Ex. No.: 8 a.

A PYTHON PROGRAM TO IMPLEMENT ADA BOOSTING

Aim:

To implement a python program for Ada Boosting.

Algorithm:

Step 1: Import Necessary Libraries

Import numpy as np.

Import pandas as pd.

Import DecisionTreeClassifier from sklearn.tree.

Import train_test_split from sklearn.model_selection.

Import accuracy_score from sklearn.metrics.

Step 2: Load and Prepare Data

Load your dataset using pd.read_csv() (e.g., df = pd.read_csv('data.csv')).

Separate features (X) and target (y).

Split the dataset into training and testing sets using train_test_split().

Step 3: Initialize Parameters

Set the number of weak classifiers n_estimators.

Initialize an array weights for instance weights, setting each weight to 1 / number_of_samples.

Step 4: Train Weak Classifiers

Loop for n_estimators iterations:

Train a weak classifier using DecisionTreeClassifier(max_depth=1) on the training data weighted by weights.

Predict the target values using the trained weak classifier.

Calculate the error rate err as the sum of weights of misclassified samples divided by the sum of all weights.

Compute the classifier's weight alpha using 0.5 * np.log((1 - err) / err).

Update the weights: multiply the weights of misclassified samples by np.exp(alpha)

and the weights of correctly classified samples by np.exp(-alpha).

Normalize the weights so that they sum to 1.

Append the trained classifier and its weight to lists classifiers and alphas.

Step 5: Make Predictions

For each sample in the testing set:

Initialize a prediction score to 0.

For each trained classifier and its weight:

Add the classifier's prediction (multiplied by its weight) to the prediction score.

Take the sign of the prediction score as the final prediction.

Step 6: Evaluate the Model

Compute the accuracy of the AdaBoost model on the testing set using

accuracy_score().

Step 7: Output Results

Print or plot the final accuracy and possibly other evaluation metrics.

PROGRAM:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree
from mlxtend.plotting import plot_decision_regions

df = pd.DataFrame()
df['X1'] = [1,2,3,4,5,6,6,7,9,9]
df['X2'] = [5,3,6,8,1,9,5,8,9,2]
df['label'] = [1,1,0,1,0,1,0,1,0,0]

df['weights'] = 1 / df.shape[0]
display(df)

sns.scatterplot(x='X1', y='X2', hue='label', data=df)
plt.show()

x = df[['X1','X2']].values
```

```
y = df['label'].values

dt1 = DecisionTreeClassifier(max_depth=1)
dt1.fit(x, y)

plt.figure(figsize=(8,4))
plot_tree(dt1, filled=True, feature_names=['X1','X2'])
plt.show()

plot_decision_regions(x, y, clf=dt1, legend=2)
plt.show()

df['y_pred'] = dt1.predict(x)
display(df)

def calculate_model_weight(error):
    return 0.5 * np.log((1 - error) / error)

error = np.sum(df['weights'] * (df['label'] != df['y_pred']))
alpha1 = calculate_model_weight(error)
print(f"Model 1 alpha: {alpha1:.3f}")

def update_row_weights(row, alpha):
    if row['label'] == row['y_pred']:
        return row['weights'] * np.exp(-alpha)
    else:
        return row['weights'] * np.exp(alpha)

df['updated_weights'] = df.apply(lambda row: update_row_weights(row,
alpha1), axis=1)
display(df)

df['normalized_weights'] = df['updated_weights'] /
df['updated_weights'].sum()
display(df)

print(f"Sum normalized weights: {df['normalized_weights'].sum()}")

df['cumsum_upper'] = np.cumsum(df['normalized_weights'])
df['cumsum_lower'] = df['cumsum_upper'] - df['normalized_weights']

display(df[['X1','X2','label','weights','y_pred','updated_weights','normal
ized_weights','cumsum_lower','cumsum_upper']])

def create_new_dataset(df):
```

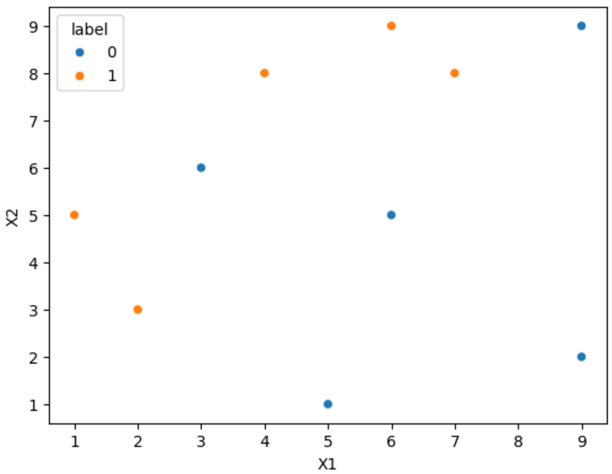
```
    indices = []
    n = df.shape[0]
    if n == 0:
        return indices
    for _ in range(n):
        a = np.random.random()
        for idx, row in df.iterrows():
            if row['cumsum_lower'] < a <= row['cumsum_upper']:
                indices.append(idx)
                break
    if len(indices) == 0:
        indices = np.random.choice(df.index, size=n,
replace=True).tolist()
    return indices

index_values = create_new_dataset(df)
print("Resampled indices (1st):", index_values)

second_df = df.loc[index_values, ['X1','X2','label','normalized_weights']]
second_df = second_df.reset_index(drop=True)
display(second_df)

x2 = second_df[['X1','X2']].values
y2 = second_df['label'].values

dt2 = DecisionTreeClassifier(max_depth=1)
dt2.fit(x2, y2)

plt.figure(figsize=(8,4))
plot_tree(dt2, filled=True, feature_names=['X1','X2'])
plt.show()

plot_decision_regions(x2, y2, clf=dt2, legend=2)
plt.show()

second_df['y_pred'] = dt2.predict(x2)

error2 = np.sum(second_df['normalized_weights'] * (second_df['label'] !=
second_df['y_pred']))
alpha2 = calculate_model_weight(error2)
print(f"Model 2 alpha: {alpha2:.3f}")

def update_row_weights_2(row, alpha=alpha2):
    if row['label'] == row['y_pred']:
        return row['normalized_weights'] * np.exp(-alpha)
```

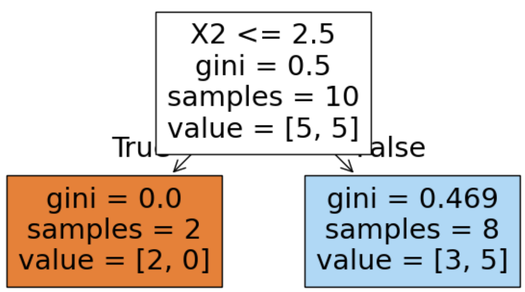
```
    else:
        return row['normalized_weights'] * np.exp(alpha)

second_df['updated_weights'] = second_df.apply(update_row_weights_2,
axis=1)
# Add epsilon to avoid zero-sum
second_df['updated_weights'] += 1e-10
second_df['normalized_weights'] = second_df['updated_weights'] /
second_df['updated_weights'].sum()

print("Second df normalized weights sum:",
second_df['normalized_weights'].sum())
print("Second df normalized weights:\n", second_df['normalized_weights'])
print("Second df shape:", second_df.shape)

second_df['cumsum_upper'] = np.cumsum(second_df['normalized_weights'])
second_df['cumsum_lower'] = second_df['cumsum_upper'] -
second_df['normalized_weights']

display(second_df[['X1','X2','label','normalized_weights','y_pred','cumsum
_lower','cumsum_upper']])

index_values2 = create_new_dataset(second_df)
print("Resampled indices (2nd):", index_values2)
third_df = second_df.loc[index_values2,
['X1','X2','label','normalized_weights']]
third_df = third_df.reset_index(drop=True)
display(third_df)

x3 = third_df[['X1','X2']].values
y3 = third_df['label'].values

if x3.shape[0] == 0:
    print("Warning: third_df is empty, skipping training dt3")
else:
    dt3 = DecisionTreeClassifier(max_depth=1)
    dt3.fit(x3, y3)

    plt.figure(figsize=(8,4))
    plot_tree(dt3, filled=True, feature_names=['X1','X2'])
    plt.show()

    plot_decision_regions(x3, y3, clf=dt3, legend=2)
    plt.show()
```

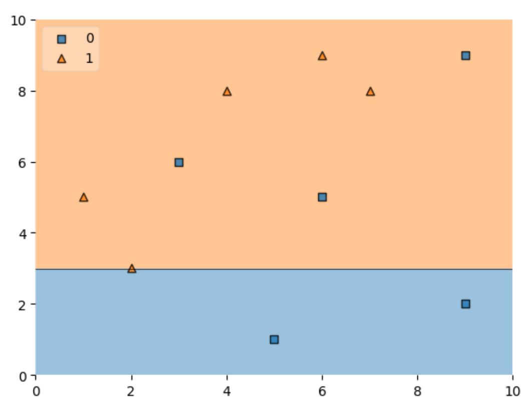
```
    third_df['y_pred'] = dt3.predict(x3)

    error3 = np.sum(third_df['normalized_weights'] * (third_df['label'] !=
third_df['y_pred']))
    alpha3 = calculate_model_weight(error3)
    print(f"Model 3 alpha: {alpha3:.3f}")

    print("Alphas:", alpha1, alpha2, alpha3)

    query = np.array([1,5]).reshape(1,2)
    pred1 = dt1.predict(query)[0]
    pred2 = dt2.predict(query)[0]
    pred3 = dt3.predict(query)[0]

    combined_score = alpha1*pred1 + alpha2*pred2 + alpha3*pred3
    final_pred = np.sign(combined_score)

    print(f"Query point {query.flatten()} predictions:")
    print(f" dt1: {pred1}, dt2: {pred2}, dt3: {pred3}")
    print(f" Combined weighted sum: {combined_score}")
    print(f" Final prediction (sign): {final_pred}")

    query2 = np.array([9,9]).reshape(1,2)
    pred1_2 = dt1.predict(query2)[0]
    pred2_2 = dt2.predict(query2)[0]
    pred3_2 = dt3.predict(query2)[0]

    combined_score2 = alpha1*pred1_2 + alpha2*pred2_2 + alpha3*pred3_2
    final_pred2 = np.sign(combined_score2)

    print(f"\nQuery point {query2.flatten()} predictions:")
    print(f" dt1: {pred1_2}, dt2: {pred2_2}, dt3: {pred3_2}")
    print(f" Combined weighted sum: {combined_score2}")
    print(f" Final prediction (sign): {final_pred2}")
```

OUTPUT :

	X1	X2	label	weights
0	1	5	1	0.1
1	2	3	1	0.1
2	3	6	0	0.1
3	4	8	1	0.1
4	5	1	0	0.1
5	6	9	1	0.1
6	6	5	0	0.1
7	7	8	1	0.1
8	9	9	0	0.1
9	9	2	0	0.1







	X1	X2	label	weights	y_pred
0	1	5	1	0.1	1
1	2	3	1	0.1	1
2	3	6	0	0.1	1
3	4	8	1	0.1	1
4	5	1	0	0.1	0
5	6	9	1	0.1	1
6	6	5	0	0.1	1
7	7	8	1	0.1	1
8	9	9	0	0.1	1
9	9	2	0	0.1	0

```
Model 1 alpha: 0.424
```

	X1	X2	label	weights	y_pred	updated_weights
0	1	5	1	0.1	1	0.065465
1	2	3	1	0.1	1	0.065465
2	3	6	0	0.1	1	0.152753
3	4	8	1	0.1	1	0.065465
4	5	1	0	0.1	0	0.065465
5	6	9	1	0.1	1	0.065465
6	6	5	0	0.1	1	0.152753
7	7	8	1	0.1	1	0.065465
8	9	9	0	0.1	1	0.152753
9	9	2	0	0.1	0	0.065465

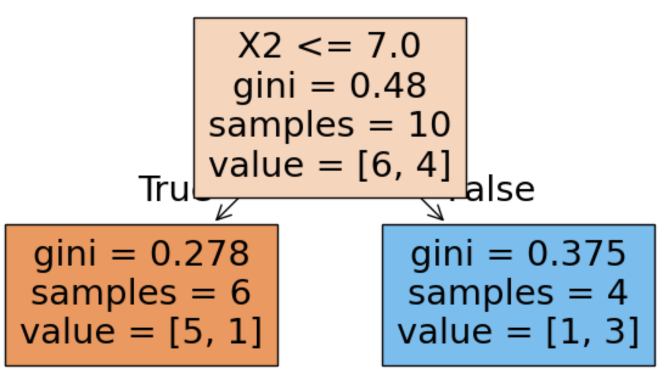
	X1	X2	label	weights	y_pred	updated_weights	normalized_weights
0	1	5	1	0.1	1	0.065465	0.071429
1	2	3	1	0.1	1	0.065465	0.071429
2	3	6	0	0.1	1	0.152753	0.166667
3	4	8	1	0.1	1	0.065465	0.071429
4	5	1	0	0.1	0	0.065465	0.071429
5	6	9	1	0.1	1	0.065465	0.071429
6	6	5	0	0.1	1	0.152753	0.166667
7	7	8	1	0.1	1	0.065465	0.071429
8	9	9	0	0.1	1	0.152753	0.166667
9	9	2	0	0.1	0	0.065465	0.071429

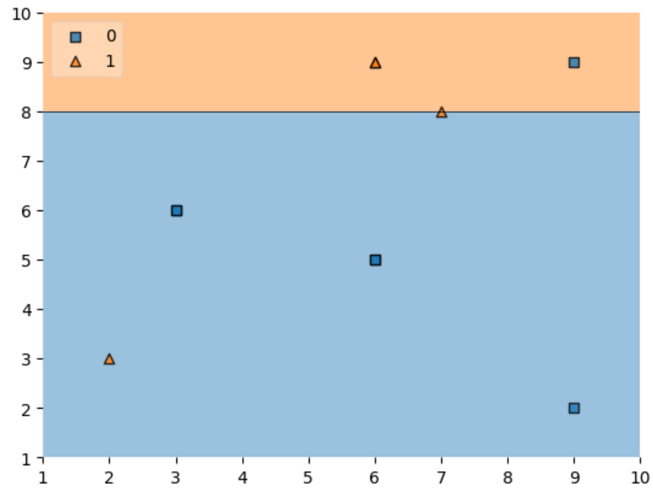
```
Sum normalized weights: 0.9999999999999999
```

	X1	X2	label	weights	y_pred	updated_weights	normalized_weights	cumsum_lower	cumsum_upper
0	1	5	1	0.1	1	0.065465	0.071429	0.000000	0.071429
1	2	3	1	0.1	1	0.065465	0.071429	0.071429	0.142857
2	3	6	0	0.1	1	0.152753	0.166667	0.142857	0.309524
3	4	8	1	0.1	1	0.065465	0.071429	0.309524	0.380952
4	5	1	0	0.1	0	0.065465	0.071429	0.380952	0.452381
5	6	9	1	0.1	1	0.065465	0.071429	0.452381	0.523810
6	6	5	0	0.1	1	0.152753	0.166667	0.523810	0.690476
7	7	8	1	0.1	1	0.065465	0.071429	0.690476	0.761905
8	9	9	0	0.1	1	0.152753	0.166667	0.761905	0.928571
9	9	2	0	0.1	0	0.065465	0.071429	0.928571	1.000000

```
Resampled indices (1st): [6, 9, 1, 5, 6, 5, 2, 7, 8, 2]
```

	X1	X2	label	normalized_weights
0	6	5	0	0.166667
1	9	2	0	0.071429
2	2	3	1	0.071429
3	6	9	1	0.071429
4	6	5	0	0.166667
5	6	9	1	0.071429
6	3	6	0	0.166667
7	7	8	1	0.071429
8	9	9	0	0.166667
9	3	6	0	0.166667



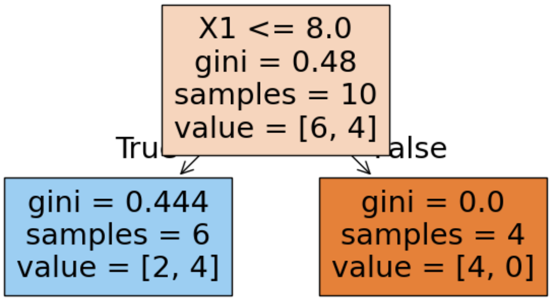


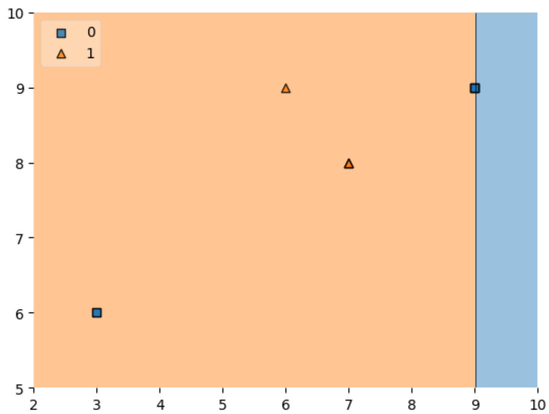

```
Model 2 alpha: 0.582
Second df normalized weights sum: 1.0000000000000002
Second df normalized weights:
 0    0.097222
1    0.041667
2    0.133333
3    0.041667
4    0.097222
5    0.041667
6    0.097222
7    0.041667
8    0.311111
9    0.097222
Name: normalized_weights, dtype: float64
Second df shape: (10, 6)
```

	X1	X2	label	normalized_weights	y_pred	cumsum_lower	cumsum_upper
0	6	5	0	0.097222	0	0.000000	0.097222
1	9	2	0	0.041667	0	0.097222	0.138889
2	2	3	1	0.133333	0	0.138889	0.272222
3	6	9	1	0.041667	1	0.272222	0.313889
4	6	5	0	0.097222	0	0.313889	0.411111
5	6	9	1	0.041667	1	0.411111	0.452778
6	3	6	0	0.097222	0	0.452778	0.550000
7	7	8	1	0.041667	1	0.550000	0.591667
8	9	9	0	0.311111	1	0.591667	0.902778
9	3	6	0	0.097222	0	0.902778	1.000000

```
Resampled indices (2nd): [8, 7, 7, 7, 9, 8, 9, 8, 8, 5]
```

	X1	X2	label	normalized_weights
0	9	9	0	0.311111
1	7	8	1	0.041667
2	7	8	1	0.041667
3	7	8	1	0.041667
4	3	6	0	0.097222
5	9	9	0	0.311111
6	3	6	0	0.097222
7	9	9	0	0.311111
8	9	9	0	0.311111
9	6	9	1	0.041667






```
Model 3 alpha: 0.711
Alphas: 0.4236489301936017 0.5815754049028405 0.7106928404470749
Query point [1 5] predictions:
 dt1: 1, dt2: 0, dt3: 1
 Combined weighted sum: 1.1343417706406766
 Final prediction (sign): 1.0

Query point [9 9] predictions:
 dt1: 1, dt2: 1, dt3: 0
 Combined weighted sum: 1.0052243350964423
 Final prediction (sign): 1.0
```

RESULT:

Thus the python program to implement Adaboosting has been executed successfully and the results have been verified and analyzed.

Ex. No.: 8b

A PYTHON PROGRAM TO IMPLEMENT GRADIENT BOOSTING

Aim:

To implement a python program using the gradient boosting model.

Algorithm:

Step 1: Import Necessary Libraries

Import numpy as np.

Import pandas as pd.

Import train_test_split from sklearn.model_selection.

Import DecisionTreeRegressor from sklearn.tree.

Import mean_squared_error from sklearn.metrics.

Step 2: Prepare the Data

Load your dataset into a DataFrame using pd.read_csv('your_dataset.csv').

Split the dataset into features (X) and target (y).

Use train_test_split to split the data into training and testing sets.

Step 3: Initialize Parameters

Set the number of boosting rounds (e.g., n_estimators = 100).

Set the learning rate (e.g., learning_rate = 0.1).

Initialize an empty list to store the weak learners (decision trees).

Initialize an empty list to store the learning rates for each round.

Step 4: Initialize the Base Model

Compute the initial prediction as the mean of the target values (e.g., F0 = np.mean(y_train)).

Initialize the predictions to the base model's prediction (e.g., F

= np.full(y_train.shape, F0)).

Step 5: Iterate Over Boosting Rounds

For each boosting round:

Compute the pseudo-residuals (negative gradient of the loss function) (e.g., residuals

= y_train - F).

Fit a decision tree to the pseudo-residuals.

Make predictions using the fitted tree (e.g., tree_predictions = tree.predict(X_train)).

Update the predictions by adding the learning rate multiplied by the tree predictions

(e.g., F += learning_rate * tree_predictions).

Append the fitted tree and the learning rate to their respective lists.

Step 6: Make Predictions on Test Data

Initialize the test predictions with the base model's prediction (e.g., F_test =

np.full(y_test.shape, F0)).

For each fitted tree and its learning rate:

Make predictions on the test data using the fitted tree.

Update the test predictions by adding the learning rate multiplied by the tree predictions.

Step 7: Evaluate the Model

Compute the mean squared error on the training data.

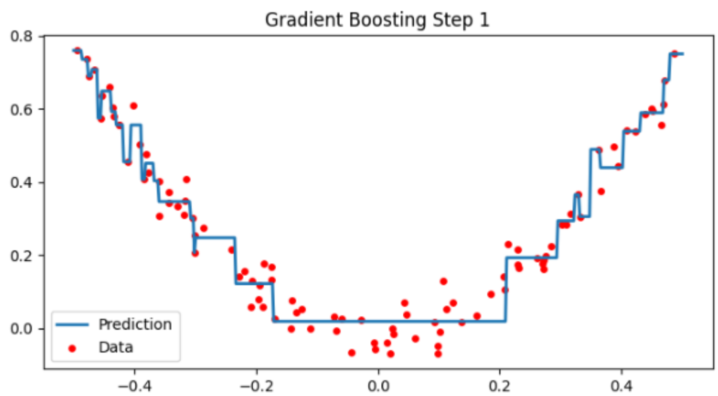
Compute the mean squared error on the test data.

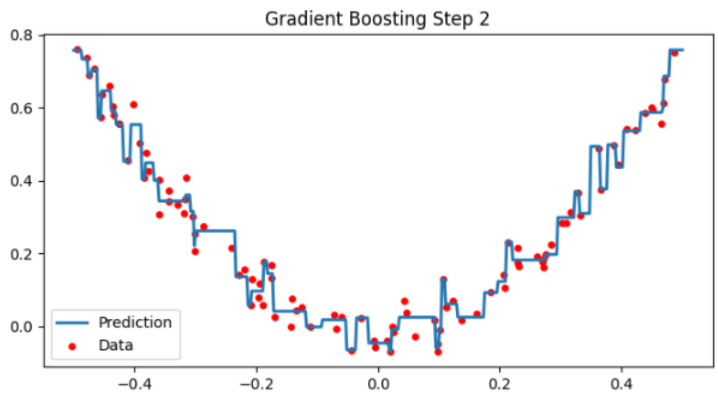
PROGRAM:

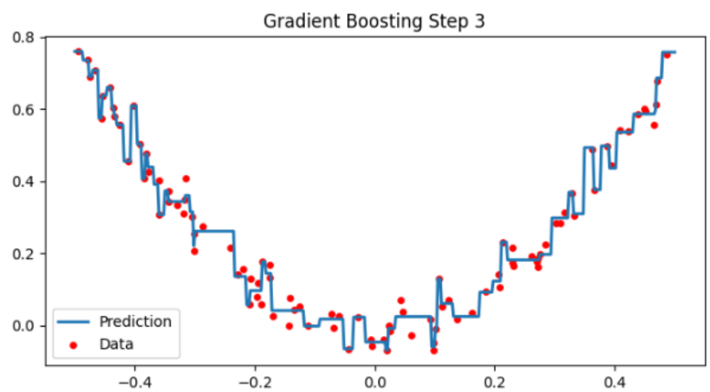
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100)

def gradient_boost(X, y, number, lr, count=1, regs=None, residual=None,
original_y=None):
    if regs is None:
        regs = []
    if original_y is None:
        original_y = y.copy()
    if residual is None:
        residual = y.copy()

    if number == 0:
        return regs

    tree_reg = DecisionTreeRegressor(max_depth=5, random_state=42)
```

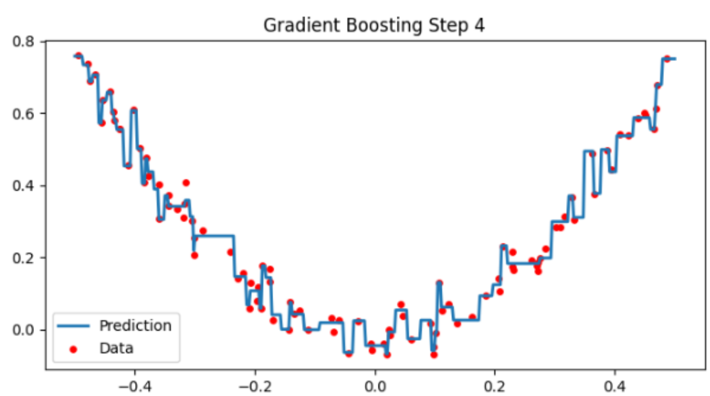
```
    tree_reg.fit(X, residual)
    regs.append(tree_reg)

    # Predict the sum of all trees scaled by learning rate
    x1 = np.linspace(-0.5, 0.5, 500).reshape(-1, 1)
    y_pred = sum(lr * reg.predict(x1) for reg in regs)

    # Plotting
    plt.figure(figsize=(8,4))
    plt.plot(x1, y_pred, linewidth=2, label='Prediction')
    plt.scatter(X, original_y, color='red', s=15, label='Data')
    plt.title(f'Gradient Boosting Step {count}')
    plt.legend()
    plt.show()

    # Update residuals for next iteration
    residual = original_y - sum(lr * reg.predict(X) for reg in regs)

    # Recursive call
    return gradient_boost(X, y, number - 1, lr, count + 1, regs, residual,
original_y)

regs = gradient_boost(X, y, number=5, lr=1)
```

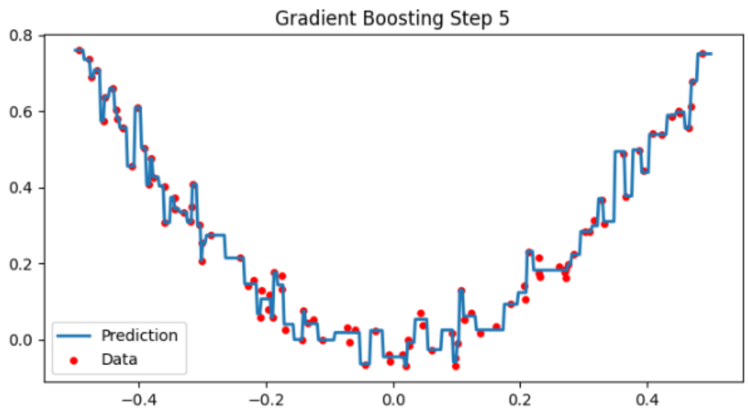
OUTPUT:


Gradient Boosting Step 1
Prediction
Data
0.8
0.6
0.4
0.2
0.0
−0.4
−0.2
0.0
0.2
0.4


Gradient Boosting Step 2
Prediction
Data
0.8
0.6
0.4
0.2
0.0
−0.4
−0.2
0.0
0.2
0.4


Gradient Boosting Step 3
Prediction
Data
0.8
0.6
0.4
0.2
0.0
−0.4
−0.2
0.0
0.2
0.4


Gradient Boosting Step 4
Prediction
Data
0.8
0.6
0.4
0.2
0.0
−0.4
−0.2
0.0
0.2
0.4



RESULT:

Thus, the python program to implement gradient boosting for the standard uniform distribution has been successfully implemented and the results have been verified and analyzed.