ML Models: Theory and Implementation

Lenny Pelhate

April 2025

Contents

1 Regression

1.1 Linear Regression

Linear Regression models the relationship between a scalar dependent variable y (the target) and one or more variables $x_1, x_2, \ldots, x_n$ (the features), where the observations are assumed to be independent, and the features are assumed to not be multicollinear.

The model assumes a linear relationship between the input features and the target variable, along with additive noise.

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\varepsilon}$$

1.1.1 Hypotheses

Linear regression assumes the following:

1. **Linearity**: The relationship between input features and the target is linear (and additive).
2. **Independence of Observations**: Observations are independent of each other.
3. **No multicollinearity**: Features are not highly correlated with each other.
4. **Homoscedasticity**: Constant variance of the errors.
5. **Normality**: The residuals (errors) are normally distributed.
6. **Independence of Errors**: The residuals (errors) are independent of each other.
7. **No Autocorrelation of Errors**: The residuals (errors) should not exhibit a pattern or correlation, especially in time series data.

1.1.2 Mathematical Formulation

- $\mathbf{X} \in \mathbb{R}^{m \times n}$: **Input feature matrix**, where m is the number of samples (observations) and n is the number of features (with a column of ones for the bias term)
- $\mathbf{y} \in \mathbb{R}^m$: **Target vector** (the true output values)
- $\boldsymbol{\theta} \in \mathbb{R}^n$: **Parameter vector** (including the bias term)
- $\boldsymbol{\varepsilon} \in \mathbb{R}^m$: **Residual (error) vector**

Each input sample can be represented by a feature vector:

$$\mathbf{x}^{(i)} = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{i(n-1)} \end{bmatrix} \in \mathbb{R}^n \quad \text{for } i = 1, 2, \ldots, m$$

where the first element is the bias term (1), and the remaining elements are the feature values for the i-th observation. The input feature matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ can be written as:

$$\mathbf{X} = \begin{bmatrix} \left(\mathbf{x}^{(1)}\right)^\top \\ \left(\mathbf{x}^{(2)}\right)^\top \\ \vdots \\ \left(\mathbf{x}^{(m)}\right)^\top \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1(n-1)} \\ 1 & x_{21} & x_{22} & \cdots & x_{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{m(n-1)} \end{bmatrix}$$

Assumptions:

- **Independence of Observations:** This refers to the relationship between rows (samples) in the dataset. The assumption of independence means that the values of one observation do not provide any information about the values of another observation. Mathematically, we can express this as:

  $$\forall 1 \leq i, j \leq m, \quad \mathbf{x}^{(i)} \perp\!\!\!\perp \mathbf{x}^{(j)}$$

  where $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ represent two different feature vectors corresponding to the i-th and j-th observations.

- **No Multicollinearity:** The features (columns of $\mathbf{X}$) are assumed to be linearly independent. Multicollinearity occurs when one feature is a linear combination of others, leading to issues in matrix inversion. Formally, we require that:

  $$\text{rank}(\mathbf{X}^\top \mathbf{X}) = n \quad \text{or equivalently,} \quad \mathbf{X}^\top \mathbf{X} \text{ is invertible}$$

  This ensures the uniqueness of the solution.

The target vector $\mathbf{y}$ is given by:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

where y_i is the target value for the i-th sample.

The parameter vector $\boldsymbol{\theta}$ is:

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n-1} \end{bmatrix}$$

where θ_0 is the bias term and $\theta_1, \theta_2, \ldots, \theta_{n-1}$ are the weights for each feature.

The residual (error) vector $\boldsymbol{\varepsilon}$ is:

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_m \end{bmatrix}$$

where each $\varepsilon_i = y_i - \mathbf{x}_i^\top \boldsymbol{\theta}$ represents the difference between the true value y_i and the predicted value $\mathbf{x}_i^\top \boldsymbol{\theta}$ for the i-th observation.

1.1.3 Model Prediction

We model the relationship between inputs and outputs using a linear function with an additive noise term $\boldsymbol{\varepsilon} \in \mathbb{R}^m$:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\varepsilon}$$

The model prediction $\hat{\mathbf{y}}$ (predicted target vector) is given by the matrix product:

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta} \in \mathbb{R}^m$$

1.1.4 Loss Function to Minimize

Perspective 1: Maximum Likelihood Estimation (MLE)

- **Gaussian Likelihood of a Single Observation:**

  $$y^{(i)} \mid \mathbf{x}^{(i)}, \boldsymbol{\theta} \sim \mathcal{N}(\mathbf{x}^{(i)\top}\boldsymbol{\theta}, \sigma^2)$$

  where $\mathbf{x}^{(i)}$ is the corresponding input feature vector for the i-th observation.

- **Likelihood Function:** Given the Gaussian noise assumption, the probability (likelihood) of observing $\mathbf{y}$ given $\mathbf{X}$ and $\boldsymbol{\theta}$ is:

  $$\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta} \sim \mathcal{N}(\mathbf{X}\boldsymbol{\theta}, \sigma^2\mathbf{I})$$

  $$p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y^{(i)} - \mathbf{x}^{(i)\top}\boldsymbol{\theta})^2\right)$$

- **Log-Likelihood:** Taking the log of the likelihood function (to simplify optimization), we get the log-likelihood:

  $$\log p(\mathbf{y} \mid \mathbf{X}, \boldsymbol{\theta}) = -\frac{m}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{m}(y^{(i)} - \mathbf{x}^{(i)\top}\boldsymbol{\theta})^2$$

- **Maximum Likelihood Estimation (MLE):** Maximizing the log-likelihood is equivalent to minimizing the sum of squared errors:

  $$\boldsymbol{\theta}_{\text{MLE}} = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)\top}\boldsymbol{\theta} - y^{(i)}\right)^2$$

- **Mean Squared Error (MSE) Loss:** To simplify computation and make the loss function scale-invariant, we define the Mean Squared Error cost function:

  $$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)\top}\boldsymbol{\theta} - y^{(i)}\right)^2 = \frac{1}{2m}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2$$

  The factor $\frac{1}{2m}$ is included for convenience in gradient derivations, as it cancels out the 2 in the derivative of the squared term, and $\frac{1}{m}$ normalizes the loss per sample.

- **Optimization Problem:**

  $$\boldsymbol{\theta}_{\text{MLE}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \frac{1}{2m}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2$$

Perspective 2: Ordinary Least Squares (OLS)

- **Minimize Residual Sum of Squares (RSS) also called the Sum of Squared Errors (SSE):** Find the parameter vector $\boldsymbol{\theta}$ that minimizes the sum of squared residuals (i.e., the squared ℓ_2 norm of the error vector):

$$\|\boldsymbol{\varepsilon}\|^2 = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2$$

- **Optimization Problem:**

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2$$

Note: These two problems (MLE and RSS) are equivalent.

1.1.5 Analytical Solution (Normal Equation)

Given the problem:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \frac{1}{2m}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2$$

Expanding the squared term:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\left(\mathbf{y}^\top\mathbf{y} - \mathbf{y}^\top\mathbf{X}\boldsymbol{\theta} - (\mathbf{X}\boldsymbol{\theta})^\top\mathbf{y} + \boldsymbol{\theta}^\top\mathbf{X}^\top\mathbf{X}\boldsymbol{\theta}\right)$$

Notice that $(\mathbf{X}\boldsymbol{\theta})^\top\mathbf{y}$ is a scalar, and its transpose is itself, i.e., $(\mathbf{X}\boldsymbol{\theta})^\top\mathbf{y} = \mathbf{y}^\top\mathbf{X}\boldsymbol{\theta}$. Thus, the expression becomes:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\left(\mathbf{y}^\top\mathbf{y} - 2\mathbf{y}^\top\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^\top\mathbf{X}^\top\mathbf{X}\boldsymbol{\theta}\right)$$

The cost function $J(\boldsymbol{\theta})$ is convex and differentiable:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\left(\mathbf{X}^\top\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^\top\mathbf{y}\right)$$

Setting the gradient to zero yields:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^*) = \frac{1}{m}\mathbf{X}^\top(\mathbf{X}\boldsymbol{\theta}^* - \mathbf{y}) = 0$$

Solving for $\boldsymbol{\theta}^*$:

$$\boldsymbol{\theta}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

This is known as the **normal equation**.

1.1.6 Approaches to Address Multicollinearity

Multicollinearity occurs when two or more independent variables in a regression model are highly linearly correlated. This means that one variable can be (approximately) predicted from the others.

Problem of Mulicollinearity:

- **Unstable coefficients**: Small changes in data can lead to large swings in coefficient estimates.
- **Interpretation difficulty**: The individual effect of each predictor becomes hard to isolate.
- **Overfitting risk**: Especially problematic in small datasets.
- **Computational issues**: The matrix $\mathbf{X}^\top\mathbf{X}$ in the normal equation

  $$\boldsymbol{\theta} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

  becomes nearly singular (non-invertible), leading to numerical instability.

Solutions:

- Remove one of the correlated variables.
- Combine correlated features.
- Apply regularization:
  - **Ridge Regression (L2)**: Penalizes large coefficients, reducing variance.
  - **Lasso Regression (L1)**: Can drive some coefficients to zero.
- Use Principal Component Analysis (PCA) to transform variables into uncorrelated components.

1.1.7 Regularization

To prevent overfitting or deal with multicollinearity, regularization techniques add a penalty term to the cost function.

L2 Regularization (Ridge Regression)

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2 + \frac{\lambda}{2m}\|\boldsymbol{\theta}\|^2$$

Solution:

$$\boldsymbol{\theta}_{\text{ridge}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$$

L1 Regularization (Lasso Regression)

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|^2 + \frac{\lambda}{m}\|\boldsymbol{\theta}\|_1$$

Unlike Ridge, Lasso does not have a closed-form solution because the L_1 norm is not differentiable. Instead, it is solved using optimization techniques like coordinate descent or gradient-based methods.

1.1.8 Implementation:

Linear Regression Implementation in Python

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Generate some synthetic data for illustration
# X: feature matrix, y: target variable
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.normal(0, 1, (100, 1))

# Create and train the linear regression model
lin_reg = LinearRegression()
lin_reg.fit(X, y)

# Make predictions
y_pred = lin_reg.predict(X)

# Output the model parameters
print("Intercept:", lin_reg.intercept_)
print("Coefficient:", lin_reg.coef_)

# Plot the results
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X, y_pred, color='red', label='Linear regression line')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

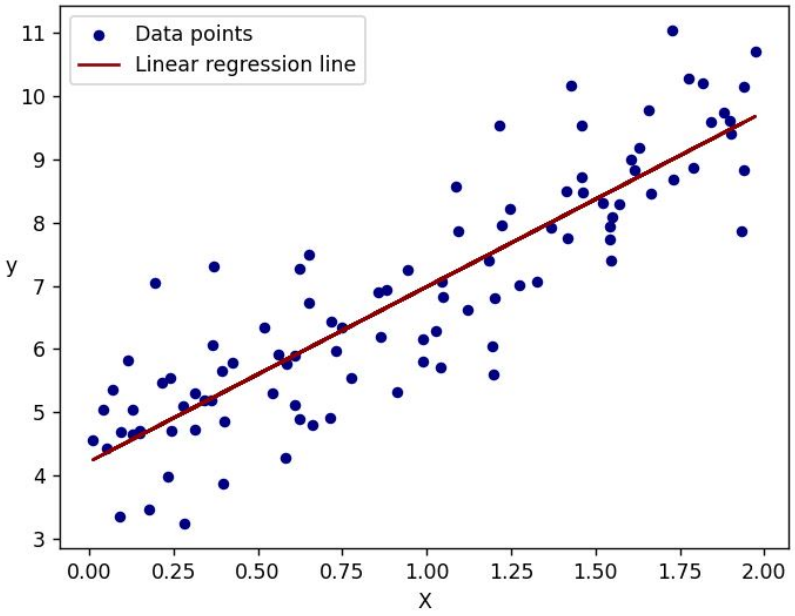

Figure 1: Example of Linear Regression

1.2 Logistic Regression

Logistic Regression is a statistical method used for binary **classification** problems, though it can also be extended to multiclass classification using techniques like One-vs-All (OvA) or Softmax Regression. The goal of logistic regression is to model the probability that a given input belongs to a particular class.

Note: Despite being called "regression", it's a classification model.

1.2.1 Hypotheses

Below are the key hypotheses made in the context of both binary and multiclass logistic regression:

- **Linearity in the Log-Odds:** Logistic regression assumes that the log-odds (logarithm of the odds) of the outcome is linearly related to the predictor variables. This can be expressed as:

$$\log\left(\frac{P(y=1 \mid \mathbf{x})}{1-P(y=1 \mid \mathbf{x})}\right) = \mathbf{x}^\top \boldsymbol{\theta}$$

  where $P(y=1 \mid \mathbf{x})$ is the probability of the outcome being 1 given the input features $\mathbf{x}$, and $\boldsymbol{\theta}$ is the parameter vector.

- **Independence of Observations:** Logistic regression assumes that the observations are independent of each other. This means that the value of one observation should not provide any information about another observation.

- **No Multicollinearity:** The features in the model should not be highly correlated with each other. This assumption ensures that each feature provides unique and independent information about the outcome.

- **Homoskedasticity**: Logistic regression assumes that the error variance is consistent across all levels of the independent variables.
- **No Outliers:** Logistic regression assumes that there are no extreme outliers in the data.
- **Sufficient Sample Size:** Logistic regression requires a sufficiently large sample size to ensure reliable parameter estimates. Small datasets can lead to overfitting and unreliable results.
- **Independence of Errors:** The residuals (errors) in logistic regression should be independent of each other. This assumption is important for ensuring the reliability of statistical tests.
- **Multiclass Hypothesis (for Multiclass Logistic Regression):** In multiclass logistic regression, the assumption is that the log-odds for each class relative to the baseline class are linearly related to the features.

1.2.2 Binary Logistic Regression

In binary logistic regression, the target variable y takes one of two values: 0 or 1. The model estimates the probability that the target y is 1, given the input features $\mathbf{x}$.

The key idea behind logistic regression is to apply a **logistic function** (also known as the **sigmoid function**) to the linear combination of the input features. This maps the output of the linear model to a probability in the range $[0, 1]$.

Mathematical Formulation:

The model's prediction can be represented as:

$$P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{x}^\top \boldsymbol{\theta})$$

where $\sigma(z)$ is the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

This function outputs a value between 0 and 1, representing the probability that the instance belongs to class 1. The decision rule is then to classify the input as class 1 if $P(y = 1 \mid \mathbf{x}) \geq 0.5$, and class 0 otherwise.

Loss Function:

For binary classification, the cost function is the **logistic loss function** (or **binary cross-entropy**):

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log P(y^{(i)} = 1 \mid \mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - P(y^{(i)} = 1 \mid \mathbf{x}^{(i)})) \right]$$

where m is the number of training examples, and $y^{(i)}$ is the true label for the i-th sample.

Optimization Problem:

The goal of logistic regression is to find the parameter vector $\boldsymbol{\theta}$ that minimizes the cost function. This is typically done using optimization algorithms like **Gradient Descent** or **Newton's Method**.

The gradient of the cost function with respect to $\boldsymbol{\theta}$ for binary logistic regression is:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma(\mathbf{x}^{(i)\top} \boldsymbol{\theta}) - y^{(i)} \right) \mathbf{x}^{(i)}$$

Once the gradient is computed, the parameters are updated iteratively using gradient descent:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

where α is the learning rate.

1.2.3 Multiclass Logistic Regression

When the target variable y can take more than two values (i.e., a multiclass classification problem), logistic regression can be extended using the **softmax function**.

This is known as **multinomial logistic regression** or **softmax regression**.

Mathematical Formulation:

For a multiclass problem with K classes, the probability that the target variable y belongs to class k is given by:

$$P(y = k \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \boldsymbol{\theta}_k}}{\sum_{j=1}^{K} e^{\mathbf{x}^\top \boldsymbol{\theta}_j}}$$

where $\boldsymbol{\theta}_k$ is the parameter vector for class k, and the denominator ensures that the probabilities sum to 1.

In this case, the model estimates the probability for each class, and the class with the highest probability is chosen as the predicted class.

Loss Function:

For multiclass classification, the cost function is the **categorical cross-entropy**, which generalizes the binary case:

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log P(y^{(i)} = k \mid \mathbf{x}^{(i)})$$

where $y_k^{(i)}$ is an indicator that is 1 if the i-th example belongs to class k, and 0 otherwise.

Optimization Problem:

The goal of logistic regression is to find the parameter vector $\boldsymbol{\theta}$ that minimizes the cost function. This is typically done using optimization algorithms like **Gradient Descent** or **Newton's Method**.

For multiclass logistic regression, the gradient for class k is:

$$\nabla_{\boldsymbol{\theta}_k} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(P(y^{(i)} = k \mid \mathbf{x}^{(i)}) - y_k^{(i)} \right) \mathbf{x}^{(i)}$$

Once the gradient is computed, the parameters are updated iteratively using gradient descent:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

where α is the learning rate.

1.2.4 Summary of Key Differences from Linear Regression

While linear regression is used for continuous prediction, logistic regression is used for classification tasks. Here are the main differences:

- In linear regression, the output is continuous, whereas in logistic regression, the output is a probability between 0 and 1 (for binary classification).
- The model in logistic regression uses a sigmoid (or softmax for multiclass) to model probabilities, while linear regression directly models the relationship between input features and the target.
- The cost function in linear regression is the Mean Squared Error (MSE), while logistic regression uses cross-entropy loss (binary or categorical).

Link from Logistic Regression to Linear Regression via the Inverse Sigmoid:

Logistic regression models the probability that a binary target variable $y \in \{0, 1\}$ belongs to the positive class, given a feature vector $\mathbf{x}$. It does this using the sigmoid function:

$$P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}$$

Here, $\mathbf{w}^\top \mathbf{x}$ is a linear combination of the input features, just as in linear regression.

To understand the link to linear regression, we apply the inverse of the sigmoid function to both sides:

$$\sigma^{-1}(P(y = 1 \mid \mathbf{x})) = \log \left(\frac{P(y = 1 \mid \mathbf{x})}{1 - P(y = 1 \mid \mathbf{x})} \right) = \mathbf{w}^\top \mathbf{x}$$

This transformation shows that logistic regression is effectively modeling the **log-odds** of the outcome as a linear function of the input features. That is,

$$\log\left(\frac{P(y=1 \mid \mathbf{x})}{P(y=0 \mid \mathbf{x})}\right) = \mathbf{w}^\top \mathbf{x}$$

This is the key conceptual bridge between logistic and linear regression:

- Linear regression predicts a continuous outcome directly from $\mathbf{w}^\top \mathbf{x}$.
- Logistic regression uses $\mathbf{w}^\top \mathbf{x}$ to model the log-odds of a binary outcome.

Thus, logistic regression can be seen as applying a sigmoid function to the output of a linear regression model, mapping it into the range $[0, 1]$ for probabilistic interpretation.

1.2.5 Implementation

Logistic Regression Implementation in Python

```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt

# Generate synthetic binary classification data
X, y = make_classification(n_samples=200, n_features=2,
                           n_redundant=0, n_informative=2,
                           random_state=42, n_clusters_per_class
    =1)

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)

# Initialize and train logistic regression model
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Predict on test set
y_pred = clf.predict(X_test)

# Evaluate model
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))

# Visualize the data
plt.figure(figsize=(8, 6))
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='bwr',
    edgecolor='k', s=30, label='True labels')

# Create mesh grid
x_min, x_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
y_min, y_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                     np.linspace(y_min, y_max, 100))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot decision boundary (class 0 in blue, class 1 in red)
plt.contourf(xx, yy, Z, alpha=0.2, cmap='bwr')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Logistic Regression Decision Boundary')
plt.legend()
plt.show()
```

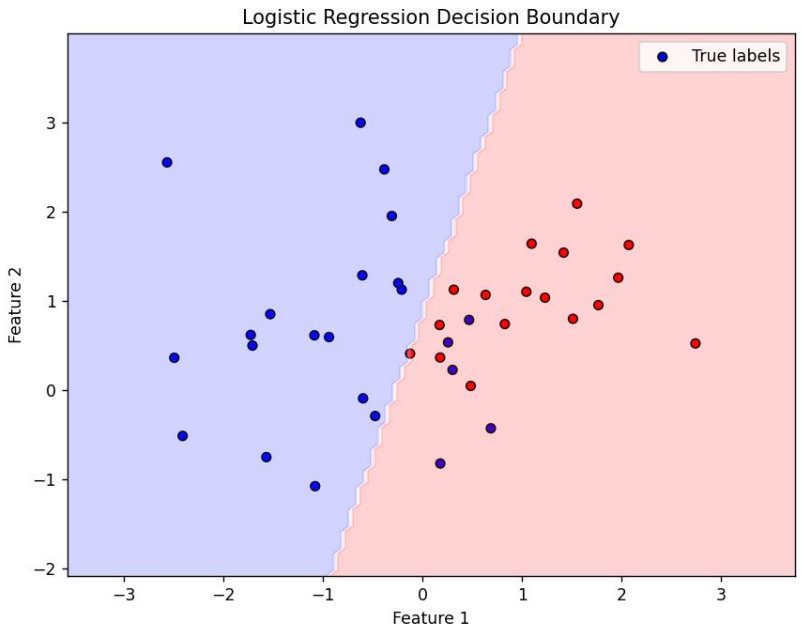

Figure 2: Example of Logistic Regression

- The logistic decision boundary is the boundary where the probability of class 0 = class 1
- This is non-linear if the features aren't perfectly linearly separable

2 SVM (Support Vector Machine)

2.1 SVM for Classification

2.1.1 Presentation

Support Vector Machines (SVMs) are supervised learning models used primarily for **classification** tasks. The fundamental idea behind SVMs is to find the **optimal hyperplane** that separates different classes with the **maximum margin**. This margin is defined as the distance between the hyperplane and the closest data points from each class, known as **support vectors**.

2.1.2 Hypotheses

The SVM model makes the following key assumptions:

- Data is approximately linearly separable in the original feature space or in a higher-dimensional feature space, with slack variables (ξ_i) introduced to allow for some misclassification or margin violations.
- Maximizing the margin between classes leads to better generalization on unseen data.
- The decision boundary is determined entirely by the support vectors, making the model robust to outliers not serving as support vectors.

2.1.3 Mathematical Formulation

For a binary classification problem with training data $\{(\mathbf{x}_i, y_i)\}_{1 \leq i \leq n}$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$, the SVM aims to find a hyperplane defined by:

$$h(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$$

The decision rule for classification is given by:

$$\text{class}(\mathbf{x}) = \text{sign}(h(\mathbf{x})) = \text{sign}(\mathbf{w}^T \mathbf{x} + b)$$

For the linearly separable case, we want to satisfy:

$$\begin{cases} \mathbf{w}^T \mathbf{x}_i + b \geq 1 & \text{if } y_i = 1 \\ \mathbf{w}^T \mathbf{x}_i + b \leq -1 & \text{if } y_i = -1 \end{cases}$$

These constraints can be combined into:

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i \in \{1, 2, \ldots, n\}$$

The distance from a point $\mathbf{x}$ to the hyperplane is $\frac{|\mathbf{w}^T \mathbf{x} + b|}{\|\mathbf{w}\|}$. For support vectors that lie exactly on the margin boundaries, this distance equals $\frac{1}{\|\mathbf{w}\|}$. Therefore, the width of the margin is $\frac{2}{\|\mathbf{w}\|}$.

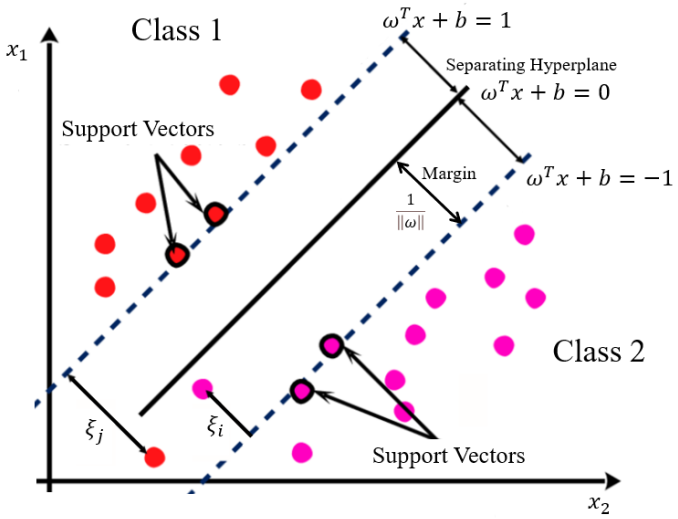

Figure 3: SVM binary classification example in a two-dimensional feature space

To maximize the margin, we need to minimize $\|\mathbf{w}\|$ or equivalently, minimize $\frac{1}{2}\|\mathbf{w}\|^2$ (the factor $\frac{1}{2}$ is introduced for derivation convenience). This leads to the following optimization problem:

$$\begin{cases} \min\limits_{\mathbf{w},b} & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \qquad \forall i = 1, 2, \ldots, n \end{cases}$$

For non-linearly separable data, we introduce **slack variables** $\xi_i \geq 0$ to allow for misclassifications:

$$\begin{cases} \min\limits_{\mathbf{w},b,\boldsymbol{\xi}} & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} & \begin{cases} y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i & \forall i = 1, 2, \ldots, n \\ \xi_i \geq 0 & \forall i = 1, 2, \ldots, n \end{cases} \end{cases}$$

The parameter $C > 0$ controls the trade-off between maximizing the margin and minimizing the classification error.

2.1.4 Dual Formulation

Primal Problem:

The constrained optimization problem can be solved using Lagrange multipliers. The Lagrangian is:

$$L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + \xi_i] - \sum_{i=1}^{n}\beta_i\xi_i$$

where $\alpha_i \geq 0$ and $\beta_i \geq 0$ are Lagrange multipliers.

At the optimal point, the derivatives of L with respect to the primal variables are zero:

$$\begin{cases} \frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0 & \implies \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \\ \frac{\partial L}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 & \implies \sum_{i=1}^{n} \alpha_i y_i = 0 \\ \frac{\partial L}{\partial \xi_i} = C - \alpha_i - \beta_i = 0 & \implies \alpha_i + \beta_i = C \end{cases}$$

Dual Problem:

Substituting these back into the Lagrangian and eliminating the primal variables leads to the dual optimization problem:

$$\begin{cases} \max\limits_{\boldsymbol{\alpha}} & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{s.t.} & \begin{cases} \sum_{i=1}^{n} \alpha_i y_i = 0 \\ 0 \leq \alpha_i \leq C \qquad \forall i = 1, 2, \ldots, n \end{cases} \end{cases}$$

Once the optimal $\boldsymbol{\alpha}^*$ is found, the optimal weight vector is:

$$\mathbf{w}^* = \sum_{i=1}^{n} \alpha_i^* y_i \mathbf{x}_i$$

The bias term b^* can be computed using any support vector $\mathbf{x}_s$ with $0 < \alpha_s^* < C$:

$$b^* = y_s - \sum_{i=1}^{n} \alpha_i^* y_i \mathbf{x}_i^T \mathbf{x}_s$$

2.1.5 Proof of the Dual Problem

To derive the dual problem, we begin by substituting the expressions for $\mathbf{w}$ (from the solution found in the primal problem) and the constraints into the Lagrangian:

$$\begin{aligned} L(\boldsymbol{\alpha}) &= \frac{1}{2} \left\| \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \right\|^2 + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i [y_i (\sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j^T \mathbf{x}_i + b) - 1 + \xi_i] - \sum_{i=1}^{n} \beta_i \xi_i \\ &= \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + \sum_{i=1}^{n} (C - \alpha_i - \beta_i) \xi_i - b \sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i \end{aligned}$$

Since $\alpha_i + \beta_i = C$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$, the second and third terms vanish, leaving:

$$L(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

This is the objective function of the dual problem, which we aim to maximize subject to the constraints $\sum_{i=1}^{n} \alpha_i y_i = 0$ and $0 \leq \alpha_i \leq C$.

2.1.6 The Kernel Trick

The dual formulation reveals an important feature: the data appears only in the form of dot products $\mathbf{x}_i^T \mathbf{x}_j$. This enables the **kernel trick**, which implicitly maps the data to a higher-dimensional feature space without explicitly computing the coordinates in that space.

Let $\phi : \mathbb{R}^d \to \mathcal{H}$ be a mapping to a higher-dimensional (possibly infinite-dimensional) feature space $\mathcal{H}$. The kernel function $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is defined as:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

Replacing the dot product in the dual problem with the kernel function:

$$\begin{cases} \max\limits_{\boldsymbol{\alpha}} & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \\ \\ \text{s.t.} & \begin{cases} \sum_{i=1}^n \alpha_i y_i = 0 \\ \\ 0 \leq \alpha_i \leq C \qquad \forall i = 1, 2, \ldots, n \end{cases} \end{cases}$$

The decision function becomes:

$$h(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^n \alpha_i^* y_i K(\mathbf{x}_i, \mathbf{x}) + b^* \right)$$

Common kernel functions include:

- **Linear**: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$
- **Polynomial**: $K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma \mathbf{x}_i^T \mathbf{x}_j + r)^d$, where $\gamma > 0$, $r \geq 0$, and $d \in \mathbb{N}$
- **Radial Basis Function (RBF)**: $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$, where $\gamma > 0$
- **Sigmoid**: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i^T \mathbf{x}_j + r)$, where $\gamma > 0$ and $r \geq 0$

For a function to be a valid kernel, it must satisfy Mercer's theorem, which states that K must be symmetric and positive semi-definite, i.e., $\sum_{i=1}^n \sum_{j=1}^n c_i c_j K(\mathbf{x}_i, \mathbf{x}_j) \geq 0$ for all $n \in \mathbb{N}$, $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, and $c_1, \ldots, c_n \in \mathbb{R}$.

The kernel trick transforms the SVM from a linear classifier in the original feature space to a non-linear classifier capable of finding complex decision boundaries, all while maintaining the computational efficiency of the dual formulation.

2.2 SVM for Regression (SVR)

Support Vector Machines can be extended from classification to regression problems, maintaining many of the same advantages. While SVM for classification aims to find a maximum-margin hyperplane separating classes, SVR aims to find a function that approximates the training points with at most ϵ deviation while remaining as flat as possible.

2.2.1 ϵ-SVR

Model and Hypotheses

In ϵ-SVR, we aim to find a function $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ with $\mathbf{w} \in \mathcal{X}$, $b \in \mathbb{R}$ such that:

- The function has at most ϵ deviation from the actual targets y_i for all training data.
- The function is as flat as possible (i.e., $\|\mathbf{w}\|$ is minimized).

Primal Formulation

For cases where exact solutions within ϵ precision may not exist, we introduce slack variables ξ_i, ξ_i' to allow for some errors. The optimization problem becomes:

$$\begin{cases} \min\limits_{\mathbf{w},b,\xi,\xi'} & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n(\xi_i + \xi_i') \\ \text{s.t.} & \begin{cases} y_i - (\mathbf{w}^T\mathbf{x}_i + b) \leq \epsilon + \xi_i \\ (\mathbf{w}^T\mathbf{x}_i + b) - y_i \leq \epsilon + \xi_i' \\ \xi_i, \xi_i' \geq 0 \qquad \forall i = 1, \ldots, n \end{cases} \end{cases}$$

where $C > 0$ is a regularization parameter that determines the trade-off between the flatness of h and the amount up to which deviations larger than ϵ are tolerated.

Dual Formulation

Primal Problem:

The primal optimization problem is formed by the Lagrangian:

$$\begin{aligned} L(\mathbf{w}, b, \xi_i, \xi_i', \alpha_i, \alpha_i', \eta_i, \eta_i') = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n(\xi_i + \xi_i') - \sum_{i=1}^n \alpha_i(\epsilon + \xi_i - y_i + \mathbf{w}^T\mathbf{x}_i + b) \\ - \sum_{i=1}^n \alpha_i'(\epsilon + \xi_i' + y_i - \mathbf{w}^T\mathbf{x}_i - b) - \sum_{i=1}^n(\eta_i\xi_i + \eta_i'\xi_i') \end{aligned}$$

where $\alpha_i, \alpha_i', \eta_i, \eta_i' \geq 0$ are Lagrange multipliers.

Taking the partial derivatives of L with respect to the primal variables $(\mathbf{w}, b, \xi_i, \xi_i')$ and setting them to zero:

$$\begin{cases} \frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^n(\alpha_i' - \alpha_i)\mathbf{x}_i = 0 & \implies \mathbf{w} = \sum_{i=1}^n(\alpha_i - \alpha_i')\mathbf{x}_i \\ \frac{\partial L}{\partial b} = \sum_{i=1}^n(\alpha_i - \alpha_i') = 0 & \\ \frac{\partial L}{\partial \xi_i} = C - \alpha_i - \eta_i = 0 & \implies \alpha_i = C - \eta_i \leq C \\ \frac{\partial L}{\partial \xi_i'} = C - \alpha_i' - \eta_i' = 0 & \implies \alpha_i' = C - \eta_i' \leq C \end{cases}$$

Dual Problem:

Substituting these back into the Lagrangian and simplifying, we obtain the dual optimization problem:

$$\begin{cases} \max\limits_{\alpha,\alpha'} & -\frac{1}{2}\sum_{i,j=1}^{n}(\alpha_i-\alpha_i')(\alpha_j-\alpha_j')\,\mathbf{x}_i^T\mathbf{x}_j - \epsilon\sum_{i=1}^{n}(\alpha_i+\alpha_i') + \sum_{i=1}^{n} y_i(\alpha_i-\alpha_i') \\ \text{s.t.} & \begin{cases} \sum_{i=1}^{n}(\alpha_i-\alpha_i') = 0 \\ 0 \le \alpha_i, \alpha_i' \le C \qquad \forall i = 1,\ldots,n \end{cases} \end{cases}$$

Support Vector Expansion:

From the KKT conditions, we can determine that for $\alpha_i \in]0, C[$ or $\alpha_i' \in]0, C[$, the corresponding training points $(\mathbf{x}_i, y_i)$ satisfy:

$$y_i - \mathbf{w}^T\mathbf{x}_i - b = \epsilon \quad \text{or} \quad y_i - \mathbf{w}^T\mathbf{x}_i - b = -\epsilon$$

These points are **support vectors**. The regression function can be written as:

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = \sum_{i=1}^{n}(\alpha_i - \alpha_i')\,\mathbf{x}_i^T\mathbf{x}_i + b$$

Computing the Bias Term b:

For any support vector x_i with $\alpha_i \in]0, C[$, we have $\eta_i > 0$, which implies $\xi_i = 0$ and:

$$y_i - \mathbf{w}^T\mathbf{x}_i - b = \epsilon$$

Similarly, for $\alpha_i' \in]0, C[$, we have:

$$y_i - \mathbf{w}^T\mathbf{x}_i - b = -\epsilon$$

We can compute b by averaging over all such support vectors:

$$b = \frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}}\left(y_i - \sum_{j=1}^{n}(\alpha_j - \alpha_j')\,\mathbf{x}_j^T\mathbf{x}_i \mp \epsilon\right)$$

where $\mathcal{S}$ is the set of indices of support vectors, and the sign before ϵ depends on whether α_i or α_i' is in $]0, C[$.

Kernelized Version:

The kernel trick can be applied to handle nonlinear regression problems. We replace the inner product $\mathbf{x}_i^T\mathbf{x}_j$ with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$, where ϕ maps the inputs to a higher-dimensional feature space:

$$h(\mathbf{x}) = \sum_{i=1}^{n}(\alpha_i - \alpha_i')K(\mathbf{x}_i, \mathbf{x}) + b$$

The dual optimization problem becomes:

$$\begin{cases} \max\limits_{\alpha,\alpha'} & -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}(\alpha_i-\alpha_i')(\alpha_j-\alpha_j')K(\mathbf{x}_i,\mathbf{x}_j) - \epsilon\sum_{i=1}^{n}(\alpha_i+\alpha_i') + \sum_{i=1}^{n} y_i(\alpha_i-\alpha_i') \\ \text{s.t.} & \begin{cases} \sum_{i=1}^{n}(\alpha_i-\alpha_i') = 0 \\ 0 \leq \alpha_i, \alpha_i' \leq C \qquad \forall i = 1,\ldots,n \end{cases} \end{cases}$$

2.2.2 ν-SVR

Model and Hypothesis

The ν-SVR formulation introduces a parameter $\nu \in]0,1]$ that serves as an upper bound on the fraction of training errors and a lower bound on the fraction of support vectors. This addresses a limitation of ϵ-SVR where the appropriate value of ϵ must be specified in advance.

Primal Formulation

The primal optimization problem for ν-SVR is:

$$\begin{cases} \min\limits_{\mathbf{w},b,\xi,\xi',\epsilon} & \frac{1}{2}\|\mathbf{w}\|^2 + C\left(\nu\epsilon + \frac{1}{n}\sum_{i=1}^{n}(\xi_i+\xi_i')\right) \\ \text{s.t.} & \begin{cases} y_i - \mathbf{w}^T\mathbf{x}_i - b \leq \epsilon + \xi_i \\ \mathbf{w}^T\mathbf{x}_i + b - y_i \leq \epsilon + \xi_i' \\ \xi_i, \xi_i' \geq 0 \quad \forall i = 1,\ldots,n \\ \epsilon \geq 0 \end{cases} \end{cases}$$

Here, ϵ is also a variable to be optimized, not a fixed parameter.

Dual Formulation

Primal Problem:

The Lagrangian for ν-SVR is:

$$\begin{aligned} L(\mathbf{w},b,\epsilon,\xi,\xi',\alpha,\alpha',\eta,\eta',\beta) = \frac{1}{2}\|\mathbf{w}\|^2 + C\nu\epsilon + \frac{C}{n}\sum_{i=1}^{n}(\xi_i+\xi_i') \\ - \sum_{i=1}^{n}\alpha_i(\epsilon+\xi_i - y_i + \mathbf{w}^T\mathbf{x}_i + b) \\ - \sum_{i=1}^{n}\alpha_i'(\epsilon+\xi_i' + y_i - \mathbf{w}^T\mathbf{x}_i - b) \\ - \sum_{i=1}^{n}(\eta_i\xi_i + \eta_i'\xi_i') - \beta\epsilon \end{aligned}$$

Taking partial derivatives with respect to the primal variables and setting them to zero:

$$\begin{cases} \frac{\partial L}{\partial \mathbf{w}} = 0 & \implies \mathbf{w} = \sum_{i=1}^{n} (\alpha_i - \alpha_i') \mathbf{x}_i \\ \frac{\partial L}{\partial b} = 0 & \implies \sum_{i=1}^{n} (\alpha_i - \alpha_i') = 0 \\ \frac{\partial L}{\partial \xi_i} = 0 & \implies \frac{C}{n} - \alpha_i - \eta_i = 0 \Rightarrow \alpha_i \leq \frac{C}{n} \\ \frac{\partial L}{\partial \xi_i'} = 0 & \implies \frac{C}{n} - \alpha_i' - \eta_i' = 0 \Rightarrow \alpha_i' \leq \frac{C}{n} \\ \frac{\partial L}{\partial \epsilon} = 0 & \implies C\nu - \sum_{i=1}^{n} (\alpha_i + \alpha_i') - \beta = 0 \end{cases}$$

Since $\beta \geq 0$, we have $\sum_{i=1}^{n} (\alpha_i + \alpha_i') \leq C\nu$.

Dual Problem:

The dual problem becomes:

$$\begin{cases} \max\limits_{\alpha, \alpha'} & -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (\alpha_i - \alpha_i')(\alpha_j - \alpha_j') K(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^{n} y_i (\alpha_i - \alpha_i') \\ \text{s.t.} & \begin{cases} \sum_{i=1}^{n} (\alpha_i - \alpha_i') = 0 \\ \sum_{i=1}^{n} (\alpha_i + \alpha_i') \leq C\nu \\ 0 \leq \alpha_i, \alpha_i' \leq \frac{C}{n} \qquad \forall i = 1, \ldots, n \end{cases} \end{cases}$$

Relationship Between ϵ-SVR and ν-SVR:

For any given ν-SVR solution with parameter ν yielding an optimal ϵ^*, there exists an ϵ-SVR problem with parameter $\epsilon = \epsilon^*$ that yields the same solution (up to scaling of the Lagrange multipliers). However, ν-SVR has the advantage that ν has a more intuitive interpretation and can be selected based on the desired fraction of support vectors.

3 Ensemble Methods

Ensemble methods are powerful machine learning techniques that combine multiple individual models (often called **base learners** or **weak learners**) to form a more robust and accurate model, typically referred to as a **strong learner**. The two main paradigms of ensemble methods are **Bagging** and **Boosting**.

3.1 Bagging (Bootstrap Aggregating)

3.1.1 Presentation

Bagging involves training multiple base models independently on different bootstrapped subsets of the training data and then combining their predictions, typically by averaging in regression tasks or using majority voting in classification.

3.1.2 Hypotheses

- Each base learner is unstable (e.g., high variance like decision trees)
- Reducing variance improves generalization

3.1.3 Mathematical Formulation

Given a training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{1 \leq i \leq m}$, we create B bootstrap samples $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(B)}$ by sampling with replacement from $\mathcal{D}$. Each base learner $h^{(b)}(\mathbf{x})$ is trained independently on its respective bootstrap dataset $\mathcal{D}^{(b)}$.

- For **regression**, the aggregated prediction is:

$$\hat{y}_{\text{bag}}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} h^{(b)}(\mathbf{x})$$

- For **classification**, the majority vote is used:

$$\hat{y}_{\text{bag}}(\mathbf{x}) = \text{mode}\left(\{h^{(b)}(\mathbf{x})\}_{1 \leq b \leq B}\right)$$

3.1.4 Variance Reduction Proof

Let's consider the regression case. Assume that each base learner $h^{(b)}(\mathbf{x})$ has expected value $\mathbb{E}[h^{(b)}(\mathbf{x})] = f(\mathbf{x})$ (i.e., they are unbiased estimators) and variance $\text{Var}[h^{(b)}(\mathbf{x})] = \sigma^2$.

For the bagged predictor:

$$\hat{y}_{\text{bag}}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} h^{(b)}(\mathbf{x})$$

The expected value is:

$$\begin{aligned}\mathbb{E}[\hat{y}_{\text{bag}}(\mathbf{x})] &= \mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B} h^{(b)}(\mathbf{x})\right] \\ &= \frac{1}{B}\sum_{b=1}^{B} \mathbb{E}[h^{(b)}(\mathbf{x})] \\ &= f(\mathbf{x})\end{aligned}$$

So the bagged estimator remains unbiased.

If the bootstrap samples were completely independent (which is an approximation), the variance would be:

$$\begin{aligned}\text{Var}[\hat{y}_{\text{bag}}(\mathbf{x})] &= \text{Var}\left[\frac{1}{B}\sum_{b=1}^{B} h^{(b)}(\mathbf{x})\right] \\ &= \frac{1}{B^2}\sum_{b=1}^{B} \text{Var}[h^{(b)}(\mathbf{x})] \\ &= \frac{\sigma^2}{B}\end{aligned}$$

In practice, the predictions from base models trained on bootstrap samples are positively correlated, with a correlation coefficient of ρ. The actual variance of the bagged estimator is given by:

$$\begin{aligned}\text{Var}[\hat{y}_{\text{bag}}(\mathbf{x})] &= \text{Var}\left[\frac{1}{B}\sum_{b=1}^{B} \hat{y}_b(\mathbf{x})\right] \\ &= \frac{1}{B^2}\sum_{b=1}^{B} \text{Var}[\hat{y}_b(\mathbf{x})] + \frac{1}{B^2}\sum_{b\neq b'} \text{Cov}(\hat{y}_b(\mathbf{x}), \hat{y}_{b'}(\mathbf{x}))\end{aligned}$$

For $b \neq b'$, the covariance between the predictions of different base models is $\rho\sigma^2$. Since there are $B(B-1)$ such pairs of different base models, the second sum becomes:

$$\frac{1}{B^2}\sum_{b\neq b'} \rho\sigma^2 = \frac{B(B-1)\rho\sigma^2}{B^2} = \frac{(B-1)\rho\sigma^2}{B}.$$

Thus, we get:

$$\begin{aligned}\text{Var}[\hat{y}_{\text{bag}}(\mathbf{x})] &= \frac{\sigma^2}{B} + \frac{(B-1)\rho\sigma^2}{B} \\ &= \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2\end{aligned}$$

As the number of base models B increases, the variance of the ensemble approaches $\rho\sigma^2$, which is generally lower than the variance of a single model, σ^2, as long as $\rho \neq 1$.

Note: The variance cannot shrink below $\rho\sigma^2$.

3.1.5 Out-of-Bag (OOB) Error

In ensemble methods like bagging (e.g., Random Forests), the **Out-of-Bag (OOB) error** provides an estimate of model performance without the need for a separate validation set. During the training process, each base model is trained on a bootstrap sample, meaning that some instances from the original dataset are not used in the training of a particular model. These unused instances are referred to as **out-of-bag** samples.

For each base model, the predictions on these out-of-bag samples are used to estimate the model's performance. Specifically, each data point's prediction is based on the base models that did not include that point in their bootstrap sample. The OOB error is then computed by aggregating these individual predictions and calculating the error rate, typically using majority voting for classification or averaging for regression.

For example, for each observation $(\mathbf{x}_i, y_i)$, we aggregate predictions only from models that didn't use this observation in training:

$$\hat{y}_{\text{OOB}}(\mathbf{x}_i) = \frac{1}{|\{b : (\mathbf{x}_i, y_i) \notin \mathcal{D}^{(b)}\}|} \sum_{b:(\mathbf{x}_i,y_i)\notin\mathcal{D}^{(b)}} h^{(b)}(\mathbf{x}_i)$$

The OOB error estimate is then:

$$\text{OOB Error} = \frac{1}{m} \sum_{i=1}^{m} L(y_i, \hat{y}_{\text{OOB}}(\mathbf{x}_i))$$

Where L is a loss function (e.g., squared error for regression or binary loss for classification).

The OOB error is particularly useful because it provides a reliable estimate of the generalization error without requiring additional data, making it both computationally efficient and effective for model evaluation.

3.1.6 Random Forests: Extending Bagging

Random Forests are an extension of bagging (Bootstrap Aggregating) that introduces additional randomness into the model-building process, further improving the model's robustness and accuracy.

While bagging trains each base model (e.g., decision trees) independently on bootstrapped subsets of the data, Random Forests enhance this by selecting a random subset of features at each node of the decision tree. This random feature selection helps decorrelate the trees in the ensemble and prevents overfitting.

The core idea behind Random Forests is to build an ensemble of decision trees, each trained on a bootstrap sample of the data. At each split in the decision trees, a random subset of features is considered, instead of using all available features. This additional randomness ensures that the trees are less correlated, improving the ensemble's generalization performance.

The mathematical formulation remains the same as bagging, but with lower correlation ρ between trees.

Training process:

1. **Bootstrap Sampling**: For each tree, a bootstrap sample (random sample with replacement) is drawn from the original dataset.
2. **Random Feature Selection**: At each node of the decision tree, a random subset of features is chosen. The *best split* is selected based on these features.
3. **Tree Construction**: The tree is built recursively, splitting nodes until a stopping criterion (e.g., maximum depth, minimum samples per leaf) is met.
4. **Ensemble Aggregation**: Once all trees are trained, the Random Forest aggregates their predictions. For regression, the final prediction is typically the average of all individual tree predictions. For classification, a majority vote is used.

Choosing the Best Split at Each Node

At each node, we aim to find the best split to maximize the homogeneity of the child nodes. The following criteria (measures of inpurity Q) are commonly used:

- for **classification:**
  Assume we have n features, m observations, and K classes.
  Let $p_k = \frac{m_k}{m}$ be the proportion of samples belonging to class k in the node.
  - **Misclassification Error:**
    $$Q_{\text{Misclassification Error}} = 1 - \max_{1 \le k \le K} (p_k)$$
    * Minimizes the proportion of misclassified samples.
  - **Gini Index:**
    $$Q_{\text{Gini Index}} = \sum_{k=1}^{K} p_k(1 - p_k) = 1 - \sum_{k=1}^{K} p_k^2$$
    * Measures impurity; minimizes the likelihood of misclassification.
    * The Gini index can also be interpreted as a form of variance reduction of class probabilities when splitting the data. A good split minimizes the Gini index, which corresponds to reducing the variance of the predicted classes, making the class distributions in the child nodes more homogeneous.
  - **Cross-Entropy:**
    $$Q_{\text{Cross-Entropy}} = -\sum_{k=1}^{K} p_k \log(p_k)$$
    * Measures the uncertainty of the classification; minimizes the distance from the true class distribution.
- for **regression:**
  Assume we have n features and m observations with continuous target values.
  Let $\{y^{(i)}\}_{1 \le i \le m} \subset \mathbb{R}$ denote the target values corresponding to each observation.

– **Mean Squared Error (MSE):**

$$Q_{\text{MSE}} = \frac{\text{Sum of Squared Errors (SSE)}}{m} = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h^{(b)}(\mathbf{x}^{(i)})\right)^2$$

  * Measures the average squared difference between true and predicted values, penalizing larger errors more heavily.

– **Mean Absolute Error (MAE):**

$$Q_{\text{MAE}} = \frac{1}{m}\sum_{i=1}^{m}\left|y^{(i)} - h^{(b)}(\mathbf{x}^{(i)})\right|$$

  * Measures the average absolute difference between true and predicted values, treating all errors equally.

– **Coefficient of Determination ($\mathbf{R^2}$):**

$$Q_{R^2} = 1 - \frac{\text{Sum of Squared Errors (SSE)}}{\text{Total Sum of Squares (TSS)}} = 1 - \frac{\sum_{i=1}^{m}\left(y^{(i)} - h^{(b)}(\mathbf{x}^{(i)})\right)^2}{\sum_{i=1}^{m}\left(y^{(i)} - \bar{y}\right)^2}$$

where $\bar{y}$ is the average of the observed target values $y^{(i)}$, where N is the number of data points.
Note: Sum of Squared Errors (SSE) is the same thing as Residual Sum of Squares (RSS)

  * R^2 Represents the proportion of variance in the true values explained by the model, with values closer to 1 indicating better fit.

Node Split Optimization Problem: The goal is to minimize the impurity measure Q after the split, which is computed as:

$$Q_{\text{split}} = \min_{\substack{x_j \in \{x_1, x_2, \ldots, x_n\} \\ t \in \mathbb{R}}} \left\{ \frac{m_L}{m} \cdot Q(y_L) + \frac{m_R}{m} \cdot Q(y_R) \right\}$$

where:

- x_j: is the chosen feature among the n features.
- t: is the threshold used to split the data.
- m: is the total number of data points at the current node before the split.
- m_L: is the number of samples in the left child node after the split.
- m_R: is the number of samples in the right child node after the split.
- $Q(y_L)$: is the impurity of the target variable y in the left child node.
- $Q(y_R)$: is the impurity of the target variable y in the right child node.

This means we are considering all possible ways to split the data by selecting a feature $x_j \in \{x_1, x_2, \ldots, x_n\}$ and a threshold $t \in \mathbb{R}$, where x_j is the feature being used for the split and t is the threshold value for the split.

Example: Random Forests with 3 Decision Trees

Assume we have a simple dataset with four samples and three features: X_1, X_2, X_3. The goal is to build a Random Forest with three decision trees, each trained on a bootstrap sample of the data. At each split of each tree, a random subset of features is selected.

Tree	Bootstrap Sample	Features Considered for First Split	First Split	Features Considered for Second Split	Second Split
T_1	Sample 1, 2, 4	X_1, X_2	$X_1 \leq 5$	X_2, X_3	$X_2 \leq 3$
T_2	Sample 1, 3, 4	X_1, X_3	$X_1 \leq 4$	X_3	$X_3 \leq 2$
T_3	Sample 2, 3, 4	X_2, X_3	$X_3 \leq 3$	X_1, X_2	$X_1 \leq 6$

Explanation of the Table:

- **Bootstrap Sample**: Each tree is trained on a random bootstrap sample of the data. For example, Tree 1 is trained on samples 1, 2, and 4, while Tree 2 is trained on samples 1, 3, and 4.
- **Features Considered for Splits**: For each split in the tree, only a random subset of features is considered. For example, Tree 1 considers features X_1 and X_2 for the first split, while Tree 2 considers X_1 and X_3.
- **Split Conditions**: At each node, a condition is applied based on the selected feature, such as $X_1 \leq 5$ for Tree 1's first split.

Advantages of Random Forests:

- **Reduced Overfitting**: By averaging multiple decision trees and introducing randomness, Random Forests typically perform better than individual decision trees, which are prone to overfitting.
- **Reduced Variance**: The random selection of features further decorrelates the trees, reducing variance.
  - Typical values: $m \approx \lfloor \sqrt{p} \rfloor$ for classification and $m \approx \lfloor \frac{p}{3} \rfloor$ for regression
- **Feature Importance**: Random Forests can be used to estimate the importance of different features in making predictions.
- **Robustness**: Due to the ensemble nature of the model, Random Forests are less sensitive to noise and outliers.

Random Forest Algorithm:

Algorithm 1 Random Forest Algorithm

Input: Training dataset $\mathcal{D}$, number of trees B, number of features to sample $s \leq m$ per tree
for $b = 1$ to B **do**
  Draw a bootstrap sample $\mathcal{D}^{(b)}$ from the training data
  Grow a decision tree T_b:
  while node is not a leaf **do**
    Randomly select $s \leq m$ features
    Find the best split among the s features
    Split the node
  end while
end for
Output: Set of B decision trees $\{T_1, T_2, \ldots, T_B\}$

Implementation:

Random Forest Implementation in Python

```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the iris dataset
data = load_iris()
X = data.data  # Features
y = data.target  # Target labels

# Split the dataset 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)

# Initialize the Random Forest Classifier
rf_classifier = RandomForestClassifier(n_estimators=100,
    random_state=42)

# Train the model
rf_classifier.fit(X_train, y_train)
# Make predictions on the test set
y_pred = rf_classifier.predict(X_test)
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

print(f"Accuracy of Random Forest: {accuracy:.4f}")
```

3.1.7 Extra-Trees: Extremely Randomized Trees

Extremely Randomized Trees (Extra-Trees) is an ensemble learning method that extends Random Forests by increasing the randomness during tree construction. Unlike Random Forests, which use bootstrap sampling, Extra-Trees builds trees using the entire training dataset. Additionally, at each node, Extra-Trees selects a random subset of features and a random split point for each feature, rather than optimizing the split for impurity.

Training process:

1. **No Bootstrap Sampling**: Unlike Random Forests, Extra Trees uses the entire original dataset for each tree, without drawing a bootstrap sample.
2. **Random Feature Selection**: At each node of the decision tree, a random subset of features is selected. The split points for these features are chosen *randomly*, rather than optimizing for the best split.
3. **Tree Construction**: The tree is constructed recursively, splitting nodes until a stopping criterion (e.g., maximum depth, minimum samples per leaf) is met.
4. **Ensemble Aggregation**: After all trees are trained, the Extra Trees ensemble aggregates their predictions. For regression, the final prediction is the average of all individual tree predictions. For classification, a majority vote is used.

Example: Extra-Trees with 3 Decision Trees

Assume we have a simple dataset with four samples and three features: X_1, X_2, X_3. The goal is to build an Extra-Trees model with three decision trees, each trained on the entire dataset (no bootstrap sampling). At each split of each tree, a random subset of features is selected, and the split points are chosen randomly rather than optimized.

Tree	Dataset (No Bootstrap)	Features Considered for First Split	First Split	Features Considered for Second Split	Second Split
T_1	Sample 1, 2, 3, 4	X_1, X_2	$X_1 \leq 5$	X_2, X_3	$X_2 \leq 3$
T_2	Sample 1, 2, 3, 4	X_1, X_3	$X_1 \leq 4$	X_3	$X_3 \leq 2$
T_3	Sample 1, 2, 3, 4	X_2, X_3	$X_3 \leq 3$	X_1, X_2	$X_1 \leq 6$

Explanation of the Table:

- **Dataset (No Bootstrap)**: Each tree is trained on the entire dataset (without bootstrap sampling). For example, Tree 1 is trained on the full dataset consisting of samples 1, 2, and 4.
- **Features Considered for Splits**: At each node, only a random subset of features is selected. For example, Tree 1 considers features X_1 and X_2 for the first split, while Tree 2 considers X_1 and X_3.
- **Split Points**: In Extra Trees, the split points for each feature are randomly chosen, not optimized. For example, Tree 1 splits on $X_1 \leq 5$ for the first split, but the split points are selected randomly rather than by minimizing impurity.

Advantages of Extremely Randomized Trees:

- **Reduced variance**: The added randomness helps prevent overfitting.
- **Faster training**: No search for optimal splits speeds up training.
- **Better generalization**: Randomness improves model robustness.

Extremely Randomized Trees Algorithm:

Algorithm 2 Extra-Trees Algorithm

for $b = 1$ to B **do**
  Use the entire training set (no bootstrap)
  Grow a decision tree T_b:
  while node is not a leaf **do**
    Randomly select $s \leq m$ features from the total m features
    For each feature, randomly select a split point
    Split using the feature with the best randomly selected split point
  end while
end for

This additional randomization can further reduce variance and computational cost.

Implementation:

Extremely Randomized Trees Implementation in Python

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.metrics import accuracy_score

# Load data
data = load_iris()
X, y = data.data, data.target

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

# Train Extra-Trees
et = ExtraTreesClassifier(n_estimators=100)
et.fit(X_train, y_train)

# Evaluate
y_pred = et.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

3.2 Boosting

3.2.1 Presentation

Boosting trains base models sequentially, where each new model focuses on correcting the errors of the previous ones. The final prediction is a weighted combination of all base learners.

3.2.2 Hypotheses

- Base learners are weak (e.g., slightly better than random guessing).
- Model performance improves by emphasizing previous mistakes.

3.2.3 Mathematical Formulation

Boosting builds a strong classifier $H(\mathbf{x})$ as a linear combination of weak learners:

$$H(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})$$

Where:

- $h_t(\mathbf{x})$ is the weak learner at iteration t,
- $\alpha_t \in \mathbb{R}$ is the weight assigned to learner h_t,
- T is the total number of boosting rounds.

Each new learner h_t is trained on a modified version of the dataset, where examples misclassified by the previous learners are given more importance. This reweighting process forces the next learner to focus on the "hard" cases that previous learners struggled with.

In the case of **classification**, the final prediction is usually made by taking the sign of the weighted sum:

$$\hat{y} = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})\right)$$

In the case of **regression**, the final prediction is the raw sum:

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

Boosting methods differ in how they choose α_t, update sample weights, and compute the loss — popular examples include **AdaBoost**, **Gradient Boosting**, and **XGBoost**.

3.2.4 Adaboost: The Original Boosting Algorithm

AdaBoost (Adaptive Boosting) is an ensemble learning method that combines multiple weak learners to form a strong classifier. It operates by iteratively training weak learners on weighted versions of the data, focusing more on the observations that previous models misclassified.

- **Entries:** A dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{1 \leq i \leq m}$, the size T of the ensemble.
  - $\mathbf{x}_i \in \mathbb{R}^n$ is the feature vector of the i-th sample (with n features)
  - $y_i \in \{-1, 1\}$ is the class label of the i-th sample (assuming a binary classification problem)
  - m is the total number of samples in the dataset
- **Initialization:** All observations are initially assigned equal weights: $w_i = \frac{1}{m}$. This means each sample has the same influence on the first weak learner.
- **Iterative Training:** For each boosting round $t = 1, \ldots, T$:
  1. A weak learner $h_t(\mathbf{x})$ is trained on the dataset using the current weights $w_i^{(t)}$.
  2. The **weighted classification error** is computed as:
  $$\varepsilon_t = \sum_{i=1}^{m} w_i^{(t)} \cdot \mathbf{1}(h_t(\mathbf{x}_i) \neq y_i)$$
  3. The **learning rate** is quantified by:
  $$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$
  4. The **observation weights** are updated using:
  $$w_i^{(t+1)} \leftarrow w_i^{(t)} \cdot \exp(-\alpha_t y_i h_t(\mathbf{x}_i))$$
  This increases the weight of misclassified samples (making them more influential) and decreases the weight of correctly classified ones:
     - If the prediction is wrong (i.e. $y_i h_t(\mathbf{x}_i) = -1$): the weight increases.
     - If the prediction is correct (i.e. $y_i h_t(\mathbf{x}_i) = +1$): the weight decreases.
  5. The weights are then normalized so that $\sum_{i=1}^{m} w_i = 1$, i.e., $\forall 1 \leq i \leq m$:
  $$w_i^{(t+1)} \leftarrow \frac{w_i^{(t+1)}}{\sum_{i=1}^{m} w_i^{(t+1)}} = \frac{w_i^{(t)} \cdot \exp(-\alpha_t y_i h_t(\mathbf{x}_i))}{\sum_{i=1}^{m} w_i^{(t)} \cdot \exp(-\alpha_t y_i h_t(\mathbf{x}_i))}$$
- **Final Classifier:** After T rounds, the final prediction is a weighted majority vote (or sum) of all weak learners:
$$H(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}) \right)$$

where α_t is the weight of the weak learner $h_t(\mathbf{x})$, computed based on its error rate ε_t as:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$

The learning rate ensures that weak learners with small errors are given higher weights, thus placing more emphasis on correctly classifying harder examples. This adaptive mechanism allows the model to focus on improving its performance by emphasizing the weak learners that contribute the most to reducing the overall classification error.

This process allows AdaBoost to adaptively focus on hard examples and progressively build a strong classifier by combining many weak models.

Algorithm

Algorithm 3 AdaBoost Algorithm

Input: Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, number of rounds T

Initialize weights: $w_i^{(1)} = \frac{1}{m}$ for $i = 1, 2, \ldots, m$

for $t = 1$ to T **do**

Train weak classifier $h_t(\mathbf{x})$ using weights $w_i^{(t)}$

Compute weighted error:

$$\varepsilon_t = \sum_{i=1}^{m} w_i^{(t)} \cdot \mathbf{1}(h_t(\mathbf{x}_i) \neq y_i)$$

Compute model weight:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$

Update weights:

$$w_i^{(t+1)} \leftarrow w_i^{(t)} \cdot \exp\left(-\alpha_t y_i h_t(\mathbf{x}_i)\right) \quad \text{for } i = 1, \ldots, m$$

Normalize weights so that $\sum_{i=1}^m w_i^{(t+1)} = 1$

end for

Output: Final classifier

$$H(\mathbf{x}) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}) \right)$$

AdaBoost Error Bound

We define the training error of the final classifier $H(\mathbf{x})$ as:

$$\varepsilon = \frac{1}{m} \sum_{i=1}^{m} \mathbf{1}(H(\mathbf{x}_i) \neq y_i)$$

Theorem: The training error ε of the AdaBoost classifier is bounded by:

$$\varepsilon \leq \exp\left(-2\sum_{t=1}^{T}\left(\frac{1}{2}-\varepsilon_t\right)^2\right)$$

Proof:
Step 1: $H(\mathbf{x}_i) \neq y_i$ if and only if $y_i \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}_i) < 0$. Therefore:

$$\begin{aligned}\varepsilon &= \frac{1}{m}\sum_{i=1}^{m}\mathbf{1}\left(y_i\sum_{t=1}^{T}\alpha_t h_t(\mathbf{x}_i) < 0\right)\\ &\leq \frac{1}{m}\sum_{i=1}^{m}\exp\left(-y_i\sum_{t=1}^{T}\alpha_t h_t(\mathbf{x}_i)\right)\end{aligned}$$

The inequality holds because the exponential function is always non-negative, and exceeds 1 when its argument is negative.

Step 2: Define Z_t as the normalization factor in round t:

$$Z_t = \sum_{i=1}^{m} w_i^{(t+1)} = \sum_{i=1}^{m} w_i^{(t)}\exp(-\alpha_t y_i h_t(\mathbf{x}_i))$$

We can show that:

$$\begin{aligned}Z_t &= \sum_{i=1}^{m} w_i^{(t)}\exp(-\alpha_t y_i h_t(\mathbf{x}_i))\\ &= \sum_{i:h_t(\mathbf{x}_i)=y_i} w_i^{(t)}e^{-\alpha_t} + \sum_{i:h_t(x_i)\neq y_i} w_i^{(t)}e^{\alpha_t}\\ &= (1-\varepsilon_t)e^{-\alpha_t} + \varepsilon_t e^{\alpha_t}\end{aligned}$$

Substituting the value of $\alpha_t = \frac{1}{2}\ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$:

$$\begin{aligned}Z_t &= (1-\varepsilon_t)\sqrt{\frac{\varepsilon_t}{1-\varepsilon_t}} + \varepsilon_t\sqrt{\frac{1-\varepsilon_t}{\varepsilon_t}}\\ &= 2\sqrt{\varepsilon_t(1-\varepsilon_t)}\\ &= \sqrt{1-4\left(\frac{1}{2}-\varepsilon_t\right)^2}\\ &\leq \exp\left(-2\left(\frac{1}{2}-\varepsilon_t\right)^2\right)\end{aligned}$$

where we used the inequality $\sqrt{1-x} \leq e^{-x/2}$ for $x \in [0,1]$.

Step 3: The product of the normalization factors gives:

$$\prod_{t=1}^{T} Z_t \leq \exp\left(-2\sum_{t=1}^{T}\left(\frac{1}{2}-\varepsilon_t\right)^2\right)$$

Step 4: For any training example $(\mathbf{x}_i, y_i)$, the final weight can be expressed as:

$$\begin{aligned} w_i^{(T+1)} &= w_i^{(1)} \prod_{t=1}^{T} \frac{\exp(-\alpha_t y_i h_t(\mathbf{x}_i))}{Z_t} \\ &= \frac{1}{m} \frac{\exp\left(-\sum_{t=1}^{T} \alpha_t y_i h_t(\mathbf{x}_i)\right)}{\prod_{t=1}^{T} Z_t} \end{aligned}$$

Rearranging, we get:

$$\exp\left(-\sum_{t=1}^{T} \alpha_t y_i h_t(\mathbf{x}_i)\right) = m \cdot w_i^{(T+1)} \cdot \prod_{t=1}^{T} Z_t$$

Step 5: Summing over all examples and using the bounds from Steps 1-4:

$$\begin{aligned} \varepsilon &\leq \frac{1}{m} \sum_{i=1}^{m} \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}_i)\right) \\ &= \frac{1}{m} \sum_{i=1}^{m} m \cdot w_i^{(T+1)} \cdot \prod_{t=1}^{T} Z_t \\ &= \left(\sum_{i=1}^{m} w_i^{(T+1)}\right) \cdot \prod_{t=1}^{T} Z_t \\ &= 1 \cdot \prod_{t=1}^{T} Z_t \\ &\leq \exp\left(-2 \sum_{t=1}^{T} \left(\frac{1}{2} - \varepsilon_t\right)^2\right) \end{aligned}$$

The bound shows that as long as each weak learner performs better than random guessing (i.e., $\varepsilon_t < \frac{1}{2}$), the term $\frac{1}{2} - \varepsilon_t$ is positive, causing the training error to decrease exponentially with the number of boosting rounds T.

This bound explains why AdaBoost is effective when weak learners consistently outperform random guessing. The error decreases exponentially in T if each $\varepsilon_t < \frac{1}{2}$, and the larger the margin $\frac{1}{2} - \varepsilon_t$, the faster the convergence.

The upper bound also suggests that AdaBoost may overfit if T becomes too large, especially with noisy data where weak learners might struggle to maintain $\varepsilon_t < \frac{1}{2}$.

AdaBoost as Gradient Descent

AdaBoost can be viewed as performing gradient descent to minimize the following exponential loss function:

$$L(y, H(\mathbf{x})) = \exp(-y \cdot H(\mathbf{x}))$$

where:

- $y \in \{-1, 1\}$ represents the true label for binary classification (either class -1 or class 1).
- $H(\mathbf{x})$ is the output of the model for the input $\mathbf{x}$, which is typically a weighted sum of weak classifiers.
- The loss function penalizes incorrect classifications more heavily, as the exponential function increases exponentially when the prediction $H(\mathbf{x})$ does not match the true label y.
- AdaBoost iteratively adjusts the weights of the weak classifiers to minimize this exponential loss, focusing more on the misclassified data points in each iteration, similar to gradient descent optimization.

In gradient descent, we calculate the gradient of the loss with respect to the model parameters and update the parameters in the direction that reduces the loss:

$$\nabla_\theta L(y, H(\mathbf{x}))$$

In AdaBoost, however, this gradient is not computed directly, but the effect is similar: rather than directly minimizing the loss, after each round of boosting, the algorithm updates the weights of the weak classifiers in such a way that misclassified points are given more importance in the next iteration.

3.3 Gradient Boosting

Gradient Boosting is a powerful ensemble technique that builds a strong predictor by sequentially adding weak learners, typically decision trees, in a stage-wise manner. It generalizes the boosting framework by optimizing an arbitrary differentiable loss function using gradient descent in function space.

- **Objective:** Minimize a loss function $L(y, H(\mathbf{x}))$, where $H(\mathbf{x})$ is the ensemble model.
- **Initialization:** Start with a constant model that minimizes the loss over all training examples. For regression tasks, this is often the mean of the target values.

  $$H_0(\mathbf{x}) = \arg\min_{\gamma \in \mathbb{R}} \sum_{i=1}^{m} L\left(y_i, \gamma\right)$$

  where γ is a constant prediction applied to all inputs.
- **Procedure:** Starting from the initial model $H_0(\mathbf{x})$, Gradient Boosting constructs the model iteratively for $t \in \{1, 2, \ldots, T\}$ iterations:

  $$H_t(\mathbf{x}) = H_{t-1}(\mathbf{x}) + \gamma_t h_t(\mathbf{x})$$

  where:
  - $H_t(\mathbf{x})$ is the updated ensemble model
  - $H_{t-1}(\mathbf{x})$: is the ensemble model before iteration t, i.e., the sum of the first $t-1$ weak learners.

- $h_t(\mathbf{x})$ is the base learner fitted to the negative gradient of the loss function at iteration t
- γ_t is a step size (often chosen via line search)

$$\gamma_t = \arg\min_{\gamma \in \mathbb{R}} \sum_{i=1}^{m} L\left(y_i, H_{t-1}(\mathbf{x_i}) + \gamma \cdot h_t(\mathbf{x_i})\right)$$

- **Gradient Step:** At each iteration, the pseudo-residuals are computed as:

$$r_i^{(t)} = -\frac{\partial L(y_i, H_{t-1}(\mathbf{x_i}))}{\partial H_{t-1}(\mathbf{x_i})} \quad \forall i = 1, ..., m$$

  This is the key idea of gradient boosting, i.e., computing pseudo-residuals, which are the negative gradients of the loss function with respect to the current model's predictions. These residuals represent how much the current model's predictions deviate from the true labels, essentially indicating the direction in which the model should improve. The pseudo-residuals act as the new targets for training the weak learner $h_t(\mathbf{x})$.

  - For example, if we are using the squared error loss for regression, the gradient of the loss with respect to the prediction $H_{t-1}(\mathbf{x}_i)$ at iteration $t-1$ is simply the residual:

$$r_i^{(t)} = y_i - H_{t-1}(\mathbf{x}_i)$$

    In this case, the residuals are just the differences between the true values y_i and the model's current predictions $H_{t-1}(\mathbf{x}_i)$.

- **Train the Weak Learner:** Once the pseudo-residuals $r_i^{(t)}$ are computed, we train a weak learner (typically a shallow decision tree) to predict these residuals. The weak learner is trained on the dataset $\{(\mathbf{x}_i, r_i^{(t)})\}_{i=1}^m$, where $\mathbf{x}_i$ is the input and $r_i^{(t)}$ is the target (pseudo-residual). This step allows the weak learner to model the errors in the current model and provide corrections to improve predictions.

- **Update the Model:** Once the weak learner $h_t(\mathbf{x})$ has been trained, its predictions (based on the residuals) are used to update the model. However, the predictions are not added directly to the model. Instead, they are multiplied by a step size (learning rate) γ_t, which controls the contribution of the weak learner to avoid overfitting:

$$H_t(\mathbf{x}) = H_{t-1}(\mathbf{x}) + \gamma_t h_t(\mathbf{x})$$

- **Final Model:** After T iterations, the final prediction is given by:

$$H_T(\mathbf{x}) = H_0(\mathbf{x}) + \sum_{t=1}^{T} \gamma_t h_t(\mathbf{x})$$

Gradient Boosting is highly flexible and can be applied to regression, classification, and ranking tasks. It is the foundation for popular libraries such as **XGBoost**, **LightGBM**, and **CatBoost**.

Algorithm

Algorithm 4 Gradient Boosting Algorithm

Initialize with a constant model:

$$H_0(\mathbf{x}) = \arg\min_{\gamma \in \mathbb{R}} \sum_{i=1}^{m} L(y_i, \gamma)$$

for $t = 1$ to T **do**

Compute pseudo-residuals (negative gradients):

$$r_i^{(t)} = -\frac{\partial L(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)} \quad \text{for } i = 1, ..., m$$

Fit a base learner $h_t(\mathbf{x})$ to the pseudo-residuals $\{(\mathbf{x}_i, r_i^{(t)})\}_{1 \le i \le m}$

Compute the multiplier by solving:

$$\gamma_t = \arg\min_{\gamma \in \mathbb{R}} \sum_{i=1}^{m} L(y_i, H_{t-1}(\mathbf{x}_i) + \gamma h_t(\mathbf{x}_i))$$

Update the model:

$$H_t(\mathbf{x}) = H_{t-1}(\mathbf{x}) + \gamma_t h_t(\mathbf{x})$$

end for

Output the final model:

$$H_T(\mathbf{x}) = H_0(\mathbf{x}) + \sum_{t=1}^{T} \gamma_t h_t(\mathbf{x})$$

Common Loss Functions

- **Regression with L2 loss**:

$$L(y, H(\mathbf{x})) = \frac{1}{2}(y - H(\mathbf{x}))^2$$

  - Pseudo-residuals: $r_i^{(t)} = y_i - H_{t-1}(\mathbf{x}_i)$

- **Binary classification with logistic loss** ($y \in \{-1, 1\}$):

$$L(y, H(\mathbf{x})) = \log(1 + \exp(-2yH(\mathbf{x})))$$

  - Pseudo-residuals: $r_i^{(t)} = \frac{2y_i}{1+\exp(2y_i H_{t-1}(\mathbf{x}_i))}$

3.3.1 Implementation

Gradient Boosting Implementation in Python

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np

# Create a synthetic dataset
X, y = make_regression(n_samples=1000, n_features=20,
    n_informative=15, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=42)

# Train a Gradient Boosting regressor
gbr = GradientBoostingRegressor(n_estimators=100, learning_rate
    =0.1, max_depth=3, random_state=42)
gbr.fit(X_train, y_train)

# Make predictions and evaluate
y_pred = gbr.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print(f"Gradient Boosting RMSE: {rmse:.4f}")

# Learning curve analysis
test_scores = np.zeros(100)
for i, y_pred in enumerate(gbr.staged_predict(X_test)):
    test_scores[i] = mean_squared_error(y_test, y_pred)

# Optimal number of trees to avoid overfitting
optimal_n_trees = np.argmin(test_scores) + 1
print(f"Optimal number of trees: {optimal_n_trees}")
```

3.3.2 XGBoost (Extreme Gradient Boosting)

XGBoost extends the gradient boosting framework by introducing several key enhancements. These improvements aim to make the model more robust and efficient. The key features include:

1. **Regularization**: XGBoost introduces a regularization term into the objective function to control overfitting and improve generalization. The objective function is now defined as:

$$\mathrm{J}(H, T, \gamma, \lambda) = \sum_{i=1}^{m} L(y_i, H(\mathbf{x}_i)) + \sum_{j=1}^{N} \Omega(T_j)$$

   where:

   - $L(y_i, H(\mathbf{x}_i))$ is the loss function, with $H(\mathbf{x}_i)$ being the prediction of the ensemble model for the i-th input sample.

- N is the number of trees.
- The regularization term is $\Omega(T_j) = \gamma N_j + \frac{1}{2}\lambda\|w_j\|^2$, where:
  - N_j is the number of leaves in the tree T_j, where T_j is the weak learner.
  - γ controls the complexity of the tree (e.g., the number of leaves).
  - λ is a regularization parameter that penalizes large values of the weights w_j associated with each leaf.

This regularization term helps to control the complexity of the individual trees and prevents overfitting, making XGBoost more robust.

2. **Newton Boosting**: XGBoost uses second-order Taylor expansion to approximate the loss function, improving the efficiency of the optimization process by utilizing both the first and second derivatives (gradients and Hessians) of the loss function. The update to the ensemble model H at the t-th iteration is given by:

$$L(y_i, H_{t-1}(\mathbf{x}_i) + T_j^{(t)}(\mathbf{x}_i)) \approx L(y_i, H_{t-1}(\mathbf{x}_i)) + \nabla_i L \cdot T_j^{(t)}(\mathbf{x}_i) + \frac{1}{2}\nabla_i^2 L \cdot T_j^{(t)2}(\mathbf{x}_i)$$

where:

- $\nabla_i L = \frac{\partial L(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)}$ is the gradient of the loss function with respect to the model's prediction at the previous iteration
- $\nabla_i^2 L = \frac{\partial^2 L(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}^2(\mathbf{x}_i)}$ is the Hessian of the loss function with respect to the model's prediction at the previous iteration
- $T_j^{(t)}(\mathbf{x}_i)$ represents the prediction of the j-th tree T_j at the t-th iteration for the input $\mathbf{x}_i$

By incorporating the second-order information (Hessians), XGBoost improves the convergence rate and the quality of the learned model.

3. **Optimal Split Finding**: XGBoost uses these gradients and Hessians to train the next tree by minimizing the approximation of the regularized loss function. For the j-th tree at iteration t, the objective is minimized with respect to the leaf values of the tree (the values that the tree outputs). For each leaf in a decision tree, XGBoost calculates the optimal weight w^* as follows:

$$w_j^* = -\frac{\sum_{i \in I_j} \nabla_i L}{\sum_{i \in I_j} \nabla_i^2 L + \lambda}$$

where:

- I_j is the set of data points assigned to the j-th leaf
- $\nabla_i L$ and $\nabla_i^2 L$ are the gradient and Hessian of the loss function with respect to the model's prediction for data point i
- λ is the regularization parameter (L2 regularization by default) on the leaf weight

The weight w_j^* represents the optimal prediction value for the i-th leaf of the j-th tree that minimizes the loss function (considering both the gradients and Hessians of the loss function, along with the regularization term).

Gain:
The **gain** measures the improvement in the loss function resulting from a split at a particular node. It is calculated by comparing the score of the parent node to the sum of the scores of the left and right child nodes. The formula is:

$$\text{Gain} = \frac{1}{2}\left[\underbrace{\frac{\left(\sum_{i\in I_L}\nabla_i L\right)^2}{\sum_{i\in I_L}\nabla_i^2 L+\lambda}}_{\text{Score of left child}} + \underbrace{\frac{\left(\sum_{i\in I_R}\nabla_i L\right)^2}{\sum_{i\in I_R}\nabla_i^2 L+\lambda}}_{\text{Score of right child}} - \underbrace{\frac{\left(\sum_{i\in I}\nabla_i L\right)^2}{\sum_{i\in I}\nabla_i^2 L+\lambda}}_{\text{Score of parent}}\right] - \gamma$$

where:

- I_L and I_R denote the sets of data points assigned to the left and right child nodes,
- I is the set of data points in the current (parent) node,
- γ is the regularization term that penalizes splits with low gain.

Gain determines whether a split should be made by evaluating its impact on reducing the loss, while avoiding overfitting by discouraging weak or unnecessary splits. The optimal leaf weight w_j^* specifies the best prediction value after the split. Together, they guide both the structure of the tree and the values assigned to its leaves.

Summary:

- **Regularization** helps prevent overfitting and keeps the model simpler.
- **Newton Boosting** improves optimization using second-order derivatives, speeding up convergence.
- **Optimal split finding** leverages both gradients and Hessians to identify the best decision tree splits, improving predictive accuracy.

Implementation

XGBoost Implementation in Python

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load dataset
boston = load_boston()
X = boston.data
y = boston.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_state=42)

# Convert the datasets into DMatrix format (XGBoost's internal
    data structure)
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set the parameters for the model
params = {
    # regression task with squared error loss
    'objective': 'reg:squarederror',
    # maximum depth of trees
    'max_depth': 6,
    # step size for each iteration
    'learning_rate': 0.1,
    # number of boosting rounds
    'n_estimators': 100,
    # evaluation metric (Root Mean Squared Error)
    'eval_metric': 'rmse'
}

# Train the XGBoost model
bst = xgb.train(params, dtrain, num_boost_round=100)

# Make predictions on the test data
y_pred = bst.predict(dtest)

# Calculate and print the mean squared error of the predictions
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.4f}")

# Optional: Save the model
bst.save_model('xgboost_model.json')

# Optional: Load the model and make predictions again
bst_loaded = xgb.Booster()
bst_loaded.load_model('xgboost_model.json')
y_pred_loaded = bst_loaded.predict(dtest)
```

3.3.3 LightGBM (Light Gradient Boosting Machine)

LightGBM is a gradient boosting framework that improves both speed and accuracy, particularly on large datasets. It achieves this through several key innovations:

1. **Gradient-based One-Side Sampling (GOSS)**: To reduce the cost of computing information gain, GOSS retains all instances with large gradients (as they carry more learning signal) and randomly samples from those with small gradients. This ensures that the model still learns from the most informative samples while saving computational resources.

2. **Exclusive Feature Bundling (EFB)**: Many sparse features in high-dimensional data are mutually exclusive, meaning they rarely take non-zero values simultaneously. EFB exploits this by bundling such features into a single feature without introducing conflict, significantly reducing the number of effective features and improving training efficiency.

3. **Leaf-wise Tree Growth (Best-first)**: Unlike traditional level-wise growth used in algorithms like XGBoost, LightGBM grows trees by always splitting the leaf with the largest potential reduction in loss (delta loss). This leaf-wise strategy often leads to deeper trees with fewer splits, improving accuracy, especially on complex datasets. However, it may risk overfitting if not properly regularized.

These techniques make LightGBM particularly well-suited for high-dimensional and large-scale data, offering fast training speed, lower memory usage, and better performance compared to conventional boosting methods.

3.3.4 CatBoost (Categorical Boosting)

CatBoost is a gradient boosting algorithm that excels in handling categorical features and mitigating prediction shift caused by target leakage. Its innovations include:

1. **Ordered Target Statistics**: Traditional methods of encoding categorical features, such as target encoding, often suffer from target leakage—where information from the target variable "leaks" into the model. CatBoost solves this by using an ordering technique. For each data point, it calculates the target statistic (e.g., mean target value for a category) using only the data points that precede it in a random permutation. This ensures that future information is not used to encode current data, reducing overfitting and making the model more robust.

2. **Oblivious Decision Trees**: CatBoost builds symmetric trees, also known as oblivious trees, where all nodes at the same depth use the same splitting condition (feature and threshold). This leads to more compact and regularized models. Oblivious trees are faster to evaluate and have improved generalization, especially on small or noisy datasets.

3. **Symmetric Trees**: Because of the fixed structure of oblivious trees, CatBoost models are deterministic with respect to data ordering. This means the training is unaffected by the order of samples, improving consistency and reducing variance between runs.

4. **Efficient Categorical Feature Handling**: CatBoost can natively handle categorical features without requiring extensive preprocessing. It automatically converts categorical values into numerical representations through statistical methods, avoiding manual encoding like one-hot or label encoding.

Thanks to these techniques, CatBoost is particularly strong when working with datasets that contain a large number of categorical features, and it often achieves high accuracy with minimal feature engineering or tuning.

3.4 Stacking

Stacking is an ensemble learning technique that combines the predictions of multiple base models using a **meta-learner**, aiming to leverage the strengths of different models to improve overall performance.

3.4.1 Idea

1. Train multiple base models (which can be of different types) on the training data.
2. Use cross-validation to generate out-of-fold predictions from each base model. These predictions form a new feature space.
3. Train a meta-model (also called a **blender** or **second-level model**) on this new feature space.

3.4.2 Mathematical Formulation

For K base models, the stacking process is:

- Train base models $h_k(\mathbf{x})$ for $k = 1, 2, \ldots, K$
- Generate predictions $p_{ik} = h_k(\mathbf{x}_i)$ for each sample i
- Construct a new feature vector $\mathbf{p}_i = (p_{i1}, p_{i2}, \ldots, p_{iK})$
- Train a meta-model $g(\mathbf{p}_i)$ on these vectors
- Final prediction: $\hat{y}_i = g(\mathbf{p}_i)$

3.4.3 Difference Between Bagging and Stacking

- **Model Diversity:**
  - **Bagging** (e.g., Random Forest) typically uses the same type of base model (e.g., decision trees) and trains them on different bootstrap samples.
  - **Stacking** allows the use of heterogeneous models (e.g., combining SVMs, decision trees, and neural networks).
- **Combination Method:**
  - **Bagging** combines predictions by averaging (for regression) or voting (for classification).

  - **Stacking** learns how to combine predictions using a trainable meta-model.

- **Data Usage:**
  - **Bagging** uses different subsets of data for training each model through bootstrap sampling.
  - **Stacking** uses cross-validation to prevent overfitting and generate out-of-fold predictions for meta-model training.

- **Goal:**
  - **Bagging** reduces variance by averaging noisy models.
  - **Stacking** aims to improve predictive power by learning how to best combine diverse model outputs.

3.4.4 Implementation

Stacking for Multiclass Classification Implementation in Python

```
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict,
    train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
import numpy as np

# Load the iris dataset
iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=42)

# Base learners
base_learners = [
    ('rf', RandomForestClassifier(n_estimators=100, random_state
    =42)),
    ('gb', GradientBoostingClassifier(n_estimators=100,
    random_state=42)),
    ('svc', SVC(probability=True, random_state=42))
]

# Generate cross-validation predictions for each base learner
X_meta_train = np.zeros((X_train.shape[0], len(base_learners) * 3)
    )  # 3 classes
X_meta_test = np.zeros((X_test.shape[0], len(base_learners) * 3))

for i, (name, model) in enumerate(base_learners):
    # Train on full training data and predict test data
    model.fit(X_train, y_train)
    X_meta_test[:, i*3:(i+1)*3] = model.predict_proba(X_test)

    # Cross-validation predictions for training meta-features
    cv_proba = cross_val_predict(model, X_train, y_train, cv=5,
    method='predict_proba')
    X_meta_train[:, i*3:(i+1)*3] = cv_proba

# Train meta-learner
meta_learner = LogisticRegression(max_iter=1000, random_state=42)
meta_learner.fit(X_meta_train, y_train)

# Make predictions
y_pred = meta_learner.predict(X_meta_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Stacking Ensemble Accuracy: {accuracy:.4f}")
```

3.5 Bias-Variance Tradeoff

The bias-variance tradeoff is crucial for understanding the performance of machine learning models. It describes the relationship between a model's complexity and its ability to generalize.

When considering ensemble methods or individual models, the bias-variance decomposition can be applied to both the **real risk** and **empirical risk**:

- **Real Risk:** The real risk (expected prediction error) can be decomposed into variance, bias, and irreducible error:

$$\mathbb{E}[(y - h(\mathbf{x}))^2] = \underbrace{\text{Var}(h(\mathbf{x}))}_{\text{Variance of the predictions}} + \underbrace{[\text{Bias}(h(\mathbf{x}))]^2}_{\text{Bias } \mathbb{E}[h(\mathbf{x})]-y} + \underbrace{\sigma^2}_{\text{Irreducible error}}$$

  Note: σ^2 is the variance of the noise in the data and, it is often assumed to be independent of $\mathbf{x}$.

- **Empirical Risk:** The empirical risk (calculated from a finite sample) introduces additional bias and variance due to the randomness of the sample. The difference between empirical and real risk can be decomposed into:

$$\hat{R}_n(h) - R(h) = \underbrace{\mathbb{E}[\hat{R}_n(h)] - R(h)}_{\text{Bias due to finite sample size}} + \underbrace{\left(\hat{R}_n(h) - \mathbb{E}[\hat{R}_n(h)]\right)}_{\text{Variance due to random fluctuations of the sample}}$$

Both decompositions provide insight into how different sources of error (variance, bias, sample size) contribute to the overall prediction error, both in theoretical and empirical settings.

Ensemble methods manage this tradeoff by combining multiple models to reduce bias and variance.

- **Bagging:**
  - **Bias:** Bagging does not significantly change the bias of individual base learners, often maintaining or slightly reducing it.
  - **Variance:** It reduces variance by averaging over multiple models, preventing overfitting.
  - **Effect:** Bagging helps when base models have small bias but high variance (e.g., decision trees).
- **Boosting:**
  - **Bias:** Boosting reduces bias by sequentially focusing on the errors of previous models.
  - **Variance:** Boosting can reduce variance, but excessive rounds may increase it due to overfitting.
  - **Effect:** Boosting reduces both bias and variance if properly tuned, but overfitting may occur with too many iterations.

- **Stacking:**
  - **Bias:** Stacking can reduce bias by combining different models that correct each other's weaknesses.
  - **Variance:** Variance reduction depends on the diversity of the base models. Highly correlated base models may not reduce variance effectively.
  - **Effect:** Stacking improves both bias and variance when diverse models are used.

Managing the Bias-Variance Tradeoff

- **Bagging:** Primarily reduces variance by using high-variance base models (e.g., decision trees).
- **Boosting:** Requires tuning of hyperparameters (e.g., number of iterations, learning rate) to balance bias and variance.
- **Stacking:** Depends on selecting diverse base models and an appropriate meta-model to balance the tradeoff.

In conclusion, ensemble methods effectively manage the bias-variance tradeoff, offering a way to improve performance by combining multiple models. Careful selection and tuning are essential to avoid overfitting or underfitting.

3.6 Summary of Ensemble Methods

Ensemble methods combine multiple models to achieve performance that surpasses any individual model. These techniques manage the bias-variance tradeoff effectively:

- **Bagging** techniques like Random Forests reduce variance through randomization and averaging, making them excellent for high-variance base learners.
- **Boosting** algorithms like AdaBoost, Gradient Boosting, and XGBoost reduce bias by sequentially focusing on difficult examples, creating powerful models even from weak learners.
- **Stacking** leverages the strengths of diverse models by combining their predictions through a meta-learner.

The choice of ensemble method depends on the specific problem characteristics, available computational resources, and the nature of the base learners. Modern implementations continue to push the state-of-the-art in machine learning performance across various domains.

4 Prototype Methods and Nearest Neighbors

4.1 KNN (K-Nearest Neighbors) Classification

Not yet implemented

4.2 K-Means Clustering

Not yet implemented

4.3 Hierarchical Clustering

Not yet implemented

4.4 DBSCAN

Not yet implemented

5 Dimensionality Reduction

5.1 Principal Component Analysis (PCA)

Not yet implemented

5.2 t-Distributed Stochastic Neighbor Embedding (t-SNE)

Not yet implemented

5.3 Linear Discriminant Analysis (LDA)

Not yet implemented

6 Model Evaluation

6.1 Cross-Validation

Not yet implemented

6.2 Confusion Matrix

Not yet implemented

6.3 ROC and AUC

Not yet implemented

6.4 Precision, Recall, F1 Score

Not yet implemented