\*\*Team Members-\*\*

1. Nikita Sharma – A20588827

2. Naga Sai Kumar Potti – A20539661

3. Nikita Rai – A20592025

4. Stephen Amankwa – A20529876

\*\*Boosting Trees\*\*

This project implements a Gradient Boosting Classifier entirely from first principles, using custom-built decision tree regressors.

## \*\*What Does the Model Do? \*\*

This implementation performs binary classification using the gradient boosting framework. Each iteration adds a regression tree trained on the residuals of the current predictions. The ensemble converges toward the true labels by minimizing log loss.

\*\*Features:- \*\*

- A handcrafted Regression Tree  
- Gradient boosting using residuals  
- Feature importance tracking  
- Subsampling (stochastic gradient boosting)  
- Custom metrics and data generation  
- Clean test cases with visualization

## \*\*How to Run:- \*\*

\*\*Run all test cases: \*\*  
python3 -m test\_visual\_classifier  
  
\*\*Run training script: \*\*  
python3 tree.py  
python3 metrics.py  
python3 dataset.py  
python3 gradient\_boosting.py

[Before running the main file (gradient\_boosting.py) we need to run - tree.py, metrics.py and dataset.py or else the program will throw error]

\*\*Testing & Evaluation:- \*\*

## \*\*Tested On Datasets: \*\*

- Clean Moons   
- Noisy Moons   
- Easy Classification   
- Overlapping Classes   
- High-Dimensional

## \*\*Testing Approach: \*\*

* Visual evaluation using plots (decision boundaries, ROC curves, confusion matrices)
* Metric-based validation (accuracy, precision, recall, F1-score)
* Stratified train-test splitting to preserve class balance
* Comprehensive testing across:
* Easy and hard decision boundaries
* Clean and noisy data
* Low- and high-dimensional feature spaces

\*\*Evaluation Metrics: \*\*

- Accuracy

- Confusion matrix

- ROC AUC

- Visualizations (decision boundary)

Accuracy is around 97%, showing high robustness on structured and lightly noisy data.

\*\*Parameters:- \*\*

Parameter Description

`n\_estimators` - Number of boosting iterations

`learning\_rate` - Shrinks tree contribution to prevent overfitting

`max\_depth` - Tree depth, controls model complexity

## \*\*Tunable Parameters: \*\*

GradientBoostingClassifier(  
 n\_estimators,  
 learning\_rate,  
 max\_depth,  
 subsample  
)

\*\*Testing Example:- \*\*  
  
1. \*\*Clean moons\*\*

X1, y1 = make\_moons(n\_samples=1000, noise=0.01, random\_state=42)

run\_case("Clean Moons", X1, y1)

2. \*\* Noisy moons\*\*

X2, y2 = make\_moons(n\_samples=1000, noise=0.25, random\_state=42)

run\_case("Noisy Moons", X2, y2)

3. \*\*Easy separation (class\_sep=2) \*\*

X3, y3 = make\_classification(1000, 4, informative=4, redundant=0, class\_sep=2, random\_state=42)

run\_case("Easy Classification", X3, y3)

4. \*\*Overlapping classes (class\_sep=0.5) \*\*

X4, y4 = make\_classification(1000, 4, informative=2, redundant=2, class\_sep=0.5, random\_state=42)

run\_case("Overlapping Classes", X4, y4)

5. \*\*High dimensional (10 features) \*\*

X5, y5 = make\_classification(1000, 10, informative=6, redundant=2, class\_sep=1.0, random\_state=42)

run\_case("High Dimensional", X5, y5)

\*\*Usage Example:- \*\*

# Test demo

if \_\_name\_\_ == "\_\_main\_\_":

from sklearn.datasets import make\_classification

X, y = make\_classification(n\_samples=100, n\_features=2, n\_informative=2,

n\_redundant=0, random\_state=42)

clf = GradientBoostingClassifier(n\_estimators=10, learning\_rate=0.1, max\_depth=2)

clf.fit(X, y)

preds = clf.predict(X)

acc = np.mean(preds == y)

print("Training accuracy:", acc)

\*\*Questions:- \*\*

\*\*Q1. What does the model you have implemented do, and when should it be used? \*\*

The model is a Gradient Boosting Classifier built from scratch. We have developed the base learners as custom decision tree regressors. It optimizes for logistic loss, making it suitable for binary classification tasks. The model uses an additive approach to build ensembles of weak learners, where each new tree attempts to fix the mistakes of the previous one. This approach is an application of gradient descent on function spaces.

The model works best in scenarios such as the following:

• When high-accuracy binary classification is required.

• While working with structured or tabular data, even when there is a slight amount of noise.

• When performance needs to be visualized with ROC curves, decision boundaries, and confusion matrices.

The model performs best with moderate bias and medium-sized datasets, where additive learning is especially important, like: fraud detection, medical diagnosis, or spam filtering.

\*\*Q2. How did you test your model to determine if it is working reasonably correctly? \*\*

The model was challenged with a number of synthetic and structured datasets that mimic real-world classification problems, including but not limited to:

• Low noise moons Cleanly separable data.

• Noisy overlapping class distributions.

• High-dimensional structured feature sets.

• Class imbalance and separability datasets.

To assess each scenario’s model performance, we used:

• Accuracy, precision, recall, F1-score

• Confusion matrix heatmap.

• ROC and AUC score graphs.

• Plots of decision boundaries (for 2D datasets).

The model proved to achieve the overall accuracy upto 97% with well-structured datasets. Even in datasets subjected to moderate noise, the model maintained reliable performance.

\*\*Q3. What parameters have you exposed to users of your implementation in order to tune performance? \*\*

Users can tune the model using the following hyperparameters:

| Parameter | Purpose |
| --- | --- |
| n\_estimators | Number of boosting iterations (i.e., trees in the ensemble) |
| learning\_rate | Shrinks each tree’s contribution to avoid overfitting |
| max\_depth | Controls the complexity of individual trees (bias-variance tradeoff) |

Example Usage:

model = CustomGradientBoostingClassifier(n\_estimators=300, learning\_rate=0.05, max\_depth=2)

model.fit(X\_train, y\_train)

predictions = model.predict(X\_test)

These parameters offer flexibility to control both model capacity and generalization strength.

\*\*Q4. Are there specific inputs that your implementation has trouble with? Could you work around these? \*\*

Yes, like with most boosting models, our implementation can be challenged by:  
Known Limitations:  
  
Known Limitations & Workarounds

Limitations:

a. Very noisy datasets

b. Unbalanced class distributions without stratification

c. Datasets where feature scaling dramatically affects splits

Workaround:

a. Use stratify=y in train/test splits

b. Add early stopping based on validation accuracy

c. Include feature selection or feature importance scoring

Known Limitations & Workarounds