Machine Learning Regression Models

Linear Regression

What is Linear Regression?

Linear Regression is a supervised learning algorithm used for predicting a continuous target variable based on one or more input features. It assumes a linear relationship between the dependent variable (Y) and independent variables (X).

Types of Linear Regression

- **Simple Linear Regression**: Uses one independent variable to predict the target.
- **Multiple Linear Regression**: Uses multiple independent variables.

Mathematical Formulation

1. Simple Linear Regression

Y=b0+b1X+ϵY = b_0 + b_1X + \epsilonY=b0+b1X+ϵ

- YYY = Target variable
- XXX = Independent variable
- b0b_0b0 = Intercept
- b1b_1b1 = Slope of the line
- ϵ\epsilonϵ = Error term

2. Multiple Linear Regression

Y=b0+b1X1+b2X2+...+bnXn+ϵY = b_0 + b_1X_1 + b_2X_2 + ... + b_nX_n + \epsilonY=b0+b1X1+b2X2+...+bnXn+ϵ

- X1,X2,...,XnX_1, X_2, ..., X_nX1,X2,...,Xn are multiple independent variables

Not all Data is Perfectly Linear

Sometimes, data does not follow a strict linear pattern. In such cases, a simple linear model may not fit well, and we might need more flexible models like Polynomial Regression, Ridge, or Lasso.

Example Code: Simple Linear Regression

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression


# Sample dataset

X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)

y = np.array([2, 4, 5, 4, 5])


# Train model

model = LinearRegression()

model.fit(X, y)


# Predictions

y_pred = model.predict(X)
```

Polynomial Regression

Why Polynomial Regression?

When the relationship between the independent and dependent variables is non-linear, Polynomial Regression helps by adding higher-degree terms.

Mathematical Formulation

$Y=b0+b1X+b2X2+b3X3+...+bnXn+\epsilon+\epsilon$

Example Code: Polynomial Regression

```
from sklearn.preprocessing import PolynomialFeatures

from sklearn.pipeline import make_pipeline


# Create polynomial model

poly_model = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())


# Train model

poly_model.fit(X, y)


# Predictions

y_poly_pred = poly_model.predict(X)
```

Polynomial Regression Parameters

```
model_poly = poly_model.named_steps['linearregression']

print("Intercept:", model_poly.intercept_)

print("Coefficients:", model_poly.coef_)
```

Ridge and Lasso Regression

Why Ridge and Lasso?

When there are too many independent variables, a standard linear model may overfit. Ridge and Lasso help prevent overfitting by adding a penalty (regularization) to the model.

1. Ridge Regression

- Adds an L2 penalty (λ∑bi2\lambda \sum b_i^2λ∑bi2) to the coefficients to reduce overfitting.
- **Equation:** Y=b0+b1X1+b2X2+...+bnXn+λ∑bi2+ϵ

Example Code: Ridge Regression

```
from sklearn.linear_model import Ridge

ridge_model = Ridge(alpha=1.0)

ridge_model.fit(X, y)
```

Ridge Regression Parameters

```
print("Intercept:", ridge_model.intercept_)

print("Coefficients:", ridge_model.coef_)
```

2. Lasso Regression

- Adds an L1 penalty (λ∑|bi|\lambda \sum |b_i|λ∑|bi|) to shrink some coefficients to zero, effectively performing feature selection.
- **Equation:** Y=b0+b1X1+b2X2+...+bnXn+λ∑|bi|+ϵ

Example Code: Lasso Regression

```
from sklearn.linear_model import Lasso

lasso_model = Lasso(alpha=0.1)

lasso_model.fit(X, y)
```

Lasso Regression Parameters

```
print("Intercept:", lasso_model.intercept_)

print("Coefficients:", lasso_model.coef_)
```

Model	When to Use
Simple Linear Regression	Use this when you have only **one** independent variable (feature) and the relationship between X and Y is a **straight line**. Example: Predicting salary based on years of experience.
Multiple Linear Regression	Use this when you have **multiple** independent variables and their combined effect determines the output. Example: Predicting house prices based on size, location, and number of rooms.
Polynomial Regression	Use this when the data follows a **curved** pattern instead of a straight line. Example: Predicting a car's fuel efficiency based on speed (as speed increases, fuel efficiency may first improve, then drop).
Ridge Regression	Use this when you have **too many features**, and some of them are highly related to each other (multicollinearity). Ridge helps to **reduce overfitting** by making the model simpler.
Lasso Regression	Use this when you want to **remove unimportant features** and keep only the most useful ones. Lasso helps in **feature selection** by reducing some coefficients to **zero**.

Decision Tree: Regression & Classification

A **Decision Tree** is a supervised learning algorithm used for both **classification** and **regression** problems. It works by splitting data into smaller subsets using a tree-like structure, where each node represents a decision based on a feature.

Decision Tree for Regression (Decision Tree Regressor)

Unlike Decision Tree Classification, **Decision Tree Regression** predicts continuous values. It still works by splitting data into nodes based on conditions, but instead of class labels at the leaves, it predicts numerical values.

🔹 **How It Works:**

1. The dataset is **recursively split** based on the feature that minimizes variance in each step.
2. At each split, the model chooses the best feature and value that results in the least variance within child nodes.
3. The final prediction is the average value of all data points in a given leaf node.

Mathematical Formulation

For a given split, the variance is calculated as:

$$Var(S)=n1i=1∑n(yi−y¯)2$$

- yiy_iyi = actual values in a node
- y¯\bar{y}y¯ = Mean of the values in that node

The split is chosen to minimize the variance in child nodes.

Example Code: Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

import numpy as np


# Sample dataset

X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])

y = np.array([2.5, 4.5, 7.5, 8.5, 10, 15, 18, 22, 26, 32])  # Continuous
values


# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Train model

model = DecisionTreeRegressor(max_depth=3)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Random Forest: Regression & Classification

Random Forest is an **ensemble learning method** that combines multiple decision trees to improve accuracy and reduce overfitting. It is used for both **classification** and **regression** problems.

2. Random Forest for Regression

What is Random Forest Regression?

- Used for **predicting continuous values** (e.g., house prices, temperature, sales).
- Instead of majority voting, it takes the **average prediction** from all decision trees.
- Helps to **reduce variance** and improve prediction stability.

🔹 **How It Works:**

1. Creates multiple **Decision Tree Regressors**, each trained on a random subset of data.
2. Each tree makes a prediction for a new input.
3. The final prediction is the **average** of all tree outputs.

Example Code: Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

import numpy as np


# Sample dataset
```

```
X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])

y = np.array([2.5, 4.5, 7.5, 8.5, 10, 15, 18, 22, 26, 32])  # Continuous values


# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Train model

model = RandomForestRegressor(n_estimators=100, max_depth=3, random_state=42)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Key Differences: Decision Tree vs Random Forest

Feature	Decision Tree	Random Forest
Overfitting	High (prone to overfitting)	Low (reduces overfitting)
Performance	Less accurate	More accurate
Training Speed	Faster	Slower (trains multiple trees)
Interpretability	Easy to interpret	Hard to interpret
Use Case	When you need a quick decision-making model	When accuracy and robustness are more

		important

Support Vector Machine (SVM): Regression & Classification

Support Vector Machine (SVM) is a powerful algorithm used for **both classification and regression** tasks. It works by finding the best decision boundary (hyperplane) that **maximizes the margin** between different classes in classification and **minimizes error** in regression.

2. SVM for Regression (SVR)

What is SVR (Support Vector Regression)?

- Instead of classification, SVR **predicts continuous values**.
- Tries to **fit a hyperplane** within a margin (epsilon ϵ\epsilonϵ) where most data points lie.
- Uses support vectors to **minimize errors while maintaining the margin**.

Mathematical Expression

For regression: $|y-(w \cdot x+b)| \leq \epsilon$

Where:

- ϵ = margin of tolerance (no penalty for predictions within this range).
- SVR tries to **keep most points within the margin** while penalizing those outside.

Example Code: SVM Regressor (SVR)

```
from sklearn.svm import SVR

from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import mean_squared_error

import numpy as np


# Sample dataset

X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])

y = np.array([2.5, 4.5, 7.5, 8.5, 10, 15, 18, 22, 26, 32])  # Continuous
values


# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Train SVR model

model = SVR(kernel='rbf', C=1.0, epsilon=0.1)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Key Difference between: SVM vs SVR :

Features	**SVM (Classification)**	**SVR (Regression)**
Output Type	Discrete classes (e.g., 0 or 1)	Continuous values (e.g., 0.5, 3.7)

Decision Boundary	Separates classes	Fits a margin around the data
Kernel Trick	Used to separate nonlinear data	Used to capture complex patterns
Optimization Goal	Maximizes Margin	Minimizes error within ϵ margin
Use Case	Image recognition, fraud detection, disease classification	House price prediction, stock forecasting, salary estimation

XGBoost: Regression & Classification

XGBoost (Extreme Gradient Boosting) is an advanced machine learning algorithm that builds **decision trees sequentially**, improving the model at each step. It is **fast, scalable, and powerful**, often winning machine learning competitions.

2. XGBoost for Regression

What is XGBoost Regression?

- Used for **continuous outputs** (e.g., house price prediction).
- Works similarly to classification but **minimizes error instead of classification loss**.
- Uses **gradient descent** to correct errors at each boosting step.

Example Code: XGBoost Regressor

```
import xgboost as xgb

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error
```

```
import numpy as np


# Sample dataset

X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])

y = np.array([2.5, 4.5, 7.5, 8.5, 10, 15, 18, 22, 26, 32])  # Continuous
values


# Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Train XGBoost regressor

model = xgb.XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=3)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Parameters of XGBoost Regressor

Parameter	Description
n_estimators	Number of trees (higher = better but risk of overfitting).
learning_rate	Controls how fast the model learns (lower values prevent overfitting).
max_depth	Controls how complex each tree is (higher depth = more patterns captured).

objective	`reg:squarederror` for regression tasks.
subsample	Fraction of training data used for each boosting round (reduces overfitting).

Key Differences: XGBoost Classifier vs. Regressor

Feature	XGBoost Classifier	XGBoost Regressor
Output Type	Categorical (0, 1, 2...)	Continuous (real numbers)
Loss Function	Log Loss, Cross-Entropy	Mean Squared Error (MSE)
Evaluation Metric	Accuracy, F1-score	RMSE, R-squared
Use Case	Fraud detection, disease classification	Stock price prediction, sales forecasting

LightGBM: Regression & Classification

LightGBM (Light Gradient Boosting Machine) is an **optimized gradient boosting framework** designed for **high efficiency, speed, and scalability**. It is **faster than XGBoost** because it uses **leaf-wise splitting** instead of level-wise splitting.

2. LightGBM for Regression

What is LightGBM Regression?

- Used for **continuous outputs** (e.g., house price prediction).
- Works similarly to classification but **minimizes error instead of classification loss**.

- **Faster and more memory-efficient** than XGBoost.

Example Code: LightGBM Regressor

```
import lightgbm as lgb

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

import numpy as np


# Sample dataset

X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])

y = np.array([2.5, 4.5, 7.5, 8.5, 10, 15, 18, 22, 26, 32])  # Continuous
values


# Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Train LightGBM regressor

model = lgb.LGBMRegressor(n_estimators=100, learning_rate=0.1, max_depth=-1)

model.fit(X_train, y_train)


# Predictions

y_pred = model.predict(X_test)
```

Parameters of LightGBM Regressor

Parameter	Description
`n_estimators`	Number of trees (higher = better but risk of overfitting).
`learning_rate`	Controls how fast the model learns (lower values prevent overfitting).
`max_depth`	Controls how complex each tree is (`-1` means no limit).
`num_leaves`	Number of leaves in each tree (higher values capture more patterns but may overfit).
`objective`	`regression` for continuous target values.

Key Differences: LightGBM vs. XGBoost

Feature	LightGBM	XGBoost
Speed	Faster	Slower
Memory Usage	Lower	Higher
Splitting Method	Leaf-wise	Level-wise
Performance on Large Datasets	Better	Good but slower

Categorical Features	Can handle directly	Needs encoding

KNN for Regression

- Predicts values by averaging the k nearest neighbors' outputs.

Example Code: KNN Regressor

```
from sklearn.neighbors import KNeighborsRegressor

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

import numpy as np


# Sample dataset

X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])

y = np.array([2.5, 4.5, 7.5, 8.5, 10, 15, 18, 22, 26, 32])  # Continuous values


# Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Train KNN Regressor

model = KNeighborsRegressor(n_neighbors=3)

model.fit(X_train, y_train)


# Predictions
```

```
y_pred = model.predict(X_test)
```

Parameters of KNN Regressor

Parameter	Description
`n_neighbors`	Number of nearest neighbors.
`weights`	Weighting scheme for neighbors.
`p`	Distance metric (`1` = Manhattan, `2` = Euclidean).