AdaBoost (Adaptive Boosting)

`from sklearn.ensemble import` `AdaBoostClassifier`

`from sklearn.ensemble import` `AdaBoostRegressor`

- *`loss='linear'`*

`estimator=None`

`AdaBoostClassifier`

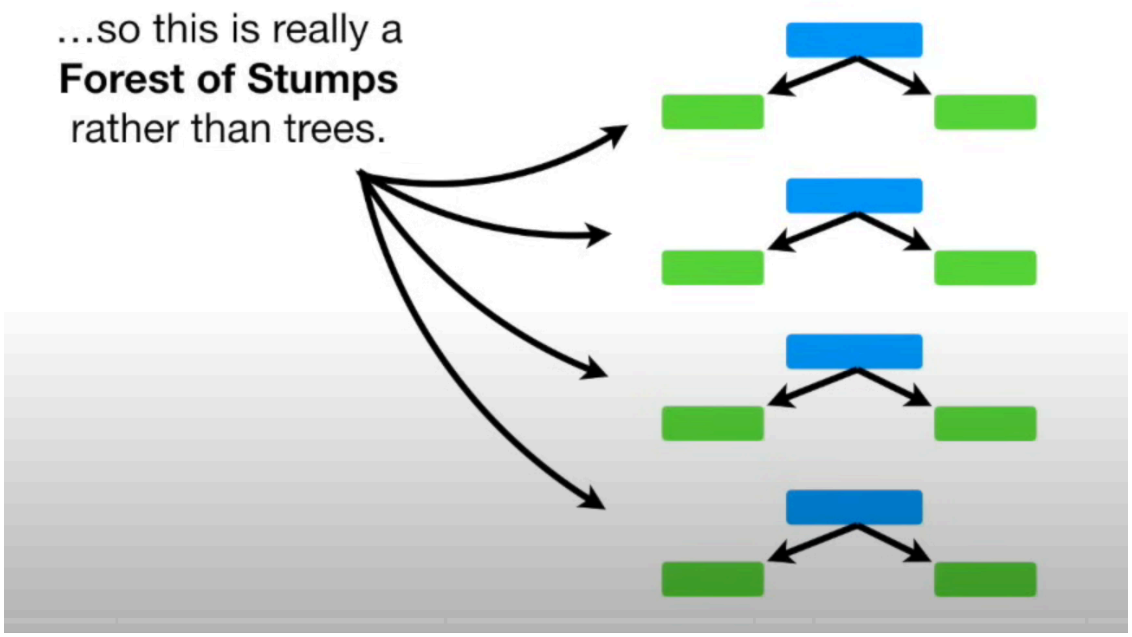
- If `None`, then the base estimator is [`DecisionTreeClassifier`](#) initialized with `max_depth=1`.

`AdaBoostRegressor`

- If `None`, then the base estimator is [`DecisionTreeRegressor`](#) initialized with `max_depth=3`

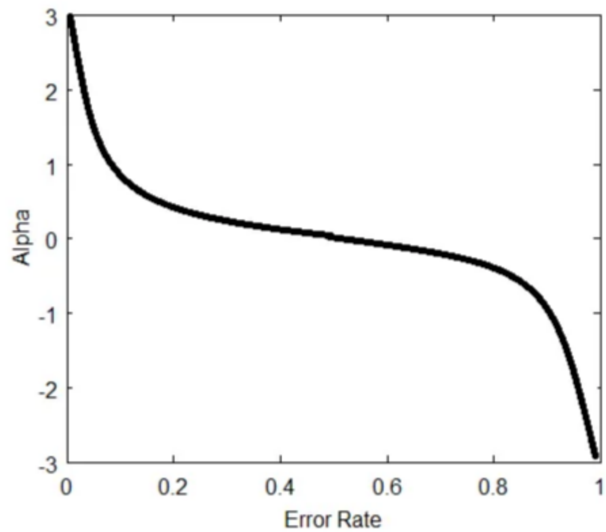
`n_estimators=50` `,` *`learning_rate=1.0`* `,`

- Combines Decision Trees with a depth of 1, called **Decision Stumps.**
- It works by focusing on the mistakes of previous models and giving more weight to the difficult-to-predict samples.
- **Combines multiple weak classifiers** to form a **strong classifier**.
- It assigns **weights to misclassified samples** to improve future predictions.



How Does AdaBoost Work?

1. **Train Weak Learner**:
    - A weak model (e.g., Decision Tree with depth = 1) is trained on the data.
2. **Calculate Errors**:
    - Identify misclassified samples.
3. **Adjust Weights**:
    - Initial weigh → $1/n$
    - Increase (**Boost**) the weight of misclassified samples, making them more important for the next weak learner.
4. **Train Next Weak Learner**:
    - The new weak learner focuses more on the misclassified samples.
5. **Repeat Process**:
    - Combine multiple weak learners into a strong learner.



$$\alpha_t = \frac{1}{2} ln \frac{(1 - TotalError)}{TotalError}$$

- 👆Formula to calculate the model weight.

Advantages of AdaBoost

1. **High Accuracy**:
    - Often achieves high accuracy by combining multiple weak models.
2. **No Need for Parameter Tuning**:
    - AdaBoost has fewer hyperparameters compared to other algorithms.
3. **Handles Both Classification and Regression**:
    - Can be used for both classification (AdaBoostClassifier) and regression (AdaBoostRegressor).

Disadvantages of AdaBoost

1. **Sensitive to Noisy Data**:
   - AdaBoost can overfit if the data contains noise or outliers.
2. **Computationally Expensive**:
   - Training multiple models can be computationally expensive.
3. **Requires Weak Models**:
   - The performance of AdaBoost depends on the quality of the weak models.
4. S**lower Training**:
   - Training multiple models can be computationally expensive.

Why Use AdaBoost?

1. **Improves Accuracy**:
   - By combining multiple weak models, AdaBoost can achieve high accuracy.
2. **Handles Complex Data**:
   - AdaBoost can capture complex patterns in the data by focusing on difficult samples.
3. **No Need for Deep Trees**:
   - Unlike Random Forest, AdaBoost uses very simple models (e.g., Decision Stumps), which are faster to train.

Python code for AdaBoost

Dataset: Telco Customer Churn

gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	...	DeviceProtection	TechSupport	StreamingTV	Strea
Female	0	Yes	No	1	No	No phone service	DSL	No	...	No	No	No	
Male	0	No	No	34	Yes	No	DSL	Yes	...	Yes	No	No	
Male	0	No	No	2	Yes	No	DSL	Yes	...	No	No	No	
Male	0	No	No	45	No	No phone service	DSL	Yes	...	Yes	Yes	No	
Female	0	No	No	2	Yes	No	Fiber optic	No	...	No	No	No	
...	...	...	...	...	...	...	...	...	...	...	...	...	
Male	0	Yes	Yes	24	Yes	Yes	DSL	Yes	...	Yes	Yes	Yes	
Female	0	Yes	Yes	72	Yes	Yes	Fiber optic	No	...	Yes	No	Yes	
Female	0	Yes	Yes	11	No	No phone service	DSL	Yes	...	No	No	No	
Male	1	Yes	No	4	Yes	Yes	Fiber optic	No	...	No	No	No	
Male	0	No	No	66	Yes	No	Fiber optic	Yes	...	Yes	Yes	Yes	

```
import pandas as pd
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset (replace with your dataset)
# Example: Telco Customer Churn dataset
data = pd.read_csv("https://raw.githubusercontent.com/treselle-systems/customer_churn_analysis/refs/heads/master/WA_Fn-UseC_-Telco-Customer-Churn.csv")

# Preprocess the data (simplified example)
X = data.drop(columns=["Churn"])  # Features
y = data["Churn"]  # Target (Churn: Yes/No)

# Convert categorical variables to numerical (if needed)
X = pd.get_dummies(X, drop_first=True)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Create an AdaBoost model with Decision Stumps
ada = AdaBoostClassifier(random_state=42)

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

# Make predictions
y_pred = ada.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.7979176526265973
Classification Report:
               precision    recall  f1-score   support

          No       0.82      0.92      0.87      1539
         Yes       0.68      0.48      0.56       574

    accuracy                           0.80      2113
   macro avg       0.75      0.70      0.72      2113
weighted avg       0.79      0.80      0.79      2113
```

`pd.get_dummies(X, drop_first=True)` is a function in the Pandas library that performs one-hot encoding on categorical variables in a DataFrame `X`.

- **Categorical Variables:** It identifies categorical columns within the DataFrame `X`.
- **Creating Dummy Variables:** For each unique category in a categorical column, it creates a new binary column (a "dummy" variable).
- **Binary Representation:** If a row has the specific category in the original column, the corresponding dummy variable gets a value of 1. Otherwise, it gets a value of 0.

`drop_first=True`:

- **Reducing Redundancy:** When you have a categorical variable with *n* categories, you only need *n-1* dummy variables to represent it. The *n*th category can be inferred when all the other dummy variables are 0.

- **Avoiding Multicollinearity:** In statistical models (like linear regression), including all *n* dummy variables can lead to multicollinearity, where independent variables are highly correlated. This can cause problems with model estimation.
- **Dropping the First Category:** `drop_first=True` drops the *first* dummy variable that would have been created. This removes the redundancy and avoids multicollinearity.

Let's say you have a DataFrame `X` with a "color" column:

```
   color
0    red
1   blue
2  green
3    red
```

When you apply `pd.get_dummies(X, drop_first=True)`, the result would be:

```
   color_blue  color_green
0           0            0
1           1            0
2           0            1
3           0            0
```

Is `pd.get_dummies()` Always Necessary?

- **Yes**, if your dataset contains **categorical variables** (text data like "Male" or "Female").
- **No**, if your dataset already contains only numerical data.

Gradient Boosting does not necessarily require `pd.get_dummies` (one-hot encoding)

Select Best Features

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.5],
    'base_estimator__max_depth': [1, 2, 3]
}

grid_search = GridSearchCV(AdaBoostClassifier(), param_grid, cv=5, scoring='a
ccuracy', n_jobs=-1)
grid_search.fit(X_train, y_train)

print(f"\nBest Parameters: {grid_search.best_params_}")
best_ada = grid_search.best_estimator_
test_accuracy = best_ada.score(X_test, y_test)
print(f"Test Accuracy of Best Model: {test_accuracy:.4f}")
```

🔴**NOTE: 👆 This code hanged the system.**

- **Took 13+ min to run.**

```
Best Parameters: {'learning_rate': 0.5, 'n_estimators': 200}
Test Accuracy of Best Model: 0.8008
```

Python Code for AdaBoost Regression

```
# Step 1: Import necessary libraries
import pandas as pd
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Step 2: Load the dataset
from sklearn.datasets import fetch_california_housing
california = fetch_california_housing()
```

```
# Convert the dataset into a pandas DataFrame for better visualization
X = pd.DataFrame(california.data, columns=california.feature_names)  # Feature
s
y = pd.Series(california.target)  # Target (house prices)

# Step 3: Split the data into training and testing sets
# 80% of the data is used for training, and 20% is used for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
=42)

# Step 4: Create an AdaBoostRegressor model
# base_estimator = The weak model (Decision Tree with max_depth=3)
# n_estimators = Number of weak models to train
ada = AdaBoostRegressor(n_estimators=100, learning_rate=0.1,random_state=4
2)

# Step 5: Train the model
ada.fit(X_train, y_train)

# Step 6: Make predictions
y_pred = ada.predict(X_test)

# Step 7: Evaluate the model
# Mean Squared Error (MSE): Lower is better
mse = mean_squared_error(y_test, y_pred)
# R² Score: Closer to 1 is better
r2 = r2_score(y_test, y_pred)

print("Mean Squared Error:", mse)
print("R² Score:", r2)
```

```
Mean Squared Error: 0.568195768061393
R² Score: 0.5663981417281327
```

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.5]
}

grid_search = GridSearchCV(AdaBoostRegressor(), param_grid, cv=5, scoring
='r2', n_jobs=-1)
grid_search.fit(X_train, y_train)

print(f"\nBest Parameters: {grid_search.best_params_}")
best_ada_reg = grid_search.best_estimator_
r2_best = best_ada_reg.score(X_test, y_test)
print(f"Test R² with Best Model: {r2_best:.4f}")
```

```
Best Parameters: {'learning_rate': 0.1, 'n_estimators': 50}
Test R² with Best Model: 0.5684
```

💡 **This dataset gives much better result using GradientBoostingRegressor**

`GradientBoostingRegressor(n_estimators=200, learning_rate=0.1,max_depth=4,random_state=42)`

```
Mean Squared Error: 0.237774390615781
R² Score: 0.8185494799226947
```

Hyperparameters of AdaBoost

Hyperparameter	Default	What It Does	Effect on Model
base_estimator	None (Defaults to DecisionTreeRegressor(max_depth=3))	Defines the weak learner	More complex base learners

Hyperparameter	Default	What It Does	Effect on Model
		(usually a decision tree).	can lead to overfitting.
n_estimators	50	Number of weak learners (iterations of boosting).	More estimators → Better performance but longer training time.
learning_rate	1	Controls how much each weak model contributes to the final prediction.	Lower values prevent overfitting, but require more estimators.

Recommendation: Use `DecisionTreeRegressor(max_depth=1)`