ML algorithms retrospective

1 Multiple Regression Example

Multiple Regression is a statistical technique used to predict the value of one dependent variable based on two or more independent variables. Unlike simple linear regression, which has only one predictor, multiple regression considers multiple factors simultaneously to make predictions.

Suppose we have a dataset with the following details:

Size (sq ft)	Bedrooms	Bathrooms	Price ($)
2000	3	2	300000
1600	2	1	220000
2400	4	3	360000
1416	3	2	232000
3000	4	3	540000

Table 1: House dataset with features and prices

1.1 Formulating the Model

In multiple linear regression, we aim to predict the target variable Y (house price) using the equation:

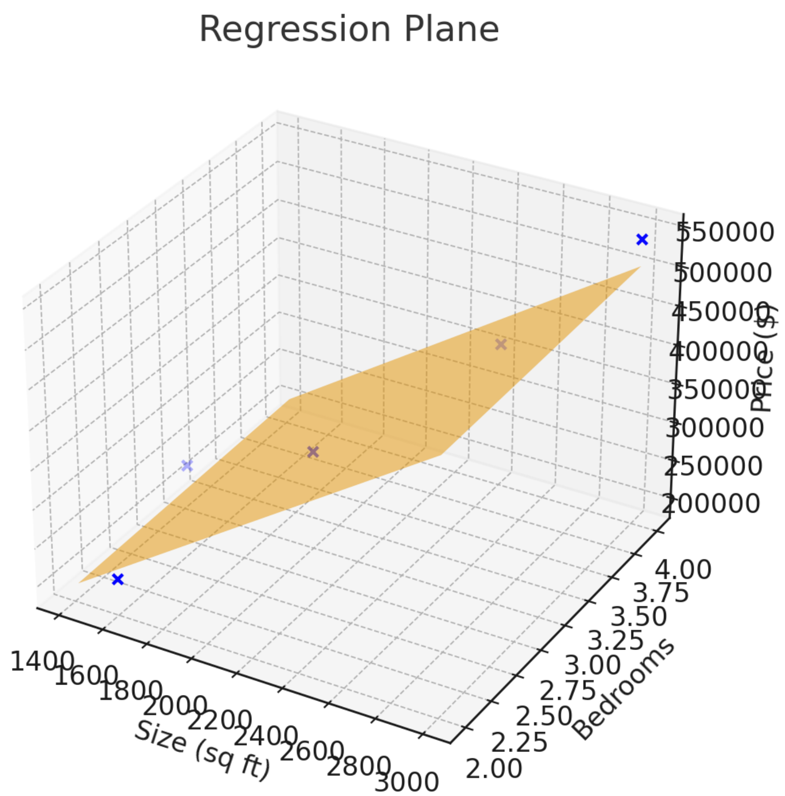
$$Y = b_0 + b_1X_1 + b_2X_2 + b_3X_3 + \ldots + b_nX_n \tag{1}$$

where:

- Y is the target variable (house price),
- X_1, X_2, X_3 are the features (size, bedrooms, bathrooms),
- b_0 is the intercept,
- b_1, b_2, b_3 are the coefficients for each feature.

1.2 Fitting the Model

Using the data, we find the optimal values of b_0, b_1, b_2, and b_3 by minimizing the mean squared error (MSE) between the predicted and actual prices.



The regression plane represents the model's predictions, giving a visual sense of how the features are related to the house price.

The general form for a multiple regression model with two features X_1 and X_2 is:

$$Y = b_0 + b_1X_1 + b_2X_2$$

where:

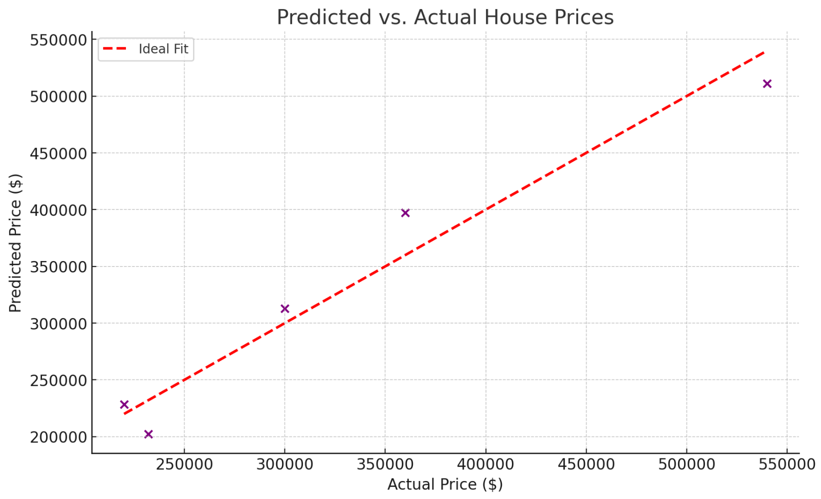
- Y is the target variable (e.g., house price),
- X_1 and X_2 are the features (e.g., house size and number of bedrooms),
- b_0 is the intercept,
- b_1 and b_2 are the coefficients of X_1 and X_2, respectively.

You can calculate the coefficients b_0, b_1, and b_2 with minimizes the sum of squared differences between actual and predicted values. In matrix form, the equation to compute the coefficients is:

$$\mathbf{b} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

where:

- $\mathbf{X}$ is the matrix of feature values, including a column of 1s for the intercept,
- $\mathbf{y}$ is the vector of target values,
- $\mathbf{b}$ is the vector of coefficients $[b_0, b_1, b_2]$.



This scatter plot compares the predicted house prices to the actual prices. The red dashed line represents the ideal line where predicted prices would perfectly match actual prices, helping us evaluate model performance.

1.3 Predicting a New House Price

Once the model is trained, we can predict a new house price given specific values for the features. For example, suppose we have:

- Size (sq ft): 2500
- Bedrooms: 3
- Bathrooms: 2

The predicted house price can be calculated by substituting these values into the regression equation with the calculated coefficients.

1.4 Implementation

Below is a Python implementation using `scikit-learn` to demonstrate how this model can be trained and used to make predictions:

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Example data
X = np.array([
    [2000, 3, 2],
    [1600, 2, 1],
    [2400, 4, 3],
    [1416, 3, 2],
    [3000, 4, 3]
])
y = np.array([300000, 220000, 360000, 232000, 540000])

# Create the model and fit it
model = LinearRegression()
model.fit(X, y)

# Predict for a new house
new_house = np.array([[2500, 3, 2]])
predicted_price = model.predict(new_house)
print("Predicted house price:", predicted_price[0])
```

2 Decision Tree Example

Decision trees are a type of supervised learning algorithm used for classification and regression tasks. Let's consider a classification problem to predict whether a person will buy a certain product based on their age and income level.

Suppose we have the following dataset:

Age (years)	Income Level	Purchase (Yes/No)
25	Low	No
45	High	Yes
35	Medium	Yes
22	Medium	No
60	High	Yes
30	Low	No

Table 2: Dataset for Decision Tree Example

2.1 Building the Decision Tree

A decision tree uses a series of if-then-else conditions to divide the dataset based on feature values, creating branches that lead to a decision (or classification). For this dataset, the algorithm might choose to split first on **Income Level** and then on **Age** to separate instances into groups with similar purchase behaviors.

Gini is a measure of how often a randomly chosen element from a set would be incorrectly classified if it were randomly labeled according to the distribution of labels in the set. It is commonly used in decision trees to evaluate the quality of a split.

$$\text{Gini} = 1 - \sum_{i=1}^{n} p_i^2$$

where:

- n is the number of classes,
- p_i is the proportion of instances of class i in the node.

The Gini impurity is always between 0 and 0.5 for a binary classification. A Gini impurity of 0 means the node is pure (contains instances of only one class), while a Gini impurity closer to 0.5 indicates a more mixed node.

Suppose a node has 10 samples, with the following distribution:

- 6 samples are in class "Yes"
- 4 samples are in class "No"

$$\text{Proportion of "Yes" } (p_{\text{yes}}) = \frac{6}{10} = 0.6$$
$$\text{Proportion of "No" } (p_{\text{no}}) = \frac{4}{10} = 0.4$$

Substitute these values into the formula:

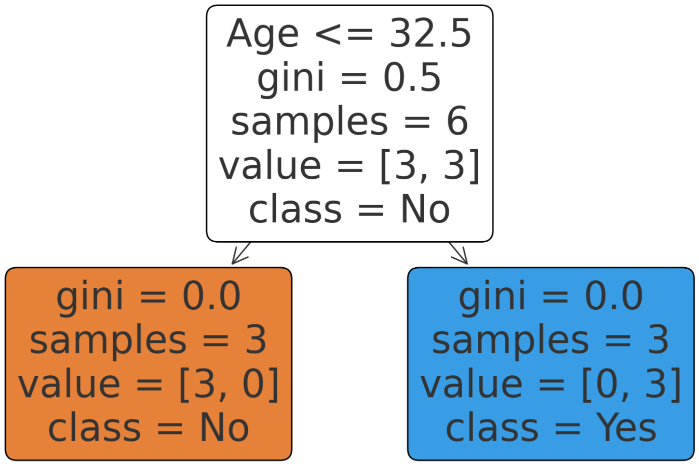
$$\text{Gini} = 1 - (p_{\text{yes}}^2 + p_{\text{no}}^2)$$

$$\text{Gini} = 1 - (0.6^2 + 0.4^2)$$
$$\text{Gini} = 1 - (0.36 + 0.16)$$
$$\text{Gini} = 1 - 0.52 = 0.48$$

The Gini impurity of this node is 0.48, which indicates that it is somewhat mixed but leans towards class "Yes".

This weighted average is then used to determine if the split reduces the overall impurity of the tree. The tree algorithm will typically choose the split that minimizes the Gini impurity.

Decision Tree Example with Age and Income Level



Here's a decision tree using Age and Income Level to predict Purchase. The tree splits the data based on the Age feature, leading to pure nodes where each branch corresponds to a decision about whether the person is likely to make a purchase.

2.2 Predicting a New Outcome

Once the tree is built, it can be used to predict whether a new individual will make a purchase. For example:

- **Age**: 40
- **Income Level**: Medium

Using the rules established by the decision tree, we follow the branches based on the conditions met by the individual's age and income level to reach a prediction.

2.3 Implementation

Below is a Python implementation using `scikit-learn` to demonstrate how a decision tree can be trained and used for predictions:

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Example data
X = np.array([
    [25, 1],  # Low income
    [45, 3],  # High income
    [35, 2],  # Medium income
    [22, 2],  # Medium income
    [60, 3],  # High income
    [30, 1]   # Low income
])
y = np.array([0, 1, 1, 0, 1, 0])  # 0 = No, 1 = Yes

# Create the model and fit it
model = DecisionTreeClassifier()
model.fit(X, y)

# Predict for a new individual
new_person = np.array([[40, 2]])  # Age 40, Medium income
prediction = model.predict(new_person)
print("Purchase prediction:", "Yes" if prediction[0] == 1 else "No")
```

This code will output a purchase prediction based on the input features, demonstrating the application of a decision tree algorithm.

3 Random Forests Example

Random Forests is an ensemble learning method commonly used for classification and regression tasks. It builds multiple decision trees and merges them to improve accuracy and prevent overfitting.

Suppose we have the same house dataset, but we now want to predict house prices using Random Forest regression.

Size (sq ft)	Bedrooms	Bathrooms	Price ($)
2000	3	2	300,000
1600	2	1	220,000
2400	4	3	360,000
1416	3	2	232,000
3000	4	3	540,000

Table 3: House dataset with features and prices

3.1 How Random Forests Work

Random Forests build multiple decision trees on random subsets of data and average their predictions to make the final prediction. For regression, this means averaging the predictions of all trees in the forest.

3.2 Training the Model

Each decision tree in the Random Forest is trained on a random subset of the dataset. This technique, known as bootstrap aggregation or "bagging," helps reduce variance and improve model robustness.

In a Random Forest, each decision tree is trained on a random subset (sample) of the data with replacement, a technique known as Bootstrap Aggregating (Bagging). Bagging helps reduce the variance of the model and improves its robustness by combining the predictions of multiple trees.

For a given input x, each decision tree T_i in the Random Forest makes its own prediction:

$$\hat{y}_i = T_i(x)$$

where:

- $T_i(x)$ is the prediction from the i-th decision tree.

The Random Forest aggregates predictions from all trees by averaging them. If there are M trees in the forest, the overall prediction $\hat{y}_{\text{RF}}$ for an input x is:

$$\hat{y}_{\text{RF}} = \frac{1}{M} \sum_{i=1}^{M} \hat{y}_i$$

where:

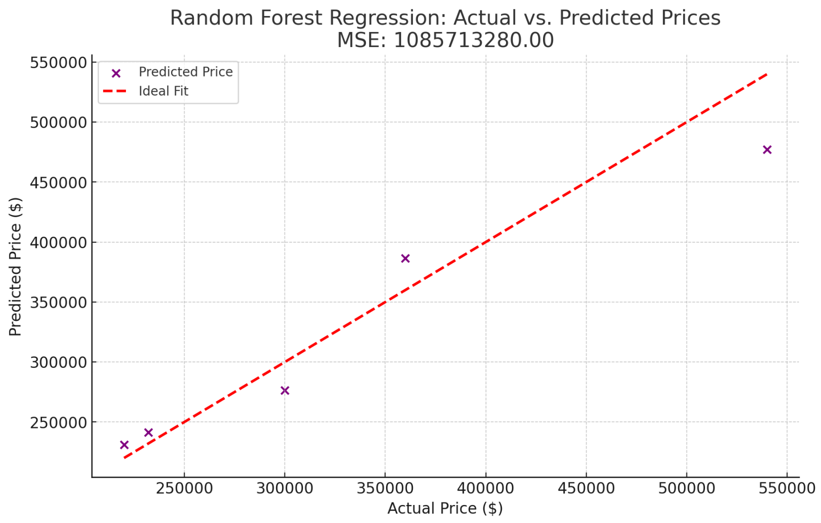
- M is the total number of trees in the forest,
- $\hat{y}_i$ is the prediction from the i-th tree.

This averaging helps reduce the variance of the predictions and improves the model's robustness and accuracy.

Each tree is trained on a random subset of the data. For each tree, samples are selected with replacement, meaning that some data points may be chosen multiple times while others are left out. The probability that a particular sample is not chosen in a random subset is approximately $1 - \frac{1}{n}$, and with n repetitions, the fraction of samples not chosen approaches:

$$\lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n \approx \frac{1}{e} \approx 0.368$$

This indicates that about 36.8% of the data is left out for each tree, a set known as the Out-Of-Bag samples.



3.3 Predicting a New House Price

After training, we can predict the house price by inputting features such as:

- Size (sq ft): 2500
- Bedrooms: 3
- Bathrooms: 2

The Random Forest model will generate predictions from each tree, and the average of these predictions will be the final predicted price.

3.4 Implementation

Below is a Python implementation using `scikit-learn` to demonstrate Random Forest regression:

```
from sklearn.ensemble import RandomForestRegressor
import numpy as np

# Example data
X = np.array([
    [2000, 3, 2],
    [1600, 2, 1],
    [2400, 4, 3],
    [1416, 3, 2],
    [3000, 4, 3]
])
y = np.array([300000, 220000, 360000, 232000, 540000])
```

```
# Create the model and fit it
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X, y)

# Predict for a new house
new_house = np.array([[2500, 3, 2]])
predicted_price = model.predict(new_house)
print("Predicted house price:", predicted_price[0])
```

This Python code demonstrates how Random Forests can be applied for regression tasks, showing an alternative to multiple regression with potentially improved performance.

4 K-Nearest Neighbors (KNN) Example

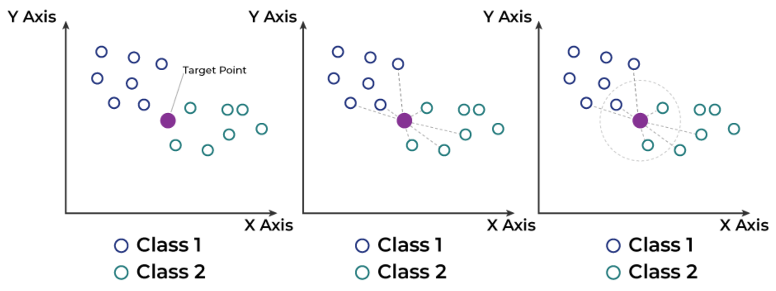
In this section, we demonstrate an example of the K-Nearest Neighbors (KNN) algorithm, a simple yet powerful classification method. Suppose we have a dataset of points with two features:

Feature 1	Feature 2	Class
2.0	3.0	A
1.0	1.0	A
2.5	3.5	B
3.0	4.0	B
5.0	6.0	A

Table 4: Sample dataset for KNN with features and classes

4.1 Algorithm Overview

The KNN algorithm classifies a data point based on the majority class among its k nearest neighbors. We calculate the distance (e.g., Euclidean) between points to determine the neighbors.



- **Left Panel**: The dataset consists of points belonging to two classes:
  - **Class 1** (shown as blue circles)
  - **Class 2** (shown as teal circles)

  The **target point** (purple) is the point we want to classify. The KNN algorithm will classify this point based on the majority class among its k nearest neighbors.

- **Middle Panel (KNN with $k = 3$)**: In this example, the KNN algorithm uses **3 neighbors** to classify the target point. The algorithm calculates the distance (typically Euclidean) between the target point and all other points in the dataset. The three closest points (neighbors) are identified, as shown by the dashed lines connecting the target point to its 3 nearest neighbors. The majority class among these three neighbors is **Class 2** (teal), so the target point is classified as **Class 2**.

- **Right Panel (KNN with $k = 5$)**: Here, the KNN algorithm uses **5 neighbors** to classify the target point. Again, the algorithm calculates distances and finds the 5 nearest neighbors. The 5 nearest neighbors include points from both **Class 1** and **Class 2**. However, the majority of these neighbors belong to **Class 1**. Therefore, the target point is classified as **Class 1** when $k = 5$.

Key Points Illustrated in the Image

- **Distance Calculation**: The KNN algorithm relies on distance metrics (e.g., Euclidean distance) to identify the nearest neighbors.

- **Effect of k**: The classification of the target point can change depending on the value of k. In this example, using $k = 3$ results in a **Class 2** classification, while $k = 5$ leads to a **Class 1** classification.

- **Majority Voting**: The final classification is based on the majority class among the k nearest neighbors, showing how KNN uses local information to make decisions.

This illustration effectively demonstrates the impact of the choice of k in the KNN algorithm and how it affects the decision for classifying new points.

4.2 Example Calculation

Suppose we want to classify a new point with the following features:

- Feature 1: 3.5

- Feature 2: 3.5

1. **Calculate the Distance**: Compute the distance from this new point to each point in the dataset. 2. **Select Nearest Neighbors**: Choose the k points closest to the new point. For instance, if $k = 3$, select the 3 closest neighbors. 3. **Classify Based on Majority**: Assign the class that appears most frequently among the k neighbors to the new point.

4.3 Implementation

Below is a Python implementation using `scikit-learn` to demonstrate the KNN algorithm:

```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np

# Example data
X = np.array([
    [2.0, 3.0],
    [1.0, 1.0],
    [2.5, 3.5],
    [3.0, 4.0],
    [5.0, 6.0]
])
y = np.array(['A', 'A', 'B', 'B', 'A'])

# Create the model and set k=3
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X, y)

# Predict for a new point
new_point = np.array([[3.5, 3.5]])
predicted_class = model.predict(new_point)
print("Predicted class:", predicted_class[0])
```

This code demonstrates the KNN classification for a new point based on its nearest neighbors in the dataset.