Project 2

Group Name: A20560777

Group Members:

Pooja Pranavi Nalamothu (CWID: A20560777)

Gradient Boosting Tree

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

Gradient Boosting Tree (GBT) is an algorithm used in supporting machine learning for regression problems. It develops an additive model that is constructed sequentially through using predictions from decisions trees which each step aims at capturing residuals of the trees employed earlier on. This approach makes it flexible for use when dealing with large amounts of datasets, and good for recognizing non-linear patterns.

When to use it:

* When you are working with numeric features and your target variable is a continuous variable (regression problem).
* When it is difficult to express a dependence between an independent variable and a target by reference to the linear equation of regression.
* When it is necessary to understand, which features affect models’ decisions in decision trees.
* It is most appropriate for datasets of considerable size but not for datasets to which a large number of observations is attributed as the GBT models can be resource demanding when used on large datasets.

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

Synthetic Dataset:

To further evaluate this model, we tested it on a synthetic data which was created using a linear weighted sum of all the features with some noise added. The use of the model was an efficient approach to reducing the Mean Squared Error (MSE) and proposed values that were near the actual numbers.

Real Datasets:

Applied the model to various real-world datasets:

* Energy Efficiency Dataset: Predicted heating loads in buildings.
* Medical Cost Dataset: Forecast of insurance to charges depending on demographic and health characteristics.
* Auto MPG Dataset: In these models fuel efficiency in terms of car’s MPG has been predicted based on vehicle characteristics.
* Wine Quality Dataset: To reflect the quality of wine three general metrics of quality where used to predict the quality scores of the wine given in function of the chemical features.

Validation Metrics:

* Summarized the model’s validation by manually calculating the Mean Squared Error (MSE) for accuracy’s sake, not without prebuilt libraries.
* Calculated probabilities of the light level to ensure that they were realistic, that is, within the optimal range of the target values and consistent with the data trends.

Debugging:

* Measures the numbers of input features, targets, and prediction to standardization among datasets.
* The findings were then analyzed relative to baseline models such as mean predictions to establish improved outcomes.

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

The following parameters are exposed in the implementation:

n\_estimators: Number of decision trees to train.

* Higher values allow better learning but increase training time.
* Default: 50.

learning\_rate: Determines the contribution of each tree to the final prediction.

* Smaller values prevent overfitting but require more trees.
* Default: 0.1.

max\_depth: Maximum depth of individual trees.

* Controls the complexity of each tree and prevents overfitting.
* Default: 3.

loss: Loss function for computing gradients.

* Current implementation supports squared error for regression

1. **Are there specific inputs that your implementation has trouble with?**

Categorical Features:

It is also important to note that the current development does not directly cater for categorical variables. Such features must be encoded by users using one-hot encoding or label encoding.

Large Datasets:

An important limitation of handling very large datasets (e.g., millions of samples) involves high computational costs, owing to the sequential structure of gradient boosting.

Imbalanced Data:

In the case of having highly skewed distributions in a target variable, it becomes hard for a model to predict an event that rarely occurs.

High-Dimensional Features:

In particular, if the number of features is large enough and varies, for example, in thousands, then the splits of the decision tree are also large enough and may not generalize well, so the model will begin to over-fit.

**Given more time, could you work around these, or is it fundamental?**

Yes, most of these issues can be addressed with additional time:

* Categorical Handling: One of the modifications carried out is to try to incorporate directly splits of categorical variables in the decision tree.
* Scalability: The use of parallelization, or incremental boosting technique such as chunking of the data will go a long way in ensuring that the techniques do not give out negative results.
* Loss Function Extension: It will also enhance its support for other types of losses such as absolute error, Huber, or type of classification loss.

However, some challenges such as overfitting when working with high dimensional features are all inherent characteristics of a decision tree model and may only be solved either with external data preprocessing tools or by use of other architectures of the model.

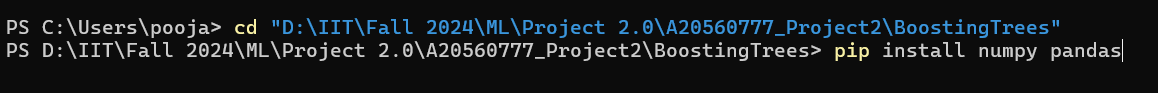
**How to run the code**

Step 1:

Clone or download the code: Ensure all the datasets are in the correct paths as mentioned in the code.

Step 2:

Install necessary dependencies



Step 3:

Run the script



Step 4:

Ensure all the datasets are properly loaded and called

Note: Update paths in the script, if required.

**Basic Usage:**

* For Synthetic Data, you can run this code directly

CODE:

import numpy as np

class DecisionTreeRegressor:

"""

A simple decision tree regressor for fitting residuals.

"""

def \_\_init\_\_(self, max\_depth=3):

self.max\_depth = max\_depth

self.tree = None

def \_split(self, X, y):

"""

Find the best split for a dataset.

"""

best\_split = {"feature": None, "threshold": None, "loss": float("inf")}

n\_samples, n\_features = X.shape

for feature in range(n\_features):

thresholds = np.unique(X[:, feature])

for threshold in thresholds:

left\_mask = X[:, feature] <= threshold

right\_mask = ~left\_mask

if np.sum(left\_mask) == 0 or np.sum(right\_mask) == 0:

continue

left\_residuals = y[left\_mask]

right\_residuals = y[right\_mask]

# Mean squared error as loss

loss = (

np.sum((left\_residuals - np.mean(left\_residuals)) \*\* 2) +

np.sum((right\_residuals - np.mean(right\_residuals)) \*\* 2)

)

if loss < best\_split["loss"]:

best\_split = {

"feature": feature,

"threshold": threshold,

"loss": loss,

"left\_mask": left\_mask,

"right\_mask": right\_mask,

}

return best\_split

def \_build\_tree(self, X, y, depth):

"""

Recursively build the decision tree.

"""

if depth >= self.max\_depth or len(set(y)) == 1:

return {"value": np.mean(y)}

split = self.\_split(X, y)

if split["feature"] is None:

return {"value": np.mean(y)}

left\_tree = self.\_build\_tree(X[split["left\_mask"]], y[split["left\_mask"]], depth + 1)

right\_tree = self.\_build\_tree(X[split["right\_mask"]], y[split["right\_mask"]], depth + 1)

return {

"feature": split["feature"],

"threshold": split["threshold"],

"left": left\_tree,

"right": right\_tree,

}

def fit(self, X, y):

self.tree = self.\_build\_tree(X, y, 0)

def \_predict\_one(self, x, tree):

"""

Predict a single sample using the tree.

"""

if "value" in tree:

return tree["value"]

feature = tree["feature"]

threshold = tree["threshold"]

if x[feature] <= threshold:

return self.\_predict\_one(x, tree["left"])

else:

return self.\_predict\_one(x, tree["right"])

def predict(self, X):

return np.array([self.\_predict\_one(x, self.tree) for x in X])

class GradientBoostingTree:

"""

Gradient Boosting Tree implementation with explicit gamma calculation.

"""

def \_\_init\_\_(self, n\_estimators=100, learning\_rate=0.1, max\_depth=3, loss="squared\_error"):

self.n\_estimators = n\_estimators

self.learning\_rate = learning\_rate

self.max\_depth = max\_depth

self.trees = []

self.init\_prediction = None

self.loss = loss

def \_gradient(self, y, y\_pred):

"""

Compute the gradient of the loss function.

"""

if self.loss == "squared\_error":

return y - y\_pred

raise ValueError("Unsupported loss function")

def \_gamma(self, residuals, region):

"""

Compute the optimal gamma for a region as per Equation (10.30).

"""

return np.mean(residuals[region])

def fit(self, X, y):

"""

Train the gradient boosting tree model.

"""

self.init\_prediction = np.mean(y) # Start with the mean prediction

predictions = np.full\_like(y, self.init\_prediction, dtype=np.float64)

for \_ in range(self.n\_estimators):

# Compute residuals (negative gradients)

residuals = self.\_gradient(y, predictions)

# Train a decision tree on residuals

tree = DecisionTreeRegressor(max\_depth=self.max\_depth)

tree.fit(X, residuals)

self.trees.append(tree)

# Update predictions with the tree's contribution

tree\_predictions = tree.predict(X)

for region in np.unique(tree\_predictions):

mask = tree\_predictions == region

gamma = self.\_gamma(residuals, mask)

predictions[mask] += self.learning\_rate \* gamma

def predict(self, X):

"""

Predict target values for input data X.

"""

predictions = np.full((X.shape[0],), self.init\_prediction, dtype=np.float64)

for tree in self.trees:

predictions += self.learning\_rate \* tree.predict(X)

return predictions

# Example Usage

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

# Import necessary libraries

import numpy as np

# Generate synthetic regression data

def make\_synthetic\_regression(n\_samples=100, n\_features=7, noise=0.1, random\_state=42):

np.random.seed(random\_state)

X = np.random.rand(n\_samples, n\_features) # Features: random values in [0, 1]

coefficients = np.random.rand(n\_features) # Random coefficients for linear relation

y = X @ coefficients + noise \* np.random.randn(n\_samples) # Linear relationship + noise

return X, y

# Compute mean squared error manually

def mean\_squared\_error(y\_true, y\_pred):

return np.mean((y\_true - y\_pred) \*\* 2)

# Generate data

X, y = make\_synthetic\_regression(n\_samples=100, n\_features=7, noise=0.1, random\_state=42)

y = y / np.std(y) # Normalize target for simplicity

# Train Gradient Boosting Tree

model = GradientBoostingTree(n\_estimators=50, learning\_rate=0.1, max\_depth=3, loss="squared\_error")

model.fit(X, y)

# Predict

predictions = model.predict(X)

# Evaluate

mse = mean\_squared\_error(y, predictions)

print(f"Mean Squared Error: {mse:.4f}")

print("Predictions for new data:", predictions[:10]) # Display first 10 predictions

* For other datasets

Ensure the correct dataset is loaded and is in the correct directory.

Define X and y for the corresponding features and target variables.

Once you've modified the file path, you can use pytest to run all tests, including real-world datasets and synthetic dataset

* Open a terminal in the directory containing your test script.
* Run Pytest with the following command: pytest test\_datasetname.py