Machine Learning Algorithms - Supervised Algorithms

Hao Le Vu

IUH

2025

Outline

1. Data Preprocessing
   - Feature Scaling
2. Supervised Learning
   - Regression

Feature Scaling: Why It Matters

- Many machine learning algorithms (e.g., **Gradient Descent**, **KNN**, **SVM**) are sensitive to the **magnitude of features**.
- If features are on different scales, the model may:
  - Converge slowly or fail to converge.
  - Give higher weight to features with larger ranges.
  - Produce biased decision boundaries.
- Example:
  - Height in **cm**: 150–200
  - Weight in **kg**: 40–90
  - Without scaling, distance-based algorithms like KNN are dominated by height.

Goal

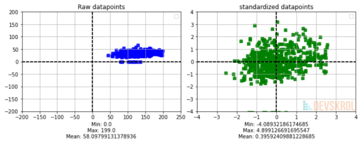
Transform features so they are **comparable** and have similar ranges.

Standardization (Z-score Normalization)

Definition

$$z = \frac{x - \mu}{\sigma}$$

- Transforms features so that the **mean** $= 0$ and **standard deviation** $= 1$.
- Removes units and centers data for algorithms sensitive to feature magnitude.
- Commonly used in:
  - **Logistic Regression**
  - **Support Vector Machines (SVM)**
  - **K-Means Clustering**



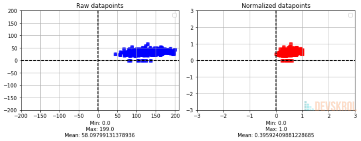
Data after standardization is centered at 0 with unit variance.

Min-Max Normalization

Definition

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

- Scales features to a fixed range, usually $[0, 1]$ or $[-1, 1]$.
- Preserves relationships between data points.
- Commonly used in:
  - **Neural Networks**
  - **Gradient Descent optimization**
  - Image pixel normalization (0-255 $\rightarrow$ 0-1)



Data is rescaled into the range [0, 1].

Python Example: Feature Scaling

```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import numpy as np

# Example data: [Height (cm), Weight (kg)]
data = np.array([
    [160, 55],
    [170, 80],
    [180, 72],
    [190, 95]
])

# Standardization
standard_scaler = StandardScaler()
standardized_data = standard_scaler.fit_transform(data)
print("Standardized Data:\n", standardized_data)

# Min-Max Normalization
minmax_scaler = MinMaxScaler()
normalized_data = minmax_scaler.fit_transform(data)
print("Normalized Data:\n", normalized_data)
```

Linear Regression

Model

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + \epsilon$$

- Predicts continuous outcomes (e.g., housing prices).
- Uses gradient descent to minimize Mean Squared Error (MSE):

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Python Example: Linear Regression

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

# Example data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([2, 4, 6, 8, 10])

# Train model
model = LinearRegression()
model.fit(X, y)

# Predict
pred = model.predict([[6]])
print(f"Prediction for input 6: {pred}")
```

Decision Tree Regression: Concept

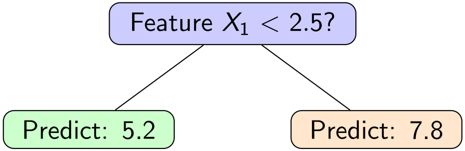
Definition

- A **non-linear regression algorithm** that predicts continuous values.
- Splits the dataset into regions and fits a constant value in each region.
- Works by recursively partitioning the feature space.

Key Idea

- Each split minimizes the variance (or MSE) of target values in the child nodes.
- Final prediction = average value of the training samples in a leaf node.

Decision Tree Regression Visualization



Each leaf predicts a continuous value, the mean of its region's samples.

Splitting Criterion for Regression Trees

Mean Squared Error (MSE)

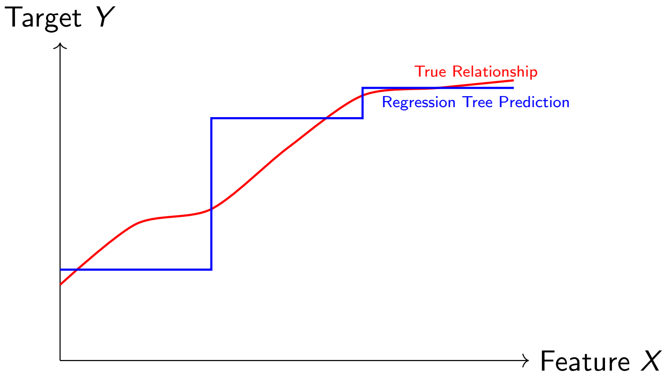
$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y})^2$$

- y_i = actual target value
- $\hat{y}$ = predicted target value (mean of samples in node)

Goal of Splitting

- Choose the split that **minimizes the weighted average MSE** of child nodes.
- This ensures each child node has more homogeneous target values.

MSE and Prediction Fit



Regression trees approximate continuous data using stepwise constant predictions.

Overfitting and Pruning

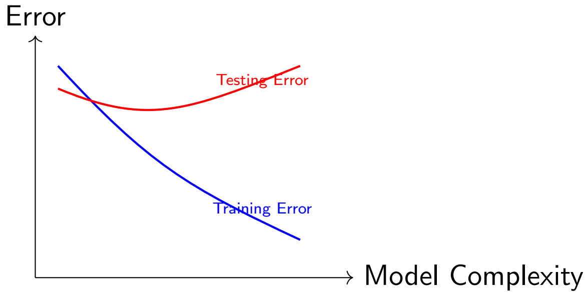
Overfitting Issue

- Deep trees may perfectly fit training data but fail to generalize.
- Leads to high variance and poor performance on unseen data.

Regularization Techniques

- **max_depth** – Limit depth of the tree.
- **min_samples_split** – Minimum samples required to split a node.
- **min_samples_leaf** – Minimum samples required in a leaf.
- **Pruning** – Build full tree, then remove unimportant branches.

Overfitting Visualization



Increasing complexity decreases training error but may cause overfitting.

Decision Tree Regression: Pros and Cons

Advantages

- Easy to interpret and visualize.
- Non-linear relationships handled well.
- Requires little data preprocessing.

Limitations

- Prone to overfitting without regularization.
- Predictions are not smooth (stepwise).
- Small data changes can lead to different trees.

Python Example: Decision Tree Regression

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import numpy as np

# Example dataset
X = np.array([[1], [2], [3], [4], [5], [6]])
y = np.array([1.2, 1.9, 3.0, 3.5, 3.6, 3.7])

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_sta

# Train model
model = DecisionTreeRegressor(max_depth=3, random_state=42)
model.fit(X_train, y_train)

# Evaluate
y_pred = model.predict(X_test)
print("MSE:", mean_squared_error(y_test, y_pred))
```

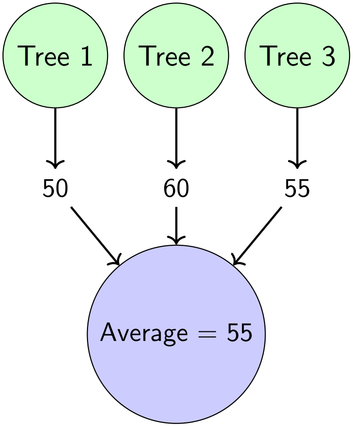
Random Forest Regression: Concept

What is Random Forest Regression?

- An **ensemble learning method** that combines multiple Decision Trees to improve accuracy and robustness.
- Each tree predicts a target value, and the forest prediction is the **average of all trees**.
- Helps reduce overfitting compared to a single Decision Tree.

Key Idea

- Build many independent trees using random subsets of data and features.
- Combine their outputs to get a more stable and generalizable prediction.



Combine tree outputs by averaging for final prediction.

How Random Forest Regression Works

Training Process

1. Draw **random bootstrap samples** from the training dataset.
2. For each tree:
   - Select a random subset of features.
   - Build a Decision Tree using this subset.
3. Repeat to create T trees.

Prediction Formula

For a given input x, the forest prediction is the average of predictions from all trees:

$$\hat{y} = \frac{1}{T} \sum_{t=1}^{T} h_t(x)$$

where:

- T = number of trees in the forest,
- $h_t(x)$ = prediction from the t^{th} decision tree.

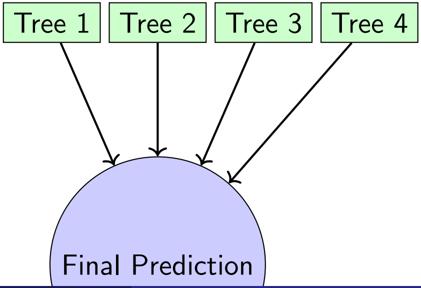
Random Forest: Strengths and Weaknesses

Advantages

- Reduces overfitting compared to a single Decision Tree.
- Works well with high-dimensional datasets.
- Handles both continuous and categorical variables.
- Robust to outliers and noise in the dataset.

Limitations

- Can be computationally expensive with many trees.
- Less interpretable than a single tree.
- May still overfit if the number of trees or depth is not properly tuned.



Python Example: Random Forest Regression

```python
from sklearn.ensemble import RandomForestRegressor
import numpy as np
import matplotlib.pyplot as plt

# Example data
X = np.array([[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]])
y = np.array([2, 4, 5, 7, 8, 8, 10, 12, 12, 13])

# Train random forest regressor
model = RandomForestRegressor(n_estimators=100, max_depth=3, random_state=0)
model.fit(X, y)

# Predict
X_test = np.linspace(0, 10, 100).reshape(-1, 1)
y_pred = model.predict(X_test)

# Plot results
plt.scatter(X, y, color='red', label='Data')
plt.plot(X_test, y_pred, color='blue', label='Random Forest Prediction')
plt.legend()
plt.show()
```