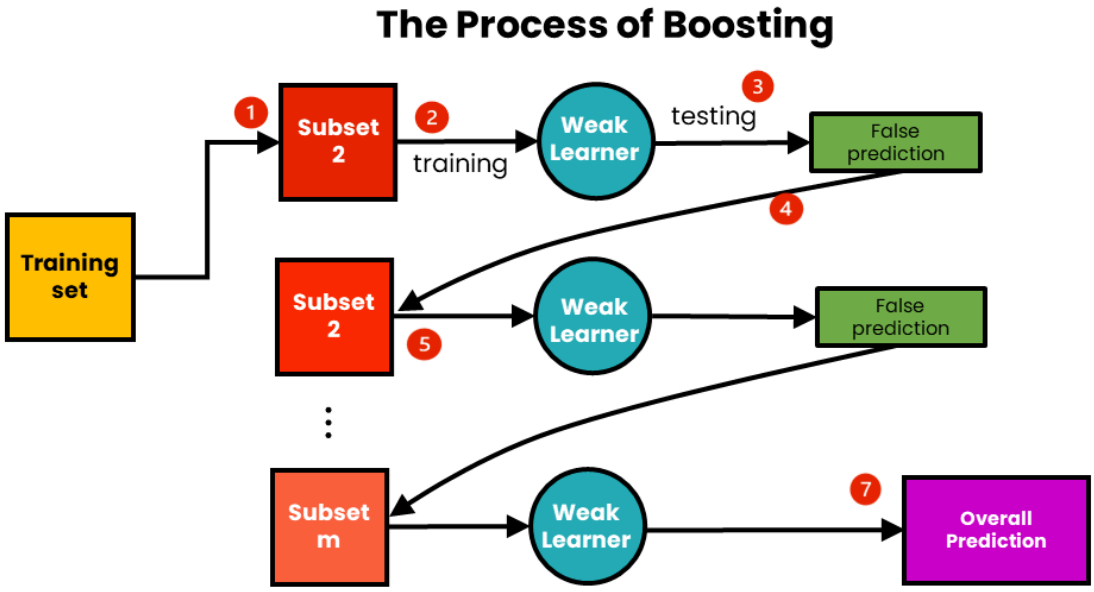
Gradient Boosting: Regression

Boosting



- Models are trained sequentially
- Most of the times all the models get **same data**
- **Reduces Bias**: Boosting focuses on correcting errors, which helps reduce bias.
    - eg. Shallow DT, Linear Models

Gradient Boosting

- Used for both regression and classification tasks.
- It is particularly effective when the data has complex, **non-linear relationships**
- It often performs well even with **little hyperparameter tuning**.
- Used in ML competitions

- Implementations in most major machine learning libraries, such as **scikit-learn, XGBoost, LightGBM, and CatBoost.**

Key Concepts of Gradient Boosting:

1. **Sequential Learning**:
    - Gradient Boosting builds models sequentially. Each new model focuses on correcting the mistakes (residuals) of the previous models.
2. **Weak Learners**:
    - The base models (weak learners) are usually shallow decision trees. These models are only slightly better than random guessing.
3. **Gradient Descent**:
    - The algorithm uses gradient descent to minimize a loss function (e.g., **mean squared error** for regression or **log loss** for classification).
4. **Additive Model**:
    - The final model is an additive combination of all the weak learners.
    - We treat each model as a **function** & the overall model is additive combination of all these **weaker models**.

How Gradient Boosting Works

1. **Initialize a Weak Model (Base Learner)**
    - The first model is a simple prediction (often the mean for regression or equal probability for classification).
2. **Calculate the Residuals (Errors)**
    - Compute the difference between actual and predicted values (errors).
3. **Train a New Model on Residuals**
    - A new decision tree learns from these errors and tries to reduce them.

4. **Update Predictions**
    - The new tree's predictions are added to the previous predictions, making the overall model better.
5. **Repeat Until Convergence**
    - The process continues, adding more trees until the model stops improving or reaches a set number of trees.

> 💡 `njobs` **is not available because here, we're sequentially training the models.**

High Level Overview:

Step 1: Find out $f_0(x)$→ The first function

i. Calculates the **mean** & gives it as a prediction

ii. Calculates the **residues (error)**

Step 2: the remaining models are DTs.

- Fit a DT
    - eg. Data is divided into 3 regions with `max_depth=3`
    - 👆 **DTs will be shallow**
- It will predict a result.

Step 3: Find out the Residue (error)

- Send the errors to the next model
- ***DT Input → X***
- ***y= Residues from prev model***

```
tree1.fit(df['X'].values.reshape(100,1),df['res1'].values)
```

- ***Predicted value:***

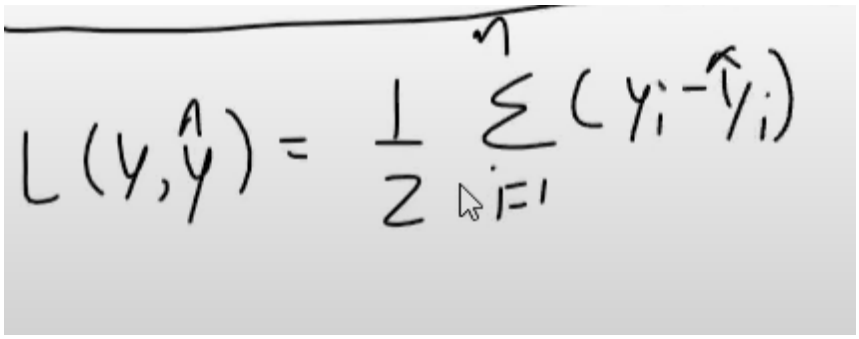
```
df['pred2'] = 0.265458 + tree1.predict(X_test.reshape(500, 1))
```

- `0.265458` is residue from prev model.
- REPEAT THIS PROCESS

Step 4: Combine all the results

- $m_0 + m_1 + m_2 + m_3$
- m_0 is the average and remaining are predictions from decision trees.
- ***DT Output → Sum of Prediction of all previous models***

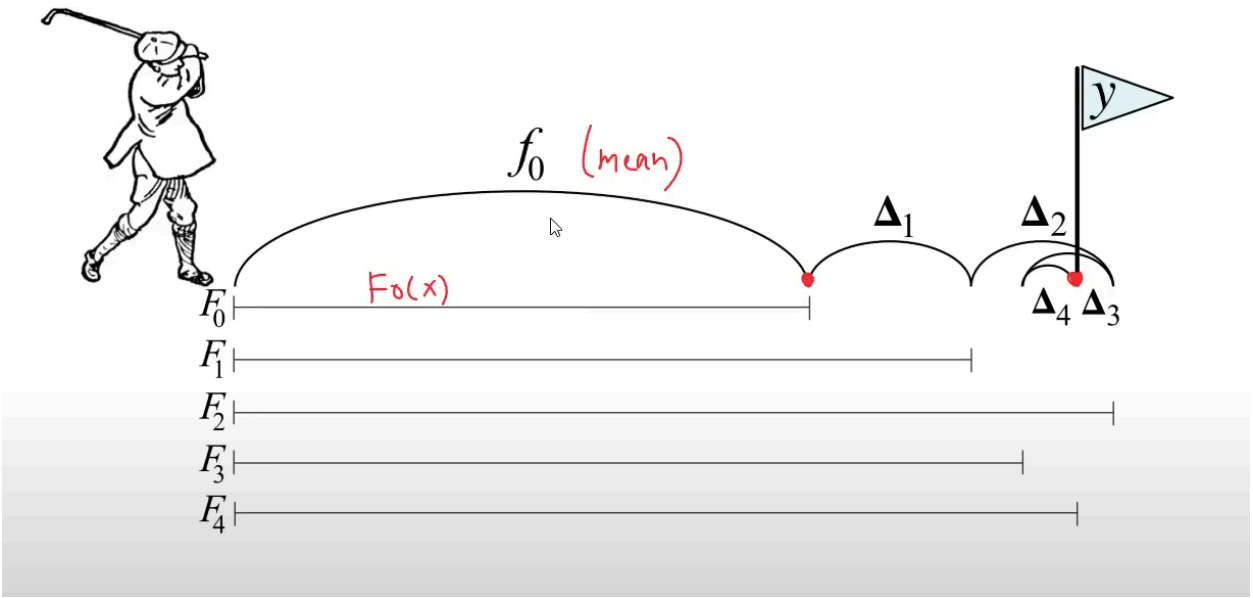
Loss Function in Gradient Boosting:



More errors

Errors have decreased

	X	y	pred1	res1	pred2	res2
0	-0.125460	0.051573	0.265458	-0.213885	0.018319	0.033254
1	0.450714	0.594480	0.265458	0.329021	0.605884	-0.011404
2	0.231994	0.166052	0.265458	-0.099407	0.215784	-0.049732
3	0.098658	-0.070178	0.265458	-0.335636	0.018319	-0.088497
4	-0.343981	0.343986	0.265458	0.078528	0.305964	0.038022
...	...	...	...	...	...	...
95	-0.006204	-0.040675	0.265458	-0.306133	0.018319	-0.058994
96	0.022733	-0.002305	0.265458	-0.267763	0.018319	-0.020624
97	-0.072459	0.032809	0.265458	-0.232650	0.018319	0.014489
98	-0.474581	0.689516	0.265458	0.424057	0.660912	0.028604
99	-0.392109	0.502607	0.265458	0.237148	0.487796	0.014810



- Parameter space is in case of linear regression when you assume a linear relationship.
  - $y = mx + c$
- In case of non-parametric models, we don't assume any distribution.
  - So, there can be infinite number of functions
- It is the space where models (functions) are searched for to map inputs to outputs.
- Linear regression applies gradient Descent in parametric space
  - ***Gradient Boosting does the same thing in functional space.***

Python code for Gradient Boost

```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
```

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Load the California Housing dataset
california = fetch_california_housing()
X = pd.DataFrame(california.data, columns=california.feature_names)
y = california.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
e=42)

# Initialize GradientBoostingRegressor
model = GradientBoostingRegressor(
    n_estimators=200,  # Number of boosting stages
    max_depth=4,       # Maximum depth of each tree
    random_state=42    # Random seed for reproducibility
)

# Train the model
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

r2 = r2_score(y_test, y_pred)


print(f"R^2 Score: {r2:.2f}")

# Feature Importance
feature_importance = model.feature_importances_
feature_names = california.feature_names

print("\nFeature Importance:")
```

```
for feature, importance in zip(feature_names, feature_importance):
    print(f"{feature}: {importance:.4f}")
```

```
R^2 Score: 0.82

Feature Importance:
MedInc: 0.5751
HouseAge: 0.0424
AveRooms: 0.0233
AveBedrms: 0.0090
Population: 0.0074
AveOccup: 0.1282
Latitude: 0.1035
Longitude: 0.1111
```

Key Hyperparameters in Gradient Boosting:

1. `n_estimators` *=100* :

- **Description**: The number of boosting stages (trees) to build.
- **Role**: More trees generally improve performance, but too many can lead to overfitting.
- **Typical Values**: 50 to 500 (start with 100 and adjust based on performance).

2. `learning_rate` *=0.1* :

- **Description**: Controls the contribution of each tree to the final model.
- **Role**: A smaller learning rate requires more trees (`n_estimators`) but can lead to better generalization.
    - **Small learning rate**: Takes small steps toward the minimum, which can lead to better convergence but requires more iterations.

    - **Large learning rate**: Takes larger steps, which may speed up training but risks overshooting the minimum or causing instability.
- **Typical Values**: **0.01 to 0.2** (start with 0.1 and adjust).

3. `max_depth =3` :

- **Description**: The maximum depth of each decision tree.
- **Role**: ***Deeper trees can capture more complex patterns but may overfit***.
- **Typical Values**: 3 to 10 (start with 3 or 4 and increase if needed).

4. `min_samples_split =2` :

- **Description**: The minimum number of samples required to split an internal node.
- **Role**: **Prevents overfitting** by controlling the growth of trees.
- **Typical Values**: 2 to 10 (start with 2 and increase if overfitting occurs).

5. `min_samples_leaf=1` :

- **Description**: The minimum number of samples required to be at a leaf node.
- **Role**: Prevents overfitting by ensuring leaves have a minimum number of samples.
- **Typical Values**: 1 to 5 (start with 1 and increase if overfitting occurs).

6. `subsample=1.0` :

- **Description**: The fraction of samples to use for training each tree.
- **Role**: Introduces randomness and can prevent overfitting.
- **Typical Values**: 0.8 to 1.0 (start with 1.0 and reduce if overfitting occurs).

7. `max_features =None` :

- **Description**: The number of features to consider when looking for the best split.

- **Role**: Controls the randomness in feature selection. Lower values reduce variance but increase bias.
- **Typical Values**: `'sqrt'` (square root of total features) or `'log2'` (logarithm of total features).

8. `loss` *`'squared_error'`*:

- **Description**: The loss function to optimize.
- **Role**: Determines the type of problem (regression or classification) and the loss to minimize.
- **Typical Values**:
  - For regression: `'squared_error'` (default), `'absolute_error'`, `'huber'`.
  - For classification: `'log_loss'` (default), `'exponential'`.

9. `random_state`:

- **Description**: Seed for random number generation.
- **Role**: Ensures reproducibility of results.
- **Typical Values**: Any integer (e.g., 42).

10. `verbose`:

- **Description**: Controls the amount of output during training.
- **Role**: Useful for debugging and monitoring training progress.
- **Typical Values**: 0 (no output), 1 (progress updates).

11. `validation_fraction=0.1`:

- **Description**: The fraction of training data to set aside for early stopping.
- **Role**: ***Helps prevent overfitting by stopping training if validation performance doesn't improve.***
- **Typical Values**: 0.1 to 0.2.

12. `n_iter_no_change` *`=None`*:

- **Description**: The number of iterations to wait before stopping if validation performance doesn't improve.
- **Role**: Used for early stopping.
- **Typical Values**: 5 to 10.

13. `ccp_alpha=0.0` :

- **Description**: Complexity parameter for pruning trees.
- **Role**: Controls the trade-off between model complexity and accuracy.
- **Typical Values**: 0.0 (no pruning) to 0.1 (aggressive pruning).