MDS 552 - Applied Machine Learning

by
Rajan Adhikari

July 18, 2025

Before We Begin

Course Overview

Laboratory Works

The practical component of this course is crucial for developing hands-on skills in machine learning. Students will implement various algorithms using high-level programming languages, with a preference for Python and the Scikit-Learn library due to their widespread use in the industry.

Key Focus Areas:

- Implementation of supervised learning algorithms (e.g., linear regression, decision trees)
- Implementation of unsupervised learning models (e.g., k-means clustering, PCA)
- Implementation of reinforcement learning algorithms
- Exploration of deep learning architectures (CNN, RNN) from scratch

Example Laboratory Task:

Implement a k-nearest neighbors (KNN) classifier from scratch to classify iris flowers based on their sepal and petal measurements. Compare your implementation with Scikit-Learn's KNN classifier in terms of accuracy and performance.

```python
import numpy as np
from collections import Counter

class KNNClassifier:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predictions = [self._predict(x) for x in X]
        return np.array(predictions)

    def _predict(self, x):
        distances = [np.sqrt(np.sum((x - x_train)**2)) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

# Usage
knn = KNNClassifier(k=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
```

Assignments

Unit 2: Supervised Learning

1. **Linear Regression**
   a. Implement linear regression from scratch using batch gradient descent on a toy dataset.
   b. Apply linear regression using Scikit-Learn on a real-world dataset. Compare with the scratch implementation.
2. **Gradient Descent Techniques**
   a. Implement stochastic gradient descent for linear regression on a toy dataset. Compare with batch gradient descent.
3. **Locally Weighted Regression**
   a. Implement locally weighted regression from scratch on a toy dataset. Visualize and discuss results.
4. **Logistic Regression and Classification**
   a. Implement logistic regression from scratch for binary classification on a toy dataset.
   b. Apply logistic regression using Scikit-Learn on a real-world dataset. Evaluate with confusion matrix, precision-recall, and ROC curves.
5. **Linear Classifiers**
   a. Implement a linear support vector machine (SVM) classifier from scratch on a toy dataset.
   b. Use Scikit-Learn to implement SVM for multi-class classification on a real-world dataset. Compare with the scratch implementation.
6. **K-Nearest Neighbors**
   a. Implement a K-nearest neighbors (KNN) classifier from scratch on a toy dataset.
   b. Apply KNN using Scikit-Learn on a real-world dataset. Evaluate performance with different values of K.
7. **Decision Trees and Random Forest**

a. Implement a decision tree classifier from scratch on a toy dataset.

b. Use Scikit-Learn to implement a random forest classifier on a real-world dataset. Discuss how ensemble learning improves performance.

Unit 3: Unsupervised Learning

8. **K-Means Clustering**

   a. Implement K-means clustering from scratch on a toy dataset.

   b. Apply K-means clustering using Scikit-Learn on a real-world dataset. Evaluate clustering results and discuss choice of K.

9. **Density Based Clustering: DBSCAN**

   a. Implement DBSCAN clustering from scratch on a toy dataset.

   b. Apply DBSCAN using Scikit-Learn on a real-world dataset. Evaluate and compare clustering performance with K-means.

10. **Principal Component Analysis (PCA)**

    a. Implement PCA from scratch on a toy dataset to reduce dimensionality.

    b. Use Scikit-Learn to apply PCA on a high-dimensional real-world dataset. Visualize and interpret principal components.

11. **Outlier Detection**

    a. Implement an outlier detection method from scratch using clustering approaches on a toy dataset.

    b. Apply outlier detection using Scikit-Learn on a real-world dataset. Evaluate and discuss results.

Unit 4: Model Evaluation and Selection

12. **Model Evaluation Metrics**

    a. Implement confusion matrices and basic evaluation metrics (accuracy, precision, recall, F1-score) from scratch.

    b. Use Scikit-Learn to evaluate classifier decision functions and generate precision-recall and ROC curves on a real-world dataset.

13. **Cross-Validation**

    a. Implement k-fold cross-validation from scratch to evaluate a classifier's performance on a toy dataset.

    b. Apply k-fold cross-validation using Scikit-Learn on a real-world dataset. Discuss advantages of cross-validation in model evaluation.

Unit 5: Reinforcement Learning

14. **Markov Decision Process (MDP)**

    a. Implement MDP and define value and policy functions from scratch.

    b. Apply value iteration and policy iteration algorithms on a toy problem.

15. **Reinforcement Learning Applications**

    a. Implement a reinforcement learning algorithm (e.g., Q-learning) from scratch to solve a toy problem.

    b. Apply reinforcement learning using a library like OpenAI Gym on a real-world problem. Discuss challenges and adaptations required.

Unit 6: Neural Network and Deep Learning

16. **Neural Networks Basics**

    a. Implement a feed-forward neural network from scratch using numpy on a toy dataset.

    b. Use Scikit-Learn or TensorFlow/Keras to implement a multi-layer neural network on a real-world dataset. Compare performance with scratch implementation.

17. **Convolutional Neural Networks (CNNs)**

    a. Implement a basic CNN architecture from scratch for image classification on a toy dataset.

    b. Use TensorFlow/Keras to implement a CNN for image classification on a real-world dataset (e.g., MNIST or CIFAR-10). Evaluate and compare performance metrics.

18. **Recurrent Neural Networks (RNNs)**

    a. Implement a simple RNN from scratch using numpy for sequence prediction on a toy dataset.

    b. Use TensorFlow/Keras to implement RNN for text processing on a real-world dataset (e.g., sentiment analysis or text generation).

Course References

- Machine Learning Specialization by Andrew Ng - YouTube
- Machine Learning — Andrew Ng — Full Course — Stanford University - YouTube
- Machine Learning (Stanford) - YouTube
- Machine Learning Playlist - YouTube
- ”Pattern Recognition and Machine Learning” by Christopher M. Bishop
- ”Deep Learning” by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

Contents

Chapter 1

Introduction to Machine Learning

Machine learning is a branch of artificial intelligence that focuses on creating systems that can learn and improve from experience without explicit programming. The field has evolved significantly since its inception.

- **1959:** Arthur Samuel coined the term "Machine Learning," defining it as the "field of study that gives computers the ability to learn without being explicitly programmed."
- **1997:** Tom M. Mitchell provided a more formal definition: "A computer program is said to learn from experience E with respect to some class of tasks T and a performance measure P if its performance in tasks T, as measured by P, improves with experience E."

Example: Consider an email spam filter. The task T is classifying emails, the experience E is the training data of labeled emails, and the performance measure P is the accuracy of the classification.

Types of Machine Learning

1.1 Supervised Learning

In supervised learning, the algorithm learns from labeled data, where both input features and target outputs are provided. The goal is to learn a mapping from inputs to outputs that can generalize well to unseen data.

Key Characteristics

- **Uses labeled training data:** The training dataset includes input-output pairs, where the input features are associated with corresponding target labels.
- **Clear feedback on predictions during training:** The algorithm receives feedback on its predictions, allowing it to adjust and improve over time.
- **Objective is to learn a mapping from inputs to outputs:** The main aim is to model the relationship between the input features and the target labels, enabling accurate predictions on new, unseen data.

1.1.1 Types of Supervised Learning

Supervised learning can be broadly categorized into two main types: regression and classification.

1. **Regression**

   Regression algorithms are used to predict continuous values. They model the relationship between the input features and the target variable, which is a continuous value.

   - **Example:** Predicting house prices based on features like size, location, and age. The goal is to learn a function that maps input features to the continuous target variable (house price).

2. **Classification**

   Classification algorithms are used to categorize data into discrete classes. The target variable consists of discrete labels, and the objective is to assign the correct label to each input instance.

   - **Example:** Classifying emails as spam or not spam. The goal is to learn a function that maps input features to discrete classes (spam or not spam).

Common Algorithms : Several algorithms are commonly used in supervised learning, each with its own strengths and applications:

- **Linear Regression:** Used for predicting continuous values by modeling the linear relationship between input features and the target variable.
- **Logistic Regression:** Used for binary classification problems, modeling the probability that an instance belongs to a particular class.
- **Decision Trees:** Tree-based models that split the data into subsets based on feature values, used for both regression and classification.
- **Random Forests:** An ensemble method that combines multiple decision trees to improve prediction accuracy and control overfitting.
- **Support Vector Machines (SVM):** Used for classification and regression, SVMs find the hyperplane that best separates the classes in the feature space.
- **Neural Networks:** Complex models that mimic the human brain, capable of learning complex patterns and representations for both regression and classification tasks.

Applications : Supervised learning algorithms have a wide range of applications across different fields:

- **Credit scoring in financial institutions:** Predicting the creditworthiness of individuals based on their financial history and other relevant features.
- **Disease diagnosis in healthcare:** Classifying medical images or patient data to diagnose diseases, improving early detection and treatment.
- **Sentiment analysis in social media monitoring:** Analyzing social media posts to determine the sentiment (positive, negative, neutral) expressed by users.
- **Facial recognition in security systems:** Identifying and verifying individuals based on their facial features for security and authentication purposes.

1.2 Unsupervised Learning

Unsupervised learning algorithms work with unlabeled data, attempting to find inherent structures or patterns. Unlike supervised learning, where the model is trained on labeled data, unsupervised learning explores the data to find hidden patterns or groupings without any prior knowledge of the labels.

Key Characteristics

- **No labeled training data:** The algorithm works with data that does not have predefined labels or outcomes.
- **Objective is to discover hidden patterns or structures in data:** The main goal is to identify underlying structures, clusters, or associations within the dataset.
- **Often used for exploratory data analysis:** These algorithms are used to gain insights into the data and to discover new, previously unknown information.

Types of Unsupervised Learning Unsupervised learning can be broadly categorized into several types, each serving different purposes:

1. **Clustering**

   Clustering algorithms group similar data points together based on certain similarity criteria. The aim is to partition the data into clusters where data points within the same cluster are more similar to each other than to those in other clusters.

   - **Example:** Customer segmentation for targeted marketing. By clustering customers based on their purchasing behavior, businesses can create targeted marketing strategies for different customer segments.

2. **Dimensionality Reduction**

   Dimensionality reduction techniques reduce the number of features in the dataset while preserving as much important information as possible. This is useful for simplifying models, reducing computation time, and helping in visualizing high-dimensional data.

   - **Example:** Compressing images while retaining key visual information. Techniques like PCA can reduce the dimensionality of image data, making storage and processing more efficient without losing significant details.

3. **Anomaly Detection**

   Anomaly detection algorithms identify data points that are significantly different from the majority of the data. These anomalies can indicate rare but important events or errors.

   - **Example:** Detecting fraudulent transactions in banking systems. Anomaly detection can identify transactions that deviate significantly from normal behavior, which may indicate fraud.

Common Algorithms : Several algorithms are commonly used in unsupervised learning:

- **K-means Clustering:** A method that partitions data into K clusters, where each data point belongs to the cluster with the nearest mean.
- **Hierarchical Clustering:** A method that builds a hierarchy of clusters, either agglomeratively (bottom-up) or divisively (top-down).
- **Principal Component Analysis (PCA):** A dimensionality reduction technique that transforms data into a new coordinate system where the greatest variances come to lie on the first coordinates (principal components).
- **t-SNE (t-Distributed Stochastic Neighbor Embedding):** A non-linear dimensionality reduction technique well-suited for embedding high-dimensional data for visualization in a low-dimensional space.
- **Autoencoders:** Neural network models used for learning efficient representations of data, typically for the purpose of dimensionality reduction or feature learning.

Applications : Unsupervised learning algorithms have a wide range of applications across different fields:

- **Market basket analysis in retail:** Identifying patterns in purchase behavior to understand product associations and customer preferences.
- **Topic modeling in natural language processing:** Discovering the underlying topics present in a collection of documents.
- **Anomaly detection in manufacturing quality control:** Detecting defective products or unusual patterns in the production process.
- **Recommendation systems in e-commerce and streaming platforms:** Grouping users with similar behaviors to provide personalized recommendations.

1.3 Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment. Unlike supervised learning, where the model learns from labeled examples, or unsupervised learning, where it finds patterns in unlabeled data, RL learns through a process of trial and error, guided by rewards or penalties.

Elements of RL : Following are the key components of RL:

- **Agent:** The learner or decision-maker.
- **Environment:** The world in which the agent operates.
- **State:** The current situation of the agent in the environment.
- **Action:** A move the agent can make.
- **Reward:** Feedback from the environment, indicating the desirability of the action.
- **Policy:** The strategy the agent employs to determine the next action.

- **Value Function:** The expected long-term return with discount, as opposed to the short-term reward.
- **Q-function:** Similar to the value function, but takes an additional parameter of the current action.

The RL Process : Understanding the step-by-step process of reinforcement learning is crucial for grasping how RL algorithms operate. The following outlines the typical RL process:

- **Step 1: Observation** The agent observes the current state of the environment.
  - Example: In a chess game, the agent observes the current position of all pieces on the board.
- **Step 2: Action Selection** Based on this state, the agent chooses an action according to its policy.
  - Example: The chess AI decides to move a particular piece to a new position.
- **Step 3: Environment Transition** The environment transitions to a new state as a result of the agent's action.
  - Example: The chess board is updated with the new position of the moved piece.
- **Step 4: Reward Allocation** The environment provides a reward to the agent.
  - Example: The chess AI might receive a small positive reward for capturing an opponent's piece, or a large reward for winning the game.
- **Step 5: Policy Update** The agent uses this information (new state and reward) to update its policy and value estimates.
  - Example: The chess AI updates its understanding of which moves are beneficial in certain board positions based on the rewards received.
- **Step 6: Iteration** Steps 1-5 repeat, with the agent continuously improving its policy through multiple episodes of interaction with the environment.
  - Example: The chess AI plays many games, continuously refining its strategy based on the outcomes of each game.

Key Considerations:

- **Exploration vs. Exploitation:** During the action selection step, the agent must balance between exploring new actions (to potentially discover better strategies) and exploiting known good actions (to maximize immediate rewards).
- **Delayed Rewards:** In many RL scenarios, the true value of an action may not be immediately apparent. The agent must learn to associate actions with delayed rewards that may come several steps later.
- **State Representation:** The way the state is represented can significantly impact the agent's ability to learn. Good state representations capture all relevant information while remaining computationally manageable.

- **Policy Representation:** The policy can be represented in various ways, from simple lookup tables to complex neural networks, depending on the complexity of the problem.

Understanding this process is fundamental to implementing and troubleshooting RL algorithms. As students progress through the course, they will explore various algorithms that implement this basic process in different ways, optimizing for different aspects of the learning problem.

Common Algorithms

- **Q-Learning:** A model-free algorithm that learns the value of an action in a particular state.
- **SARSA (State-Action-Reward-State-Action):** Similar to Q-Learning, but uses the current policy to choose the next action.
- **Deep Q-Network (DQN):** Combines Q-Learning with deep neural networks to handle high-dimensional state spaces.
- **Policy Gradient Methods:** Directly optimize the policy without using a value function.
- **Actor-Critic Methods:** Combine value function approximation with policy optimization.

Common Examples :

- **Grid World:** Consider a simple grid world where an agent must navigate from a start position to a goal, avoiding obstacles. The state is the agent's position, actions are movements (up, down, left, right), and rewards are given for reaching the goal (+1) or penalties for hitting obstacles (-1).
- **Maze Navigation:** Consider a robot learning to navigate a maze. The state is the robot's current position, actions are movements (left, right, forward, backward), rewards are given for reaching the goal or penalties for hitting walls, and the policy is the strategy the robot develops to efficiently navigate the maze.
- **Stock Trading:** An AI agent learning to trade stocks. The state is the current market conditions (stock prices, economic indicators), actions are buy, sell, or hold decisions for various stocks, rewards are the profits or losses made from trades, and the policy is the trading strategy that maximizes long-term returns.
- **Game of Chess:** An AI learning to play chess. The state is the current board configuration, actions are the possible moves, rewards are given for winning (+1), drawing (0), or losing (-1) the game, and the policy is the strategy for selecting moves that lead to victory.
- **Traffic Light Control:** An intelligent traffic management system learning to optimize traffic flow. The state is the current traffic density at various intersections, actions are adjusting traffic light timings, rewards are based on reduced wait times and increased throughput, and the policy is the strategy for coordinating lights to minimize overall traffic congestion.
- **Energy Management:** A smart home system learning to optimize energy usage. The state includes current energy consumption, time of day, and occupancy status. Actions involve turning devices on/off or adjusting thermostat settings. Rewards are based on energy savings while maintaining comfort levels. The policy determines how to balance energy efficiency with user comfort preferences.

- **Robotic Arm Control:** A robotic arm learning to pick and place objects. The state is the current position and orientation of the arm and target object. Actions are the joint movements of the arm. Rewards are given for successfully grasping and placing objects, with penalties for dropping or misplacing them. The policy determines the sequence of movements for efficient and accurate object manipulation.

Applications : Some of the real world applications includes:

- **Game Playing:** RL has achieved superhuman performance in games like Go (AlphaGo) and chess (AlphaZero).
- **Robotics:** Learning complex motor skills and navigation in uncertain environments.
- **Autonomous Vehicles:** Developing driving policies that can handle diverse traffic scenarios.
- **Resource Management:** Optimizing resource allocation in computer systems or power grids.
- **Recommender Systems:** Personalizing content recommendations on platforms like Netflix or YouTube.
- **Finance:** Developing trading strategies and portfolio management.

Reinforcement Learning represents a powerful paradigm for developing autonomous systems that can learn and adapt to complex, dynamic environments. As research in this field continues to advance, we can expect to see increasingly sophisticated RL applications across various domains, from robotics and autonomous vehicles to personalized medicine and smart city management.

Chapter 2

Supervised Learning

2.1 Linear Regression

Linear regression is a fundamental statistical and machine learning technique used to model the relationship between variables. It's a simple yet powerful algorithm widely employed in data science and predictive analytics.

Before we jump into mathematical exploration let's setup the notation we will be using throughout this book. We will denote the input variables by x_j, which are also known as input features. The output or target variable we aim to predict will be denoted by y. Each pair $(X^{(i)}, y^{(i)})$ is referred to as a training example. The dataset comprising m training examples, denoted as $\{(X^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, is called the training set. Note that the superscript (i) is merely an index within the training set and does not represent exponentiation.

2.1.1 Simple Linear Regression

Simple Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data. In this, there's a single input variable (x) and a single output variable (y). One variable is considered explanatory (Independent Variables or Predictors), while the other is dependent (Response Variable) . For instance, a researcher might use linear regression to relate individuals' weights to their heights. The relationship between these variables is represented by a straight line (linear line), often referred to as the "best fit line" or "regression line."
This line is described by the equation:

$$y = w_0 + w_1 x$$

Where:

- y is the dependent variable
- x is the independent variable
- w_0 is the y-intercept
- w_1 is the slope of the line

Using simple linear regression, we can answer the following questions:

- Is there a relationship between dependent variable and Independent Variables?

- How strong is the relationship between dependent and Independent variables?
- How is the association between dependent and Independent variables? This gives the individual contribution towards the value of dependent variable.
- How accurately can we predict the independent variables with unseen data?
- Is the relationship linear?

For example, Given a simple linear regression model to predict a person's weight (Weight) based on their height (Height), the relationship can be expressed with the following equation:

$$\text{Weight} = w_0 + w_1 \cdot \text{Height} + \epsilon$$

where:

- Weight is the dependent variable, representing the weight of the person.
- Height is the independent variable, representing the height of the person.
- w_0 is the intercept term, indicating the predicted weight when height is zero.
- w_1 is the slope coefficient, representing the average change in weight for each one-centimeter increase in height.
- ϵ is the error term, accounting for the variation in weight not explained by height.

We can answer the above questions using the given relations as:
1. **Is there a relationship between dependent and independent variables?** Yes, if the slope w_1 is significantly different from zero, it indicates that there is a relationship between height and weight. For instance, if $w_1 = 0.5$, it suggests that an increase in height by one centimeter is associated with an increase in weight by 0.5 kilograms.
2. **How strong is the relationship between dependent and independent variables?** The strength of the relationship can be assessed by the magnitude of w_1 and the coefficient of determination (R^2). A larger absolute value of w_1 or a higher R^2 indicates a stronger relationship.
3. **How is the association between dependent and independent variables?** The slope w_1 quantifies the association between height and weight. For example, if $w_1 = 0.5$, it means that for each additional centimeter in height, the weight increases by 0.5 kilograms on average.
4. **How accurately can we predict the dependent variable with unseen data?** The accuracy of predictions on unseen data can be evaluated using metrics such as the mean squared error (MSE) or the root mean squared error (RMSE) on a test dataset. Lower values of these metrics indicate more accurate predictions.
5. **Is the relationship linear?** The model assumes a linear relationship between height and weight, as expressed by the equation Weight $= w_0 + w_1 \cdot$ Height $+ \epsilon$. A linear relationship implies that the change in weight is proportional to the change in height.

Thus, the goal of simple linear regression is to find the optimal values for the intercept (w_0) and slope (w_1) that minimize the difference between predicted and actual values. Simple linear regression provides a way to understand and quantify the relationship between height and weight, allowing us to make predictions and assess the strength and nature of this relationship in a straightforward manner.

Ordinary Least Square (OLS)

OLS is an analytical approach that finds the best-fitting line by minimizing the sum of the squares of the residuals. It provides a closed-form solution for the optimal parameters. But it is limited to only few parameters particularly we use it only if there is single independent variables i.e. simple linear regression.
Given x and y in the data D, using Ordinary Least Square (OLS) method, we can solve the relationship between independent and dependent variable and determine the coefficients w_0 and w_1 as:

$$\hat{w}_1 = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{m}(x_i - \bar{x})^2},$$

$$\hat{w}_0 = \bar{y} - \hat{w}_1\bar{x},$$

**Derivation of OLS is out of scope for the given syllabus*
However, in the real world, we won't be limited to single independent variable. So, let's focus in the scenarios where there are multiple independent variables.

2.1.2 Derivation of Multiple Linear Regression

Consider a scenario where we are interested to predict the price of house in Kathmandu, which can determined through knowing number of features that influence the house price, for example - no. of bedrooms, location, no. of floors, no. of bedrooms etc. This can be represented in the table as:
We can write the relationship of independent variables $x_1, x_2, ..., x_n$, collectively X *(as matrix*

Table 2.1: Sample Data for House Price Prediction

price (y)	no_of_bedrooms (x_1)	no_of_rooms (x_2)	...	location (x_n)
y_1	x_{11}	x_{12}	.	x_{1n}
.	.	.	.	.
.	.	.	.	.
y_m	x_{m1}	x_{m2}	.	x_{mn}

of $m \times n$ size) and dependent variable y in the form of linear equation as:

$$y = w_0 + w_1x_1 + w_2x_2... + w_nx_n \tag{2.1}$$

which means, each value of y can be obtained from the corresponding value of $x_1, x_2, ..., x_n$ for each observation. We denote each observation i.e. i-th observation by $(\mathbf{x}^{(i)}, y^{(i)})$, where, $i = 1, 2, ..., m$ as we have m observations in our dataset.

We may obtain the true value of w_j, such that, $i = 1, 2, 3, ..n$ directly by solving the equation through linear system of equations but m and n can be very very large that the computation is too costly or almost computationally inefficient. Thus, we need better method for obtaining the value of w_j. We will discuss about it shortly.

Before that, say, we have obtained the value of w_i, and with that we can compute the value of y for given x. For the sake of comparison, we keep apart the value obtained from the equation 2.2, using the value of w_i, from the actual value y . We call this predicted value which is denoted using $\hat{y}$ and is written as:

$$\hat{y} = w_0 + w_1x_1 + w_2x_2... + w_nx_n \tag{2.2}$$

Now, the difference of actual value and predicted value means the error signifying the gap between them which we call a error. This error has to be minimum or almost 0 to accurately define the relationship.

Residuals

In regression analysis, the difference between the actual value (y) and the predicted value ($\hat{y}$) is called the residual or error. Mathematically, we can express this as:

$$e = \hat{y} - y$$

and total error for all observations as:

$$TotalError = \sum_{i=1}^{m} e^{(i)} \tag{2.3}$$

where, i in $(e)^{(i)}$ refers to the error of i-th observation.

We try to minimize the value of **Total Error** and keep it near to 0 as much as possible. But, it is hard to have a grasp on smaller value of $e^{(i)}$ as it can be very small while error approach close to 0 and sometimes the error might be negative or positive. This creates another problem that there's chance a negative error value of one observation may cancel the positive value of another observation and total error results to 0 despite being the presence of error on our prediction.

Thus, to avoid the case of negative error canceling the positive error, we may take absolute value of error so that all errors are positive. Then, total error becomes:

$$TotalError = \sum_{i=1}^{m} |e^{(i)}| \tag{2.4}$$

and this is known as **Total Absolute Error**.

Alternatively, we can square each error and take the sum of them. The sum of these residuals, squared to avoid negative values canceling out positive ones, is called the Residual Sum of Squares (RSS):

$$RSS = {e^{(1)}}^2 + {e^{(2)}}^2 + \ldots + {e^{(m)}}^2 = \sum_{i=1}^{m} {e^{(i)}}^2$$

Where m is the number of observations.

To evaluate the model's performance, we often use the Mean Squared Error (MSE) cost function, which is the average of squared errors:

$$MSE = \frac{1}{m}\sum_{i=1}^{m} {e^{(i)}}^2 = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2 \tag{2.5}$$

Gradient Descent

To minimize the error, we will use the optimization technique known as Gradient Descent. Gradient Descent is an iterative optimization algorithm that finds the minimum of a function by taking steps proportional to the negative of the gradient of the function at the current point. In the context of linear regression, we use it to minimize the cost function (MSE).

The update rule for the gradient descent is given as:

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(w) \tag{2.6}$$

Where α is the learning rate and $J(w)$ is the cost function (*Cost function refers to the average of collective error from all training observations.*), which we try to minimize.

Linear regression aims to model the relationship between a dependent variable y and one or more independent variables x by fitting a linear equation to observed data. The equation for a multiple linear regression model is:

$$\hat{y} = h(x) = w_0 + w_1 x_1 + ... + w_n x_n$$

where:

- $\hat{y} = h(x)$ is the predicted value.
- w_0 is the bias term.
- w_j is the coefficient of j-th independent variable
- x_j is the independent variable.

Cost Function

To measure the accuracy of our model, we use a cost function, often the Mean Squared Error (MSE). Since the cost function is using MSE, hence, the rule is also known as Least Mean Square.

$$J(w_j) = \frac{1}{2m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2 \tag{2.7}$$

where:

- y_i is the actual value for the i-th training example.
- $h(x_i)$ is the predicted value for the i-th training example.

Update Rule

Let's now consider we have only one training example in our dataset for ease in derivation, We update the parameter w_j using the gradient descent update rule:

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} \frac{1}{2} (h(x) - y)^2 \tag{2.8}$$

We compute the partial derivative of the cost function with respect to each parameter w_j:

$$\begin{aligned}
\frac{\partial}{\partial w_j}\frac{1}{2}(J(w_j))^2 &= \frac{\partial}{\partial w_j}\frac{1}{2}(h(x)-y)^2 \\
&= \frac{1}{2}\frac{\partial}{\partial w_j}(h(x)-y)^2 \\
&= \frac{1}{2}\frac{\partial}{\partial (h(x)-y)}(h(x)-y)^2 . \frac{\partial}{\partial w_j}(h(x)-y) \\
&= \frac{1}{2}.2(h(x)-y)\frac{\partial}{\partial w_j}(h(x)-y) \\
&= (h(x)-y)\cdot\frac{\partial}{\partial w_j}(w_0 + w_1x_1 + \ldots + w_nx_n - y) \\
&= (h(x)-y)\cdot\frac{\partial}{\partial w_j}(w_0 + w_1x_1 + \ldots + w_jx_j + \ldots + w_nx_n - y) \\
&= (h(x)-y)\cdot x_j
\end{aligned}$$

Now, substituting the value of derivative in 2.8, we get:

$$w_j := w_j - \alpha(h(x)-y)\cdot x_j \tag{2.9}$$

for single observation in training data.
This is called **LMS update rule**, and also known as **Widrow-Hoff learning rule**.
Considering m observations in training data, we have a two scenarios - stochastic and batch gradient descent.

2.1.3 Stochastic Gradient Descent

Unlike batch gradient descent, which uses the entire dataset to compute the gradient, SGD updates the model parameters using only a single training example at a time. This can significantly speed up the training process, especially for large datasets.
The update rule is:

$$w_j = w_j + \alpha(y^{(i)} - h(x^{(i)}))\cdot x_j^{(i)}$$

This method is faster and requires less memory, but it can produce noisier gradients.
Characteristics:

1. **Single Example Utilization**: SGD uses only one training example to compute the gradient at each step, making it much faster per iteration compared to batch gradient descent.

2. **Noisy Updates**: The gradient updates in SGD are noisier due to the use of single examples, which can lead to a more erratic convergence path but can also help escape local minima.

3. **Faster Convergence**: Each update is faster because it uses only one training example, leading to quicker parameter updates and potentially faster convergence.

4. **Efficient for Large Datasets**: SGD can handle very large datasets since it does not require the entire dataset to be loaded into memory.

Algorithm 1 Stochastic Linear Regression

```
Require: Training data $X$, targets $y$, learning rate $\alpha$, number of epochs $n$
Ensure: Optimized weights $w$
 1: Initialize weights $w \leftarrow$ random small values
 2: for $epoch = 1$ to $n$ do
 3:     Shuffle training data $(X, y)$
 4:     for each $(x_i, y_i)$ in $(X, y)$ do
 5:         $prediction \leftarrow w^T \cdot x_i$
 6:         $error \leftarrow prediction - y_i$
 7:         $gradient \leftarrow error \cdot x_i$
 8:         $w \leftarrow w - \alpha \cdot gradient$
 9:     end for
10: end for
11: return $w$
```

2.1.4 Batch Gradient Descent

In batch gradient descent, we compute the gradient using the entire training set before performing each update. The update rule is:

$$w_j = w_j + \alpha \sum_{i=1}^{m} (y^{(i)} - h(x^{(i)})) \cdot x_j^{(i)}$$

This method produces stable gradients and convergence but can be computationally expensive for large datasets.

Characteristics

1. **Full Dataset Utilization**: Batch gradient descent uses the entire training set to compute the gradient of the cost function. This ensures that each step is based on comprehensive information, leading to a more accurate direction for parameter updates.
2. **Costly Computation**: Because it processes the entire dataset in each iteration, batch gradient descent can be computationally expensive and slow, especially for large datasets.
3. **Stable Convergence**: The algorithm generally converges steadily toward the global minimum of the cost function, provided the learning rate is appropriately chosen.

Thus, batch gradient descent is a robust and stable optimization technique, particularly effective for smaller datasets where the computational cost is manageable.

Pen & Paper

Given the dataset with two features and the target variable:

Observation	x_1	x_2	y
1	1	2	4
2	2	3	7
3	3	4	10

The hypothesis function for linear regression is:

$$h(x) = w_0 + w_1 x_1 + w_2 x_2$$

Algorithm 2 Batch Linear Regression

```
Require: Training data $X$, targets $y$, learning rate $\alpha$, number of iterations $n$
Ensure: Optimized weights $w$
1: Initialize weights $w \leftarrow 0$
2: $m \leftarrow$ number of training examples
3: for $i = 1$ to $n$ do
4:     $predictions \leftarrow X \cdot w$
5:     $errors \leftarrow predictions - y$
6:     $gradient \leftarrow \frac{1}{m} X^T \cdot errors$
7:     $w \leftarrow w - \alpha \cdot gradient$
8: end for
9: return $w$
```

The cost function J is:

$$J(w_0, w_1, w_2) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - h(x_i))^2$$

The gradients of the cost function with respect to w_0, w_1, and w_2 are:

$$\frac{\partial J}{\partial w_0} = -\frac{1}{m} \sum_{i=1}^{m} (h(x_i) - y_i)$$

$$\frac{\partial J}{\partial w_1} = -\frac{1}{m} \sum_{i=1}^{m} (h(x_i) - y_i) \cdot x_{1i}$$

$$\frac{\partial J}{\partial w_2} = -\frac{1}{m} \sum_{i=1}^{m} (h(x_i) - y_i) \cdot x_{2i}$$

Gradient Descent Algorithm: 1. Initialize parameters: Let's first initialize the parameters as:

$$w_0 = 0, \quad w_1 = 0, \quad w_2 = 0$$

Next, set the learning rate:

$$\alpha = 0.01$$

2. Iteration 1: We now compute the predictions:

$$h(x_i) = w_0 + w_1 x_{1i} + w_2 x_{2i}$$

$$\text{For } (x_1, x_2) = (1, 2): \quad h(x_1) = 0 + 0 \cdot 1 + 0 \cdot 2 = 0$$

$$\text{For } (x_1, x_2) = (2, 3): \quad h(x_1) = 0 + 0 \cdot 2 + 0 \cdot 3 = 0$$

$$\text{For } (x_1, x_2) = (3, 4): \quad h(x_1) = 0 + 0 \cdot 3 + 0 \cdot 4 = 0$$

- Next, let's compute the gradients:

$$\frac{\partial J}{\partial w_0} = -\frac{1}{3} \left[(0-4) + (0-7) + (0-10) \right] = \frac{21}{3} = 7$$

$$\frac{\partial J}{\partial w_1} = -\frac{1}{3} \left[(0-4) \cdot 1 + (0-7) \cdot 2 + (0-10) \cdot 3 \right] = \frac{16}{3} \approx 5.33$$

$$\frac{\partial J}{\partial w_2} = -\frac{1}{3}\left[(0-4)\cdot 2 + (0-7)\cdot 3 + (0-10)\cdot 4\right] = \frac{23}{3} \approx 7.67$$

- Finally, we need to update the parameters:

$$w_0 := w_0 - \alpha \cdot \frac{\partial J}{\partial w_0} = 0 - 0.01 \cdot 7 = -0.07$$

$$w_1 := w_1 - \alpha \cdot \frac{\partial J}{\partial w_1} = 0 - 0.01 \cdot 5.33 = -0.0533$$

$$w_2 := w_2 - \alpha \cdot \frac{\partial J}{\partial w_2} = 0 - 0.01 \cdot 7.67 = -0.0767$$

3. Iteration 2: - Compute predictions with updated parameters:

$$h(x_i) = -0.07 + (-0.0533 \cdot x_{1i}) + (-0.0767 \cdot x_{2i})$$

$$\text{For } (x_1, x_2) = (1,2): \quad h(x_1) = -0.07 - 0.0533 \cdot 1 - 0.0767 \cdot 2 = -0.277$$

$$\text{For } (x_1, x_2) = (2,3): \quad h(x_1) = -0.07 - 0.0533 \cdot 2 - 0.0767 \cdot 3 = -0.432$$

$$\text{For } (x_1, x_2) = (3,4): \quad h(x_1) = -0.07 - 0.0533 \cdot 3 - 0.0767 \cdot 4 = -0.589$$

- Compute gradients:

$$\frac{\partial J}{\partial w_0} = -\frac{1}{3}\left[(-0.277-4) + (-0.432-7) + (-0.589-10)\right] = 7.004$$

$$\frac{\partial J}{\partial w_1} = -\frac{1}{3}\left[(-0.277-4)\cdot 1 + (-0.432-7)\cdot 2 + (-0.589-10)\cdot 3\right] = 5.339$$

$$\frac{\partial J}{\partial w_2} = -\frac{1}{3}\left[(-0.277-4)\cdot 2 + (-0.432-7)\cdot 3 + (-0.589-10)\cdot 4\right] = 7.673$$

- Update parameters:

$$w_0 := w_0 - \alpha \cdot \frac{\partial J}{\partial w_0} = -0.07 - 0.01 \cdot 7.004 = -0.140$$

$$w_1 := w_1 - \alpha \cdot \frac{\partial J}{\partial w_1} = -0.0533 - 0.01 \cdot 5.339 = -0.107$$

$$w_2 := w_2 - \alpha \cdot \frac{\partial J}{\partial w_2} = -0.0767 - 0.01 \cdot 7.673 = -0.153$$

4. Iteration 3: - Compute predictions with updated parameters:

$$h(x_i) = -0.140 + (-0.107 \cdot x_{1i}) + (-0.153 \cdot x_{2i})$$

$$\text{For } (x_1, x_2) = (1,2): \quad h(x_1) = -0.140 - 0.107 \cdot 1 - 0.153 \cdot 2 = -0.587$$

$$\text{For } (x_1, x_2) = (2,3): \quad h(x_1) = -0.140 - 0.107 \cdot 2 - 0.153 \cdot 3 = -0.828$$

$$\text{For } (x_1, x_2) = (3,4): \quad h(x_1) = -0.140 - 0.107 \cdot 3 - 0.153 \cdot 4 = -1.071$$

- Compute gradients:

$$\frac{\partial J}{\partial w_0} = -\frac{1}{3}\left[(-0.587-4) + (-0.828-7) + (-1.071-10)\right] = 7.028$$

$$\frac{\partial J}{\partial w_1} = -\frac{1}{3}\left[(-0.587-4)\cdot 1 + (-0.828-7)\cdot 2 + (-1.071-10)\cdot 3\right] = 5.348$$

$$\frac{\partial J}{\partial w_2} = -\frac{1}{3}\left[(-0.587-4)\cdot 2 + (-0.828-7)\cdot 3 + (-1.071-10)\cdot 4\right] = 7.686$$

- Update parameters:

$$w_0 := w_0 - \alpha \cdot \frac{\partial J}{\partial w_0} = -0.140 - 0.01 \cdot 7.028 = -0.210$$

$$w_1 := w_1 - \alpha \cdot \frac{\partial J}{\partial w_1} = -0.107 - 0.01 \cdot 5.348 = -0.161$$

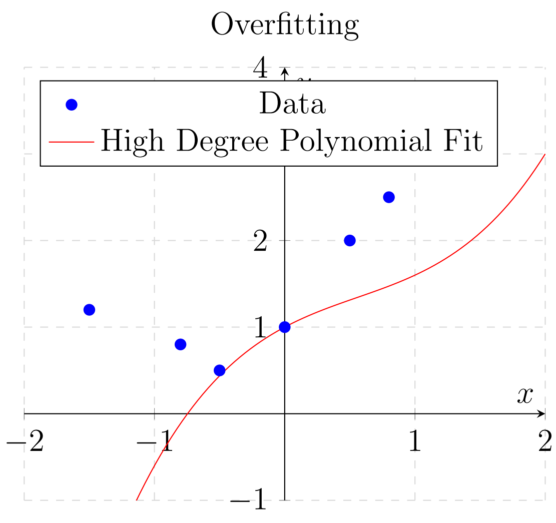
$$w_2 := w_2 - \alpha \cdot \frac{\partial J}{\partial w_2} = -0.153 - 0.01 \cdot 7.686 = -0.230$$

2.2 Overfitting and Underfitting

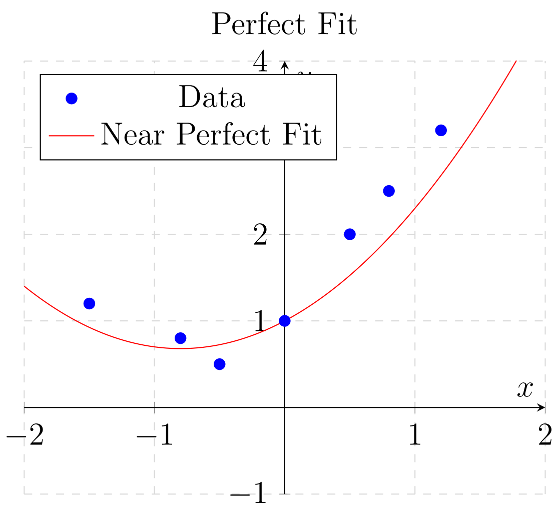
In the field of machine learning, **overfitting** and **underfitting** are two fundamental concepts that characterize the common problems encountered when training models. They describe how well a model learns the patterns in the training data and, crucially, how well it is able to **generalize** its learning to predict or classify new, unseen data points. Achieving a good balance between fitting the training data and generalizing to new data is essential for building effective machine learning models.

2.2.1 Overfitting

Overfitting occurs when a machine learning model is excessively complex relative to the amount and complexity of the training data. The model learns the training data's details and noise to such an extent that it negatively impacts its ability to generalize to new data. Essentially, the model memorizes the training examples and their noise rather than learning the underlying true relationship or distribution governing the data.
As a result, an overfitted model exhibits excellent performance (e.g., very low error or high accuracy) on the training data but performs significantly worse on unseen data, such as validation or test sets. This is because the noise and specific patterns learned from the training set are not present in the unseen data.



The diagram above illustrates overfitting in regression. A complex model (represented by the wiggly red line) passes very close to or exactly through all the training data points (blue circles), including potential noise. While its error on this specific set of points is minimal, it is unlikely to accurately capture the underlying simpler trend and will likely perform poorly on new data points that don't follow this exact path.



Key Characteristics

- High accuracy on training data
- Poor performance on validation or test data
- The model is too complex for the underlying problem
- The model captures noise in the training data as if it were a meaningful pattern

Causes

- Too many features relative to the number of training examples
- A model that is too complex (e.g., too many layers or nodes in a neural network)
- Training for too many epochs

Example: Suppose you are using a polynomial regression to fit a dataset. If you choose a very high-degree polynomial, the model might pass through all the data points exactly, resulting in a very low error on the training set. However, such a model will likely fail to predict new data points accurately because it has learned the noise in the training data.

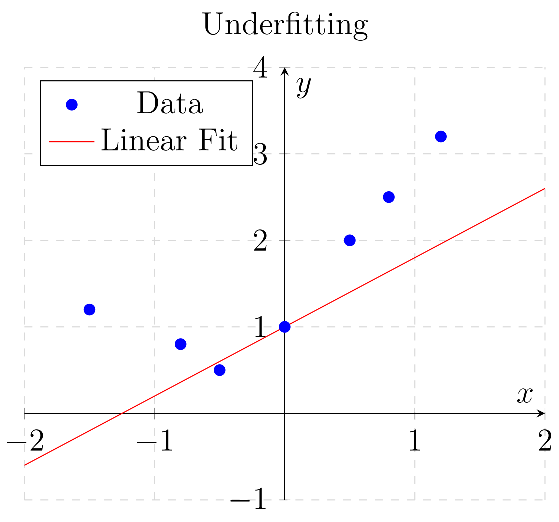
Mitigation Strategies:

1. Simplify the model: Use fewer parameters or simpler algorithms.
2. Regularization: Techniques like L1 or L2 regularization penalize large coefficients and help prevent overfitting.
3. Cross-validation: Use cross-validation to ensure that the model performs well on different subsets of the data.
4. More data: Providing more training data can help the model to generalize better.

2.2.2 Underfitting

Underfitting happens when a model is too simple to capture the underlying pattern in the data. As a result, the model performs poorly on both the training data and unseen data (test data).

Suppose we are using a linear regression to fit a dataset with a clear non-linear relationship. A linear model will not be able to capture the non-linear patterns, resulting in high errors on both the training and test sets.



Key Characteristics

- Poor accuracy on training data
- Poor performance on validation or test data
- The model is too simple to capture the complexity of the underlying problem
- The model fails to capture important patterns in the data

Causes

- Too few features or relevant features not included
- A model that is too simple (e.g., using a linear model for a non-linear problem)
- Insufficient training time or data

Mitigation Strategies

1. Increase model complexity: Use more parameters or more sophisticated algorithms to better capture the underlying pattern in the data.
2. Feature engineering: Create new features or use polynomial features to better capture the relationships in the data.
3. Parameter tuning: Adjust the parameters of the model to find a better fit.

2.3 The Bias-Variance Tradeoff

The Bias-Variance Tradeoff is a fundamental concept in machine learning that helps us understand the sources of prediction error in supervised learning models. It highlights the inherent conflict between a model's ability to simplify complex relationships (**bias**) and its sensitivity to fluctuations in the training data (**variance**).

2.3.1 The Bias-Variance Tradeoff

Understanding **overfitting** and **underfitting** is closely related to the bias-variance tradeoff. In supervised learning, the prediction error can be broken down into three main components: the bias, the variance, and the irreducible error. This relationship is mathematically expressed as:

$$Error = (Bias[h(x; D)])^2 + Var[\hat{f}(x; D)] + \sigma^2$$

You can refer to https://www.inf.ed.ac.uk/teaching/courses/mlsc/Notes/Lecture4/BiasVariance.pdf for the derivation.
Let's break down each component:

1. **Bias**: This represents the error introduced by the simplifying assumptions a learning algorithm makes. It's the difference between the average prediction of our model and the true underlying function we're trying to estimate. For instance, if you try to approximate a non-linear function $f(x)$ with a linear model, the systematic error in your predictions $h(x)$ is primarily due to these simplifying assumptions. A high bias model essentially "misses" the true relationship between features and target due to its inherent simplicity.

2. **Variance**: This quantifies how much the learning method $h(x)$ varies around its mean prediction when trained on different datasets. It reflects how sensitive the model is to the specific training data it has seen. A high variance model is very flexible and will fit the training data extremely closely, even capturing noise, leading to significant changes in predictions if the training data were slightly different.

3. **Irreducible Error**: Denoted by σ^2, this is the inherent noise in the data itself that no model, no matter how perfect, can reduce. It comes from unmeasured variables, random fluctuations, or fundamental limits of measurement. This error sets a lower bound on the expected error on a test dataset, meaning even the best possible model will still have at least this much error.

Since all three error components are non-negative, the irreducible error forms a fundamental lower bound on the expected prediction error on new, unseen data.

—

2.3.2 Bias

Bias refers to the error that arises from approximating a complex real-world problem with a simplified model. It's the consistent difference between the average prediction of our model and the actual value we're trying to predict. Think of it as a systematic error that pushes predictions in a particular direction.

High Bias: When a model has high bias, it means the model is too simplistic to capture the underlying patterns and complexities in the data. This simplicity leads to significant and systematic errors in predictions, a phenomenon commonly known as **underfitting**. A high bias model essentially makes strong assumptions about the data's structure that might not hold true. For example, trying to fit a curved relationship with a straight line will result in high bias. This is often seen in models with low complexity, such as a basic linear regression model used on data that exhibits a non-linear relationship, or a decision tree with a very limited depth. The model consistently misses the mark because its fundamental structure is inadequate for the data.

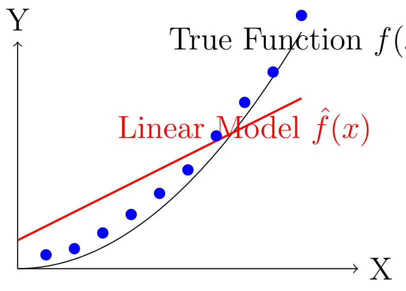


Figure 2.1: Illustration of High Bias (Underfitting). The linear model clearly fails to capture the quadratic trend of the data.

Example: Imagine you're trying to predict house prices based on their size. If the true relationship between size and price is non-linear (e.g., larger houses might have diminishing returns on price per square foot after a certain point), and you use a simple linear regression model, your model will consistently underestimate prices for very large houses and overestimate for very small ones. This consistent deviation is an example of ***high bias****. To reduce this bias, you might need a more flexible model like polynomial regression or a non-linear regression technique that can capture the curvature in the data.*

Low Bias: Conversely, low bias means the model is complex and flexible enough to capture the underlying patterns and relationships in the data. This complexity allows the model to approximate the true function more closely, reducing systematic errors. However, having low bias doesn't automatically mean the model is perfect; it simply indicates that the model's structure is capable of learning the intricacies of the data. For instance, if the true relationship is quadratic and you use a polynomial regression model of degree 2, you're likely to achieve low bias.

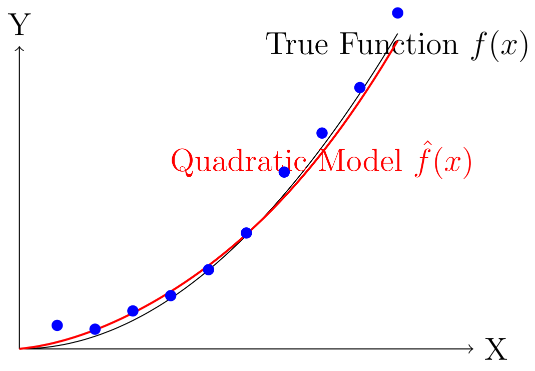


Figure 2.2: Illustration of Low Bias (Appropriate Fit). The quadratic model closely follows the true quadratic function.

2.3.3 Variance

Variance refers to how much the model's predictions would change if we used a different training dataset. In essence, it measures how scattered or inconsistent the predicted values are from the correct value due to variations in the training data. It's also known as **Variance Error** or **Error due to Variance**. A model with high variance is highly sensitive to the specific data points in its training set, even noise, and will produce very different predictions if given a slightly different training set.

High Variance: A model with high variance pays too much attention to the specifics of the training data, capturing not only the underlying true pattern but also the noise and random fluctuations present in that particular training set. This phenomenon is known as **overfitting**. As a result, such a model performs exceptionally well on the training data because it has essentially memorized it, but it performs poorly on new, unseen data. This is because it has learned the noise specific to the training set rather than the generalizable patterns. For instance, a very deep decision tree might create branches for every single outlier in the training data, leading to high variance.
Example: Consider a very complex decision tree, perhaps one that has been allowed to grow to its maximum depth without any pruning. If you train this tree on one dataset of customer behavior, it might learn very specific rules like "if a customer lives in zip code 12345, is between 30 and 35 years old, and bought a specific product last Tuesday, then they will buy product X." This level of detail makes the model very specific to that training set, including any random fluctuations or anomalies. If you were to train the same model on a slightly different dataset, even from the same population, it would likely produce drastically different and inconsistent rules and predictions, demonstrating ***high variance****. A simpler, pruned decision tree would have lower variance as it focuses on more general patterns.*

Low Variance: Low variance indicates that the model's predictions don't vary significantly when trained on different datasets drawn from the same underlying distribution. Such models are more robust and stable, as their predictions are less influenced by the specific quirks of any single training set. However, a model with low variance might be too simple to capture all the intricate patterns in the data, potentially leading to high bias.

2.3.4 Tradeoff

The interplay between bias and variance is central to model selection and performance. Ideally, we want a model that has both low bias and low variance. However, these two error sources often have an inverse relationship, forming what is known as the **bias-variance tradeoff**. As you decrease one, the other tends to increase.
Given the definitions of bias and variance, we can categorize models into four possible combinations, as depicted in Figure 2.3:
The ultimate goal in machine learning is to find an optimal balance between bias and variance to minimize the total prediction error on unseen data.

1. **Underfitting (High Bias, Low Variance):** This occurs when the model is too simple to capture the underlying trend and relationships within the data. It makes strong, often incorrect, assumptions about the data's structure, leading to systematic errors. While such a model is very stable and consistent across different datasets (low variance) because its predictions don't change much, it consistently performs poorly on both training and test data. An example would be using a linear regression model to predict highly non-linear stock market fluctuations.

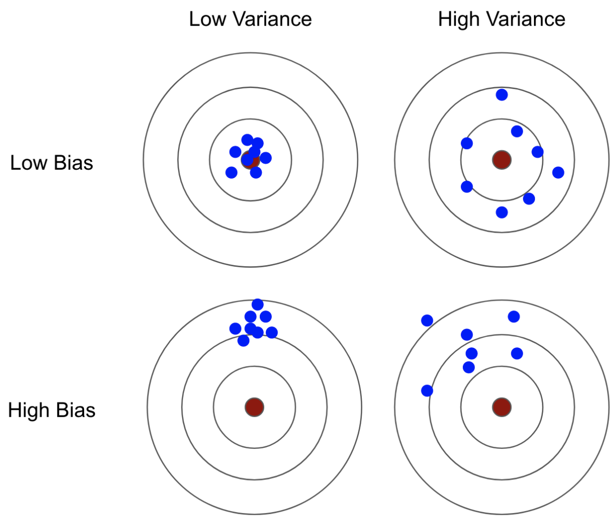


Figure 2.3: Possible combinations of Bias Variance. This visual helps categorize models based on their bias and variance characteristics.

2. **Overfitting (Low Bias, High Variance):** This happens when the model is excessively complex and flexible, allowing it to capture even the noise and random fluctuations present in the training data. This leads to a model that fits the training data exceptionally well (low bias), but it fails to generalize to new, unseen data because it has learned the specific "noise" of the training set rather than the true underlying patterns. Consequently, its predictions vary wildly with slight changes in the training data (high variance). A very deep neural network trained on a small dataset without sufficient regularization can easily overfit.

3. **Optimal Model:** The best model achieves a sweet spot where there's a good balance between bias and variance. This model is complex enough to accurately capture the significant underlying patterns in the data (thus having low bias) but not so complex that it learns the noise and overfits (thus maintaining low variance). This balance leads to the lowest overall prediction error on new, unseen data, representing robust generalization.

This graph 2.4 provides a clear visual representation of the bias-variance tradeoff. As **model complexity** increases along the x-axis (e.g., adding more features, increasing polynomial degree, deepening a neural network), we observe the following trends:

- **Bias (red curve) decreases:** Initially, with simpler models, bias is high because the model cannot capture the true underlying relationship. As complexity increases, the model becomes more flexible and capable of learning the data's intricacies, leading to a reduction in systematic errors.

- **Variance (blue curve) increases:** Conversely, as model complexity grows, the model becomes more sensitive to the specific training data points, including noise. This heightened sensitivity causes its predictions to fluctuate more significantly if trained on different datasets, leading to higher variance.

- **Total error (black curve):** This curve represents the sum of squared bias, variance, and irreducible error. On the left side of the graph, where models are simpler, we see high bias and low variance, resulting in high total error (**underfitting**). As we move right towards more complex models, we encounter a region of optimal balance where the total error is minimized. This is the "sweet spot." Beyond this point, as complexity continues

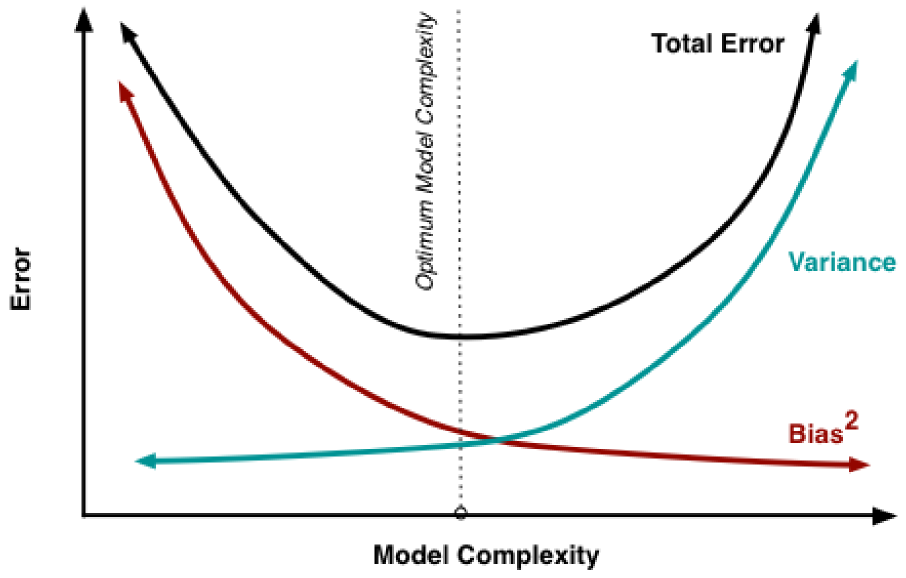


Figure 2.4: Bias Variance Tradeoff. This graph beautifully illustrates how model complexity impacts bias, variance, and the total error.

to increase, we enter the realm of **overfitting**, characterized by very low bias but rapidly increasing high variance, leading to a rise in the total error.

The graph demonstrates that the ultimate goal in model selection and tuning is to find the optimal level of complexity that minimizes total error. This optimal point represents the best compromise between the model's ability to accurately capture the underlying patterns in the training data (low bias) and its ability to generalize well to new, unseen data (low variance). Achieving this balance is crucial for building effective and reliable machine learning models.

2.4 Locally Weighted Regression

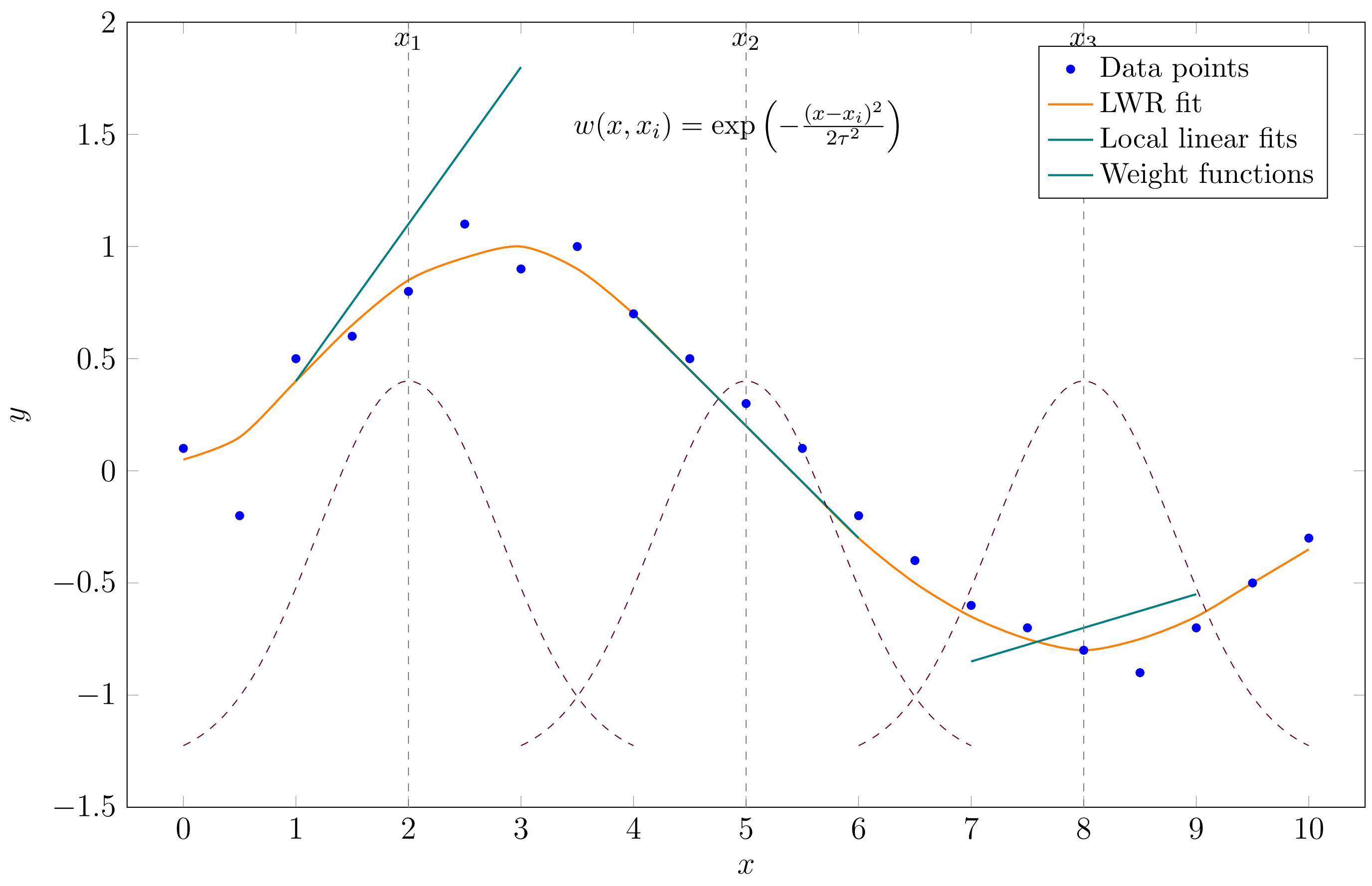
Locally weighted regression is a non-parametric method that performs regression around a point of interest using only the training data local to that point. It's particularly useful when the relationship between variables is non-linear.

The main idea behind locally weighted regression is to fit a model that's locally relevant to each point of interest, rather than trying to fit a single global model to all the data. This approach enables the model to capture local patterns effectively, making it more flexible than global linear regression. Consequently, it can capture nonlinear relationships without the need for explicitly specifying a complex model structure.

The prediction is a weighted average of points local to it, with weights decreasing as a function of the distance between x and the query point x^*:

$$eta_{ii} = \exp\left(\frac{(x - x_i)^2}{2\tau^2}\right)$$

Where τ is a bandwidth parameter that controls the size of the local neighborhood.



Here's a detailed explanation of the graph:

- Blue dots: These represent the data points, showing a non-linear pattern.
- orange curve: This shows the overall LWR fit. It smoothly follows the trend of the data points, adapting to local patterns.
- Teal lines: These represent local linear fits at three different points ($x_1 = 2, x_2 = 5, x_3 = 8$). This is a key feature of LWR - at each point, it fits a straight line to the nearby data, weighted by proximity.
- Purple dashed curves: These represent the weighting functions centered at x_1, x_2, and x_3. They're shown below the x-axis to avoid cluttering the main plot. The bell shapes illustrate how points closer to the center receive higher weights.
- Gray dashed lines: These vertical lines mark the positions x_1, x_2, and x_3, where the local linear fits and weighting functions are centered.
- Equation: The weighting function equation is displayed on the graph for reference.

But, if the linear regression can be used to learn non linear pattern, why it isn't use? While Locally Weighted Regression (LWR) is a powerful and flexible technique for non-linear regression, it does have some disadvantages compared to other non-linear learning techniques. Let's discuss these limitations and compare LWR to other methods: Disadvantages of Locally Weighted Regression:

1. Computational Complexity: LWR is computationally expensive, especially for large datasets. It requires recalculating the weights and fitting a local model for each prediction point. This makes it slower than methods that build a single global model.

2. Memory Requirements: LWR is a lazy learning algorithm, meaning it stores all training data. For large datasets, this can lead to significant memory usage.
3. Bandwidth Selection: The choice of bandwidth (or kernel width) is crucial and can significantly affect performance. Selecting the optimal bandwidth often requires cross-validation, adding to computational costs.
4. Curse of Dimensionality: LWR's performance can degrade in high-dimensional spaces. As the number of features increases, the concept of "local" becomes less meaningful.
5. Extrapolation: LWR typically performs poorly when extrapolating beyond the range of the training data.
6. Lack of a Global Model: LWR doesn't provide a compact, global model of the data. This can make it harder to interpret or extract global insights from the model.

Algorithm 3 Locally Weighted Regression using Gradient Descent

```
Require: Training data $(X, y)$, query point $x_q$, kernel function $K$, bandwidth $\tau$, learning rate $\alpha$, num_iterations, convergence threshold $\epsilon$
Ensure: Predicted value $\hat{y}_q$
 1: function LocallyWeightedRegression($X, y, x_q, K, \tau, \alpha$, num_iterations, $\epsilon$)
 2:     $m \leftarrow$ number of training observations
 3:     $n \leftarrow$ number of features
 4:     $w \leftarrow$ random$(n, 1)$                                  ▷ Initialize coefficients
 5:     for $t = 1$ to num_iterations do
 6:         $\eta \leftarrow$ diag$(0, \ldots, 0)$                      ▷ Re-initialize $\eta$ for this iteration
 7:         for $i = 1$ to $m$ do
 8:             $\eta_{ii} \leftarrow K\left(\frac{||x_i - x_q||}{\tau}\right)$          ▷ Calculate weights
 9:         end for
10:         $\nabla J \leftarrow \frac{1}{m} X^T \eta (Xw - y)$              ▷ Compute gradient
11:         $w_{\text{new}} \leftarrow w - \alpha \nabla J$                  ▷ Update coefficients
12:         if $||w_{\text{new}} - w|| < \epsilon$ then
13:             break                                              ▷ Convergence check
14:         end if
15:         $w \leftarrow w_{\text{new}}$
16:     end for
17:     $\hat{y}_q \leftarrow x_q^T w$                                  ▷ Make prediction
18:     return $\hat{y}_q$
19: end function
20: function KernelFunction($u$)
21:     return $\exp(-u^2/2)$                                      ▷ Gaussian kernel
22: end function
```

Locally weighted linear regression is an example of a **non-parametric algorithm**. The term "non-parametric" essentially means that the resources required to represent the hypothesis h increase linearly with the size of the training set i.e. the learning algorithm doesn't assume any underlying distribution of data. Unlike unweighted linear regression, which is a **parametric learning algorithm** with a fixed and finite number of parameters (the w_i's), locally weighted linear regression does not have a predetermined number of parameters. In parametric learning, once we determine the w_i's, we can discard the training data and still make predictions. However, for locally weighted linear regression, we must retain the entire training dataset to make future predictions.

2.5 Logistic Regression

Logistic regression is used to classify an observation into one of two classes i.e. it is used for binary classification. Consider a binary classification problem where the response variable y takes values in $\{0, 1\}$. The goal of logistic regression is to train a classifier that can make a binary decision about the class of a new input observation.

Given a training data D consisting of input features $x_1, x_2, .., x_n$, logistic regression models the probability $p(y = 1 \mid \mathbf{x})$ of the positive class given the input vector $\mathbf{x}$ or the probability $p(y = 0 \mid \mathbf{x})$ of the negative class given the input vector $\mathbf{x}$. Note that for given $x^{(i)}$, corresponding $y^{(i)}$ is given in data and called as label.

Logistic Regression is similar to the regression problem, except that the values of y we wan tot predict takes discrete values.

As logistic regression is binary classification, we have two classes where all the data points belongs, i.e. $y \in \{0, 1\}$.

For instance, we are trying to classify email to spam and ham classes through spam classifier. $x^{(i)}$ refers to the i-th training dataset that either belongs to ham class or spam class . Thus, $y^{(i)}$ has value 1 if it is spam, 0 otherwise.
Using linear regression, we predict h(x) as:

$$h(x) = \hat{y} = \sum w_j x_j = \mathbf{w}^T \mathbf{x}$$

where the value of $\hat{y}$ is continuous.

Let's say instead of predicting continuous values for $\hat{y}$, we are taking probabilities P. Thus the value of P must also lie within 0 and 1. But h(x) consists of value ranging from – Inf to + Inf.

So instead of just taking probabilities we take odds, written as

$$odds = \frac{p}{1-p}$$

The odds are defined as the probability that the event will occur divided by the probability that the event will not occur. Thus, Odds are nothing but the ratio of the probability of success and probability of failure. The value of odds lies between 0 to +ve infinity. And, If the odds are high (million to one), the probability is almost 1.00. If the odds are tiny (one to a million), the probability is tiny, almost zero.

To convert from probability to odds, divide the probability by one minus that probability. So if the probability is 10% or 0.10 , then the odds are 0.1/0.9 or '1 to 9' or 0.111. To convert from odds to a probability, divide the odds by one plus the odds. So to convert odds of 1/9 to a probability, divide 1/9 by 10/9 to obtain the probability of 0.10.

But, the problem here with the odds is that the range is restricted as values of odds lies between 0 to Infinity and we don't want a restricted range because if we do so then our correlation will decrease. By restricting the range it is difficult to model a variable. Hence, in order to control this we take the log of odds which has a range from $(-\infty, +\infty)$.

So, we get,

$$\log\left(\frac{p}{1-p}\right) = \sum_{j=1}^{n} w_j x_j = \mathbf{w}^T \ \mathbf{x} = z$$

where $z = \mathbf{w}^T \ \mathbf{x}$

Now, taking exponentiation on both side

$$\exp\left(\log\left(\frac{p}{1-p}\right)\right) = \exp(z)$$

$$\frac{p}{1-p} = \exp(z)$$

$$p = \exp(z) - p \cdot \exp(z)$$

$$p + p \cdot \exp(z) = \exp(z)$$

$$p(1 + \exp(z)) = \exp(z)$$

$$p = \frac{\exp(z)}{1 + \exp(z)}$$

Now, dividing numerator and denominator by $\exp(z)$

$$p = \frac{\exp(z)/\exp(z)}{(1 + \exp(z))/\exp(z)}$$

$$p = \frac{1}{1/\exp(z) + 1}$$

$$p = \frac{1}{\exp(-z) + 1}$$

$$p = \frac{1}{1 + \exp(-z)}$$

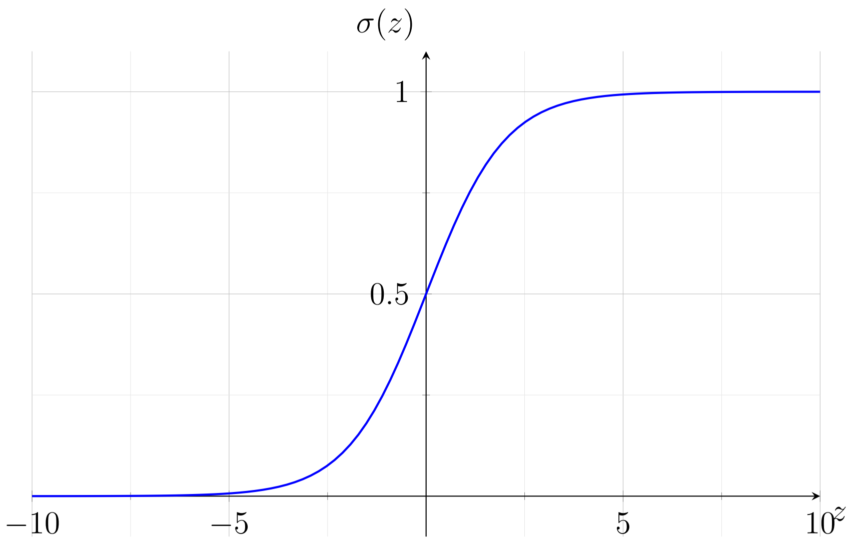
where $z = \mathbf{w}^T \ \mathbf{x}$

This gives us the sigmoid function or the logistic function which is the function of z or $\mathbf{w}^T\mathbf{x}$. The logistic regression uses a sigmoid function as the hypothesis $h(x)$ given by the equation we just derived:

$$h(x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}} = \frac{1}{1 + e^{-z}} = \sigma(z) \tag{2.10}$$

where, $z = \mathbf{w}^T\mathbf{x} = w_0 + w_1x_1 + \cdots + w_nx_n$.

Graphically, the sigmoid function is represented as:



From above, we have our hypothesis for logistic function as:

$$h(x) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

Here, as $\sigma(z)$ tends towards 1 as $z \to \infty$, and $\sigma(z)$ tends towards 0 as $z \to -\infty$. This now shows the value of $h(x)$ is bounded within 0 and 1.

Let's now derive the update rule for logistic regression. Assume the probability of $y = 1$ given $\mathbf{x}$ can be modeled using the logistic function:

$$P(y = 1 \mid \mathbf{x}; \mathbf{w}) = h(x)$$

then,

$$P(y = 0 \mid \mathbf{x}; \mathbf{w}) = 1 - h(x)$$

We can compactly write the above two form as:

$$p(y \mid \mathbf{x}; \mathbf{w}) = (h(x))^y (1 - h(x))^{1-y}$$

Assuming that the m training examples were generated independently, we can then write down the likelihood of the parameters as

$$\begin{aligned} L(w) &= \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}; w) \\ &= \prod_{i=1}^{m} \left(h(x^{(i)})\right)^{y^{(i)}} \left(1 - h(x^{(i)})\right)^{1-y^{(i)}} \end{aligned}$$

Taking the logarithm (log-likelihood) on both side, we obtain:

$$\ell(\mathbf{w}) = \sum_{i=1}^{m} \left[y_i \log h(x^{(i)}) + (1 - y_i) \log \left(1 - h(x^{(i)})\right) \right]$$

which gives the form of **Log-Loss** or **Cross Entropy Loss** used for binary classification. Now, to find $\mathbf{w}$, we maximize the log-likelihood function $\ell(\mathbf{w})$ using gradient ascent. It is given as:

$$w_j = w_j + \alpha \frac{\partial}{\partial w_j} l(w)$$

We start with the logistic regression loss function (log-loss or cross-entropy loss) for a single data point (x, y):

$$\ell(w) = -y \log(h(x)) - (1 - y) \log(1 - h(x))$$

where $h(x) = \sigma(w^\top x)$ is the hypothesis function using the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$. To find the gradient of the loss with respect to the weight w_j, we apply the chain rule. The partial derivative of the loss function with respect to w_j is:

$$\frac{\partial \ell(w)}{\partial w_j} = -y \frac{\partial \log(h(x))}{\partial w_j} - (1 - y) \frac{\partial \log(1 - h(x))}{\partial w_j}$$

Next, we calculate the derivatives of the logarithmic terms. For the first term:

$$\frac{\partial \log(h(x))}{\partial w_j} = \frac{1}{h(x)} \frac{\partial h(x)}{\partial w_j}$$

And for the second term:

$$\frac{\partial \log(1-h(x))}{\partial w_j} = \frac{1}{1-h(x)}\frac{\partial(1-h(x))}{\partial w_j} = -\frac{1}{1-h(x)}\frac{\partial h(x)}{\partial w_j}$$

Substituting these back into the derivative of the loss function, we get:

$$\frac{\partial \ell(w)}{\partial w_j} = -y\left(\frac{1}{h(x)}\frac{\partial h(x)}{\partial w_j}\right) - (1-y)\left(-\frac{1}{1-h(x)}\frac{\partial h(x)}{\partial w_j}\right)$$

Simplify this expression:

$$\frac{\partial \ell(w)}{\partial w_j} = \left(\frac{(1-y)}{1-h(x)} - \frac{y}{h(x)}\right)\frac{\partial h(x)}{\partial w_j}$$

Recall that $h(x) = \sigma(w^\top x)$ and thus $\frac{\partial h(x)}{\partial w_j} = h(x)(1-h(x))x_j$. Substituting this into our equation:

$$\frac{\partial \ell(w)}{\partial w_j} = \left(\frac{(1-y)}{1-h(x)} - \frac{y}{h(x)}\right)h(x)(1-h(x))x_j$$

Simplify the term inside the parenthesis:

$$\frac{\partial \ell(w)}{\partial w_j} = \left(\frac{(1-y)h(x) - y(1-h(x))}{h(x)(1-h(x))}\right)h(x)(1-h(x))x_j$$

The $h(x)(1-h(x))$ terms cancel out:

$$\frac{\partial \ell(w)}{\partial w_j} = \left((1-y)h(x) - y(1-h(x))\right)x_j$$

Distribute $h(x)$ and $(1-h(x))$:

$$\frac{\partial \ell(w)}{\partial w_j} = (h(x) - yh(x) - y + yh(x))x_j$$

Combine like terms:

$$\frac{\partial \ell(w)}{\partial w_j} = (h(x) - y)x_j$$

Thus, the final gradient of the logistic regression loss with respect to the weight w_j is:

$$\frac{\partial \ell(w)}{\partial w_j} = (h(x) - y)x_j$$

On solving, we obtain the update rule for $\mathbf{w}$ using gradient ascent a:

$$w_j = w_j + \alpha\,(y - h(x))\,x_j$$

where α is the learning rate.
This is the **stochastic gradient ascent rule**. If we compare this to the LMS update rule, we see that it looks identical; but this is not the same algorithm, because $h(x)$ is now defined as a non-linear function of $\mathbf{w}^T\mathbf{x}$.
Thus, the sigmoid function gives us a way to take an instance $\mathbf{x}$ and compute the probability $p(y = 1 \mid \mathbf{x})$ and to take a decision about which class to apply to a test instance we use the decision boundary defined by threshold say 0.5. Mathematically, this is given by:

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y = 1 \mid x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 4 Logistic Regression using Gradient Descent

```
Require: Training data $(X, y)$, learning rate $\alpha$, num_iterations, convergence threshold $\epsilon$
Ensure: Learned parameters $\theta$
 1: function LogisticRegression($X, y, \alpha$, num_iterations, $\epsilon$)
 2:     $m \leftarrow$ number of training examples
 3:     $n \leftarrow$ number of features
 4:     $\mathbf{w} \leftarrow$ zeros($n$, 1)                          ▷ Initialize parameters
 5:     for $t = 1$ to num_iterations do
 6:         $z \leftarrow X\mathbf{w}$
 7:         $\hat{y} \leftarrow \frac{1}{1+e^{-z}}$                        ▷ Sigmoid function
 8:         $\nabla J \leftarrow \frac{1}{m} X^T(\hat{y} - y)$              ▷ Compute gradient
 9:         $\mathbf{w}_{\text{new}} \leftarrow \mathbf{w} - \alpha \nabla J$   ▷ Update parameters
10:         if $||\mathbf{w}_{\text{new}} - \mathbf{w}|| < \epsilon$ then
11:             break                                       ▷ Convergence check
12:         end if
13:         $\mathbf{w} \leftarrow \mathbf{w}_{\text{new}}$
14:     end for
15:     return $\theta$
16: end function
```

Pen & Paper

Train a Logistic Regression model in the following data:

x_1	x_2	y
2	3	1
1	1	0
4	5	1
2	1	0
3	2	1

Logistic regression uses the sigmoid function to model the probability of the target variable. The model is defined as:

$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function, $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the feature vector, and b is the bias term.
The cost function for logistic regression is the cross-entropy loss:

$$E = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})\right]$$

where m is the number of training examples, and $\hat{y}^{(i)}$ is the predicted probability for the i-th example.

Gradient Descent We will update the weights and bias using gradient descent. The gradients of the cost function with respect to $\mathbf{w}$ and b are:

$$\frac{\partial E}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})x_j^{(i)}$$

$$\frac{\partial E}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

The weight update rules are:

$$w_j := w_j - \alpha\frac{\partial E}{\partial w_j}$$

$$b := b - \alpha\frac{\partial E}{\partial b}$$

where α is the learning rate.
Let's initialize the parameters and hyperparameter we need:
Learning rate $(\alpha) = 0.1$
Initial weights $(\mathbf{w}) = [0, 0]$
Initial bias $(b) = 0$

Iteration 1 - First, let's compute the predictions

$$\hat{y}^{(i)} = \sigma(0) = 0.5$$

- Now, compute the gradients

$$\frac{\partial E}{\partial w_1} = \frac{1}{5}\left[(0.5-1)\cdot 2 + (0.5-0)\cdot 1 + (0.5-1)\cdot 4 + (0.5-0)\cdot 2 + (0.5-1)\cdot 3\right]$$

$$= \frac{1}{5}\left[-1 + 0.5 - 2 + 1 - 2\right] = -0.5$$

$$\frac{\partial E}{\partial w_2} = \frac{1}{5}\left[(0.5-1)\cdot 3 + (0.5-0)\cdot 1 + (0.5-1)\cdot 5 + (0.5-0)\cdot 1 + (0.5-1)\cdot 2\right]$$

$$= \frac{1}{5}\left[-1.5 + 0.5 - 2.5 + 0.5 - 1\right] = -0.5$$

$$\frac{\partial E}{\partial b} = \frac{1}{5}\left[(0.5-1) + (0.5-0) + (0.5-1) + (0.5-0) + (0.5-1)\right]$$

$$= \frac{1}{5}\left[-0.5 + 0.5 - 0.5 + 0.5 - 0.5\right] = -0.1$$

- Update Weights and Bias

$$w_1 := 0 - 0.1\cdot(-0.5) = 0.05$$

$$w_2 := 0 - 0.1\cdot(-0.5) = 0.05$$

$$b := 0 - 0.1\cdot(-0.1) = 0.01$$

Iteration 2 - Compute Predictions

$$\hat{y}^{(i)} = \sigma(0.05\cdot x_1^{(i)} + 0.05\cdot x_2^{(i)} + 0.01)$$

$$\hat{y}^{(1)} = \sigma(0.05\cdot 2 + 0.05\cdot 3 + 0.01) = \sigma(0.31) \approx 0.577$$

$$\hat{y}^{(2)} = \sigma(0.05\cdot 1 + 0.05\cdot 1 + 0.01) = \sigma(0.11) \approx 0.527$$

$$\hat{y}^{(3)} = \sigma(0.05\cdot 4 + 0.05\cdot 5 + 0.01) = \sigma(0.51) \approx 0.625$$

$$\hat{y}^{(4)} = \sigma(0.05\cdot 2 + 0.05\cdot 1 + 0.01) = \sigma(0.16) \approx 0.540$$

$$\hat{y}^{(5)} = \sigma(0.05 \cdot 3 + 0.05 \cdot 2 + 0.01) = \sigma(0.31) \approx 0.577$$

- Compute Gradients

$$\frac{\partial J}{\partial w_1} = \frac{1}{5}\left[(0.577 - 1) \cdot 2 + (0.527 - 0) \cdot 1 + (0.625 - 1) \cdot 4 + (0.540 - 0) \cdot 2 + (0.577 - 1) \cdot 3\right]$$

$$= \frac{1}{5}\left[-0.846 + 0.527 - 1.5 + 1.08 - 1.269\right] = -0.401$$

$$\frac{\partial J}{\partial w_2} = \frac{1}{5}\left[(0.577 - 1) \cdot 3 + (0.527 - 0) \cdot 1 + (0.625 - 1) \cdot 5 + (0.540 - 0) \cdot 1 + (0.577 - 1) \cdot 2\right]$$

$$= \frac{1}{5}\left[-1.269 + 0.527 - 1.875 + 0.540 - 0.846\right] = -0.584$$

$$\frac{\partial J}{\partial b} = \frac{1}{5}\left[(0.577 - 1) + (0.527 - 0) + (0.625 - 1) + (0.540 - 0) + (0.577 - 1)\right]$$

$$= \frac{1}{5}\left[-0.423 + 0.527 - 0.375 + 0.540 - 0.423\right] = -0.231$$

Update Weights and Bias

$$w_1 := 0.05 - 0.1 \cdot (-0.401) = 0.0901$$

$$w_2 := 0.05 - 0.1 \cdot (-0.584) = 0.1084$$

$$b := 0.01 - 0.1 \cdot (-0.231) = 0.0331$$

Iteration 3 - Compute Predictions

$$\hat{y}^{(i)} = \sigma(0.0901 \cdot x_1^{(i)} + 0.1084 \cdot x_2^{(i)} + 0.0331)$$

$$\hat{y}^{(1)} = \sigma(0.0901 \cdot 2 + 0.1084 \cdot 3 + 0.0331) = \sigma(0.4333) \approx 0.606$$
$$\hat{y}^{(2)} = \sigma(0.0901 \cdot 1 + 0.1084 \cdot 1 + 0.0331) = \sigma(0.2316) \approx 0.558$$
$$\hat{y}^{(3)} = \sigma(0.0901 \cdot 4 + 0.1084 \cdot 5 + 0.0331) = \sigma(0.7357) \approx 0.676$$
$$\hat{y}^{(4)} = \sigma(0.0901 \cdot 2 + 0.1084 \cdot 1 + 0.0331) = \sigma(0.3317) \approx 0.582$$
$$\hat{y}^{(5)} = \sigma(0.0901 \cdot 3 + 0.1084 \cdot 2 + 0.0331) = \sigma(0.4344) \approx 0.606$$

- Compute Gradients

$$\frac{\partial E}{\partial w_1} = \frac{1}{5}\left[(0.606 - 1) \cdot 2 + (0.558 - 0) \cdot 1 + (0.676 - 1) \cdot 4 + (0.582 - 0) \cdot 2 + (0.606 - 1) \cdot 3\right]$$

$$= \frac{1}{5}\left[-0.788 + 0.558 - 1.296 + 1.164 - 1.182\right] = -0.548$$

$$\frac{\partial E}{\partial w_2} = \frac{1}{5}\left[(0.606 - 1) \cdot 3 + (0.558 - 0) \cdot 1 + (0.676 - 1) \cdot 5 + (0.582 - 0) \cdot 1 + (0.606 - 1) \cdot 2\right]$$

$$= \frac{1}{5}\left[-1.188 + 0.558 - 1.62 + 0.582 - 0.788\right] = -0.691$$

$$\frac{\partial E}{\partial b} = \frac{1}{5}\left[(0.606 - 1) + (0.558 - 0) + (0.676 - 1) + (0.582 - 0) + (0.606 - 1)\right]$$

$$= \frac{1}{5}\left[-0.394 + 0.558 - 0.324 + 0.582 - 0.394\right] = -0.176$$

- Finally, update Weights and Bias

$$w_1 := 0.0901 - 0.1 \cdot (-0.548) = 0.1449$$

$$w_2 := 0.1084 - 0.1 \cdot (-0.691) = 0.1775$$

$$b := 0.0331 - 0.1 \cdot (-0.176) = 0.0507$$

This completes the third iteration with updated w_1, w_2 and b.

2.6 The Perceptron Learning Algorithm

As in logistic regression but instead of using $\sigma(x)$, we use any function g(x), say a threshold function as below:

$$g(x) = \begin{cases} 1 & \text{if } z >= 0 \\ 0 & \text{otherwise} \end{cases}$$

If we then let $h(x) = g(\mathbf{w}^T\mathbf{x})$ as before but using this modified definition of g, and if we use the update rule

$$w_j = w_j + \alpha(y^{(i)} - h(x^{(i)})x_j^{(i)}$$

then we have the **Perceptron Learning Algorithm**. It was introduced by Frank Rosenblatt in 1957 and is a type of linear classifier. It learns a linear function for binary classification by iteratively updating weights based on misclassified examples until convergence. The perceptron learning algorithm is a fundamental algorithm in the field of supervised learning and neural networks. It is designed to classify input data into two classes (binary classification) based on a linear decision boundary.

2.7 Support Vector Machine

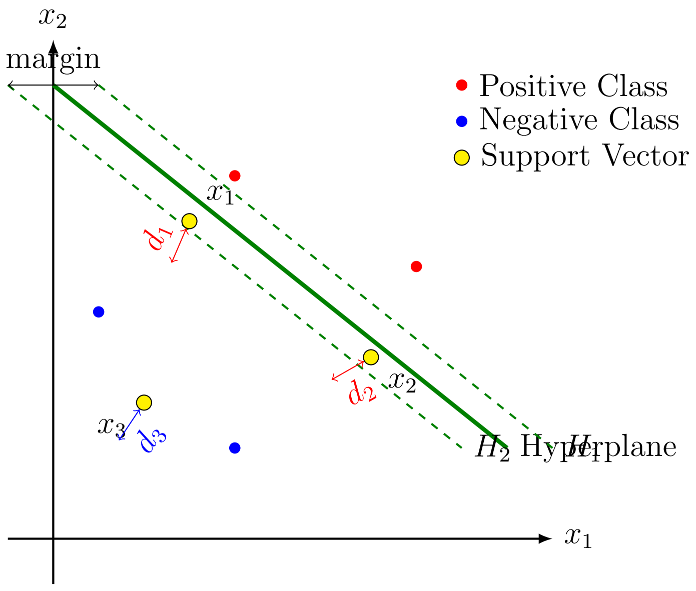
Support Vector Machines (SVMs) are a powerful class of supervised learning algorithms used for both linear and non-linear classification problems. They are widely applied in various domains, including text categorization, image recognition, and bioinformatics. SVM's are also known for regression and outlier detection. Here, we will consdier SVM for the case of clssification.
The key idea of SVM as a classifier is to find hyperplane that separates the difference class. Let's consider a simple scenario of binary classification for the derivation.
Consider a binary classification problem with a training dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, where:

- $\mathbf{x}^{(i)} \in \mathbb{R}^n$ are the input features
- $y^{(i)} \in \{-1, +1\}$ are the corresponding class labels
- m is the number of training samples
- n refers to the number of input features

Our goal is to find a hyperplane that separates the two classes with the maximum margin. We write out classifier as:

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

where, $g(z) = 1$ if $z >= 0$ and $g(z) = -1$ otherwise. The use of w and b separately, here, helps us to treat the intercept term b distinguishably from other coefficients.



Geometric Margin: $\gamma = \min(d_1, d_2, d_3)$
Functional Margin: $\hat{\gamma} = y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$

2.7.1 Hard Margin SVM: Linearly Separable Data

We begin by considering the case where the training data points from two classes can be perfectly separated by a straight line (in 2D) or a hyperplane (in higher dimensions).

Linear Separability and the Hyperplane : Let the training dataset be $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ is the i-th data point with n features, and $y^{(i)} \in \{-1, +1\}$ is its corresponding class label. We assume that this data is linearly separable, meaning there exists at least one hyperplane that can perfectly separate the positive $(+1)$ and negative (-1) class points.
The equation of a hyperplane in the n-dimensional feature space $\mathbb{R}^n$ is given by:

$$\mathbf{w}^T\mathbf{x} + b = 0 \tag{2.11}$$

where $\mathbf{w} \in \mathbb{R}^n$ is the vector normal to the hyperplane, and $b \in \mathbb{R}$ is the bias term (determining the offset of the hyperplane from the origin).

Decision Function : Once a hyperplane $(\mathbf{w}, b)$ is determined, the decision function for classifying a new point $\mathbf{x}$ is based on the sign of the expression $\mathbf{w}^T\mathbf{x} + b$:

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b) \tag{2.12}$$

For correct classification of all training points $(\mathbf{x}^{(i)}, y^{(i)})$ in the linearly separable case, we require that each point lies on the correct side of the hyperplane. Mathematically, this means the sign of $\mathbf{w}^T\mathbf{x}^{(i)} + b$ must match the true label $y^{(i)}$. This can be expressed as:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) > 0, \quad \forall i = 1, \ldots, m \tag{2.13}$$

This inequality ensures that the term $(\mathbf{w}^T\mathbf{x}^{(i)} + b)$ has the same sign as $y^{(i)}$, meaning all points are on the correct side.

Margin Maximization : The geometric margin of a data point $\mathbf{x}^{(i)}$ with respect to the hyperplane $(\mathbf{w}, b)$ is the perpendicular distance from the point to the hyperplane. It is given by:

$$\gamma^{(i)} = \frac{y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b)}{\|\mathbf{w}\|} \tag{2.14}$$

where $\|\mathbf{w}\|$ is the Euclidean norm of the vector $\mathbf{w}$. The term $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b)$ is called the functional margin. For correctly classified points, the functional margin is positive.
The objective of the Support Vector Machine is to find the hyperplane that maximizes the minimum geometric margin among all training points:

$$\max_{\mathbf{w},b} \min_{i=1,\ldots,m} \gamma^{(i)} = \max_{\mathbf{w},b} \min_{i=1,\ldots,m} \frac{y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b)}{\|\mathbf{w}\|} \tag{2.15}$$

Maximizing the minimum geometric margin ensures the largest possible separation between the two classes.

Constraint Normalization : To simplify the optimization problem, we can fix the functional margin of the points closest to the hyperplane (the support vectors) to be equal to 1. Since the geometric margin is $\gamma^{(i)} = \hat{\gamma}^{(i)}/\|\mathbf{w}\|$, maximizing the geometric margin $\gamma^{(i)}$ while setting the minimum functional margin $\hat{\gamma} = \min_i \hat{\gamma}^{(i)}$ to 1 is equivalent to maximizing $1/\|\mathbf{w}\|$, which in turn is equivalent to minimizing $\|\mathbf{w}\|$.
By choosing an appropriate scaling of $\mathbf{w}$ and b, we can enforce that for the points closest to the hyperplane (the support vectors), the functional margin is exactly 1:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) = 1 \quad \text{for support vectors} \tag{2.16}$$

For all other training points (which are further away from the hyperplane), the functional margin will be greater than 1. Thus, for all training points, we have the constraint:

$$y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1, \quad \forall i = 1, \ldots, m \tag{2.17}$$

This is the fundamental **hard margin SVM constraint** that all training points must satisfy. With this normalization, the geometric margin is simply $1/\|\mathbf{w}\|$.

Primal Optimization Problem : With the constraint normalized, the problem of maximizing the geometric margin $(1/\|\mathbf{w}\|)$ is equivalent to minimizing $\|\mathbf{w}\|$. For mathematical convenience in differentiation, we minimize $\frac{1}{2}\|\mathbf{w}\|^2$. This leads to the following convex quadratic optimization problem, known as the primal SVM optimization problem:

$$\min_{\mathbf{w},b} \quad \frac{1}{2}\|\mathbf{w}\|^2 \tag{2.18}$$

$$\text{subject to} \quad y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) - 1 \geq 0, \quad \forall i = 1, \ldots, m \tag{2.19}$$

This is a constrained optimization problem that can be solved using techniques like quadratic programming.

Lagrangian Formulation : To solve this constrained optimization problem, particularly to handle the inequality constraints and prepare for the dual formulation (which is often easier to solve and enables the use of kernels), we introduce non-negative Lagrange multipliers $\alpha_i \geq 0$ for each constraint. The Lagrangian function is formed by combining the objective function and the constraints:

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{m} \alpha_i [y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) - 1] \tag{2.20}$$

where $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_m]^T$ is the vector of Lagrange multipliers. The problem is to find the saddle point of the Lagrangian.

Karush-Kuhn-Tucker (KKT) Conditions : The optimal solution $(\mathbf{w}^*, b^*, \boldsymbol{\alpha}^*)$ of the constrained optimization problem must satisfy the Karush-Kuhn-Tucker (KKT) conditions. These conditions are derived by setting the partial derivatives of the Lagrangian with respect to the primal variables ($\mathbf{w}$ and b) to zero and incorporating the primal and dual feasibility constraints and the complementary slackness condition.
The KKT conditions are:

$$\text{Stationarity:} \quad \frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)} = 0 \tag{2.21}$$

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \tag{2.22}$$

$$\text{Dual Feasibility:} \quad \alpha_i \geq 0, \quad \forall i = 1, \ldots, m \tag{2.23}$$

$$\text{Primal Feasibility:} \quad y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1 \geq 0, \quad \forall i = 1, \ldots, m \tag{2.24}$$

$$\text{Complementary Slackness:} \quad \alpha_i [y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1] = 0, \quad \forall i = 1, \ldots, m \tag{2.25}$$

The stationarity conditions ensure that the gradient of the Lagrangian is zero at the optimum. Dual feasibility ensures that the Lagrange multipliers are non-negative. Primal feasibility ensures that the original problem's constraints are satisfied. The complementary slackness condition is particularly insightful: it implies that for any training point i, either its Lagrange multiplier α_i is zero, or the constraint $y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1 = 0$ is active (i.e., the point lies exactly on the margin boundaries $y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) = 1$).

Dual Formulation : From the KKT stationarity condition with respect to $\mathbf{w}$, we have a crucial relationship expressing the weight vector $\mathbf{w}$ as a linear combination of the training data points:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)} \tag{2.26}$$

Substituting this expression for $\mathbf{w}$ back into the Lagrangian and using the other KKT conditions (especially $\sum_{i=1}^{m} \alpha_i y^{(i)} = 0$), we eliminate $\mathbf{w}$ and b from the Lagrangian and obtain the dual optimization problem, which is solely in terms of the Lagrange multipliers $\boldsymbol{\alpha}$:

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} (\mathbf{x}^{(i)})^T \mathbf{x}^{(j)} \tag{2.27}$$

$$\text{subject to} \quad \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \tag{2.28}$$

$$\alpha_i \geq 0, \quad \forall i = 1, \ldots, m \tag{2.29}$$

This is a convex quadratic programming problem in $\boldsymbol{\alpha}$, which can be solved efficiently using standard quadratic programming solvers. Notice that the data points only appear in the form of dot products $(\mathbf{x}^{(i)})^T \mathbf{x}^{(j)}$. This is a key feature that allows for the "kernel trick" to handle non-linear classification.

Support Vectors : The points $\mathbf{x}^{(i)}$ for which the optimal Lagrange multipliers $\alpha_i^* > 0$ are called **support vectors**. From the KKT complementary slackness condition $\alpha_i[y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1] = 0$, if $\alpha_i > 0$, the term in the square brackets must be zero, meaning $y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1 = 0$, or $y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) = 1$. This confirms that support vectors are precisely the points that lie exactly on the margin boundaries. For points that are not support vectors, $\alpha_i = 0$, and they lie outside the margin boundaries.

Decision Boundary : Once the dual problem is solved for the optimal $\boldsymbol{\alpha}^*$, the optimal weight vector $\mathbf{w}^*$ can be computed directly using the derived relationship:

$$\mathbf{w}^* = \sum_{i=1}^{m} \alpha_i^* y^{(i)} \mathbf{x}^{(i)} \tag{2.30}$$

The optimal bias term b^* can be found by using the KKT complementary slackness condition for any support vector $\mathbf{x}^{(s)}$ (i.e., any point with $\alpha_s^* > 0$). For such a point, we know $y^{(s)}(\mathbf{w}^{*T}\mathbf{x}^{(s)} + b^*) = 1$. Since $y^{(s)}$ is either +1 or -1, $y^{(s)} \cdot y^{(s)} = 1$. Multiplying by $y^{(s)}$ gives $y^{(s)} \cdot y^{(s)}(\mathbf{w}^{*T}\mathbf{x}^{(s)} + b^*) = y^{(s)} \cdot 1$, which simplifies to $\mathbf{w}^{*T}\mathbf{x}^{(s)} + b^* = y^{(s)}$. Rearranging for b^*:

$$b^* = y^{(s)} - (\mathbf{w}^*)^T \mathbf{x}^{(s)} \tag{2.31}$$

In practice, it's common to average the value of b^* calculated using several support vectors for numerical stability.
The decision function for classifying a new point $\mathbf{x}$ is then:

$$f(\mathbf{x}) = \text{sign}((\mathbf{w}^*)^T \mathbf{x} + b^*) \tag{2.32}$$

Substituting the expression for $\mathbf{w}^*$ derived from the dual problem, the decision function can be written purely in terms of the support vectors and the dot product with the new point $\mathbf{x}$:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{m} \alpha_i^* y^{(i)} (\mathbf{x}^{(i)})^T \mathbf{x} + b^* \right) \tag{2.33}$$

This form is particularly important because it shows that the decision boundary is determined only by the support vectors (those with $\alpha_i^* > 0$) and relies only on dot products between data points.

2.7.2 Soft Margin SVM: Non-Linearly Separable Data

In most real-world scenarios, data is not perfectly linearly separable. To handle this, the **Soft Margin SVM** is introduced. It allows for some misclassification or violation of the margin constraints to find a hyperplane that still provides a good separation.
This is achieved by introducing non-negative **slack variables** $\xi_i \geq 0$ (Greek letter xi) for each training point i. The slack variable ξ_i measures the degree to which the i-th point violates the hard margin constraint $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1$.

- If $\xi_i = 0$, the point satisfies the hard margin constraint $(y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1)$.
- If $0 < \xi_i < 1$, the point is inside the margin but on the correct side of the hyperplane $(0 < y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) < 1)$.
- If $\xi_i = 1$, the point is exactly on the hyperplane $(y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) = 0)$.
- If $\xi_i > 1$, the point is on the wrong side of the hyperplane (misclassified) $(y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) < 0)$.

The optimization objective is modified to include a penalty for constraint violations (measured by the sum of slack variables), in addition to minimizing $\|\mathbf{w}\|^2$. This leads to the soft margin primal optimization problem:

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m} \xi_i \tag{2.34}$$

$$\text{subject to} \quad y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i, \quad \forall i = 1, \dots, m \tag{2.35}$$

$$\xi_i \geq 0, \quad \forall i = 1, \dots, m \tag{2.36}$$

Here, $C > 0$ is a regularization parameter (a hyperparameter) that controls the trade-off between maximizing the margin (minimizing $\|\mathbf{w}\|^2$) and minimizing the total amount of constraint violation ($\sum \xi_i$).

- A small value of C emphasizes maximizing the margin, allowing for more constraint violations (more misclassification).
- A large value of C emphasizes minimizing constraint violations, potentially leading to a narrower margin but fewer training errors.

This soft margin primal problem is also a convex quadratic program and can be solved using similar techniques as the hard margin case, leading to a dual formulation that involves the slack variables and the parameter C. The dual formulation of the soft margin SVM is the basis for practical SVM implementations and still relies only on dot products, enabling the kernel trick for non-linear separation.

2.7.3 Kernel Trick

For non-linear classification, we can replace the dot product $\mathbf{x}_i^T\mathbf{x}_j$ with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$. The kernel function computes the inner product of the data points in the transformed feature space without explicitly performing the transformation, making the computation more efficient. This leads to the Kernelized SVM dual problem:

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{2.37}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.38}$$

$$0 \leq \alpha_i \leq C, \quad \forall i = 1, \ldots, n \tag{2.39}$$

The decision function becomes:

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b\right) \tag{2.40}$$

Common kernel functions include:

- Linear: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_j$
- Polynomial: $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T\mathbf{x}_j + c)^d$
- Radial Basis Function (RBF): $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$

Pen & Paper

Consider the following data points:

- Positively labeled data: $(3, 1)$, $(3, -1)$, $(6, 1)$, $(6, -1)$
- Negatively labeled data: $(1, 0)$, $(0, 1)$, $(0, -1)$, $(-1, 0)$

We need to determine the equation of the hyperplane that divides the above data points

Algorithm 5 Support Vector Machine

Require: Training data X, labels $y \in \{-1, 1\}$, regularization parameter C, learning rate α, number of iterations n, kernel function $KERNEL$

Ensure: Optimized dual variables α, bias b

```
 1: function SVM($X$, $y$, $C$, $\alpha$, $n$, $K$)
 2:     Initialize $\alpha \leftarrow 0$, $b \leftarrow 0$
 3:     for $i = 1$ to $n$ do
 4:         for each $(x_j, y_j)$ in $(X, y)$ do
 5:             $E_j \leftarrow \sum_{i=1}^{m} \alpha_i y_i KERNEL(x_i, x_j, kernel_type) + b - y_j$
 6:             $\alpha_j^{old} \leftarrow \alpha_j$
 7:             $\alpha_j \leftarrow \text{clip}(\alpha_j - y_j E_j / K(x_j, x_j), 0, C)$
 8:             $b \leftarrow b - E_j - y_j(\alpha_j - \alpha_j^{old}) K(x_j, x_j)$
 9:         end for
10:     end for
11:     return $\alpha$, $b$
12: end function
13: function Kernel($x_1$, $x_2$, $kernel\_type$)
14:     if $kernel\_type$ = 'linear' then
15:         return $x_1^T x_2$
16:     else if $kernel\_type$ = 'polynomial' then
17:         return $(1 + x_1^T x_2)^d$                    ▷ $d$ is the polynomial degree
18:     else if $kernel\_type$ = 'rbf' then
19:         return $\exp(-\gamma \|x_1 - x_2\|^2)$        ▷ $\gamma$ is the RBF parameter
20:     end if
21: end function
```

into two classes using SVM with a linear kernel. Additionally, we predict the class of the point $(5, 2)$.
Solution: Since we are using a linear kernel, we identify the support vectors as $(1, 0)$, $(3, 1)$, and $(3, -1)$ (denoted as s_1, s_2, and s_3 respectively).
Given the support vectors:

$$s_1 = (1, 0), \quad s_2 = (3, 1), \quad s_3 = (3, -1)$$

We formulate the following system of equations:

$$\begin{aligned} a \cdot s_1 \cdot s_1 + b \cdot s_2 \cdot s_1 + c \cdot s_3 \cdot s_1 &= -1 \\ a \cdot s_1 \cdot s_2 + b \cdot s_2 \cdot s_2 + c \cdot s_3 \cdot s_2 &= 1 \\ a \cdot s_1 \cdot s_3 + b \cdot s_2 \cdot s_3 + c \cdot s_3 \cdot s_3 &= 1 \end{aligned}$$

Since we are using a linear kernel, we use dot products. The first equation has a -1 on the right-hand side since s_1 belongs to the negatively labeled class, while the other two equations have $+1$ since s_2 and s_3 belong to the positively labeled class.
Adding the bias $b = 1$ to each support vector, we get:

$$s_1 = (1, 0, 1), \quad s_2 = (3, 1, 1), \quad s_3 = (3, -1, 1)$$

Computing the dot products:

$$\begin{aligned} s_1 \cdot s_1 &= 2, & s_1 \cdot s_2 &= 4, & s_1 \cdot s_3 &= 4, \\ s_2 \cdot s_2 &= 11, & s_3 \cdot s_3 &= 11 \end{aligned}$$

Substituting into the equations:

$$\begin{aligned} 2a + 4b + 4c &= -1 \\ 4a + 11b + 9c &= 1 \\ 4a + 9b + 11c &= 1 \end{aligned}$$

Solving these equations, we obtain:

$$a = -3.5, \quad b = 0.75, \quad c = 0.75$$

The weight vector of the hyperplane is given by:

$$\mathbf{w} = a \cdot s_1 + b \cdot s_2 + c \cdot s_3$$

$$\mathbf{w} = -3.5 \cdot (1, 0, 1) + 0.75 \cdot (3, 1, 1) + 0.75 \cdot (3, -1, 1)$$
$$\mathbf{w} = (1, 0, -2)$$

Thus, the equation of the hyperplane is:

$$y = w_1 x_1 + w_2 x_2 + w_0 = 1 \cdot x_1 + 0 \cdot x_2 - 2$$

$$y = x_1 - 2$$

Finally, to predict the class of the point $(5, 2)$:

$$y = x_1 - 2 = 5 - 2 = 3$$

Since $y > 0$, the point $(5, 2)$ belongs to the positive class.

2.8 Naive Bayes

Naive Bayes is a probabilistic machine learning algorithm based on Bayes' Theorem. It is particularly useful for classification tasks and is widely applied in areas such as spam detection, sentiment analysis, and document classification. Naive Bayes is a model with high bias and low variance.

2.8.1 Bayes' Theorem

The foundation of Naive Bayes is Bayes' Theorem, which states:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{2.41}$$

Where:

- $P(y|x)$ is the posterior probability
- $P(x|y)$ is the likelihood
- $P(y)$ is the prior probability
- $P(x)$ is the marginal likelihood (or evidence)

Let $\mathbf{x} = (x_1, \dots, x_n) \in \{0, 1\}^n$ be a feature vector representing n binary features, and let $y \in \{0, 1\}$ be the class label. We aim to find the probability $P(y|\mathbf{x})$, which is the probability of the class label given the feature vector.
Then we can write:

$$\begin{aligned} P(y|\mathbf{x}) &= \frac{P(\mathbf{x}|y)P(y)}{P(\mathbf{x})} \\ &= \frac{P(x_1, \dots, x_n|y)P(y)}{P(\mathbf{x})} \end{aligned}$$

The "Naive" Assumption: In the Naive Bayes classifier, the "naive" assumption is that the features x_i are conditionally independent given the class label y. This means that the presence or absence of one feature does not affect the presence or absence of any other feature, given the class.
For dependent events, the joint probability can be expressed as:

$$P(x_1, \dots, x_n|y) = P(x_1|y) \cdot P(x_2|y, x_1) \cdot P(x_3|y, x_1, x_2) \cdots P(x_n|y, x_1, \dots, x_{n-1})$$

Applying the naive assumption of conditional independence, this simplifies to:

$$P(x_1, \dots, x_n|y) = P(x_1|y) \cdot P(x_2|y) \cdot P(x_3|y) \cdots P(x_n|y) = \prod_{j=1}^{n} P(x_j|y)$$

Substituting this into Bayes' theorem, we get:

$$P(y|\mathbf{x}) = \frac{\prod_{j=1}^{n} P(x_j|y) \cdot P(y)}{P(\mathbf{x})}$$

Since $P(\mathbf{x})$ is the same for both classes when comparing $P(y = 1|\mathbf{x})$ and $P(y = 0|\mathbf{x})$ to make a classification decision, we can often ignore it:

$$P(y|\mathbf{x}) \propto \prod_{j=1}^{n} P(x_j|y) \cdot P(y)$$

For binary classification, we are often interested in the ratio of the probabilities of the two classes, $y = 1$ and $y = 0$. This ratio is:

$$\begin{aligned}\frac{P(y = 1|\mathbf{x})}{P(y = 0|\mathbf{x})} &= \frac{\frac{\prod_{j=1}^{n} P(x_j|y=1)\cdot P(y=1)}{P(\mathbf{x})}}{\frac{\prod_{j=1}^{n} P(x_j|y=0)\cdot P(y=0)}{P(\mathbf{x})}} \\ &= \frac{\prod_{j=1}^{n} P(x_j|y = 1) \cdot P(y = 1)}{\prod_{j=1}^{n} P(x_j|y = 0) \cdot P(y = 0)}\end{aligned}$$

Taking the natural logarithm of both sides gives the log-odds ratio:

$$\log \frac{P(y = 1|\mathbf{x})}{P(y = 0|\mathbf{x})} = \log \frac{P(y = 1)}{P(y = 0)} + \sum_{j=1}^{n} \log \frac{P(x_j|y = 1)}{P(x_j|y = 0)} \tag{2.42}$$

Define the following parameters for simplicity:

$$\begin{aligned}\phi_y &= P(y = 1) \\ \phi_{j|y=1} &= P(x_j = 1|y = 1) \\ \phi_{j|y=0} &= P(x_j = 1|y = 0)\end{aligned}$$

Using these definitions, and assuming binary features, the conditional probabilities can be modeled using the Bernoulli distribution:

$$\begin{aligned}P(x_j|y = 1) &= \phi_{j|y=1}^{x_j} \cdot (1 - \phi_{j|y=1})^{1-x_j} \\ P(x_j|y = 0) &= \phi_{j|y=0}^{x_j} \cdot (1 - \phi_{j|y=0})^{1-x_j}\end{aligned}$$

Substituting these into the log-odds ratio expression 2.42 gives:

$$\begin{aligned}\log \frac{P(y = 1|\mathbf{x})}{P(y = 0|\mathbf{x})} &= \log \frac{\phi_y}{1 - \phi_y} + \sum_{j=1}^{n} \log \frac{\phi_{j|y=1}^{x_j} \cdot (1 - \phi_{j|y=1})^{1-x_j}}{\phi_{j|y=0}^{x_j} \cdot (1 - \phi_{j|y=0})^{1-x_j}} \\ &= \log \frac{\phi_y}{1 - \phi_y} + \sum_{j=1}^{n} \left(x_j \log \frac{\phi_{j|y=1}}{\phi_{j|y=0}} + (1 - x_j) \log \frac{1 - \phi_{j|y=1}}{1 - \phi_{j|y=0}} \right)\end{aligned}$$

This final log-odds ratio is used for making predictions:

- If $\log \frac{P(y=1|\mathbf{x})}{P(y=0|\mathbf{x})} > 0$, predict $y = 1$.
- Otherwise, predict $y = 0$.

Naive Bayes Classifier For a feature vector $\mathbf{x} = (x_1, \ldots, x_n)$ and a class variable y, the Naive Bayes classifier predicts the class $\hat{y}$ that maximizes the posterior probability:

$$\hat{y} = \arg\max_y P(y|x_1, \ldots, x_n) \tag{2.43}$$

Using Bayes' theorem and the naive assumption, this can be written as:

$$\hat{y} = \arg\max_y \frac{P(x_1, \ldots, x_n|y) \cdot P(y)}{P(x_1, \ldots, x_n)} = \arg\max_y \left(\prod_{j=1}^{n} P(x_j|y) \cdot P(y) \right) \tag{2.44}$$

Since the denominator $P(x_1, \ldots, x_n)$ does not depend on y, we can ignore it for the argmax operation.

Parameter Estimation: The parameters ϕ_y, $\phi_{j|y=1}$, and $\phi_{j|y=0}$ are estimated from the training data using maximum likelihood estimation (MLE). For binary features and a binary class label:

$$\phi_y = P(y = 1) = \frac{\sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 1\}}{m} \tag{2.45}$$

$$\phi_{j|y=1} = P(x_j = 1|y = 1) = \frac{\sum_{i=1}^{m} \mathbf{1}\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 1\}}{\sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 1\}} \tag{2.46}$$

$$\phi_{j|y=0} = P(x_j = 1|y = 0) = \frac{\sum_{i=1}^{m} \mathbf{1}\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 0\}}{\sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 0\}} \tag{2.47}$$

Here, m is the number of training examples, and $\mathbf{1}\{\cdot\}$ is the indicator function, which is 1 if the condition inside is true and 0 otherwise.

Laplace Smoothing (Add-One Smoothing): To handle the case where a feature value does not appear in the training data for a particular class (resulting in a zero probability), Laplace smoothing is often used. For binary features, the smoothed estimates are:

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{m} \mathbf{1}\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 1\} + 2} \tag{2.48}$$

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{m} \mathbf{1}\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 0\} + 2} \tag{2.49}$$

And for the prior probability:

$$\phi_y = \frac{\sum_{i=1}^{m} \mathbf{1}\{y^{(i)} = 1\} + 1}{m + 2} \tag{2.50}$$

This ensures that no probability is exactly zero.

Example and Context : Consider a spam detection problem where each email is represented by a binary feature vector indicating the presence or absence of certain words. The class label y indicates whether the email is spam (1) or not spam (0). Using Naive Bayes, we can predict the probability that a new email is spam based on its features and the parameters estimated from a training dataset of labeled emails.

2.8.2 Types of Naive Bayes

1. **Gaussian Naive Bayes**

   Gaussian Naive Bayes is a type of Naive Bayes classifier used for continuous data. It assumes that the features follow a normal (Gaussian) distribution within each class. For each feature, the model estimates the mean and variance for each class from the training data. These parameters are then used to calculate the conditional probability of a feature given the class using the Gaussian probability density function.

   The probability density function of a Gaussian distribution is given by:

   $$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2.51}$$

   Where:

   - x_i is the i-th feature.
   - y is the class label.
   - μ_y is the mean of the i-th feature for class y, estimated from the training data.
   - σ_y^2 is the variance of the i-th feature for class y, estimated from the training data.

2. **Multinomial Naive Bayes**

   Multinomial Naive Bayes is a variant of the Naive Bayes classifier that is well-suited for discrete data, particularly for text classification problems where the features represent the frequency or count of words or tokens in a document. It assumes that the features are multinomially distributed.

   For a given class y, the likelihood of observing a particular feature count vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is given by the multinomial distribution:

   $$P(\mathbf{x}|y) = \frac{(\sum_{i=1}^n x_i)!}{\prod_{i=1}^n x_i!} \prod_{i=1}^n P(w_i|y)^{x_i} \tag{2.52}$$

   Where:

   - x_i is the count of the i-th word (or token) in the document.
   - n is the total number of distinct words (vocabulary size).
   - $P(w_i|y)$ is the probability of the i-th word occurring given the class y.

   The probability $P(w_i|y)$ is estimated from the training data using:

   $$P(w_i|y) = \frac{\text{count}(w_i, y) + \alpha}{\sum_{j=1}^n (\text{count}(w_j, y) + \alpha)} \tag{2.53}$$

   Where:

   - $\text{count}(w_i, y)$ is the number of times word w_i appears in documents of class y in the training data.
   - α is a smoothing parameter (often Laplace smoothing with $\alpha = 1$) to avoid zero probabilities.
   - The denominator is the total count of all words in all documents of class y, plus α times the vocabulary size.

3. **Bernoulli Naive Bayes**

   Bernoulli Naive Bayes is a type of Naive Bayes classifier that works with binary data. This means it deals with situations where each feature can be either "on" or "off" (e.g., a word is either present or absent in a document). The features are binary variables representing the occurrence (1) or non-occurrence (0) of an attribute. This makes it particularly useful for text classification tasks, such as spam detection, where the presence or absence of certain words is a strong indicator.

   For a given class y, the likelihood of a binary feature vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is given by:

$$P(\mathbf{x}|y) = \prod_{i=1}^{n} P(x_i|y) \tag{2.54}$$

   Where the conditional probability of each binary feature x_i given class y is modeled as a Bernoulli distribution:

$$P(x_i = 1|y) = p_{i|y} \tag{2.55}$$

$$P(x_i = 0|y) = 1 - p_{i|y} \tag{2.56}$$

   These probabilities $p_{i|y}$ are estimated from the training data using:

$$p_{i|y} = \frac{\text{count}(x_i = 1, y) + \alpha}{\text{count}(y) + 2\alpha} \tag{2.57}$$

   Where:

   - $\text{count}(x_i = 1, y)$ is the number of training instances of class y where the i-th feature is present (has a value of 1).
   - $\text{count}(y)$ is the total number of training instances of class y.
   - α is a smoothing parameter (often Laplace smoothing with $\alpha = 1$). The $+2\alpha$ in the denominator accounts for the two possible values of the binary feature.

In Naive Bayes, a problem arises if a feature x_j does not appear in the training data for a particular class C. In such cases, the probability $P(x_j \mid C)$ can be zero, which leads to a zero product in the Naive Bayes classifier, making it unable to handle such scenarios well.

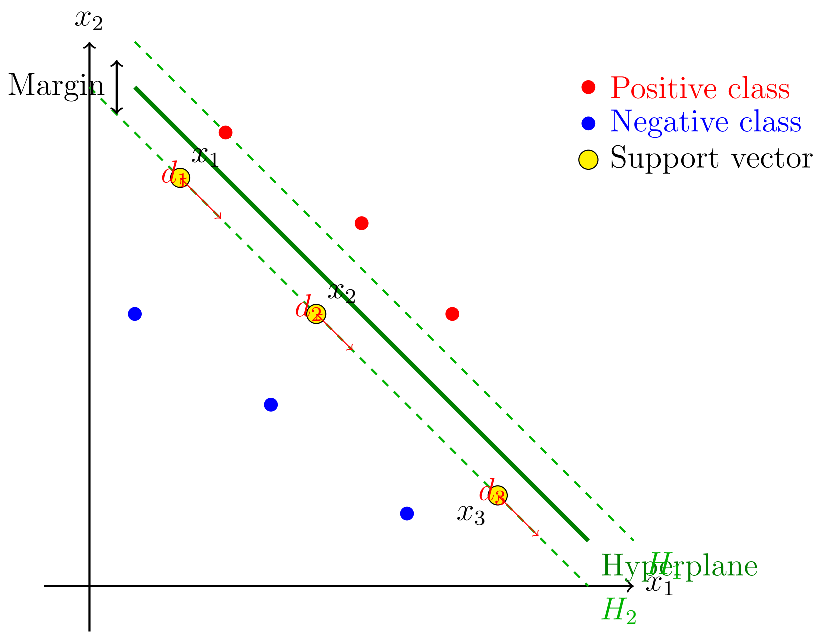
Laplace smoothing, also known as additive smoothing, is a technique used to handle the problem of zero probabilities. Without smoothing, the conditional probability of a feature x_j given a class y is given by:

$$p(x_j \mid y) = \frac{N_{j,y}}{N_y}$$

where $N_{j,y}$ represents the count of instances where $x_j = 1$ in class y, and N_y is the total number of instances in class y.
Applying Laplace smoothing with parameter α, the smoothed conditional probability is:

$$p(x_j \mid y) = \frac{N_{j,y} + \alpha}{N_y + \alpha \cdot K}$$

Here, α is the smoothing parameter (commonly set to 1), and K denotes the number of possible values for the feature. For binary features, $K = 2$.



Geometric Margin: $\gamma = \min(d_1, d_2, d_3)$
Functional Margin: $\hat{\gamma} = y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$

Figure 2.5: SVM Hyperplane, Margin, and Support Vectors

The smoothed probability of the feature x_j being zero, given the class y, is:

$$p(x_j = 0 \mid y) = \frac{(N_y - N_{j,y}) + \alpha}{N_y + \alpha \cdot K}$$

For the class prior probability $p(y)$, Laplace smoothing adjusts the count as follows:

$$p(y) = \frac{N_y + \alpha \cdot C}{M + \alpha \cdot C}$$

where N_y is the count of instances of class y, C is the number of distinct classes, and M is the total number of instances in the training data.

Pen & Paper

Given the following dataset:

Fur	Feathers	Swim	Type
Yes	No	No	Mammal
Yes	Yes	No	Bird
No	No	Yes	Mammal
Yes	No	Yes	Mammal
No	Yes	No	Bird
No	No	No	Mammal
No	Yes	Yes	Bird
Yes	Yes	Yes	Mammal
No	Yes	Yes	Bird

We aim to predict the class (Type) of a new animal with the following features:

- Fur: Yes
- Feathers: No

- Swim: Yes

Solution: Using the Naive Bayes classifier, we calculate the posterior probabilities for each class given the features. The class with the highest posterior probability will be our prediction.

Step 1: Calculate Priors The priors $P(\text{Mammal})$ and $P(\text{Bird})$ are calculated as follows:

$$P(\text{Mammal}) = \frac{\text{Number of Mammals}}{\text{Total number of examples}} = \frac{5}{9}$$

$$P(\text{Bird}) = \frac{\text{Number of Birds}}{\text{Total number of examples}} = \frac{4}{9}$$

Step 2: Calculate Likelihoods For the feature Fur:

$$P(\text{Fur} = \text{Yes} \mid \text{Mammal}) = \frac{\text{Number of Mammals with Fur}}{\text{Number of Mammals}} = \frac{3}{5}$$

$$P(\text{Fur} = \text{Yes} \mid \text{Bird}) = \frac{\text{Number of Birds with Fur}}{\text{Number of Birds}} = \frac{1}{4}$$

For the feature Feathers:

$$P(\text{Feathers} = \text{No} \mid \text{Mammal}) = \frac{\text{Number of Mammals without Feathers}}{\text{Number of Mammals}} = \frac{4}{5}$$

$$P(\text{Feathers} = \text{No} \mid \text{Bird}) = \frac{\text{Number of Birds without Feathers}}{\text{Number of Birds}} = \frac{1}{4}$$

For the feature Swim:

$$P(\text{Swim} = \text{Yes} \mid \text{Mammal}) = \frac{\text{Number of Mammals that Swim}}{\text{Number of Mammals}} = \frac{2}{5}$$

$$P(\text{Swim} = \text{Yes} \mid \text{Bird}) = \frac{\text{Number of Birds that Swim}}{\text{Number of Birds}} = \frac{3}{4}$$

Step 3: Calculate Posteriors We use Bayes' theorem to calculate the posterior probabilities:

For Mammal:

$$\begin{aligned}
&P(\text{Mammal} \mid \text{Fur} = \text{Yes}, \text{Feathers} = \text{No}, \text{Swim} = \text{Yes}) \\
&= P(\text{Mammal}) \cdot P(\text{Fur} = \text{Yes} \mid \text{Mammal}) \cdot P(\text{Feathers} = \text{No} \mid \text{Mammal}) \cdot P(\text{Swim} = \text{Yes} \mid \text{Mammal}) \\
&= \frac{5}{9} \cdot \frac{3}{5} \cdot \frac{4}{5} \cdot \frac{2}{5} \\
&= \frac{5}{9} \cdot \frac{24}{125} \\
&= \frac{120}{1125}
\end{aligned}$$

For Bird:

$$
\begin{aligned}
&P(\text{Bird} \mid \text{Fur} = \text{Yes}, \text{Feathers} = \text{No}, \text{Swim} = \text{Yes}) \\
&= P(\text{Bird}) \cdot P(\text{Fur} = \text{Yes} \mid \text{Bird}) \cdot P(\text{Feathers} = \text{No} \mid \text{Bird}) \cdot P(\text{Swim} = \text{Yes} \mid \text{Bird}) \\
&= \frac{4}{9} \cdot \frac{1}{4} \cdot \frac{1}{4} \cdot \frac{3}{4} \\
&= \frac{4}{9} \cdot \frac{3}{64} \\
&= \frac{12}{576}
\end{aligned}
$$

Step 4: Compare Posteriors Let's now compare posteriors computed above and make decision,

$$P(\text{Mammal} \mid \text{Fur} = \text{Yes}, \text{Feathers} = \text{No}, \text{Swim} = \text{Yes}) \approx 0.1067$$

$$P(\text{Bird} \mid \text{Fur} = \text{Yes}, \text{Feathers} = \text{No}, \text{Swim} = \text{Yes}) \approx 0.0208$$

Since $P(\text{Mammal} \mid \text{Fur} = \text{Yes}, \text{Feathers} = \text{No}, \text{Swim} = \text{Yes}) > P(\text{Bird} \mid \text{Fur} = \text{Yes}, \text{Feathers} = \text{No}, \text{Swim} = \text{Yes})$, we predict the new animal as **Mammal**.

2.9 K-Nearest Neighbor (KNN)

The k-Nearest Neighbors (k-NN) algorithm is a non-parametric, instance-based learning algorithm that is used for both classification and regression tasks. It relies on the distance metric to determine the similarity between data points.

Algorithm 6 K-Nearest Neighbors Algorithm

```
Require: Training dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{M}$, query point $\mathbf{x}_q$, number of neighbors $k$, distance
    function distance($\mathbf{a}, \mathbf{b}$) (e.g., Euclidean distance)
Ensure: Predicted label or value $\hat{y}_q$ for $\mathbf{x}_q$
 1: Initialize an empty list distances = []
 2: for each $(\mathbf{x}_i, y_i)$ in $D$ do
 3:     $d_i \leftarrow$ distance($\mathbf{x}_q, \mathbf{x}_i$)
 4:     Append $(d_i, y_i)$ to distances
 5: end for
 6: Sort distances in ascending order based on $d_i$
 7: neighbors $\leftarrow$ first $k$ elements of distances
 8: if classification task then
 9:     $\hat{y}_q \leftarrow$ mode(labels in neighbors)
10: else if regression task then
11:     $\hat{y}_q \leftarrow$ mean(values in neighbors)
12: end if
13: return $\hat{y}_q$
```

The choice of k is crucial and can significantly impact the performance of the KNN algorithm. A small k can make the model sensitive to noise in the data, while a large k might smooth out important local patterns. Techniques like cross-validation can be used to find an optimal value for k.

It's also important to note that the KNN algorithm is sensitive to the scale of the features. Therefore, it's often recommended to preprocess the data by applying feature scaling techniques such as standardization or normalization before applying the algorithm.

KNN - Step by step

1. Initialize an empty list called `distances`.
2. For each data point $(\mathbf{x}_i, y_i)$ in the training dataset D:
   (a) Calculate the distance between the query point $\mathbf{x}_q$ and $\mathbf{x}_i$.
   (b) Append the pair $(distance, y_i)$ to the `distances` list.
3. Sort the `distances` list in ascending order based on the calculated distances.
4. Select the first k elements from the sorted `distances` list as the nearest neighbors.
5. If it's a classification task:
   - Assign the most frequent label among the k nearest neighbors to the query point.
6. If it's a regression task:
   - Assign the average value of the k nearest neighbors to the query point.
7. Return the predicted label or value for the query point.

2.9.1 Distance Metric

The most commonly used distance metric in k-NN is the Euclidean distance. For two points $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$, the Euclidean distance $d(x, y)$ is given by:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Other distance metrics, such as Manhattan distance or Minkowski distance, can also be used depending on the problem requirements. The choice of distance metric can significantly impact the performance of the k-NN algorithm. Here are some other commonly used distance metrics:

1. **Manhattan Distance (L1 Norm):**
$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} |x_i - y_i|$$
2. **Minkowski Distance:**
$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$
Note: Euclidean is a special case where $p = 2$, and Manhattan where $p = 1$.
3. **Cosine Similarity:**
$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$
Note: Convert to distance as $d = 1 - similarity$.

4. **Hamming Distance:**

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \nVdash_{x_i \neq y_i}$$

Used for categorical variables, counts the number of disagreements.

5. **Mahalanobis Distance:**

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^T S^{-1} (\mathbf{x} - \mathbf{y})}$$

Where S is the covariance matrix of the dataset.

Choosing a Metric: The choice of distance metric depends on the nature of your data:

- Euclidean distance is suitable for continuous variables in low-dimensional space.
- Manhattan distance can be preferable in high-dimensional spaces.
- Cosine similarity is often used for text data or when the magnitude of vectors is not important.
- Hamming distance is used for binary or categorical data.
- Mahalanobis distance accounts for correlations in the dataset and is scale-invariant.

It's often beneficial to experiment with different metrics to find the one that performs best for your specific dataset and problem.

Pen & Paper

Consider a simple 2D dataset with two classes. We want to classify a new point $x_q = (2.5, 3.5)$ using k-NN with $k = 3$ in the following data.

x_1	x_2	Class
1.0	1.0	A
2.0	2.0	A
3.0	3.0	B
4.0	4.0	B
5.0	5.0	B

First, calculate the Euclidean distance between x_q and all points in the training dataset:

$$\begin{aligned}
d((2.5, 3.5), (1.0, 1.0)) &= \sqrt{(2.5-1.0)^2 + (3.5-1.0)^2} \\
&= \sqrt{1.5^2 + 2.5^2} = \sqrt{2.25 + 6.25} \\
&= \sqrt{8.5} \approx 2.92 \\
d((2.5, 3.5), (2.0, 2.0)) &= \sqrt{(2.5-2.0)^2 + (3.5-2.0)^2} \\
&= \sqrt{0.5^2 + 1.5^2} = \sqrt{0.25 + 2.25} \\
&= \sqrt{2.5} \approx 1.58 \\
d((2.5, 3.5), (3.0, 3.0)) &= \sqrt{(2.5-3.0)^2 + (3.5-3.0)^2} \\
&= \sqrt{0.5^2 + 0.5^2} = \sqrt{0.25 + 0.25} \\
&= \sqrt{0.5} \approx 0.71 \\
d((2.5, 3.5), (4.0, 4.0)) &= \sqrt{(2.5-4.0)^2 + (3.5-4.0)^2} \\
&= \sqrt{1.5^2 + 0.5^2} = \sqrt{2.25 + 0.25} \\
&= \sqrt{2.5} \approx 1.58 \\
d((2.5, 3.5), (5.0, 5.0)) &= \sqrt{(2.5-5.0)^2 + (3.5-5.0)^2} \\
&= \sqrt{2.5^2 + 1.5^2} = \sqrt{6.25 + 2.25} \\
&= \sqrt{8.5} \approx 2.92
\end{aligned}$$

Let's now select 3 nearest neighbors. The 3 nearest neighbors are:

x_1	x_2	Class
3.0	3.0	B
2.0	2.0	A
4.0	4.0	B

The most common class among the 3 nearest neighbors is **B**. Therefore, the predicted class for x_q is **B**.

2.10 Decision Trees

Decision trees are a type of supervised learning algorithm used for both classification and regression tasks. They work by splitting the data into subsets based on the value of input features, creating a tree-like model of decisions.

Mathematical Form of a Decision Tree A decision tree can be represented as a function that maps input features $\mathbf{x}$ to a target value y. For classification, the function can be expressed as:

$$y = f(\mathbf{x}) = \sum_{j=1}^{J} c_j I(\mathbf{x} \in R_j) \tag{2.58}$$

where J is the number of leaf nodes, c_j is the class label for leaf j, and R_j is the region defined by the decision rules from the root to leaf j.

For regression, the function is:

$$y = f(\mathbf{x}) = \sum_{j=1}^{J} \hat{y}_j I(\mathbf{x} \in R_j) \tag{2.59}$$

where $\hat{y}_j$ is the predicted value for region R_j.

2.10.1 Algorithms for Decision Trees

Several algorithms can be used to construct decision trees, each with different criteria for splitting the nodes. Here are some common algorithms:

1. ID3 (Iterative Dichotomiser 3) ID3 uses information gain to decide the splitting criterion. Information gain is based on the concept of entropy from information theory.

- Entropy of a set S:

  $$H(S) = -\sum_{i=1}^{c} p_i \log_2 p_i \tag{2.60}$$

  where p_i is the proportion of examples in class i.

- Information Gain for attribute A:

  $$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v) \tag{2.61}$$

  where S_v is the subset of S for which attribute A has value v.

2. C4.5 C4.5 is an extension of ID3 that handles both continuous and discrete attributes and uses a modified information gain ratio for splitting.

- Gain Ratio for attribute A:

  $$GR(S, A) = \frac{IG(S, A)}{H_A(S)} \tag{2.62}$$

  where

  $$H_A(S) = -\sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \log_2 \left(\frac{|S_v|}{|S|} \right) \tag{2.63}$$

3. CART (Classification and Regression Trees) CART constructs binary trees using the Gini impurity measure for classification and variance reduction for regression.

- Gini Impurity for a set S:

  $$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2 \tag{2.64}$$

  where p_i is the proportion of examples in class i.

- Variance Reduction for regression:

  $$Var(S) = \frac{1}{|S|} \sum_{i=1}^{|S|} (y_i - \bar{y})^2 \tag{2.65}$$

  where $\bar{y}$ is the mean of the target values in set S.

Algorithm 7 ID3 Decision Tree Algorithm

```
Require: Training data $D$, Feature set $F$, Target attribute $target$
Ensure: Decision Tree
1: function ID3($D$, $F$, $target$)
2:     if all examples in $D$ have the same $target$ value then
3:         return leaf node with that $target$ value
4:     else if $F$ is empty then
5:         return leaf node with most common $target$ value in $D$
6:     else
7:         best_feature $\leftarrow \arg\max_{f \in F}$ InformationGain$(D, f)$
8:         $tree \leftarrow$ new decision tree with root test $best\_feature$
9:         for each value $v$ of $best\_feature$ do
10:            $D_v \leftarrow$ subset of $D$ where $best\_feature$ has value $v$
11:            if $D_v$ is empty then
12:                Add leaf node to $tree$ with most common $target$ value in $D$
13:            else
14:                $subtree \leftarrow$ ID3$(D_v, F \setminus \{best\_feature\}, target)$
15:                Add $subtree$ to $tree$ as child corresponding to $best\_feature = v$
16:            end if
17:        end for
18:        return $tree$
19:    end if
20: end function
21: function InformationGain($D$, $feature$)
22:    $H(D) \leftarrow -\sum_{c \in C} p(c) \log_2 p(c)$        ▷ Entropy of dataset
23:    $H(D|feature) \leftarrow \sum_{v \in values(feature)} \frac{|D_v|}{|D|} H(D_v)$
24:    return $H(D) - H(D|feature)$
25: end function
26: function Predict($tree$, $example$)
27:    if $tree$ is a leaf node then
28:        return the $target$ value of the leaf node
29:    else
30:        $feature \leftarrow$ the test at the root of $tree$
31:        $v \leftarrow$ value of $feature$ in $example$
32:        $subtree \leftarrow$ child of $tree$ corresponding to $feature = v$
33:        return Predict($subtree$, $example$)
34:    end if
35: end function
```

4. CHAID (Chi-squared Automatic Interaction Detection) CHAID uses the chi-squared test to find the best split. It can handle both categorical and continuous data by merging categories that are not significantly different.

- Chi-squared statistic for attribute A:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \tag{2.66}$$

  where O_{ij} is the observed frequency and E_{ij} is the expected frequency of category i and class j.

Pen & Paper

Consider a given dataset:

Fur	Feathers	Swim	Type
Yes	No	No	Mammal
Yes	Yes	No	Bird
No	No	Yes	Mammal
Yes	No	Yes	Mammal
No	Yes	No	Bird
No	No	No	Mammal
No	Yes	Yes	Bird
Yes	Yes	Yes	Mammal
No	Yes	Yes	Bird

Entropy Calculation Entropy $H(\text{Type})$ measures the impurity of the dataset with respect to the target variable 'Type'.

$$H(\text{Type}) = - \sum_{c \in \{\text{Mammal}, \text{Bird}\}} P(c) \log_2 P(c)$$

where $P(\text{Mammal}) = \frac{5}{9}$ and $P(\text{Bird}) = \frac{4}{9}$.

$$H(\text{Type}) = - \left(\frac{5}{9} \log_2 \frac{5}{9} + \frac{4}{9} \log_2 \frac{4}{9} \right) \approx 0.991$$

This indicates a relatively high initial entropy, implying the dataset contains mixed types.

Information Gain Calculation Information Gain $\text{IG}(A)$ for a feature A is the reduction in entropy achieved by partitioning the data according to A.
For Fur
Entropy $H(\text{Type}|\text{Fur} = \text{Yes})$:

$$H(\text{Type}|\text{Fur} = \text{Yes}) = - \left(\frac{3}{4} \log_2 \frac{3}{4} + \frac{1}{4} \log_2 \frac{1}{4} \right) \approx 0.811$$

Entropy $H(\text{Type}|\text{Fur} = \text{No})$:

$$H(\text{Type}|\text{Fur} = \text{No}) = - \left(\frac{2}{5} \log_2 \frac{2}{5} + \frac{3}{5} \log_2 \frac{3}{5} \right) \approx 0.971$$

$$\text{IG(Fur)} \approx 0.991 - \left(\frac{4}{9} \cdot 0.811 + \frac{5}{9} \cdot 0.971\right) \approx 0.071$$

The Information Gain (IG) for Fur is calculated to be 0.071, indicating that Fur provides the highest reduction in entropy among the features considered.
For Feathers
Entropy $H(\text{Type}|\text{Feathers} = \text{Yes})$:

$$H(\text{Type}|\text{Feathers} = \text{Yes}) = -\left(\frac{2}{4}\log_2\frac{2}{4} + \frac{2}{4}\log_2\frac{2}{4}\right) = 1.0$$

Entropy $H(\text{Type}|\text{Feathers} = \text{No})$:

$$H(\text{Type}|\text{Feathers} = \text{No}) = -\left(\frac{3}{5}\log_2\frac{3}{5} + \frac{2}{5}\log_2\frac{2}{5}\right) \approx 0.971$$

$$\text{IG(Feathers)} \approx 0.991 - \left(\frac{4}{9} \cdot 1.0 + \frac{5}{9} \cdot 0.971\right) \approx 0.026$$

Feathers provide the least Information Gain (IG), indicating it is less effective for splitting the dataset compared to other features.
For Swim
Entropy $H(\text{Type}|\text{Swim} = \text{Yes})$:

$$H(\text{Type}|\text{Swim} = \text{Yes}) = -\left(\frac{3}{4}\log_2\frac{3}{4} + \frac{1}{4}\log_2\frac{1}{4}\right) \approx 0.811$$

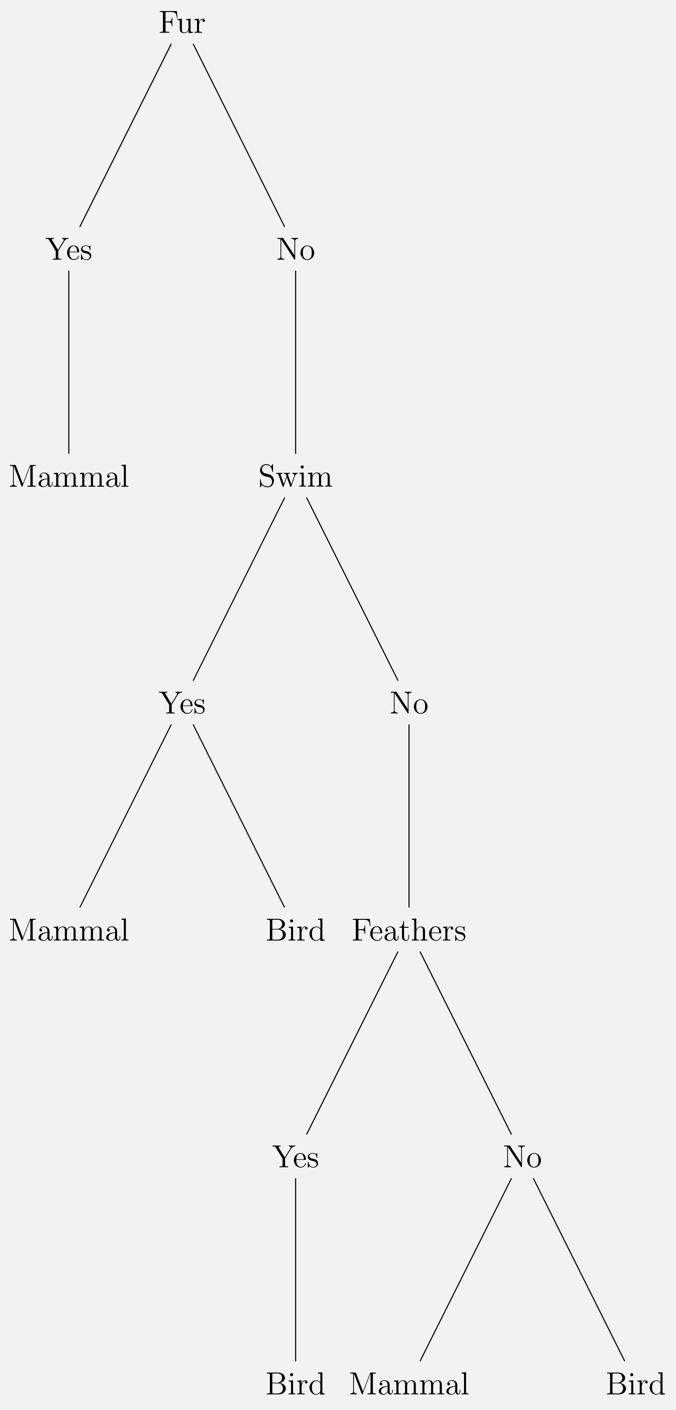
Entropy $H(\text{Type}|\text{Swim} = \text{No})$:

$$H(\text{Type}|\text{Swim} = \text{No}) = -\left(\frac{2}{5}\log_2\frac{2}{5} + \frac{3}{5}\log_2\frac{3}{5}\right) \approx 0.971$$

$$\text{IG(Swim)} \approx 0.991 - \left(\frac{4}{9} \cdot 0.811 + \frac{5}{9} \cdot 0.971\right) \approx 0.071$$

Swim provides a similar Information Gain (IG) to Fur, suggesting it is also effective for partitioning the dataset.

Decision Tree Construction Based on the highest Information Gain, Fur is chosen as the root node for the decision tree. Here's how the tree is constructed:



The decision tree begins with a split on the 'Fur' attribute. For animals that have fur ('Fur = Yes'), all instances in our dataset belong to the 'Mammal' category. This means if an animal has fur, it is classified as a mammal. For animals without fur ('Fur = No'), the dataset further splits based on whether they swim ('Swim'). Among animals without fur but that do swim ('Swim = Yes'), most are mammals except one, which is a bird. If an animal does not have fur and does not swim ('Swim = No'), the classification then depends on whether they have feathers ('Feathers'). If they do have feathers ('Feathers = Yes'), all these animals are birds. If they lack feathers ('Feathers = No'), they include a mix of both mammals and birds.

2.11 Multi-class Classification

In machine learning, multi-class classification is a problem where we need to classify instances into one of three or more classes. This is an extension of binary classification, which deals with only two classes. While binary classification techniques like logistic regression can be adapted for multi-class problems (e.g., using one-vs-rest or one-vs-one strategies), there are more efficient approaches specifically designed for multi-class scenarios.

2.11.1 Softmax Regression for Multi-class Classification

Softmax Regression, also known as Multinomial Regression, is a generalization of logistic regression for multi-class classification problems. It's particularly useful when we need to assign probabilities to each class in a multi-class problem.
The softmax function is given by:

$$h(x) = g(z_j) = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)} \tag{2.67}$$

where z_j is the input to the softmax function for class j, and K is the total number of classes.

The softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between 0 and 1 that sum to 1, which can be interpreted as probabilities.

Softmax regression, also known as multinomial logistic regression, is a generalization of logistic regression to multiple classes. It models the probability of each class given a set of features.

Pen & Paper

Consider a dataset with three classes (0, 1, and 2) and two features:

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{4}$$

where

$$\mathbf{x}^{(1)} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad y^{(1)} = 0$$

$$\mathbf{x}^{(2)} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \quad y^{(2)} = 1$$

$$\mathbf{x}^{(3)} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \quad y^{(3)} = 2$$

$$\mathbf{x}^{(4)} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \quad y^{(4)} = 1$$

Assume we have a weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ initialized as follows:

$$\mathbf{W} = \begin{pmatrix} 0.2 & 0.4 & 0.6 \\ 0.5 & 0.3 & 0.2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$$

Solutions: Let's get started with the linear combinations:
For an instance $\mathbf{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$:

$$\mathbf{z} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$$

$$\mathbf{z} = \begin{pmatrix} 0.2 & 0.5 \\ 0.4 & 0.3 \\ 0.6 & 0.2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$$

$$\mathbf{z} = \begin{pmatrix} 0.2 \cdot 1 + 0.5 \cdot 2 \\ 0.4 \cdot 1 + 0.3 \cdot 2 \\ 0.6 \cdot 1 + 0.2 \cdot 2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$$

$$\mathbf{z} = \begin{pmatrix} 0.2 + 1.0 \\ 0.4 + 0.6 \\ 0.6 + 0.4 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \end{pmatrix}$$

$$\mathbf{z} = \begin{pmatrix} 1.2 \\ 1.2 \\ 1.3 \end{pmatrix}$$

Now, let's compute the softmax score:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{3} e^{z_k}} \quad \text{for } j = 1, 2, 3$$

$$\sigma(\mathbf{z}) = \begin{pmatrix} \frac{e^{1.2}}{e^{1.2}+e^{1.2}+e^{1.3}} \\ \frac{e^{1.2}}{e^{1.2}+e^{1.2}+e^{1.3}} \\ \frac{e^{1.3}}{e^{1.2}+e^{1.2}+e^{1.3}} \end{pmatrix}$$

compute $e^{1.2}$ and $e^{1.3}$:

$$e^{1.2} \approx 3.320, \quad e^{1.3} \approx 3.669$$

Substitute these values into the softmax function:

$$\sigma(\mathbf{z}) = \begin{pmatrix} \frac{3.320}{3.320+3.320+3.669} \\ \frac{3.320}{3.320+3.320+3.669} \\ \frac{3.669}{3.320+3.320+3.669} \end{pmatrix}$$

$$\sigma(\mathbf{z}) = \begin{pmatrix} \frac{3.320}{10.309} \\ \frac{3.320}{10.309} \\ \frac{3.669}{10.309} \end{pmatrix}$$

$$\sigma(\mathbf{z}) = \begin{pmatrix} 0.322 \\ 0.322 \\ 0.356 \end{pmatrix}$$

The softmax scores represent the probabilities of the instance $\mathbf{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ belonging to each class:

$$P(y = 0 \mid \mathbf{x}) = 0.322, \quad P(y = 1 \mid \mathbf{x}) = 0.322, \quad P(y = 2 \mid \mathbf{x}) = 0.356$$

The model predicts class 2 for this instance because it has the highest probability.

2.12 Ensemble Learning

Ensemble learning is a powerful technique in machine learning that combines multiple models to improve prediction accuracy and robustness. The concept draws inspiration from real-world decision-making processes where we often seek multiple opinions before making important choices. In machine learning, an ensemble refers to a collection of models that work together to solve a particular problem. These models, often called base learners or weak learners, collaborate to produce better predictions than any single model could achieve alone.

To illustrate this concept, consider the process of buying a laptop. You might read expert reviews, ask friends for recommendations, compare specifications across brands, and test devices in a store. By combining these diverse sources of information, you aim to make a more informed decision. Ensemble learning operates on a similar principle, aggregating insights from multiple models to arrive at a more accurate and reliable prediction.

The strength of ensemble learning lies in the diversity of its constituent models and the method used to combine their outputs. This diversity allows the ensemble to capture different aspects of the underlying patterns in the data, often leading to improved overall performance.

A crucial concept in understanding ensemble learning is the bias-variance tradeoff. Bias refers to the error due to oversimplified assumptions in the learning algorithm, while variance is the error due to the model's sensitivity to small fluctuations in the training set. Different ensemble techniques target either bias reduction, variance reduction, or both, allowing for fine-tuned management of this tradeoff.

By combining multiple models, ensembles can be more resilient to uncertainties in the data and provide more stable predictions across different datasets. This resilience is particularly valuable in real-world applications where data can be noisy or incomplete.

Ensemble methods can be categorized based on the homogeneity of their constituent models. Homogeneous ensembles use multiple instances of the same type of model, while heterogeneous ensembles combine different types of models. Both approaches have their merits and are chosen based on the specific requirements of the problem at hand.

Three common ensemble methods are *bagging*, *boosting*, and *stacking.*

2.12.1 Bagging

Bagging, short for Bootstrap Aggregating, is an ensemble method designed to improve the stability and accuracy of machine learning algorithms. It reduces variance and helps to avoid overfitting. The general approach of bagging can be summarized in the following steps:

1. **Bootstrap Sampling:** Generate B bootstrap samples from the original dataset $\mathcal{D}$. Each bootstrap sample $\mathcal{D}_b$ is created by randomly sampling m instances from $\mathcal{D}$ *with replacement*. This means that some instances from the original dataset may appear multiple times in a bootstrap sample, while others may not appear at all.

2. **Model Training:** Train an independent base model f_b on each of the B bootstrap samples $\mathcal{D}_b$. These base models are typically of the same type (e.g., all decision trees, all support vector machines, etc.).

3. **Prediction Aggregation:** Combine the predictions from all B trained models to form a final, aggregated prediction for new, unseen data points. The aggregation method depends on the nature of the learning task (regression or classification).

2.12.2 Mathematical Model of Bagging:

Let the original dataset be denoted by $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ represents the i-th feature vector in an n-dimensional space, and $y^{(i)}$ is the corresponding target variable. For regression tasks, $y^{(i)} \in \mathbb{R}$, while for classification tasks with K classes, $y^{(i)} \in \{1, 2, \ldots, K\}$.
Bootstrap Sampling: We generate B bootstrap datasets, $\{\mathcal{D}_b\}_{b=1}^{B}$, where each $\mathcal{D}_b$ contains m instances sampled with replacement from $\mathcal{D}$:

$$\mathcal{D}_b = \{(\mathbf{x}_b^{(i)}, y_b^{(i)})\}_{i=1}^{m}, \quad \text{for } b = 1, 2, \ldots, B.$$

Here, each $(\mathbf{x}_b^{(i)}, y_b^{(i)})$ is randomly selected from the original dataset $\mathcal{D}$.
Model Training: For each bootstrap sample $\mathcal{D}_b$, we train a base model $f_b(\mathbf{x})$. This results in a collection of B trained models: $\{f_b(\mathbf{x})\}_{b=1}^{B}$.
Out-of-Bag (OOB) Samples and Error Estimation: An important aspect of bagging is the concept of out-of-bag (OOB) samples. For each bootstrap sample $\mathcal{D}_b$, approximately one-third (more precisely, $(1 - 1/e) \approx 0.632$ of the original data points are included, meaning about $1 - 0.632 = 0.368$ or roughly one-third are left out. These left-out instances for a particular bootstrap sample constitute the OOB samples for the model trained on that sample.
The OOB samples provide a valuable mechanism for estimating the generalization error of the bagged ensemble without the need for a separate validation set. The OOB error is calculated as follows:

1. For each instance $(\mathbf{x}^{(i)}, y^{(i)})$ in the original dataset $\mathcal{D}$, identify the set of models $\{f_b\}$ for which this instance was an OOB sample.

2. Obtain the predictions from these OOB models for $\mathbf{x}^{(i)}$.

3. Aggregate these OOB predictions (by averaging for regression or majority voting for classification) to get an OOB prediction $\hat{y}_{OOB}^{(i)}$.

4. Compare the OOB prediction $\hat{y}_{OOB}^{(i)}$ with the true label $y^{(i)}$ to calculate the error for this instance.

The overall OOB error is then the average of these errors across all instances in the original dataset. This provides an almost unbiased estimate of the bagged ensemble's performance on unseen data.

Prediction Aggregation:

For Regression: For a new instance $\mathbf{x}$, the final prediction $\hat{y}$ from the bagged ensemble is the simple average of the predictions from all B individual models:

$$\hat{y} = \frac{1}{B} \sum_{b=1}^{B} f_b(\mathbf{x}).$$

For Classification: For a new instance $\mathbf{x}$, the final predicted class $\hat{y}$ is determined by majority voting among the predictions of the B individual classifiers. Let $f_b(\mathbf{x}) \in \{1, 2, \ldots, K\}$ be the class label predicted by the b-th model. The final prediction is:

$$\hat{y} = \arg \max_{k \in \{1,\ldots,K\}} \sum_{b=1}^{B} \mathbb{I}(f_b(\mathbf{x}) = k),$$

where $\mathbb{I}(f_b(\mathbf{x}) = k)$ is an indicator function that equals 1 if the b-th model predicts class k, and 0 otherwise.

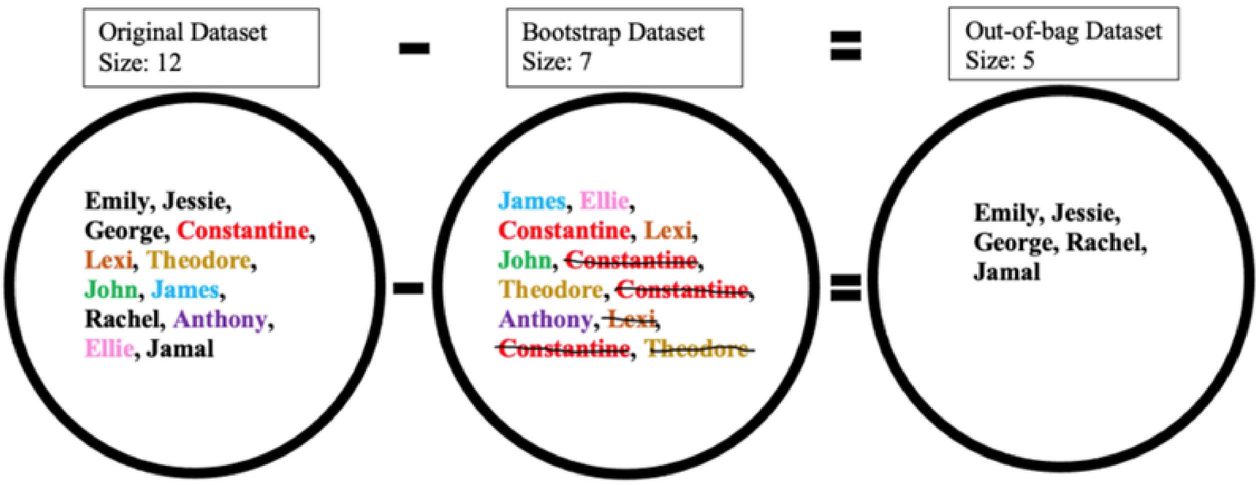

Original Dataset
Size: 12
Bootstrap Dataset
Size: 7
Out-of-bag Dataset
Size: 5
Emily, Jessie, George, Constantine, Lexi, Theodore, John, James, Rachel, Anthony, Ellie, Jamal
James, Ellie, Constantine, Lexi, John, ~~Constantine~~, Theodore, ~~Constantine~~, Anthony, ~~Lexi~~, ~~Constantine~~, ~~Theodore~~
Emily, Jessie, George, Rachel, Jamal

Figure 2.6: Illustration of Out-of-bag Samples: For each bootstrap sample (and the model trained on it), some original data points are left out. These are the OOB samples for that specific model and can be used for evaluation.

2.13 Random Forest: An Extension of Bagging with Feature Randomness

Random Forest is a sophisticated and widely adopted ensemble learning algorithm, particularly effective for both classification and regression tasks. Introduced by Leo Breiman in his seminal 2001 paper, it builds upon the principles of bagging by incorporating an additional layer of randomness in the feature selection process. This combination leads to a powerful and robust model that generally exhibits high predictive accuracy and a reduced tendency to overfit the training data. At its core, a Random Forest operates by constructing a multitude of decision trees during the training phase and then aggregating their predictions—either through majority voting for classification or by averaging for regression.

The remarkable effectiveness of Random Forest stems from the synergistic interplay of two key concepts: **bootstrap aggregating (bagging)** and **random feature selection**.

As discussed earlier, bagging involves training each tree in the forest on a different bootstrap sample of the training data. This introduces diversity among the trees and helps to decrease the variance of the ensemble. Crucially, the out-of-bag (OOB) samples, which are the data points not included in the training set of a particular tree, provide an internal mechanism for performance estimation.

Random feature selection further enhances the diversity of the trees. When growing each tree, at each node, instead of considering all the features to find the best split, the algorithm randomly selects a subset of features (typically of size m, where m is significantly smaller than the total number of features M). The best split is then chosen only from this randomly selected subset. This process ensures that the trees in the forest are decorrelated, which further improves the generalization ability of the ensemble. A common heuristic for the size of the feature subset in classification is $m = \sqrt{M}$.

2.13.1 How the Random Forest Algorithm Works

The Random Forest algorithm proceeds through the following fundamental steps:
1. Bootstrap Sampling: Similar to standard bagging, the algorithm generates multiple bootstrap datasets by randomly sampling the original training data with replacement. Each bootstrap sample has the same size as the original dataset but contains a different subset of instances.
2. Decision Tree Construction with Random Feature Selection: For each bootstrap dataset, a decision tree is grown. However, with a crucial modification: at each node in the tree-building process, instead of considering all the features to determine the best split, the algorithm performs the following:

- A random subset of m features (where $m < M$, the total number of features) is selected.
- The best feature to split the current node is chosen from this randomly selected subset based on a splitting criterion (e.g., Gini impurity or entropy reduction for classification, variance reduction for regression).

Each tree is typically grown to its maximum depth without pruning, which allows the individual trees to have low bias. The randomness introduced in both the data sampling (bootstrapping) and the feature selection at each split contributes to the overall robustness and generalization capability of the forest.
3. Aggregation of Predictions: Once a sufficient number of trees have been grown (determined by the hyperparameter `n_estimators`), the Random Forest makes predictions for new data points by aggregating the predictions of all the individual trees:

- **For Classification Tasks:** The final predicted class is the one that receives the majority of the votes from all the trees in the forest.
- **For Regression Tasks:** The final prediction is the average of the predictions made by all the individual trees.

2.13.2 Key Features and Benefits

The Random Forest algorithm offers several compelling advantages that have contributed to its widespread popularity:

- **High Accuracy:** By combining the predictions of many decorrelated trees, Random Forests typically achieve high predictive accuracy, often outperforming single decision trees and other machine learning algorithms.
- **Robustness to Overfitting:** The bagging and random feature selection techniques work together to reduce the variance of the model and prevent overfitting, making it generalize well to unseen data.
- **Implicit Feature Importance Estimation:** Random Forests provide a natural way to estimate the importance of each feature in the prediction process. This can be valuable for feature selection and understanding the underlying data relationships.

- **Handles High-Dimensional Data:** The random feature selection makes Random Forests well-suited for datasets with a large number of features.
- **Handles Missing Values and Outliers:** The tree-based nature of the algorithm makes it relatively robust to missing values and outliers in the input data.
- **Internal Validation (OOB Error):** As mentioned earlier, the OOB samples provide an efficient way to estimate the generalization error without the need for a separate validation set.
- **Parallelization:** The training of individual trees in a Random Forest can be easily parallelized, which can significantly speed up the training process, especially for large datasets and a large number of trees.

2.13.3 Advantages

In summary, the key advantages of Random Forest include:

- Excellent predictive performance across a wide range of applications.
- Reduced risk of overfitting compared to individual decision trees.
- Ability to handle both continuous and categorical features.
- Provides estimates of feature importance.
- Effective for high-dimensional datasets.
- Offers an internal estimate of generalization error (OOB error).
- Can be parallelized for faster training.
- Generally requires less hyperparameter tuning compared to some other complex algorithms.

2.13.4 Limitations

Despite its numerous strengths, Random Forest also has some limitations:

- **Interpretability:** Random Forests are often considered "black box" models, making it more challenging to interpret the decision-making process compared to a single decision tree. Understanding the combined effect of hundreds or thousands of trees can be difficult.
- **Computational Cost:** Training a large number of trees can be computationally expensive, especially for very large datasets.
- **Prediction Speed:** For real-time applications requiring very fast predictions, the prediction speed of a Random Forest (which involves averaging or voting across many trees) might be slower than that of simpler models.
- **Potential Bias in Imbalanced Datasets:** Similar to other tree-based methods, Random Forests can be biased towards the majority class in imbalanced datasets. Techniques like weighted sampling or using balanced bootstrap samples can help mitigate this issue.
- **Performance on Certain Types of Data:** While generally robust, Random Forests might not perform as well as specialized algorithms for certain types of data or specific problem structures (e.g., very sparse data or linear relationships).

2.13.5 Important Hyperparameters

The performance of a Random Forest model is influenced by several key hyperparameters that need to be carefully tuned:

- **Number of Trees (`n_estimators`):** This parameter controls the number of trees in the forest. Generally, increasing the number of trees tends to improve the stability and reduce overfitting, up to a point where the improvement plateaus. However, a larger number of trees also increases the training time.
- **Maximum Depth of the Trees (`max_depth`):** This limits the maximum depth of each individual tree. Setting a maximum depth can help to prevent overfitting, especially for noisy datasets. If set to `None`, trees are grown until all leaves are pure or until all leaves contain less than `min_samples_split` samples.
- **Minimum Number of Samples Required to Split an Internal Node (`min_samples_split`):** This specifies the minimum number of samples required to split an internal node. Increasing this value can help to regularize the model and prevent overfitting.
- **Minimum Number of Samples Required at a Leaf Node (`min_samples_leaf`):** This specifies the minimum number of samples required to be at a leaf node. A higher value can smooth the decision boundaries and reduce overfitting.
- **Maximum Number of Features to Consider When Looking for the Best Split (`max_features`):** This parameter controls the size of the random subset of features to consider at each node split. Common choices include `'sqrt'` (the square root of the total number of features), `'log2'` (the base-2 logarithm of the total number of features), or a fraction of the total number of features. Reducing this value can increase the diversity among the trees and reduce overfitting.
- **Bootstrap (`bootstrap`):** A boolean parameter indicating whether bootstrap samples should be used when building trees. It is usually set to `True` for Random Forest.
- **Out-of-Bag Score (`oob_score`):** A boolean parameter indicating whether to use out-of-bag samples to estimate the generalization score. Setting it to `True` can provide a useful internal validation of the model's performance.

2.14 Feature Importance in Random Forest

Random Forests offer valuable insights into the relative importance of the input features in the prediction process. Two primary methods are commonly used to evaluate feature importance:

- **Mean Decrease in Impurity (MDI) / Gini Importance:** This method measures the average decrease in impurity (e.g., Gini impurity for classification, variance for regression) across all trees in the forest when a particular feature is used for splitting. Features that lead to a larger decrease in impurity are considered more important. This method is computationally efficient as it is calculated during the training process. However, it can be biased towards features with more categories (for categorical features) or higher cardinality.
- **Permutation Importance / Mean Decrease in Accuracy (MDA):** This method directly assesses the impact of a feature on the model's performance. After the model is trained, for each feature, the values of that feature are randomly shuffled (permuted)

across the OOB samples (or a separate validation set). The decrease in model accuracy (or increase in error) resulting from this permutation is then measured. A larger decrease in accuracy indicates that the feature was more important to the model's predictions. Permutation importance is generally considered more reliable than MDI as it directly measures the feature's impact on performance and is less prone to biases related to feature cardinality. However, it is computationally more expensive as it requires performing predictions multiple times after shuffling each feature.

Understanding feature importance can be crucial for tasks such as feature selection (identifying the most relevant features), gaining insights into the underlying data relationships, and building more interpretable models by focusing on the most influential variables.

2.14.1 Boosting

Boosting, on the other hand, trains models sequentially, with each new model focusing on the errors of the previous ones. It assigns higher weights to misclassified instances in subsequent iterations, allowing the ensemble to gradually improve its performance on difficult cases. The basic idea of Boosting is to train weak learners sequentially, each trying to correct is predecessor.

Thus, Boosting is an ensemble learning technique which builds a robust machine learning model by eliminating the weaknesses of weak learners. This is achieved through combining multiple weak learners which has high bias but less variance and turning them together into strong learners by minimizing errors. In boosting each training sample is assigned an equal weight, initially. Then, fitted with the model and for the samples with wrong prediction, weight for them is increased. Then the succeeding model tries to compensate for the weakness of its preceding model. With each succeeding model, the weak rules from preceding model are utilized to form a strong model.

AdaBoost, Gradient Boosting Machines (GBM), and XGBoost are well-known boosting algorithms.

For example, we have a task on hand to classify whether a given model is SPAM or NOT-SPAM. To classify emails as SPAM, say, we have to following rules learned by each learner:

Classifier 1: *Emails that contains only links are SPAM.*
Classifier 2: *Emails That contains a word like 'million', '$', 'congratulations' etc. are SPAM.*
Classifier 3*: Emails from unknown senders are SPAM.*

Individually, these rules are not enough to categorize email as SPAN or NOT-SPAM. And model that learns a single rule is a weak model or weak learner. However, together all three models form a strong rule for SPAM classification. This is exactly what boosting does using different weak classifier in each iteration and forms a strong learner.

Mathematical Model of Boosting Consider a classifer C for binary classification which predicts among 1, -1 for any input X, then the error rate on each training sample is given as:

$$err = 1/N \sum_{i=1}^{N} I(y_i! = C(x_i)) \tag{2.68}$$

and the expected error rate on future predictions is $E_{XY} I(Y! = C(X))$.

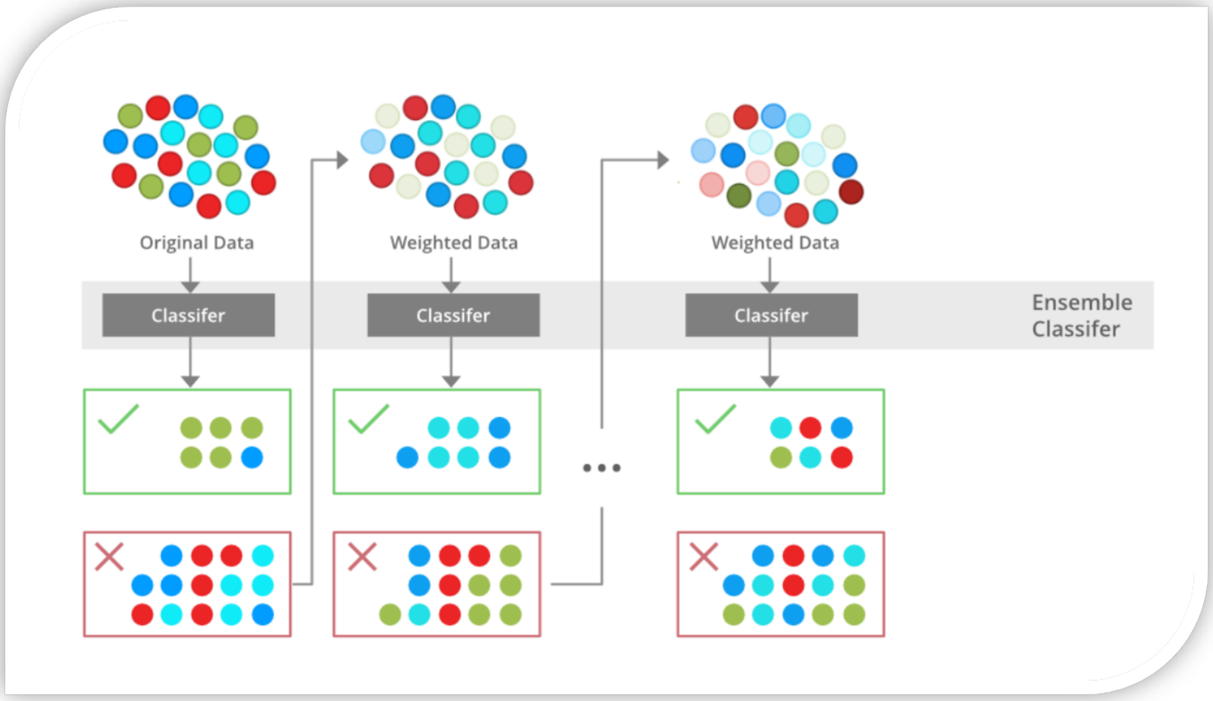


Figure 2.7: Illustration of Boosting

The main purpose of boosting technique is to sequentially apply the weak classification algorithm to the weighted data such that a sequence of weak classifiers $C_k(x), k = 1, 2, ..., K$. The prediction from all of them are combined through a weighted average technique to produce a final prediction through strong classifier as:

$$C(x) = sign(\sum_{k=1}^{K} \alpha_k C_k(x)) \tag{2.69}$$

where, α_k is computed by the boosting algorithm and is the weight contribution of each respective $C_k(x)$.
Initially, all of the weights are set to $w_i = 1/m$ where there are m no. of training examples. Boosting creates an ensemble of weak learners. For each data sample, algorithm assigns equal weights, initially. But assigns higher weights for those samples to which first weak learner predicts incorrectly. then a higher weight is assigned to those samples and the data is again fed to the second learner. Output from this second learner is again analyzed and for the samples with incorrect prediction, a higher weight is assigned. Then the new weighted inputs are again fed to the third learner and so on.
Steps:

1. First, equal weight is assigned to each data sample and first ML model is trained. The prediction from this model is analyzed.

2. The boosting algorithm asses the output of first model and increases the weight of samples for which incorrect prediction was made by the first model. Then data wight new weight is again fed to the second model which makes prediction again. The output of second model is analyzed and same as above the new weight is assigned to the samples with incorrect predictions.

3. The input is given to third model with new weight and prediction of it is assessed again

4. This process continues until the error is below the expected level.

The second model only focuses on the shortcomings of the first model. This is how the model improves their performance by correcting the mistakes the preceding one did.
Let's focus on the working of AdaBoost step by step to better understand the topic. AdaBoost is an one of the boosting technique that combines multiple weak classifiers to create a strong classifier. It adjusts the weights of incorrectly classified instances so that subsequent classifiers focus more on difficult cases.

Step-by-Step Process

1. **Initialize Weights**:

$$w_i = \frac{1}{m} \quad \text{for } i = 1, \ldots, m \tag{2.70}$$

where m is the number of training instances.
2. **Train Weak Learners**:

- For each iteration t from 1 to T (the total number of iterations or weak learners):

    1. **Train a Weak Learner**:
        - A weak learner $h_t(x)$ is trained on the training data using the current weights w_i.
    2. **Calculate the Weighted Error**:

    $$\epsilon_t = \sum_{i=1}^{m} w_i \cdot I(y_i \neq h_t(x_i)) \tag{2.71}$$

    where $I(\cdot)$ is the indicator function that equals 1 if the condition is true and 0 otherwise.
    3. **Compute the Learner's Weight**:

    $$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{2.72}$$

    This weight measures the importance of the weak learner in the final model. A lower error leads to a higher weight.
    4. **Update the Weights**:

    $$w_i \leftarrow w_i \cdot \exp(\alpha_t \cdot I(y_i \neq h_t(x_i))) \tag{2.73}$$

    Normalize the weights so that they sum to 1:

    $$w_i \leftarrow \frac{w_i}{\sum_{j=1}^{m} w_j} \tag{2.74}$$

    This ensures that the distribution of weights remains a probability distribution.

3. **Combine Weak Learners**:

- After T iterations, the final strong classifier $H(x)$ is formed by combining the weak learners, weighted by their respective α_t values:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t \cdot h_t(x) \right) \tag{2.75}$$

- The sign$(\cdot)$ function returns the predicted class label.

2.14.2 Stacking

Stacking involves training multiple diverse base models and then using another model, called a meta-learner, to combine the predictions of these base models. This method can leverage heterogeneous base models, potentially capturing a wider range of patterns in the data. A metal-learner takes input a prediction value from the various model and learns to approximate final prediction. The prediction value from machine leanings are the feature input for the meta-learner. This final layer of meta-learner is stacked on top of other machine learning models hence, the name Stacking.

Mathematical Model of Stacking: Let the base models be denoted as $h_1, h_2, \ldots, h_M$, where M is the number of base models. Each base model is trained on the training dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$. Each base model h_j produces a prediction for an instance $\mathbf{x}$:

$$z_j^{(i)} = h_j(\mathbf{x}^{(i)}) \quad \text{for } i = 1, \ldots, m \tag{2.76}$$

The predictions form a new dataset $\mathcal{D}' = \{(z_1^{(i)}, z_2^{(i)}, \ldots, z_M^{(i)}, y^{(i)})\}_{i=1}^m$.
A meta-learner H is trained on the new dataset $\mathcal{D}'$. The meta-learner uses the predictions of the base models as input features to make the final prediction:

$$H(\mathbf{z}^{(i)}) = H(z_1^{(i)}, z_2^{(i)}, \ldots, z_M^{(i)}) \tag{2.77}$$

For a new instance $\mathbf{x}$, the final prediction is made by first obtaining the predictions of the base models and then applying the meta-learner:

$$\hat{y} = H(h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_M(\mathbf{x})) \tag{2.78}$$

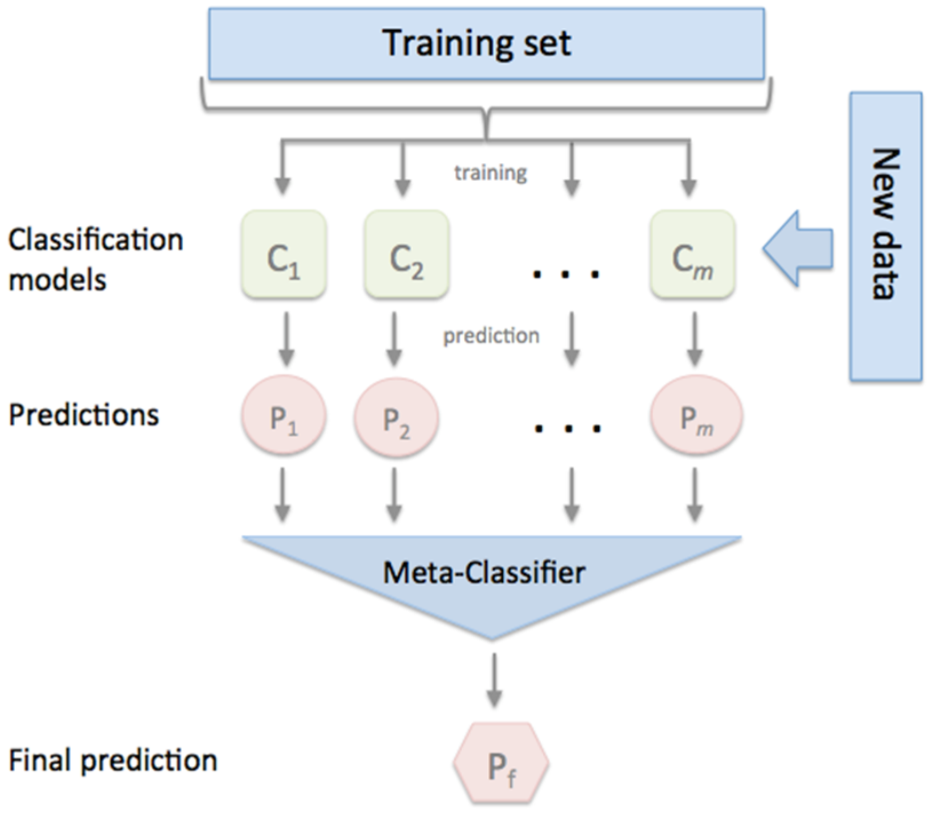


Figure 2.8: Illustration of Stacking

2.14.3 Advantages and Challenges of Ensemble Learning

Ensemble learning offers several advantages. It typically improves prediction accuracy and allows for better generalization to new data. The combination of multiple models often reduces the risk of overfitting and enables the handling of complex, non-linear relationships in data. However, these benefits come with certain challenges. Ensemble methods generally increase computational complexity, as multiple models need to be trained and maintained. There's also a potential loss of interpretability, as the final prediction is the result of combining multiple models' outputs. Care must be taken to avoid overfitting, particularly when using complex ensembles. Lastly, the effective application of ensemble learning requires careful selection of base models and ensemble methods appropriate to the problem at hand.

Chapter 3

Unsupervised Learning

3.1 Introduction to Unsupervised Learning

In contrast to supervised learning, where training examples consist of input features paired with corresponding labels, unsupervised learning methodologies operate on datasets containing only feature attributes, devoid of any explicit labels. The primary objective of unsupervised learning is to discern inherent structures and patterns within the unlabeled data, typically by identifying natural groupings or reducing the dimensionality of the feature space.
Two prominent categories of unsupervised learning techniques are:

- **Clustering**: Aims to group similar data points together based on their intrinsic characteristics.
- **Dimensionality Reduction**: Seeks to reduce the number of features while preserving essential information.

Given the absence of labeled data, unsupervised learning algorithms autonomously learn underlying patterns and relationships directly from the feature distributions.

3.2 Clustering

Clustering is the unsupervised task of partitioning a dataset into distinct groups, or clusters, such that data points within the same cluster exhibit a high degree of similarity, while data points in different clusters are dissimilar. Each identified cluster can often be represented by a central point known as a centroid, which encapsulates the typical characteristics of the observations within that cluster.

Illustrative Applications of Clustering

- **Apparel Sizing**: Categorizing individuals into size groups (e.g., small, medium, large) based on body measurements for efficient apparel manufacturing and inventory management, balancing cost-effectiveness with adequate fit.
- **Market Segmentation**: Identifying distinct groups of customers with shared characteristics (e.g., demographics, purchasing behavior, preferences) to enable targeted and more effective marketing strategies.
- **Document Organization**: Structuring a collection of text documents into a hierarchical topic structure based on the semantic similarity of their content, facilitating information retrieval and knowledge discovery.

- **Anomaly Detection**: Grouping e-commerce transactions to distinguish between normal and potentially fraudulent activities based on patterns in transaction features.

Similarity and Dissimilarity Measures in Clustering The effectiveness of clustering algorithms hinges on the ability to quantify the similarity or dissimilarity between data points. Various techniques are employed for this purpose, including:

- **Distance Metrics**: Quantify the separation between data points in the feature space (e.g., Euclidean distance, Manhattan distance). Algorithms like K-Means heavily rely on distance measures to determine cluster assignments.

- **Density-Based Measures**: Identify clusters as dense regions of data points separated by sparser areas. The DBSCAN algorithm is a prominent example that utilizes density-based similarity.

- **Cosine Similarity**: Measures the cosine of the angle between two non-zero vectors, often used in Natural Language Processing (NLP) to assess the similarity between words or documents based on their vector representations.

The choice of similarity or dissimilarity measure is crucial and depends on the nature of the data and the specific goals of the clustering task. Different clustering algorithms leverage these measures in distinct ways to uncover the underlying structure in the data.

3.2.1 K-Means Clustering

K-Means is a widely adopted and conceptually straightforward partitioning-based clustering algorithm. It aims to partition a dataset into a predefined number of K distinct, non-overlapping clusters. The algorithm employs the Euclidean distance (or other suitable distance metrics) as a measure of dissimilarity to group data points. The "Means" in K-Means refers to the method of updating cluster representatives, which are calculated as the arithmetic mean (centroid) of the data points assigned to each cluster.
Key characteristics of the K-Means algorithm include:

- **Predefined Number of Clusters (K)**: The algorithm requires the number of clusters, K, to be specified a priori. This is a crucial parameter that influences the resulting clustering.

- **Iterative Refinement**: K-Means iteratively refines the cluster assignments and centroids to minimize the within-cluster variance, effectively aiming for compact and well-separated clusters.

- **Sensitivity to Initialization**: The initial placement of the K cluster centers can significantly impact the final clustering outcome. Different initializations may lead to convergence to different local optima. Strategies such as initializing centroids far from each other or employing multiple random initializations are often used to mitigate this sensitivity.

Detailed Steps of the K-Means Algorithm

The K-Means algorithm proceeds through the following iterative steps:

Algorithm 8 K-Means Algorithm

```
1: Input: Dataset $X = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$ where each $\mathbf{x}_i \in \mathbb{R}^n$, number of clusters $k$
2: 1. Initialize $k$ cluster centers $\mathbf{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_k\}$ randomly from the data points or the
   feature space.
3: repeat
4:     2. Assignment Step: Assign each data point $\mathbf{x}_i$ to the cluster whose centroid is the
       nearest, according to a distance metric (e.g., Euclidean distance):
           $c^{(i)} = \arg \min_{j \in \{1, \dots, k\}} \|\mathbf{x}_i - \mathbf{c}_j\|^2$
5:     3. Update Step: Recalculate the centroid of each cluster $j$ as the mean of all data
       points assigned to that cluster:
           $\mathbf{c}_j = \frac{1}{|\mathcal{S}_j|} \sum_{\mathbf{x}_i \in \mathcal{S}_j} \mathbf{x}_i$
       where $\mathcal{S}_j = \{i : c^{(i)} = j\}$ is the set of indices of data points belonging to cluster $j$, and $|\mathcal{S}_j|$
       is the number of data points in cluster $j$.
6: until the cluster assignments no longer change or a maximum number of iterations is
   reached
7: Output: A set of $k$ clusters and their corresponding centroids $\mathbf{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_k\}$.
```

1. **Initialization**: Given a dataset X with m observations and n features, and a desired number of clusters k, initialize k cluster centroids. This can be done by randomly selecting k data points from the dataset or by randomly initializing k points within the feature space.

2. **Assignment**: For each data point $\mathbf{x}_i$ in the dataset, calculate the distance to each of the k cluster centroids. Assign the data point to the cluster whose centroid is the closest based on the chosen distance metric (typically the squared Euclidean distance).

3. **Update**: Once all data points have been assigned to clusters, recalculate the centroid for each cluster. The new centroid is the mean of all the data points that belong to that cluster.

4. **Convergence Check**: Compare the current cluster assignments with the assignments from the previous iteration. If the assignments have not changed for any data point, or if a predefined number of iterations has been reached, the algorithm has converged, and the process terminates.

5. **Iteration**: If the convergence criteria are not met, repeat steps 2 and 3 with the newly calculated centroids.

The K-Means algorithm aims to minimize the within-cluster sum of squares (WCSS), which is the sum of the squared distances between each data point and its assigned cluster centroid. While K-Means is efficient and relatively easy to implement, it has certain limitations, including the requirement to specify K in advance and its sensitivity to the initial choice of centroids and outliers. Techniques like the elbow method or silhouette analysis can be used to help determine a suitable value for K. Furthermore, variations of K-Means, such as K-Means++, aim to improve the initialization process and the quality of the resulting clusters.

Pen & Paper

Apply K-Means Clustering algorithm to find clusters among given data points where $K = 2$.
Data points: $(2,3),(8,2),(9,3),(3,1),(2,5),(10,3)$

1. Let X be the dataset containing m observations and n features.
2. Randomly select k cluster centers.

For $K = 2$:

$$C^{(1)} = (2,3), \quad C^{(2)} = (8,2)$$

Iteration 1

3. Calculate the distance between each data point and each cluster center.

Distance formula:

$$\text{Distance}(x_i, C^{(j)}) = \sqrt{(x_{i1} - c_{j1})^2 + (x_{i2} - c_{j2})^2}$$

Assignments:

$$\text{Cluster 1: } (2,3),(3,1),(2,5)$$
$$\text{Cluster 2: } (8,2),(9,3),(10,3)$$

6. Recalculate the new cluster center using the mean of all the data points, called the centroid.

New centroids:

$$C^{(1)} = \left(\frac{2+3+2}{3}, \frac{3+1+5}{3}\right) = \left(\frac{7}{3}, 3\right) \approx (2.33, 3)$$
$$C^{(2)} = \left(\frac{8+9+10}{3}, \frac{2+3+3}{3}\right) = \left(9, \frac{8}{3}\right) \approx (9, 2.67)$$

Iteration 2

3. Calculate the distance between each data point and each cluster center.

Assignments:

$$\text{Cluster 1: } (2,3),(3,1),(2,5)$$
$$\text{Cluster 2: } (8,2),(9,3),(10,3)$$

6. Recalculate the new cluster center using the mean of all the data points, called the centroid.

New centroids:

$$C^{(1)} = \left(\frac{2+3+2}{3}, \frac{3+1+5}{3}\right) = \left(\frac{7}{3}, 3\right) \approx (2.33, 3)$$
$$C^{(2)} = \left(\frac{8+9+10}{3}, \frac{2+3+3}{3}\right) = \left(9, \frac{8}{3}\right) \approx (9, 2.67)$$

Since there is no change in the assignments or centroids between Iteration 1 and Iteration 2, the algorithm has converged.
The final clusters are:

$$\text{Cluster 1: } (2,3),(3,1),(2,5)$$
$$\text{Cluster 2: } (8,2),(9,3),(10,3)$$

The final centroids are:

$$C^{(1)} \approx (2.33, 3)$$
$$C^{(2)} \approx (9, 2.67)$$

Impact of Random Initialization A notable characteristic of the standard K-Means algorithm is its sensitivity to the initial placement of the cluster centroids. The random selection of initial cluster centers can lead to different clustering outcomes across multiple runs on the same dataset. This is because the algorithm may converge to different local optima of the within-cluster sum of squares objective function, resulting in suboptimal or varying cluster configurations. The quality and characteristics of the final clusters are thus significantly influenced by the initial starting points.

K-Means++: An Improved Initialization Technique To address the issue of sensitivity to random initialization and improve the stability and quality of the K-Means algorithm, the K-Means++ algorithm provides a more intelligent strategy for selecting the initial cluster centers. The core idea is to choose initial centroids that are spread out, rather than being clustered together, which increases the likelihood of finding a better clustering solution.

Steps of the K-Means++ Initialization Algorithm The K-Means++ initialization procedure unfolds as follows:

1. **First Centroid Selection**: Randomly select the first cluster center $\mathbf{c}_1$ uniformly from the set of data points.

2. **Distance Calculation**: For each data point $\mathbf{x}_i$ in the dataset, calculate the squared Euclidean distance $d(\mathbf{x}_i, \mathbf{C}_t)$ to the nearest cluster center already chosen in the set $\mathbf{C}_t = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_t\}$, where t is the number of centroids already selected. This distance is given by $d(\mathbf{x}_i, \mathbf{C}_t) = \min_{j \in \{1,\ldots,t\}} \|\mathbf{x}_i - \mathbf{c}_j\|^2$.

3. **Next Centroid Selection**: Select the next cluster center $\mathbf{c}_{t+1}$ from the remaining data points with a probability proportional to $d(\mathbf{x}_i, \mathbf{C}_t)$. In other words, a data point is more likely to be chosen as the next centroid if it is far away from the currently selected centroids. This can be implemented by calculating the probability $p_i = \frac{d(\mathbf{x}_i,\mathbf{C}_t)}{\sum_{j=1}^{m} d(\mathbf{x}_j,\mathbf{C}_t)}$ for each data point $\mathbf{x}_i$, and then sampling the next centroid based on this probability distribution.

4. **Iteration**: Repeat steps 2 and 3 until K cluster centers have been selected. The resulting set of K initial centroids is then used to proceed with the standard K-Means iterative optimization.

By strategically selecting initial centroids that are well-separated, K-Means++ generally leads to faster convergence and a higher likelihood of finding better quality clusters compared to random initialization. While the subsequent iterative steps of K-Means remain the same, the improved starting configuration provided by K-Means++ significantly enhances the overall performance and robustness of the clustering process.

3.2.2 Hierarchical Clustering

Hierarchical clustering is a family of clustering algorithms that build a hierarchy of clusters, represented by a tree-like structure known as a dendrogram. Unlike partitioning algorithms like K-Means, hierarchical clustering does not require specifying the number of clusters beforehand. Instead, the hierarchical structure allows for the exploration of data at different levels of granularity, and the desired number of clusters can be determined by ”cutting” the dendrogram at an appropriate level.

Techniques of Hierarchical Clustering

There are two main approaches to hierarchical clustering: agglomerative (bottom-up) and divisive (top-down).

- **Agglomerative Clustering (Bottom-Up Approach):** Agglomerative clustering starts by considering each data point as a single cluster. In each step, it iteratively merges the two most similar clusters until all data points belong to a single cluster, or a predefined stopping criterion is met (e.g., reaching a certain number of clusters). The sequence of merges forms the hierarchical structure.

Algorithm 9 Agglomerative Clustering Algorithm

```
1: Input: Dataset $X = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$ where each $\mathbf{x}_i \in \mathbb{R}^n$
2:
3: Initialize each data point as a singleton cluster: $\mathcal{C}_i = \{\mathbf{x}_i\}$ for $i = 1, 2, \dots, m$. Let the set of
   clusters be $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_m\}$.
4:
5: Compute the pairwise distance matrix $\mathbf{D}$ between all pairs of clusters in $\mathcal{C}$. The distance
   between two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ can be defined using different linkage methods (e.g., single
   linkage, complete linkage, average linkage, Ward's method).
6:
7: repeat
8:     Find the pair of clusters $(\mathcal{C}_a, \mathcal{C}_b)$ in $\mathcal{C}$ with the minimum distance according to $\mathbf{D}$:
9:         $(\mathcal{C}_a, \mathcal{C}_b) = \arg\min_{\mathcal{C}_i, \mathcal{C}_j \in \mathcal{C}, i \neq j} \mathbf{D}(\mathcal{C}_i, \mathcal{C}_j)$
10:
11:    Merge the two closest clusters $\mathcal{C}_a$ and $\mathcal{C}_b$ to form a new cluster $\mathcal{C}_{ab} = \mathcal{C}_a \cup \mathcal{C}_b$.
12:
13:    Update the set of clusters by removing $\mathcal{C}_a$ and $\mathcal{C}_b$ and adding $\mathcal{C}_{ab}$:
14:        $\mathcal{C} = (\mathcal{C} \setminus \{\mathcal{C}_a, \mathcal{C}_b\}) \cup \{\mathcal{C}_{ab}\}$
15:
16:    Update the distance matrix $\mathbf{D}$ by calculating the distances between the new cluster $\mathcal{C}_{ab}$
   and the remaining clusters in $\mathcal{C}$. The method for calculating these new distances depends
   on the chosen linkage criterion.
17: until only one cluster remains in $\mathcal{C}$ or a predefined number of clusters is reached
18:
19: Output: The hierarchy of clusters, typically represented as a dendrogram, which illustrates
   the merging process. The final cluster assignments can be obtained by choosing a level in
   the dendrogram.
```

Detailed Steps of the Agglomerative Clustering Algorithm

1. **Initialization:** Start with each data point as an individual cluster.

2. **Distance Computation:** Calculate the distance between all pairs of clusters. The choice of distance metric (e.g., Euclidean, Manhattan) and linkage method significantly affects the results. Common linkage methods include:
   - **Single Linkage (Minimum Linkage):** The distance between two clusters is the minimum distance between any two points in the two clusters. It can lead to a "chaining" effect.
   - **Complete Linkage (Maximum Linkage):** The distance between two clusters is the maximum distance between any two points in the two clusters. It tends to produce more compact clusters.
   - **Average Linkage:** The distance between two clusters is the average of the distances between all pairs of points, one from each cluster. It's a compromise between single and complete linkage.
   - **Ward's Method:** It minimizes the increase in within-cluster variance after merging two clusters. It often produces clusters of similar size.
3. **Iteration:**
   (a) Find the two clusters with the smallest distance between them.
   (b) Merge these two clusters into a single, larger cluster.
   (c) Update the distance matrix based on the chosen linkage method to reflect the distances between the new cluster and the remaining clusters.
4. **Termination:** Repeat the iteration step until only a single cluster remains, or a desired number of clusters is obtained. The history of merges is recorded and can be visualized as a dendrogram.

Pen & Paper

Apply agglomerative clustering to the following dataset.

	A	B	C	D	E	F
A	0	2.236	4.472	5	7.071	7.211
B	2.236	0	2.236	2.828	5	5.385
C	4.472	2.236	0	1	3.162	4
D	5	2.828	1	0	2.236	3
E	7.071	5	3.162	2.236	0	1.414
F	7.211	5.385	4	3	1.414	0

The upper triangular section is the mirror reflection of the lower triangular matrix. You can omit the upper triangular portion.

Merging Clusters Now consider every point as a single cluster. We merge two clusters having minimal distance.

	A	B	C	D	E	F
A	0	2.236	4.472	5	7.071	7.211
B	2.236	0	2.236	2.828	5	5.385
C	4.472	2.236	0	1	3.162	4
D	5	2.828	1	0	2.236	3
E	7.071	5	3.162	2.236	0	1.414
F	7.211	5.385	4	3	1.414	0

C and D are having the smallest distance between them. So we merge the clusters C and D into a single cluster.

	A	B	CD	E	F
A	0	2.236	4.472	7.071	7.211
B	2.236	0	2.236	5	5.385
CD	4.472	2.236	0	3.162	4
E	7.071	5	3.162	0	1.414
F	7.211	5.385	4	1.414	0

Repeating this process results in the hierarchy tree.

- **Divisive Clustering:** Uses a top-down approach, where the whole dataset is first considered as a single cluster and then split up into two or multiple clusters such that objects in one subgroup are dissimilar to the objects in another. This process continues until we find a single member in each cluster.

Algorithm 10 Divisive Hierarchical Clustering Algorithm

```
1: Input: Dataset $X$ containing $m$ observations and $n$ features
2: Initialize one cluster $\mathcal{C} = X$, containing all $m$ data points
3: Compute a criterion to split the current cluster $\mathcal{C}$ into two subclusters $\mathcal{C}_1$ and $\mathcal{C}_2$:
4:     $(\mathcal{C}_1, \mathcal{C}_2) = \text{split\_criterion}(\mathcal{C})$
5: Recursively apply steps 2-3 to each subcluster $\mathcal{C}_1$ and $\mathcal{C}_2$ until stopping criteria are met:
6:     - Minimum cluster size
7:     - Maximum number of clusters
8:     - Other criteria based on domain knowledge
9: Output: Hierarchical clustering dendrogram or cluster assignments
```

Detailed Steps of the Divisive Clustering Algorithm

1. **Initialization:** Begin with a single cluster containing all data points.
2. **Iteration:**
   (a) Select a cluster to split. The choice of which cluster to split can be based on various criteria, such as the size of the cluster, its variance, or other measures of heterogeneity.
   (b) Apply a partitioning clustering algorithm (like K-Means, K-Medoids, or even a simpler binary split based on a principal component) to divide the selected cluster into two or more subclusters. The choice of the splitting algorithm can influence the nature of the resulting hierarchy.
   (c) Repeat this process for the newly formed clusters until a stopping criterion is met. Common stopping criteria include:
      – Each cluster contains only a single data point.
      – A predefined number of clusters has been reached.
      – Clusters satisfy a certain homogeneity criterion (e.g., variance within clusters is below a threshold).
3. **Output:** The result is a hierarchical structure that can be visualized as a dendrogram, illustrating how the initial single cluster was successively divided into smaller clusters.

Both agglomerative and divisive hierarchical clustering provide a rich representation of the data's structure. Agglomerative methods are more commonly used due to their relative simplicity in implementation, while divisive methods can be computationally more intensive but may sometimes produce more balanced hierarchies. The choice between the two often depends on the specific dataset characteristics and the goals of the analysis.

3.2.3 Gaussian Mixture Model (GMM)

When the underlying cluster structures in a dataset deviate from the assumptions of spherical shapes and equal sizes inherent in K-Means, the Gaussian Mixture Model (GMM) offers a more flexible and probabilistic approach to clustering. GMM assumes that the data points are generated from a mixture of a finite number of Gaussian distributions, each representing a cluster. This allows for the modeling of clusters with varying shapes (ellipsoidal) and sizes.
A Gaussian Mixture Model (GMM) is a probabilistic model that posits that all data points originate from a blend of K Gaussian distributions, where the parameters of these distributions (mean and covariance) are unknown and need to be estimated. GMM is a powerful tool for both clustering and density estimation, particularly in multi-dimensional datasets where clusters may exhibit complex structures. Unlike K-Means, which performs hard assignments of data points to clusters, GMM provides a soft assignment by estimating the probability of each data point belonging to each of the Gaussian components (clusters).

Key Differences Between KMeans and GMM

- **Cluster Shape and Size**: K-Means implicitly assumes that clusters are spherical and have roughly equal variance. In contrast, GMM can model clusters with ellipsoidal shapes of different sizes and orientations due to the flexibility of the covariance matrices associated with each Gaussian component.

- **Clustering Assignment**: K-Means performs hard clustering, assigning each data point definitively to one cluster. GMM, on the other hand, performs soft clustering, providing a probability for each data point belonging to each of the K clusters. This probabilistic assignment can be more informative, especially when cluster boundaries are ambiguous or overlapping.

Underlying Assumptions of GMM

The effective application of GMM relies on the following key assumptions:

- **Mixture of Gaussians**: The dataset is assumed to be generated from a mixture of a finite number (K) of multivariate Gaussian distributions.

- **Gaussian Parameters**: Each Gaussian component c in the mixture is characterized by its own mean vector μ_c (determining the center of the cluster) and a covariance matrix Σ_c (determining the shape and orientation of the cluster).

- **Mixing Coefficients**: The probability of a data point being generated by a particular Gaussian component c is given by the mixing coefficient π_c. These coefficients are non-negative and sum to 1, i.e., $\sum_{c=1}^{K} \pi_c = 1$, representing the prior probability of each component.

Review of Gaussian Distribution

Before delving into the mechanics of GMM, it's essential to recall the properties of the Gaussian (Normal) distribution:

- **Probability Density Function (PDF) for Univariate Case**: The PDF of a one-dimensional Gaussian distribution is given by:

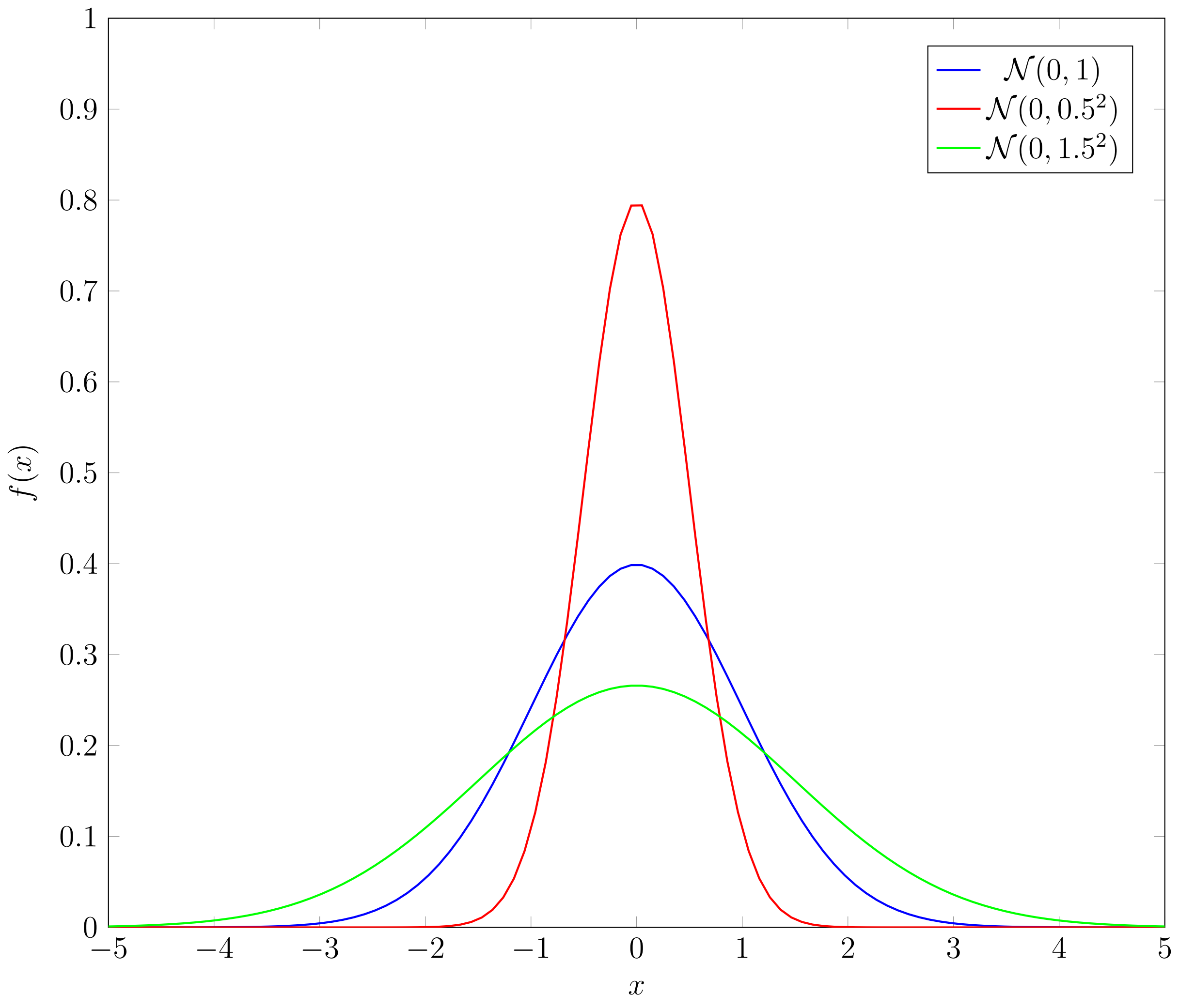
$$f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

  where μ is the mean and σ is the standard deviation.

- **Parameters**: The Gaussian distribution is fully characterized by its mean (μ), which indicates the central tendency, and its standard deviation (σ), which measures the spread or dispersion of the data.

- **Shape**: The Gaussian distribution exhibits a symmetric, bell-shaped curve centered around the mean.

- **Multivariate Generalization**: In multiple dimensions, a Gaussian distribution $\mathcal{N}(\mathbf{x}|\mu, \mathbf{\Sigma})$ is characterized by a mean vector μ (of the same dimensionality as the data) and a covariance matrix $\mathbf{\Sigma}$ (a symmetric positive-definite matrix that describes the shape, orientation, and spread of the multi-dimensional distribution).



Mathematical Formulation of GMM

A Gaussian Mixture Model represents the probability density function of a data point x as a weighted sum of the probability density functions of K Gaussian components:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$$

where:

- K is the predefined number of Gaussian components in the mixture.
- π_k are the mixing weights for each component k, satisfying $0 \leq \pi_k \leq 1$ and $\sum_{k=1}^{K} \pi_k = 1$. These weights represent the prior probability of a data point belonging to the k-th component.
- $\mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$ is the probability density function of the k-th multivariate Gaussian distribution with mean vector μ_k and covariance matrix $\mathbf{\Sigma}_k$.

Parameter Estimation using Expectation-Maximization (EM) Algorithm

The parameters of a GMM (the mixing weights π_k, the means μ_k, and the covariance matrices $\mathbf{\Sigma}_k$ for each of the K components) are typically estimated using the Expectation-Maximization (EM) algorithm. The EM algorithm is an iterative procedure designed to find maximum likelihood estimates of parameters in probabilistic models that involve latent variables (in this case, the component membership of each data point is unknown).

Detailed Steps of the EM Algorithm for GMM

1. **Initialization**: Initialize the parameters of the GMM. This involves choosing initial values for the mixing weights π_k, the mean vectors μ_k, and the covariance matrices $\mathbf{\Sigma}_k$ for each of the K Gaussian components. A common approach is to randomly initialize the means, set equal mixing weights ($\pi_k = 1/K$), and initialize the covariance matrices as scaled identity matrices or the sample covariance of the entire dataset.

2. **Expectation (E) Step**: For each data point $\mathbf{x}_n$ in the dataset (where n ranges from 1 to N, the total number of data points), compute the responsibility γ_{nk} of the k-th Gaussian component for that data point. The responsibility represents the posterior probability that data point $\mathbf{x}_n$ was generated by the k-th Gaussian component, given the current estimates of the parameters:

$$\gamma_{nk} = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \mathbf{\Sigma}_j)}$$

3. **Maximization (M) Step**: Update the parameters of each Gaussian component based on the responsibilities calculated in the E step. The updated parameters are:

   - **Component Size (N_k)**: The effective number of data points assigned to the k-th component:

$$N_k = \sum_{n=1}^{N} \gamma_{nk}$$

- **Mixing Weights (π_k^{new})**: The updated mixing weight for the k-th component:

$$\pi_k^{new} = \frac{N_k}{N}$$

- **Mean Vectors (μ_k^{new})**: The updated mean vector for the k-th component:

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} \mathbf{x}_n$$

- **Covariance Matrices ($\mathbf{\Sigma}_k^{new}$)**: The updated covariance matrix for the k-th component:

$$\mathbf{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} (\mathbf{x}_n - \mu_k^{new})(\mathbf{x}_n - \mu_k^{new})^T$$

4. **Convergence Check**: Evaluate whether the algorithm has converged. Convergence can be assessed by checking if the change in the log-likelihood of the data under the current model parameters is below a certain threshold, or if the parameters themselves change very little between iterations.

5. **Iteration**: If the convergence criteria are not met, return to the E step with the updated parameter values and repeat the process until convergence is achieved.

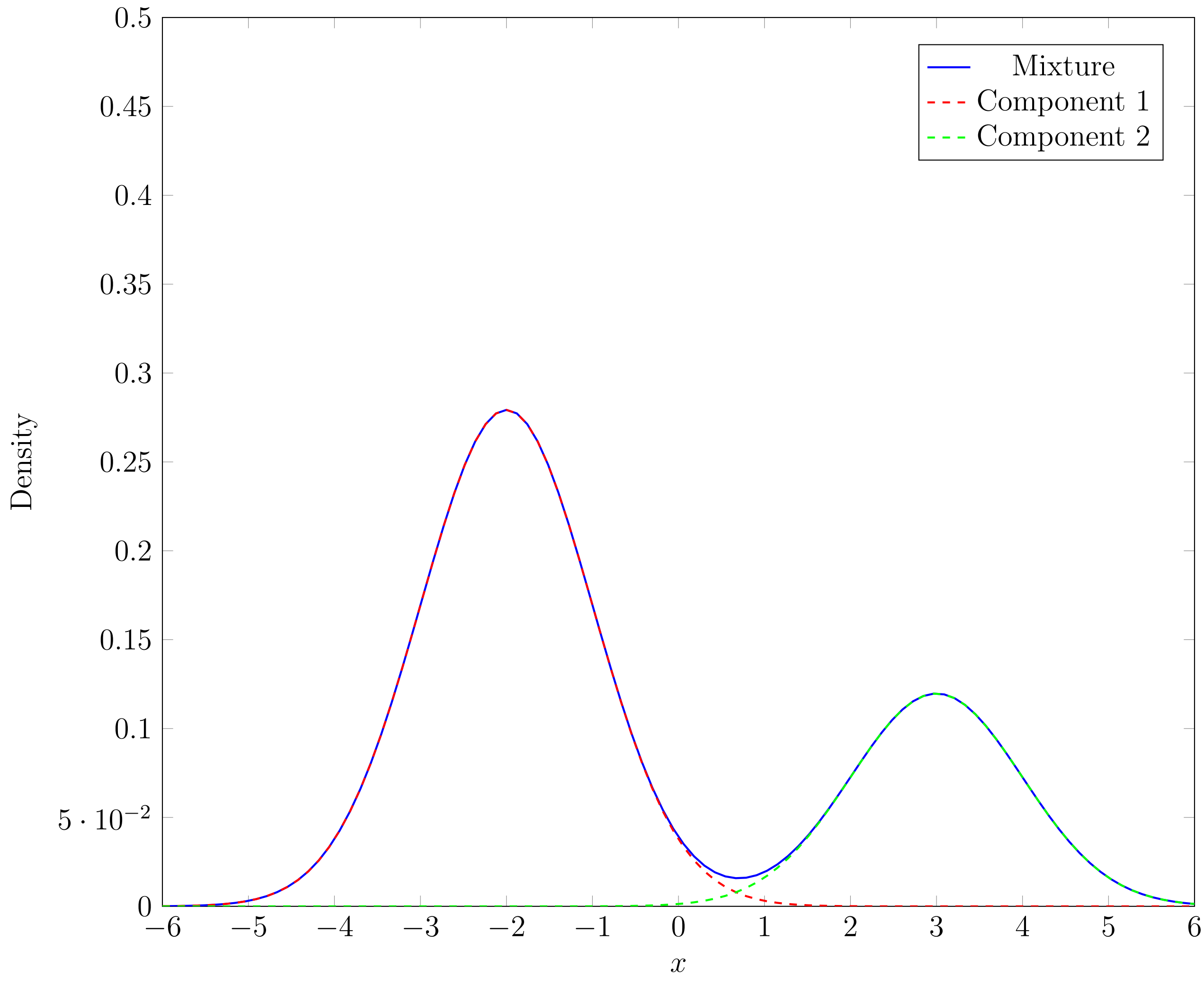
Algorithm 11 EM Algorithm for GMM

```
procedure EMGMM($X$, $K$)
    Initialize $\pi_k^{(0)}$, $\mu_k^{(0)}$, $\mathbf{\Sigma}_k^{(0)}$ for $k = 1, \ldots, K$
    $t \leftarrow 0$
    repeat
        $t \leftarrow t + 1$                                        ▷ E Step
        for $n = 1$ to $N$ do
            for $k = 1$ to $K$ do
                Compute responsibility: $\gamma_{nk}^{(t)} = \frac{\pi_k^{(t-1)} \mathcal{N}(\mathbf{x}_n | \mu_k^{(t-1)}, \mathbf{\Sigma}_k^{(t-1)})}{\sum_{j=1}^{K} \pi_j^{(t-1)} \mathcal{N}(\mathbf{x}_n | \mu_j^{(t-1)}, \mathbf{\Sigma}_j^{(t-1)})}$
            end for
        end for                                                     ▷ M Step
        for $k = 1$ to $K$ do
            Compute component size: $N_k^{(t)} = \sum_{n=1}^{N} \gamma_{nk}^{(t)}$
            Update mixing weight: $\pi_k^{(t)} = \frac{N_k^{(t)}}{N}$
            Update mean vector: $\mu_k^{(t)} = \frac{1}{N_k^{(t)}} \sum_{n=1}^{N} \gamma_{nk}^{(t)} \mathbf{x}_n$
            Update covariance matrix: $\mathbf{\Sigma}_k^{(t)} = \frac{1}{N_k^{(t)}} \sum_{n=1}^{N} \gamma_{nk}^{(t)} (\mathbf{x}_n - \mu_k^{(t)})(\mathbf{x}_n - \mu_k^{(t)})^T$
        end for
        Check for convergence based on change in parameters or log-likelihood
    until converged
    return $\pi_k^{(t)}$, $\mu_k^{(t)}$, $\mathbf{\Sigma}_k^{(t)}$ for $k = 1, \ldots, K$
end procedure
```

EM Algorithm Pseudocode

Illustrative Example: 1D Gaussian Mixture

Consider a one-dimensional dataset generated from a mixture of two Gaussian distributions with different means and mixing proportions:



Applying the EM algorithm to a dataset sampled from this mixture would iteratively refine the estimates of the parameters. Ideally, upon convergence, the algorithm would yield parameter estimates close to the true underlying parameters:

- Mixing weights: $\pi_1 \approx 0.7$, $\pi_2 \approx 0.3$
- Means: $\mu_1 \approx -2$, $\mu_2 \approx 3$
- Standard deviations (since it's 1D, covariance is just variance σ^2): $\sigma_1 \approx 1$, $\sigma_2 \approx 1$ (assuming the variances were 1 in the generating Gaussians).

Diverse Applications of GMMs

The flexibility and probabilistic nature of GMMs make them applicable across a wide range of domains:

- **Soft Clustering**: GMMs are particularly well-suited for scenarios where data points may have partial membership in multiple clusters, providing a more nuanced understanding of cluster overlap.

- **Density Estimation**: GMMs can effectively model complex, multi-modal probability distributions, offering a powerful tool for estimating the underlying data generation process.
- **Anomaly Detection**: By fitting a GMM to the normal data distribution, data points with low probability under the model can be identified as potential anomalies or outliers.
- **Speech Recognition**: GMMs have historically played a significant role in speech recognition systems for modeling the acoustic features of phonemes.
- **Image Segmentation**: In computer vision, GMMs can be used to model the distribution of pixel colors or textures, enabling image segmentation into different regions.
- **Bioinformatics**: GMMs can be used for tasks such as clustering gene expression data or identifying subpopulations within biological datasets.
- **Financial Modeling**: In finance, GMMs can be used to model asset returns, which often exhibit non-normal behavior and multiple regimes.

Code snippet

3.3 Dimensionality Reduction

In machine learning, the dimensionality of a dataset refers to the number of features or variables. While intuitively one might think that more features always lead to better model performance, this is not necessarily the case. High-dimensional data can introduce several challenges, collectively known as the "Curse of Dimensionality." These challenges include increased model complexity, greater computational cost for training, and the risk of overfitting, where the model learns noise in the high-dimensional training data rather than the underlying patterns, leading to poor generalization on unseen data. Furthermore, adding more features does not always guarantee improved classification accuracy; irrelevant or redundant features can even degrade performance.

Dimensionality reduction techniques aim to mitigate these issues by reducing the number of features in a dataset while preserving the essential information and underlying structure. The goal is to obtain a lower-dimensional representation that simplifies the model, reduces computational demands, and potentially improves generalization performance without significantly distorting the patterns inherent in the data.

However, it's crucial to recognize that any reduction in features inherently involves some loss of information. The challenge lies in minimizing this loss and ensuring that the most important aspects of the data, relevant to the learning task, are retained in the lower-dimensional space. Therefore, selecting an appropriate dimensionality reduction technique and determining the optimal number of reduced dimensions often requires careful consideration and experimentation. Several common methods are employed for dimensionality reduction, each with its own underlying principles and suitability for different types of data:

- **Principal Component Analysis (PCA)**: A linear dimensionality reduction technique that transforms the data into a new set of orthogonal variables (principal components) that capture the maximum variance in the data.
- **Feature Selection**: A process of identifying and selecting a subset of the original features that are most relevant and informative for the task, discarding the less important or redundant ones.

- **Low Variance Filter**: A simple feature selection technique that removes features whose variance falls below a certain threshold, as features with very low variance provide little discriminatory information.

- **Linear Discriminant Analysis (LDA)**: A supervised dimensionality reduction technique primarily used for classification tasks. It aims to find a linear combination of features that best separates different classes in the data.

3.3.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a widely used linear dimensionality reduction technique. Its primary goal is to transform a high-dimensional dataset into a lower-dimensional one by identifying the directions (principal components) in the data that account for the most variance. By projecting the original data onto these principal components, PCA reduces the number of features while retaining as much of the original data's variability as possible.

Step-by-Step Explanation of PCA

Let's break down the PCA algorithm into a sequence of steps:

1. **Data Acquisition**: Begin with a dataset containing n observations and d features.

2. **Data Standardization**: It is crucial to standardize each feature (column) in the dataset to have zero mean and unit variance. This ensures that features with larger scales do not disproportionately influence the principal components. The standardization is performed by subtracting the mean of each feature and dividing by its standard deviation.

3. **Covariance Matrix Computation**: Calculate the $d \times d$ covariance matrix Σ of the standardized data. The covariance between two features measures the degree to which they change together. For a dataset X with n samples and d features, the covariance matrix can be computed as:

$$\Sigma = \frac{1}{n-1} \sum_{i=1}^{n} (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^T$$

   where $\mathbf{x}_i$ is the i-th data point (a d-dimensional vector) and μ is the mean vector of the data. For standardized data, this simplifies to $\Sigma = \frac{1}{n-1} X^T X$.

The Significance of the Covariance Matrix: The covariance matrix provides insights into the relationships between the different features in the dataset. The diagonal elements represent the variance of each feature, while the off-diagonal elements represent the covariance between pairs of features. A high positive covariance indicates that two features tend to increase or decrease together, while a high negative covariance indicates that they tend to move in opposite directions. A covariance close to zero suggests little linear relationship between the features.

4. **Eigen Decomposition**: Compute the eigenvalues and corresponding eigenvectors of the covariance matrix Σ. For a $d \times d$ covariance matrix, there will be d eigenvalues $(\lambda_1, \lambda_2, \ldots, \lambda_d)$ and d corresponding eigenvectors $(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d)$, satisfying the equation:

$$\Sigma \mathbf{v} = \lambda \mathbf{v}$$

   where λ is an eigenvalue and $\mathbf{v}$ is the corresponding eigenvector.

Algorithm 12 Principal Component Analysis (PCA)

```
1: procedure PCA($X, k$)
2:     Input: $X \in \mathbb{R}^{n \times d}$: Dataset with $n$ samples and $d$ features
3:     $k$: Number of principal components to retain ($k \leq d$)
4:
5:     Output:
6:     $W \in \mathbb{R}^{d \times k}$: Projection matrix containing the top $k$ principal components (eigenvectors)
7:     $Y \in \mathbb{R}^{n \times k}$: Data projected onto the $k$ principal components
8:                                                          ▷ 1. Standardize the data
9:     for $j = 1$ to $d$ do
10:        $\mu_j \leftarrow \frac{1}{n} \sum_{i=1}^{n} X_{ij}$                ▷ Calculate the mean of the $j$-th feature
11:        $\sigma_j \leftarrow \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (X_{ij} - \mu_j)^2}$ ▷ Calculate the standard deviation of the $j$-th feature
12:        for $i = 1$ to $n$ do
13:            $X_{ij} \leftarrow \frac{X_{ij} - \mu_j}{\sigma_j}$          ▷ Standardize the $i$-th sample of the $j$-th feature
14:        end for
15:    end for
16:
17:                                              ▷ 2. Compute the covariance matrix
18:    $\Sigma \leftarrow \frac{1}{n-1} X^T X$ ▷ For standardized data, the covariance matrix is proportional to $X^T X$
19:
20:                     ▷ 3. Compute eigenvectors and eigenvalues of the covariance matrix
21:    $\{\mathbf{v}_1, \ldots, \mathbf{v}_d\}, \{\lambda_1, \ldots, \lambda_d\} \leftarrow$ EigenDecomposition($\Sigma$)
22:                ▷ where $\lambda_i$ are the eigenvalues and $\mathbf{v}_i$ are the corresponding eigenvectors
23:
24:                                  ▷ 4. Sort eigenvectors by decreasing eigenvalues
25:    Sort $(\lambda_i, \mathbf{v}_i)$ pairs in descending order of $\lambda_i$
26:                                          ▷ such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$
27:
28:                                          ▷ 5. Select the top $k$ eigenvectors
29:    $W \leftarrow [\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_k]$        ▷ Form a projection matrix with the top $k$ eigenvectors
30:
31:                                  ▷ 6. Project the data onto the new subspace
32:    $Y \leftarrow XW$                  ▷ The $n \times k$ matrix $Y$ contains the projected data
33:
34:    Output: $W, Y$
35: end procedure
```

5. **Principal Component Selection**: Sort the eigenvalues in descending order. The eigenvector associated with the largest eigenvalue corresponds to the first principal component, which captures the direction of maximum variance in the data. The eigenvector associated with the second largest eigenvalue is the second principal component, capturing the next highest variance in a direction orthogonal to the first, and so on.

6. **Feature Vector Formation**: Select the top k eigenvectors corresponding to the k largest eigenvalues to form a $d \times k$ projection matrix $W = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_k]$. The choice of k determines the reduced dimensionality of the data. A common approach to selecting k is to consider the percentage of variance explained by the first k principal components.

7. **Data Projection**: Project the original (standardized) data X onto the new k-dimensional subspace spanned by the top k principal components. This is done by multiplying the data matrix X (of size $n \times d$) with the projection matrix W (of size $d \times k$):

$$Y = XW$$

The resulting matrix Y (of size $n \times k$) contains the projected data in the lower-dimensional space.

Key Properties of Principal Components

- **Orthogonality**: The principal components (eigenvectors of the covariance matrix) are orthogonal to each other. This means they are uncorrelated, and each captures a unique aspect of the data's variance.

- **Variance Ordering**: The principal components are ordered by the amount of variance they explain. The first principal component accounts for the largest proportion of the total variance in the original data, the second principal component accounts for the next largest proportion, and so on. This property allows us to reduce dimensionality by discarding components that explain less variance, thereby retaining the most important information.

Variations of PCA

While standard PCA is a powerful linear dimensionality reduction technique, several variations have been developed to address its limitations or to adapt it to specific types of data and requirements.

Kernel PCA Kernel PCA extends the linear PCA approach to handle non-linear relationships in the data. It achieves this by using the "kernel trick," which implicitly maps the original data into a higher-dimensional feature space where linear PCA can be applied to find non-linear principal components in the original space.
Algorithm Overview:

1. **Kernel Selection**: Choose a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$ that defines the similarity between pairs of data points in the implicitly defined high-dimensional space (e.g., Gaussian radial basis function (RBF) kernel: $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$, where $\gamma > 0$ is a kernel parameter).

2. **Kernel Matrix Computation**: Construct the $n \times n$ kernel matrix K, where $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$.

3. **Centering the Kernel Matrix**: Center the kernel matrix to ensure that the data has zero mean in the feature space. The centered kernel matrix $\tilde{K}$ is computed as:

$$\tilde{K} = K - \mathbf{1}_n K - K\mathbf{1}_n + \mathbf{1}_n K \mathbf{1}_n$$

   where $\mathbf{1}_n$ is an $n \times n$ matrix with all elements equal to $1/n$.

4. **Eigendecomposition of Centered Kernel Matrix**: Perform eigendecomposition on the centered kernel matrix $\tilde{K}$ to find its eigenvalues and eigenvectors.

5. **Projection**: Project the data onto the principal components (eigenvectors corresponding to the largest eigenvalues) in the feature space. The projection of a data point $\mathbf{x}_i$ onto the k-th principal component α_k is given by $y_i^{(k)} = \sum_{j=1}^{n} \alpha_{jk} K(\mathbf{x}_i, \mathbf{x}_j)$, where α_{jk} is the j-th component of the k-th eigenvector.

Advantages:

- **Non-linear Pattern Capture**: Kernel PCA can effectively uncover and reduce the dimensionality of data with complex, non-linear structures that linear PCA might miss.
- **Flexibility through Kernel Choice**: The choice of kernel function allows adaptation to different types of non-linear relationships in the data.

Limitations:

- **Kernel Selection and Parameter Tuning**: The performance of Kernel PCA is highly dependent on the choice of the kernel function and its parameters, which can be challenging to select and tune.
- **Computational Cost for Large Datasets**: Computing and performing eigendecomposition on the $n \times n$ kernel matrix can be computationally expensive for large datasets, with a complexity of $O(n^3)$.
- **Interpretation of Principal Components**: The principal components in Kernel PCA are in the implicit feature space, making their interpretation in the original data space less straightforward.

Incremental PCA (IPCA) Incremental PCA is an adaptation of PCA designed to handle very large datasets that cannot fit into memory at once. Instead of processing the entire dataset at once, IPCA processes the data in smaller batches (increments) and updates the principal components iteratively.
Implementation Overview:

1. **Initialization**: Initialize the PCA model with a small initial batch of data.
2. **Batch Processing**: For each subsequent batch of data:
   - Project the new data onto the current set of principal components.
   - Update the principal components based on the new batch and the previously processed data. This typically involves updating the mean and covariance estimates incrementally.
3. **Iteration**: Repeat the batch processing step until all data has been processed.

Advantages:

- **Memory Efficiency for Large Datasets**: IPCA can process datasets that are too large to fit into memory, as it only requires loading and processing small batches at a time.
- **Handling Streaming Data**: IPCA can be used to perform dimensionality reduction on streaming data, where new data points arrive sequentially.

Limitations:

- **Potential for Suboptimal Results**: The incremental nature of the algorithm might lead to slightly different principal components compared to standard batch PCA, especially if the data distribution changes significantly across batches or if the number of samples in each batch is small.
- **Sensitivity to Data Order**: The order in which the data batches are presented can potentially affect the final result, although this effect is often minor with sufficiently large and representative batches.

Sparse PCA Sparse PCA aims to find principal components that are sparse, meaning they have few non-zero components. This is in contrast to standard PCA, where the principal components are typically dense linear combinations of the original features. Sparse PCA can lead to more interpretable principal components, as each component is related to a smaller subset of the original features, effectively performing implicit feature selection.
Optimization Problem (Simplified): Sparse PCA can be formulated as an optimization problem that seeks to minimize the reconstruction error while simultaneously imposing a sparsity constraint on the principal components (loading vectors). A common approach involves adding an L1 norm penalty to the loadings:

$$\min_{U\in\mathbb{R}^{n\times k}, V\in\mathbb{R}^{d\times k}} \|X - UV^T\|_F^2 + \alpha\|V\|_1$$

where U contains the principal component scores, V contains the sparse principal component loadings, $\|\cdot\|_F$ denotes the Frobenius norm, $\|\cdot\|_1$ is the L1 norm that encourages sparsity in the columns of V, and α is a regularization parameter that controls the degree of sparsity.
Advantages:

- **Improved Interpretability**: The sparse principal components are easier to interpret because each component is primarily influenced by a small number of original features.
- **Implicit Feature Selection**: By identifying which original features have non-zero loadings in the principal components, Sparse PCA implicitly performs feature selection.

Limitations:

- **Non-convex Optimization**: The optimization problem in Sparse PCA is generally non-convex, which can make finding the global optimum challenging.
- **Parameter Tuning**: The regularization parameter α needs to be carefully tuned to achieve the desired level of sparsity and performance.
- **Computational Complexity**: Sparse PCA algorithms can be more computationally intensive than standard PCA.

Robust PCA Robust PCA is specifically designed to address the challenge of dimensionality reduction in the presence of outliers or noise within the dataset. The core principle of Robust PCA is to decompose the original data matrix X into two additive components: a low-rank matrix L and a sparse matrix S, such that $X = L + S$. The low-rank matrix L is intended to capture the underlying structure of the clean, low-dimensional data, while the sparse matrix S is designed to contain the outliers or corruptions, which are assumed to be sparse (i.e., affecting only a small number of entries in the data matrix).

The rationale behind this decomposition is that if the underlying data truly lies in a low-dimensional subspace (hence, L is low-rank), then deviations from this structure caused by outliers or noise will appear as sparse errors in the data matrix. By separating these sparse errors, Robust PCA aims to recover a clean, low-rank representation of the data that is not unduly influenced by the outliers.

Optimization Problem (Simplified): The problem of Robust PCA is typically formulated as an optimization problem that seeks to find the low-rank component L and the sparse component S that best explain the data X. A common approach involves minimizing a combination of the rank of L (often approximated by the nuclear norm, which is the sum of the singular values) and the sparsity of S (often promoted by the L1 norm, which is the sum of the absolute values of its entries):

$$\min_{L,S} \|L\|_* + \lambda \|S\|_1 \quad \text{subject to} \quad X = L + S$$

where $\|L\|_* = \sum_i \sigma_i(L)$ is the nuclear norm of L (sum of its singular values), $\|S\|_1 = \sum_{i,j} |S_{ij}|$ is the L1 norm of S, and λ is a positive regularization parameter that balances the importance of low-rankness and sparsity.

Advantages:

- **Effective Handling of Outliers**: Robust PCA is particularly advantageous when dealing with datasets contaminated by significant outliers or gross errors, as it attempts to isolate these corruptions in the sparse component S, allowing for a cleaner, low-rank representation L.

- **Recovery of Underlying Structure**: By separating the sparse noise, Robust PCA can better recover the true, low-dimensional structure of the data, which might be obscured by outliers in standard PCA.

Limitations:

- **Computational Complexity**: Solving the optimization problem involved in Robust PCA is generally more computationally intensive than standard PCA, especially for large datasets. Iterative algorithms are typically required to find the optimal L and S.

- **Parameter Tuning**: The regularization parameter λ plays a crucial role in the performance of Robust PCA. It needs to be appropriately tuned based on the expected level and nature of the outliers in the data. Choosing a suboptimal λ can lead to either under- or over-penalizing the sparse component.

- **Assumptions about Data and Outliers**: Robust PCA relies on the assumption that the underlying clean data is low-rank and that the outliers are sparse. If these assumptions are violated (e.g., if the outliers themselves exhibit a low-rank structure or if the clean data is inherently high-rank), the performance of Robust PCA may degrade.

Randomized PCA Randomized PCA is a computationally efficient approximation of standard PCA, particularly useful for very large datasets. It leverages random projections to quickly find an approximate low-rank subspace that captures most of the variance in the data.
Key Idea: Instead of directly computing the covariance matrix and its eigendecomposition, Randomized PCA uses random matrices to project the high-dimensional data into a lower-dimensional space, where standard PCA can then be applied more efficiently.
Algorithm Overview:

1. **Random Projection**: Generate a random matrix $\Omega \in \mathbb{R}^{d \times l}$, where l is a target lower dimension slightly larger than the desired number of principal components k (e.g., $l = k+p$, where p is a small oversampling parameter). The entries of Ω are typically drawn from a Gaussian or Rademacher distribution.

2. **Projection of Data**: Project the data matrix $X \in \mathbb{R}^{n \times d}$ onto the lower-dimensional space using the random matrix: $Y = X\Omega \in \mathbb{R}^{n \times l}$.

3. **QR Decomposition**: Compute the QR decomposition of Y: $Y = QR$, where $Q \in \mathbb{R}^{n \times l}$ has orthonormal columns that span the range of Y, and $R \in \mathbb{R}^{l \times l}$ is an upper triangular matrix.

4. **Projecting Original Data onto Subspace**: Project the original data X onto the subspace spanned by the columns of Q: $B = Q^T X \in \mathbb{R}^{l \times d}$.

5. **Singular Value Decomposition (SVD)**: Compute the SVD of the smaller matrix B: $B = \tilde{U}\Sigma V^T$, where $\tilde{U} \in \mathbb{R}^{l \times l}$, $\Sigma \in \mathbb{R}^{l \times d}$ (only the first l singular values are non-zero), and $V \in \mathbb{R}^{d \times d}$.

6. **Principal Components**: The approximate principal components of the original data X are given by $U = Q\tilde{U} \in \mathbb{R}^{n \times l}$. The first k columns of U correspond to the top k principal components. The projection of the original data onto these k principal components is given by the first k columns of U multiplied by the first k rows of ΣV^T. Alternatively, it can be obtained by taking the first k columns of U and computing $XU[:, : k]$.

Advantages:

- **Computational Efficiency for Large Datasets**: Randomized PCA can be significantly faster than standard PCA, especially for very high-dimensional data, as it avoids the expensive computation of the full covariance matrix and its eigendecomposition.

- **Memory Efficiency**: It can be implemented in a way that requires less memory than standard PCA.

Limitations:

- **Approximation**: Randomized PCA provides an approximation of the principal components, not the exact solution. The accuracy of the approximation depends on the oversampling parameter p.

- **Sensitivity to Random Seed**: The results can vary slightly depending on the random seed used to generate the random projection matrix.

Comparison of PCA Variations

Variation	**Best Used When**	**Key Advantage**
Kernel PCA	Data has non-linear patterns	Can capture complex relationsh
Incremental PCA	Datasets are very large or streaming	Memory-efficient for large data
Sparse PCA	Interpretability of components is important	Produces sparse, interpretable c
Robust PCA	Data contains significant outliers or noise	Robust to outliers; separates no
Randomized PCA	Datasets are extremely large and speed is critical	Computationally very efficient

3.3.2 Low Rank Approximations

Low-rank approximations (LRA) constitute a powerful class of techniques aimed at reducing the dimensionality of high-dimensional data by representing a given matrix A with another matrix B of a significantly lower rank, while ensuring that B closely approximates A according to a chosen norm.

Formally, given a matrix $A \in \mathbb{R}^{m\times n}$ with an inherent rank r, the objective of LRA is to find a matrix $B \in \mathbb{R}^{m\times n}$ with a rank k such that $k < r$, and the difference between A and B, measured by a suitable matrix norm (commonly the Frobenius norm $\|\cdot\|_F$), is minimized:

$$\min_{B:\text{rank}(B)\leq k} \|A - B\|_F$$

The pursuit of low-rank approximations is driven by several compelling motivations:

- **Data Compression**: High-dimensional datasets often contain redundancies. LRA can capture the essential information in a lower-dimensional space, leading to reduced storage requirements and more efficient data handling.

- **Noise Reduction**: By focusing on the dominant low-rank structure of the data, LRA can effectively filter out high-frequency noise or less significant variations, leading to a cleaner representation of the underlying signal.

- **Feature Extraction**: LRA can help in identifying the most salient underlying factors or components that contribute most to the variance in the data, effectively performing feature extraction.

- **Computational Efficiency**: Operations involving lower-rank matrices are generally less computationally expensive. Approximating a high-rank matrix with a low-rank one can significantly speed up subsequent computations in various algorithms.

The rank of a matrix is a fundamental property that denotes the number of linearly independent rows or columns it contains. A rank-k matrix B can be expressed as a product of two lower-dimensional matrices:

$$A \approx B = XY^T$$

where $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{n\times k}$. This factorization significantly reduces the number of parameters needed to represent the matrix from mn entries in A to $k(m+n)$ entries in X and Y. This reduction is particularly substantial when the chosen rank k is much smaller than both m and n ($\min(m, n)$).

3.3.3 Singular Value Decomposition (SVD)

The Singular Value Decomposition (SVD) is a cornerstone of linear algebra and provides a powerful framework for obtaining the optimal low-rank approximation of a matrix in terms of the Frobenius norm. It decomposes any real or complex matrix into a product of three other matrices.
For a matrix $A \in \mathbb{R}^{m\times n}$ with rank r, the SVD is given by:

$$A = U\Sigma V^T$$

where:

- $U \in \mathbb{R}^{m\times m}$ is an orthogonal matrix whose columns are the left singular vectors of A. These vectors form an orthonormal basis for the column space of A.
- $V \in \mathbb{R}^{n\times n}$ is an orthogonal matrix whose columns are the right singular vectors of A. These vectors form an orthonormal basis for the row space of A.
- $\Sigma \in \mathbb{R}^{m\times n}$ is a rectangular diagonal matrix with non-negative real numbers on the diagonal, known as the singular values of A. These singular values are typically arranged in descending order: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$, and $\sigma_i = 0$ for $i > r$.

Key Properties of SVD

1. **Orthonormal Bases**: The left singular vectors (columns of U) and the right singular vectors (columns of V) form orthonormal bases for the column space and row space of A, respectively. This means that within each set, the vectors are mutually orthogonal and have a unit norm.
2. **Singular Value Importance**: The singular values σ_i quantify the amount of variance captured along the direction of the corresponding pair of singular vectors (u_i and v_i). Larger singular values indicate more significant directions in the data.
3. **Rank Determination**: The number of non-zero singular values is equal to the rank of the matrix A.

Geometric Interpretation of SVD

Geometrically, the SVD can be viewed as a sequence of linear transformations that map vectors from the row space of A to its column space. For a vector $\mathbf{x}$ in the row space:

1. V^T applies a rotation or reflection to $\mathbf{x}$, aligning it with the principal directions of variance.
2. Σ scales the components of the transformed vector along these principal directions by the singular values. Larger singular values correspond to greater scaling.
3. U applies another rotation or reflection to the scaled vector, mapping it to the column space of A.

This interpretation highlights how SVD decomposes the action of the matrix A into fundamental operations of rotation, scaling, and another rotation, revealing the intrinsic structure and dominant directions within the data.

Optimal Low Rank Approximation via SVD (Eckart-Young Theorem)

A crucial result, known as the Eckart-Young theorem (specifically, the Eckart-Young-Mirsky theorem for the Frobenius norm), states that the best rank-k approximation A_k of the original matrix A (in terms of the Frobenius norm) can be constructed by taking the sum of the first k terms of the SVD:

$$A_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

where $\mathbf{u}_i$ is the i-th column of U (the i-th left singular vector), $\mathbf{v}_i$ is the i-th column of V (the i-th right singular vector), and σ_i is the i-th singular value. The rank of A_k is k, and it minimizes $\|A - A_k\|_F$ among all matrices of rank at most k. The squared error of this approximation is given by the sum of the squares of the discarded singular values: $\|A - A_k\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2$.

Algorithm for Computing SVD

Computing the full SVD can be computationally intensive, especially for large matrices. Several algorithms exist, with varying trade-offs between speed and accuracy. The power iteration method, outlined in the following, is a relatively simple iterative approach to approximating dominant singular values and vectors:

Algorithm 13 Power Iteration Method for SVD (Approximation)

```
1: procedure ApproximateSVD($A$, $k$, $iterations$)
2:     Initialize $V_k \in \mathbb{R}^{n \times k}$ with random orthonormal columns
3:     for $i = 1$ to $iterations$ do
4:         $U_k \leftarrow AV_k$
5:         $U_k \leftarrow$ orthonormalize($U_k$)                ▷ e.g., using QR decomposition
6:         $V_k \leftarrow A^T U_k$
7:         $V_k \leftarrow$ orthonormalize($V_k$)
8:     end for
9:     $\Sigma_k \leftarrow U_k^T A V_k$ ▷ $\Sigma_k$ will be a $k \times k$ (or $k \times n$ if $m > k$, $m \times k$ if $n > k$) diagonal matrix
   of top singular values
10:    return $U_k$, $\Sigma_k$, $V_k$
11: end procedure
```

More robust and accurate algorithms for SVD computation include the Golub-Kahan bidiagonalization algorithm and its variations, which are typically implemented in numerical libraries.

Diverse Applications of Low Rank Approximations and SVD

The ability of SVD to provide optimal low-rank approximations has led to its widespread use in various fields:

- **Principal Component Analysis (PCA)**: When PCA is applied to a centered data matrix X, the SVD of X is closely related to the eigendecomposition of the covariance matrix $X^T X$. The right singular vectors V (or their first k columns) correspond to the principal components, and the squared singular values are proportional to the eigenvalues, representing the variance explained by each component.

- **Image Compression**: An image can be represented as a matrix of pixel intensities. By computing the SVD of this matrix and keeping only the top k singular values and their corresponding singular vectors, a compressed version of the image can be reconstructed. The compression ratio increases with decreasing k, at the cost of some loss in image quality.

- **Recommendation Systems**: In collaborative filtering, the user-item interaction matrix (e.g., ratings) is often sparse. SVD can be used to factorize this matrix into lower-dimensional latent factor matrices representing user preferences and item characteristics. These latent factors can then be used to predict missing ratings and make recommendations.

- **Noise Reduction and Denoising**: By performing an SVD of noisy data and reconstructing it using only the singular vectors associated with the largest singular values, which are assumed to capture the underlying signal, the noise associated with smaller singular values can be effectively reduced.

- **Latent Semantic Analysis (LSA) in Natural Language Processing**: SVD can be applied to the term-document matrix to uncover underlying semantic relationships between words and documents, reducing the dimensionality of the document space and improving information retrieval.

Illustrative Example: Image Compression using SVD

Consider a grayscale image represented by an $m \times n$ matrix A, where each entry corresponds to the intensity of a pixel. Image compression using SVD involves the following steps:

1. **Compute SVD**: Calculate the Singular Value Decomposition of the image matrix $A = U\Sigma V^T$.

2. **Choose Rank k**: Select a rank k such that $k < r = \text{rank}(A)$. This k determines the number of singular values and vectors to keep, and hence the compression level.

3. **Reconstruct Compressed Image**: Compute the rank-k approximation $A_k = U_k \Sigma_k V_k^T$, where U_k consists of the first k columns of U, V_k consists of the first k columns of V, and Σ_k is the $k \times k$ diagonal matrix containing the top k singular values.

The compressed image A_k requires storing only the first k left singular vectors (mk elements), the top k singular values (k elements), and the first k right singular vectors (nk elements), totaling $k(m + n + 1)$ storage units, compared to mn for the original image. The compression ratio is approximately $\frac{k(m+n+1)}{mn}$. The quality of the compressed image depends on the value of k; a smaller k leads to higher compression but also greater information loss and potentially lower visual fidelity. The error introduced by the approximation is related to the magnitude of the discarded singular values.

3.3.4 Latent Semantic Analysis (LSA)

Latent Semantic Analysis (LSA) is a sophisticated technique within the realm of natural language processing (NLP) that employs Singular Value Decomposition (SVD) to uncover the underlying semantic relationships between words and documents within a large collection (corpus) of text. The core idea is to transcend the limitations of simple keyword matching by identifying patterns of word co-occurrence across documents, thereby inferring conceptual similarities that might not be evident from surface-level word overlap.
LSA is predominantly utilized for several key NLP tasks:

- **Concept Searching**: Enabling the retrieval of documents that are semantically related to a query, even if they do not share the exact keywords used in the query.
- **Automated Document Categorization**: Grouping documents based on their latent semantic content, facilitating automated organization and topic modeling.
- **Information Retrieval**: Enhancing the accuracy and relevance of search results by considering the contextual meaning of words and the semantic relationships between documents and queries.
- **Text Summarization**: Identifying the most important underlying concepts within a document to generate concise and meaningful summaries.

The fundamental principle behind LSA is the distributional hypothesis, which posits that words appearing in similar contexts tend to have similar meanings. By analyzing the statistical patterns of how words co-occur across a corpus of documents, LSA can deduce these latent semantic associations.
Consider a text corpus comprising:

- m unique words (terms) in the entire collection.
- n individual documents.

The first step in LSA is to construct a term-document matrix D of dimensions $m \times n$. Each element D_{ij} in this matrix quantifies the importance or weight of the i-th term in the j-th document. This weighting is often achieved using the Term Frequency-Inverse Document Frequency (TF-IDF) scheme, which reflects how important a word is to a document in a corpus. Terms that appear frequently in a specific document but infrequently across the corpus are given higher weights.
Once the term-document matrix D is constructed, SVD is applied:

$$D = U\Sigma V^T$$

The resulting matrices from the SVD provide a decomposition of the semantic structure of the corpus:

- U (dimension $m \times m$) is a matrix where each row represents a term, and each column corresponds to a latent semantic concept. The values in U indicate the strength of association between each term and each concept.
- Σ (dimension $m \times n$) is a diagonal matrix containing the singular values, ordered by magnitude. These singular values represent the strength or importance of each latent semantic concept in capturing the variance in the original term-document matrix.
- V^T (dimension $n \times n$) is a matrix where each row represents a latent semantic concept, and each column corresponds to a document. The values in V^T indicate the strength of association between each concept and each document.

The LSA Process in Detail

The application of LSA typically involves the following sequence of steps:

1. **Corpus Preparation**: The initial stage involves collecting the text documents that constitute the corpus. These documents are then preprocessed, which may include tokenization (splitting text into words), stop-word removal (eliminating common, non-informative words like ”the”, ”is”, ”and”), and stemming or lemmatization (reducing words to their root form).

2. **Term-Document Matrix Creation**: A term-document matrix D is constructed, where rows represent the unique terms in the corpus and columns represent the documents. The entries of the matrix are typically the TF-IDF weights of the terms in the documents.

3. **Singular Value Decomposition (SVD)**: The SVD algorithm is applied to the term-document matrix D to obtain the matrices U, Σ, and V^T.

4. **Dimensionality Reduction**: A crucial step in LSA is reducing the dimensionality of the semantic space. This is achieved by selecting only the top k singular values from Σ (and their corresponding first k columns of U and first k rows of V^T). This yields the reduced matrices U_k $(m \times k)$, Σ_k $(k \times k$ diagonal matrix), and V_k^T $(k \times n)$. The choice of k determines the number of latent semantic dimensions retained.

5. **Latent Space Transformation**: Both documents and queries can be projected into this lower-dimensional "semantic" space. A document j is represented by the j-th column of V_k (or V_k^T, depending on convention and scaling), and a query (represented as a vector of term weights) can be projected by $q_k = q^T U_k \Sigma_k^{-1}$.

6. **Similarity Computation**: In this reduced semantic space, the similarity between documents, or between a query and a document, can be computed using measures such as cosine similarity. Documents that are close to each other in this space are considered semantically similar.

The Role of Dimensionality Reduction in LSA

Dimensionality reduction is a key aspect of LSA. By truncating the SVD and keeping only the top k singular values and their associated singular vectors, we obtain a low-rank approximation of the original term-document matrix:

$$D_k = U_k \Sigma_k V_k^T$$

This reduced representation aims to capture the most significant underlying semantic dimensions in the corpus while filtering out noise, less important variations in word usage, and the idiosyncrasies of individual documents. The selection of the optimal number of dimensions k is a critical task.

- **Too Few Dimensions (k is too small)**: Retaining too few singular values may lead to a loss of important semantic information, resulting in a poor representation of the underlying concepts.

- **Too Many Dimensions (k is too large)**: Keeping too many singular values might retain noise and specific details of the corpus that do not generalize well, potentially hindering the discovery of broader semantic relationships.

The optimal value of k is often determined empirically, through experimentation and evaluation of the performance of LSA on specific tasks. For large text corpora, k is typically chosen to be significantly smaller than the number of terms or documents, often ranging from 100 to 300 dimensions.

Pen & Paper

Let's consider a simplified example to understand how LSA can identify semantically similar documents even with limited word overlap.

1. **Example Documents**:
   - Doc 1: ”The cat sat on the mat.”
   - Doc 2: ”The dog lay on the rug.”
   - Doc 3: ”The bird flew in the sky.”
2. **Term-Document Matrix (Term Frequency for Simplicity)**:

	Doc 1	Doc 2	Doc 3
the	1	1	1
cat	1	0	0
sat	1	0	0
on	1	1	1
mat	1	0	0
dog	0	1	0
lay	0	1	0
rug	0	1	0
bird	0	0	1
flew	0	0	1
in	0	0	1
sky	0	0	1

3. **Apply SVD and Reduce Dimensions**: After applying SVD to this matrix, we might choose to reduce the dimensionality to, say, 2 latent semantic dimensions by keeping the top two singular values and their corresponding vectors.
4. **Semantic Space**: In this reduced 2-dimensional semantic space, the representations of the documents would likely be transformed such that Doc 1 (”cat”, ”mat”) and Doc 2 (”dog”, ”rug”) are closer to each other than either is to Doc 3 (”bird”, ”sky”). This proximity reflects the underlying semantic theme of animals and resting places, even though the specific words used are different. LSA captures this latent relationship through the patterns of co-occurrence with other words (like ”the” and ”on”).

Advantages and Limitations of LSA

Advantages:

- **Captures Semantic Relationships**: LSA can uncover non-obvious semantic associations between words and documents that go beyond simple keyword matching.
- **Handles Synonymy and Polysemy (to some extent)**: By considering the context in which words appear, LSA can mitigate the issues of synonymy (different words with similar meanings) and polysemy (one word with multiple meanings).
- **Reduces Noise and Sparsity**: The dimensionality reduction step helps in filtering out less important variations in word usage and can lead to a denser, more robust representation compared to the original sparse term-document matrix.

Limitations:

- **Bag-of-Words Assumption**: LSA treats documents as bags of words, disregarding the order and grammatical structure of the words, which can be crucial for understanding meaning.
- **Limited Handling of Negation and Complex Structures**: LSA struggles with capturing negations (e.g., "not happy") and more complex linguistic relationships that depend on word order or syntax.
- **Context Dependency**: The semantic space learned by LSA is static and corpus-dependent. It may not effectively adapt to new contexts or domains without retraining.
- **Choice of Dimensionality (k)**: Selecting the optimal number of reduced dimensions k is not always straightforward and can significantly impact the performance of LSA. It often requires empirical tuning.
- **Interpretation of Latent Dimensions**: The derived latent semantic dimensions are not always easily interpretable or aligned with intuitive semantic concepts.

3.3.5 Canonical Correlation Analysis (CCA)

Canonical Correlation Analysis (CCA) is a multivariate statistical technique that aims to identify and quantify the relationships between two sets of variables. Unlike methods that examine the correlation between individual pairs of variables, CCA seeks to find linear combinations of the variables within each set that exhibit the maximum correlation with each other. These linear combinations are called canonical variables, and their correlations are called canonical correlations.
Given two sets of variables, represented by the matrices $\mathbf{X} \in \mathbb{R}^{n\times p}$ and $\mathbf{Y} \in \mathbb{R}^{n\times q}$, where n is the number of observations, p is the number of variables in $\mathbf{X}$, and q is the number of variables in $\mathbf{Y}$, CCA seeks to find weight vectors $\mathbf{a} \in \mathbb{R}^p$ and $\mathbf{b} \in \mathbb{R}^q$ such that the correlation between the linear combinations $\mathbf{u} = \mathbf{Xa}$ and $\mathbf{v} = \mathbf{Yb}$ is maximized.
Mathematically, the objective is to solve the following maximization problem:

$$\max_{\mathbf{a},\mathbf{b}} \operatorname{corr}(\mathbf{Xa}, \mathbf{Yb}) = \max_{\mathbf{a},\mathbf{b}} \frac{\mathbf{a}^T\mathbf{X}^T\mathbf{Yb}}{\sqrt{\mathbf{a}^T\mathbf{X}^T\mathbf{Xa}}\sqrt{\mathbf{b}^T\mathbf{Y}^T\mathbf{Yb}}}$$

CCA finds a sequence of pairs of canonical variables $(\mathbf{u}_1, \mathbf{v}_1), (\mathbf{u}_2, \mathbf{v}_2), \ldots, (\mathbf{u}_m, \mathbf{v}_m)$, where $m = \min(p, q)$, such that the correlation between each pair is maximized, and each subsequent pair is uncorrelated with all preceding pairs.

Key Concepts in CCA

- **Canonical Variables**: These are the derived linear combinations of the original variables from each set that exhibit maximal correlation. The first pair of canonical variables has the highest correlation, the second pair has the next highest correlation (while being uncorrelated with the first pair), and so on.
- **Canonical Correlation**: This is the correlation coefficient between a pair of canonical variables. It quantifies the strength of the linear relationship between the two sets of variables captured by that specific pair of canonical variables. The canonical correlations are between 0 and 1.
- **Canonical Weights (or Coefficients)**: These are the elements of the weight vectors **a** and **b** that define the linear combinations forming the canonical variables. They indicate the contribution of each original variable to its respective canonical variable.

Algorithm 14 Canonical Correlation Analysis (CCA)

```
1: **Input:** Two datasets $X \in \mathbb{R}^{n \times p}$ and $Y \in \mathbb{R}^{n \times q}$ containing $n$ observations and $p$ and $q$ features respectively
2:
3: Center the data (subtract the mean of each variable):
4:     $\bar{X} = X - \mathbf{1}_n \cdot \bar{\mathbf{x}}^T$, where $\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n} X_i$ (row vector of means)
5:     $\bar{Y} = Y - \mathbf{1}_n \cdot \bar{\mathbf{y}}^T$, where $\bar{\mathbf{y}} = \frac{1}{n}\sum_{i=1}^{n} Y_i$ (row vector of means)
6:
7: Compute the covariance matrices:
8:     $C_{XX} = \frac{1}{n-1}\bar{X}^T\bar{X}$    ▷ Covariance matrix of $X$
9:     $C_{YY} = \frac{1}{n-1}\bar{Y}^T\bar{Y}$    ▷ Covariance matrix of $Y$
10:    $C_{XY} = \frac{1}{n-1}\bar{X}^T\bar{Y}$    ▷ Cross-covariance matrix between $X$ and $Y$
11:    $C_{YX} = C_{XY}^T = \frac{1}{n-1}\bar{Y}^T\bar{X}$    ▷ Cross-covariance matrix between $Y$ and $X$
12:
13: Solve the generalized eigenvalue problems:
14:    $(C_{XX}^{-1}C_{XY}C_{YY}^{-1}C_{YX})\mathbf{a}_i = \lambda_i^2\mathbf{a}_i$    ▷ Eigenproblem for canonical weights of $X$
15:    $(C_{YY}^{-1}C_{YX}C_{XX}^{-1}C_{XY})\mathbf{b}_i = \lambda_i^2\mathbf{b}_i$    ▷ Eigenproblem for canonical weights of $Y$
16:    where $\lambda_i$ are the canonical correlations (square root of the eigenvalues), and $\mathbf{a}_i$ and $\mathbf{b}_i$ are the corresponding canonical weight vectors.
17:
18: **Output:** Canonical weight vectors $\mathbf{a}_i, \mathbf{b}_i$ and canonical correlations $\lambda_i$ for $i = 1, \ldots, m$, where $m = \min(p, q)$.
```

The typical steps involved in performing CCA are:

1. **Data Preprocessing**: Standardize or center the variables in both datasets $\mathbf{X}$ and $\mathbf{Y}$ to have zero mean. This is often done to ensure that variables with larger scales do not dominate the analysis.

2. **Covariance Matrix Computation**: Calculate the covariance matrices for each set of variables (C_{XX} and C_{YY}) and the cross-covariance matrix between the two sets (C_{XY}).

3. **Eigenvalue Problem Solving**: Solve the generalized eigenvalue problems derived from the covariance matrices. This typically involves finding the eigenvalues and eigenvectors of the matrices $C_{XX}^{-1}C_{XY}C_{YY}^{-1}C_{YX}$ and $C_{YY}^{-1}C_{YX}C_{XX}^{-1}C_{XY}$. The eigenvalues ($\lambda_i^2$) are the squares of the canonical correlations, and the eigenvectors ($\mathbf{a}_i$ and $\mathbf{b}_i$) are the canonical weight vectors.

4. **Canonical Variable Computation**: Once the canonical weight vectors $\mathbf{a}_i$ and $\mathbf{b}_i$ are obtained, the canonical variables are computed as the linear combinations of the original variables: $\mathbf{u}_i = \mathbf{X}\mathbf{a}_i$ and $\mathbf{v}_i = \mathbf{Y}\mathbf{b}_i$.

5. **Interpretation**: The canonical correlations λ_i indicate the strength of the relationship between the i–th pair of canonical variables. The canonical weight vectors help in understanding which original variables contribute most to these relationships.

Applications of CCA

CCA is a versatile technique with applications in various fields where the relationship between two multidimensional datasets is of interest:

- **Neuroscience**: Researchers use CCA to investigate the relationships between different modalities of brain activity, such as EEG and fMRI data, or between neural activity and behavioral measures.
- **Genomics and Bioinformatics**: CCA can be employed to explore the associations between gene expression levels and other biological characteristics, such as drug response or disease status.
- **Social Sciences and Economics**: CCA can help in analyzing the interplay between sets of social indicators (e.g., education, income, health) and economic indicators (e.g., GDP, unemployment rate, inflation).
- **Information Retrieval and Multimedia Analysis**: CCA can be used to find relationships between different representations of the same data, such as textual descriptions and visual features of images or videos.

Pen & Paper

In our simplified numerical example:

$$\mathbf{X} = \begin{pmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{pmatrix}, \quad \mathbf{Y} = \begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{pmatrix}$$

After standardizing and computing the covariance matrices:

$$\mathbf{X}_{\text{std}} = \begin{pmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{Y}_{\text{std}} = \begin{pmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{pmatrix}$$

$$\mathbf{C}_{XX} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{C}_{YY} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{C}_{XY} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Solving the eigenvalue problems (note that the inverse of $\mathbf{C}_{XX}$ and $\mathbf{C}_{YY}$ are not uniquely defined as their determinant is 0, indicating collinearity within the variables. A pseudo-inverse would be needed in a practical implementation. However, for illustrative purposes, if we proceed conceptually, we will have the following: The equation $\mathbf{C}_{XX}^{-1}\mathbf{C}_{XY}\mathbf{C}_{YY}^{-1}\mathbf{C}_{YX}\mathbf{a} = \lambda^2\mathbf{a}$ would yield eigenvalues λ^2. In this specific case, due to the perfect linear dependence within each set and the perfect correlation between the sets, the largest canonical correlation will be 1. The corresponding eigenvectors **a** and **b** (canonical weights) define the linear combinations that yield maximally correlated canonical variables.
The computed canonical variables were:

$$\mathbf{u} = \mathbf{X}_{\text{std}}\mathbf{a} = \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix}, \quad \mathbf{v} = \mathbf{Y}_{\text{std}}\mathbf{b} = \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix}$$

The correlation between **u** and **v** is 1, indicating a perfect linear relationship captured by the canonical variables derived from the two original sets of variables. Although simplified, this example illustrates the goal of CCA in finding maximally correlated linear combinations across two datasets.

Chapter 4

Model Evaluation and Model Selection

4.1 Model Evaluation

Model evaluation is a crucial step in the machine learning process. It involves assessing the performance of a trained model on unseen data to determine its effectiveness and generalizability. In the context of machine learning, data is typically divided into two main categories: training data and test data.
Training data is the subset of the overall dataset that is used to teach the model. During the training process, the model learns patterns and relationships within this data, adjusting its parameters to minimize errors and improve its predictive capabilities. On the other hand, test data remains unseen by the model during training. This subset is reserved for evaluating the model's performance on new, previously unseen examples.
The importance of test data cannot be overstated. It serves as a proxy for real-world data that the model will encounter once deployed. By evaluating the model on test data, we can assess whether it has truly learned generalizable patterns or if it has merely memorized the training data (a phenomenon known as overfitting). Overfitting occurs when a model learns the training data too well, including its noise and specific details, leading to poor performance on new, unseen data.
It's worth noting that while it's common practice to split a dataset into 80% training (or 70% training) and 20% test (or 30% test), these ratios are not set in stone. The exact split can vary depending on the volume of available data and the specific requirements of the project. Generally, it's advisable to use at least 70% of the data for training, but this can be adjusted based on the dataset's size and the complexity of the problem at hand. Additionally, a separate **validation set** is often used during the model development and hyperparameter tuning phase to avoid overfitting to the test set.

4.1.1 Importance of Model Evaluation

Model evaluation is not a one-size-fits-all process. Different algorithms and types of data may require different evaluation metrics. The choice of the right evaluation metric is crucial and depends on the specific problem being solved, the business objectives, and the nature of the data. For instance, in fraud detection, the cost of a false negative (failing to identify a fraudulent transaction) might be much higher than the cost of a false positive (incorrectly flagging a legitimate transaction). Therefore, metrics that focus on minimizing false negatives (like recall) might be more important. Similarly, when evaluating a regression problem, the interpretability of the error (e.g., in the original units of the target variable) might be a key consideration, leading to the choice of RMSE or MAE over MSE.
A comprehensive understanding of various evaluation metrics is essential for data scientists

and machine learning engineers. This knowledge allows them to experiment with different techniques and select the most appropriate method for assessing their models. Both regression and classification techniques can be evaluated using a variety of metrics, each offering unique insights into the model's performance. There is no single "best" metric; the optimal choice is context-dependent.

4.2 Regression Evaluation Metrics

Regression models are designed to predict continuous values. As such, the evaluation metrics for regression take into account the continuous nature of the predictions. Some of the most widely used regression metrics include Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-Squared Error (R^2).

4.2.1 Basic Error Calculation

At the most fundamental level, error in regression is calculated as the difference between the actual value and the predicted value:

$$Error\,(e_i) = Actual\,value\,(y_i) - predicted\,value\,(\hat{y}_i)$$

For a dataset with m training examples, we can calculate the Total Error as the sum of individual errors:

$$Total\,Error\,(E) = \sum_{i=1}^{m}(y_i - \hat{y}_i)$$

The Mean Error (Bias) is then simply the average of these individual errors:

$$Mean\,Error\,(\bar{e}) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)$$

However, these basic error calculations have limitations. They don't provide an accurate measure of model performance due to three potential issues:

1. The error may be skewed towards large negative values, indicating systematic under-prediction.
2. The error may be skewed towards large positive values, indicating systematic over-prediction.
3. The errors may cancel each other out, resulting in a mean close to zero even if individual errors are large, masking significant prediction inaccuracies.

To address these issues, we need more sophisticated error metrics that aggregate the magnitude of the errors, regardless of their direction.

4.2.2 Mean Squared Error (MSE)

One way to avoid the problem of error cancellation is to square each error value before summing them. This gives us the Total Squared Error:

$$Total\,Squared\,Error\,(SSE) = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

The Mean Squared Error (MSE) is then calculated as the average of these squared errors:

$$Mean\,Squared\,Error\,(MSE) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

MSE is widely used in regression analysis. It provides a measure of the average squared difference between the predicted and actual values. Squaring the errors penalizes larger errors more heavily than smaller ones. However, it's important to note that MSE is sensitive to outliers. Even a few outliers can significantly increase the MSE, potentially giving a misleading impression of the model's overall performance. The units of MSE are also squared relative to the original target variable, which can make interpretation less intuitive.
Example: Suppose a model predicts house prices, and for three houses, the actual prices are \$300k, \$400k, and \$500k, while the predictions are \$320k, \$380k, and \$530k. The squared errors are $(300 - 320)^2 = 400$, $(400 - 380)^2 = 400$, and $(500 - 530)^2 = 900$. The MSE is $(400 + 400 + 900)/3 = 566.67\,(k\$)^2$.

4.2.3 Root Mean Squared Error (RMSE)

To address the scale issue of MSE (since it's in squared units), we often use the Root Mean Squared Error (RMSE). RMSE is simply the square root of MSE:

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

RMSE has the advantage of being in the same units as the target variable, making it more interpretable. It represents the standard deviation of the residuals (prediction errors). Like MSE, RMSE is also sensitive to outliers because it is based on the squared errors, but the effect is brought back to the original scale. It is a commonly used metric and is often preferred when larger errors are particularly undesirable.
Example (continued): The RMSE for the house price predictions would be $\sqrt{566.67} \approx 23.8\,k\$$. This means that, on average, the model's predictions are about \$23,800 away from the actual prices.

4.2.4 R-Squared (R^2)

Another commonly used metric in regression is R-Squared (R^2), also known as the coefficient of determination. R-Squared measures the proportion of variance in the dependent variable that is predictable from the independent variable(s). It provides an indication of the goodness of fit of the model. It's calculated as:

$$R^2 = 1 - \frac{Sum\,of\,Squares\,Residual\,(SSR)}{Total\,Sum\,of\,Squares\,(SST)}$$

Where:

$$SSR = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

And:

$$SST = \sum_{i=1}^{m}(y_i - \bar{y})^2$$

Here, $\bar{y}$ is the mean of the actual values. SSR represents the variance of the errors (unexplained variance by the model), and SST represents the total variance in the dependent variable. R-Squared essentially compares the performance of the fitted model to that of a simple model that always predicts the mean of the dependent variable. The value of R^2 ranges from 0 to 1, with values closer to 1 indicating that a larger proportion of the variance in the target variable is explained by the model, hence a better fit. An R^2 of 1 indicates a perfect fit, where the model explains all the variance in the dependent variable. An R^2 of 0 suggests that the model explains none of the variance and performs no better than simply predicting the mean.
Generally, an R^2 value above 0.5 is often considered acceptable for many regression problems, though this can vary significantly depending on the specific domain and application. In some fields, even a lower R^2 might be considered meaningful if the phenomenon being modeled is inherently noisy or complex. A limitation of R^2 is that it does not penalize the addition of irrelevant independent variables to the model; in fact, R^2 can increase even if the added variables do not improve the model's predictive power. **Adjusted R-Squared** addresses this limitation by penalizing the inclusion of unnecessary predictors.
Example (continued): Suppose the mean of the actual house prices is $\bar{y} = (300+400+500)/3 = 400\,k\$$. Then, $SST = (300-400)^2+(400-400)^2+(500-400)^2 = 10000+0+10000 = 20000\,(k\$)^2$. We found $SSR = 400+400+900 = 1700\,(k\$)^2$. Therefore, $R^2 = 1-(1700/20000) = 1-0.085 = 0.915$. This indicates that approximately 91.5

4.2.5 Mean Absolute Error (MAE)

Instead of squaring the errors, we can also take their absolute values. This approach, known as Mean Absolute Error (MAE), avoids the problem of negative errors canceling out positive ones and provides a measure of the average magnitude of the errors in the original units of the target variable:

$$Mean\,Absolute\,Error\,(MAE) = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i|$$

MAE is less sensitive to outliers compared to MSE and RMSE because it treats all errors equally, regardless of their magnitude. It provides a more robust measure of the average prediction error when the dataset contains extreme values. MAE is also more easily interpretable than MSE as it is in the same units as the target variable.
Example (continued): The absolute errors for the house price predictions are $|300-320| = 20$, $|400-380| = 20$, and $|500-530| = 30$. The MAE is $(20+20+30)/3 = 23.33\,k\$$. On average, the model's predictions are about \$23,330 away from the actual prices.

4.2.6 Mean Absolute Percentage Error (MAPE)

For scenarios where we're interested in the relative size of errors, we can use the Mean Absolute Percentage Error (MAPE):

$$Mean\,Absolute\,Percentage\,Error\,(MAPE) = \frac{1}{m}\sum_{i=1}^{m}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\%$$

MAPE expresses the average absolute error as a percentage of the actual values. For instance, a MAPE of 20% indicates that, on average, the model's predictions deviate from the actual values by 20%. Lower MAPE values indicate more accurate predictions, with 0% being a perfect prediction. However, MAPE has some limitations. It is undefined if any of the actual values are zero, and it can be asymmetric, meaning that a positive error of a certain percentage has a different impact on the MAPE than a negative error of the same percentage. Also, it can put a heavier penalty on negative errors when the actual value is small.

Example (continued): The percentage errors are $|(300 - 320)/300| \times 100\% = 6.67\%$, $|(400 - 380)/400| \times 100\% = 5\%$, and $|(500 - 530)/500| \times 100\% = 6\%$. The MAPE is $(6.67 + 5 + 6)/3 = 5.89\%$. On average, the model's predictions are about 5.89

4.3 Classification Evaluation Metrics

While regression models predict continuous values, classification models predict discrete categories. As such, they require a different set of evaluation metrics that assess the model's ability to correctly assign instances to these categories. Some of the most common classification metrics include Accuracy, Recall, Precision, F1-score, and ROC (Receiver Operating Characteristic) curve with its associated AUC (Area Under the Curve).

To understand these metrics, we first need to introduce the concept of a confusion matrix.

4.3.1 Confusion Matrix

A confusion matrix is a table that describes the performance of a classification model on a set of test data for which the true values are known. For a binary classification problem (where we predict either True (1) or False (0), often referred to as the positive and negative classes, respectively), the confusion matrix has four categories:

- **True Positive (TP)**: Cases where the model correctly predicted the positive class (actual: True, predicted: True).
- **False Positive (FP)**: Cases where the model incorrectly predicted the positive class (actual: False, predicted: True). Also known as a Type I error.
- **False Negative (FN)**: Cases where the model incorrectly predicted the negative class (actual: True, predicted: False). Also known as a Type II error.
- **True Negative (TN)**: Cases where the model correctly predicted the negative class (actual: False, predicted: False).

	Predicted	
Actual	True (Positive)	False (Negative)
True (Positive)	True Positive (TP)	False Negative (FN)
False (Negative)	False Positive (FP)	True Negative (TN)

Table 4.1: Confusion Matrix for Binary Classification

For multi-class classification problems with n classes, the confusion matrix is an $n \times n$ matrix where the (i, j)-th entry represents the number of instances that actually belong to class i but were predicted to be in class j. The diagonal entries represent the correctly classified instances for each class.

4.3.2 Accuracy

Accuracy is one of the most intuitive and commonly used metrics for classification. It's simply the ratio of correct predictions to the total number of predictions:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

While accuracy is easy to understand and compute, it can be misleading in cases of imbalanced datasets. For instance, in a disease prediction task where 95% of the population is healthy, a model that always predicts "not ill" would have 95% accuracy, despite being useless for identifying ill patients. The dominance of the majority class in the calculation overshadows the model's poor performance on the minority class.
Consider the following confusion matrix for a disease prediction task: The accuracy for this

	Predicted	
Actual	Ill	Not Ill
Ill	48 (TP)	2 (FN)
Not Ill	5 (FP)	45 (TN)

Table 4.2: Disease Prediction Confusion Matrix

model would be:

$$Accuracy = \frac{48 + 45}{48 + 2 + 45 + 5} = \frac{93}{100} = 0.93$$

This 93% accuracy might seem impressive at first glance. However, it doesn't tell the whole story. The model misclassified 5 ill patients as not ill (False Negatives), which could have severe consequences in a medical context. This highlights the need for additional metrics that provide a more nuanced view of the model's performance, particularly its ability to correctly identify instances of the positive class.

4.3.3 Precision and Recall

Precision and recall are two metrics that provide more detailed insights into a classification model's performance, especially for imbalanced datasets. They focus on the model's ability to correctly identify positive instances and avoid incorrect positive predictions.
Precision is the ratio of correctly predicted positive observations (True Positives) to the total number of observations predicted as positive (True Positives + False Positives):

$$Precision = \frac{TP}{TP + FP}$$

Precision answers the question: "Of all the instances the model labeled as positive, what proportion was actually positive?" High precision means that when the model predicts the positive class, it is very likely to be correct.
Recall (also known as sensitivity, true positive rate (TPR), or hit rate) is the ratio of correctly predicted positive observations (True Positives) to the total number of actual positive observations (True Positives + False Negatives):

$$Recall = \frac{TP}{TP + FN}$$

Recall answers the question: "Of all the actual positive instances, what proportion did the model correctly identify?" High recall means that the model is good at finding all the positive instances.

The value of both Precision and Recall ranges between 0 and 1, where 1 represents perfect precision or recall, and 0 represents the worst possible score.
For our disease prediction example:

$$Precision = \frac{48}{48+5} = \frac{48}{53} \approx 0.9066$$
$$Recall = \frac{48}{48+2} = \frac{48}{50} = 0.96$$

Precision of approximately 0.91 tells us that when our model predicts a patient is ill, it is correct about 91
The choice between optimizing for precision or recall depends on the specific problem and the relative costs of false positives and false negatives.

False Positive (FP)

A false positive occurs when the model incorrectly predicts the positive class for an instance that actually belongs to the negative class. In other words, the model signals an occurrence of an event when it hasn't actually happened.
Example: In a medical test for a disease, a false positive means the test indicates the presence of the disease in a healthy person. This could lead to unnecessary anxiety, further tests, and potentially even unnecessary treatment.

False Negative (FN)

A false negative occurs when the model incorrectly predicts the negative class for an instance that actually belongs to the positive class. This means the model fails to detect an event that has actually occurred.
Example: In the same medical test for a disease, a false negative means the test fails to detect the disease in a person who actually has it. This could delay necessary treatment and have serious health consequences.

Significance of False Positives and False Negatives

The significance of false positives and false negatives depends on the context of the problem and the associated costs or risks.

- **False Positives (Type I Error):**
  - May lead to unnecessary actions, such as additional tests, treatments, or interventions, which can be costly and cause distress.
  - In contexts like spam detection, a false positive might mark a legitimate email as spam, causing the user to miss important communication.
  - In security systems, a false positive might cause unwarranted alarms and responses, wasting resources and potentially desensitizing operators.
- **False Negatives (Type II Error):**
  - Can result in missed detections, such as not identifying a disease that needs treatment, potentially leading to the progression of the illness and worse outcomes.
  - In fraud detection, a false negative might let fraudulent transactions pass unnoticed, leading to financial losses.
  - In safety systems, a false negative might overlook a critical threat, leading to potential harm or damage.

4.3.4 F1 Score

The F1 score is the harmonic mean of precision and recall, providing a single score that balances both metrics. It is particularly useful when you want to find a compromise between precision and recall, especially in situations with imbalanced class distributions. The harmonic mean is used instead of a simple arithmetic mean because it penalizes extreme values, thus a high F1 score requires both precision and recall to be relatively high.

$$F1\,Score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

The F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.
For our disease prediction example:

$$F1\,Score = \frac{2 \times 0.9066 \times 0.96}{0.9066 + 0.96} = \frac{1.736752}{1.8666} \approx 0.9304$$

The F1 score of approximately 0.93 suggests a good balance between precision and recall for this model.
The choice between focusing on precision, recall, or the F1 score depends on the specific application and the relative importance of minimizing false positives versus false negatives. If false positives are costly, we might prioritize precision. If missing positive instances is critical, we might prioritize recall. If a balance is desired, the F1 score is a good metric to consider.

Pen & Paper

In this example, we consider a binary classification problem where we classify emails as either "Spam" or "Not Spam." We use the confusion matrix to calculate key performance metrics: Precision, Recall, F1 Score, and Accuracy.
Suppose we have the following confusion matrix for a test set of 100 emails:

	Predicted Spam	**Predicted Not Spam**
Actual Spam	40	10
Actual Not Spam	20	30

Table 4.3: Confusion Matrix

From this confusion matrix, we derive the following values:

- True Positives (TP) = 40
- False Positives (FP) = 20
- True Negatives (TN) = 30
- False Negatives (FN) = 10

Computing accuracy,

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{40 + 30}{40 + 30 + 20 + 10} = \frac{70}{100} = 0.70$$

Calculating Precision,

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{40}{40 + 20} = \frac{40}{60} = 0.67$$

Recall is,

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{40}{40+10} = \frac{40}{50} = 0.80$$

Now, Computing F1 Score,

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = 2 \times \frac{0.67 \times 0.80}{0.67 + 0.80} = 2 \times \frac{0.536}{1.47} = \frac{1.072}{1.47} = 0.73$$

Interpretation

- **Accuracy (0.70):** This means that 70% of the total emails were correctly classified by the model. Accuracy is a useful measure but can be misleading if the dataset is imbalanced. While an accuracy of 0.70 indicates that the model correctly classifies a majority of the emails, it can be misleading if the dataset is imbalanced (i.e., a significantly larger number of non-spam emails compared to spam emails). In this case, accuracy alone is not sufficient to judge the model's effectiveness, as it doesn't account for the balance between precision and recall.

- **Precision (0.67):** This means that 67% of the emails predicted as spam were actually spam. High precision indicates a low false positive rate. A precision of 0.67 is relatively good, meaning the model does not produce an excessive number of false positives (non-spam emails incorrectly labeled as spam). However, there is still a notable 33% false positive rate, which may not be acceptable in all scenarios.

- **Recall (0.80):** This means that 80% of the actual spam emails were correctly identified by the model. High recall indicates a low false negative rate. A recall of 0.80 is quite high, indicating that the model successfully identifies most spam emails. This suggests that the model is effective at capturing spam but is not perfect, as 20% of actual spam emails were missed.

- **F1 Score (0.73):** The F1 Score is a balance between precision and recall. An F1 score of 0.73 suggests that the model maintains a decent balance between precision and recall. This balanced measure shows that while the model is good, there is still room for improvement, particularly in reducing false positives or increasing true positives.

Overall, the model shows a balanced performance with decent precision and high recall. It effectively captures most of the spam emails while maintaining a moderate level of correctness in its spam predictions. To further improve the model, efforts could focus on reducing the false positive rate (increasing precision) without significantly affecting the recall. This could involve fine-tuning the model, adjusting thresholds, or incorporating additional features.

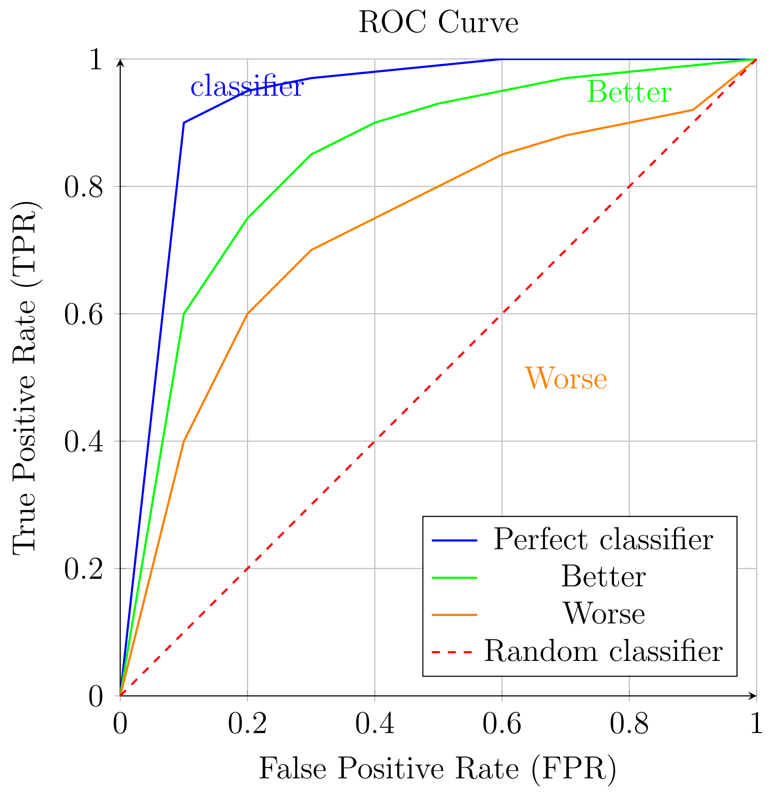
4.3.5 ROC - Receiver Operating Characteristic Curve

The Receiver Operating Characteristic (ROC) curve is a graphical representation of a classification model's performance across all classification thresholds. It plots two parameters:

- True Positive Rate (TPR), which is equivalent to Recall

- False Positive Rate (FPR), which is calculated as 1 - Specificity, where Specificity is the True Negative Rate

The ROC curve provides a visual means to assess how well the classifier distinguishes between the positive and negative classes.



Interpretations: ROC Curves:

- **Perfect Classifier (Blue Curve):** This classifier perfectly separates the positive and negative classes, achieving a TPR of 1 with a very low FPR. This results in a curve that quickly rises to the top left corner and stays there.
- **Better Classifier (Green Curve):** This classifier performs well, with a high TPR and relatively low FPR. The curve rises steeply, indicating good performance but not perfect.
- **Worse Classifier (Orange Curve):** This classifier performs worse than the better classifier, with a less steep rise in TPR and a higher FPR for the same thresholds.

Random Classifier (Red Dashed Line): This line represents the performance of a random classifier that has no predictive power. It follows a diagonal line from (0,0) to (1,1), indicating equal rates of true and false positives.
To create ROC curve, we need to plot the FPR values agains TPR values at different decision thresholds.

4.3.6 AUC - Area Under the ROC Curve

The Area Under the Curve (AUC) is a numerical measure that summarizes the performance of a binary classifier across all possible threshold values. The AUC is computed from the ROC (Receiver Operating Characteristic) curve, which plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. The AUC provides an aggregate measure of a classifier's performance, regardless of the specific threshold used for classification.

- AUC $= 1$: Perfect classifier. The ROC curve will pass through the top left corner of the plot, indicating that the classifier perfectly separates positive and negative instances.

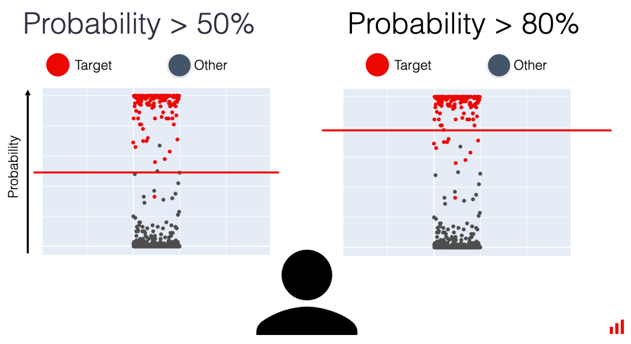


Figure 4.1: Enter Caption

- AUC = 0.5: Random classifier. The ROC curve is a diagonal line, indicating that the classifier is no better than random guessing.
- AUC ¡ 0.5: Worse than random. This indicates that the classifier performs worse than a random guesser. Such a classifier can be inverted to achieve better-than-random performance.

Thus, when comparing multiple classifiers, the one with the highest AUC is generally considered the best in terms of distinguishing between positive and negative classes.
AUC is particularly useful in scenarios where the classes are imbalanced. Since AUC considers both TPR and FPR, it provides a more holistic view of the classifier's performance than accuracy alone.
Consider a medical test designed to detect a disease. The test results in a continuous score, and a threshold is applied to classify individuals as positive (disease present) or negative (disease absent). The ROC curve and the AUC help in evaluating the test's performance:

- **High AUC:** Indicates that the test is effective in distinguishing between diseased and healthy individuals.
- **Low AUC:** Suggests that the test is not effective, possibly resulting in many false positives (healthy individuals incorrectly diagnosed as diseased) or false negatives (diseased individuals incorrectly diagnosed as healthy).

4.4 Multi-class Evaluation Metrics

Evaluating multi-class classification models requires different approaches compared to binary classification. We'll explore various metrics and techniques for assessing multi-class model performance.

4.4.1 Multi-class Confusion Matrix

For a problem with K classes, the confusion matrix is a $K \times K$ matrix where the element at position (i, j) represents the number of instances of true class A that were classified as class P.
For example, consider a 3-class problem with classes A, B, and C:
From this confusion matrix, we can calculate various metrics for each class:

- True Positive (TP): Instances correctly classified as the given class (diagonal elements).

Actual \ Predicted	A	B	C
A	50	10	5
B	8	100	7
C	4	6	80

Table 4.4: Example of a 3x3 Confusion Matrix

- False Positive (FP): Instances incorrectly classified as the given class (sum of column minus TP).
- False Negative (FN): Instances of the given class incorrectly classified as other classes (sum of row minus TP).
- True Negative (TN): Instances correctly classified as not belonging to the given class (sum of all elements not in the class's row or column).

Calculations for Class A

$$\begin{aligned}
\text{TP(A)} &= 50\\
\text{FP(A)} &= (50+8+4)-50 = 12\\
\text{FN(A)} &= (50+10+5)-50 = 15\\
\text{TN(A)} &= 100+7+6+80 = 193
\end{aligned}$$

Calculations for Class B

$$\begin{aligned}
\text{TP(B)} &= 100\\
\text{FP(B)} &= (10+100+6)-100 = 16\\
\text{FN(B)} &= (8+100+7)-100 = 15\\
\text{TN(B)} &= 50+5+4+80 = 139
\end{aligned}$$

Calculations for Class C

$$\begin{aligned}
\text{TP(C)} &= 80\\
\text{FP(C)} &= (5+7+80)-80 = 12\\
\text{FN(C)} &= (4+6+80)-80 = 10\\
\text{TN(C)} &= 50+10+8+100 = 168
\end{aligned}$$

4.4.2 Multi-class Metrics

In multi-class scenarios, we often calculate metrics for each class separately and then aggregate them. There are two main approaches to aggregation:

Macro-averaging

Macro-averaging gives equal weight to each class, regardless of its frequency in the dataset.

$$\text{Macro-averaged Precision} = \frac{1}{K}\sum_{k=1}^{K}\text{Precision}^{(k)} \tag{4.1}$$

$$\text{Macro-averaged Recall} = \frac{1}{K}\sum_{k=1}^{K}\text{Recall}^{(k)} \tag{4.2}$$

$$\text{Macro-averaged F1 Score} = \frac{1}{K}\sum_{k=1}^{K}\text{F1 Score}^{(k)} \tag{4.3}$$

Where K is the number of classes, and Precision$^{(k)}$, Recall$^{(k)}$, and F1 Score$^{(k)}$ are the respective metrics for the k-th class.

Micro-averaging

Micro-averaging aggregates the contributions of all classes to compute the average metric. It gives equal weight to each instance classification decision.

$$\text{Micro-averaged Precision} = \frac{\sum_{k=1}^{K}\text{TP}^{(k)}}{\sum_{k=1}^{K}(\text{TP}^{(k)} + \text{FP}^{(k)})} \tag{4.4}$$

$$\text{Micro-averaged Recall} = \frac{\sum_{k=1}^{K}\text{TP}^{(k)}}{\sum_{k=1}^{K}(\text{TP}^{(k)} + \text{FN}^{(k)})} \tag{4.5}$$

$$\text{Micro-averaged F1 Score} = \frac{2 * \text{Micro Precision} * \text{Micro Recall}}{\text{Micro Precision} + \text{Micro Recall}} \tag{4.6}$$

4.4.3 When to Use Micro vs Macro Averaging

- Use micro-averaging when you want to weight each instance or prediction equally. It's particularly useful for imbalanced datasets where class imbalance is a natural characteristic of the problem.
- Use macro-averaging when you want to give equal importance to each class, regardless of its frequency. This is useful when all classes are equally important, despite class imbalance.
- Consider using weighted macro-averaging for imbalanced datasets where you want to account for class frequency while still giving some importance to minority classes.

Pen & Paper

Given the above 3x3 confusion matrix for a multi-class classification problem, let's calculate macro and micro metrics and interpret the results.
on

4.5 Calculations

4.5.1 Step 1: Calculate TP, FP, FN, TN for each class

For each class:

- TP (True Positive): Correctly classified instances
- FP (False Positive): Instances incorrectly classified as this class
- FN (False Negative): Instances of this class incorrectly classified as other classes
- TN (True Negative): Instances correctly classified as not belonging to this class

$$\begin{aligned}
&\text{Class A:} \quad TP_A = 50, \quad FP_A = 8 + 4 = 12, \quad FN_A = 10 + 5 = 15, \quad TN_A = 100 + 7 + 6 + 80 = 193\\
&\text{Class B:} \quad TP_B = 100, \quad FP_B = 10 + 6 = 16, \quad FN_B = 8 + 7 = 15, \quad TN_B = 50 + 5 + 4 + 80 = 139\\
&\text{Class C:} \quad TP_C = 80, \quad FP_C = 5 + 7 = 12, \quad FN_C = 4 + 6 = 10, \quad TN_C = 50 + 10 + 8 + 100 = 168
\end{aligned}$$

4.5.2 Step 2: Calculate Precision, Recall, and F1-score for each class

For each class:

- Precision = TP / (TP + FP)
- Recall = TP / (TP + FN)
- F1-score = 2 * (Precision * Recall) / (Precision + Recall)

$$\begin{aligned}
&\text{Class A:} \quad \text{Precision}_A = \frac{50}{50+12} \approx 0.8065, \quad \text{Recall}_A = \frac{50}{50+15} \approx 0.7692, \quad \text{F1}_A \approx 0.7873\\
&\text{Class B:} \quad \text{Precision}_B = \frac{100}{100+16} \approx 0.8621, \quad \text{Recall}_B = \frac{100}{100+15} \approx 0.8696, \quad \text{F1}_B \approx 0.8658\\
&\text{Class C:} \quad \text{Precision}_C = \frac{80}{80+12} \approx 0.8696, \quad \text{Recall}_C = \frac{80}{80+10} \approx 0.8889, \quad \text{F1}_C \approx 0.8791
\end{aligned}$$

4.5.3 Step 3: Calculate Macro-averaged metrics

Macro-averaged metrics give equal weight to each class:

$$\begin{aligned}
\text{Macro Precision} &= \frac{\text{Precision}_A + \text{Precision}_B + \text{Precision}_C}{3} \approx 0.8461\\
\text{Macro Recall} &= \frac{\text{Recall}_A + \text{Recall}_B + \text{Recall}_C}{3} \approx 0.8426\\
\text{Macro F1} &= \frac{\text{F1}_A + \text{F1}_B + \text{F1}_C}{3} \approx 0.8441
\end{aligned}$$

4.5.4 Step 4: Calculate Micro-averaged metrics

Micro-averaged metrics aggregate the contributions of all classes:

$$\begin{aligned}
\text{Micro Precision} &= \frac{TP_A + TP_B + TP_C}{TP_A + TP_B + TP_C + FP_A + FP_B + FP_C} = \frac{230}{230+40} \approx 0.8519 \\
\text{Micro Recall} &= \frac{TP_A + TP_B + TP_C}{TP_A + TP_B + TP_C + FN_A + FN_B + FN_C} = \frac{230}{230+40} \approx 0.8519 \\
\text{Micro F1} &= 2 * \frac{\text{Micro Precision} * \text{Micro Recall}}{\text{Micro Precision} + \text{Micro Recall}} \approx 0.8519
\end{aligned}$$

Interpretation

1. **Individual Class Performance:**
   - Class A has the lowest performance across all metrics, indicating it's the most challenging class to predict correctly.
   - Class C has the highest performance, suggesting it's the easiest to distinguish from the others.
   - Class B shows balanced performance between precision and recall.
2. **Macro vs. Micro Averages:**
   - The macro-averaged metrics (0.8461, 0.8426, 0.8441) are slightly lower than the micro-averaged metrics (all 0.8519).
   - This suggests that the model performs slightly better on more frequent classes, as micro-averaging gives more weight to classes with more instances.
3. **Overall Model Performance:**
   - With both macro and micro F1-scores above 0.84, the model shows good overall performance across all classes.
   - The closeness of macro and micro averages indicates relatively balanced performance across classes, despite some variations.
4. **Areas for Improvement:**
   - Focus on improving the model's performance for Class A, particularly its recall, which is the lowest among all classes.
   - Investigate why Class A is more challenging to predict correctly and consider strategies to address this (e.g., feature engineering, data augmentation).

4.6 Advanced Model Selection Techniques

Model selection is a fundamental aspect of building effective machine learning systems. It involves choosing the most appropriate model from a collection of candidate models for a given task and dataset. Advanced model selection techniques go beyond simple train-test splits and employ more rigorous evaluation strategies to ensure the selected model generalizes well to unseen data.

4.6.1 Cross-Validation Strategies

Cross-validation (CV) is a robust statistical method for evaluating the performance of machine learning models and assessing how well they are likely to generalize to new, independent data. By partitioning the data into multiple subsets and iteratively training and evaluating the model, CV provides a more reliable estimate of the model's predictive capabilities than a single train-test split. The choice of cross-validation strategy depends critically on the characteristics of the data, such as its size, class distribution (for classification), and temporal nature (for time series data).

K-Fold Cross-Validation

The dataset is divided into k equal-sized folds. The model is trained and evaluated k times, with each fold serving as the test set once and the remaining $k-1$ folds as the training set. The final performance is the average across all k evaluations.

Algorithm 15 K-Fold Cross-Validation

```
1: procedure KFoldCV($data, k, model$)
2:     $n \leftarrow$ number of samples in $data$
3:     Split $data$ into $k$ disjoint folds $fold^{(1)}, fold^{(2)}, \dots, fold^{(k)}$ of approximately equal size
4:     $evaluation\_scores \leftarrow$ empty list
5:     for $i = 1$ to $k$ do
6:         $test\_fold \leftarrow fold^{(i)}$    ▷ Select the $i$-th fold as the test set
7:         $train\_folds \leftarrow$ all folds in $\{fold^{(1)}, \dots, fold^{(k)}\} \setminus \{fold^{(i)}\}$    ▷ Use the remaining $k-1$ folds for training
8:         $model$.train($train\_folds$)    ▷ Train the given model on the training folds
9:         $score \leftarrow model$.evaluate($test\_fold$)    ▷ Evaluate the trained model on the test fold
10:        Append $score$ to $evaluation\_scores$
11:    end for
12:    $average\_score \leftarrow$ average of $evaluation\_scores$
13:    return $average\_score$
14: end procedure
```

Advantages:

- Provides a less biased estimate of generalization performance than a single train-test split.
- Utilizes all data for both training and testing (in different iterations).
- Relatively easy to implement.

Disadvantages:

- Can be computationally expensive, especially for large datasets and complex models (model needs to be trained k times).
- The choice of k can influence the results (e.g., lower k can lead to higher bias, higher k can lead to higher variance).
- Not ideal for time series data due to the potential for data leakage from future to past.

Stratified K-Fold Cross-Validation

A variation of K-Fold designed for classification tasks with imbalanced datasets. It ensures that each fold contains approximately the same proportion of samples from each target class as the original dataset, providing a more reliable evaluation of model performance across all classes.

Algorithm 16 Stratified K-Fold Cross-Validation

```
1: procedure StratifiedKFoldCV($data, labels, k, model$)
2:     Split $data$ into $k$ stratified folds based on $labels$
3:     $evaluation\_scores \leftarrow$ empty list
4:     for $i = 1$ to $k$ do
5:         $test\_indices \leftarrow$ indices of samples in the $i$-th fold
6:         $train\_indices \leftarrow$ indices of samples in all other folds
7:         $X\_train \leftarrow data[train\_indices]$
8:         $y\_train \leftarrow labels[train\_indices]$
9:         $X\_test \leftarrow data[test\_indices]$
10:        $y\_test \leftarrow labels[test\_indices]$
11:        Train $model$ on $X\_train, y\_train$
12:        $score \leftarrow model.\text{evaluate}(X\_test, y\_test)$
13:        Append $score$ to $evaluation\_scores$
14:    end for
15:    $average\_score \leftarrow$ average of $evaluation\_scores$
16:    return $average\_score$
17: end procedure
```

Advantages:

- Ensures that each fold is representative of the class distribution in the original dataset.
- Provides a more accurate evaluation for imbalanced datasets compared to standard K-Fold.
- Helps in assessing the model's ability to generalize to minority classes.

Disadvantages

- Primarily applicable to classification tasks.
- Shares the computational cost of standard K-Fold.
- The stratification process might be complex for multi-label classification or more intricate data structures.

Leave-One-Out Cross-Validation (LOOCV)

An extreme case of K-Fold where k equals the number of samples. Each sample is used as the test set once, with the rest used for training. LOOCV offers low bias but can be computationally expensive and exhibit high variance, especially with small datasets.
Advantages:

- Provides a nearly unbiased estimate of the test error as almost all data is used for training in each iteration.
- Deterministic result for a given model and data order.

Algorithm 17 Leave-One-Out Cross-Validation (LOOCV)

```
1: procedure LOOCV($data, labels, model$)
2:     $n \leftarrow$ number of samples in $data$
3:     $evaluation\_scores \leftarrow$ empty list
4:     for $i = 1$ to $n$ do
5:         $test\_sample \leftarrow data[i]$
6:         $test\_label \leftarrow labels[i]$
7:         $train\_data \leftarrow data$[all indices except $i$]
8:         $train\_labels \leftarrow labels$[all indices except $i$]
9:         Train $model$ on $train\_data, train\_labels$
10:        $score \leftarrow model$.evaluate($test\_sample, test\_label$)
11:        Append $score$ to $evaluation\_scores$
12:    end for
13:    $average\_score \leftarrow$ average of $evaluation\_scores$
14:    return $average\_score$
15: end procedure
```

- Can be useful for very small datasets where splitting into multiple folds might leave too little data for training.

Disadvantages

- Computationally very expensive for large datasets (model trained n times).
- High variance in the performance estimate due to the very small test set size (single sample).
- The error estimate can be pessimistic if the true test error is influenced by the size of the training set.

Leave-P-Out Cross-Validation (LPOCV)

A generalization of LOOCV where p samples are held out as the test set in each iteration. While offering potentially lower bias, the number of iterations can be prohibitively large, making it impractical for most scenarios.
Advantages:

- Can provide a less biased estimate than K-Fold with a small k.

Disadvantages

- Computationally extremely expensive as the number of iterations is $\binom{n}{p}$, which grows rapidly with n and p.
- Can suffer from high variance in the performance estimate.
- Generally impractical for datasets of even moderate size.

Shuffle-Split Cross-Validation (or Random Permutation Cross-Validation)

The dataset is repeatedly and randomly split into training and test sets of predefined sizes. This allows for flexible control over the size of the training and testing sets and the number of iterations. It's useful for large datasets but doesn't guarantee that each data point will be in a test set the same number of times.
Advantages:

Algorithm 18 Leave-P-Out Cross-Validation (LPOCV)

```
1: procedure LPOCV($data, labels, p, model$)
2:     $n \leftarrow$ number of samples in $data$
3:     $indices \leftarrow$ all indices from 0 to $n - 1$
4:     $test\_indices\_combinations \leftarrow$ all combinations of $p$ indices from $indices$
5:     $evaluation\_scores \leftarrow$ empty list
6:     for $test\_indices$ in $test\_indices\_combinations$ do
7:         $train\_indices \leftarrow$ all indices in $indices \setminus test\_indices$
8:         $X\_train \leftarrow data[train\_indices]$
9:         $y\_train \leftarrow labels[train\_indices]$
10:        $X\_test \leftarrow data[test\_indices]$
11:        $y\_test \leftarrow labels[test\_indices]$
12:        Train $model$ on $X\_train, y\_train$
13:        $score \leftarrow model.\text{evaluate}(X\_test, y\_test)$
14:        Append $score$ to $evaluation\_scores$
15:    end for
16:    $average\_score \leftarrow$ average of $evaluation\_scores$
17:    return $average\_score$
18: end procedure
```

Algorithm 19 Shuffle-Split Cross-Validation

```
1: procedure ShuffleSplitCV($data, labels, n\_splits, test\_size, train\_size, random\_state, model$)
2:     $n \leftarrow$ number of samples in $data$
3:     $evaluation\_scores \leftarrow$ empty list
4:     Initialize random number generator with $random\_state$
5:     for $i = 1$ to $n\_splits$ do
6:         Randomly shuffle the indices of the data
7:         Determine the indices for the test set (size $test\_size$)
8:         Determine the indices for the training set (size $train\_size$)
9:         $X\_train \leftarrow data[\text{train indices}]$
10:        $y\_train \leftarrow labels[\text{train indices}]$
11:        $X\_test \leftarrow data[\text{test indices}]$
12:        $y\_test \leftarrow labels[\text{test indices}]$
13:        Train $model$ on $X\_train, y\_train$
14:        $score \leftarrow model.\text{evaluate}(X\_test, y\_test)$
15:        Append $score$ to $evaluation\_scores$
16:    end for
17:    $average\_score \leftarrow$ average of $evaluation\_scores$
18:    return $average\_score$
19: end procedure
```

- Offers flexibility in choosing the sizes of the training and test sets in each split.
- Allows for controlling the number of iterations, thus managing computational cost.
- Can be particularly useful for large datasets.
- Can generate non-overlapping train/test sets if desired.

Disadvantages

- Does not guarantee that each data point will be in a test set the same number of times.
- If the number of splits is not large enough, it might not provide as comprehensive an evaluation as K-Fold.

Time Series Cross-Validation

Specifically designed for time series data, this technique respects the temporal order of observations. Common approaches include expanding and rolling windows, where training data always precedes the test data in time, preventing unrealistic "future peeking" during evaluation.
Expanding Window Cross-Validation Algorithm:

Algorithm 20 Expanding Window Time Series Cross-Validation

```
procedure ExpandingWindowCV($time\_series\_data, n\_splits, model$)
    $n \leftarrow$ length of $time\_series\_data$
    $split\_size \leftarrow \lfloor n/(n\_splits + 1) \rfloor$
    $evaluation\_scores \leftarrow$ empty list
    for $i = 1$ to $n\_splits$ do
        $train\_end\_index \leftarrow i \times split\_size$
        $test\_start\_index \leftarrow train\_end\_index + 1$
        $test\_end\_index \leftarrow (i + 1) \times split\_size$
        if $test\_end\_index > n$ then
            $test\_end\_index \leftarrow n$
        end if
        $train\_data \leftarrow time\_series\_data[0 : train\_end\_index]$
        $test\_data \leftarrow time\_series\_data[test\_start\_index : test\_end\_index]$
        if $length(test\_data) > 0$ then
            Train $model$ on $train\_data$
            $predictions \leftarrow model$.predict($test\_data$)
            $score \leftarrow evaluate(test\_data, predictions)$    ▷ Use appropriate time series evaluation metric
            Append $score$ to $evaluation\_scores$
        end if
    end for
    $average\_score \leftarrow$ average of $evaluation\_scores$
    return $average\_score$
end procedure
```

Rolling Window Cross-Validation Algorithm:
Advantages (Time Series CV):

- Respects the temporal order of the data, preventing data leakage.

Algorithm 21 Rolling Window Time Series Cross-Validation

```
1: procedure RollingWindowCV($time\_series\_data, train\_window\_size, test\_window\_size, step\_size,$
2:     $n \leftarrow$ length of $time\_series\_data$
3:     $evaluation\_scores \leftarrow$ empty list
4:     $start\_index \leftarrow 0$
5:     while $start\_index + train\_window\_size + test\_window\_size \leq n$ do
6:         $train\_start \leftarrow start\_index$
7:         $train\_end \leftarrow start\_index + train\_window\_size$
8:         $test\_start \leftarrow train\_end$
9:         $test\_end \leftarrow train\_end + test\_window\_size$
10:        $train\_data \leftarrow time\_series\_data[train\_start : train\_end]$
11:        $test\_data \leftarrow time\_series\_data[test\_start : test\_end]$
12:        Train $model$ on $train\_data$
13:        $predictions \leftarrow model$.predict($test\_data$)
14:        $score \leftarrow evaluate(test\_data, predictions)$   ▷ Use appropriate time series evaluation metric
15:        Append $score$ to $evaluation\_scores$
16:        $start\_index \leftarrow start\_index + step\_size$
17:    end while
18:    $average\_score \leftarrow$ average of $evaluation\_scores$
19:    return $average\_score$
20: end procedure
```

- Provides a more realistic evaluation of how the model will perform on future, unseen data.
- Allows for assessing model stability over time.

Cons (Time Series CV):

- Can be more complex to implement correctly.
- The choice of window sizes and step size can significantly impact the evaluation.
- May require careful consideration of stationarity and other time series properties.

The selection of the appropriate cross-validation strategy is a crucial decision in the model selection process. It ensures a reliable and unbiased assessment of how well a model is likely to perform on new, unseen data, ultimately leading to the deployment of more robust and effective machine learning solutions. Understanding the nuances of each technique allows practitioners to tailor their evaluation process to the specific characteristics of their data and the goals of their modeling task.

4.6.2 Hyperparameter Optimization

Hyperparameter optimization, often referred to as hyperparameter tuning, is the critical process of selecting the optimal set of hyperparameters for a machine learning algorithm. Hyperparameters are the parameters of a learning algorithm that are set prior to the training process and control aspects of how the algorithm learns from the data. Finding the right combination of these settings is essential for achieving peak model performance, such as maximizing accuracy, minimizing error, or improving generalization to unseen data.
The space of possible hyperparameter values can be vast and complex, and the optimal settings often depend on the specific dataset and the learning task at hand. Manual tuning can be

time-consuming and inefficient, necessitating automated and intelligent search strategies. Let's delve deeper into the advanced techniques for hyperparameter optimization:

Grid Search with Cross-Validation

Grid search is a foundational and systematic method for hyperparameter tuning. It operates by defining a discrete set of possible values for each hyperparameter that one wishes to optimize. These sets of values form a multi-dimensional grid, and the grid search algorithm then exhaustively evaluates every possible combination of hyperparameter values from this grid.

For each combination of hyperparameters, a model is trained on a subset of the data and its performance is evaluated on a held-out validation set. To obtain a more robust estimate of the model's performance for each hyperparameter combination, grid search is typically coupled with cross-validation. Cross-validation involves splitting the training data into multiple folds, training the model on a subset of these folds, and evaluating it on the remaining fold. This process is repeated for each fold, and the performance metric (e.g., accuracy, F1-score) is averaged across the folds to provide a more reliable assessment of how well the model generalizes.

While grid search guarantees that every specified hyperparameter combination is evaluated, its primary drawback is its computational cost. The number of combinations to evaluate grows exponentially with the number of hyperparameters and the number of values specified for each. This "curse of dimensionality" can make grid search impractical for models with many hyperparameters or when the range of potentially optimal values for each hyperparameter is large.

Example: Consider tuning a Support Vector Machine (SVM) classifier, where we want to optimize the regularization parameter 'C', the kernel type, and the kernel coefficient 'gamma'.

Algorithm 22 Grid Search for SVM

```
1:  procedure SVMGridSearch($X, y$)
2:      $param\_grid \leftarrow \{$
3:          'C': [0.1, 1, 10, 100],                    ▷ Regularization parameter values
4:          'kernel': ['linear', 'rbf'],                ▷ Kernel types
5:          'gamma': [0.01, 0.1, 1]                    ▷ Kernel coefficient values for 'rbf'
6:      $\}$
7:      $svm \leftarrow$ SVC()                            ▷ Initialize the Support Vector Classifier
8:      $grid\_search \leftarrow$ GridSearchCV($svm, param\_grid, cv = 5$)    ▷ Initialize GridSearchCV
    with the SVM model, hyperparameter grid, and 5-fold cross-validation
9:      $grid\_search$.fit($X, y$)      ▷ Fit the grid search to the training data, evaluating all
    combinations
10:     return $grid\_search$.best_params_     ▷ Return the hyperparameter combination that
    yielded the best cross-validation score
11: end procedure
```

In this example, the `param_grid` defines the set of hyperparameter values to be explored. Grid search will train and evaluate an SVM model for each of the $4 \times 2 \times 3 = 24$ possible combinations of these hyperparameter values, using 5-fold cross-validation for each combination. The `best_params_` attribute of the fitted `grid_search` object will then provide the hyperparameter settings that achieved the highest average cross-validation score.

Random Search

Random search offers a more computationally efficient alternative to grid search, particularly when the hyperparameter space is high-dimensional and not all hyperparameters are equally influential on model performance. Instead of exhaustively trying all combinations in a predefined grid, random search samples a fixed number of hyperparameter combinations from specified probability distributions for each hyperparameter.
The key advantage of random search is that it explores a potentially wider and more diverse range of hyperparameter values within a given computational budget. In high-dimensional spaces, the optimal values for some hyperparameters might lie in regions that are sparsely covered by a grid search, whereas random sampling has a higher probability of hitting these important regions. Furthermore, random search can be more effective at identifying the impact of individual hyperparameters. If a hyperparameter does not significantly affect the performance, random search will effectively try many different values for it while focusing on the more critical ones.
The number of iterations (i.e., the number of randomly sampled hyperparameter combinations to evaluate) is a key parameter in random search. A larger number of iterations increases the likelihood of finding a good or optimal hyperparameter setting but also increases the computational cost. Similar to grid search, random search is typically used in conjunction with cross-validation to obtain reliable performance estimates for each sampled hyperparameter combination.

Example: Let's consider tuning a Random Forest classifier by optimizing the number of trees (*n_estimators*), the maximum depth of the trees (*max_depth*), the minimum number of samples required to split an internal node (*min_samples_split*), and the minimum number of samples required to be at a leaf node (*min_samples_leaf*). Some of these hyperparameters span a continuous or large range of integer values, which can make grid search inefficient.

Algorithm 23 Random Search for Random Forest

```
1: procedure RFRandomSearch($X, y$)
2:     $param\_distributions \leftarrow$ {
3:         'n_estimators': randint(10, 200),
4:         'max_depth': [None] + list(range(5, 30)),
5:         'min_samples_split': randint(2, 20),
6:         'min_samples_leaf': randint(1, 10)
7:     }
8:     $rf \leftarrow$ RandomForestClassifier()
9:     $random\_search \leftarrow$ RandomizedSearchCV( $rf$, $param\_distributions$, $n\_iter = 100$, $cv = 5$ )
10:    $random\_search$.fit($X, y$)
11:    return $random\_search$.best_params_
12: end procedure
```

In this example, *param_distributions* defines the probability distributions from which to sample hyperparameter values. For instance, *n_estimators* will be randomly chosen from a uniform distribution between 10 and 199 (inclusive), while *max_depth* will be randomly selected from either `None` or an integer between 5 and 29. The *n_iter* parameter specifies the number of random combinations to try. `RandomizedSearchCV` will perform 100 such random samplings, evaluate each using 5-fold cross-validation, and ultimately return the best set of hyperparameters found.

Chapter 5

Reinforcement Learning

5.1 Fundamentals of RL

Reinforcement Learning (RL) is a type of machine learning where an **agent** learns to make decisions by interacting with an **environment**. The core idea is that the agent receives a **reward** signal after taking an **action** in a particular **state**, and its goal is to learn a **policy** that maximizes the cumulative future rewards.

- **Agent**: The agent is the intelligent entity that performs actions and learns from the outcomes. It observes the environment, makes decisions based on its internal state (which often reflects the environment's state and its learned knowledge), and aims to achieve a long-term goal by maximizing cumulative reward. Think of it as the protagonist in the RL story.

  Example:

  - In a robot navigation task, the agent is the robot itself, deciding where to move.
  - In a financial trading system, the agent is the algorithm deciding whether to buy, sell, or hold a stock.
  - In a video game, the agent could be a character controlled by an AI trying to beat a level.

- **Environment**: The environment is the external system with which the agent interacts. It receives the agent's actions and transitions to a new state, subsequently emitting a reward signal to the agent. The environment defines the rules of interaction and the consequences of the agent's actions.

  Example:

  - For the robot navigation agent, the environment is the physical space, including walls, obstacles, and the target destination.
  - For the financial trading agent, the environment is the stock market, including price movements, trading volume, and news.
  - For the video game agent, the environment is the game world, including the level layout, enemies, power-ups, and game mechanics.

- **State**:A state provides a summary of the current situation in the environment relevant to the agent's decision-making. It's the information the agent uses to decide what action to take. The state must ideally be sufficient to predict the future without needing to know the entire history of interactions (this relates to the Markov Property). It is denoted by $s \in S$, where S is the set of all possible states.

Example:

- In a game of chess, the state is the arrangement of all pieces on the board.
- In a robot navigation task on a grid, the state could be the robot's (x, y) coordinates.
- In a self-driving car scenario, the state might include sensor data (camera, lidar), GPS location, speed, and the status of traffic signals.

- **Action**:An action is a specific move or decision made by the agent. Actions are the means by which the agent influences the environment, leading to state transitions and potential rewards. The set of available actions can vary depending on the current state. It is denoted by $a \in A$, where A is the set of all possible actions.**Example**:
  - In chess, moving a specific piece from one square to another is an action.
  - In the grid navigation task, possible actions might be 'move North', 'move South', 'move East', 'move West'.
  - For a robot arm, actions could be moving a joint by a certain angle or gripping an object.
- **Reward**:The reward is a scalar feedback signal that the environment sends to the agent after each time step or action. It is the primary signal indicating how well the agent is performing concerning its goal. The agent's objective is to learn a strategy to maximize the cumulative reward it receives over the long run. Rewards are typically numerical values.**Example**:
  - In chess, a positive reward could be received for capturing an opponent's piece or winning the game, while a negative reward (punishment) might be given for losing a piece or losing the game.
  - In robot navigation, a positive reward might be given for reaching the target, a small negative reward for each step taken (to encourage efficiency), and a large negative reward for colliding with an obstacle.
  - In financial trading, the reward could be the profit made from a trade.
- **Policy**:The policy, denoted by π, is the agent's strategy – it tells the agent what action to take in each state. It's the core component that the RL agent learns. The policy essentially maps states to probabilities of selecting each action. A policy can be:
  - **Deterministic**: For each state s, the policy specifies exactly one action $a = \pi(s)$.
  - **Stochastic**: For each state s, the policy specifies a probability distribution over actions, $\pi(a|s) = P(A_t = a|S_t = s)$. This means the agent chooses action a with probability $\pi(a|s)$ when in state s.

  Learning in RL typically involves finding an optimal policy π^* that maximizes the expected cumulative reward.

5.2 Markov Property

The **Markov Property** is a fundamental concept in stochastic processes, stating that the future is independent of the past given the present. In simpler terms, if a process has the Markov Property, knowing the current state provides all the information necessary to predict the probabilities of future states; the history leading up to the current state is irrelevant. This is often referred to as being ”memoryless”.
Mathematically, for a sequence of states $X_0, X_1, X_2, \ldots$, the Markov Property holds if for any time t and any states $s_0, s_1, \ldots, s_t, s'$,

$$P(X_{t+1} = s'|X_t = s_t, X_{t-1} = s_{t-1}, \ldots, X_0 = s_0) = P(X_{t+1} = s'|X_t = s_t)$$

This simplification is crucial in many mathematical models, including those used in RL.
In the context of Reinforcement Learning, the environment is said to be Markov if the next state s_{t+1} and the reward r_{t+1} depend only on the current state s_t and the action a_t taken, not on the sequence of states and actions that occurred before time t:

$$P(s_{(t+1)}, r_{(t+1)}|s_t, a_t, s_{(t-1)}, a_{(t-1)}, \ldots, s_0, a_0) = P(s_{(t+1)}, r_{(t+1)}|s_t, a_t)$$

If an environment satisfies the Markov Property, it is significantly easier for an agent to learn an optimal policy because the current state provides a sufficient statistic for decision-making.
The change in state from s_t to s_{t+1} is called a transition. The probability of this transition occurring, given the current state and action, is the **Transition Probability**, denoted by $P(s'|s, a) = \Pr(S_{t+1} = s'|S_t = s, A_t = a)$.
Transition Matrix (P): For processes with a finite number of states, the transition probabilities can be organized into a matrix. In a Markov Chain (where transitions are state-dependent but not action-dependent), the matrix P has dimensions $|S| \times |S|$, where $|S|$ is the number of states. The element P_{ij} is the probability of transitioning from state s_i to state s_j:

$$P = \begin{pmatrix} P_{11} & P_{12} & \cdots & P_{1|S|} \\ P_{21} & P_{22} & \cdots & P_{2|S|} \\ \vdots & \vdots & \ddots & \vdots \\ P_{|S|1} & P_{|S|2} & \cdots & P_{|S||S|} \end{pmatrix}.$$

Each row i sums to 1, as $\sum_{j=1}^{|S|} P_{ij} = 1$.
In an MDP, where transitions depend on actions, we have a separate transition matrix (or function) for each action $a \in A$. For a given action a, the transition probability from s to s' is $P(s'|s, a)$.
Initial Distribution: This is a probability distribution over the state space S at the beginning of a process (time $t = 0$). It specifies the probability of starting in each possible state. For example, if you're modeling a game, the initial distribution might place a probability of 1 on the ”start of game” state and 0 on all other states.

5.3 Markov Chain

A **Markov Chain** is a stochastic process that satisfies the Markov Property. It describes a sequence of states where the probability of the next state depends only on the current state. Markov Chains are used to model systems that transition between states probabilistically over time, without an external agent influencing these transitions.
Example: Consider a simple weather model with two states: ”Sunny” (S) and ”Rainy” (R). A Markov Chain could model the weather transitions if the probability of tomorrow's weather depends only on today's weather. Suppose the transition probabilities are:

- P(Sunny tomorrow|Sunny today) = 0.9
- P(Rainy tomorrow|Sunny today) = 0.1
- P(Sunny tomorrow|Rainy today) = 0.3
- P(Rainy tomorrow|Rainy today) = 0.7

The transition matrix would be:

$$P = \begin{pmatrix} 0.9 & 0.1 \\ 0.3 & 0.7 \end{pmatrix}$$

where the first row is for "Sunny today" and the second row for "Rainy today", and columns are "Sunny tomorrow" and "Rainy tomorrow". This chain evolves over time based solely on the current weather state.

5.4 The Markov Reward Process (MRP)

A **Markov Reward Process (MRP)** is a Markov Chain augmented with a reward function. It models a system that transitions between states according to Markovian probabilities and provides a numerical reward when a transition occurs or when entering a state. An MRP adds value to the states in a Markov Chain.
An MRP is defined by a tuple (S, P, R, γ), where:

- S: A finite set of states.
- P: A state transition probability matrix, $P(s'|s)$.
- R: A reward function. $R(s)$ is the expected reward received when transitioning out of state s, or $R(s, s')$ is the reward received during the transition from s to s'. Often, $R(s)$ is defined as the expected reward given the current state s, averaging over possible next states and their associated rewards.
- γ: A discount factor, $0 \leq \gamma < 1$. This value determines the present value of future rewards. Rewards received in the distant future are less valuable than immediate rewards.

In an MRP, there is no agent and no actions; the process simply unfolds according to the transition probabilities, yielding rewards along the way. A key concept in MRPs is the **Value Function**. The value of a state s in an MRP, $V(s)$, is the expected total discounted future reward starting from state s. This can be calculated using the Bellman equation for MRPs.
Example: Taking the weather Markov Chain example, we can add rewards. Suppose being in a "Sunny" state gives a reward of +1 and being in a "Rainy" state gives a reward of -0.5 (maybe you prefer sunny days). $R(\text{Sunny}) = +1$ $R(\text{Rainy}) = -0.5$ With a discount factor $\gamma = 0.9$, we can calculate the value of starting on a sunny day or a rainy day by considering the expected sequence of states and rewards. The MRP framework allows us to formalize this calculation.

5.5 The Markov Decision Process (MDP)

The **Markov Decision Process (MDP)** is the standard mathematical framework for modeling sequential decision-making problems in Reinforcement Learning. It extends the MRP by introducing an agent and actions. In an MDP, the transitions and rewards depend not only on the current state but also on the action taken by the agent. The agent's goal is to choose actions that maximize the cumulative discounted reward over time.
An MDP is formally defined by the tuple (S, A, P, R, γ), where:

1. **States (S)**:A finite (or countably infinite) set of states. As described in the fundamentals, this represents the possible situations the agent can be in.**Example**: States in a simple grid world might be the coordinates (x, y) on the grid.

2. **Actions (A)**:A finite (or countably infinite) set of actions available to the agent. The set of available actions might be the same for all states, or it might vary depending on the current state, denoted as $A(s)$.**Example**: In the grid world, actions could be {'North', 'South', 'East', 'West'}.

3. **Transition Probability (P)**:The transition function describes the dynamics of the environment. For each state $s \in S$ and action $a \in A$, $P(s' \mid s, a)$ gives the probability of transitioning to state s' when taking action a in state s. This is where the Markov Property is essential: the next state depends only on the current state and action. The environment can be stochastic, meaning that taking the same action in the same state might lead to different next states with certain probabilities.

$$P(s' \mid s, a) = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a)$$

Example: In the grid world, taking action 'North' from state (x, y) might usually lead to $(x, y + 1)$. However, if the floor is slippery, there might be a 0.8 probability of moving North, a 0.1 probability of slipping East to $(x + 1, y)$, and a 0.1 probability of slipping West to $(x-1, y)$. So, $P((x, y+1)|(x, y), \text{'North'}) = 0.8$, $P((x+1, y)|(x, y), \text{'North'}) = 0.1$, and $P((x-1, y)|(x, y), \text{'North'}) = 0.1$. The sum of probabilities for all possible next states from (x, y) with action 'North' must be 1.

4. **Reward Function (R)**:The reward function specifies the immediate reward the agent receives. This reward can depend on the current state s, the action taken a, and the resulting next state s'. The most general form is $R(s, a, s')$. However, it is often simplified:

   - $R(s, a)$: The expected reward depends only on the state and action, $R(s, a) = \sum_{s'} P(s'|s, a)R(s, a, s')$.
   - $R(s')$: The reward depends only on the next state.
   - $R(s)$: The reward depends only on the current state (less common in action-dependent MDPs).

   The reward function is crucial because it defines the goal of the RL problem. The agent learns to maximize the expected cumulative reward signal specified by R.**Example**: In the grid world:

   - Reaching the goal state might give a large positive reward (+100).
   - Falling into a pit might give a large negative reward (-100).
   - Taking any step might give a small negative reward (-1) to encourage finding shorter paths.
   - In this case, the reward could be defined as $R(s, a, s')$, depending on the transition. For instance, $R((x, y), \text{'North'}, (x, y + 1)) = -1$, unless $(x, y + 1)$ is the goal state, where it would be $+100$.

5. **Discount Factor (γ)**: The discount factor $\gamma \in [0, 1)$ is a parameter that discounts future rewards. A reward received k steps in the future is worth γ^k times a reward received immediately. Discounting ensures that the sum of future rewards converges in infinite-horizon problems and allows the agent to prioritize more immediate rewards.

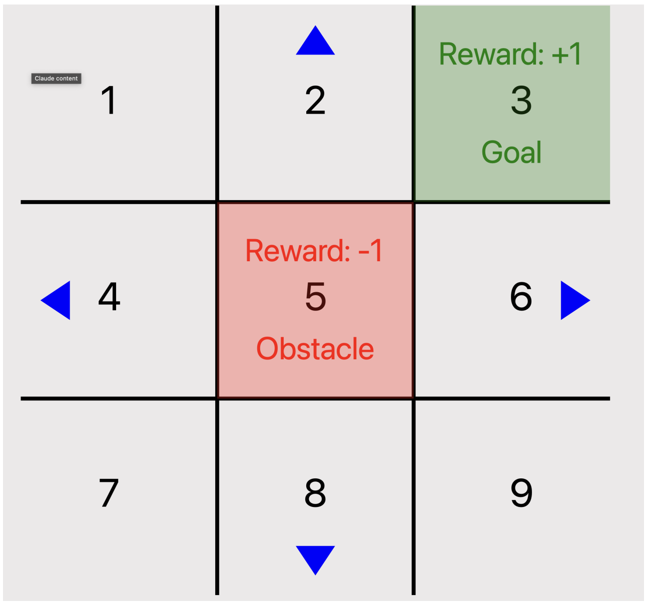


Figure 5.1: 3x3 grid world

- If $\gamma = 0$, the agent is myopic and only considers the immediate reward.
- If γ is close to 1 (e.g., 0.99), the agent is farsighted and considers future rewards almost as much as immediate ones.

The choice of γ is important and depends on the problem. For episodic tasks (tasks that end), γ can sometimes be 1, but for continuing tasks (tasks that go on forever), $\gamma < 1$ is necessary for the value calculations to converge.

In Reinforcement Learning, the agent exists within this MDP framework. At each time step, the agent observes its current state s_t, chooses an action a_t based on its policy π, and the environment transitions to a new state s_{t+1} and provides a reward r_{t+1} according to the MDP's dynamics (P and R). The agent's goal is to learn or find the optimal policy π^* that maximizes the expected cumulative discounted reward, known as the expected return $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. The Markov property of the states is what makes dynamic programming and many RL algorithms applicable and tractable for finding this optimal policy.

5.5.1 Grid World Example

Let's consider a simple grid world where an agent moves in 3×3 grid.
An agent can start at any non-goal state in the grid world. 3 represents the goal state and 5 is the obstacle state. The agent can move in one of four directions: up, down, left, or right, as represented by blue arrows.

5.5.2 Action Space

The **action space**, denoted by A, refers to the set of all possible actions an agent can take while interacting with the environment. The nature of the action space significantly influences the choice of reinforcement learning algorithms. Action spaces can be broadly categorized into discrete and continuous.

1. Discrete Action Space

In a **discrete action space**, the set of possible actions available to the agent is finite and countable. Each action is a distinct, separate choice.
Example: In a simple grid environment, an agent might only be able to move in four cardinal directions.

- **Up**: Move one cell up.
- **Down**: Move one cell down.
- **Left**: Move one cell left.
- **Right**: Move one cell right.

In this case, the action space A is a finite set:

$$A = \{\text{Up}, \text{Down}, \text{Left}, \text{Right}\}$$

For computational purposes, these actions are often mapped to integers, e.g., Up=0, Down=1, Left=2, Right=3. Many classic RL problems, such as board games (Chess, Go) or simple navigation tasks, feature discrete action spaces.

2. Continuous Action Space

In a **continuous action space**, the set of possible actions is infinite and continuous. Actions are represented by real-valued vectors or scalars within a certain range. This is common in tasks requiring fine-grained control.
Example: Consider a robotic arm. The agent might control the torque applied to each joint, or the velocity of an end-effector.

- Controlling joint torques: The action could be a vector $(\tau_1, \tau_2, \ldots, \tau_n)$, where τ_i is the torque applied to the i-th joint, and each τ_i can be any real number within a physical limit, e.g., $\tau_i \in [\tau_{i,\min}, \tau_{i,\max}]$.
- Controlling velocity: The action could be a 2D or 3D velocity vector (v_x, v_y) or (v_x, v_y, v_z), where each component can take any real value within a range. For instance, a car's steering angle and acceleration are continuous actions.

In a simple 1D continuous action space, A could be an interval on the real line:

$$A = [a_{\min}, a_{\max}]$$

where $a_{\min}$ and $a_{\max}$ define the lower and upper bounds of the action value. More generally, it could be a multi-dimensional space like $A \subset \mathbb{R}^n$. Problems in robotics, autonomous driving, and control systems often involve continuous action spaces.

5.5.3 Policy

A **policy**, denoted by π, is the agent's strategy. It defines how the agent chooses an action based on the current state. In essence, the policy is the brain of the agent, determining its behavior. The goal of Reinforcement Learning is typically to find an optimal policy that maximizes the expected cumulative reward.
Initially, an agent might start with a simple policy, such as a **random policy**. A random policy selects any available action in a given state with equal probability, without any knowledge

of the environment's dynamics or rewards. For example, in the grid world, a random policy would choose to move Up, Down, Left, or Right with 25% probability each (or with probability summing to 1 over valid actions from the current state).
Through interaction with the environment, receiving rewards and observing next states, the agent learns which actions are more beneficial in which states. The learning process involves updating the policy iteratively to favor actions that lead to higher cumulative rewards.

Optimal Policy

The **optimal policy**, denoted by π^*, is the policy that yields the greatest possible expected cumulative reward (return) from every state compared to any other policy. Finding the optimal policy is the primary objective in many RL problems.
The optimal policy dictates the best action to take in each state to achieve the agent's goal as efficiently and effectively as possible according to the defined reward function.
Consider a 3x3 grid world where the agent starts at (1,1), a goal is at (3,3) (let's use (row, col) or state numbers for clarity, say states 1-9 reading left-to-right, top-to-bottom), and there's an obstacle at (2,2) (state 5). An optimal policy would map each state to an action that moves the agent towards state 9 (the goal) while avoiding state 5 (the obstacle). If states are numbered 1 to 9:

- State 1 (top-left): Move Right
- State 2: Move Right
- State 3 (top-right): Move Down (or another action leading towards goal, depends on full grid layout)
- State 4 (middle-left): Move Down
- State 5 (center, obstacle): (No action, or any action leading out if possible) - agent should ideally not enter this state under optimal policy
- State 6 (middle-right): Move Down
- State 7 (bottom-left): Move Right
- State 8 (bottom-middle): Move Right
- State 9 (bottom-right, goal): Terminal state (no action needed)

This is a simplified view; the exact optimal actions depend on the transition probabilities (e.g., is movement deterministic?) and the reward structure (e.g., are there step costs?).
Policies can be categorized based on whether they always choose the same action for a state or sample from a distribution.

Deterministic Policy

A **deterministic policy** specifies a single, unique action for each state. Given a state s, a deterministic policy μ will always output the same action a. There is no randomness in the action selection process once the state is known.
Formally, a deterministic policy μ is a function that maps states to actions:

$$\mu : S \to A \tag{5.1}$$

So, for any state $s \in S$, the action taken is deterministically given by $a = \mu(s)$.

Example: In the grid world, a deterministic policy might be: "If in state (1,1), always move Right"; "If in state (1,2), always move Down". The optimal policy example shown above is a deterministic policy.

Stochastic Policy

A **stochastic policy**, on the other hand, specifies a probability distribution over the actions for each state. When in a state s, a stochastic policy π provides the probability $\pi(a|s)$ of taking each possible action a. The agent then samples an action according to this distribution. This introduces randomness into the agent's behavior.
Formally, a stochastic policy π is a function that maps states to probability distributions over actions:

$$\pi : S \to \mathcal{P}(A) \tag{5.2}$$

where $\mathcal{P}(A)$ is the set of probability distributions over the action space A. The probability of taking action a in state s is denoted by $\pi(a \mid s) = \Pr(A_t = a \mid S_t = s)$. For discrete action spaces, the sum of probabilities for all actions in a given state must equal 1: $\sum_{a \in A} \pi(a \mid s) = 1$ for all $s \in S$.
Example: In the grid world, a stochastic policy for state (1,1) might be: move Right with probability 0.8, move Down with probability 0.1, and move Up or Left with probability 0.05 each (assuming Up/Left are valid actions from (1,1) or handling boundary conditions).
Stochastic policies are particularly useful in environments with partial observability, where the agent doesn't have complete information about the state, or in multi-agent environments to make the agent's behavior unpredictable. They are also crucial in many policy gradient algorithms.
Stochastic policies are implemented differently depending on whether the action space is discrete or continuous.

Categorical Policy

A **Categorical policy** is used when the action space is **discrete**. For each state s, the policy outputs a probability for each action a in the finite action set A. These probabilities form a categorical distribution over the actions.
Formally, for a state s, the policy π outputs a vector of probabilities $(\pi(a_1 \mid s), \pi(a_2 \mid s), \ldots, \pi(a_{|A|} \mid s))$, where $|A|$ is the number of discrete actions.

$$\pi(a \mid s) \geq 0 \quad \text{for all } a \in A$$

$$\sum_{a \in A} \pi(a \mid s) = 1$$

The agent then samples an action a from this distribution.
Example: In the grid world with actions {Up, Down, Left, Right}, the categorical policy for state (1,1) might output probabilities 0.1, 0.6, 0.15, 0.15 for (Up, Down, Left, Right) respectively. When in state (1,1), the agent would choose Down 60% of the time, Left 15%, Right 15%, and Up 10%.

Gaussian Policy

A **Gaussian policy** is used when the action space is **continuous**. For each state s, the policy typically outputs the parameters (mean and standard deviation) of a Gaussian (Normal) distribution for the continuous action(s). The agent then samples an action from this distribution.

Formally, for a state s, the policy outputs a mean vector $\mu(s)$ and often a standard deviation vector $\sigma(s)$ (or a covariance matrix). The action a is sampled from the distribution $\mathcal{N}(\mu(s), \Sigma(s))$, where $\Sigma(s)$ is the covariance matrix (often simplified to a diagonal matrix where the diagonal elements are the variances $\sigma_i(s)^2$).

$$a \sim \mathcal{N}(\mu(s), \sigma^2(s)) \quad \text{(for a 1D action space)}$$

The probability density function for a 1D action a in state s is:

$$p(a \mid s) = \frac{1}{\sqrt{2\pi\sigma^2(s)}} \exp\left(-\frac{(a-\mu(s))^2}{2\sigma^2(s)}\right).$$

The policy network in deep RL typically learns to output $\mu(s)$ and $\sigma(s)$ (or parameters from which $\sigma(s)$ is derived) for any given state s.
Example: In the robotic arm control task (1 joint, continuous torque action), a Gaussian policy for a specific state (e.g., arm at a certain angle and velocity) might output a mean torque of 1.5 Nm and a standard deviation of 0.3 Nm. The agent would then sample a torque value from a normal distribution centered at 1.5 with a standard deviation of 0.3 to apply to the joint.
In summary, the action space defines *what* the agent can do, while the policy defines *how* the agent chooses *which* action to do in any given state. The type of action space dictates the appropriate type of stochastic policy (Categorical for discrete, Gaussian for continuous).

5.5.4 Episode and Trajectory

Reinforcement Learning tasks can often be broken down into sequences of interactions. An **Episode** represents one complete sequence of interaction between the agent and the environment, starting from an initial state and ending in a terminal state. Episodic tasks are those that naturally break down into episodes, like a game that ends.
An episode typically unfolds as follows:

- **Start**: The environment is initialized, placing the agent in a specific initial state s_0. This initial state may be fixed or sampled from an initial state distribution.
- **Interaction Loop**: At each time step $t = 0, 1, 2, \ldots$, the agent observes the current state s_t, selects an action a_t according to its policy $\pi(a_t|s_t)$, the environment transitions to a new state s_{t+1} based on $P(s_{t+1}|s_t, a_t)$, and the agent receives a reward r_{t+1}.
- **End**: The episode terminates when a specific condition is met, usually reaching a terminal state. A terminal state could be a goal state, a failure state, or simply reaching a maximum number of steps.

Example: In the grid world environment:

- An episode starts with the agent placed at the 'Start' cell.
- The agent chooses moves (Up, Down, Left, Right), receives rewards (e.g., -1 for each step, +100 for reaching goal, -100 for falling into a pit).
- The episode ends when the agent reaches the 'Goal' cell or falls into a 'Pit' cell, or if a step limit is reached.

A **Trajectory** (also called a **Rollout** or **Sample Path**) is the specific sequence of states, actions, and rewards experienced by the agent during a single episode. It is a chronological

record of the agent's interaction with the environment from the beginning to the end of an episode.
Formally, a trajectory τ is represented as the sequence:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T), \tag{5.3}$$

where:

- s_t is the state at time step t.
- a_t is the action taken in state s_t at time step t.
- r_{t+1} is the reward received after taking action a_t and transitioning to state s_{t+1}.
- s_T is the terminal state.
- T is the total number of steps in the episode (the length of the trajectory, often referring to the number of rewards/transitions).

Each time an agent plays through an episode, it generates a unique trajectory.
Example: In the grid world, a trajectory could be: (Start (1,1)), Action: Right, Reward: -1, State: (1,2), Action: Right, Reward: -1, State: (1,3), Action: Down, Reward: -1, State: (2,3), Action: Down, Reward: -1, State: (3,3) (Goal), Reward: +100, State: (3,3) (Terminal). The full trajectory sequence would be: $(s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4, r_5, s_5)$. Note that the last state s_T is often included, but no action or reward follows it within that episode.

5.5.5 Return

The **Return** G_t is the total accumulated reward from time step t onwards within a trajectory. It is a fundamental concept in RL as the agent's objective is to maximize its expected return. The definition of return can vary slightly depending on whether the task is episodic or continuing, and the presence of a discount factor.
For tasks that continue indefinitely (continuing tasks), the concept of total undiscounted reward can lead to infinite values. Therefore, a **discount factor** γ is typically used to weigh future rewards, making them less valuable than immediate rewards.
The standard definition of the discounted return G_t from time step t is the sum of future rewards, discounted by γ:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{5.4}$$

where r_{t+k+1} is the reward received $k+1$ steps after time t (i.e., at time $t+k+1$), and γ is the discount factor ($0 \le \gamma < 1$). The requirement $\gamma < 1$ ensures that the sum converges for continuing tasks.
For **episodic tasks**, which end in a terminal state, the sum is finite. If the episode ends at time step T, the return from time $t < T$ is:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}. \tag{5.5}$$

In episodic tasks, the discount factor γ can sometimes be set to 1 (i.e., $0 \le \gamma \le 1$), as the finite sum will converge even without discounting. In this case, $G_t = \sum_{k=t+1}^{T} r_k$ (using r_k as reward at step k). This is often referred to as the **cumulative return** or **undiscounted return**.
RL algorithms aim to learn a policy that maximizes the expected return from each state. Methods like Monte Carlo estimate the expected return by averaging the *sample returns* (G_t calculated from actual trajectories) collected from multiple episodes.

5.5.6 A Discount Factor

We have already discussed a little about discount factor and discounted return. Let's get into detail of it now.
The discount factor γ is a value between 0 and 1 (i.e., $0 \leq \gamma \leq 1$) that determines the present value of future rewards. It is used to calculate the return, which is the sum of rewards an agent receives from a given time step onward, adjusted for the passage of time.
Formally, the return G_t from time step t is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \tag{5.6}$$

where:

- r_{t+k} is the reward received at time step $t + k$,
- γ is the discount factor.

The discount factor is crucial parameter in RL that influences how future rewards are valued compared to immediate rewards. It plays a significant role in the formulation of the return and the value functions, guiding the agent's learning process. The value of discount factor ranges from 0 to 1.

- **Immediate Rewards**: When γ is close to 1, the agent values future rewards almost as much as immediate rewards. This leads to a long-term perspective, where the agent considers the cumulative reward over a longer horizon.
- **Future Rewards Discounting**: When γ is less than 1, future rewards are discounted. This means that rewards received further in the future are valued less than immediate rewards. A lower γ leads to a more short-sighted perspective, focusing on immediate gains rather than long-term benefits.

Pen & Paper

The choice of γ affects the agent's behavior and the learning process:

- **High γ (close to 1)**: Encourages the agent to consider long-term rewards. This is useful in environments where long-term planning is crucial. However, it can also make learning slower because the agent has to consider a larger number of future rewards. Consider a reward sequence where the rewards received at each time step are r_0, r_1, r_2, and so on. The return G_t starting from time step t can be expressed as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{5.7}$$

For a discount factor $\gamma = 0.9$, the return from time step $t = 0$ is:

$$G_0 = \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots$$

Substituting $\gamma = 0.9$:

$$G_0 = (0.9)^0 r_0 + (0.9)^1 r_1 + (0.9)^2 r_2 + (0.9)^3 r_3 + \cdots$$

$$G_0 = 1 \cdot r_0 + 0.9 \cdot r_1 + (0.9)^2 \cdot r_2 + (0.9)^3 \cdot r_3 + \cdots$$

$$G_0 = r_0 + 0.9r_1 + 0.81r_2 + 0.729r_3 + \cdots$$

From the equation above, we can observe how the discount factor $\gamma = 0.9$ affects the weighting of rewards:

- **At Time Step 0**: The reward r_0 is weighted by a discount factor of 1 (i.e., no discounting). The immediate reward is valued fully.
- **At Time Step 1**: The reward r_1 is weighted by a discount factor of 0.9. This indicates that the immediate reward is valued highly, but future rewards are still considered important.
- **At Time Step 2**: The reward r_2 is weighted by a discount factor of 0.81 (i.e., $(0.9)^2$). This shows that while the reward is less valued compared to immediate rewards, it is still relatively significant.

Implications

As observed from the above example:

- The weight assigned to rewards decreases exponentially with the discount factor γ, but at a slower rate compared to smaller discount factors.
- A large discount factor means that rewards received in the future are valued almost as much as immediate rewards.
- By setting γ to a large value, such as 0.9, we emphasize long-term rewards more, leading the agent to consider both immediate and future rewards when making decisions.

• **Low γ (close to 0)**: Makes the agent more focused on immediate rewards. This can simplify learning and make it faster, but it may result in suboptimal policies if the environment requires consideration of future consequences.
Consider a reward sequence where the rewards received at each time step are r_0, r_1, r_2, and so on. The return G_t starting from time step t can be expressed as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

For a discount factor $\gamma = 0.2$, the return from time step $t = 0$ is:

$$G_0 = \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots$$

Substituting $\gamma = 0.2$:

$$G_0 = (0.2)^0 r_0 + (0.2)^1 r_1 + (0.2)^2 r_2 + (0.2)^3 r_3 + \cdots$$

$$G_0 = 1 \cdot r_0 + 0.2 \cdot r_1 + (0.2)^2 \cdot r_2 + (0.2)^3 \cdot r_3 + \cdots$$

$$G_0 = r_0 + 0.2r_1 + 0.04r_2 + 0.008r_3 + \cdots$$

From the equation above, we can observe how the discount factor $\gamma = 0.2$ affects the weighting of rewards:

- **At Time Step 0**: The reward r_0 is weighted by a discount factor of 1 (i.e., no discounting). This means that the immediate reward is valued fully.

  - **At Time Step 1**: The reward r_1 is weighted by a discount factor of 0.2. This shows that the immediate reward is given significantly more importance than the reward at the next time step.
  - **At Time Step 2**: The reward r_2 is weighted by a discount factor of 0.04 (i.e., $(0.2)^2$). The importance of this reward is much less compared to the immediate reward.

  We can observe from the above that:

  - The weight assigned to rewards decreases exponentially with the discount factor γ.
  - A small discount factor means that rewards received further in the future are valued much less compared to immediate rewards.
  - By setting γ to a small value, such as 0.2, we emphasize immediate rewards over long-term rewards, leading the agent to prioritize short-term gains over long-term benefits.

  This behavior is useful in environments where immediate rewards are crucial, or when quick decision-making is required. However, in scenarios where long-term planning is important, a higher discount factor would be more appropriate.

- $\gamma = 0$: Consider a reward sequence where the rewards received at each time step are r_0, r_1, r_2, and so on. The return G_t starting from time step t can be expressed as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

  For a discount factor $\gamma = 0$, the return from time step $t = 0$ is:

$$G_0 = \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots$$

  Substituting $\gamma = 0$:

$$G_0 = (0)^0 r_0 + (0)^1 r_1 + (0)^2 r_2 + (0)^3 r_3 + \cdots$$

$$G_0 = 1 \cdot r_0 + 0 \cdot r_1 + 0 \cdot r_2 + 0 \cdot r_3 + \cdots$$

$$G_0 = r_0$$

  From the equation above, we can observe how the discount factor $\gamma = 0$ affects the weighting of rewards:

  - **At Time Step 0**: The reward r_0 is weighted by a discount factor of 1. This means that the immediate reward is fully valued.
  - **At Subsequent Time Steps**: The rewards r_1, r_2, r_3, and so on are all weighted by a discount factor of 0. Consequently, these future rewards do not contribute to the return.

  This shows us that when discount factor is set to 0, we only consider the immediate reward r_0 and not the reward obtained from the future time steps. Setting γ to 0 results in the following:

- **Immediate Reward Focus**: Only the reward received at the current time step r_0 is considered. All future rewards are disregarded, meaning the agent is only concerned with immediate gains.
- **Short-Sighted Behavior**: The agent's decisions are based solely on immediate rewards, without any consideration for future consequences. This can lead to myopic strategies where long-term benefits are ignored.
- **Simplified Learning**: This setting simplifies the learning problem by reducing it to considering only the immediate reward. However, this might be unsuitable for environments where long-term planning is important.

5.6 Value Functions

In Reinforcement Learning, **Value Functions** are essential tools for evaluating states and actions. They estimate the expected cumulative future reward (Return) an agent can receive by starting from a particular state or taking a particular action in a state and then following a specific policy. The value function helps the agent understand the long-term consequences of being in a certain situation or making a certain decision, guiding the learning process towards maximizing the total reward.
There are two primary types of value functions, both defined with respect to a given policy π:

5.6.1 State-Value Function (V^π)

The **state-value function**, denoted by $V^\pi(s)$, represents the expected return when starting from state $s \in S$ and following policy π for all subsequent time steps. It quantifies how good it is to be in state s if the agent acts according to policy π.
Since the sequence of states, actions, and rewards that follow state s can be random (due to a stochastic environment or a stochastic policy), we consider the *expected* return. The return G_t from time step t is the total discounted future reward: $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$, where r_{t+k+1} is the reward received at step $t+k+1$ and γ is the discount factor.
Formally, the state-value function is defined as:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s\right], \quad \forall s \in S \tag{5.8}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value taken over all possible random trajectories starting from state s and following policy π.
Example Scenario: Grid World Let's use a simple grid world example to illustrate the value function. Imagine an agent moving on a grid, receiving rewards for visiting certain cells.

Deterministic Policy Example for V^π

Consider a scenario where both the environment transitions are deterministic (an action always leads to the same next state) and the policy π is deterministic (each state maps to exactly one action). In this case, starting from state s and following π always results in the exact same unique trajectory. The expected return $V^\pi(s)$ is simply the return of this single, deterministic trajectory.

Suppose, starting from state A and following a deterministic policy, the agent traces a path through states B, C, D, E and receives the following sequence of rewards:

$$A \xrightarrow{\text{action}} B \xrightarrow{\text{action}} C \xrightarrow{\text{action}} D \xrightarrow{\text{action}} E \xrightarrow{\text{action}} \text{End}$$

Rewards received after each action: $r_{AB} = +1$, $r_{BC} = +1$, $r_{CD} = -1$, $r_{DE} = +1$, $r_{E,\text{End}} = +1$. Let's assume a discount factor $\gamma = 1$ for calculating the return in this simple example. The return starting from state A is the sum of all subsequent rewards:

$$G_{\text{from A}} = r_{AB} + r_{BC} + r_{CD} + r_{DE} + r_{E,\text{End}} = +1 + 1 - 1 + 1 + 1 = 3$$

The return starting from state D is the sum of rewards from D onwards:

$$G_{\text{from D}} = r_{DE} + r_{E,\text{End}} = +1 + 1 = 2$$

The return starting from state E is the sum of rewards from E onwards:

$$G_{\text{from E}} = r_{E,\text{End}} = +1$$

The value of a terminal state is typically 0, as no more rewards are received. In this deterministic case, the state-value function $V^\pi(s)$ is equal to the return calculated for the unique trajectory starting from s:

$$V^\pi(A) = 3$$

$$V^\pi(D) = 2$$

$$V^\pi(E) = 1$$

$$V^\pi(\text{Terminal State}) = 0$$

Stochastic Policy Example for V^π

In more realistic scenarios, either the environment or the policy (or both) are stochastic. This means that starting from state s and following policy π can lead to different possible trajectories, each occurring with a certain probability and yielding a different return. The state-value function $V^\pi(s)$ is the *expected* value over the returns of all these possible trajectories, weighted by their probability of occurring.
Suppose our policy π for state A is stochastic:

$$\pi(\text{down}|A) = 0.8, \quad \pi(\text{right}|A) = 0.2$$

For other states, let's assume the policy and environment transitions are deterministic for simplicity in illustrating the initial stochasticity. Assume $\gamma = 1$.

- If the agent chooses 'down' from state A (with probability 0.8 according to π), suppose this leads to a subsequent deterministic path (A $\rightarrow$ D $\rightarrow$ E $\rightarrow$ End) with a return of 3 (as calculated in the deterministic example).

- If the agent chooses 'right' from state A (with probability 0.2 according to π), suppose this leads to a different subsequent deterministic path (A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ End) with associated rewards $r_{AB} = -1$, $r_{BC} = +1$, $r_{C,\text{End}} = +1$. The return for this path would be $-1 + 1 + 1 = 1$.

The state-value $V^\pi(A)$ is the expected return, averaging over the initial actions chosen by the policy:

$V^\pi(A) = P(\text{choose down from A}) \times (\text{Expected Return if starting with down}) + P(\text{choose right from A}) \times (\text{E}$

In this example, where the subsequent paths are deterministic:

$$V^\pi(A) = \pi(\text{down}|A) \times (\text{Return of A} \xrightarrow{\text{down}} \text{path}) + \pi(\text{right}|A) \times (\text{Return of A} \xrightarrow{\text{right}} \text{path})$$

$$V^\pi(A) = 0.8 \times 3 + 0.2 \times 1 = 2.4 + 0.2 = 2.6$$

This shows that $V^\pi(s)$ considers all possible sequences of events from state s under policy π, weighted by their probabilities.

5.6.2 Action-Value Function (Q^π)

The **action-value function**, denoted by $Q^\pi(s,a)$, represents the expected return when starting from state $s \in S$, taking a specific action $a \in A$ at that moment, and then following policy π for all subsequent steps. It answers the question: "How good is it to take action a in state s if I behave according to policy π afterwards?" $Q^\pi(s,a)$ is widely known as the Q-function.
Formally, $Q^\pi(s,a)$ is defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s, A_t = a\right], \quad \forall s \in S, a \in A \tag{5.9}$$

The expectation here is taken over the environment's probabilistic transitions resulting from taking action a in state s and the subsequent trajectory generated by following policy π.
Example for Q-Function: Using the same grid world and policy π (with $\gamma = 1$ and deterministic environment after the first step from A, as in the previous example):

- To compute $Q^\pi(A, \text{down})$, we evaluate the expected return *given* the agent starts in A and takes the action 'down'. Assuming the subsequent path is deterministic (A $\xrightarrow{\text{down}}$ D $\rightarrow$ E $\rightarrow$ End), the return is 3.

$$Q^\pi(A, \text{down}) = \text{Expected Return starting with (A, down)} = 3$$

- To compute $Q^\pi(A, \text{right})$, we evaluate the expected return *given* the agent starts in A and takes the action 'right'. Assuming the subsequent path is deterministic (A $\xrightarrow{\text{right}}$ B $\rightarrow$ C $\rightarrow$ End), the return is 1.

$$Q^\pi(A, \text{right}) = \text{Expected Return starting with (A, right)} = 1$$

- To compute $Q^\pi(D, \text{right})$, we evaluate the expected return *given* the agent starts in D and takes the action 'right'. Assuming the subsequent path is deterministic (D $\xrightarrow{\text{right}}$ E $\rightarrow$ End), the return is 2 (rewards +1, +1 from D).

$$Q^\pi(D, \text{right}) = \text{Expected Return starting with (D, right)} = 2$$

The state-value function $V^\pi(s)$ and action-value function $Q^\pi(s,a)$ are closely related. The value of a state $V^\pi(s)$ is the expected value of the action-values from that state, where the expectation is taken over the actions chosen by the policy π:

$$V^\pi(s) = \sum_{a \in A} \pi(a \mid s) Q^\pi(s,a) = \mathbb{E}_{a \sim \pi(\cdot|s)}[Q^\pi(s,a)]$$

Using our example values for state A:

$$V^\pi(A) = \pi(\text{down}|A)Q^\pi(A, \text{down}) + \pi(\text{right}|A)Q^\pi(A, \text{right})$$

$$V^\pi(A) = 0.8 \times 3 + 0.2 \times 1 = 2.4 + 0.2 = 2.6$$

This confirms that the state-value is the probability-weighted average of the action-values in that state.
Key Difference:

- The **state-value function** ($V^\pi(s)$) evaluates the expected future reward from state s assuming the agent *continues to follow policy π* from s onwards.
- The **action-value function** ($Q^\pi(s, a)$) evaluates the expected future reward from state s *given that the agent takes action a first*, and then follows policy π from the resulting next state onwards.

The Q-function is particularly important because once the optimal Q-function (Q^*) is known, deriving the optimal policy is straightforward: in any state s, the optimal action is simply the one that maximizes $Q^*(s, a)$. Learning the Q-function is the basis for popular algorithms like Q-learning.

5.7 Optimality and Optimal Policies

In Reinforcement Learning, the goal is typically to find an **optimal policy**, which is a policy that achieves the maximum possible cumulative future reward (Return) over the agent's lifetime in the environment. An optimal policy guides the agent to behave in the best possible way in every state to maximize its long-term reward.
The concept of optimality is formalized through **optimal value functions**. These are the value functions corresponding to an optimal policy. There might be multiple optimal policies, but they all share the same optimal value functions.

5.7.1 Optimal Value Functions

The optimal value functions represent the best possible performance achievable in the MDP.

- The **optimal state-value function**, denoted by $V^*(s)$, is the maximum expected return achievable from state $s \in S$ under *any* policy. It signifies the highest possible value of being in state s.
$$V^*(s) = \max_\pi V^\pi(s), \quad \forall s \in S \tag{5.10}$$
- The **optimal action-value function**, denoted by $Q^*(s, a)$, is the maximum expected return achievable from state $s \in S$ by taking action $a \in A$ and then following an optimal policy thereafter. It signifies the highest possible value of taking action a in state s.
$$Q^*(s, a) = \max_\pi Q^\pi(s, a), \quad \forall s \in S, a \in A \tag{5.11}$$

These optimal value functions are related. The optimal value of a state s is the maximum of the optimal action-values from that state:

$$V^*(s) = \max_{a \in A} Q^*(s, a) \tag{5.12}$$

Conversely, the optimal action-value $Q^*(s, a)$ is the expected immediate reward plus the discounted optimal value of the next state s', averaged over the possible next states:

$$Q^*(s, a) = \mathbb{E}[r_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a] = \sum_{s' \in S} P(s' \mid s, a)[R(s, a, s') + \gamma V^*(s')] \quad (5.13)$$

Substituting the first relationship ($V^*(s) = \max_a Q^*(s, a)$) into the second gives the Bellman Optimality Equation for $Q^*(s, a)$:

$$Q^*(s, a) = \sum_{s' \in S} P(s' \mid s, a) \left[R(s, a, s') + \gamma \max_{a' \in A} Q^*(s', a') \right] \quad (5.14)$$

This fundamental equation states that the optimal Q-value of a state-action pair is the expected value of the immediate reward plus the discounted maximum Q-value over all actions in the next state.
Example of Optimal Q-Value: Suppose for state A, we have evaluated two different policies, π_1 and π_2, and found their action-values for taking action 'down':

$$Q^{\pi_1}(A, \text{down}) = 4$$

$$Q^{\pi_2}(A, \text{down}) = 2$$

The optimal action-value $Q^*(A, \text{down})$ is the highest possible expected return achievable by taking 'down' from A and then acting optimally. While simply taking the maximum of arbitrary policies' Q-values doesn't define Q^*, it illustrates that Q^* is indeed the supremum over all policies. In this simple comparison, assuming these were the only relevant policies, the optimal Q-value for (A, down) would be:

$$Q^*(A, \text{down}) = \max(Q^{\pi_1}(A, \text{down}), Q^{\pi_2}(A, \text{down}), \ldots)$$

If we assume the optimal policy indeed yields $Q^*(A, \text{down}) = 4$, this value represents the true maximum expected return from taking 'down' in state A.

5.7.2 The Optimal Policy (π^*)

An **optimal policy**, π^*, is any policy that, for all states $s \in S$, chooses actions that are optimal with respect to the optimal action-value function $Q^*(s, a)$. That is, an optimal policy chooses an action a in state s that maximizes $Q^*(s, a)$.
An optimal deterministic policy $\pi^*(s)$ can be directly derived from $Q^*(s, a)$:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a), \quad \forall s \in S \quad (5.15)$$

This means that if an agent knows the optimal Q-function, it can achieve optimal performance by simply choosing the action with the highest Q-value in every state.
If multiple actions a yield the same maximal $Q^*(s, a)$ value in a state s, any of these actions can be chosen by an optimal policy. This indicates that there might be multiple optimal deterministic policies, and also optimal stochastic policies (that assign non-zero probability only to actions maximizing $Q^*(s, a)$).
Finding the optimal policy or the optimal value functions is the central challenge in Reinforcement Learning. Algorithms like **Q-learning** and **Deep Q-Networks (DQN)** are popular methods that aim to estimate the optimal action-value function $Q^*(s, a)$ from the agent's experience, and then derive the optimal policy from the learned Q^*.
In summary, optimal value functions (V^* and Q^*) represent the best possible performance in an MDP, and the optimal policy (π^*) is the strategy that achieves this performance by always selecting actions that maximize the optimal action-value function Q^*.

5.8 The Bellman Equation

The Bellman equation, named after Richard Bellman, is a fundamental component in solving the Markov Decision Process (MDP). Solving the MDP involves finding the optimal policy. The Bellman equation is extensively used in reinforcement learning to find the optimal value and Q functions recursively. These functions are crucial because once we have the optimal value or Q function, we can derive the optimal policy.

5.8.1 Using Value Function

The Bellman equation states that the value of a state can be obtained as the sum of the immediate reward and the discounted value of the next state. If we perform an action a in state s, move to the next state s', and receive a reward r, the Bellman equation of the value function can be expressed as:

$$V(s) = R(s, a, s') + \gamma V(s') \tag{5.16}$$

where:

- $R(s, a, s')$ denotes the immediate reward received for transitioning from state s to state s' with action a.
- γ is the discount factor.
- $V(s')$ represents the value of the next state s'.

Consider the following example. Suppose we generate a trajectory τ using some policy π:
To compute the value of state s_2, the Bellman equation is:

$$V(s_2) = R(s_2, a_2, s_3) + \gamma V(s_3) \tag{5.17}$$

where $R(s_2, a_2, s_3)$ is the immediate reward, denoted as r_2, and $\gamma V(s_3)$ is the discounted value of the next state.
Thus, the Bellman equation for the value function using policy π is:

$$V_\pi(s) = R(s, a, s') + \gamma V_\pi(s') \tag{5.18}$$

When dealing with a stochastic environment, where the next state is not guaranteed, the Bellman equation incorporates the transition probabilities:

$$V_\pi(s) = \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V_\pi(s')] \tag{5.19}$$

where:

- $P(s'|s, a)$ denotes the probability of transitioning to state s' from state s with action a.
- $R(s, a, s') + \gamma V_\pi(s')$ is the Bellman backup.

This equation accounts for the stochastic nature of the environment. Similarly, if the policy π is stochastic, the Bellman equation of the value function incorporates the probability of choosing actions according to the policy:

$$V_\pi(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V_\pi(s')] \tag{5.20}$$

The expectation form of the Bellman equation is:

$$V_\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}\left[R(s, a, s') + \gamma V_\pi(s')\right] \tag{5.21}$$

5.8.2 Using Q Function

The Bellman equation for the state-action value function, or the Q function, is similar to the value function equation but incorporates both state and action. It is expressed as:

$$Q(s,a) = R(s,a,s') + \gamma Q(s',a') \tag{5.22}$$

where:

- $R(s,a,s')$ is the immediate reward for taking action a in state s and moving to state s'.
- $Q(s',a')$ is the Q value of the next state-action pair.

For a given trajectory τ using policy π, the Q value of a state-action pair (s_2,a_2) is:

$$Q(s_2,a_2) = R(s_2,a_2,s_3) + \gamma Q(s_3,a_3) \tag{5.23}$$

When considering stochastic environments, the Bellman equation for the Q function is:

$$Q_\pi(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma Q_\pi(s',a')] \tag{5.24}$$

For stochastic policies, the Bellman equation for the Q function incorporates the policy's action probabilities:

$$Q_\pi(s,a) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma Q_\pi(s',a')] \tag{5.25}$$

5.9 Dynamic Programming

Dynamic programming (DP) is a powerful method used in reinforcement learning (RL) to solve complex problems by breaking them down into simpler subproblems. In the context of RL, it's particularly useful for solving Markov Decision Processes (MDPs) when we have a complete model of the environment.

Dynamic Programming refers to a set of algorithms used to solve the Markov Decision Process (MDP) when a complete model of the environment is available. In other words, DP methods assume access to the environment's dynamics, i.e., transition probabilities and rewards. The key idea is to compute the solution for each sub-problem once and store it, so when the same sub-problem arises again, we can reuse the solution instead of recalculating it. This approach helps reduce computational time significantly and makes DP a powerful tool in various fields, including computer science, mathematics, and bioinformatics.

In the context of Reinforcement Learning (RL), DP is used to solve Markov Decision Processes (MDPs) when the environment's model is known.

Dynamic programming is a model-based method, meaning it can be used to find the optimal policy only when the model dynamics—specifically, the transition probabilities of the environment—are known. DP methods rely on having a complete understanding of how actions affect the state transitions and rewards within the system. Without this knowledge of the environment's dynamics, DP cannot be applied. In such cases, other methods like Monte Carlo or Temporal-Difference (TD) learning, which do not require a model, are used in Reinforcement Learning.

Two important DP-based methods for finding the optimal policy in RL are:

1. Value Iteration: This method updates the value function directly by iteratively refining estimates of the optimal value.
2. Policy Iteration: This method alternates between evaluating a given policy and improving it, repeating the process until convergence to the optimal policy.

5.10 Value Iteration

Value Iteration is a key dynamic programming algorithm used in reinforcement learning to solve a Markov Decision Process (MDP) when the model (transition probabilities P and rewards R) is known. Unlike Policy Iteration, which alternates between evaluating a policy and improving it, Value Iteration focuses directly on finding the optimal value function, $V^*(s)$. Once the optimal value function is determined, the optimal policy $\pi^*(s)$ can be straightforwardly derived. The Value Iteration algorithm is a widely used method for determining the optimal strategy in an MDP.

The value iteration algorithm works by iteratively computing and updating the value function estimates for each state. It starts with an initial estimate of the value function and repeatedly applies an update rule based on the Bellman Optimality Equation. This update rule aims to converge to the optimal value function, which represents the maximum expected cumulative reward an agent can receive from each state by following the optimal policy.

In Value Iteration, the primary goal is to compute the optimal value function V^*. Once V^* is found, the optimal policy π^* is implicitly defined and can be extracted by taking the action that maximizes the expected immediate reward plus the discounted optimal value of the next state.

The working mechanism of the Value Iteration algorithm is as follows:

Steps of Value Iteration

Value Iteration is an iterative algorithm that proceeds in steps $k = 0, 1, 2, \ldots$:

1. **Initialization:** Begin by initializing the value function for all states arbitrarily, denoted as $V_0(s)$. A common and simple initialization is setting all state values to zero, $V_0(s) = 0$ for all $s \in S$. Alternatively, small random values can be used.

2. **Iterative Updates:** Repeat the following update step for all states $s \in S$ until the value function converges. In each iteration $k + 1$, the value estimate for state s, $V_{k+1}(s)$, is computed based on the value estimates from the previous iteration $V_k(s')$ for all possible next states s'. This update uses the Bellman Optimality backup:

$$V_{k+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V_k(s')\right], \quad \forall s \in S \tag{5.26}$$

   This update rule simultaneously considers all possible actions a from state s and selects the action that yields the maximum expected value.
   The term $\sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V_k(s')\right]$ is the action-value $Q_k(s,a)$ for state-action pair (s,a) using the value function V_k. Thus, the update can also be written as:

$$V_{k+1}(s) = \max_{a \in A} Q_k(s,a) \tag{5.27}$$

   This iterative process propagates optimal value estimates backward through the state space.

3. **Convergence:** Value Iteration is guaranteed to converge to the unique optimal value function $V^*(s)$ as the number of iterations $k \to \infty$, provided the MDP has a finite number of states and actions and the discount factor γ satisfies $0 \leq \gamma < 1$. Convergence is often detected by checking if the maximum change in the value function across all states in an iteration is below a predefined small threshold ϵ, i.e., $\|V_{k+1} - V_k\|_\infty < \epsilon$.

Deriving the Optimal Policy

Once the Value Iteration algorithm has converged and yielded the optimal value function $V^*(s)$, the optimal policy $\pi^*(s)$ can be readily derived. For each state s, the optimal policy selects the action a that maximizes the expected immediate reward plus the discounted optimal value of the next state. This step is equivalent to performing a policy improvement step based on V^*:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V^*(s')\right], \quad \forall s \in S \tag{5.28}$$

This derivation utilizes the relationship between the optimal value function and the optimal action-value function, $Q^*(s,a)$. The term being maximized is precisely $Q^*(s,a)$:

$$Q^*(s,a) = \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V^*(s')\right] \tag{5.29}$$

Thus, the optimal policy takes the action that maximizes the optimal Q-value:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s,a) \tag{5.30}$$

The optimal value function $V^*(s)$ is indeed the maximum of the optimal Q-values for state s:

$$V^*(s) = \max_{a \in A} Q^*(s,a) \tag{5.31}$$

Value Iteration is a widely used algorithm due to its ease of implementation and its ability to identify the optimal policy by computing the optimal value function. It does not require explicitly examining every conceivable state-action sequence. However, iterating over every state and action in each iteration can be computationally costly for MDPs with large state and action spaces, which is known as the curse of dimensionality.

As long as the MDP is finite and the discount factor γ is strictly less than 1 ($0 \leq \gamma < 1$), the Value Iteration method is guaranteed to converge to the unique optimal value function V^* and implicitly defines an optimal policy π^*. The optimal value function obtained from Value Iteration represents the maximum expected cumulative reward that can be obtained from each state by following the optimal policy.

Algorithm 24 Value Iteration Algorithm

```
1: Input: MDP model ($P(s'|s,a)$, $R(s,a,s')$), discount factor $\gamma$, sets $S$ and $A$, small threshold $\epsilon > 0$.
2: Output: Optimal value function $V^*$.
3: Initialize:
4:    Initialize value function $V(s) = 0$ for all $s \in S$.
5:    Initialize $V_{\text{next}}(s) = 0$ for all $s \in S$.
6: repeat
7:     $\Delta \leftarrow 0$
8:     for each state $s \in S$ do
9:         $v_{\text{old}} \leftarrow V(s)$
10:        Calculate the maximum expected value for state $s$:
           $Q_{\text{values}} \leftarrow []$
11:        for each action $a \in A$ do
12:            Compute action-value $Q(s,a)$ using the Bellman Optimality backup:
                   $$Q(s,a) = \sum_{s' \in S} P(s'|s,a)\left[R(s,a,s') + \gamma V(s')\right].$$
13:            Add $Q(s,a)$ to $Q_{\text{values}}$.
14:        end for
15:        $V_{\text{next}}(s) \leftarrow \max(Q_{\text{values}})$
16:        $\Delta \leftarrow \max(\Delta, |V_{\text{next}}(s) - v_{\text{old}}|)$
17:    end for
18:    $V(s) \leftarrow V_{\text{next}}(s)$ for all $s \in S$ (Synchronous update)
19: until $\Delta < \epsilon$
20: Return: Optimal value function $V^*(V)$. The optimal policy $\pi^*$ can be derived from $V^*$ afterwards by $\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$.
```

Pen & Paper

Value Iteration is a dynamic programming algorithm that finds the optimal value function $V^*(s)$ by iteratively applying the Bellman Optimality Update. Once $V^*(s)$ is found, the optimal policy $\pi^*(s)$ can be derived. This section demonstrates Value Iteration on the same simple 2x2 Grid World.

Problem Description

We are solving a simple 2×2 Grid World problem using the Value Iteration algorithm. The environment is defined as follows:

- **Grid Layout and States:** The grid has four states: $S_1 = (1,1)$, $S_2 = (1,2)$, $S_3 = (2,1)$, and $S_4 = (2,2)$.
- **Actions:** Available actions from each state are **Up**, **Down**, **Left**, and **Right**.
- **Rewards:** The description states:
  - $R(S_4) = 1$
  - $R(S_1) = R(S_2) = R(S_3) = 0$.

Based on the Value Iteration calculations below, the reward structure appears to be interpreted such that an immediate reward of 1 is obtained under specific conditions related to state S_4, while transitions between other states yield 0 reward. A possible interpretation matching the calculations is that a reward of 1 is associated with certain actions leading to/from S_4, or that S_4 itself provides a reward in its value calculation.

- **Transitions:** Moving outside the grid keeps the agent in the same state. We assume deterministic transitions within the grid as described in the Policy Iteration example:
  - S_1: Up $\to S_1$, Down $\to S_3$, Left $\to S_1$, Right $\to S_2$.
  - S_2: Up $\to S_2$, Down $\to S_4$, Left $\to S_1$, Right $\to S_2$.
  - S_3: Up $\to S_1$, Down $\to S_3$, Left $\to S_3$, Right $\to S_4$.
  - S_4: Up $\to S_2$, Down $\to S_4$, Left $\to S_3$, Right $\to S_4$. (Assuming actions from S4 transition deterministically).
- **Discount factor (γ):** $\gamma = 0.9$.

Value Iteration Update Equation

The Value Iteration algorithm iteratively applies the Bellman Optimality Equation for $V^*(s)$. Starting with an initial value function $V^{(0)}$, the value function at iteration $k+1$, $V^{(k+1)}(s)$, is computed for each state s based on the values at iteration k, $V^{(k)}(s')$. The update equation is given by:

$$V^{(k+1)}(s) = \max_{a\in A} \sum_{s'\in S} P(s'|s,a)\left[R(s,a,s') + \gamma V^{(k)}(s')\right], \quad \forall s \in S \tag{5.32}$$

where $R(s,a,s')$ is the reward for the transition from s to s' via action a. The term inside the max is the action-value $Q^{(k)}(s,a)$ at iteration k. Thus, the update is $V^{(k+1)}(s) = \max_{a\in A} Q^{(k)}(s,a)$.

Initialization

At iteration $k = 0$, the values of all states are typically initialized to zero:

$$V^{(0)}(S_1) = 0, \quad V^{(0)}(S_2) = 0, \quad V^{(0)}(S_3) = 0, \quad V^{(0)}(S_4) = 0.$$

Iterative Computation

We iteratively compute $V^{(k+1)}(s)$ for each state using the Value Iteration update equation.

Iteration 1 Using the initialized values $V^{(0)}(s) = 0$, the updated values $V^{(1)}(s)$ are computed. Assuming deterministic transitions as described, the sum $\sum_{s'} P(s'|s,a)[\dots]$ simplifies to the single term $[\dots]$ for the deterministic next state. Assuming $R(s,a,s') = 0$ for transitions from S_1, S_2, S_3 and a specific reward handling for S_4 as implied by the calculation $1 + 0.9 \cdot 0$:

- **State S_1 ($V^{(0)}(S_1) = 0$):** Possible next states with $V^{(0)} = 0$: S_2 (Right), S_3 (Down), S_1 (Left, Up).

$$\begin{aligned} V^{(1)}(S_1) &= \max\left\{ \underbrace{R(S_1, R, S_2) + 0.9 \cdot V^{(0)}(S_2)}_{\text{Right}}, \underbrace{R(S_1, D, S_3) + 0.9 \cdot V^{(0)}(S_3)}_{\text{Down}}, \dots \right\} \\ &= \max\{\underbrace{0 + 0.9 \cdot 0}_{\text{Right}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Down}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Left}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Up}}\} = 0. \end{aligned}$$

- **State S_2 ($V^{(0)}(S_2) = 0$):** Possible next states with $V^{(0)} = 0$: S_4 (Down), S_2 (Up, Right), S_1 (Left).

$$\begin{aligned} V^{(1)}(S_2) &= \max\left\{ \underbrace{R(S_2, D, S_4) + 0.9 \cdot V^{(0)}(S_4)}_{\text{Down}}, \underbrace{R(S_2, R, S_2) + 0.9 \cdot V^{(0)}(S_2)}_{\text{Right}}, \dots \right\} \\ &= \max\left\{ \underbrace{R(S_2, D, S_4) + 0.9 \cdot 0}_{\text{Down}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Right}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Left}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Up}} \right\} \qquad = \max\{R(S_2, \end{aligned}$$

Based on the provided result $V^{(1)}(S_2) = 0$, this implies $R(S_2, D, S_4)$ must be 0 in this calculation.

$$V^{(1)}(S_2) = \max\left\{ \underbrace{0.9 \cdot 0}_{\text{Down (to } S_4)}, \underbrace{0.9 \cdot 0}_{\text{Right (to } S_2)}, \underbrace{0.9 \cdot 0}_{\text{Left (to } S_1)}, \underbrace{0.9 \cdot 0}_{\text{Up (to } S_2)} \right\} = 0.$$

- **State S_3 ($V^{(0)}(S_3) = 0$):** Possible next states with $V^{(0)} = 0$: S_4 (Right), S_3 (Down, Left), S_1 (Up).

$$\begin{aligned} V^{(1)}(S_3) &= \max\left\{ \underbrace{R(S_3, R, S_4) + 0.9 \cdot V^{(0)}(S_4)}_{\text{Right}}, \underbrace{R(S_3, D, S_3) + 0.9 \cdot V^{(0)}(S_3)}_{\text{Down}}, \dots \right\} \\ &= \max\left\{ \underbrace{R(S_3, R, S_4) + 0.9 \cdot 0}_{\text{Right}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Down}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Left}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Up}} \right\} \qquad = \max\{R(S_3, \end{aligned}$$

Based on the provided result $V^{(1)}(S_3) = 0$, this implies $R(S_3, R, S_4)$ must be 0 in this calculation.

$$V^{(1)}(S_3) = \max\left\{ \underbrace{0.9 \cdot 0}_{\text{Right (to } S_4)}, \underbrace{0.9 \cdot 0}_{\text{Down (to } S_3)}, \underbrace{0.9 \cdot 0}_{\text{Left (to } S_3)}, \underbrace{0.9 \cdot 0}_{\text{Up (to } S_1)} \right\} = 0.$$

- **State S_4 ($V^{(0)}(S_4) = 0$):** Possible next states with $V^{(0)} = 0$: S_2, S_4, S_3 depending on action.

$$\begin{aligned} V^{(1)}(S_4) &= \max_a \left\{ R(S_4, a, s') + 0.9 \cdot V^{(0)}(s') \right\} \\ &= \max_a \left\{ R(S_4, a, s') + 0.9 \cdot 0 \right\} = \max_a \{R(S_4, a, s')\}. \end{aligned}$$

Based on the provided calculation $\max\{1 + 0.9 \cdot 0\} = 1$, this implies there is at least one action from S_4 that yields an immediate reward of 1 (e.g., $R(S_4, \text{Stay}, S_4) = 1$), and this is the maximum possible immediate reward from S_4.

$$V^{(1)}(S_4) = \max\left\{ \underbrace{1 + 0.9 \cdot 0}_{\text{Any action yielding } R=1} \right\} = 1.$$

Thus, after iteration 1, based on the provided calculations:

$$V^{(1)}(S_1) = 0, \quad V^{(1)}(S_2) = 0, \quad V^{(1)}(S_3) = 0, \quad V^{(1)}(S_4) = 1.$$

This calculation implies a reward structure where transitions from S_1, S_2, S_3 yield 0 reward, and at least one action from S_4 yields a reward of 1, with $V^{(0)}$ being 0. Note that this contradicts the common reward structure $R(s, a, s') = 1$ if $s' = S_4$ used in some examples.

Iteration 2 Repeating the process for iteration 2 using $V^{(1)} = \{0, 0, 0, 1\}$.

- **State** S_1 **(**$V^{(1)}(S_1) = 0$**):** Next states S_2, S_3 have $V^{(1)} = 0$. Next state S_1 has $V^{(1)} = 0$.

$$\begin{aligned} V^{(2)}(S_1) &= \max\left\{R(S_1, R, S_2) + 0.9 \cdot V^{(1)}(S_2), \dots\right\} \\ &= \max\left\{0 + 0.9 \cdot 0, 0 + 0.9 \cdot 0, 0 + 0.9 \cdot 0, 0 + 0.9 \cdot 0\right\} = 0. \end{aligned}$$

- **State** S_2 **(**$V^{(1)}(S_2) = 0$**):** Next state S_4 (Down) has $V^{(1)} = 1$. Next states S_2, S_1 have $V^{(1)} = 0$.

$$\begin{aligned} V^{(2)}(S_2) &= \max\left\{R(S_2, D, S_4) + 0.9 \cdot V^{(1)}(S_4), \dots\right\} \\ &= \max\left\{\underbrace{R(S_2, D, S_4) + 0.9 \cdot 1}_{\text{Down}}, \underbrace{R(S_2, R, S_2) + 0.9 \cdot V^{(1)}(S_2)}_{\text{Right}}, \dots\right\} \\ &= \max\left\{R(S_2, D, S_4) + 0.9, \underbrace{0 + 0.9 \cdot 0}_{\text{Right}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Left}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Up}}\right\}. \end{aligned}$$

Based on the provided result $V^{(2)}(S_2) = 1$, this implies $R(S_2, D, S_4) + 0.9 = 1$, so $R(S_2, D, S_4) = 0.1$, and this is the maximum Q-value.

$$V^{(2)}(S_2) = \max\left\{\underbrace{1 + 0.9 \cdot 0}_{\text{This calculation does not use } V^{(1)} \text{ values consistent with } R\text{=0 or } R\text{=1 for S4 entry and } V^{(1)}\text{=(0,0,0,1)}}, \dots\right.$$

The provided calculation $\max\{1 + 0.9 \cdot 0, \dots\} = 1$ for $V^{(2)}(S_2)$ is inconsistent with using $V^{(1)} = (0, 0, 0, 1)$ and a standard reward function. If we assume $R(s, a, s') = 1$ when $s' = S_4$, then $Q(S_2, D) = 1 + 0.9V^{(1)}(S_4) = 1 + 0.9 \cdot 1 = 1.9$. Based on the user's provided calculation result for $V^{(2)}(S_2)$:

$$V^{(2)}(S_2) = 1.$$

- **State** S_3 **(**$V^{(1)}(S_3) = 0$**):** Next state S_4 (Right) has $V^{(1)} = 1$. Next states S_3, S_1 have $V^{(1)} = 0$.

$$\begin{aligned} V^{(2)}(S_3) &= \max\left\{R(S_3, R, S_4) + 0.9 \cdot V^{(1)}(S_4), \dots\right\} \\ &= \max\left\{\underbrace{R(S_3, R, S_4) + 0.9 \cdot 1}_{\text{Right}}, \underbrace{0 + 0.9 \cdot 0}_{\text{Down}}, \dots\right\}. \end{aligned}$$

Based on the provided result $V^{(2)}(S_3) = 1$, this implies $R(S_3, R, S_4) + 0.9 = 1$, so $R(S_3, R, S_4) = 0.1$, and this is the maximum Q-value. Similar inconsistency in the calculation steps as for S_2. Based on the user's provided calculation result for $V^{(2)}(S_3)$:

$$V^{(2)}(S_3) = 1.$$

- **State S_4 ($V^{(1)}(S_4) = 1$):** Next states S_2 ($V^{(1)} = 0$), S_4 ($V^{(1)} = 1$), S_3 ($V^{(1)} = 0$).

$$V^{(2)}(S_4) = \max_a \Big\{ R(S_4, a, s') + 0.9 \cdot V^{(1)}(s') \Big\}.$$

  Using the reward structure $R(S_4, \text{Stay}, S_4) = 1$ and $R = 0$ otherwise: $Q(S_4, \text{Stay}) = 1 + 0.9V^{(1)}(S_4) = 1 + 0.9 \cdot 1 = 1.9$. $Q(S_4, \text{Up}) = 0 + 0.9V^{(1)}(S_2) = 0 + 0.9 \cdot 0 = 0$. $Q(S_4, \text{Left}) = 0 + 0.9V^{(1)}(S_3) = 0 + 0.9 \cdot 0 = 0$. $Q(S_4, \text{Down/Right}) = 0 + 0.9V^{(1)}(S_4) = 0 + 0.9 \cdot 1 = 0.9$. $V^{(2)}(S_4) = \max(1.9, 0, 0.9) = 1.9$. Based on the provided calculation result $V^{(2)}(S_4) = 1$, this implies an inconsistency with standard Value Iteration using $V^{(1)} = (0, 0, 0, 1)$ and typical reward structures. Based on the user's provided calculation result for $V^{(2)}(S_4)$:

$$V^{(2)}(S_4) = 1 \quad \text{(unchanged from Iteration 1 result)}.$$

Thus, after iteration 2, based on the provided calculations:

$$V^{(2)}(S_1) = 0, \quad V^{(2)}(S_2) = 1, \quad V^{(2)}(S_3) = 1, \quad V^{(2)}(S_4) = 1.$$

The value function estimates have changed from Iteration 1 to Iteration 2 for states S_2 and S_3. Value Iteration continues until the maximum change in value across all states is below a small threshold ϵ.

Final Values After sufficient iterations, the value function estimates would converge. Based on the provided example, the final converged values are stated as:

$$V^*(S_1) = 0, \quad V^*(S_2) = 1, \quad V^*(S_3) = 1, \quad V^*(S_4) = 1.$$

These values match the results obtained after Iteration 2, suggesting convergence occurred at or before this point in the provided example.
This example demonstrates the iterative nature of Value Iteration, applying the Bellman Optimality backup to update state values. While the specific numerical calculations provided for Iterations 1 and 2 show some inconsistencies with standard Value Iteration formula given typical reward structures ($R(s, a, s') = 1$ upon entering S4 or $R(s) = 1$ in S4) and the previous iteration's values, the overall process of updating $V^{(k+1)}$ based on $V^{(k)}$ and the max operation is correctly illustrated.

5.11 Policy Iteration

Policy Iteration is a fundamental dynamic programming algorithm widely used in reinforcement learning to find the optimal policy for a Markov Decision Process (MDP) when the full model (transition probabilities P and reward function R) is known. It is one of the foundational algorithms, alongside Value Iteration, for solving MDPs. Policy iteration works by iterating a given policy until it converges to the optimal policy π^*.

The objective of Policy Iteration is to converge to an optimal policy π^*, which is a policy that maximizes the expected cumulative reward for every state in the state space. The algorithm achieves this through a sequence of alternating steps: Policy Evaluation and Policy Improvement.

Steps of Policy Iteration

The Policy Iteration algorithm proceeds in iterations, with each iteration consisting of two main phases:

1. **Policy Evaluation:** In this step, the algorithm calculates the state-value function $V^\pi(s)$ for the current policy π. This involves determining the expected cumulative return for each state if the agent follows the current policy. Theoretically, this step requires solving a system of linear equations given by the Bellman Expectation Equation for $V^\pi(s)$ for all states $s \in S$:

$$V^\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^\pi(s')\right] \tag{5.33}$$

For large state spaces, solving this system directly can be computationally expensive. In practice, this step is often performed iteratively using methods like **iterative policy evaluation**, where the value function is updated repeatedly for all states until it converges to V^π.

$$V_{k+1}(s) = \sum_{a \in A} \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V_k(s')\right] \tag{5.34}$$

This iterative process continues until the value function estimates stabilize, i.e., $\|V_{k+1} - V_k\|_\infty < \epsilon$ for a small tolerance ϵ. This step estimates the expected cumulative reward for each state under the current policy π.

Algorithm 25 Policy Iteration

```
1: **Input:** MDP model ($P(s'|s,a)$, $R(s,a,s')$), discount factor $\gamma$, sets $S$ and $A$.
2: **Output:** Optimal policy $\pi^*$ and optimal value function $V^*$.
3: **Initialize:**
4:     Assign an arbitrary initial policy $\pi_0(s)$ for all $s \in S$.
5:     Initialize value function $V_0(s) = 0$ for all $s \in S$.
6:     Set $k = 0$ (iteration counter).
7: repeat
8:     $k \leftarrow k + 1$
9:     $\pi_{k-1} \leftarrow \pi$ (Store the policy from the previous iteration)
10:    Policy Evaluation:
11:        Compute $V_{\pi_{k-1}}(s)$ for all $s \in S$. This is typically done by iteratively applying the Bellman Expectation backup:
12:        Initialize $V(s)$ arbitrarily (e.g., from $V_{k-1}$ or zeros).
13:    repeat
14:        $\Delta \leftarrow 0$
15:        for each state $s \in S$ do
16:            $v \leftarrow V(s)$
17:            Compute $V_{new}(s) = \sum_{s' \in S} P(s'|s,\pi_{k-1}(s)) \left[R(s,\pi_{k-1}(s),s') + \gamma V(s')\right]$.
18:            $V(s) \leftarrow V_{new}(s)$
19:            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
20:        end for
21:    until $\Delta < \epsilon$ (a small threshold for convergence of $V$)
22:    $V_k \leftarrow V$ (Store the converged value function)
23:    Policy Improvement:
24:    policy_stable $\leftarrow$ true
25:    for each state $s \in S$ do
26:        old_action $\leftarrow \pi_{k-1}(s)$
27:        Compute the action-value function $Q(s,a)$ for each action $a \in A$ using $V_k$:
               $$Q(s,a) = \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma V_k(s')\right]. \quad (5.35)$$
28:        Determine the best action:
               $$\pi_k(s) = \arg\max_{a \in A} Q(s,a). \quad (5.36)$$
29:        if $\pi_k(s) \neq$ old_action then
30:            policy_stable $\leftarrow$ false
31:        end if
32:    end for
33:    $\pi \leftarrow \pi_k$ (Update the working policy)
34: until policy_stable
35: **Return:** Optimal policy $\pi^*$ ($\pi_k$) and optimal value function $V^*$ ($V_k$).
```

2. **Policy Improvement:** In this step, the algorithm updates the current policy π to a new policy π' by making it greedy with respect to the value function V^π calculated in the Policy Evaluation step. For each state s, the new policy selects the action a that maximizes the action-value function $Q^\pi(s,a)$. The Q-function is calculated using the

current value function $V^\pi(s')$:

$$Q^\pi(s,a) = \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^\pi(s')\right] \tag{5.37}$$

The improved policy $\pi'(s)$ for each state s is then derived as:

$$\pi'(s) = \arg\max_{a\in A} Q^\pi(s,a) \tag{5.38}$$

This step ensures that the policy is improved by choosing actions that yield the highest expected reward from each state, given the current evaluation of state values. If multiple actions yield the same maximum Q-value, the policy can choose any of them (or a stochastic mix).

3. **Iteration:** The algorithm alternates between Policy Evaluation and Policy Improvement. The improved policy from step 2 becomes the current policy for step 1 of the next iteration. This iterative process continues until the policy converges, meaning the Policy Improvement step yields a policy that is identical to the policy evaluated in that iteration. When the policy can no longer be improved, it is guaranteed to be the optimal policy π^*.

Convergence of Policy Iteration

Policy Iteration is guaranteed to converge to the optimal policy π^* and the optimal value function $V^*(s)$ for finite MDPs, provided the discount factor γ satisfies $0 \le \gamma < 1$. For episodic tasks, convergence is also guaranteed with $\gamma = 1$. Policy Improvement step guarantees that the new policy is strictly better than or equal to the current policy ($V^{\pi_{k+1}}(s) \ge V^{\pi_k}(s)$ for all s). Since there is a finite number of policies in a finite MDP, and the policy strictly improves whenever it is not optimal, the algorithm must converge to the optimal policy in a finite number of iterations (policy changes).
At convergence, the optimal value function $V^*(s)$ satisfies the Bellman Optimality Equation:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^*(s')\right] \tag{5.39}$$

And the optimal policy π^* is one that is greedy with respect to V^*, meaning it satisfies:

$$\pi^*(s) = \arg\max_a \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^*(s')\right] \tag{5.40}$$

This also implies that $\pi^*(s) = \arg\max_a Q^*(s,a)$, where $Q^*(s,a)$ is the optimal action-value function.

Comparison with Value Iteration

Policy Iteration and Value Iteration are both dynamic programming methods for solving MDPs with known models. They have key differences in their approach:

- **Evaluation vs. Improvement Focus:** Policy Iteration performs full policy evaluation in each iteration before improving the policy. Value Iteration, conversely, directly updates the value function using the Bellman Optimality Equation, effectively combining value evaluation and policy improvement implicitly in each step.

- **Computational Cost Per Iteration:** A full Policy Evaluation step typically involves solving a system of linear equations (or performing multiple iterative updates until convergence), which can be computationally expensive, especially for MDPs with a very large number of states. Value Iteration's update step (a maximization over actions and summation over next states) is generally computationally cheaper per state per iteration.
- **Number of Iterations:** While Policy Iteration's Policy Evaluation step may take many internal iterations to converge, the number of *policy improvement* steps (outer loop iterations) is often small in practice compared to the number of value updates required for Value Iteration to converge to a policy. Policy Iteration converges in a finite number of policy changes.

For MDPs with very large state and action spaces, both standard Policy Iteration and Value Iteration can become intractable due to the "curse of dimensionality." In such cases, approximation methods, such as **Generalized Policy Iteration (GPI)** which underlies many modern RL algorithms, are used. Policy Iteration remains a foundational algorithm due to its strong theoretical guarantees and its clear separation of policy evaluation and improvement concepts.

Pen & Paper

Policy Iteration is a classical dynamic programming algorithm used to find the optimal policy for a finite MDP. It guarantees convergence to the optimal policy in a finite number of iterations for finite MDPs. The algorithm alternates between two steps: Policy Evaluation and Policy Improvement.

Problem Description

We will illustrate the Policy Iteration algorithm using a simple 2×2 Grid World problem. Let's define the environment:

- **Grid Layout and States:** The environment is a 2×2 grid. We label the states as S_1, S_2, S_3, S_4. A common mapping is $S_1 = (1, 1)$ (top-left), $S_2 = (1, 2)$ (top-right), $S_3 = (2, 1)$ (bottom-left), and $S_4 = (2, 2)$ (bottom-right).
- **Actions:** From each state, the agent can attempt to move **Up**, **Down**, **Left**, or **Right**. We assume these actions are deterministic if they stay within the grid.
- **Rewards:** The reward structure influences the agent's goal. The description states:
  - $R(S_4) = 1$
  - $R(S_1) = R(S_2) = R(S_3) = 0$.

  Based on the provided Policy Evaluation calculations, this reward structure appears to be simplified, where a reward of 1 is potentially associated with reaching or being in state S_4, while transitions between other states yield 0 reward. In the calculations below, it seems S_4 is treated as a special state with a fixed value or a reward gained upon entry/presence that factors into value propagation in a specific way to match the numbers provided.
- **Transitions:** Moving outside the grid boundaries by attempting an action keeps the agent in the current state. Assuming deterministic movement within boundaries:
  - From S_1: Up $\to S_1$, Down $\to S_3$, Left $\to S_1$, Right $\to S_2$.

  - From S_2: Up $\to S_2$, Down $\to S_4$, Left $\to S_1$, Right $\to S_2$.
  - From S_3: Up $\to S_1$, Down $\to S_3$, Left $\to S_3$, Right $\to S_4$.
  - From S_4: Up $\to S_2$, Down $\to S_4$, Left $\to S_3$, Right $\to S_4$. (Assuming actions from S4 transition deterministically as well).

- **Discount factor (γ):** $\gamma = 0.9$. Future rewards are discounted by this factor.

Policy Iteration Procedure: The Policy Iteration algorithm alternates between iteratively evaluating the current policy and then improving it based on the calculated values.
Let's walk through one iteration starting with an initial policy.

1. **Initialization:** We begin by initializing a policy arbitrarily and setting the initial value function for all states.

   - Initialize the policy arbitrarily: $\pi_0(S_1) = \text{Right}$ $\pi_0(S_2) = \text{Down}$ $\pi_0(S_3) = \text{Right}$ $\pi_0(S_4) = \text{Stay}$ (assuming 'Stay' is a valid action in S4, or interpret this as the policy action from S4, e.g., mapping 'Stay' to self-loop in S4). Let's assume 'Stay' in S4 results in staying in S4.
   - Initialize the value function for all states, typically to zero: $V_0(s) = 0$ for all $s \in \{S_1, S_2, S_3, S_4\}$. (Note: The Policy Evaluation below seems to bypass the $V_0 = 0$ initialization and directly solves for V_1, implying the initial policy is evaluated fully).

2. **Policy Evaluation (Step 1):** This step calculates the value function $V^{\pi_k}(s)$ for the current policy π_k. This is done by solving the system of linear equations given by the Bellman Expectation Equation for all states $s \in S$. For a deterministic policy $\pi(s)$ in a deterministic environment:

$$V^{\pi}(s) = R(s, \pi(s), s') + \gamma V^{\pi}(s') \quad \text{(where } s' \text{ is the deterministic next state)} \tag{5.41}$$

   More generally, for a deterministic policy in a stochastic environment, this is:

$$V^{\pi}(s) = \sum_{s'} P(s'|s, \pi(s)) \left[R(s, \pi(s), s') + \gamma V^{\pi}(s')\right] \tag{5.42}$$

   In this specific example, based on the provided calculations, the Bellman equations for evaluation seem to take a simplified form, where the reward $R(s, a, s')$ is 0 for transitions not entering S_4, and $V(S_4)$ is treated as a fixed value of 1, perhaps representing a boundary condition or terminal reward accumulation. Let's assume the reward $R(s, a, s') = 0$ for transitions from S_1, S_2, S_3, and that $V(S_4)$ is fixed at 1. The equations derived from the initial policy π_0 are:

$$\pi_0(S_1) = \text{Right} \implies S_1 \to S_2 \quad \implies V(S_1) = \gamma V(S_2) \quad \text{(assuming } R = 0\text{)}$$

$$\pi_0(S_2) = \text{Down} \implies S_2 \to S_4 \quad \implies V(S_2) = \gamma V(S_4) \quad \text{(assuming } R = 0 \text{ and special } V(S_4)\text{)}$$

$$\pi_0(S_3) = \text{Right} \implies S_3 \to S_4 \quad \implies V(S_3) = \gamma V(S_4) \quad \text{(assuming } R = 0 \text{ and special } V(S_4)\text{)}$$

$$\pi_0(S_4) = \text{Stay} \implies S_4 \to S_4 \quad \implies V(S_4) = 1 \quad \text{(as given)}$$

Using $\gamma = 0.9$ and the fixed $V(S_4) = 1$, we solve these:

$$V_1(S_4) = 1$$

$$V_1(S_2) = 0.9 \cdot V_1(S_4) = 0.9 \cdot 1 = 0.9$$
$$V_1(S_3) = 0.9 \cdot V_1(S_4) = 0.9 \cdot 1 = 0.9$$
$$V_1(S_1) = 0.9 \cdot V_1(S_2) = 0.9 \cdot 0.9 = 0.81$$

So, the value function for the initial policy π_0 is $V^{\pi_0} = \{V(S_1) = 0.81, V(S_2) = 0.9, V(S_3) = 0.9, V(S_4) = 1\}$.

3. **Policy Improvement (Step 1):** This step generates a new, improved policy π_{k+1} by acting greedily with respect to the value function V^{π_k} calculated in the evaluation step. For each state s, the agent selects the action a that maximizes the action-value function $Q^{\pi_k}(s,a)$, which is calculated using the current values $V^{\pi_k}(s')$:

$$Q^{\pi_k}(s,a) = \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^{\pi_k}(s')\right]. \tag{5.43}$$

The new policy $\pi_{k+1}(s)$ for each state s is then:

$$\pi_{k+1}(s) = \arg\max_{a \in A} Q^{\pi_k}(s,a)$$

Let's calculate the Q-values for all states and actions using V^{π_0} and assuming $R(s,a,s') = 0$ for transitions from S_1, S_2, S_3, and $R(S_4, Stay, S_4) = 1$, with $V(S_4) = 1$ fixed for calculation targets:

For State S_1 ($V(S_1) = 0.81$): $Q(S_1, \text{Right}) = P(S_2|S_1, R)[R(S_1, R, S_2) + \gamma V(S_2)] = 1 \cdot [0 + 0.9 \cdot 0.9] = 0.81$
$Q(S_1, \text{Down}) = P(S_3|S_1, D)[R(S_1, D, S_3) + \gamma V(S_3)] = 1 \cdot [0 + 0.9 \cdot 0.9] = 0.81$
$Q(S_1, \text{Left}) = P(S_1|S_1, L)[R(S_1, L, S_1) + \gamma V(S_1)] = 1 \cdot [0 + 0.9 \cdot 0.81] = 0.729$
$Q(S_1, \text{Up}) = P(S_1|S_1, U)[R(S_1, U, S_1) + \gamma V(S_1)] = 1 \cdot [0 + 0.9 \cdot 0.81] = 0.729$

The maximum Q-value for S_1 is 0.81. The actions 'Right' and 'Down' are equally optimal. The updated policy $\pi_1(S_1)$ can be either Right or Down (or any stochastic mix). Keeping it as the initial 'Right' is valid if breaking ties towards the initial policy.

For State S_2 ($V(S_2) = 0.9$): $Q(S_2, \text{Down}) = P(S_4|S_2, D)[R(S_2, D, S_4) + \gamma V(S_4)] = 1 \cdot [0 + 0.9 \cdot 1] = 0.9$. (Matches $V(S_2)$)
$Q(S_2, \text{Right}) = P(S_2|S_2, R)[R(S_2, R, S_2) + \gamma V(S_2)] = 1 \cdot [0 + 0.9 \cdot 0.9] = 0.81$.
$Q(S_2, \text{Left}) = P(S_1|S_2, L)[R(S_2, L, S_1) + \gamma V(S_1)] = 1 \cdot [0 + 0.9 \cdot 0.81] = 0.729$.
$Q(S_2, \text{Up}) = P(S_2|S_2, U)[R(S_2, U, S_2) + \gamma V(S_2)] = 1 \cdot [0 + 0.9 \cdot 0.9] = 0.81$.

The maximum Q-value for S_2 is 0.9. The action 'Down' is optimal. The updated policy $\pi_1(S_2) = \text{Down}$. (Note: The user's provided Q values $Q(S_2, \text{Down}) = 1, Q(S_2, \text{Right}) = 1$ appear inconsistent with the standard formula and calculated V-values. Based on calculation, max is 0.9 for 'Down'). We will proceed with the calculated max 0.9 for 'Down'.

For State S_3 ($V(S_3) = 0.9$): $Q(S_3, \text{Right}) = P(S_4|S_3, R)[R(S_3, R, S_4) + \gamma V(S_4)] = 1 \cdot [0 + 0.9 \cdot 1] = 0.9$. (Matches $V(S_3)$)
$Q(S_3, \text{Up}) = P(S_1|S_3, U)[R(S_3, U, S_1) + \gamma V(S_1)] = 1 \cdot [0 + 0.9 \cdot 0.81] = 0.729$.
$Q(S_3, \text{Left}) = P(S_3|S_3, L)[R(S_3, L, S_3) + \gamma V(S_3)] = 1 \cdot [0 + 0.9 \cdot 0.9] = 0.81$.
$Q(S_3, \text{Down}) = P(S_3|S_3, D)[R(S_3, D, S_3) + \gamma V(S_3)] = 1 \cdot [0 + 0.9 \cdot 0.9] = 0.81$.

The maximum Q-value for S_3 is 0.9. The action 'Right' is optimal. The updated policy $\pi_1(S_3) = \text{Right}$. (Note: The user's provided Q values $Q(S_3, \text{Right}) = 1, Q(S_3, \text{Up}) = 0.81$ appear inconsistent with the standard formula and calculated V-values. Based on calculation, max is 0.9 for 'Right'). We will proceed with the calculated max 0.9 for 'Right'.

4. **State S_4 ($V(S_4) = 1$):** The problem description implies S_4 might be a goal or terminal state. The initial policy has $\pi_0(S_4) = \text{Stay}$. If we assume 'Stay' is the only action from S4, or if its value is fixed, policy improvement might not apply or the calculation might be different. Assuming we still apply the Q formula and aim to maximize Q in S_4 using $V(S_4) = 1$ and $R(S_4, a, S_4) = 1$ only for action 'Stay', $R = 0$ otherwise:

   $Q(S_4, \text{Stay}) = P(S_4|S_4, S)[R(S_4, S, S_4) + \gamma V(S_4)] = 1 \cdot [1 + 0.9 \cdot 1] = 1.9$.
   $Q(S_4, \text{Right}) = P(S_4|S_4, R)[R(S_4, R, S_4) + \gamma V(S_4)] = 1 \cdot [0 + 0.9 \cdot 1] = 0.9$.
   (Similar for Down, Left, Up actions which also lead to S4 with R=0). Max Q for S4 is 1.9 ('Stay'). Updated policy $\pi_1(S_4) = \text{Stay}$. This matches the initial policy for S_4.

   Based on the policy improvement step using the calculated V^{π_0} and Q values, the updated policy π_1 is:

$$\pi_1(S_1) \in \{\text{Right}, \text{Down}\}$$
$$\pi_1(S_2) = \text{Down}$$
$$\pi_1(S_3) = \text{Right}$$
$$\pi_1(S_4) = \text{Stay}$$

   Comparing π_1 to π_0, the policy for S_2 and S_3 has potentially changed (from allowing Right/Down to forcing Down for S2, and from Right to Right for S3, but the user's initial policy was Right for S3 anyway). The policy for S_1 remains one of the initial options. Let's assume $\pi_1(S_1) = \text{Right}$ for comparison. The policy for S_4 remains 'Stay'. Since the policy might have changed for S_2 and S_3, the algorithm proceeds to the next iteration of Policy Evaluation and Policy Improvement.

5. **Policy Evaluation (Step 2):** Now we evaluate the new policy π_1. If $\pi_1(S_1) = \text{Right}, \pi_1(S_2) = \text{Down}, \pi_1(S_3) = \text{Right}, \pi_1(S_4) = \text{Stay}$, this is the same policy that was evaluated in Step 1. Evaluating this policy again will yield the same value function: $V_2 = V_1 = \{0.81, 0.9, 0.9, 1\}$.

6. **Policy Improvement (Step 2):** We improve the policy based on V_2. Since $V_2 = V_1$, the Q-values calculated will be the same as in Policy Improvement (Step 1). The updated policy π_2 will therefore be the same as π_1.

Convergence: The algorithm converges when the policy calculated in the Policy Improvement step is the same as the policy evaluated in the Policy Evaluation step. In this case, after the first full iteration, if we assume $\pi_1(S_1) = \text{Right}$, then π_1 is identical to π_0. If there was a change (e.g., if the tie-breaking in S_1 resulted in $\pi_1(S_1) = \text{Down}$), the algorithm would iterate again. Assuming the policy stabilizes, the final policy obtained is the optimal policy π^*.
Based on the user's stated final policy and values, it implies that the policy converged to:

$$\pi^*(S_1) = \text{Right}, \quad \pi^*(S_2) = \text{Down}, \quad \pi^*(S_3) = \text{Right}, \quad \pi^*(S_4) = \text{Stay}.$$

And the corresponding optimal value function is:

$$V^*(S_1) = 0.81, \quad V^*(S_2) = 0.9, \quad V^*(S_3) = 0.9, \quad V^*(S_4) = 1.$$

This specific example, while illustrating the Policy Iteration steps, uses a potentially non-standard reward structure or value definition for S_4 to arrive at the given numerical results for the value function. However, the process of alternating between policy evaluation (solving for V^π) and policy improvement (acting greedily with respect to Q^π) is correctly demonstrated. The Policy Iteration algorithm guarantees convergence to the optimal policy in a finite number of steps for finite MDPs.