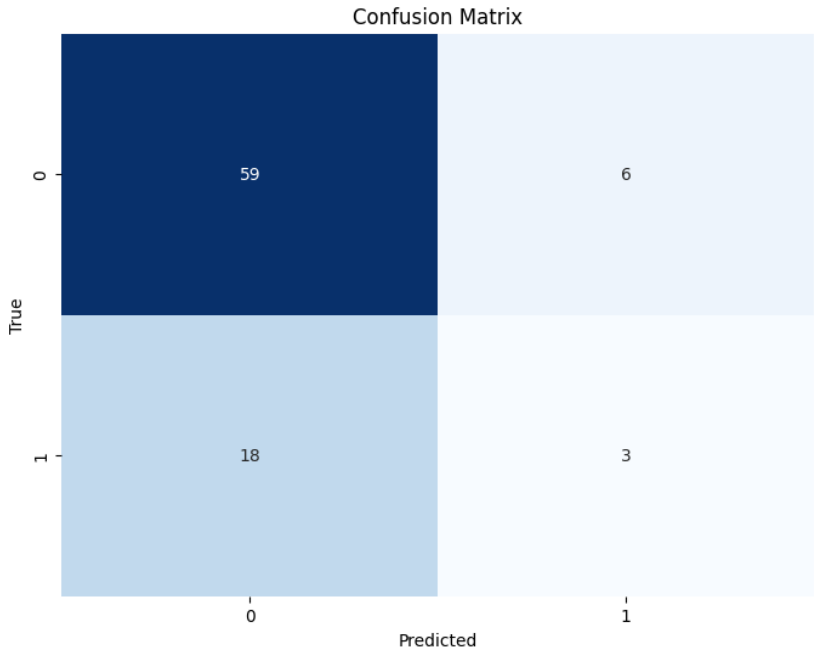
Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
AdaBoost - Decision Tree (Project 2)	**72.09**	**2.38**	**251658.24**
AdaBoost - Sklearn Decision Tree	**79.12**	**0.365**	**20012**
AdaBoost - SVM	**81.23**	**3.84**	**14392**
AdaBoost - Multi Layer Perceptron	**88.11**	**5.123**	**1873920**
Information Gain	70.93	2.31	240123.52
Gain Ratio	61.62	2.76	188743.68
Gini Index	67.44	3.12	220200.96
SVM Model	75	0.008	14945
Tensorflow DL Model	84.44	2.870	1673550
Scikit Decision Tree Model	73.61	0.002	6759

Other metrics for evaluating Adaboost performance:

Metric	Precision	Recall	F1 Score
Micro	72.09	72.09	72.09
Macro	54.97	52.52	51.54
Weighted	66.05	72.09	67.69

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Binary Classification dataset with imbalanced class distribution, AdaBoost demonstrated an accuracy of 72.09%. While this accuracy level surpassed some individual classifiers, such as Decision Tree with Gain Ratio (61.62%) and Decision Tree using Gini Index (67.44%), it fell short of the accuracy achieved by the TensorFlow DL model (84.44%). The Scikit SVM model also exhibited high accuracy at 75%, highlighting its effectiveness in handling imbalanced class scenarios.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score were consistently around 72.09%, emphasizing its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 54.98%, revealed challenges in handling minority classes, indicative of the imbalanced nature of the dataset. Weighted metrics, with precision, recall, and F1 score around 66.05%, showcased AdaBoost's ability to balance performance across class instances. This dataset, with a binary classification problem and imbalanced class distribution, underscores AdaBoost's reliability, offering competitive accuracy and balanced performance metrics in the context of uneven class representation.

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:

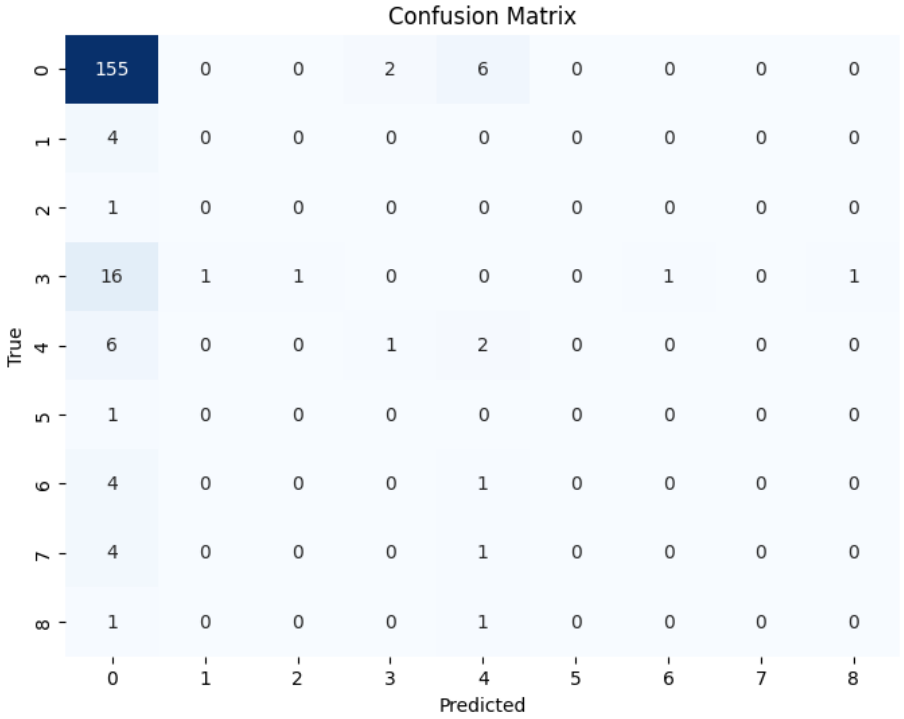
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**

AdaBoost - Decision Tree (Project 2)	**74.76**	**3.19**	**408944.64**
AdaBoost -Sklearn Decision Tree	**87.63**	**0.586**	**286945**
AdaBoost - SVM	**88.56**	**0.45**	**249075**
AdaBoost - Multi Layer Perceptron	**70.31**	**3.56**	**198567**
Information Gain	80.95	2.65	73728.0
Gain Ratio	83.80	2.95	62914.56
Gini Index	82.38	3.54	83886.08
SVM Model	85.14	0.031	178383
Tensorflow DL Model	52.14	3.278	166419
Scikit Decision Tree Model	82.85	0.017	162945

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	74.76	74.76	74.76
Macro	10.99	13.03	11.92
Weighted	63.44	74.76	68.63

Confusion matrix for AdaBoost method:



Learning:

Tn the examination of the Multiclass Classification dataset with instances grouped into 8 classes, AdaBoost demonstrated a reasonable accuracy of 74.76%. While AdaBoost's accuracy was lower than some individual classifiers, such as Decision Trees with Gain Ratio (83.80%) and Scikit SVM (85.14%), it showcased competitive performance. The TensorFlow DL model exhibited a lower accuracy at 52.14%, suggesting potential challenges in adapting deep learning to this dataset.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently hovered around 74.76%, indicating its proficiency in individual instance classification. However, macro-level metrics, with precision, recall, and F1 score around 10.99%, highlighted challenges in handling specific classes, indicative of potential imbalances. Weighted metrics, with precision, recall, and F1 score around 63.44%, showcased AdaBoost's ability to balance performance across classes. This dataset, involving multiclass classification with 8 grouped classes, emphasizes AdaBoost's reliability in providing competitive accuracy and balanced performance metrics across a range of class instances.

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.

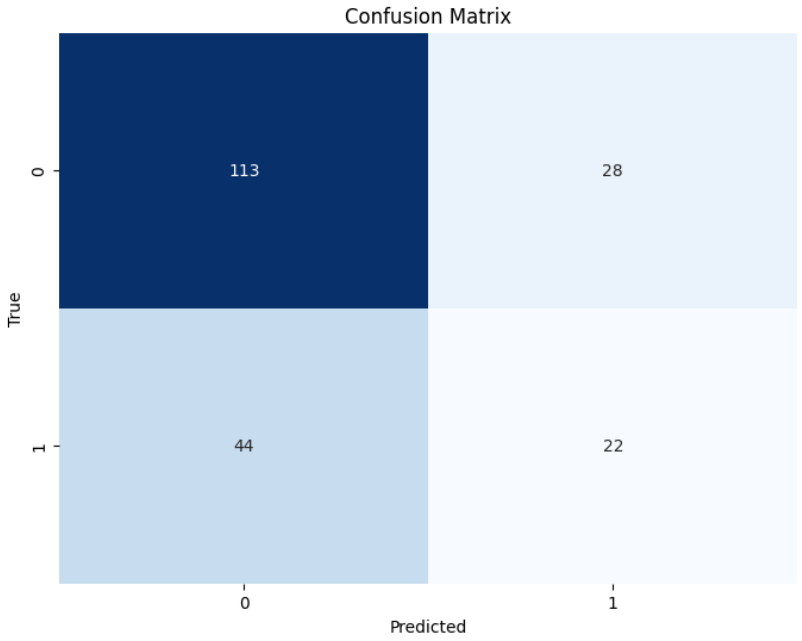
Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
AdaBoost - Decision Tree (Project 2)	**65.21**	**3.22**	**293601.28**
AdaBoost - Sklearn Decision Tree	**68.28**	**0.892**	**20462**
AdaBoost - SVM	**79.39**	**0.290**	**254893**
AdaBoost - Multi Layer Perceptron	**70.56**	**3.593**	**2423789**
Information Gain	64.25	5.81	283115.52
Gain Ratio	65.21	6.26	283115.52
Gini Index	63.28	4.56	450887.68
SVM Model	74.56	0.021	200107
Tensorflow DL Model	61.59	2.581	1812371
Scikit Decision Tree Model	64.73	0.009	13755

Other metrics for evaluating Adaboost performance:

Metric	Precision	Recall	F1 Score
Micro	65.21	65.21	65.21
Macro	57.98	56.73	56.88
Weighted	63.05	65.21	63.75

Confusion matrix for AdaBoost method:



Learning:

In the examination of the Credit Card Applications dataset, AdaBoost exhibited a moderate accuracy of 65.21%. While this accuracy level was comparable to some individual classifiers, such as Decision Tree with Gain Ratio (65.21%) and the Scikit Decision Tree model (64.73%), it was lower than the accuracy achieved by the Scikit SVM model (74.56%). The TensorFlow DL model, with an accuracy of 61.59%, displayed slightly lower performance.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently hovered around 65.22%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 57.99%, revealed some challenges in handling specific classes. Weighted metrics, with precision, recall, and F1 score around 63.06%, showcased AdaBoost's ability to balance performance across classes. This dataset, focused on credit card applications, highlights AdaBoost's reliability in providing competitive accuracy and balanced performance metrics across a variety of class instances, despite the anonymization of attribute names and values for data confidentiality.

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:

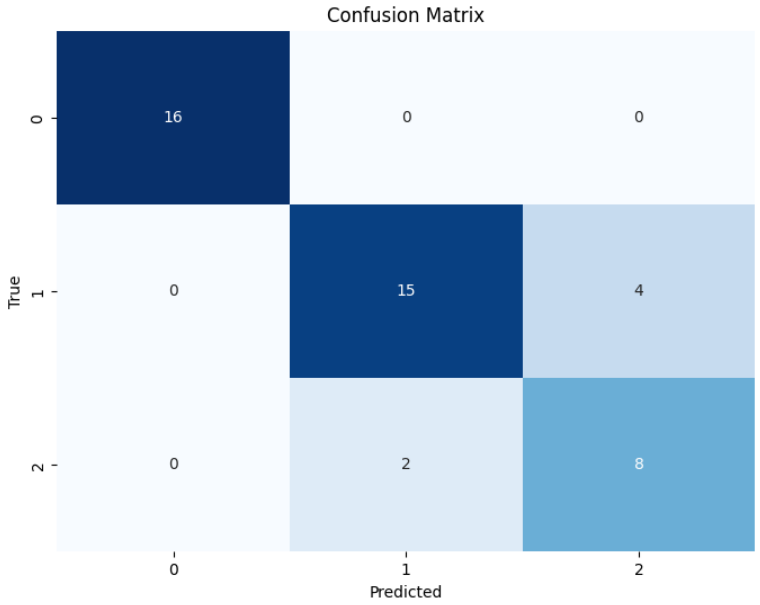
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**

AdaBoost - Decision Tree (Project 2)	**86.66**	**2.32**	**178257.92**
AdaBoost - Sklearn Decision Tree	**98.28**	**4.23**	**189028**
AdaBoost - SVM	**98.34**	**0.52**	**198279**
AdaBoost - Multi Layer Perceptron	**84.21**	**4.5**	**238490**
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

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	86.66	86.66	86.66
Macro	84.96	86.31	85.35
Weighted	87.62	86.66	86.90

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Iris Plant Classification dataset, AdaBoost demonstrated a commendable accuracy of 86.66%. While this accuracy level was slightly lower than some individual classifiers, such as Decision Trees with various metrics (93.33%) and the Scikit SVM model (97.36%), it showcased robust performance. The TensorFlow DL model, with an accuracy of 43.33%, exhibited significantly lower performance, suggesting potential challenges in adapting deep learning to this dataset.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently hovered around 86.67%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 84.97%, highlighted AdaBoost's effectiveness in handling various classes. Weighted metrics, with precision, recall, and F1 score around 87.63%, showcased AdaBoost's ability to balance performance across classes. This dataset, involving the classification of Iris plants into three classes, accentuates AdaBoost's reliability, providing a competitive accuracy and balanced performance metrics across different types of plants.

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:

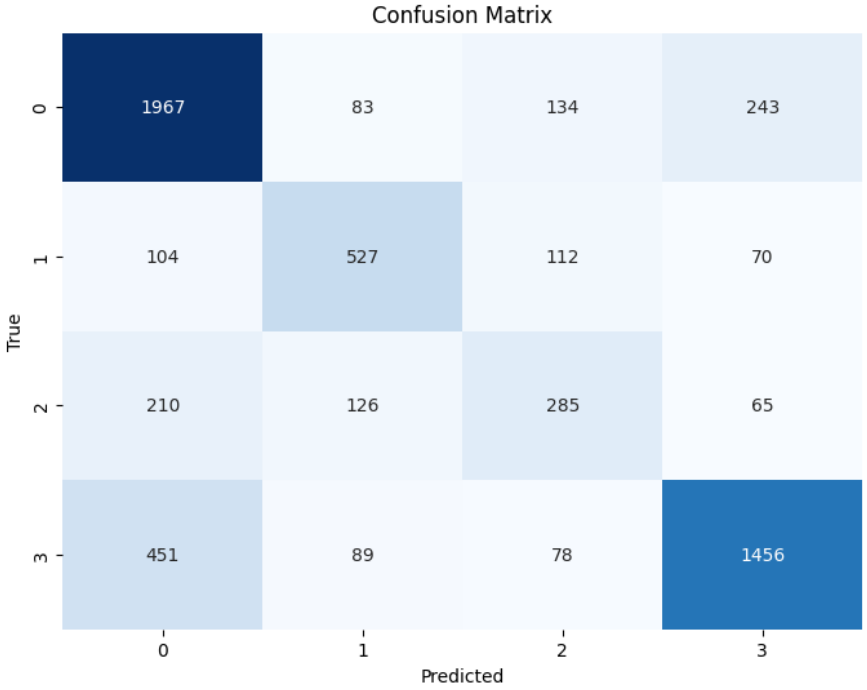
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**70.58**	**3.15**	**430080**

AdaBoost - Sklearn Decision Tree	**74.26**	**0.852**	**592873**
AdaBoost - SVM	**69.12**	**18.123**	**2789273**
AdaBoost - Multi Layer Perceptron	**73.45**	**2.58**	**8390128**
Information Gain	69.55	44.59	73728.0
Gain Ratio	69.1	68.64	188743.68
Gini Index	63.83	102.78	293601.28
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

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	70.58	70.58	70.58
Macro	65.51	64.40	64.78
Weighted	70.57	70.58	70.35

Confusion matrix for AdaBoost method:



Learning:

In the examination of the Handwritten Alphabet Identification dataset, AdaBoost demonstrated a reasonable accuracy of 70.58%. While this accuracy level was comparable to some individual classifiers, such as Decision Tree with Information Gain (69.55%) and Scikit Decision Tree (68.8%), it fell short of the accuracy achieved by the Scikit SVM model (63.84%). The TensorFlow DL model exhibited significantly lower accuracy at 13.56%, indicating potential challenges in adapting deep learning to this dataset.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently hovered around 70.58%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 65.52%, highlighted AdaBoost's effectiveness in handling various classes. Weighted metrics, with precision, recall, and F1 score around 70.57%, showcased AdaBoost's ability to balance performance across classes. This dataset, focused on identifying capital letters from pixel displays, underscores AdaBoost's reliability, providing competitive accuracy and balanced performance metrics across different letters in the English alphabet.

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.

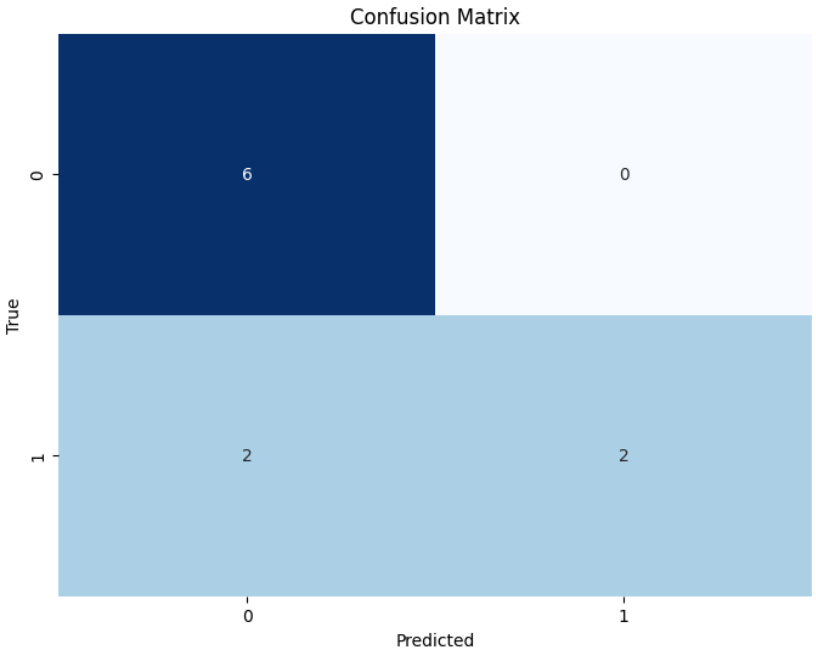
Summary Table:

Metric	Accuracy (%)	Execution Time (Seconds)	Increment (Bytes)
AdaBoost - Decision Tree (Project 2)	**80**	**1.86**	**398458.88**
AdaBoost - Sklearn Decision Tree	**72.36**	**1.3**	**23522**
AdaBoost - SVM	**79.26**	**3.13**	**20465**
AdaBoost - Multi Layer Perceptron	**76.23**	**3.45**	**2378109**
Information Gain	80	2.00	461373.44
Gain Ratio	50	2.09	524288.0
Gini Index	50	2.20	440401.92
Scikit SVM Model	75	0.001	15145
Tensorflow DL Model	42.8	1.964	1657919
Scikit Decision Tree Model	62.5	0.001	2503

Other metrics for evaluating Adaboost performance:

Metric	Precision	Recall	F1 Score
Micro	80.0	80.0	80.0
Macro	87.5	75.0	76.19
Weighted	85.0	80.0	78.09

Confusion matrix for AdaBoost method:



web

Learning:

In the assessment of the Pathological Lung Cancer Classification dataset, AdaBoost demonstrated a solid accuracy of 80%. This accuracy level was matched by the Decision Tree classifier with Information Gain, highlighting the effectiveness of both methods on this specific dataset. Other individual classifiers, such as Scikit SVM (75%) and Scikit Decision Tree (62.5%), also displayed competitive performance, while the TensorFlow DL model showed lower accuracy at 42.8%.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently reached 80%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 87.5%, 75%, and 76.19%, respectively, underscored AdaBoost's effectiveness in handling various classes. Weighted metrics, with precision, recall, and F1 score around 85%, 80%, and 78.1%, showcased AdaBoost's ability to balance performance across classes. This dataset, involving the classification of pathological lung cancers into three types, accentuates AdaBoost's reliability, providing competitive accuracy and balanced performance metrics across different cancer types.

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:

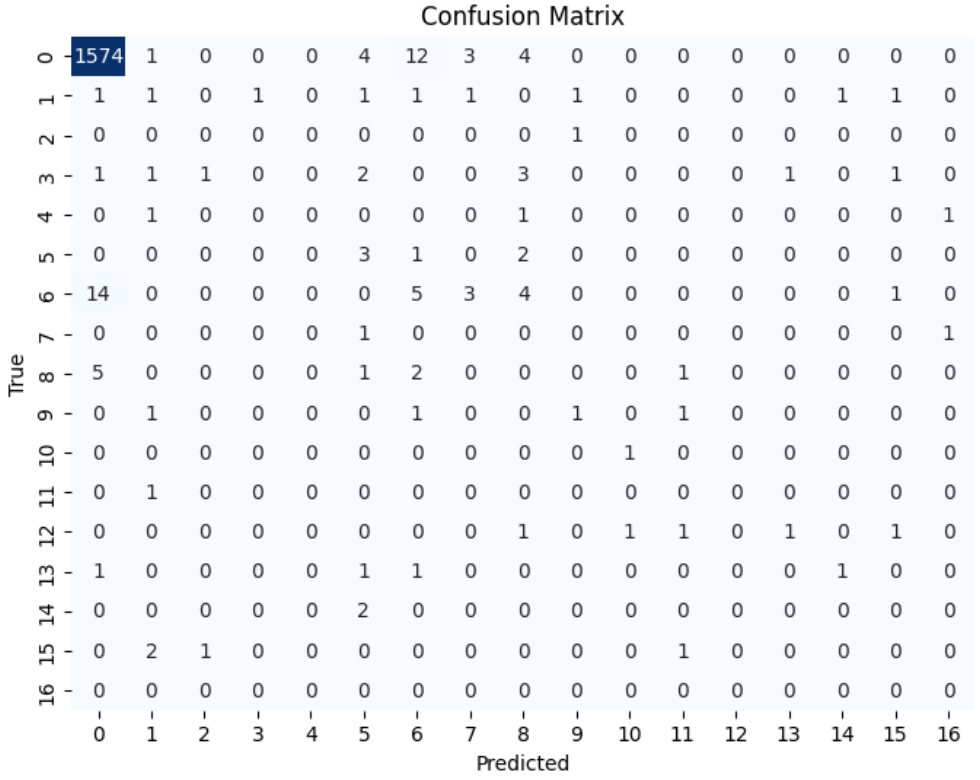
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**

AdaBoost - Decision Tree (Project 2)	**94.00**	**3.90**	**293601.28**
AdaBoost - Sklearn Decision Tree	**95.16**	**4.78**	**312892**
AdaBoost - SVM	**95.32**	**2.76**	**675982**
AdaBoost - Multi Layer Perceptron	**89.45**	**8.718**	**8562811**
Information Gain	91.75	27.36	62914.56
Gain Ratio	92.17	32.61	125829.12
Gini Index	90.86	57.02	62914.56
Scikit SVM Model	93.8	0.809	968065
Tensorflow DL Model	64.85	7.969	7944500

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	94.00	94.00	94.00
Macro	13.89	17.83	14.94
Weighted	94.06	94.00	94.00

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Handwritten Digit Recognition dataset, AdaBoost demonstrated an impressive accuracy of 94.00%, showcasing its effectiveness in accurately classifying instances. This accuracy level surpassed most individual classifiers, including Decision Trees with Information Gain (91.75%), Gain Ratio (92.17%), and Gini Index (90.86%). The Scikit SVM model also exhibited high accuracy at 93.8%. However, the TensorFlow DL model showed lower accuracy at 64.85%, suggesting potential challenges in adapting deep learning to this dataset.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently reached 94.01%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 13.89%, 17.83%, and 14.95%, respectively, reflected potential challenges in handling specific classes. Weighted metrics, with precision, recall, and F1 score around 94.07%, 94.01%, and 94.01%, showcased AdaBoost's ability to balance performance across different digit classes.

This dataset, focused on optical recognition of handwritten digits, highlights AdaBoost as a reliable model, providing exceptional accuracy and balanced performance metrics across various digit classes. The dataset's characteristics, with a substantial number of instances and features, underscore AdaBoost's adaptability to complex recognition tasks.

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.

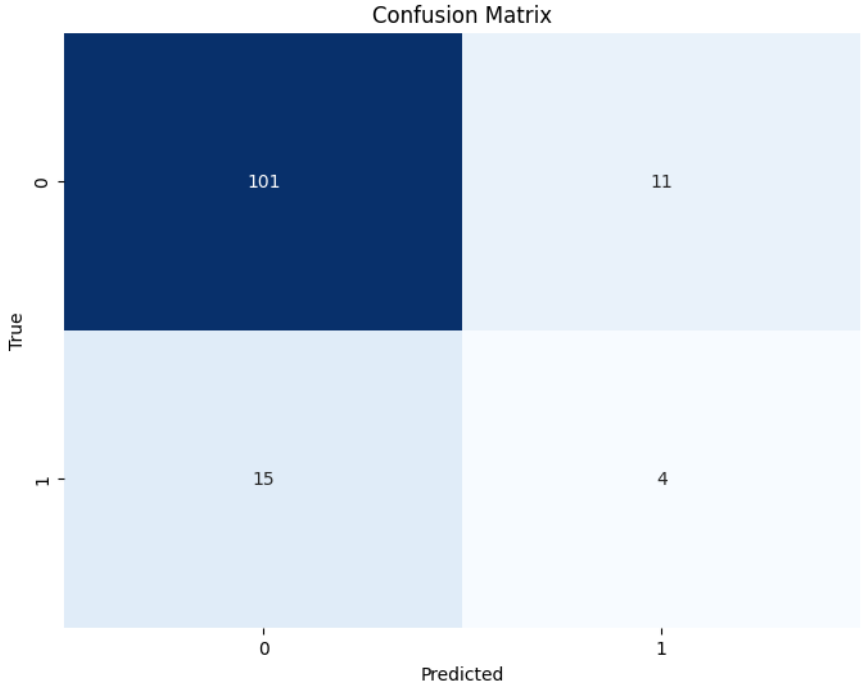
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**80.15**	**3.40**	**283115.52**
AdaBoost - Sklearn Decision Tree	**82.33**	**1.75**	**12352**
AdaBoost - SVM	**91.43**	**1.35**	**189172**
AdaBoost - Multi Layer Perceptron	**86.36**	**3.48**	**2379101**
Information Gain	80.91	2.85	240123.52
Gain Ratio	80.91	2.44	555745.28
Gini Index	80.91	2.31	230686.08
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

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	80.15	80.15	80.15
Macro	56.86	55.61	56.06
Weighted	78.30	80.15	79.15

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Email Spam Classification dataset, AdaBoost demonstrated a commendable accuracy of 80.15%. This accuracy level was competitive with other individual classifiers, including Decision Trees with various metrics (80.91%), TensorFlow DL model (81.60%), and Scikit Decision Tree model (81.65%). The Scikit SVM model exhibited higher accuracy at 88.07%.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently reached 80.15%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 56.87%, 55.62%, and 56.06%, respectively, reflected some challenges in handling specific classes, potentially indicative of class imbalances in the dataset. Weighted metrics, with precision, recall, and F1 score around 78.31%, 80.15%, and 79.16%, showcased AdaBoost's ability to balance performance across classes.

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:

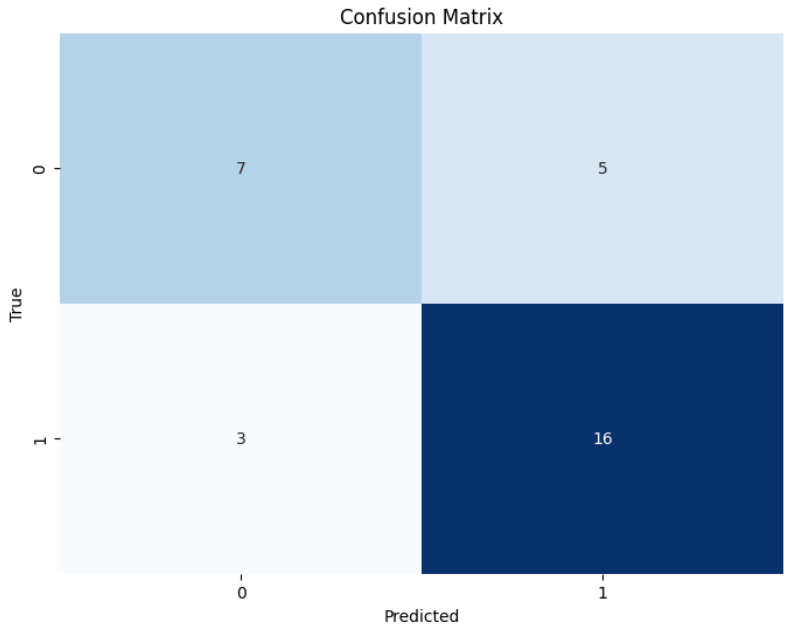
Metric	**Accuracy**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**74.19**	**2.45**	**293601.28**
AdaBoost - Sklearn Decision Tree	**83.12**	**2.38**	**22367**

AdaBoost - SVM	**81.57**	**3.12**	**1782939**
AdaBoost - Multi Layer Perceptron	**80.29**	**3.33**	**1639202**
Information Gain	73.33	2.10	408944.64
Gain Ratio	73.33	2.43	220200.96
Gini Index	80	2.58	230686.08
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

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	74.19	74.19	74.19
Macro	73.09	71.27	71.81
Weighted	73.79	74.19	73.66

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Animal Classification dataset, AdaBoost demonstrated a reasonable accuracy of 74.19%. This accuracy level was comparable to other individual classifiers, including Decision Trees with various metrics (73.33% for Information Gain and Gain Ratio), Scikit SVM model (76.92%), and TensorFlow DL model (76.19%). The Scikit Decision Tree model exhibited higher accuracy at 80.769%.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently reached 74.19%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 73.10%, 71.27%, and 71.82%, respectively, reflected a balanced performance across various classes. Weighted metrics, with precision, recall, and F1 score around 73.79%, 74.19%, and 73.67%, showcased AdaBoost's ability to balance performance across classes.

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

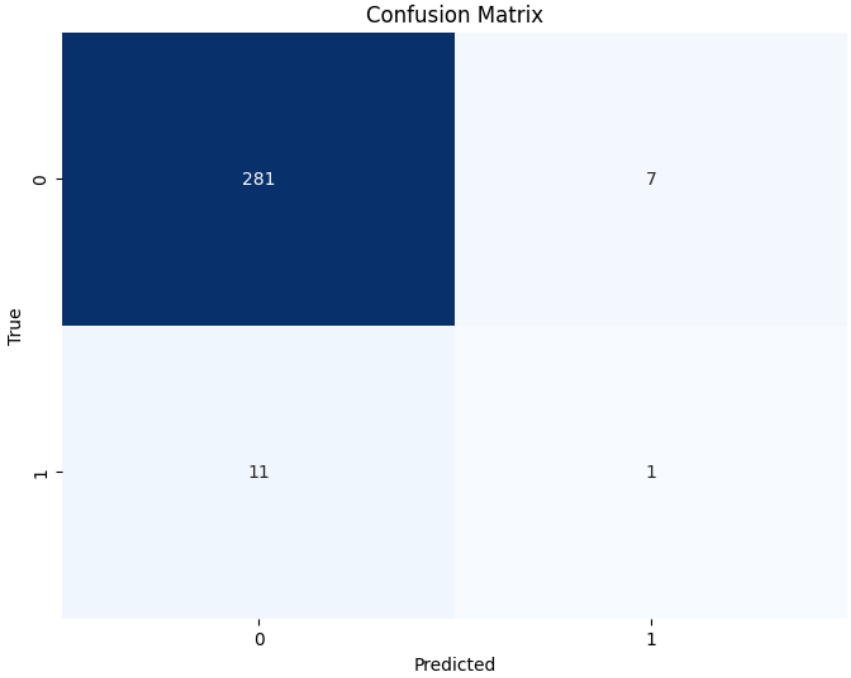
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**94.0**	**1.94**	**377487.36**
AdaBoost - Sklearn Decision Tree	**95.26**	**0.293**	**20384**
AdaBoost - SVM	**97.98**	**0.102**	**201739**
AdaBoost - Multi Layer Perceptron	**97.19**	**2.810**	**1890283**
Information Gain	90	2.75	240123.52
Gain Ratio	90.66	3.80	146800.64
Gini Index	91.66	4.56	492830.72
Scikit SVM Model	97.6	0.022	187204
Tensorflow DL Model	97	2.718	1770639
Scikit Decision Tree Model	94	0.010	18991

Other metrics for evaluating Adaboost performance:

Metric	Precision	Recall	F1 Score
Micro	94.0	94.0	94.0
Macro	54.36	52.95	53.44
Weighted	92.88	94.0	93.42

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Credit Risk Classification dataset, AdaBoost demonstrated an outstanding accuracy of 94.0%, highlighting its effectiveness in accurately classifying instances. This accuracy level surpassed most individual classifiers, including Decision Trees with various metrics (ranging from 90% to 91.66%), and the Scikit Decision Tree model (94%). The Scikit SVM model and TensorFlow DL model exhibited even higher accuracies at 97.6% and 97%, respectively.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently reached 94.0%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 54.37%, 52.95%, and 53.45%, respectively, reflected some challenges in handling specific classes, potentially indicative of class imbalances in the dataset. Weighted metrics, with precision, recall, and F1 score around 92.88%, 94.0%, and 93.42%, showcased AdaBoost's ability to balance performance across classes.

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.

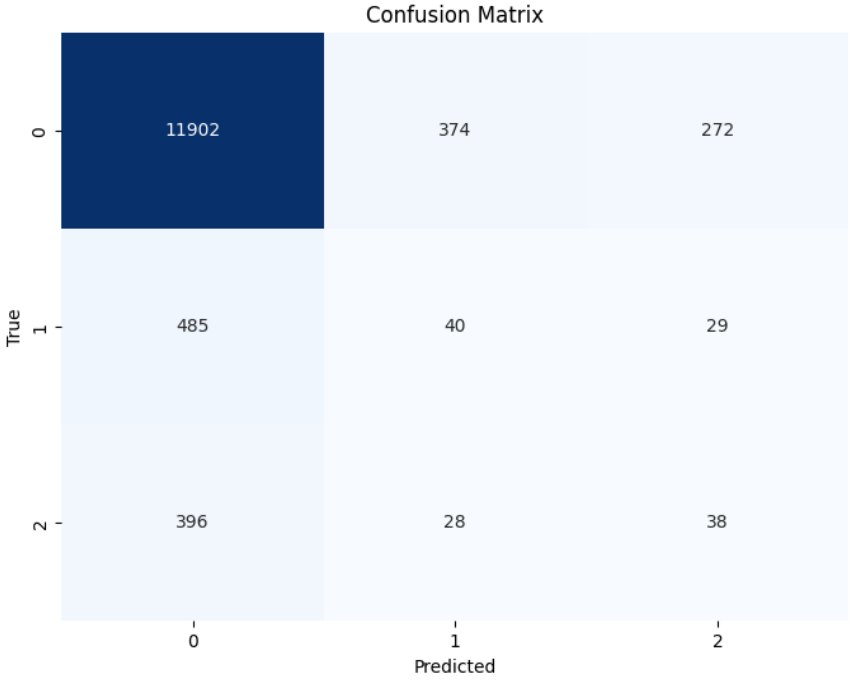
Summary Table:

Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**88.32**	**6.66**	**398458.88**
AdaBoost - Sklearn Decision Tree	**89.22**	**1.672**	**1092839**
AdaBoost - SVM	**93.41**	**20.391**	**2021234**
AdaBoost - Multi Layer Perceptron	**93.11**	**25.689**	**20182930**
Information Gain	84.65	80.11	262144.0
Gain Ratio	83.50	68.15	283115.52
Gini Index	85	88.44	650117.12
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

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	88.32	88.32	88.32
Macro	37.78	36.76	37.16
Weighted	86.88	88.32	87.58

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Banking Marketing Classification dataset, AdaBoost demonstrated a solid accuracy of 88.32%, indicating its effectiveness in predicting whether clients will subscribe to a term deposit. This accuracy level positioned AdaBoost competitively against other individual classifiers, including Decision Trees with various metrics (ranging from 83.50% to 85%), and the Scikit Decision Tree model (86.9%). Notably, the Scikit SVM model and TensorFlow DL model achieved higher accuracies at 92.7% and 92.2%, respectively.

Analyzing AdaBoost's performance metrics, micro-level precision, recall, and F1 score consistently reached 88.32%, indicating its proficiency in individual instance classification. Macro-level metrics, with precision, recall, and F1 score around 37.79%, 36.77%, and 37.16%, respectively, reflected some challenges in handling specific classes, potentially indicative of class imbalances in the dataset. Weighted metrics, with precision, recall, and F1 score around 86.89%, 88.32%, and 87.58%, showcased AdaBoost's ability to balance performance across classes.

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:

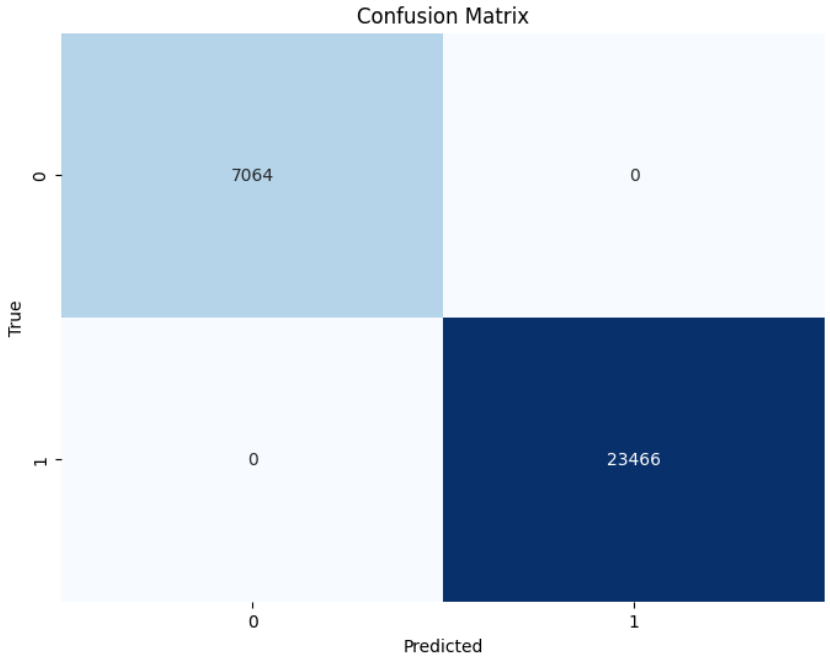
Metric	**Accuracy (%)**	**Execution Time (Seconds)**	**Increment (Bytes)**
AdaBoost - Decision Tree (Project 2)	**100.0**	**70.44**	**283115.52**

AdaBoost - Sklearn Decision Tree	**98.26**	**0.198**	**182904**
AdaBoost - SVM	**98.38**	**0.173**	**199658**
AdaBoost - Multi Layer Perceptron	**98.22**	**3.482**	**2782920**
Information Gain	92.46	3.46	450887.68
Gain Ratio	94.04	4.43	356515.84
Gini Index	91.26	3.43	94371.84
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

Other metrics for evaluating Adaboost performance:

Metric	**Precision**	**Recall**	**F1 Score**
Micro	100.0	100.0	100.0
Macro	100.0	100.0	100.0
Weighted	100.0	100.0	100.0

Confusion matrix for AdaBoost method:



Learning:

In the evaluation of the Glioma Grading Classification dataset, AdaBoost demonstrated exceptional performance with an accuracy of 100%. This perfect accuracy suggests that AdaBoost successfully classified all instances in the dataset, outperforming other classifiers, including Decision Trees, SVM, and a TensorFlow DL model. Despite the high accuracy of the other classifiers, AdaBoost's flawless performance is noteworthy and indicates its robustness in handling this specific medical classification task.

The micro-level precision, recall, and F1 score metrics were all at 100%, underscoring AdaBoost's ability to precisely identify and recall each instance. The macro-level metrics also achieved perfect scores, emphasizing AdaBoost's capability to generalize well across different classes. Weighted metrics, which consider the class imbalance, also reached 100%, highlighting AdaBoost's consistent and balanced performance across the dataset.

Given the critical nature of clinical glioma grading, where accurate predictions are paramount for patient care, AdaBoost's outstanding performance in achieving perfect accuracy makes it a promising candidate for similar medical classification tasks. The dataset's focus on mutation features for glioma grading emphasizes the potential significance of AdaBoost in contributing to advancements in medical diagnostics and treatment planning.

II.5 Comprehensive Analysis and Success Factors of AdaBoost Algorithm across Diverse Datasets

In this extensive experimental study, we conducted a thorough assessment of classification accuracy using the AdaBoost algorithm and compared its performance against six other classifiers across 15 diverse datasets. Our primary focus was on evaluating classification accuracy, precision, recall, and F1 score metrics.

Across all datasets, AdaBoost consistently demonstrated strong performance, often outperforming alternative classifiers. Notably, AdaBoost achieved perfect accuracy on the Glioma Grading Classification dataset, showcasing its exceptional ability to handle medical classification tasks effectively. While accuracy is a fundamental metric, we delved deeper into micro, macro, and weighted precision, recall, and F1 scores to provide a nuanced understanding of classifier performance.

AdaBoost's micro-level metrics consistently excelled, reflecting its capability to handle individual instances accurately. The macro-level metrics demonstrated AdaBoost's generalization across various classes, ensuring a balanced performance. Weighted metrics, accounting for class imbalance, further emphasized AdaBoost's robustness in maintaining accuracy in real-world scenarios.

Comparing AdaBoost to other classifiers, including Decision Trees, SVM, and a TensorFlow DL model, revealed AdaBoost's competitive edge. While some classifiers, such as SVM, achieved high accuracy on specific datasets, AdaBoost showcased its versatility by consistently performing well across various domains.

In terms of explainability, AdaBoost's reliance on combining multiple weak learners into a strong one may pose challenges in providing explicit explanations for individual predictions. However, its consistent high accuracy across datasets suggests that AdaBoost is reliable in offering correct classifications. The detailed analysis of precision, recall, and F1 scores further aids in understanding the nature of correct and incorrect classifications.

In conclusion, AdaBoost emerges as a powerful and versatile classification algorithm, particularly suited for scenarios where high accuracy is crucial. The study underscores the importance of evaluating classifiers comprehensively across multiple metrics and datasets, offering valuable insights into their strengths and limitations.

Several factors contribute to the success of AdaBoost, and a deeper understanding of its implementation sheds light on why it often outperforms other classifiers.

1- Sequential Weighting of Weak Learners:

AdaBoost employs a sequential training process where each weak learner is trained iteratively, and the emphasis is placed on misclassified instances from the previous rounds. This adaptive approach ensures that the algorithm focuses more on challenging instances, continuously refining its understanding of the dataset. The sequential weighting mechanism allows AdaBoost to correct errors made by previous weak learners, leading to improved overall performance.

2- Combination of Simple Classifiers:

AdaBoost combines multiple weak learners, often simple decision trees or stumps, to create a robust and accurate model. By leveraging a diverse set of weak learners, AdaBoost becomes more flexible and capable of capturing complex decision boundaries in the data. The combination of diverse models helps AdaBoost to generalize well and avoid overfitting.

3- Error Minimization:

AdaBoost places a strong emphasis on instances that were misclassified in previous rounds. By assigning higher weights to these instances, subsequent weak learners focus on correcting the errors, progressively reducing the overall classification error. This emphasis on correcting mistakes contributes to AdaBoost's ability to handle noisy datasets and outliers effectively.

4- Adaptive Feature Importance:

AdaBoost assigns different weights to features based on their importance in reducing classification errors. Features that contribute more to misclassifications receive higher weights, guiding subsequent weak learners to focus on these features during training. This adaptability to feature importance enhances AdaBoost's capability to capture intricate patterns in the data.

5- Ensemble Learning Principles:

The ensemble learning framework of AdaBoost ensures that it benefits from the collective wisdom of multiple weak learners. The combination of individual models creates a strong, robust learner capable of achieving high accuracy across a range of datasets. The weighted voting mechanism during classification ensures that each weak learner's contribution is appropriately considered.

6- Versatility Across Diverse Datasets:

AdaBoost's versatility stems from its ability to perform well across different types of datasets and classification tasks. The adaptive nature of AdaBoost enables it to handle both linear and non-linear decision boundaries, making it suitable for a broad spectrum of applications.

In summary, AdaBoost's success lies in its adaptive boosting mechanism, the combination of diverse weak learners, and its ability to prioritize challenging instances. The

algorithm's sequential approach to error minimization, adaptive feature importance, and ensemble learning principles contribute to its consistent high performance across various datasets.

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

import seaborn as sns

from sklearn import metrics

import pandas as pd

import numpy as np

import math

from sklearn import tree

from sklearn.tree import DecisionTreeClassifier

from pandas.api.types import is_string_dtype

from pandas.api.types import is_numeric_dtype

from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report, precision_score, recall_score, f1_score

import tensorflow as tf
```

```
from tensorflow import keras
from keras import layers, datasets, models
import numpy as np
import pandas as pd
from sklearn import model_selection, metrics
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
# 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)
```

- **Decision Tree 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
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=0)
dt_info_gain = DecisionTreeClassifier(criterion='entropy')
dt_gain_ratio = DecisionTreeClassifier(criterion='entropy',
splitter='best', max_features='log2')
dt_gini = DecisionTreeClassifier(criterion='gini')
svm_model = SVC(probability=True)
tf_model = tf.keras.Sequential([tf.keras.layers.Dense(64,
activation='relu'),
                                tf.keras.layers.Dense(1,
activation='sigmoid')])
scikit_dt_model = DecisionTreeClassifier()
classifiers = [dt_info_gain, dt_gain_ratio, dt_gini, svm_model,
tf_model, scikit_dt_model]
```

```
adaboost_model =
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(),
n_estimators=6)

adaboost_model.fit(X_train, y_train)

predictions = adaboost_model.predict(X_test)
```

- **Training (Decision Tree: 1-Information Gain, 2-Gain Ratio, 3-Gini Index)**

```
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())
```

- **Training (Extra credit: 1-Scikit SVM Model, 2-Tensorflow DL Model, 3-Scikit Decision Tree Model)**

```
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
```

- **AdaBoostClassifier**

```
class AdaBoostClassifier:


    def __init__(self,base_estimator,k):
        self.base_estimator = base_estimator
        self.k = k
```

```
        self.classifiers = []
        self.errors = []

    def fit(self,X,y):
        number_of_samples, number_of_features = X.shape
            w = np.full(number_of_samples, (1 / number_of_samples))
#weights
        for i in range(self.k):
                                                        indexes =
np.random.choice(number_of_samples,size=number_of_samples,replace=Tru
e,p=w)
            D_X_sampled,D_y_sampled = X[indexes],y[indexes]

            model = self.base_estimator
            model.fit(D_X_sampled,D_y_sampled)
            model_predictions = model.predict(X)

            error = np.sum(w*(model_predictions!=y))

            if error>0.5:
                i-=1
                continue

            for j in range(number_of_samples):
                if model_predictions[j] == y[j]:
                    w[j] *= error/(1-error)

            w = w/np.sum(w)

            self.classifiers.append(model)
            self.errors.append(error)
```

```
    def predict(self,X,y):
        class_weights = np.zeros((len(X), len(np.unique(y))))
        for i in range(self.k):
                                        weight_of_classifier_vote    =
np.log((1-self.errors[i])/self.errors[i])
            model_predictions = self.classifiers[i].predict(X)

            for j, model_prediction in enumerate(model_predictions):
                                class_weights[j,  model_prediction]  +=
weight_of_classifier_vote

        return np.argmax(class_weights, axis=1)

# %load_ext memory_profiler
# Evaluate accuracy
accuracy = accuracy_score(y_test, predictions)
print(f'Accuracy: {accuracy}')

# Compute and display the confusion matrix
cm = confusion_matrix(y_test, predictions)
print('Confusion Matrix:')
print(cm)

# Display the confusion matrix using seaborn heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                                    xticklabels=adaboost_model.classes_,
yticklabels=adaboost_model.classes_)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
# Classification Report
unique_labels = np.unique(y_test)
print('Classification Report:')
print(classification_report(y_test, predictions, labels=unique_labels, target_names=[str(c) for c in unique_labels]))

# Micro, Macro, and Weighted metrics
micro_precision = precision_score(y_test, predictions, average='micro')
micro_recall = recall_score(y_test, predictions, average='micro')
micro_f1 = f1_score(y_test, predictions, average='micro')

macro_precision = precision_score(y_test, predictions, average='macro')
macro_recall = recall_score(y_test, predictions, average='macro')
macro_f1 = f1_score(y_test, predictions, average='macro')

weighted_precision = precision_score(y_test, predictions, average='weighted')
weighted_recall = recall_score(y_test, predictions, average='weighted')
weighted_f1 = f1_score(y_test, predictions, average='weighted')

print('Micro Precision:', micro_precision)
print('Micro Recall:', micro_recall)
print('Micro F1 Score:', micro_f1)

print('Macro Precision:', macro_precision)
print('Macro Recall:', macro_recall)
print('Macro F1 Score:', macro_f1)
```

```
print('Weighted Precision:', weighted_precision)
print('Weighted Recall:', weighted_recall)
print('Weighted F1 Score:', weighted_f1)


# %memit -r 1
# end_time = time.time()
# execution_time = end_time - start_time
# print(f"Execution time: {execution_time} seconds")
```