MDS 552 - Applied Machine Learning

Prepared for MDS, SMSTU

by
Rajan Adhikari

Masters in Data Science
School of Mathematical Sciences
August 31, 2024

Before We Begin

Course Overview

Nature of Course: Theory + Practical (Compulsory)
Course Description: This course provides a comprehensive introduction to machine learning, focusing on its practical applications in real-world scenarios. Students will explore the three main paradigms of machine learning: Supervised Learning, Unsupervised Learning, and Reinforcement Learning. The course will cover various algorithms within each paradigm, their implementation, and evaluation metrics to determine the most suitable algorithm for specific tasks.
Throughout the course, students will:

1. Gain a solid theoretical foundation in machine learning concepts
2. Develop practical skills in implementing and applying machine learning algorithms
3. Learn to evaluate and select appropriate models for different problem domains
4. Explore cutting-edge topics in deep learning and neural networks

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.

```
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)
```

Listing 1: Sample KNN Implementation

Evaluation Criteria

The course assessment is designed to evaluate both theoretical understanding and practical skills:

- **Internal Evaluation: 30 Marks**
  - First Term Exam: 8 Marks
  - Second Term Exam: 8 Marks
  - Assignments: 5 Marks (5 Assignments, each carrying 1 mark)
  - Presentation: 5 Marks (Paper-based presentation)
  - Attendance: 2 Marks
  - Teacher's Evaluation: 2 Marks
- **Final Evaluation: 45 Marks**

Course Contents

1. Introduction to Machine Learning
2. Supervised Learning
3. Unsupervised Learning
4. Model Evaluation and Selection
5. Reinforcement Learning
6. Neural Networks and Deep Learning

Assignments

Unit 2: Supervised Learning

7. **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.
8. **Gradient Descent Techniques**
   a. Implement stochastic gradient descent for linear regression on a toy dataset. Compare with batch gradient descent.
9. **Locally Weighted Regression**

a. Implement locally weighted regression from scratch on a toy dataset. Visualize and discuss results.

10. **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.

11. **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.

12. **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.

13. **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

14. **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.

15. **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.

16. **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.

17. **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

18. **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.
19. **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

20. **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.
21. **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

22. **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.
23. **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.
24. **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 $\text{Weight} = w_0 + w_1 \cdot \text{Height} + \epsilon$. 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.

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}

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 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:

$$\hat{y} = h(x) = w_0 + w_1x_1 + ... + w_nx_n$$

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:

$$\begin{aligned}
\frac{\partial}{\partial w_j} \frac{1}{2}(J(w_j))^2 &= \frac{\partial}{\partial w_j} \frac{1}{2}(h(x) - y)^2 \\
&= \frac{1}{2} \frac{\partial}{\partial w_j}(h(x) - y)^2 \\
&= \frac{1}{2} \frac{\partial}{\partial (h(x) - y)}(h(x) - y)^2 . \frac{\partial}{\partial w_j}(h(x) - y) \\
&= \frac{1}{2} .2(h(x) - y) \frac{\partial}{\partial w_j}(h(x) - y) \\
&= (h(x) - y) \cdot \frac{\partial}{\partial w_j}(w_0 + w_1 x_1 + \ldots + w_n x_n - y) \\
&= (h(x) - y) \cdot \frac{\partial}{\partial w_j}(w_0 + w_1 x_1 + \ldots + w_j x_j + \ldots + w_n x_n - y) \\
&= (h(x) - y) \cdot x_j
\end{aligned}$$

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 α, number of epochs n
Ensure: Optimized weights w
Initialize weights $w \leftarrow$ random small values
for $epoch = 1$ to n **do**
    Shuffle training data (X, y)
    for each (x_i, y_i) in (X, y) **do**
        $prediction \leftarrow w^T \cdot x_i$
        $error \leftarrow prediction - y_i$
        $gradient \leftarrow error \cdot x_i$
        $w \leftarrow w - \alpha \cdot gradient$
    end for
end for
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.

Algorithm 2 Batch Linear Regression

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

2.1.5 Numerical Example:

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:

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

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:

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

$$\begin{aligned}
\text{For } (x_1, x_2) = (1, 2): &\quad h(x_1) = 0 + 0 \cdot 1 + 0 \cdot 2 = 0 \\
\text{For } (x_1, x_2) = (2, 3): &\quad h(x_1) = 0 + 0 \cdot 2 + 0 \cdot 3 = 0 \\
\text{For } (x_1, x_2) = (3, 4): &\quad h(x_1) = 0 + 0 \cdot 3 + 0 \cdot 4 = 0
\end{aligned}$$

- 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:

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

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

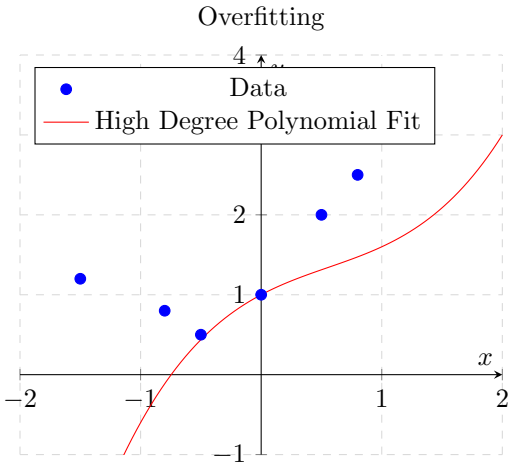
$$w_2 := w_2 - \alpha \cdot \frac{\partial J}{\partial w_2} = -0.153 - 0.01 \cdot 7.686 = -0.230$$

2.2 Overfitting and Underfitting

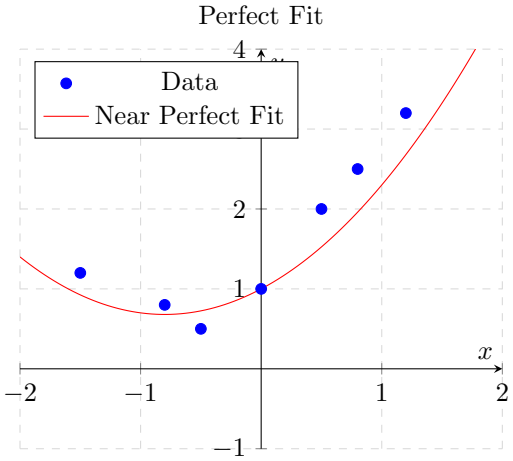
In machine learning, overfitting and underfitting are two fundamental concepts that describe how well a model fits the training data and generalizes to new, unseen data.

2.2.1 Overfitting

Overfitting occurs when a model learns the training data too well, capturing noise and outliers in the data along with the underlying pattern. As a result, the model performs well on the training data but poorly on unseen data (test data), as it has become too tailored to the specific characteristics of the training data.



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.



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.

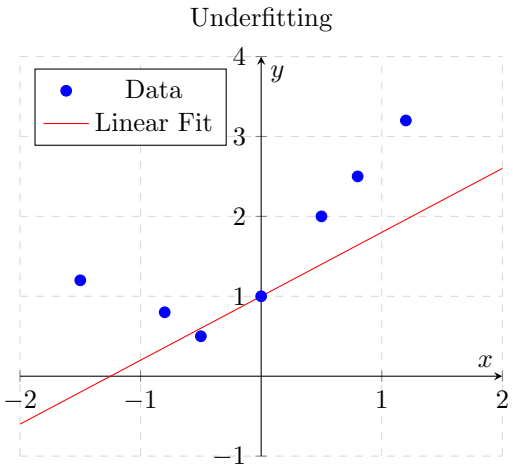
Mitigation Strategies:

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

Understanding overfitting and underfitting is closely related to the bias-variance tradeoff. In supervised settings, the prediction error e is composed of the bias , the variance and the irreducible error i.e.

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

** Refer to https://www.inf.ed.ac.uk/teaching/courses/mlsc/Notes/Lecture4/BiasVariance.pdf for the derivation*

1. **Bias**: This represents the error introduced by the assumptions made by the learning method to simplify the model. For instance, when a non-linear function $f(x)$ is approximated using a linear model, the error in the estimates $h(x)$ is caused by these simplifying assumptions.
2. **Variance**: This quantifies how much the learning method $h(x)$ varies around its mean. It reflects the sensitivity of the model to the fluctuations in the training data.
3. **Irreducible Error**: Denoted by σ^2, this is the error that cannot be reduced by any model due to the inherent noise in the data.

Since, all three errors are non-negative, the irreducible error forms a lower bound on the expected error on test dataset.

2.3.1 Bias

Bias refers to the error introduced by approximating a real-world problem, which may be complex, by a simplified model. It's the difference between the average prediction of our model and the actual value we are trying to predict.

High Bias: When a model has high bias, it means the model is too simple to capture the underlying patterns in the data. This simplicity leads to systematic errors in predictions, known as underfitting. High bias is typically observed in models with low complexity, such as linear regression with insufficient features.
Low Bias: Conversely, low bias indicates that the model is sufficiently complex to capture the underlying

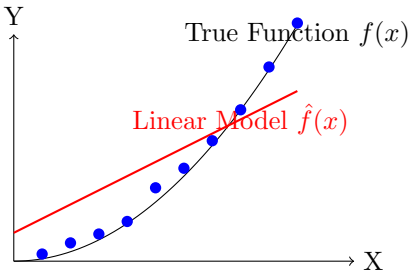


Figure 2.1: Illustration of High Bias (Underfitting).

patterns in the data. This complexity reduces systematic errors, but it doesn't necessarily mean the model is perfect.

Example: If you are using a linear regression model to fit a quadratic relationship, the model will likely have high bias because it cannot capture the curvature of the data, leading to systematic errors.

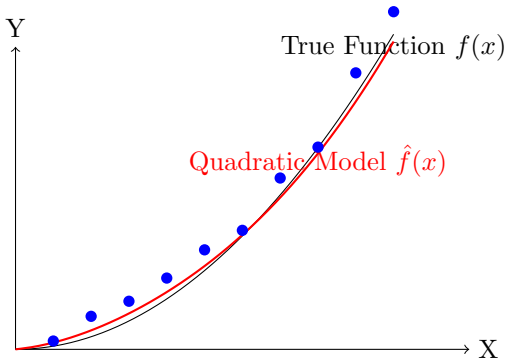


Figure 2.2: Illustration of Low Bias (Appropriate Fit).

2.3.2 Variance

Variance refers to the amount by which the model's predictions would change if we used a different training dataset. That is, it measures how scattered or inconsistent are the predicted values from the correct value due to different training data. It is also known as **Variance Error** or **Error due to Variance**.
High Variance: A model with high variance pays too much attention to the training data, capturing noise along with the underlying pattern. This overfitting results in a model that performs well on training data but poorly on new, unseen data.

Low Variance: Low variance indicates that the model's predictions do not vary significantly with different training datasets. Such models are more robust but might be too simple to capture all patterns in the data.

Example: A decision tree with many branches can capture intricate patterns in the training data, but it might also capture noise, leading to high variance.

2.3.3 Tradeoff

Thus, from the definition of bias and variance above, we have four possible combinations of bias and variance as depicted in figure 2.3.

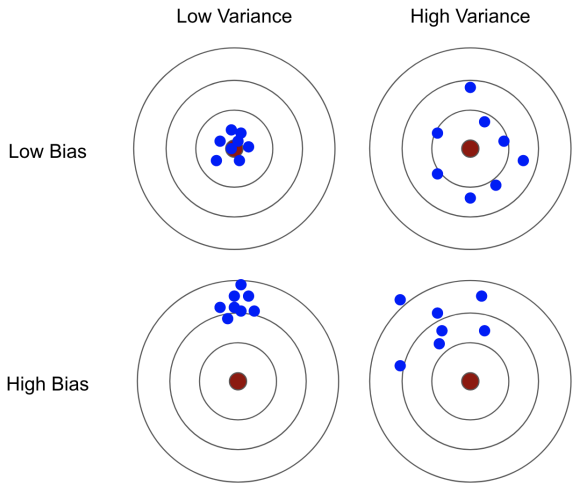


Figure 2.3: Possible combinations of Bias Variance

The goal in machine learning is to find a balance between bias and variance to minimize the total prediction error.

1. **Underfitting (High Bias, Low Variance):** Occurs when the model is too simple. It fails to capture the underlying trend of the data, resulting in high bias. Despite being stable across different

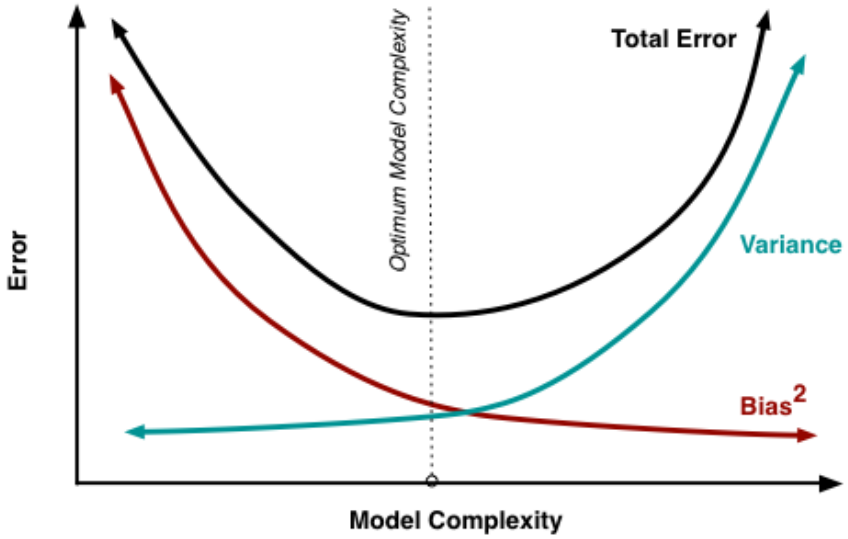


Figure 2.4: Bias Variance Tradeoff

datasets (low variance), it performs poorly overall.

2. **Overfitting (Low Bias, High Variance):** Happens when the model is too complex. It captures noise in the training data, leading to low bias but high variance. While it performs well on the training set, it generalizes poorly to new data.

3. **Optimal Model:** The best model achieves a good balance between bias and variance. This model is complex enough to capture the underlying patterns (low bias) but not too complex to overfit the noise (low variance).

This graph 2.4 illustrates that as model complexity increases along the x-axis, we see that bias (represented by the red curve) decreases, while variance (blue curve) increases. The total error (black curve) is the sum of these two components. On the left side of the graph, where models are simpler, we see high bias and low variance, indicating underfitting. As we move right towards more complex models, we encounter a region of optimal balance where the total error is minimized. Beyond this point, we enter the realm of overfitting, characterized by low bias but high variance. The graph demonstrates that the goal in model selection is to find the sweet spot of complexity that minimizes total error, balancing the tradeoff between bias and variance. This optimal point represents the best compromise between the model's ability to capture the underlying patterns in the data and its ability to generalize well to new, unseen data.

2.4 Locally Weighted Regression

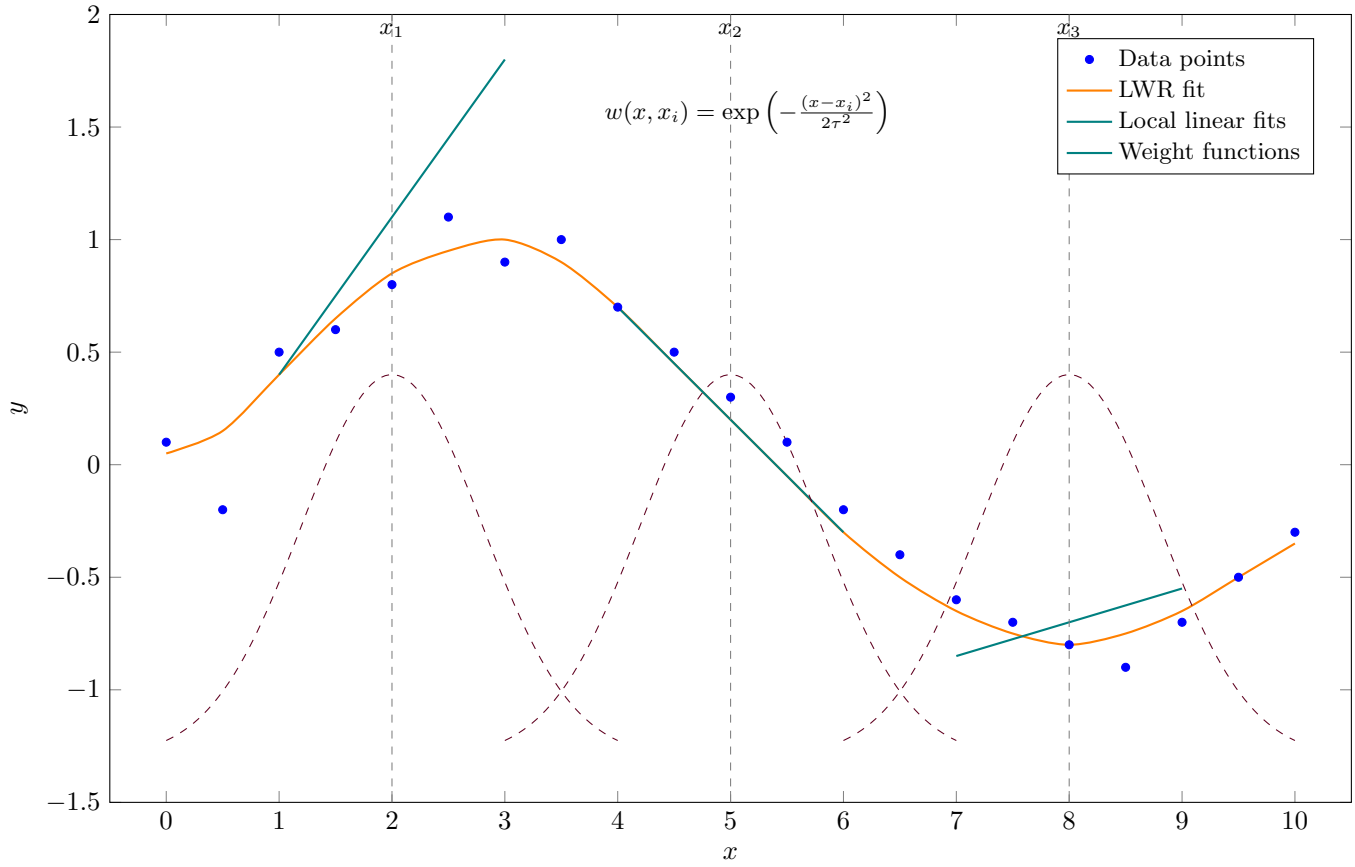
Locally weighted regression is a non-parametric method that performs regression around a point of interest using only the training data local to that point. It's particularly useful when the relationship between variables is non-linear.

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 lean 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.

Algorithm 3 Locally Weighted Regression using Gradient Descent

Require: Training data (X, y), query point x_q, kernel function K, bandwidth τ, learning rate α, num_iterations, convergence threshold ϵ
Ensure: Predicted value $\hat{y}_q$

function LocallyWeightedRegression($X, y, x_q, K, \tau, \alpha, \text{num\_iterations}, \epsilon$)
  $m \leftarrow$ number of training observations
  $n \leftarrow$ number of features
  $w \leftarrow \text{random}(n, 1)$ ▷ Initialize coefficients
  for $t = 1$ to num_iterations **do**
    $\eta \leftarrow \text{diag}(0, \ldots, 0)$ ▷ Re-initialize η for this iteration
    for $i = 1$ to m **do**
      $\eta_{ii} \leftarrow K\left(\frac{||x_i - x_q||}{\tau}\right)$ ▷ Calculate weights
    end for
    $\nabla J \leftarrow \frac{1}{m} X^T \eta (Xw - y)$ ▷ Compute gradient
    $w_{\text{new}} \leftarrow w - \alpha \nabla J$ ▷ Update coefficients
    if $||w_{\text{new}} - w|| < \epsilon$ **then**
      break ▷ Convergence check
    end if
    $w \leftarrow w_{\text{new}}$
  end for
  $\hat{y}_q \leftarrow x_q^T w$ ▷ Make prediction
  return $\hat{y}_q$
end function
function KernelFunction(u)
  return $\exp(-u^2/2)$ ▷ Gaussian kernel
end function

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.

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

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

$$\begin{aligned}
\frac{p}{1-p} &= \exp(z) \\
p &= \exp(z) - p \cdot \exp(z) \\
p + p \cdot \exp(z) &= \exp(z) \\
p(1 + \exp(z)) &= \exp(z) \\
p &= \frac{\exp(z)}{1 + \exp(z)}
\end{aligned}$$

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

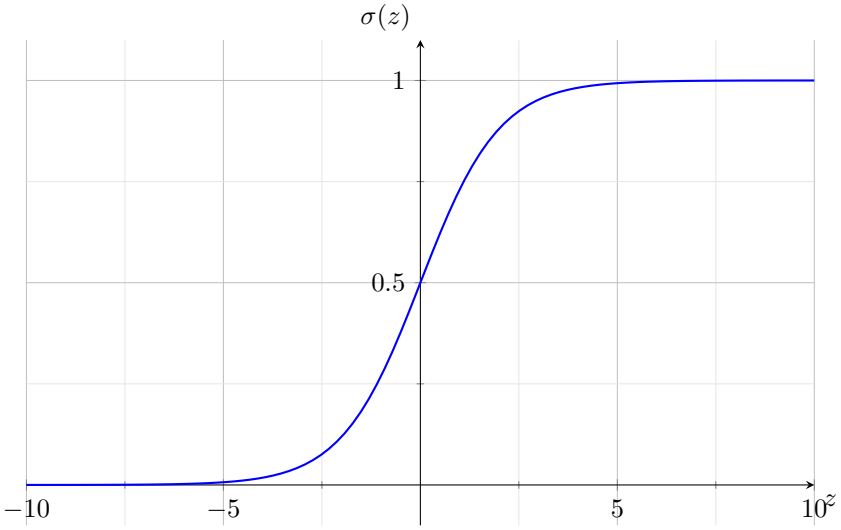
$$p = \frac{\exp(z)/\exp(z)}{(1+\exp(z))/\exp(z)}$$
$$p = \frac{1}{1/\exp(z)+1}$$
$$p = \frac{1}{\exp(-z)+1}$$
$$p = \frac{1}{1+\exp(-z)}$$

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:

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

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 α, num_iterations, convergence threshold ϵ
Ensure: Learned parameters θ

function LogisticRegression(X, y, α, num_iterations, ϵ)
  $m \leftarrow$ number of training examples
  $n \leftarrow$ number of features
  $\mathbf{w} \leftarrow \text{zeros}(n, 1)$ ▷ Initialize parameters
  for $t = 1$ to num_iterations **do**
    $z \leftarrow X\mathbf{w}$
    $\hat{y} \leftarrow \frac{1}{1+e^{-z}}$ ▷ Sigmoid function
    $\nabla J \leftarrow \frac{1}{m} X^T(\hat{y} - y)$ ▷ Compute gradient
    $\mathbf{w}_{\text{new}} \leftarrow \mathbf{w} - \alpha \nabla J$ ▷ Update parameters
    if $||\mathbf{w}_{\text{new}} - \mathbf{w}|| < \epsilon$ **then**
      break ▷ Convergence check
    end if
    $\mathbf{w} \leftarrow \mathbf{w}_{\text{new}}$
  end for
  return θ
end function

2.5.1 Numerical example

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 (α) = 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.

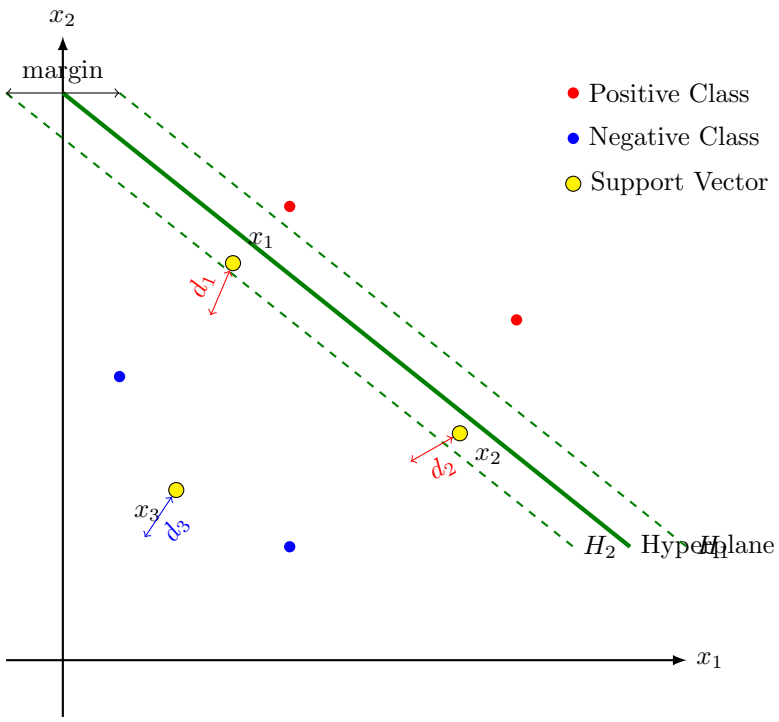
Consider a binary classification problem with a training dataset $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, where:

- $\mathbf{x}^{(i)} \in \mathbb{R}^n$ 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: $\hat{\gamma} = y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$

Linear Separability : We start by assuming that the data is linearly separable. The equation of a hyperplane in $\mathbb{R}^n$ is given by:

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

where $\mathbf{w} \in \mathbb{R}^n$ is the normal vector to the hyperplane and $b \in \mathbb{R}$ is the bias term.

Decision Function : The decision function for classifying a point $\mathbf{x}$ is:

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

For correct classification of all points, we require:

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

Margin Maximization : The geometric margin of a point $\mathbf{x}^{(i)}$ with respect to the hyperplane is given by:

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

The goal of SVM is to maximize the minimum margin over all training points:

$$\max_{\mathbf{w},b} \min_{i=1,\ldots,m} \gamma^{(i)} \tag{2.15}$$

Constraint Normalization: We can simplify our problem by normalizing the constraints. Hence, we choose the scale of $\mathbf{w}$ and b such that the point(s) closest to the hyperplane satisfy:

$$|\mathbf{w}^T\mathbf{x}_i + b| = 1 \tag{2.16}$$

This leads to the constraint:

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

We call this the **SVM constraint**.

Optimization Problem: With this normalization, maximizing the margin is equivalent to minimizing $\|\mathbf{w}\|$. For mathematical convenience, we minimize $\frac{1}{2}\|\mathbf{w}\|^2$. This leads to the primal optimization problem:

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

$$\text{subject to} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad \forall i = 1, \ldots, n \tag{2.19}$$

Lagrangian Formulation: To solve this constrained optimization problem, we introduce Lagrange multipliers $\alpha_i \geq 0$ and form the Lagrangian:

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

Karush-Kuhn-Tucker (KKT) Conditions : The KKT conditions for optimality are derived by setting the partial derivatives of the Lagrangian with respect to $\mathbf{w}$ and b to zero and incorporating the constraints:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0 \tag{2.21}$$

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.22}$$

$$\alpha_i \geq 0, \quad \forall i = 1, \ldots, n \tag{2.23}$$

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 \geq 0, \quad \forall i = 1, \ldots, n \tag{2.24}$$

$$\alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1] = 0, \quad \forall i = 1, \ldots, n \tag{2.25}$$

The above given first two conditions are derived from the stationarity condition, which ensures that the gradient of the Lagrangian with respect to the primal variables $(\mathbf{w}, b)$ is zero at the optimum. The third condition comes from the dual feasibility, ensuring non-negative Lagrange multipliers. The fourth condition is the primal feasibility, ensuring the constraints of the original problem are satisfied. The last condition, known as the complementary slackness condition, ensures that either the Lagrange multiplier α_i is zero or the constraint is active.

Dual Formulation: From the KKT conditions, we can express $\mathbf{w}$ in terms of α_i:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \tag{2.26}$$

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

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \tag{2.27}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{2.28}$$

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

Support Vectors: The points $\mathbf{x}_i$ for which $\alpha_i > 0$ are called support vectors. These are the points that lie exactly on the margin. From the KKT conditions, we can see that for these points:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1 \tag{2.30}$$

Decision Boundary: Once we have solved the dual problem for α_i, we can compute $\mathbf{w}$ using the equation from Section 10. To find b, we can use any support vector $\mathbf{x}_s$:

$$b = y_s - \mathbf{w}^T\mathbf{x}_s \tag{2.31}$$

The decision function for classifying new points becomes:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right) \tag{2.32}$$

Soft Margin SVM: In practice, data may not be linearly separable. We introduce slack variables $\xi_i \geq 0$ to allow for misclassification:

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \tag{2.33}$$

$$\text{subject to} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \forall i = 1, \ldots, n \tag{2.34}$$

$$\xi_i \geq 0, \quad \forall i = 1, \ldots, n \tag{2.35}$$

where $C > 0$ is a regularization parameter that controls the trade-off between margin maximization and misclassification.

2.7.1 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:

$$\max_{\boldsymbol{\alpha}} \quad \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{2.36}$$

$$\text{subject to} \quad \sum_{i=1}^{n}\alpha_i y_i = 0 \tag{2.37}$$

$$0 \leq \alpha_i \leq C, \quad \forall i = 1, \ldots, n \tag{2.38}$$

The decision function becomes:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n}\alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b\right) \tag{2.39}$$

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): $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$

Problem Statement: 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 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:

$$a \cdot s_1 \cdot s_1 + b \cdot s_2 \cdot s_1 + c \cdot s_3 \cdot s_1 = -1$$
$$a \cdot s_1 \cdot s_2 + b \cdot s_2 \cdot s_2 + c \cdot s_3 \cdot s_2 = 1$$
$$a \cdot s_1 \cdot s_3 + b \cdot s_2 \cdot s_3 + c \cdot s_3 \cdot s_3 = 1$$

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.

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

function SVM(X, y, C, α, n, K)
  Initialize $\alpha \leftarrow 0$, $b \leftarrow 0$
  for $i = 1$ to n **do**
    for each (x_j, y_j) in (X, y) **do**
      $E_j \leftarrow \sum_{i=1}^{m} \alpha_i y_i KERNEL(x_i, x_j, kernel_type) + b - y_j$
      $\alpha_j^{old} \leftarrow \alpha_j$
      $\alpha_j \leftarrow \text{clip}(\alpha_j - y_j E_j / K(x_j, x_j), 0, C)$
      $b \leftarrow b - E_j - y_j(\alpha_j - \alpha_j^{old})K(x_j, x_j)$
    end for
  end for
  return α, b
end function
function KERNEL(x_1, x_2, *kernel_type*)
  if *kernel_type* = 'linear' **then**
    return $x_1^T x_2$
  else if *kernel_type* = 'polynomial' **then**
    return $(1 + x_1^T x_2)^d$ ▷ d is the polynomial degree
  else if *kernel_type* = 'rbf' **then**
    return $\exp(-\gamma \|x_1 - x_2\|^2)$ ▷ γ is the RBF parameter
  end if
end function

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:

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

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's 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:

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{2.40}$$

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

Let $x \in 0, 1^n$ be a feature vector representing n binary features, and let $y \in 0, 1$ be the class label. We aim to find the probability $p(y|x)$, which is the probability of the class label given the feature vector. then we can write as:

$$\begin{aligned} p(y|x) &= \frac{p(x|y)p(y)}{p(x)} \\ &= \frac{p(x_1, \ldots, x_n|y)p(y)}{p(x)} \end{aligned}$$

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.
For dependent events, the joint probability can be expressed as:

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

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

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

Substituting this into Bayes' theorem, we get:

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

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:

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

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

$$\log \frac{p(y=1|x)}{p(y=0|x)} = \log \frac{p(y=1)}{p(y=0)} + \sum_{j=1}^{n} \log \frac{p(x_j|y=1)}{p(x_j|y=0)} \tag{2.41}$$

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, the conditional probabilities are:

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

Substituting these into the log-odds ratio expression 2.41 gives:

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

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

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

Naive Bayes Classifier For a feature vector $x=(x_1,\ldots,x_n)$ and a class variable y, the Naive Bayes classifier predicts:

$$\hat{y}=\arg\max_y P(y|x_1,\ldots,x_n) \tag{2.42}$$

Using Bayes' theorem, this can be written as:

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

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:

$$\phi_y=\frac{\sum_{i=1}^{m}\mathbf{1}\{y^{(i)}=1\}}{m} \tag{2.44}$$

$$\phi_{j|y=1}=\frac{\sum_{i=1}^{m}\mathbf{1}\{x_j^{(i)}=1 \text{ and } y^{(i)}=1\}}{\sum_{i=1}^{m}\mathbf{1}\{y^{(i)}=1\}} \tag{2.45}$$

$$\phi_{j|y=0}=\frac{\sum_{i=1}^{m}\mathbf{1}\{x_j^{(i)}=1 \text{ and } y^{(i)}=0\}}{\sum_{i=1}^{m}\mathbf{1}\{y^{(i)}=0\}} \tag{2.46}$$

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.

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. For each feature, the model estimates the mean and variance for each class. These parameters are then used to calculate the probability of a feature given the class using the Gaussian distribution formula.

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

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2.47}$$

   Where μ_y and σ_y^2 are 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 Multinomial Naive Bayes, the probability of observing a particular feature count vector given a class is computed using the multinomial distribution. For a given class y, the likelihood of a feature vector $mathbfx = (x_1, x_2, \ldots, x_n)$ is given by:

$$P(x \mid y) = P(y). \prod_{i=1}^{m} P(x_i \mid y)$$

   Where the probability of feature x_i given y is given as:

$$P(x_i|y) = \frac{\text{count}(x_i, y) + \alpha}{\sum_{j=1}^{n}(\text{count}(x_j, y) + \alpha)} \tag{2.48}$$

   Where α is a smoothing parameter.

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). This makes it particularly useful for text classification tasks, such as spam detection, where the presence or absence of certain words matters.

$$P(x_i|y) = P(i|y)x_i + (1 - P(i|y))(1 - x_i) \tag{2.49}$$

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, $K = 2$.

Algorithm 6 Naive Bayes Algorithm

procedure TRAIN(X, y)
    $N \leftarrow$ number of samples in X
    $features \leftarrow$ set of unique features in X
    $classes \leftarrow$ set of unique classes in y
    for each class c in $classes$ **do**
        $class\_count[c] \leftarrow$ count of samples with class c
        $P(c) \leftarrow class\_count[c]/N$
        for each feature f in $features$ **do**
            $feature\_count[f, c] \leftarrow$ count of samples with feature f and class c
            $P(f|c) \leftarrow (feature\_count[f, c] + 1)/(class\_count[c] + 2)$ ▷ Laplace smoothing
        end for
    end for
    return $P(c)$, $P(f|c)$ for all c and f
end procedure
procedure PREDICT($x, P(c), P(f|c)$)
    for each class c in $classes$ **do**
        $score[c] \leftarrow \log(P(c))$
        for each feature f in x **do**
            if f is present in x **then**
                $score[c] \leftarrow score[c] + \log(P(f|c))$
            else
                $score[c] \leftarrow score[c] + \log(1 - P(f|c))$
            end if
        end for
    end for
    return $\arg\max_c(score[c])$
end procedure
procedure NAIVEBAYES($X_{train}, y_{train}, X_{test}$)
    $P(c), P(f|c) \leftarrow$ TRAIN(X_{train}, y_{train})
    $predictions \leftarrow$ empty list
    for each sample x in X_{test} **do**
        $prediction \leftarrow$ PREDICT($x, P(c), P(f|c)$)
        Add $prediction$ to $predictions$
    end for
    return $predictions$
end procedure

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.

Numerical: 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:

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

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

Step 4: Compare Posteriors

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

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

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 7 Enhanced 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
Ensure: Predicted label or value $\hat{y}_q$ for $\mathbf{x}_q$

```
Initialize an empty list distances = []
for each (x_i, y_i) in D do
    d_i ← distance(x_q, x_i)                    ▷ e.g., Euclidean distance
    Append (d_i, y_i) to distances
end for
Sort distances in ascending order based on d_i
neighbors ← first k elements of distances
if classification task then
    ŷ_q ← mode(labels in neighbors)             ▷ Most frequent label
else if regression task then
    ŷ_q ← mean(values in neighbors)             ▷ Average value
end if
return ŷ_q
```

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:**

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

   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.

2.9.2 Example: KNN Classifier

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

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

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

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:

$$y = f(\mathbf{x}) = \sum_{j=1}^{J} c_j I(\mathbf{x} \in R_j) \tag{2.50}$$

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:

$$y = f(\mathbf{x}) = \sum_{j=1}^{J} \hat{y}_j I(\mathbf{x} \in R_j) \tag{2.51}$$

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:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2 p_i \tag{2.52}$$

  where p_i is the proportion of examples in class i.

- Information Gain for attribute A:

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v) \tag{2.53}$$

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

Algorithm 8 ID3 Decision Tree Algorithm

Require: Training data D, Feature set F, Target attribute *target*
Ensure: Decision Tree

```
function ID3(D, F, target)
    if all examples in D have the same target value then
        return leaf node with that target value
    else if F is empty then
        return leaf node with most common target value in D
    else
        best_feature ← arg max_{f∈F} InformationGain(D, f)
        tree ← new decision tree with root test best_feature
        for each value v of best_feature do
            D_v ← subset of D where best_feature has value v
            if D_v is empty then
                Add leaf node to tree with most common target value in D
            else
                subtree ← ID3(D_v, F \ {best_feature}, target)
                Add subtree to tree as child corresponding to best_feature = v
            end if
        end for
        return tree
    end if
end function
function InformationGain(D, feature)
    H(D) ← −Σ_{c∈C} p(c) log2 p(c)                    ▷ Entropy of dataset
    H(D|feature) ← Σ_{v∈values(feature)} (|D_v|/|D|) H(D_v)
    return H(D) − H(D|feature)
end function
function Predict(tree, example)
    if tree is a leaf node then
        return the target value of the leaf node
    else
        feature ← the test at the root of tree
        v ← value of feature in example
        subtree ← child of tree corresponding to feature = v
        return Predict(subtree, example)
    end if
end function
```

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:

$$GR(S, A) = \frac{IG(S, A)}{H_A(S)} \tag{2.54}$$

where

$$H_A(S) = -\sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \log_2 \left(\frac{|S_v|}{|S|} \right) \tag{2.55}$$

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:

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2 \tag{2.56}$$

where p_i is the proportion of examples in class i.

- Variance Reduction for regression:

$$Var(S) = \frac{1}{|S|} \sum_{i=1}^{|S|} (y_i - \bar{y})^2 \tag{2.57}$$

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

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:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \tag{2.58}$$

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

Example of Decision Tree with ID3 Algorithm

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'.

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

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

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

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

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

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

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

Feathers provide the least Information Gain (IG), indicating it is less effective for splitting the dataset compared to other features.

For Swim

Entropy $H(\text{Type}|\text{Swim} = \text{Yes})$:

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

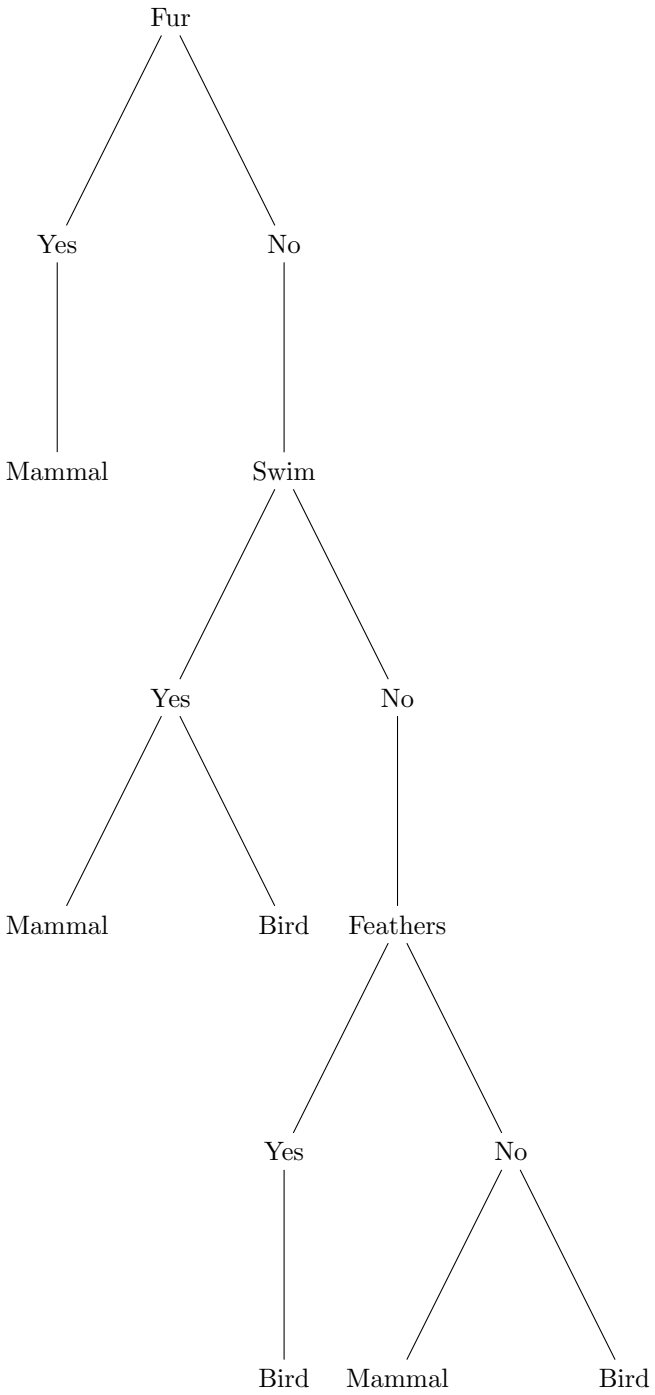
Entropy $H(\text{Type}|\text{Swim} = \text{No})$:

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

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

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:

$$h(x) = g(z_j) = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)} \tag{2.59}$$

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.

Softmax Regression Numerical Example

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. Generate multiple bootstrap samples from the original dataset.

2. Train a model on each bootstrap sample.

3. Aggregate the predictions from each model to form a final prediction.

Matehmatical Model of Bagging: Let's denote the original dataset by $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ is the feature vector and $y^{(i)} \in \mathbb{R}$ (for regression) or $y^{(i)} \in \{1, 2, \ldots, K\}$ (for classification) is the target variable.
Bootstrap sampling involves generating multiple datasets $\mathcal{D}_b$ by sampling m instances with replacement from $\mathcal{D}$. For each bootstrap sample $\mathcal{D}_b$, we have:

$$\mathcal{D}_b = \{(\mathbf{x}_b^{(i)}, y_b^{(i)})\}_{i=1}^{m},$$

where each $(\mathbf{x}_b^{(i)}, y_b^{(i)})$ is drawn from $\mathcal{D}$ with replacement.

For each bootstrap sample $\mathcal{D}_b$, a separate model $f_b(\mathbf{x})$ is trained. Let B be the total number of bootstrap samples. Thus, we have B models $\{f_b(\mathbf{x})\}_{b=1}^{B}$.
For each bootstrap sample $\mathcal{D}_b$, some instances from the original dataset $\mathcal{D}$ are not included in $\mathcal{D}_b$. These instances are referred to as out-of-bag samples. On average, about one-third of the original data points will be OOB for any given bootstrap sample.

The OOB samples can be used to estimate the prediction error of the bagged ensemble without the need for a separate validation set. The OOB error estimate is obtained as follows:

1. For each instance $(\mathbf{x}^{(i)}, y^{(i)})$ in the original dataset, aggregate the predictions from all models for which this instance was an OOB sample.

2. Compare the aggregated prediction to the true label $y^{(i)}$ to compute the error.

The OOB error provides an unbiased estimate of the generalization error of the bagged ensemble.

For Regression: The final prediction $\hat{y}$ for a new instance $\mathbf{x}$ is obtained by averaging the predictions from all models:

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

For Classification: The final prediction $\hat{y}$ for a new instance $\mathbf{x}$ is obtained by majority voting. Let $f_b(\mathbf{x})$ denote the predicted class label by the b-th model. The final predicted class $\hat{y}$ is given by:

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

where $\mathbb{I}(\cdot)$ is the indicator function, which is 1 if the argument is true and 0 otherwise.

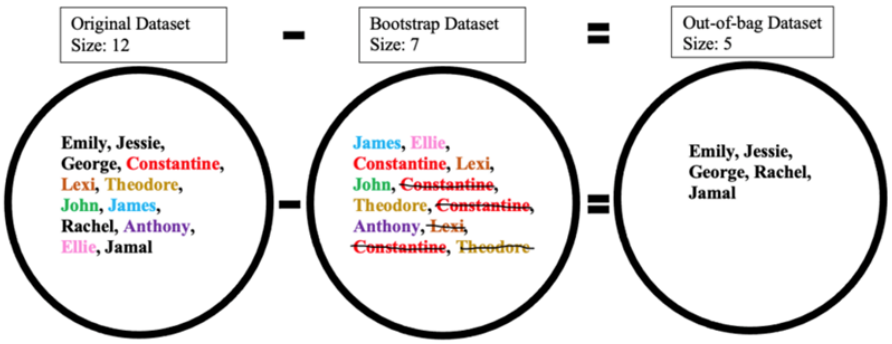


Figure 2.5: Out-of-bag Sample

2.12.2 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:

$$err = 1/N \sum_{i=1}^{N} I(y_i != C(x_i)) \tag{2.60}$$

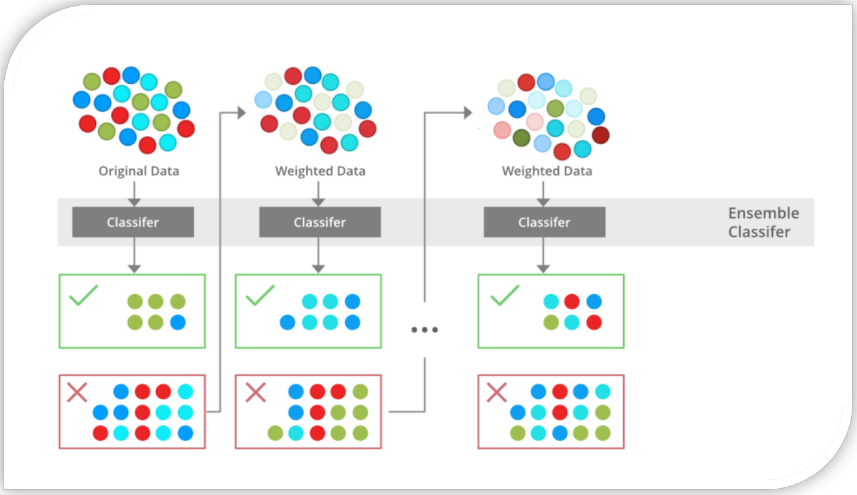


Figure 2.6: Illustration of Boosting

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

$$C(x) = sign(\sum_{k=1}^{K} \alpha_k C_k(x)) \tag{2.61}$$

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**:

$$w_i = \frac{1}{m} \quad \text{for } i = 1, \ldots, m \tag{2.62}$$

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**:

$$\epsilon_t = \sum_{i=1}^{m} w_i \cdot I(y_i \neq h_t(x_i)) \tag{2.63}$$

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

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{2.64}$$

     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**:

$$w_i \leftarrow w_i \cdot \exp(\alpha_t \cdot I(y_i \neq h_t(x_i))) \tag{2.65}$$

     Normalize the weights so that they sum to 1:

$$w_i \leftarrow \frac{w_i}{\sum_{j=1}^{m} w_j} \tag{2.66}$$

     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:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t \cdot h_t(x) \right) \tag{2.67}$$

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

2.12.3 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 $\mathbf{x}$:

$$z_j^{(i)} = h_j(\mathbf{x}^{(i)}) \quad \text{for } i = 1, \ldots, m \tag{2.68}$$

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:

$$H(\mathbf{z}^{(i)}) = H(z_1^{(i)}, z_2^{(i)}, \ldots, z_M^{(i)}) \tag{2.69}$$

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:

$$\hat{y} = H(h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_M(\mathbf{x})) \tag{2.70}$$

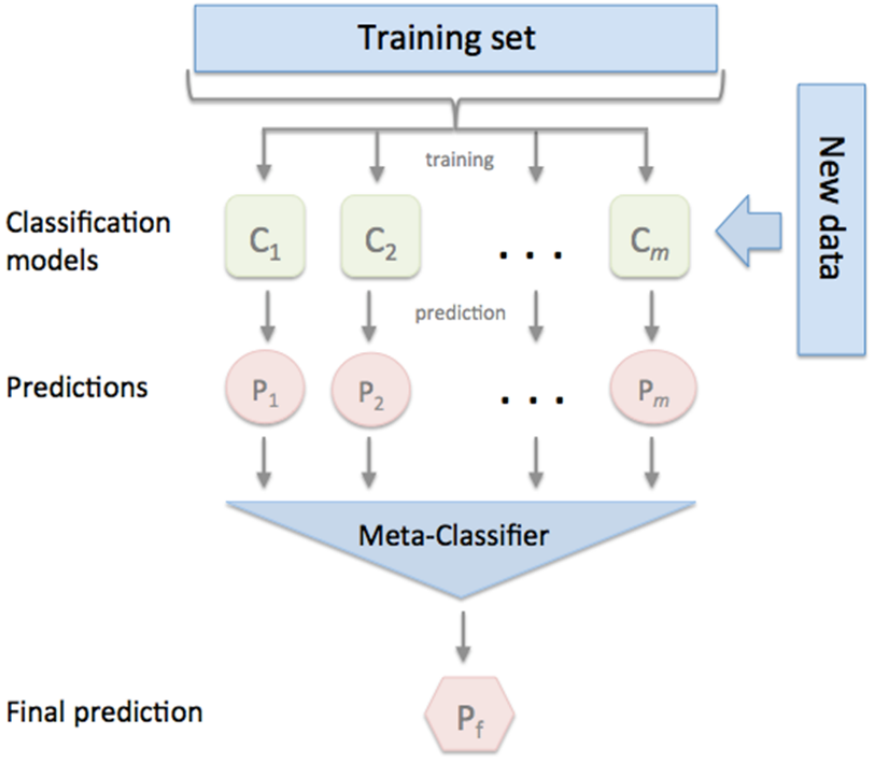


Figure 2.7: Illustration of Stacking

2.12.4 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

So far in our learning, a training example consisted of a set of input features and a label attached to each input record. Unsupervised Learning takes as training examples the set of attributes/features alone without a label. The purpose of unsupervised learning is to attempt to find natural partitions in the training set through learning from the features.

Two general strategies for Unsupervised learning include:

- Clustering
- Dimensionality Reduction

As unsupervised techniques don't use any labeled data, they learn intrinsic properties from the training data.

3.2 Clustering

The goal of clustering is to group observations that are similar to each other in an unsupervised manner. A cluster is a group of similar observations, i.e., the members within a cluster share some similar characteristics. A cluster is represented by a single point known as a centroid.

Examples of Clustering

- Example 1: Groups people of similar sizes together to make "small," "medium," and "large" T-Shirts.
  Problem: Tailor-made for each person is too expensive; one-size-fits-all does not fit all.
- Example 2: In marketing, segment customers according to their similarities for targeted marketing.
- Example 3: Given a collection of text documents, we want to organize them according to their content similarities to produce a topic hierarchy.
- Example 4: Given a collection of ecommerce transactions, cluster them into fraud and non-fraud credit card transactions to identify future fraud transactions.

Techniques for Measuring Similarity/Dissimilarity Clustering relies on the similarity/dissimilarity of data points. Similarity/dissimilarity is measured through various techniques such as:

- Distance
- Density
- Cosine Similarity

These techniques are used by various clustering algorithms depending on their nature. For example, the K-Means algorithm uses distance measures to find similarity, whereas the DBSCAN algorithm uses density. In NLP, the distance between two words or documents is measured using cosine similarity.

3.2.1 K-Means Clustering

K-Means Clustering is the most common and simple clustering algorithm. It is a partitioning-based algorithm that uses distance as a measure to find similarity.

Key Concepts

- **K:** Refers to the number of clusters expected in the given dataset. The value of K should be pre-known.
- **Means:** Refers to the averaging used to determine the centroid of a cluster.

The K clusters in K-means clustering are searched iteratively. Initially, K clusters are taken randomly, and the cluster is identified after several iterations. It is highly likely that different initial cluster centers lead to different results. Thus, it is better to select cluster centers far away from one another.

Algorithm 9 K-Means Algorithm

Input: Dataset X containing m observations and n features, number of clusters k
1. Randomly initialize k cluster centers $\mathbf{C} = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k\}$
repeat
    2. Assign each data point $\mathbf{x}_i \in X$ to the nearest cluster center:
    for $i = 1$ to m **do**
        $c^{(i)} \leftarrow \arg\min_j \|\mathbf{x}_i - \mathbf{c}_j\|$
        Assign $\mathbf{x}_i$ to cluster $c^{(i)}$
    end for

    3. Update each cluster center $\mathbf{c}_j$ to be the mean of all points assigned to cluster j:
    for $j = 1$ to k **do**
        $\mathbf{c}_j \leftarrow \frac{1}{|\{i:c^{(i)}=j\}|} \sum_{\mathbf{x}_i \in X, c^{(i)}=j} \mathbf{x}_i$
    end for
until no change in cluster assignments
Output: Cluster centers $\mathbf{C} = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k\}$

K-Means - Step by Step

1. Let X be the dataset containing m observations and n features.
2. Randomly select k cluster centers.
3. Calculate the distance between each data point and each cluster center.
4. Compare the distance of the point to each cluster center and assign the point to the cluster with the minimum distance.
5. If there is no change in set membership of the cluster, then terminate.
6. Else, recalculate the new cluster center using the mean of all the data points, called the centroid.
7. Repeat the process from step 3 to 7.

K-Means Clustering Example

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:

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

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:

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

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:

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

The final centroids are:

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

Random Initialization Using randomization to find the initial cluster center may result in different outcomes. The final clusters are highly dependent on how initial clusters are declared. To overcome this, we may use the K-means++ algorithm, an improvised version of K-Means.

K-Means++ Initialization Algorithm

1. Randomly select the first cluster center from the data points.
2. For each data point in the dataset, compute the distance with the previous cluster center.
3. Select the next cluster center from the data points which is farthest from the previously chosen cluster.
4. Repeat this process until K cluster centers have been sampled from the dataset.

Research on K-Medoids and K-Mode algorithms is advised as they are not within the scope of this syllabus.

3.2.2 Hierarchical Clustering

In hierarchical clustering, clusters are identified within a cluster itself, i.e., we tend to build an algorithm that finds a hierarchy of clusters.

Common Techniques

- **Agglomerative Clustering:** Uses a bottom-up approach, considering each observation in the dataset as a single cluster initially and gradually merging clusters as we move up the hierarchy tree until a single cluster covering the whole dataset is found.

Algorithm 10 Agglomerative Clustering Algorithm

Input: Dataset X containing m observations and n features

1. Initialize each data point $\mathbf{x}_i \in X$ as its own cluster: $\mathcal{C}_i = \{\mathbf{x}_i\},\ i = 1, 2, \ldots, m$

2. Compute the pairwise distance matrix $\mathbf{D}$ between all data points:
   $\mathbf{D}_{ij} = \text{distance}(\mathbf{x}_i, \mathbf{x}_j)$

repeat

   3. Find the closest pair of clusters $\mathcal{C}_a$ and $\mathcal{C}_b$ based on $\mathbf{D}$:
      $(\mathcal{C}_a, \mathcal{C}_b) = \arg\min_{\mathcal{C}_i, \mathcal{C}_j} \mathbf{D}_{ij}$

   4. Merge clusters $\mathcal{C}_a$ and $\mathcal{C}_b$ to form a new cluster:
      $\mathcal{C}_{ab} = \mathcal{C}_a \cup \mathcal{C}_b$

   5. Update the distance matrix $\mathbf{D}$ by recalculating distances to the new cluster $\mathcal{C}_{ab}$:
   for $i = 1$ to m **do**
      $\mathbf{D}_{i,ab} = \text{distance}(\mathcal{C}_{ab}, \mathcal{C}_i)$
   end for

   6. Update the clusters set: $\mathcal{C} = \mathcal{C} \setminus \{\mathcal{C}_a, \mathcal{C}_b\} \cup \{\mathcal{C}_{ab}\}$

until only one cluster $\mathcal{C}$ remains

Output: Hierarchical clustering dendrogram or cluster assignments

1. Consider each data point a cluster.
2. Compute the distance between each data point.
3. Repeat until K clusters remain:
4. Merge the two data points forming a single cluster.
5. Update the distance metrics using various techniques.

Agglomerative Clustering Example 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 11 Divisive Hierarchical Clustering Algorithm

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

1. Consider all data points as belonging to a single cluster.
2. Repeat until there is one data point in each cluster:
   (a) Split the dataset into clusters using algorithms such as K-Means, K-Medoids, etc.
   (b) Choose the best cluster among all possibilities.

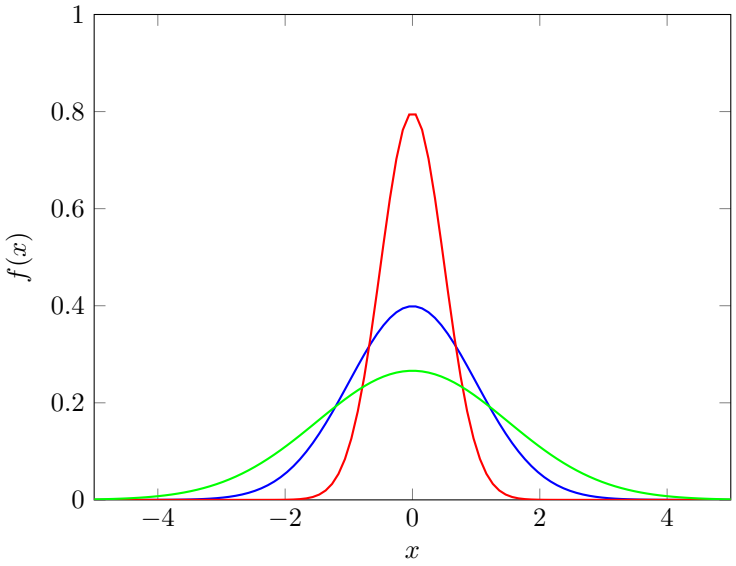
3.2.3 Gaussian Mixture Model (GMM)

A Gaussian Mixture Model (GMM) is a probabilistic model that assumes all data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. This model is particularly useful for clustering and density estimation in multi-dimensional spaces.

Gaussian Distribution

Before diving into GMMs, let's review the Gaussian (Normal) distribution:

- PDF: $f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
- Parameters: mean (μ) and standard deviation (σ)
- Symmetric, bell-shaped curve
- In multiple dimensions, characterized by mean vector and covariance matrix



Formulation of GMM

A Gaussian Mixture Model is a weighted sum of K Gaussian distributions:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

where:

- K is the number of Gaussian components
- π_k are the mixture weights (sum to 1)
- $\mathcal{N}(x|\mu_k, \Sigma_k)$ is the Gaussian distribution with mean μ_k and covariance Σ_k

Expectation-Maximization (EM) Algorithm

The EM algorithm is an iterative method to find maximum likelihood estimates of parameters in statistical models with latent variables. For GMMs, it's used to estimate the parameters π_k, μ_k, and Σ_k.

Steps in EM Algorithm

1. **Initialization:** Initialize the parameters π_k, μ_k, and Σ_k.
2. **Expectation (E) Step:** Compute the responsibility of each Gaussian for each data point:

$$\gamma_{nk} = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_n|\mu_j, \Sigma_j)}$$

3. **Maximization (M) Step:** Update the parameters:

$$N_k = \sum_{n=1}^{N} \gamma_{nk}$$

$$\pi_k^{new} = \frac{N_k}{N}$$

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} x_n$$

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} (x_n - \mu_k^{new})(x_n - \mu_k^{new})^T$$

4. Repeat steps 2-3 until convergence.

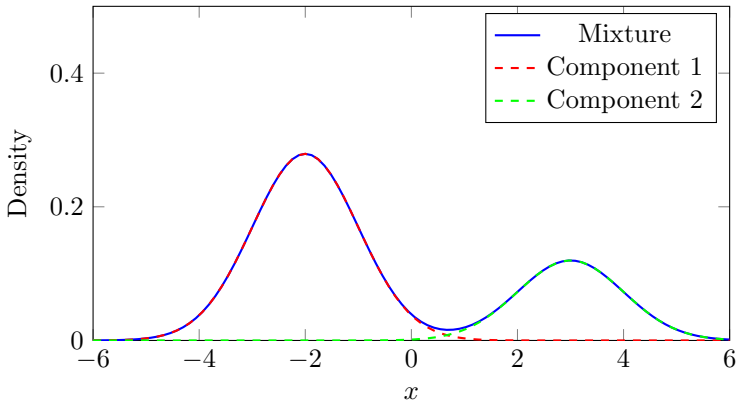
Algorithm 12 EM Algorithm for GMM

```
procedure EMGMM(X, K)
    Initialize π_k, μ_k, Σ_k for k = 1, ..., K
    while not converged do
        for n = 1 to N do
            for k = 1 to K do
                Compute γ_nk
            end for
        end for
        for k = 1 to K do
            Update N_k, π_k, μ_k, Σ_k
        end for
    end while
    return π_k, μ_k, Σ_k for k = 1, ..., K
end procedure
```

EM Algorithm Pseudocode

Example: 1D Gaussian Mixture

Consider a dataset generated from a mixture of two Gaussian distributions:



The EM algorithm would iteratively estimate the parameters of this mixture model:

- $\pi_1 \approx 0.7$, $\pi_2 \approx 0.3$
- $\mu_1 \approx -2$, $\mu_2 \approx 3$
- $\sigma_1 \approx \sigma_2 \approx 1$

Applications of GMMs

- **Clustering**: GMMs can be used for soft clustering, where each data point has a probability of belonging to each cluster. This is particularly useful when cluster boundaries are not well-defined.
- **Density Estimation**: GMMs provide a way to estimate complex probability distributions, useful in various machine learning and statistical applications.
- **Anomaly Detection**: By modeling the normal behavior of a system with a GMM, anomalies can be detected as data points with low likelihood under the model.
- **Speech Recognition**: GMMs have been widely used in speech recognition systems to model the distribution of speech features.
- **Image Segmentation**: In computer vision, GMMs can be used to model the color distribution in images for segmentation tasks.

3.3 Dimensionality Reduction

Higher the dimension of data, the more the model complexity is, and the more difficult it is to train the model. Increasing the features will not always improve classification accuracy. In the machine learning world, the "Curse of Dimensionality" is a common phrase which basically refers to the point where having more dimensions in data doesn't help in the conclusion; rather, it would be a hindrance. The purpose of dimensionality reduction is to reduce the number of dimensions or features from the dataset without distorting the pattern from the data we are trying to learn.

However, reducing the features always comes with a cost and may impact the accuracy. Thus, it may be challenging to reduce the features without any significant loss in accuracy. Dimensionality reduction deals with the process of reducing the number of variables under consideration by obtaining a smaller set of variables.

There are many common methods for dimensionality reduction such as:

- Principal Component Analysis (PCA)
- Feature Selection
- Low Variance Filter
- Linear Discriminant Analysis (LDA)

3.3.1 Principal Component Analysis (PCA)

PCA is a dimensionality reduction method often used to reduce the dimensionality of large datasets. This reduces the number of features by transforming existing datasets into completely new datasets containing a small number of features while preserving most of the information from the original datasets.

Algorithm 13 Principal Component Analysis (PCA)

procedure PCA(X, k)
  Input: $X \in \mathbb{R}^{n \times d}$: Dataset with n samples and d features

  Output:
  k: Number of principal components to retain
  Principal components: List of principal components
  Projected data: Data projected onto the principal components

  ▷ Standardize the data
  for $j = 1$ to d **do**
    $\mu_j \leftarrow \frac{1}{n} \sum_{i=1}^{n} X_{ij}$
    $\sigma_j \leftarrow \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (X_{ij} - \mu_j)^2}$
    for $i = 1$ to n **do**
      $X_{ij} \leftarrow \frac{X_{ij} - \mu_j}{\sigma_j}$
    end for
  end for

  ▷ Compute the covariance matrix
  $\Sigma \leftarrow \frac{1}{n-1} X^T X$

  ▷ Compute eigenvectors and eigenvalues of the covariance matrix
  $\{v_1, \ldots, v_d\}, \{\lambda_1, \ldots, \lambda_d\} \leftarrow \text{EigenDecomposition}(\Sigma)$

  ▷ Sort eigenvectors by decreasing eigenvalues
  Sort $\{v_1, \ldots, v_d\}$ and $\{\lambda_1, \ldots, \lambda_d\}$ in descending order of λ_i

  ▷ Select top k eigenvectors
  $W \leftarrow [v_1\ v_2\ \cdots\ v_k]$

  ▷ Project the data onto the new subspace
  $Y \leftarrow XW$
  Output: k, W, Y
end procedure

PCA - step by step

Let's understand the above algorithm step by step:

1. Consider a dataset.
2. Standardize each column in the dataset.
3. Compute the covariance matrix as:

$$\Sigma = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \mu)(X_i - \mu)^T$$

where Σ is the covariance matrix, X_i are the data points, and μ is the mean of the data points.

Note that the covariance matrix is symmetric, i.e., entries are reflections across the diagonal of the matrix.

Why Covariance? Covariance matrix is a tabular structure that summarizes the correlations between all possible pairs of features.

4. Compute the eigenvalues and eigenvectors from the covariance matrix.
5. Compute eigenvalues and eigenvectors from:

$$\Sigma\mathbf{v} = \lambda\mathbf{v}$$

where λ gives the eigenvalues and $\mathbf{v}$ gives the eigenvectors. By ranking the eigenvectors in order of their eigenvalues from highest to lowest, you get the principal components in order of significance.

The eigenvectors of the covariance matrix belonging to the largest eigenvalues are actually the directions of the axes where there is the most information (i.e., they explain the most variance from data). This is the first principal component $\mathbf{v}_1$, and so on.

6. After having the principal components, to compute the percentage of information accounted for by each component, divide the eigenvalue of each component by the sum of eigenvalues.
7. Discard the eigenvector $\mathbf{v}_2$ which is the one of lesser significance and form a feature vector with $\mathbf{v}_1$ only.
8. Reorient the data from their original axes to the ones you have calculated from the principal components.

Properties of Principal Components

- The new features generated using PCA are distinct, i.e., the covariance between the new features is 0.
- The principal components are generated in order of the variance explained by them. Hence, the first principal component captures the maximum variance, the second one captures the next highest variability, and so on.

Variations of PCA

Kernel PCA : Kernel PCA extends the idea of PCA to non-linear transformations of the data. Use the "kernel trick" to implicitly map the data to a higher-dimensional space where linear PCA can be more effective.

Algorithm Overview:

1. Choose a kernel function $K(x_i, x_j)$ (e.g., Gaussian kernel: $K(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$)
2. Compute the kernel matrix $K_{ij} = K(x_i, x_j)$
3. Center the kernel matrix: $\tilde{K} = K - 1_n K - K 1_n + 1_n K 1_n$
4. Perform eigendecomposition on $\tilde{K}$

5. Project data using the top eigenvectors

Advantages:

- Can capture non-linear patterns in the data
- Useful for datasets with complex, non-linear structures

Limitations:

- Choice of kernel function can significantly affect results
- Computationally expensive for large datasets

Incremental PCA Incremental PCA is designed to handle large datasets that cannot fit into memory all at once. Update the PCA model incrementally as new data arrives, without having to store or process the entire dataset at once.

Implementation:

1. Initialize PCA model with a small batch of data
2. For each new batch of data:
   - Project the new data onto the current principal components
   - Compute the residual (orthogonal to current PCs)
   - Update the PCA model using the residual
3. Repeat until all data is processed

Advantages:

- Memory-efficient for large datasets
- Can handle streaming data

Limitations:

- item May not converge to the exact same result as batch PCA
- item Sensitive to the order of data presentation

Sparse PCA Sparse PCA adds a sparsity constraint to the PCA problem, encouraging the algorithm to find a lower-dimensional representation with fewer non-zero components.

Find principal components that are sparse (have many zero entries) while still explaining most of the variance in the data.

Optimization Problem:

$$\min_{U,V} \|X - UV^T\|_F^2 + \alpha\|V\|_1$$

where $\|V\|_1$ is the L1 norm of V, promoting sparsity, and α is a regularization parameter.

Advantages:

- Improved interpretability of principal components
- Can perform feature selection implicitly

Limitations:

- Non-convex optimization problem
- Choice of regularization parameter can be challenging

Robust PCA Robust PCA is designed to handle datasets with outliers or noise by separating the data into a low-rank component and a sparse component. Decompose the data matrix X into a low-rank matrix L and a sparse matrix S, where S captures outliers or corruptions.

$$\min_{L,S} \text{rank}(L) + \lambda \|S\|_0 \quad \text{subject to} \quad X = L + S$$

where $\|S\|_0$ is the L0 norm (number of non-zero entries) of S, and λ is a trade-off parameter.

Advantages:

- Robust to outliers and gross corruptions in the data
- Can separate structured (low-rank) and unstructured (sparse) components of the data

Limitations:

- Computationally more expensive than standard PCA
- May struggle with datasets where outliers form a low-rank structure

Randomized PCA Randomized PCA is designed to approximate the first k principal components of a large dataset efficiently using random projections.

Key Idea: Use random projections to approximate the range of the data matrix, then perform PCA in this approximate range.

Algorithm Overview:

1. Generate a random matrix $\Omega \in \mathbb{R}^{n \times (k+p)}$, where p is an oversampling parameter
2. Compute $Y = X\Omega$
3. Compute QR decomposition of Y: $Y = QR$
4. Compute $B = Q^T X$
5. Perform SVD on B: $B = \tilde{U}\Sigma V^T$
6. Compute $U = Q\tilde{U}$

Advantages:

- Significantly faster than full PCA for large datasets
- Can handle data matrices that are too large to fit in memory

Limitations:

- Provides an approximation rather than exact PCA
- Performance can depend on the random seed used

Comparison of PCA Variations

Variation	Best Used When	Key Advantage
Kernel PCA	Non-linear patterns in data	Can capture complex relationships
Incremental PCA	Large or streaming datasets	Memory-efficient
Sparse PCA	Interpretability is crucial	Provides sparse components
Robust PCA	Data contains outliers	Separates structured and unstructured components
Randomized PCA	Very large datasets	Computationally efficient

3.3.2 Low Rank Approximations

Low-rank approximations (LRA) are a powerful technique used in data analysis, machine learning, and signal processing to reduce the dimensionality of high-dimensional data. The main idea is to find a matrix B that has a lower rank than a given matrix A but approximates A as closely as possible.

Given a matrix $A \in \mathbb{R}^{m \times n}$ of rank r, we seek to find a matrix $B \in \mathbb{R}^{m \times n}$ of rank $k < r$ that minimizes the difference between A and B according to some norm, typically the Frobenius norm:

$$\min_{B:\text{rank}(B) \leq k} \|A - B\|_F$$

The motivation behind LRA includes:

- Data compression: Reducing storage requirements for large datasets.
- Noise reduction: Separating signal from noise in data.
- Feature extraction: Identifying the most important components of data.
- Computational efficiency: Speeding up calculations involving large matrices.

The rank of a matrix is determined by the number of linearly independent rows or columns in the matrix. Mathematically, we can express a rank-k approximation of A as:

$$A \approx B = XY^T$$

where $X \in \mathbb{R}^{m \times k}$ and $Y \in \mathbb{R}^{n \times k}$. This decomposition reduces the number of parameters from mn to $k(m + n)$, which can be a significant reduction when $k \ll \min(m, n)$.

3.3.3 Singular Value Decomposition (SVD)

Singular Value Decomposition is a fundamental matrix factorization method that provides the best low-rank approximation of a matrix in terms of the Frobenius norm.

Let $A \in \mathbb{R}^{m \times n}$ with $\text{rank}(A) = r$. Then there exist orthogonal matrices $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ such that:

$$A = U\Sigma V^T$$

where $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix with non-negative real numbers on the diagonal, called singular values, in descending order: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$.

Properties of SVD

1. The columns of U are called left singular vectors and form an orthonormal basis for the column space of A. 2. The columns of V are called right singular vectors and form an orthonormal basis for the row space of A. 3. The singular values σ_i represent the "importance" of each singular vector pair in reconstructing the original matrix.

Geometric Interpretation

SVD can be interpreted geometrically as a composition of three transformations:

1. Rotation/reflection (V^T)
2. Scaling (Σ)
3. Another rotation/reflection (U)

This interpretation helps in understanding how SVD captures the essential structure of the matrix.

Low Rank Approximation using SVD

The Eckart-Young theorem states that the best rank-k approximation of A in terms of the Frobenius norm is given by:

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

where u_i and v_i are the i-th columns of U and V respectively, and σ_i is the i-th singular value.

Algorithm for Computing SVD

There are several algorithms for computing the SVD. Here's a basic outline of the power iteration method:

Algorithm 14 Power Iteration Method for SVD

```
procedure ComputeSVD(A, k)
    Initialize V_k randomly
    repeat
        U_k ← AV_k
        U_k ← orthonormalize(U_k)
        V_k ← A^T U_k
        V_k ← orthonormalize(V_k)
    until convergence
    Σ_k ← U_k^T A V_k
    return U_k, Σ_k, V_k
end procedure
```

This algorithm iteratively refines estimates of the left and right singular vectors until convergence.

Applications of Low Rank Approximations and SVD

- **Principal Component Analysis (PCA)**: PCA is a special case of SVD when the data is centered. It's used for dimensionality reduction and feature extraction.
- **Image Compression**: By keeping only the top k singular values and vectors, we can compress images while retaining most of the important information.
- **Recommendation Systems**: In collaborative filtering, SVD can be used to factorize the user-item interaction matrix into lower-dimensional latent factor matrices.
- **Noise Reduction**: By truncating the SVD to keep only the top singular values, we can often separate signal from noise in various types of data.

Example: Image Compression using SVD

3.3.4 Latent Semantic Analysis (LSA)

Latent Semantic Analysis is a powerful technique in natural language processing that leverages Singular Value Decomposition to uncover hidden (latent) relationships between words and documents in a large corpus of text.

LSA is primarily used for:

- Concept searching: Finding documents that are conceptually similar, even if they don't share exact words.
- Automated document categorization: Grouping documents based on their semantic content.
- Information retrieval: Improving search results by understanding the contextual meaning of words.

– Text summarization: Identifying the most important concepts in a document.

The key insight of LSA is that words that are used in similar contexts tend to have similar meanings. By analyzing the co-occurrence patterns of words across documents, LSA can infer semantic relationships.

Consider a collection of documents where:

– m is the number of unique words in the collection
– n is the number of documents

We construct a term-document matrix D of dimension $m \times n$, where each element D_{ij} represents the importance of term i in document j. This importance is often represented by the term frequency-inverse document frequency (TF-IDF) weight.

Applying SVD to this matrix:

$$D = U\Sigma V^T$$

where:

– U (dimension $m \times m$) represents the distribution of words across the different latent semantic concepts.
– Σ (dimension $m \times n$) is the diagonal matrix that contains the strength of each latent semantic concept.
– V^T (dimension $n \times n$) represents the distribution of latent semantic concepts across the different documents.

LSA Process

The LSA process can be broken down into several steps:

1. **Corpus Preparation**: Collect and preprocess the documents (e.g., tokenization, stop-word removal).
2. **Term-Document Matrix Creation**: Construct the matrix D using TF-IDF weights.
3. **SVD Decomposition**: Apply SVD to get U, Σ, and V^T.
4. **Dimensionality Reduction**: Keep only the top k singular values and their corresponding singular vectors.
5. **Latent Space Transformation**: Project documents and queries into this reduced "semantic" space.
6. **Similarity Computation**: Compute similarities between documents or between documents and queries in this reduced space.

Dimensionality Reduction in LSA

A key aspect of LSA is dimensionality reduction. By keeping only the top k singular values, we create a low-rank approximation of the original term-document matrix:

$$D_k = U_k \Sigma_k V_k^T$$

This reduced representation captures the most important semantic relationships while filtering out noise. The choice of k is crucial:

– Too small: Important semantic information may be lost.
– Too large: Noise and idiosyncrasies of the specific corpus may be retained.

Typically, k is chosen to be much smaller than the number of terms or documents, often in the range of 100-300 for large corpora.

Example: Document Similarity

Let's consider a small example to illustrate how LSA can be used to find similar documents:

1. Suppose we have three 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. We create a term-document matrix (using simple term frequency for simplicity):

	Doc 1	Doc 2	Doc 3
the	2	2	2
cat	1	0	0
sat	1	0	0
on	1	1	0
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 to 2 dimensions.
4. In this reduced space, we might find that Doc 1 and Doc 2 are closer to each other than to Doc 3, reflecting their semantic similarity (both about animals resting on floor coverings).

Advantages and Limitations of LSA

Advantages:

- Captures semantic relationships beyond simple keyword matching.
- Handles synonymy (different words, same meaning) and polysemy (same word, different meanings) to some extent.
- Reduces noise and sparsity in the data.

Limitations:

- Assumes a bag-of-words model, losing word order information.
- Cannot capture negations or complex linguistic structures.
- The choice of the number of dimensions (k) can be arbitrary and affect results.

3.3.5 Canonical Correlation Analysis

Canonical Correlation Analysis (CCA) is a statistical method used to understand the relationship between two sets of variables. It aims to find linear combinations of the variables in each set such that the correlation between the two sets is maximized.

Given two sets of variables $\mathbf{X} \in \mathbb{R}^{n \times p}$ 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 finds vectors $\mathbf{a}$ and $\mathbf{b}$ such that the correlation between $\mathbf{u} = \mathbf{Xa}$ and $\mathbf{v} = \mathbf{Yb}$ is maximized.

$$\max_{\mathbf{a},\mathbf{b}} \text{corr}(\mathbf{Xa}, \mathbf{Yb})$$

This is equivalent to solving the following optimization problem:

$$\max_{\mathbf{a},\mathbf{b}} \frac{\mathbf{a}^T \mathbf{X}^T \mathbf{Y} \mathbf{b}}{\sqrt{\mathbf{a}^T \mathbf{X}^T \mathbf{X} \mathbf{a}} \sqrt{\mathbf{b}^T \mathbf{Y}^T \mathbf{Y} \mathbf{b}}}$$

Key Concepts

- **Canonical Variables:** These are the linear combinations of the original variables that maximize the correlation between the two sets.
- **Canonical Correlation:** The correlation between the canonical variables.
- **Canonical Weights:** The coefficients of the linear combinations that form the canonical variables.

Algorithm 15 Canonical Correlation Analysis (CCA)

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

Center the data:
$\bar{X} = X - \mathbf{1}_n \cdot \bar{X}$, where $\bar{X} = \frac{1}{n} \sum_{i=1}^{n} X_i$
$\bar{Y} = Y - \mathbf{1}_n \cdot \bar{Y}$, where $\bar{Y} = \frac{1}{n} \sum_{i=1}^{n} Y_i$

Compute the cross-covariance matrices:
$C_{XX} = \frac{1}{n-1} \bar{X}^T \bar{X}$
$C_{YY} = \frac{1}{n-1} \bar{Y}^T \bar{Y}$
$C_{XY} = \frac{1}{n-1} \bar{X}^T \bar{Y}$

Compute the generalized eigenvalue problem:
$\mathbf{C}_{XX}^{-1} \mathbf{C}_{XY} \mathbf{C}_{YY}^{-1} \mathbf{C}_{YX} \mathbf{W} = \mathbf{W} \mathbf{\Lambda}$
where $\mathbf{W}$ are the canonical vectors and $\mathbf{\Lambda}$ are the canonical correlations

Output: Canonical vectors $\mathbf{W}$ and canonical correlations $\mathbf{\Lambda}$

The steps to perform CCA are as follows:

1. Standardize the variables in $\mathbf{X}$ and $\mathbf{Y}$.
2. Compute the covariance matrices $\mathbf{C}_{XX}$, $\mathbf{C}_{YY}$, and $\mathbf{C}_{XY}$.
3. Solve the eigenvalue problems for $\mathbf{C}_{XX}^{-1} \mathbf{C}_{XY} \mathbf{C}_{YY}^{-1} \mathbf{C}_{YX}$ and $\mathbf{C}_{YY}^{-1} \mathbf{C}_{YX} \mathbf{C}_{XX}^{-1} \mathbf{C}_{XY}$.
4. The eigenvectors corresponding to the largest eigenvalues are the canonical weights $\mathbf{a}$ and $\mathbf{b}$.
5. Compute the canonical variables $\mathbf{u} = \mathbf{X}\mathbf{a}$ and $\mathbf{v} = \mathbf{Y}\mathbf{b}$.

Applications

CCA is used in various fields such as:

- **Neuroscience:** To analyze the relationship between different types of brain activities.
- **Genomics:** To understand the relationship between gene expression and other biological variables.
- **Social Sciences:** To study the relationships between sets of social and economic indicators.

CCA - Numerical Example

Consider two sets of variables with three observations each:

$$\mathbf{X} = \begin{pmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{pmatrix}, \quad \mathbf{Y} = \begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{pmatrix}$$

First, we standardize the variables in $\mathbf{X}$ and $\mathbf{Y}$:

$$\mathbf{X}_{\text{std}} = \begin{pmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{Y}_{\text{std}} = \begin{pmatrix} -1 & -1 \\ 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Next, we compute the covariance matrices:

$$\mathbf{C}_{XX} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{C}_{YY} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{C}_{XY} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

We then solve the eigenvalue problems:

$$\mathbf{C}_{XX}^{-1}\mathbf{C}_{XY}\mathbf{C}_{YY}^{-1}\mathbf{C}_{YX} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{C}_{YY}^{-1}\mathbf{C}_{YX}\mathbf{C}_{XX}^{-1}\mathbf{C}_{XY} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

The eigenvectors corresponding to the largest eigenvalues are the canonical weights:

$$\mathbf{a} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Finally, we compute the canonical variables:

$$\mathbf{u} = \mathbf{X}_{\text{std}}\mathbf{a} = \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix}, \quad \mathbf{v} = \mathbf{Y}_{\text{std}}\mathbf{b} = \begin{pmatrix} -2 \\ 0 \\ 2 \end{pmatrix}$$

The canonical correlation is the correlation between $\mathbf{u}$ and $\mathbf{v}$, which is 1 in this case, indicating a perfect linear relationship between the two sets of variables.

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).

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.

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. For instance, when evaluating a regression problem, one might choose from metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), or Mean Absolute Percentage Error (MAPE).

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.

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:

$$Error\,(e) = Actual\,value\,(y) - predicted\,value\,(\hat{y})$$

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 is then simply the average of these individual errors:

$$Mean\,Error\,(e) = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})$$

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.
2. The error may be skewed towards large positive values.
3. The errors may cancel each other out, resulting in a mean close to zero even if individual errors are large.

To address these issues, we need more sophisticated error metrics.

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:

$$Total\,Squared\,Error\,(E) = \sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$$

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. 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.

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:

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2}$$

RMSE has the advantage of being in the same units as the target variable, making it more interpretable. It's particularly useful for scaling down the errors closer to the actual values. Like MSE, RMSE is also sensitive to outliers, but to a lesser extent.

4.2.4 R-Squared (R^2)

Another commonly used metric in regression is R-Squared (R^2). R-Squared measures the proportion of variance in the dependent variable that is predictable from the independent variable(s). It's calculated as:

$$R^2 = 1 - \frac{variance(model)}{variance(average)}$$

Where:

$$variance(model) = \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$

And:

$$variance(average) = \sum_{i=1}^{m} (y^{(i)} - \bar{y}^{(i)})^2$$

R-Squared provides a comparison of how much better the regression line is compared to a simple mean line. The value of R^2 ranges from 0 to 1, with values closer to 1 indicating a better fit. Generally, an R^2 value above 0.5 is considered acceptable for most regression problems, though this can vary depending on the specific domain and application.

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:

$$Mean\,Absolute\,Error\,(MAE) = \frac{1}{m} \sum_{i=1}^{m} |y^{(i)} - \hat{y}^{(i)}|$$

MAE is less sensitive to outliers compared to MSE and RMSE, making it a robust metric when dealing with datasets that may contain anomalies.

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.

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. 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)), the confusion matrix has four categories:

- True Positive (TP): Cases where the model correctly predicted the positive class.
- False Positive (FP): Cases where the model incorrectly predicted the positive class.
- False Negative (FN): Cases where the model incorrectly predicted the negative class.
- True Negative (TN): Cases where the model correctly predicted the negative class.

	Predicted	
Actual	True	False
True	True Positive (TP)	False Negative (FN)
False	False Positive (FP)	True Negative (TN)

Table 4.1: Confusion Matrix

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.

Consider the following confusion matrix for a disease prediction task: The accuracy for this model

	Predicted	
Actual	Ill	Not Ill
Ill	48	2
Not Ill	5	45

Table 4.2: Disease Prediction Confusion Matrix

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, 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.

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.

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations:

$$Precision = \frac{TP}{TP + FP}$$

Recall (also known as sensitivity or true positive rate) is the ratio of correctly predicted positive observations to all actual positive observations:

$$Recall = \frac{TP}{TP + FN}$$

The value of both Precision and Recall is evluated within 0 and 1.

For our disease prediction example:

$$Precision = \frac{48}{48 + 5} = 0.90$$

$$Recall = \frac{48}{48 + 2} = 0.96$$

Precision tells us that when our model predicts a patient is ill, it's correct 90% of the time. Recall tells us that our model correctly identifies 96% of all ill patients.

The choice between optimizing for precision or recall depends on the specific problem. In some cases, like spam detection, we might prioritize precision to avoid marking legitimate emails as spam. In other cases, like disease screening, we might prioritize recall to ensure we don't miss any positive cases, even if it means having some false positives.

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.

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.

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:**
  - May lead to unnecessary actions, such as additional tests, treatments, or interventions.
  - In contexts like spam detection, a false positive might mark a legitimate email as spam.
  - In security systems, a false positive might cause unwarranted alarms and responses.
- **False Negatives:**
  - Can result in missed detections, such as not identifying a disease that needs treatment.
  - In fraud detection, a false negative might let fraudulent transactions pass unnoticed.
  - In safety systems, a false negative might overlook a critical threat, leading to potential harm.

4.3.4 F1 Score

The F1 score is the harmonic mean of precision and recall, providing a single score that balances both metrics:

$$F1\,Score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} = \frac{2 * Precision * Recall}{Precision + Recall}$$

The F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. It's particularly useful when you have an uneven class distribution and you want to seek a balance between precision and recall.

Example

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.

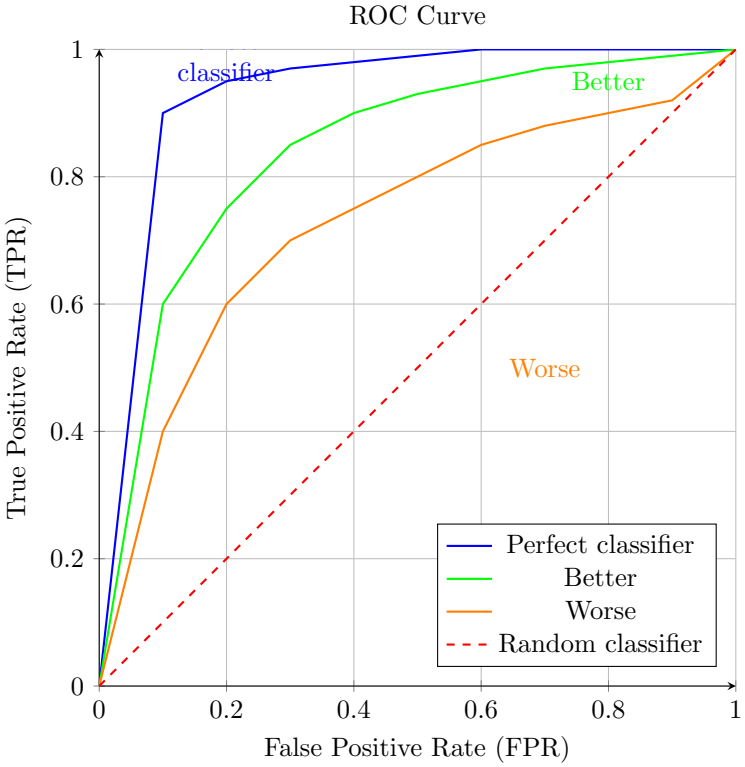
Overall, the model shows a balanced performance with decent precision and high recall. It effectively captures most of the spam emails while maintaining a moderate level of correctness in its spam predictions. To further improve the model, efforts could focus on reducing the false positive rate (increasing 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.



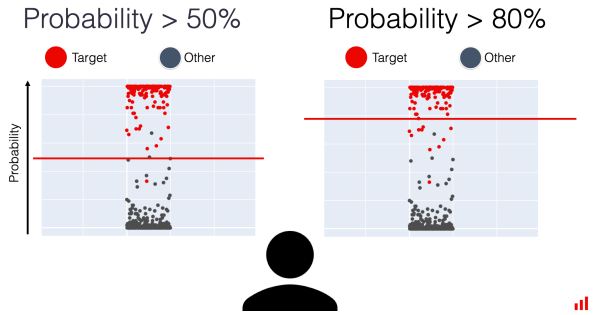


Figure 4.1: Enter Caption

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.
- 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:

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

From this confusion matrix, we can calculate various metrics for each class:

- True Positive (TP): Instances correctly classified as the given class (diagonal elements).
- 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 $\text{Precision}^{(k)}$, $\text{Recall}^{(k)}$, and $\text{F1 Score}^{(k)}$ 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.

Given the above 3x3 confusion matrix for a multi-class classification problem, let's calculate macro and micro metrics and interpret the results.

Actual \ Predicted	A	B	C
A	50	10	5
B	8	100	7
C	4	6	80

Table 4.5: Confusion Matrix

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

Class A: $TP_A = 50, \quad FP_A = 8 + 4 = 12, \quad FN_A = 10 + 5 = 15, \quad TN_A = 100 + 7 + 6 + 80 = 193$
Class B: $TP_B = 100, \quad FP_B = 10 + 6 = 16, \quad FN_B = 8 + 7 = 15, \quad TN_B = 50 + 5 + 4 + 80 = 139$
Class C: $TP_C = 80, \quad FP_C = 5 + 7 = 12, \quad FN_C = 4 + 6 = 10, \quad TN_C = 50 + 10 + 8 + 100 = 168$

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)

Class A: $\text{Precision}_A = \frac{50}{50+12} \approx 0.8065, \quad \text{Recall}_A = \frac{50}{50+15} \approx 0.7692, \quad \text{F1}_A \approx 0.7873$
Class B: $\text{Precision}_B = \frac{100}{100+16} \approx 0.8621, \quad \text{Recall}_B = \frac{100}{100+15} \approx 0.8696, \quad \text{F1}_B \approx 0.8658$
Class C: $\text{Precision}_C = \frac{80}{80+12} \approx 0.8696, \quad \text{Recall}_C = \frac{80}{80+10} \approx 0.8889, \quad \text{F1}_C \approx 0.8791$

4.5.3 Step 3: Calculate Macro-averaged metrics

Macro-averaged metrics give equal weight to each class:

$$\text{Macro Precision} = \frac{\text{Precision}_A + \text{Precision}_B + \text{Precision}_C}{3} \approx 0.8461$$
$$\text{Macro Recall} = \frac{\text{Recall}_A + \text{Recall}_B + \text{Recall}_C}{3} \approx 0.8426$$
$$\text{Macro F1} = \frac{\text{F1}_A + \text{F1}_B + \text{F1}_C}{3} \approx 0.8441$$

4.5.4 Step 4: Calculate Micro-averaged metrics

Micro-averaged metrics aggregate the contributions of all classes:

$$\text{Micro Precision} = \frac{TP_A + TP_B + TP_C}{TP_A + TP_B + TP_C + FP_A + FP_B + FP_C} = \frac{230}{230+40} \approx 0.8519$$
$$\text{Micro Recall} = \frac{TP_A + TP_B + TP_C}{TP_A + TP_B + TP_C + FN_A + FN_B + FN_C} = \frac{230}{230+40} \approx 0.8519$$
$$\text{Micro F1} = 2 * \frac{\text{Micro Precision} * \text{Micro Recall}}{\text{Micro Precision} + \text{Micro Recall}} \approx 0.8519$$

Metric	Precision	Recall	F1-score
Class A	0.8065	0.7692	0.7873
Class B	0.8621	0.8696	0.8658
Class C	0.8696	0.8889	0.8791
Macro-average	0.8461	0.8426	0.8441
Micro-average	0.8519	0.8519	0.8519

Table 4.6: Summary of Metrics

4.6 Results Summary

4.7 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).

4.8 Advanced Model Selection Techniques

Model selection is crucial in machine learning to choose the best model from a set of candidate models. Here are some advanced techniques for model selection:

4.8.1 Cross-Validation Strategies

K-Fold Cross-Validation

K-Fold Cross-Validation is a resampling procedure used to evaluate machine learning models on a limited data sample. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into.

Algorithm 16 K-Fold Cross-Validation

procedure KFoldCV($data, k, model$)
  Split $data$ into k equal-sized folds
  for $i = 1$ to k **do**
    $test\_fold \leftarrow fold^{(i)}$
    $train\_folds \leftarrow$ all folds except $fold^{(i)}$
    Train $model$ on $train\_folds$
    Evaluate $model$ on $test\_fold$
    Store evaluation score
  end for
  return Average of evaluation scores
end procedure

Stratified K-Fold Cross-Validation

Stratified K-Fold is a variation of K-Fold that returns stratified folds: each set contains approximately the same percentage of samples of each target class as the complete set. This is particularly useful for imbalanced datasets.

Certainly. I'll expand on each of these hyperparameter optimization techniques, providing more detailed explanations and examples.

4.8.2 Hyperparameter Optimization

Hyperparameter optimization is a crucial step in machine learning model development, aimed at finding the best configuration of hyperparameters to maximize model performance. Let's delve deeper into the advanced techniques mentioned:

Grid Search with Cross-Validation

Grid search is a systematic approach to hyperparameter tuning that works by exhaustively searching through a predefined set of hyperparameter values. In grid search, we define a "grid" of hyperparameter values and the algorithm evaluates the cartesian product of these sets. For each combination, the model is trained and evaluated using cross-validation. While thorough, grid search can be computationally expensive, especially when the hyperparameter space is large.

Example: Consider tuning a Support Vector Machine (SVM) classifier:

Algorithm 17 Grid Search for SVM

procedure SVMGridSearch(X, y)
  $param\_grid \leftarrow$ {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf'],
    'gamma': [0.01, 0.1, 1]
  }
  $svm \leftarrow$ SVC()
  $grid\_search \leftarrow$ GridSearchCV($svm, param\_grid, cv = 5$)
  $grid\_search$.fit(X, y)
  return $grid\_search$.best_params_
end procedure

This example searches over 24 combinations (4 C values × 2 kernel types × 3 gamma values) using 5-fold cross-validation.

Random Search

Random search is an alternative to grid search that samples hyperparameter combinations randomly from the specified distributions. Instead of exhaustively trying all combinations, random search selects a specified number of random combinations. This method can be more efficient, especially when not all hyperparameters are equally important. It often finds a good combination in less time than grid search, particularly in high-dimensional spaces.

Example: Let's consider tuning a Random Forest classifier:

Algorithm 18 Random Search for Random Forest

procedure RFRandomSearch(X, y)
  param_distributions ← {
    'n_estimators': `randint(10, 200)`,
    'max_depth': [None] + `list(range(5, 30))`,
    'min_samples_split': `randint(2, 20)`,
    'min_samples_leaf': `randint(1, 10)`
  }
  $rf \leftarrow$ `RandomForestClassifier()`
  random_search ← `RandomizedSearchCV(` rf, *param_distributions*, $n^{\text{iter}} = 100$, $cv = 5$)
  random_search.fit(X, y)
  return *random_search*.best_params_
end procedure

This example performs 100 random iterations, sampling from the specified distributions for each hyperparameter.

Chapter 5

Reinforcement Learning

Chapter 6

Neural Network

6.1 Mathematical Model of Neuron

Consider a neuron with only one input feature. Let w_1 be the weight of the input and x_1 refers to

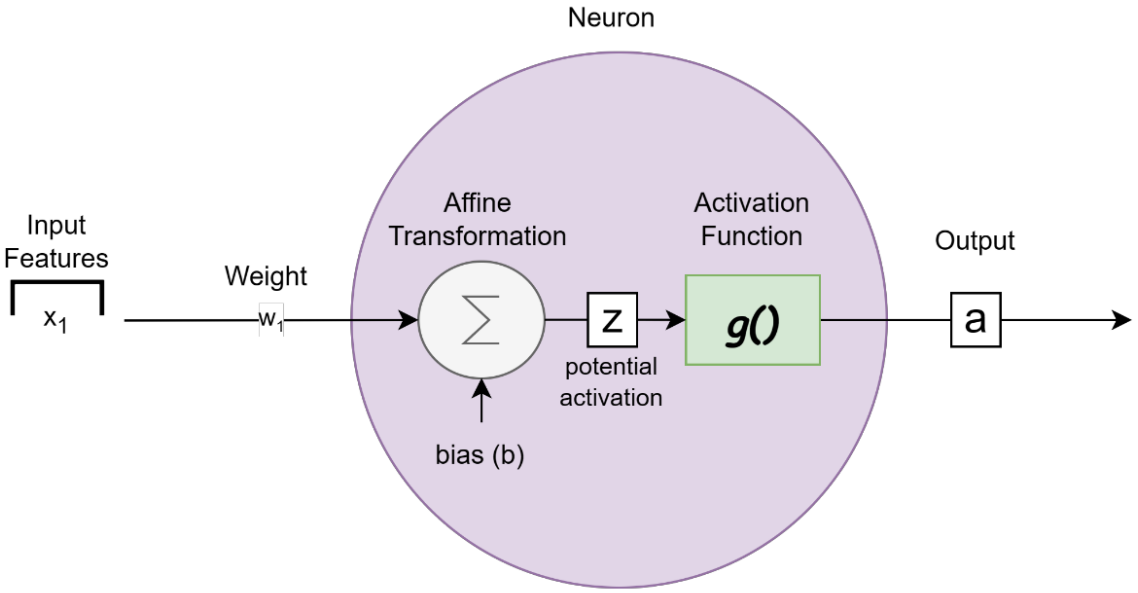


Figure 6.1: Mathematical Model of Neuron

the input and b is the bias to the network.

$$z = w_1 \cdot x_1 + b$$

Where z is the activation potential. Now, the activation potential goes through the activation function

$$a = g(z)$$

Where, $g()$ is the activation function and a is the activation value which is also the output of the neuron.

6.2 Feed Forward Neural Network

A Neural Network is a network of neurons arranged in a structure such that there are multiple neurons in each layer of the network. There is a connection from neurons of one layer of the network to another layer of the network.

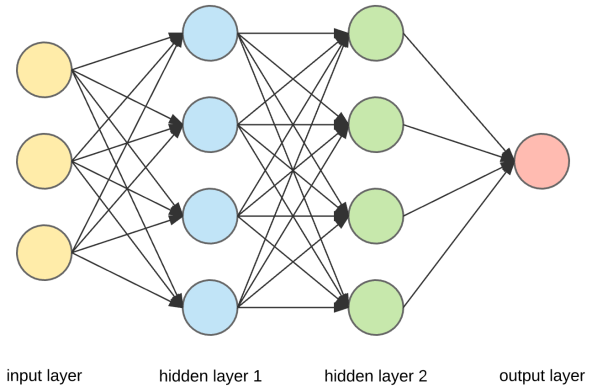


Figure 6.2: Neural Network

6.2.1 Activation Functions

Activation functions introduce non-linearity into the neural network, allowing it to learn more complex functions. Activation functions play a crucial role in neural networks by introducing non-linearity into the model, allowing it to learn complex patterns. Here, we'll explore three common activation functions: the ***Sigmoid function***, the ***Hyperbolic Tangent (tanh)*** function, and the ***Rectified Linear Unit (ReLU)***. Common activation functions include:

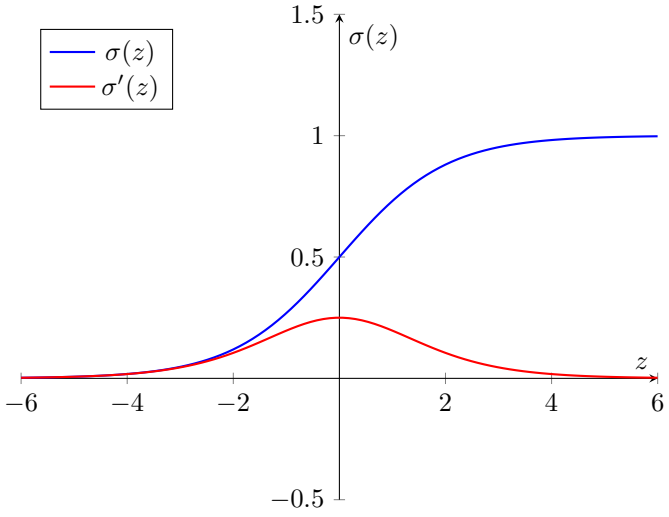
Sigmoid Function

The sigmoid function, also known as the logistic function, maps any real-valued number to the range (0, 1). It is defined as:

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

The derivative of the sigmoid function is:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$



The sigmoid function is smooth and differentiable, making it suitable for gradient-based optimization. However, it suffers from the vanishing gradient problem for very large or small inputs, where the derivative becomes close to zero.

Tanh Function

The hyperbolic tangent (tanh) function maps any real-valued number to the range (-1, 1). It is defined as:

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

The derivative of the tanh function is:

$$\tanh'(z) = 1 - \tanh^2(z)$$

The tanh function is similar to the sigmoid function but is zero-centered, which can help in certain

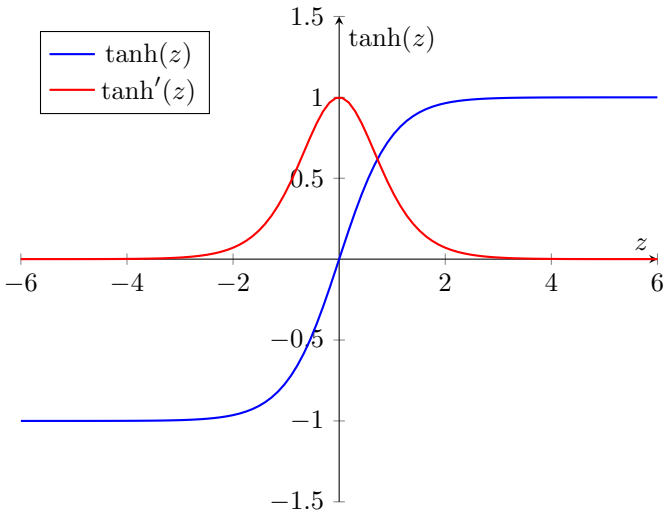


Figure 6.3: Illustration of *tanh*

scenarios. It also suffers from the vanishing gradient problem for very large or small inputs.

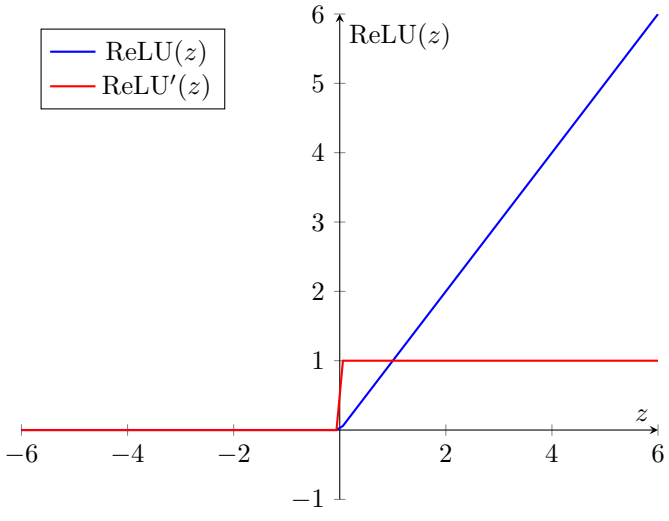
ReLU (Rectified Linear Unit)

The Rectified Linear Unit (ReLU) function is defined as:

$$\text{ReLU}(z) = \max(0, z)$$

The derivative of the ReLU function is:

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$



[h!]

The ReLU function is computationally efficient and helps mitigate the vanishing gradient problem. However, it can suffer from the "dying ReLU" problem, where neurons can get stuck in a state where they always output zero.

6.2.2 Comparison of Activation Functions

Each activation function has its strengths and weaknesses:

- **Sigmoid**: Useful for binary classification problems, but prone to vanishing gradients.
- **Tanh**: Similar to sigmoid but zero-centered, which can help in certain scenarios.
- **ReLU**: Computationally efficient and helps with vanishing gradients, but can suffer from dying neurons.

The choice of activation function depends on the specific problem and network architecture. In practice, ReLU and its variants (e.g., Leaky ReLU, ELU) are often preferred in hidden layers due to their simplicity and effectiveness in deep networks.

6.2.3 Forward Propagation

Forward propagation is the computational execution of a neural network from input to the output. We now construct a neural network step by step increasing the complexity in the network. Let's get started with a single neuron but with multiple inputs.

Step 1: Single layered, multiple inputs

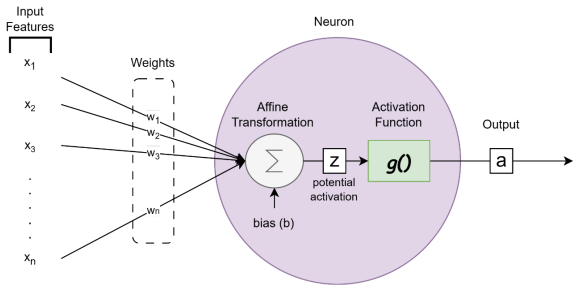


Figure 6.4: 1-Layered Neural Network

Consider a setup of single layered neural network with only one neuron having multiple input features $x_1, x_2, x_3, \ldots, x_n$. Then,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

Notation: x is a scalar and bolded $\mathbf{x}$ is a vector.

The weights associated with each input feature are $w_1, w_2, w_3, \ldots, w_n$, and we have weight vectors as:

$$\mathbf{w} = \begin{bmatrix} w_1 & w_2 & w_3 & \ldots & w_n \end{bmatrix}$$

Later, we take the dot product of $\mathbf{w}$ and $\mathbf{x}$. For our convenience, we take $\mathbf{w}$ as a row vector. If you are comfortable with $\mathbf{w}^T \cdot \mathbf{x}$, feel free to use it as a column vector.

Just remember, throughout this document, we will be using $\mathbf{w} \cdot \mathbf{x}$ for the dot product and $\mathbf{W} \cdot \mathbf{X}$ for matrix multiplication, avoiding the use of transpose.

Now, for the computation that occurs in the neuron, we first compute the activation potential through affine transformation as:

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

Here, $\mathbf{w}$ and $\mathbf{x}$ are vectors. But b is a bias occurring in a neuron and has no relation with the number of inputs, so it's always a scalar. The final output z would be a scalar (verify this yourself).

Now, applying the activation function on the activation potential, we get the activation value:

$$a = g(z)$$

Here, a is the activation value and is a scalar again, inherited from z.

Step 2: Single neuron in each layer for a two-layered network

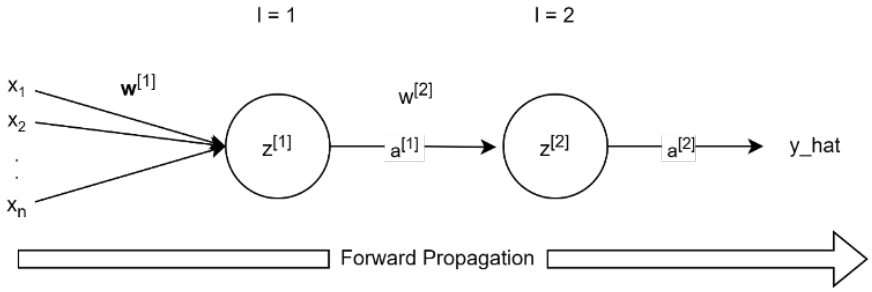


Figure 6.5: Two layered neural network with one neuron in each layer

Now, let's construct a neural network such that there are multiple layers where each layer contains only one neuron. The forward computation in the first layer $l = 1$ would be:

$$z^{[1]} = \mathbf{w}^{[1]} \cdot \mathbf{x} + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

Remember, Superscript [1] refers to the layer we are computing in but not the power raised.

Here, $\mathbf{x}$ and $\mathbf{w}$ are vectors whereas b is a scalar. Also, z and a are scalars. Each layer in the neural network is likely to use a different activation function, thus we are using $g^{[1]}$.

Thus,

- $\mathbf{x}$ is the input vector;
- $\mathbf{w}^{[1]}$ is the weight vector of layer 1;
- $b^{[1]}$ is the bias in the neuron of layer 1;
- $z^{[1]}$ is the activation potential in the neuron of layer 1;
- $a^{[1]}$ is the activation potential in the neuron of layer 1;
- $g^{[1]}$ is the activation function in layer 1.

For computation in the 2nd layer, we use the activation value from layer 1 as input and associate a new weight to this input. Thus, in the layer $l = 2$, computation would be:

$$z^{[2]} = \mathbf{w}^{[2]} \cdot a^{[1]} + b^{[2]}$$

$$a^{[2]} = g^{[2]}(z^{[2]})$$

The activation value of the second layer is the output of the neural network (hence, we can only use Sigmoid or SoftMax activation function in the output layer if the task is classification; otherwise, we can use any activation function including linear function if the task is regression).

Then,

$$\hat{y} = a^{[2]}$$

Step 3: Multiple neurons in each layer for a two-layered network

Now, let's construct a neural network such that there are multiple layers and each layer contains multiple neurons. The forward computation in the first layer $l = 1$ would be:

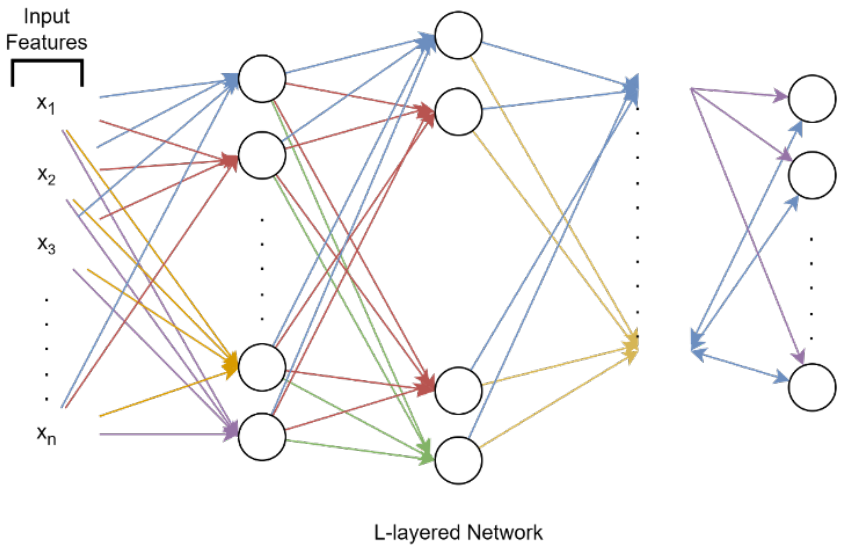


Figure 6.6: L-Layered Neural Network

For each neuron $j = 1 \dots n^{[l]}$,

$$z_j^{[1]} = \mathbf{w}_j^{[1]} \cdot \mathbf{x} + b_j^{[1]}$$

$$a_j^{[1]} = g^{[1]}(z_j^{[1]})$$

Similarly, for the layer $l = 2$, for each neuron $j = 1 \dots n^{[l]}$,

$$z_j^{[2]} = \mathbf{w}_j^{[2]} \cdot a^{[1]} + b_j^{[2]}$$

$$a_j^{[2]} = g^{[2]}(z_j^{[2]})$$

And so on for all the layers.

Forward Propagation Summary

Thus, the forward propagation is of the form:
For $l = 1 \dots L$, For each neuron $j = 1 \dots n^{[l]}$,

$$z_j^{[l]} = \mathbf{w}_j^{[l]} \cdot a^{[l-1]} + b_j^{[l]}$$

$$a_j^{[l]} = g^{[l]}(z_j^{[l]})$$

Step 4: Vectorization

We form a matrix and vector from weights and biases of neurons in a layer. Instead of operating individually on each component, we operate directly in the form of vector and matrix. This way, we perform all the computations at once.

For example, for layer $l = 1$, the weight input in each neuron is already a vector. So, for $n^{[1]}$ neurons in the first layer of the network, we will have the same number of weight vectors incoming from all the input features, i.e., each weight vector has a size of $1 \times n$, where n is the size of input features. If we bring together all $n^{[1]}$ vectors into a matrix, we get a size of $n^{[1]} \times n$.

Similarly, each neuron has a single bias, but if we stack the biases of all neurons in the first layer, we get a bias vector $n^{[1]} \times 1$.

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]} \cdot \mathbf{x} + \mathbf{b}^{[1]}$$
$$\mathbf{a}^{[1]} = g^{[1]}(\mathbf{z}^{[1]})$$

Here,

- input $\mathbf{x}$ is a vector of size $n \times 1$,
- $\mathbf{W}$ is a weight matrix of size $n^{[1]} \times n$ and
- $\mathbf{b}$ is a bias vector of size $n^{[1]} \times 1$.

Thus, $\mathbf{W} \cdot \mathbf{x}$ produces a vector of size $n^{[1]} \times 1$ and addition with bias $\mathbf{b}$ produces the activation potential vector $\mathbf{z}$ of size $n^{[1]} \times 1$. We then obtain activation vector $\mathbf{a}$ of size $n^{[1]} \times 1$.

Vectorized form of Forward Propagation:

The vectorized form of forward propagation is:
For $l = 1 \ldots L$:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{A}^{[l]} = g^{[l]}(\mathbf{Z}^{[l]})$$

Multiple Training Examples: So far we have been considering only one training examples with n features. Let's now add m training examples in our dataset and we take all training examples at once. The our input would be a matrix of size $n \times m$ represented by $\mathbf{X}$.

Then, For layer 1:
$\mathbf{W}$ is a weight matrix of size $n^{[1]} \times n$.
$\mathbf{b}$ is a bias vector of size $n^{[1]} \times 1$.

$\mathbf{W}.\mathbf{X}$ produces a vector of size $n^{[1]} \times m$, and addition with bias $\mathbf{b}$ produces the activation potential vector $\mathbf{Z}$ of size $n^{[1]} \times m$.We then obtain activation vector $\mathbf{A}$ of size $n^{[1]} \times m$. Now, we can proceed in the same way till the last layer and finally obtain the vectorized form of forward propagation as:

Generalization - Vectorized Forward Propagation in Neural Network

For $l = 1 \ldots L$:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{A}^{[l]} = g^{[l]}(\mathbf{Z}^{[l]})$$

This concludes the forward propagation step-by-step derivation.

6.3 Back Propagation

The back propagation algorithm is used to efficiently compute the gradients of the cost function with respect to the weights and biases. The algorithm involves the following steps:

1. Forward Pass: Compute the activation of each neuron and the final cost.
2. Backward Pass: Compute the gradients of the cost with respect to each weight and bias.
3. Parameter Update: Update the weights and biases using the gradients.

To understand and derive the basic form of back propagation, consider a neural network where each layer contains only one neuron, and we are considering only two layers. To make our derivation easy, we consider the 2nd layer as the L layer and the 1st layer as the L-1 layer.

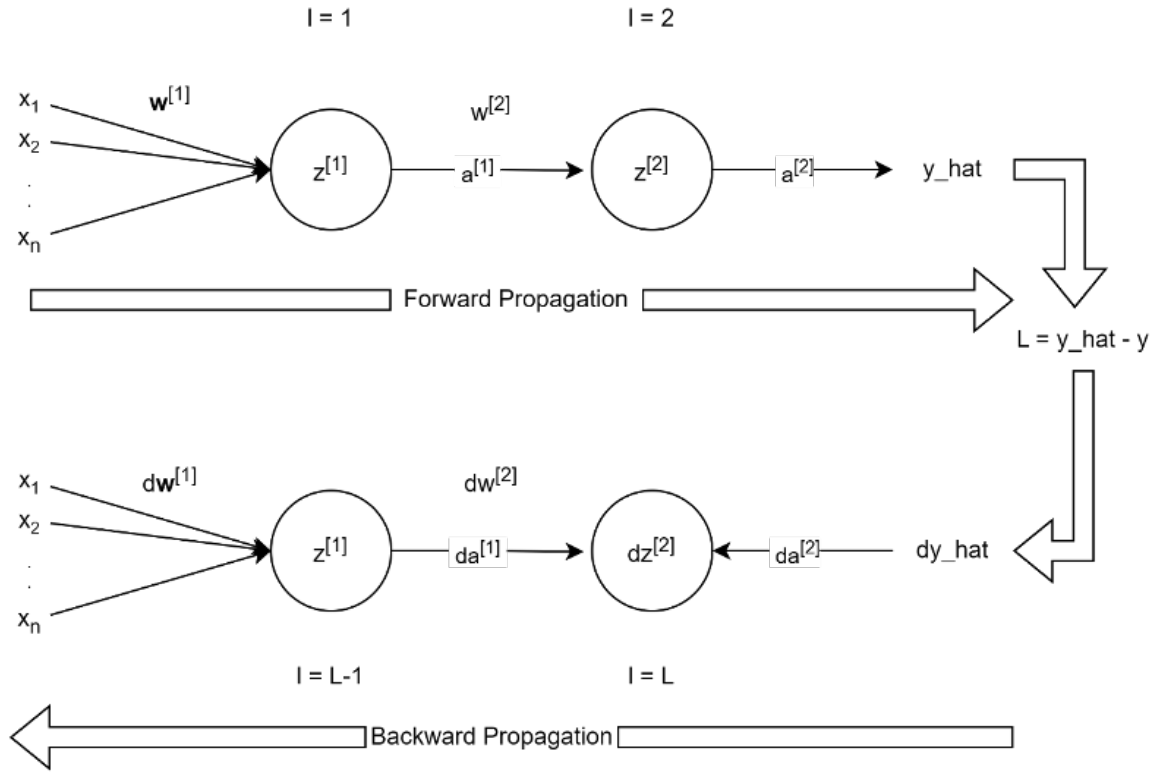


Figure 6.7: Illustration of Backward Propagation

Before proceeding to the backward propagation, let's revise the form of forward propagation in each layer:

In layer L-1:

$$z^{[L-1]} = w^{[L-1]}a^{[L-2]} + b^{[L-1]}$$

$$a^{[L-1]} = g^{[L-1]}(z^{[L-1]})$$

**Note: We use lowercase notation for z, w, and a which means we are computing for a single neuron in the particular layer.*

Similarly, in layer L:

$$z^{[L]} = w^{[L]}a^{[L-1]} + b^{[L]}$$

$$a^{[L]} = g^{[L]}(z^{[L]})$$

Since the L-th layer is the final layer in our network:

$$\hat{y} = a^{[L]}$$

At the end of forward computation, we compute the final output of the neural network as above. We then compute the error for each training example, and by taking all training examples, we generally compute an average of errors known as the cost function.

For regression, the cost function can be:

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

For classification, the cost function can be:

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

where E refers to the average error.

We can now begin the backpropagation process in the neural network. Backpropagation involves propagating the error, calculated during forward propagation by comparing the actual value (y) with the predicted value ($\hat{y}$), backward through the network.

By propagating this error backward, we update the weights and biases that were initially input into the neuron. Using gradient descent, we can determine the new weights and biases for each neuron.

Let's walk through the backpropagation process step-by-step:

Step 1: First, compute the derivative of the error with respect to $\hat{y}$, since the error is calculated using y and $\hat{y}$. The value of y remains constant throughout.

$$\frac{\partial E}{\partial \hat{y}} \tag{6.1}$$

The derivation of this expression depends on the specific form of the cost function used. Therefore, we won't delve further into the derivative's computation here.

For convenience, we introduce the notation:

$$d\hat{y} = \frac{\partial E}{\partial \hat{y}} \tag{6.2}$$

For all future computations, if the derivative of the cost is computed with respect to a variable p, we denote it by dp:

$$dp = \frac{\partial E}{\partial p} \tag{6.3}$$

Now, let's compute the derivatives in Layer L as follows:

Step 2: Compute the derivative of the cost with respect to $a^{[L]}$:

$$da^{[L]} = \frac{\partial E}{\partial a^{[L]}} = \frac{\partial E}{\partial \hat{y}} = d\hat{y} \tag{6.4}$$

Step 3: Compute the derivative of the cost with respect to $z^{[L]}$:

$$dz^{[L]} = \frac{\partial E}{\partial z^{[L]}} = \frac{\partial E}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} = da^{[L]} \cdot g'^{[L]}(z^{[L]}) \tag{6.5}$$

Here,

$$\frac{\partial a^{[L]}}{\partial z^{[L]}} = g'^{[L]}(z^{[L]}) \tag{6.6}$$

because a is the activation value obtained by applying the activation function to z.

Step 4: Compute the derivative of the cost with respect to $w^{[L]}$:

$$dw^{[L]} = \frac{\partial E}{\partial w^{[L]}} = dz^{[L]} \cdot a^{[L-1]} \tag{6.7}$$

Step 5: Compute the derivative of the cost with respect to $b^{[L]}$:

$$db^{[L]} = \frac{\partial E}{\partial b^{[L]}} = dz^{[L]} \tag{6.8}$$

Now, we compute the derivative in Layer $L-1$ as follows:

Step 6: Compute the derivative of the cost with respect to $a^{[L-1]}$:

$$da^{[L-1]} = \frac{\partial E}{\partial a^{[L-1]}} = \frac{\partial E}{\partial z^{[L]}} \cdot \frac{\partial z^{[L]}}{\partial a^{[L-1]}} = dz^{[L]} \cdot w^{[L]} \quad [\text{ from (3)}] \tag{6.9}$$

Remember, the sequence of derivatives we performed is based on the forward propagation form in the L-th layer:

$$z^{[L]} = w^{[L]} \cdot a^{[L-1]} + b^{[L]} a^{[L]} = g^{[L]}(z^{[L]}) \tag{6.10}$$

This is because backpropagation uses a chain method for computing derivatives.

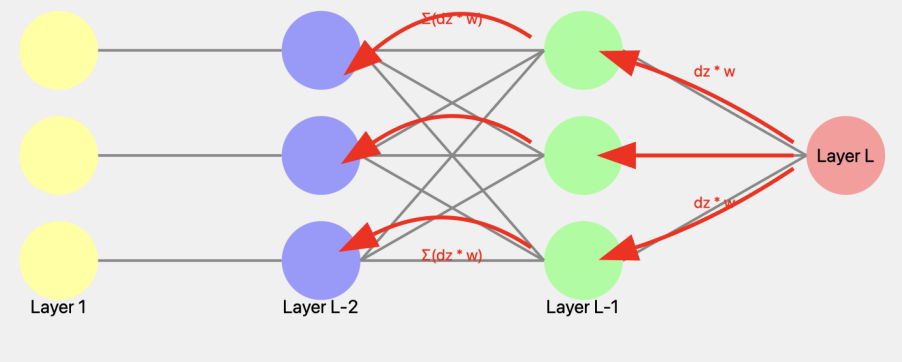


Figure 6.8: Gradient Accumulation during back propagation

Step 7: Next, compute the derivative of the cost with respect to $z^{[L-1]}$:

$$dz^{[L-1]} = \frac{\partial E}{\partial z^{[L-1]}} = \frac{\partial E}{\partial a^{[L-1]}} \cdot \frac{\partial a^{[L-1]}}{\partial z^{[L-1]}} = da^{[L-1]} \cdot g'^{[L-1]}(z^{[L-1]})$$

Step 8: Similarly, compute the derivative of the cost with respect to $w^{[L-1]}$ and $b^{[L-1]}$:

$$dw^{[L-1]} = \frac{\partial E}{\partial w^{[L-1]}} = \frac{\partial E}{\partial z^{[L-1]}} \cdot \frac{\partial z^{[L-1]}}{\partial w^{[L-1]}} = dz^{[L-1]} \cdot a^{[L-2]} \tag{6.11}$$

And,

$$db^{[L-1]} = \frac{\partial E}{\partial b^{[L-1]}} = \frac{\partial E}{\partial z^{[L-1]}} \cdot \frac{\partial z^{[L-1]}}{\partial b^{[L-1]}} = dz^{[L-1]} \cdot 1 = dz^{[L-1]} = a^{[L]} \cdot g'^{[L]}(z^{[L]}) \cdot w^{[L]} \cdot g'^{[L-1]}(z^{[L-1]}) \tag{6.12}$$

And so on for other layers like $L-2$, $L-3$, etc., down to layer 1 if they exist.

Now, if we compare the form of the backpropagation derivative for weight (w) and bias (b) in layer L and layer $L-1$, we observe a pattern. From the above derivations, we can summarize:

For Layer L:

$$dz^{[L]} = da^{[L]} \cdot g'^{[L]} = d\hat{y} \cdot g'^{[L]}$$
$$dw^{[L]} = dz^{[L]} \cdot a^{[L-1]}$$
$$db^{[L]} = dz^{[L]}$$

For Layer $L-1$:

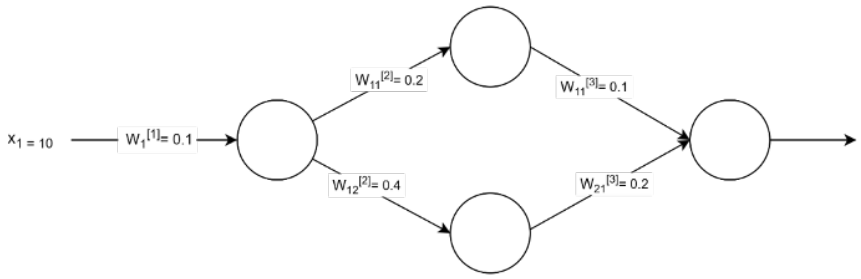
$$da^{[L-1]} = dz^{[L]} \cdot w^{[L]}$$
$$dz^{[L-1]} = da^{[L-1]} \cdot g'^{[L-1]} = dz^{[L]} \cdot w^{[L]} \cdot g'^{[L-1]}$$
$$dw^{[L-1]} = dz^{[L-1]} \cdot a^{[L-2]}$$
$$db^{[L-1]} = dz^{[L-1]}$$

And, if layer $L-2$ exists, we can write:

$$dz^{[L-2]} = dz^{[L-1]} \cdot W^{[L-1]} \cdot g'^{[L-2]}$$
$$dw^{[L-2]} = dz^{[L-2]} \cdot a^{[L-3]}$$
$$db^{[L-2]} = dz^{[L-2]}$$

But in a general neural network, there isn't just a single neuron in each layer. While the last layer (Layer L) follows the form given above, the layers from $L-1$ to the 1st layer follow a different form. Following Two key cases arise from this:

1. From the L-th layer, the error is backpropagated to all the neurons in the $L-1$ layer, and from the $L-1$ layer to the $L-2$ layer, and so on. The backpropagated error from the l-th layer is passed in its entirety to each neuron of the $(l-1)$-th layer.



2. Each neuron in the $L-1$ layer receives backpropagated error from every neuron in the L-th layer. This means each neuron in the $L-1$ layer accumulates error from all the neurons in the L-th layer. Therefore, we can express the accumulated error for neuron i in layer $l = L-1, \ldots, 1$ as:

$$\sum_{j=1}^{n^{[l]}} \left(dz_j^{[l+1]} \cdot w_{ij}^{[l+1]} \right)$$

where $n^{[l]}$ is the number of neurons in the l-th layer.

This changes the form of the equations we derived earlier, except for the last layer, which remains the same.

Backpropagation in Neural Network

For $l = L$ (i.e., the last layer), for each neuron j:

$$dz^{[L]} = d\hat{y} \cdot g'^{[L]}(z^{[L]})$$

$$dw^{[L]} = dz^{[L]} \cdot a^{[L-1]}$$

$$db^{[L]} = dz^{[L]}$$

For $l = L-1$ to 1 (i.e., all layers except the last), for each neuron i:

$$dz^{[l]} = g'^{[l]}(z^{[l]}) \cdot \sum_{j=1}^{n^{[l+1]}} \left(dz_j^{[l+1]} \cdot w_{ij}^{[l+1]} \right)$$

$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$$

$$db^{[l]} = dz^{[l]}$$

Numerical Problem 1: **Problem:** Given a neural network, perform forward propagation considering a classification problem and compute the loss. Consider a sigmoid activation function and a learning rate of 0.1. **Given:** $(x, y) = (1, 0)$.

Solution: There are three layers in the network. First, let's execute forward propagation. Since there is only one training example, we use a vector form, and for ease of calculation, we ignore bias.

In layer 1:
The weight w is a scalar as we have only one input. Thus, $w = 0.1$, and $x = a^{[0]}$.

$$a^{[1]} = \text{sigmoid}(w \cdot a^{[0]}) = \text{sigmoid}(1 \cdot 0.2) = \text{sigmoid}(0.2) = 0.55$$

In layer 2:
The weight w is a vector as we have two neurons in layer 2. Thus,

$$w = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$$

$$a^{[2]} = \text{sigmoid}(w \cdot a^{[1]}) = \text{sigmoid}\left(\begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix} \cdot 0.55 \right) = \text{sigmoid}\left(\begin{bmatrix} 0.11 \\ 0.22 \end{bmatrix} \right) = \begin{bmatrix} 0.53 \\ 0.55 \end{bmatrix}$$

In layer 3:
The weight w is again a vector as there are two inputs for a single neuron. Thus,

$$w = \begin{bmatrix} 0.1 & 0.2 \end{bmatrix}$$

$$a^{[3]} = \text{sigmoid}(w \cdot a^{[2]}) = \text{sigmoid}\left(\begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.53 \\ 0.55 \end{bmatrix} \right) = \text{sigmoid}(0.1 \cdot 0.53 + 0.2 \cdot 0.55) = \text{sigmoid}(0.163) = 0.54$$

The final output of the network is 0.54.

Computing the loss using the log loss function:

$$E = -y \log(\hat{y}) - (1-y) \log(1-\hat{y}) = -(0 \cdot \log(0.54)) - (1-0) \cdot \log(1-0.54) = 0.337242168$$

Numerical 2: Solve for the backward propagation of the given neural network in 1.

Given:

- Final output $\hat{y} = 0.54$
- True label $y = 0$
- Loss $C = 0.337$

Step 1: Compute the derivative of the loss with respect to $\hat{y}$

$$\frac{\partial E}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = -\frac{0}{0.54} + \frac{1-0}{1-0.54} = \frac{1}{0.46} \approx 2.174$$

Step 2: Compute the derivative of the loss with respect to $z^{[3]}$ Since $\hat{y} = a^{[3]}$, and $a^{[3]} = \sigma(z^{[3]})$, we use the chain rule:

$$dz^{[3]} = \frac{\partial E}{\partial z^{[3]}} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[3]}} = 2.174 \cdot \sigma'(z^{[3]})$$

where $\sigma'(z^{[3]}) = \hat{y}(1-\hat{y})$, so:

$$dz^{[3]} = 2.174 \cdot 0.54 \cdot (1-0.54) \approx 2.174 \cdot 0.2484 \approx 0.5395$$

Step 3: Compute the derivative of the loss with respect to $w^{[3]}$ and $b^{[3]}$

$$dw^{[3]} = dz^{[3]} \cdot a^{[2]} = 0.5395 \cdot \begin{bmatrix} 0.53 \\ 0.55 \end{bmatrix} \approx \begin{bmatrix} 0.2869 \\ 0.2967 \end{bmatrix}$$

$$db^{[3]} = dz^{[3]} \approx 0.5395$$

Step 4: Compute the derivative of the loss with respect to $z^{[2]}$

$$dz^{[2]} = dz^{[3]} \cdot w^{[3]} \cdot \sigma'(z^{[2]})$$

Using the values:

$$dz^{[2]} = 0.5395 \cdot \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} \cdot \sigma'(z^{[2]}) = \begin{bmatrix} 0.05395 \\ 0.1079 \end{bmatrix} \cdot \sigma'(z^{[2]})$$

where $\sigma'(z^{[2]}) = a^{[2]}(1-a^{[2]}) = \begin{bmatrix} 0.53(1-0.53) \\ 0.55(1-0.55) \end{bmatrix} \approx \begin{bmatrix} 0.2481 \\ 0.2475 \end{bmatrix}$:

$$dz^{[2]} \approx \begin{bmatrix} 0.05395 \cdot 0.2481 \\ 0.1079 \cdot 0.2475 \end{bmatrix} \approx \begin{bmatrix} 0.0134 \\ 0.0267 \end{bmatrix}$$

Step 5: Compute the derivative of the loss with respect to $w^{[2]}$ and $b^{[2]}$

$$dw^{[2]} = dz^{[2]} \cdot a^{[1]} = \begin{bmatrix} 0.0134 \\ 0.0267 \end{bmatrix} \cdot 0.55 \approx \begin{bmatrix} 0.00737 \\ 0.0147 \end{bmatrix}$$

$$db^{[2]} = dz^{[2]} \approx \begin{bmatrix} 0.0134 \\ 0.0267 \end{bmatrix}$$

Step 6: Compute the derivative of the loss with respect to $z^{[1]}$

$$dz^{[1]} = dz^{[2]} \cdot w^{[2]} \cdot \sigma'(z^{[1]})$$

Using the values:

$$dz^{[1]} = \begin{bmatrix} 0.0134 \\ 0.0267 \end{bmatrix} \cdot \begin{bmatrix} 0.2 & 0.4 \end{bmatrix} \cdot \sigma'(z^{[1]}) = \begin{bmatrix} 0.00268 \\ 0.01068 \end{bmatrix} \cdot \sigma'(z^{[1]})$$

where $\sigma'(z^{[1]}) = a^{[1]}(1 - a^{[1]}) \approx 0.55(1 - 0.55) \approx 0.2475$:

$$dz^{[1]} \approx \begin{bmatrix} 0.00268 \cdot 0.2475 \\ 0.01068 \cdot 0.2475 \end{bmatrix} \approx \begin{bmatrix} 0.00066 \\ 0.00264 \end{bmatrix}$$

Step 7: Compute the derivative of the loss with respect to $w^{[1]}$ and $b^{[1]}$

$$dw^{[1]} = dz^{[1]} \cdot x = \begin{bmatrix} 0.00066 \\ 0.00264 \end{bmatrix} \cdot 1 \approx \begin{bmatrix} 0.00066 \\ 0.00264 \end{bmatrix}$$

$$db^{[1]} = dz^{[1]} \approx \begin{bmatrix} 0.00066 \\ 0.00264 \end{bmatrix}$$

Algorithm 19 Backpropagation Algorithm

Initialize: Randomly initialize the weights $\mathbf{w}^{[l]}$ and biases $\mathbf{b}^{[l]}$ for all layers $l = 1, \ldots, L$ in the network.
repeat
  for each training example $(\mathbf{x}^{(i)}, y^{(i)})$ in the training set $\mathcal{X}$ **do**
    Forward Propagation:
    Set the input layer activation $\mathbf{a}^{[0]} = \mathbf{x}^{(i)}$.
    for each layer $l = 1$ to L **do**
      Compute the linear combination:

$$z^{[l]} = \mathbf{w}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

      Apply the activation function:

$$\mathbf{a}^{[l]} = g^{[l]}(z^{[l]})$$

    end for
    Backward Propagation:
    For the output layer $l = L$**:**
    for each neuron j **do**
      Compute the error term:

$$\delta z_j^{[L]} = \frac{\partial L}{\partial \hat{y}} \cdot g'^{[L]}(z_j^{[L]})$$

      Compute the gradient of the loss with respect to the weights and biases:

$$\delta \mathbf{w}_j^{[L]} = \delta z_j^{[L]} \cdot \mathbf{a}^{[L-1]}$$

$$\delta \mathbf{b}_j^{[L]} = \delta z_j^{[L]}$$

    end for
    For each hidden layer $l = L - 1$ **to** 1**:**
    for each neuron j **do**
      Compute the error term:

$$\delta z_j^{[l]} = g'^{[l]}(z_j^{[l]}) \sum_{k=1}^{n^{[l+1]}} \delta z_k^{[l+1]} \cdot w_{jk}^{[l+1]}$$

      Compute the gradient of the loss with respect to the weights and biases:

$$\delta \mathbf{w}_j^{[l]} = \delta z_j^{[l]} \cdot \mathbf{a}^{[l-1]}$$

$$\delta \mathbf{b}_j^{[l]} = \delta z_j^{[l]}$$

    end for
    Update the weights and biases:
    for each layer $l = 1$ to L **do**
      for each neuron j **do**
        Update the weights:

$$\mathbf{w}_j^{[l]} \leftarrow \mathbf{w}_j^{[l]} - \alpha \cdot \delta \mathbf{w}_j^{[l]}$$

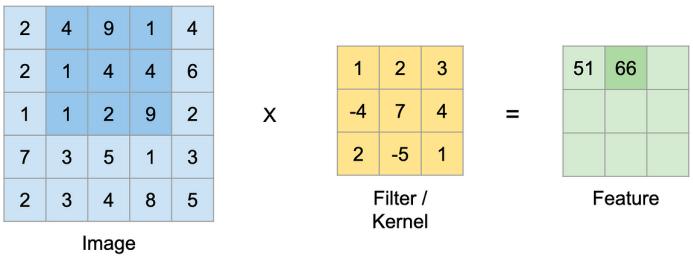
        Update the biases:

$$\mathbf{b}_j^{[l]} \leftarrow \mathbf{b}_j^{[l]} - \alpha \cdot \delta \mathbf{b}_j^{[l]}$$

      end for
    end for
  end for
until convergence

6.4 Convolutional Neural Network

Convolutional Neural Network is a type of neural network with convolutional layers in the network. A convolutional layer has a number of filters (a.k.a kernels) that are used in convolution operation. A convolution operation focuses on extracting/preserving important features from the input. Filters



are used to detect a pattern from certain region, in above case at a time small pattern from 3×3 region of input image is detected. There can be multiple filters applied at a time:

If the image is grayscale, then the image is considered as a matrix, each value in the matrix ranges from 0-255. We can even normalize these values let's say in range 0-1. 0 represents white and 1 represents black. On the other hand, If the image is colored, then three matrices representing the RGB colors with each value in range 0-255.

6.4.1 Convolution Operation

Convolution Operation is actually misnomer, since the operations they express are more accurately described as cross-correlations. Consider a two-dimensional cross correlation operation as below:

We begin with kernel window positioned to the upper left corner and perform convolution operation as

$$0 * 0 + 1 * 1 + 3 * 2 + 4 * 3 = 0 + 1 + 6 + 12 = 19$$

This is the element-wise multiplication of two matrices and summation of the values.

Next, the filter window slides by 1 and the convolution operation is conducted as:

$$1 * 0 + 2 * 1 + 4 * 2 + 5 * 3 = 0 + 2 + 8 + 15 = 25$$

After, sliding the filter fully on horizontal direction we slide the filter by 1 vertically and start again from the left-hand side

$$3 * 0 + 4 * 1 + 6 * 2 + 7 * 3 = 0 + 4 + 12 + 21 = 37$$

And slide horizontally by 1, we get

$$4 * 0 + 5 * 1 + 7 * 2 + 8 * 3 = 0 + 5 + 14 + 24 = 43$$

Thus, the final output would be:

This means the output matrix has reduced size. This can be explained by:

$$(n_h - f_h + 1) \times (n_w - f_w + 1)$$

Where,

$$\begin{aligned} n_h &= \text{height of an image i.e. no. of pixel vertically arranged in image.} \\ n_w &= \text{width of an image i.e. no. of pixel horizontally arranged in image.} \\ f_h &= \text{height of filter i.e. no. of pixel vertically arranged in filter} \\ f_w &= \text{width of filter i.e. no. of pixel horizontally arranged in image} \end{aligned}$$

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Figure 6.9: 6 × 6 grayscale Image

1	-1	-1
-1	1	-1
-1	-1	1

(a) Filter-1

-1	1	-1
-1	1	-1
-1	1	-1

(b) Filter-2

Figure 6.10: Comparison of Filter-1 and Filter-2

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Numerical Example: Consider a grey scale image of 6 × 6 size, Apply following two filters:
Solution: Applying first filter, we get
Now, applying second filter, Thus, we get two different images by applying two filters. This means, applying multiple filters in a stack produces multiple output images.

Mathematically, a convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the kernel and the scalar bias. When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully connected layer and we need to update all the components of kernel and bias during the back propagation.

6.4.2 Stride

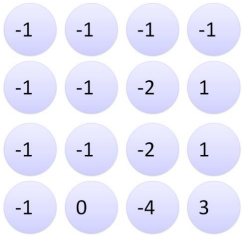
Stride refers to the sliding of window after each convolution operation i.e. stride governs how many cells the filter should move in an input after each convolution operation. Normally, stride value is 1, but it can be larger number as well. In above examples, stride value (s) is 1. Taking the stride value 2 in above numerical problem,

Image and Filter

Then the output would be:

The resulting size of the output after applying filter with stride is given, mathematically, as

$$\left\lfloor \frac{n_h + f_h + s}{s} \right\rfloor \times \left\lfloor \frac{n_w + f_w + s}{s} \right\rfloor$$



6.4.3 Padding

One of the issues with convolution is the pixel around the perimeter of image are used only once for the operation. Thus, we tend to lose the pixel from the ends of an image. To avoid so, we pad an image with 0 pixel on perimeters. We can pad 0 pixel as necessary on any side to our input

image. If we apply padding two times in image i.e. $p = 2$, then image size increases by 4 times on horizontal and vertical sides i.e. new image size becomes:

$$(n_h + 2p) \times (n_w + 2p)$$

And, after applying convolution in padded image, the size of output is:

$$\left\lfloor \frac{n_h + f_h + 2p + s}{s} \right\rfloor \times \left\lfloor \frac{n_w + f_w + 2p + s}{s} \right\rfloor$$

Also, written as,

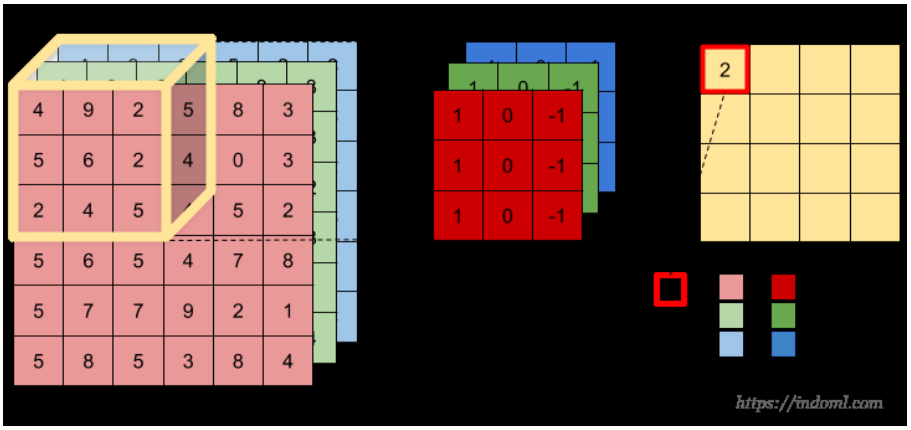
$$\left\lfloor \frac{n_h + f_h + 2p}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_w + f_w + 2p}{s} + 1 \right\rfloor$$

6.4.4 Types of Convolutions by padding

1. **Same Convolution** If the output after applying convolution operation has same size as the input it is called same convolution.
   For example, consider an image of 6×6 and we apply padding $p = 1$, and stride $s = 1$, then the size of an output after applying filter of 3×3 is
   $$\left\lfloor \frac{6 - 3 + 2 * 1 + 1}{s} \right\rfloor \times \left\lfloor \frac{6 - 3 + 2 * 1 + 1}{s} \right\rfloor = 6 \times 6$$
2. **Valid Convolution** If the output shrinks compared to input after applying convolution, then it is called valid convolution. We do not pad in inputs in case of valid convolution.
   For example, apply no padding in above example
   $$\left\lfloor \frac{6 - 3 + 2 * 0 + 1}{s} \right\rfloor \times \left\lfloor \frac{6 - 3 + 2 * 0 + 1}{s} \right\rfloor = 4 \times 4$$

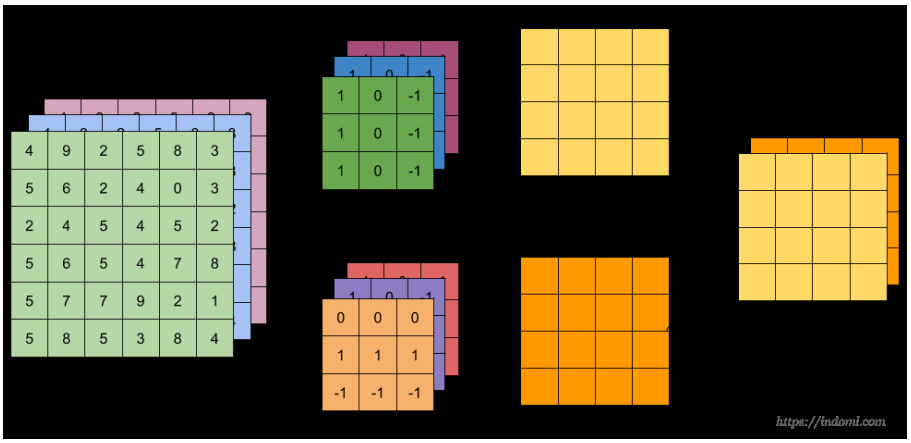
6.4.5 Convolution Operation on Volume

When an image has more than one channels, for example color image has 3 channels i.e. red, green and blue, the filter should also have matching number of channels.



And, to get the output, the volume of input is cross correlated with the same volume of filter and add them all.

In case of multiple filters applied, the output will have same number of channels as number of filters applied.



6.4.6 Pooling

Pooling is the act of combining two or more things. The pooling operation involves sliding pooling window over each channel of feature map and summarizing the features lying within the region covered by the filter.

There are three techniques of summarization viz. maximization, averaging and minimization. This makes three types of pooling:

1. Max Pooling (Very commonly used)
2. Average Pooling
3. Min Pooling (Not used)

Max pooling takes the maximum value from the pooling window of input. Similarly, average pooling takes the average of all value from the pooling window of input.

Pooling always reduces the size of input depending on the size of filter. Most commonly, pooling filter of 2×2 is used.

The reason that max pooling is common is that we are interested in those pixels which has more intensity as they contribute more to the feature in image. Pooling is also known as down sampling.

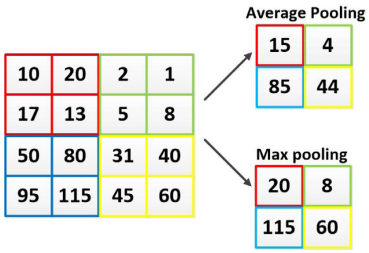


Figure 6.11: Pooling in CNN

6.4.7 Flattening

Flattening refers to the process of converting dimensional matrix to one dimensional matrix or say vector. Flattening is done before feeding the outcome of convolutional neural network or inputs like images which has matrix like structure to the fully connected neural network. Then the flattened

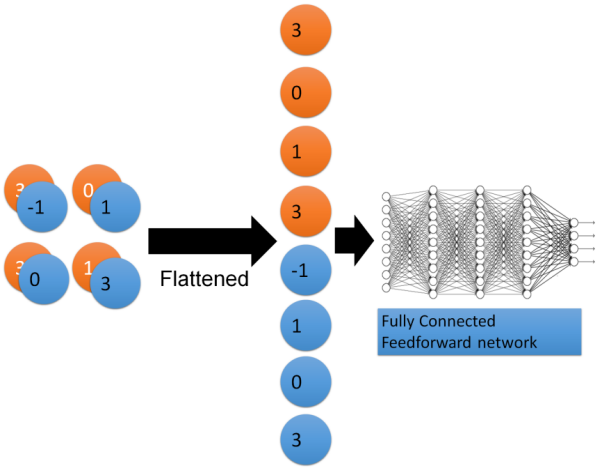


Figure 6.12: Flattening after CNN in Neural Network

output is passed to the fully connected neural network.

6.4.8 Complete Convolutional Neural Network

We can now design a convolutional neural network using everything we discussed above as below:

Numerical Example Consider a Convolutional Neural Network having three different convolutional layers in its architecture as –

Layer-1: Filter Size – 3 X 3, Number of Filters – 10, Stride – 1, Padding – 0

Layer-2: Filter Size – 5 X 5, Number of Filters – 20, Stride – 2, Padding – 0

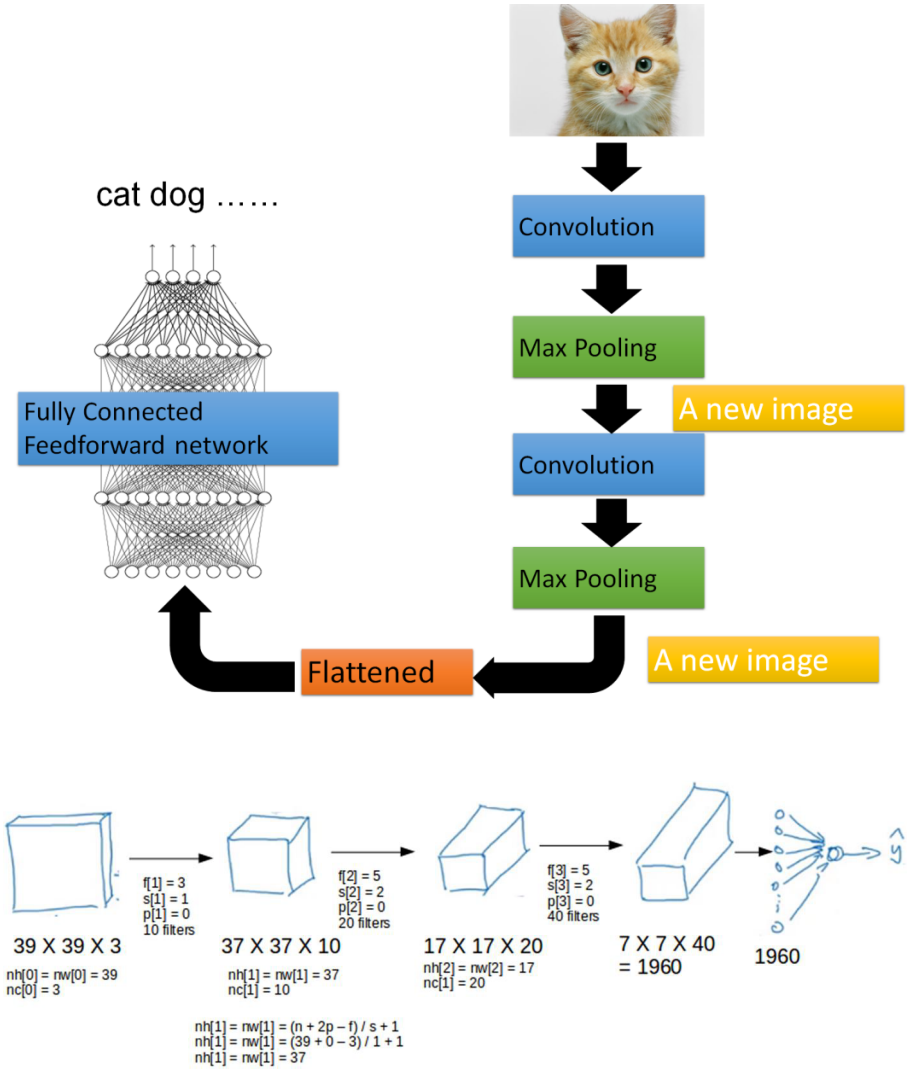
Layer-3: Filter Size – 5 X5 , Number of Filters – 40, Stride – 2, Padding – 0

If we give the input a 3-D image to the network of dimension 39 X 39, then determine the dimension of the vector after passing through a fully connected layer in the architecture.

Solution:

6.5 Recurrent Neural Network

A recurrent neural network (RNN) got its name from the architecture where neuron recurs exhibiting similar pattern. It is a type of artificial neural network which uses data that occurs in sequence.



These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate.

Like feed-forward and convolutional neural networks (CNNs), recurrent neural networks utilize training data to learn. They are distinguished by their "memory" as they take information from prior inputs to influence the current input and output. The above figure illustrates the compressed architecture and unfolded architecture of recurrent neural network. There are only two layers in the above network represented by h and o denoting hidden layer and output layer respectively. This much of layers should be enough to understand the working mechanisms of RNN. And, remember, as in fully connected neural, not all the neurons of preceding layer is connected to all the neurons of succeeding layer.

6.5.1 Forward Propagation

Let's start understanding the forward propagation in RNN. First input is the X_1 in the sequence which is the input to h_1 in the network with the weight of W_{xh}. Also the hidden layer h_1 gets some activation value from the previous sequence as an input which doesn't actually exist since this is the first one in sequence, but still we take some random activation h_0 with the weight of W_{hh}.

Then, in input layer $l = 1$, operation occurring in neuron h_1 during forward propagation can be

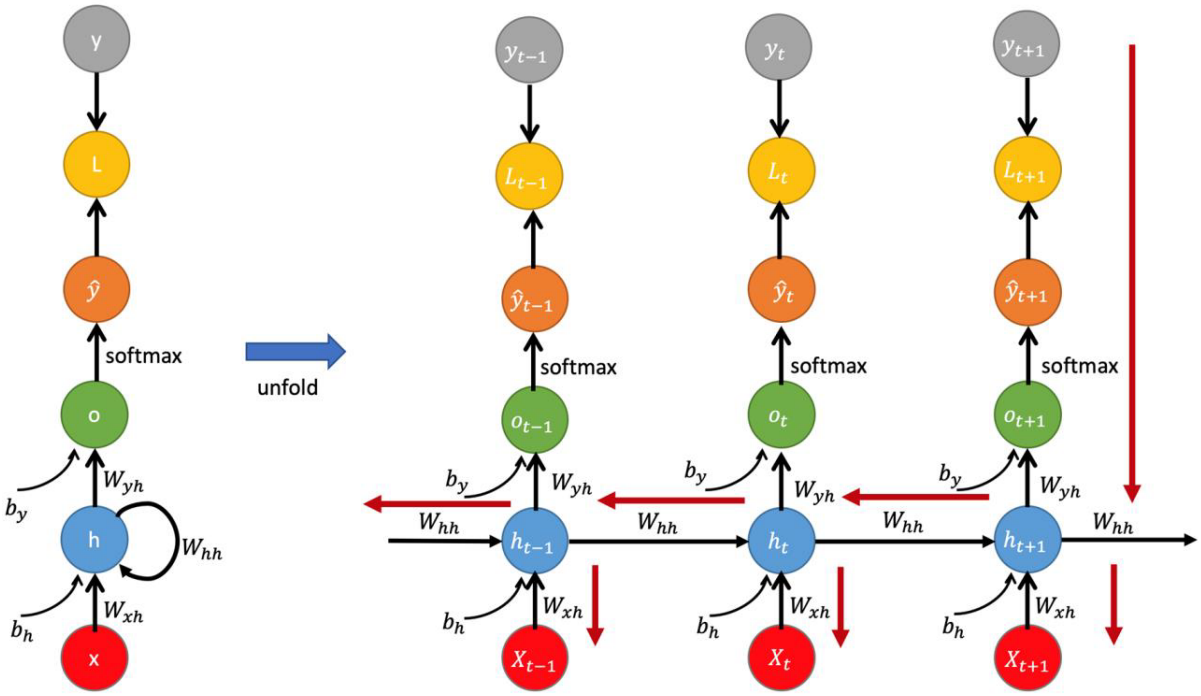


Figure 6.13: RNN architecture, folded in the left and unfolded in the right

written as:

$$z^{\langle 1 \rangle} = W_{xh} \cdot X_1 + W_{hh} \cdot a^{\langle 0 \rangle} + b^{\langle 1 \rangle}$$
$$a^{\langle 1 \rangle} = g(z^{\langle 1 \rangle})$$

The output of first hidden unit from first layer $l = 1$ goes to first hidden neuron of second layer $l = 2$ and second hidden unit of first layer $l = 1$.

Then, the h_2 hidden unit in first layer receives input $a^{\langle 1 \rangle}$ from h_1 hidden unit of first layer and $x^{\langle 2 \rangle}$ input as the second input of the input sequence.

If you are still not sure, what the sequence is then consider a sentence in English "How are you?" where the occurrence of "are" is dependent on "How" and occurrence of "you" on "How" and "are". This is the sequential problem and the word occurring later has dependent over the all the already occurred words.

Now, after the completion of computation in first unit of first layer, we can proceed for the computation in the second unit of first layer,

$$z^{\langle 2 \rangle} = W_{xh} \cdot X_2 + W_{hh} \cdot a^{\langle 1 \rangle} + b^{\langle 2 \rangle}$$
$$a^{\langle 2 \rangle} = g(z^{\langle 2 \rangle})$$

And so on.

Thus, the forward propagation in RNN, can be mathematically written as:

In hidden layer $l = 1$, for $t = 1 \dots T$

$$z^{[1]\langle t \rangle} = W_{xh}^{[1]} \cdot a_t^{[1]} + W_{hh}^{[1]} \cdot a^{\langle t-1 \rangle} + b^{[1]\langle t \rangle}$$
$$a^{\langle t \rangle} = g_h^{[1]}(z^{[1]\langle t \rangle})$$

Take a time to digest this.

Here,

W_{xh} is the weight of the inputs to the hidden unit,
W_{hh} is the weight of the inputs from and to the hidden unit, and,
W_{yh} is the weight of the output from the hidden unit when passing to another layer

Now, in second layer as in first layer, forward propagation computation happens
In output layer $l = 2$, for $t = 1 \dots T$

$$z^{[2]\langle t\rangle} = W_{oh}^{[1]} \cdot a^{[1]\langle t\rangle} + W_{hh}^{[2]} \cdot a^{[1]\langle t-1\rangle} + b^{[2]\langle t\rangle}$$
$$a^{[2]\langle t\rangle} = g_h^{[2]}(z^{[2]\langle t\rangle})$$

And so on for all the other layers, if they exist.

Generalization

Forward Propagation in RNN

for $l = 1 \dots L-1$
  for $t = 1 \dots T$
    $z^{[l]\langle t\rangle} = W_{xh}^{[l]} \cdot a^{[t]} + W_{hh}^{[l]} \cdot a^{\langle t-1\rangle} + b^{[l]\langle t\rangle}$
    $a^{[l]\langle t\rangle} = g_h^{[1]}(z^{[l]\langle t\rangle})$
Here, $W_{oh}^{[l-1]} = W_{xh}^{[l]}$
for $l = L$
  for $t = 1 \dots T$
    $z^{[L]\langle t\rangle} = W_{xh}^{[L]} \cdot a^{[L]\langle t\rangle} + W_{hh}^{[l]} \cdot a^{\langle t-1\rangle} + b^{[l]\langle t\rangle}$
    $a^{[l]\langle t\rangle} = g_h^{[1]}(z^{[l]\langle t\rangle})$

6.5.2 Backpropagation Through Time

Recurrent Neural Networks (RNNs) specialize in processing sequences of data, introducing the concept of time steps, where each step corresponds to a specific moment in the sequence. This temporal aspect enables RNNs to excel in tasks involving sequential data like text, speech, and time series.

A time step in a Recurrent Neural Network (RNN) refers to a specific moment or instance in a sequence of data being processed by the network. In the context of sequential data, such as text, speech, or time series, the RNN processes one element of the sequence at each time step.

For RNNs to learn a pattern from the sequential data, a variant of the backpropagation algorithm known as "**Backpropagation Through Time**" (BPTT) is used. Let's understand backpropagation through time step by step:

Consider we have only one time step sequence in the network, then this will be a simple neural network where there is one neuron in each layer. Consider E be the cost function used to calculate error irrespective of classification or regression function. Then, as a first step in the backpropagation we need to compute:

$$\frac{\partial E}{\partial \hat{y}}$$

And, as per the notation introduced above, in the backpropagation, we can write:

$$d\hat{y} = \frac{\partial E}{\partial \hat{y}}$$

Next, we compute the derivative of cost with respect to W_{oh}:

$$dW_{oh} = \frac{\partial E}{\partial W_{oh}} = \frac{\partial E}{d\hat{y}} \cdot \frac{\partial \hat{y}}{dW_{oh}}$$

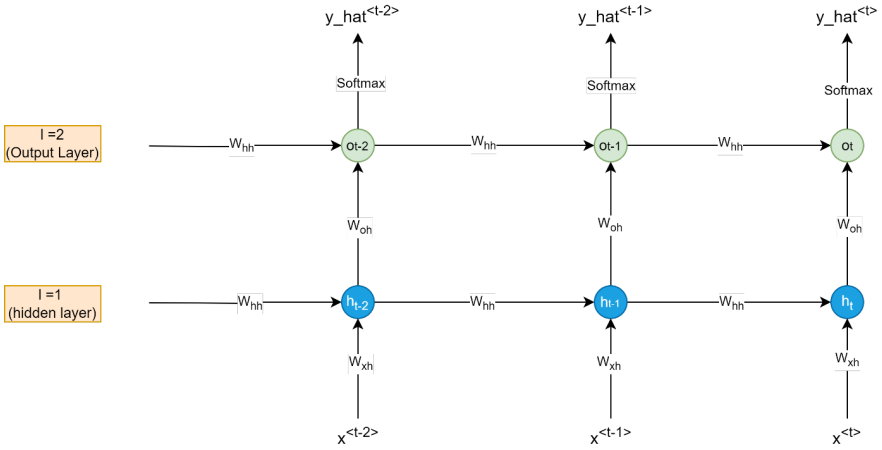


Figure 6.14: RNN unfolded

Next, we compute the derivative of cost with respect to a:

$$da = \frac{\partial E}{\partial a} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a}$$

Similarly, we compute the derivative of cost with respect to W_{hh}:

$$dW_{hh} = \frac{\partial E}{d\hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{dW_{hh}}$$

And, lastly, we compute the derivative of cost with respect to W_{xh}:

$$dW_{xh} = \frac{\partial E}{d\hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{dW_{xh}}$$

For time step $t-1$ ***based on time step*** t***:*** Then, for time step $t-1$, the loss backpropagates from the t-th sequence and from the $t-1$ sequence itself. Now, we have a situation here. Let's be very clear that at time step $t-1$, we are receiving two versions of the loss backpropagated i.e. $da^{(t-1)}$. Now, both versions of losses are used to make updates on other parameters following $a^{(t-1)}$ one by one. Let's first understand the backpropagation from the t-th sequence. During the forward propagation, W_{hh} from the $t-1$ time step propagates forward to the t-th time step along with the activation from the $t-1$ time step i.e. $a^{(t-1)}$. Thus, during the backpropagation, time step $t-1$ receives loss from the t-th step. So, in that case, we compute:

$$da^{(t-1)} = \frac{\partial E}{\partial a^{(t-1)}} = \frac{\partial E}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial a^{(t)}} \cdot \frac{\partial a^{(t)}}{\partial a^{(t-1)}}$$

Because, $a^{(t)}$ is now a function of $a^{(t-1)}$:

Remember, from the forward propagation, we have
for $t = 1, \ldots, T$:

$$z^{[1](t)} = W_{xh}^{[1]} \cdot a_t^{[1]} + W_{hh}^{[1]} \cdot a^{(t-1)} + b^{[1](t)}$$

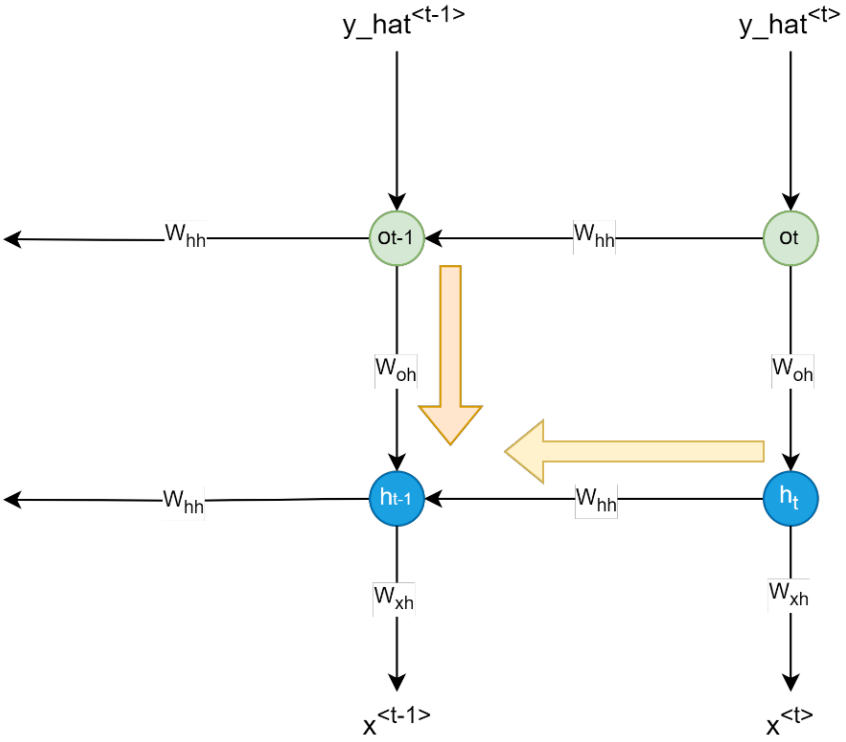
$$a^{(t)} = g_h^{[1]}(z^{[1](t)}) \quad \text{i.e.} \quad a^{(t)} = g_h^{[1]}(W_{xh}^{[1]} \cdot a_t^{[1]} + W_{hh}^{[1]} \cdot a^{(t-1)} + b^{[1](t)})$$

Which means, $a^{(t)}$ is a function of $a^{(t-1)}$.

Now, from the forward propagation, we know:

$$a^{(t-1)} = g_h^{[1]}(W_{xh}^{[1]} \cdot a_0^{[1](t-1)} + W_{hh}^{[1]} \cdot a^{(t-2)} + b^{[1](t-1)})$$

Here, $a_0^{[1](t-1)}$ is the $t-1$ input of the sequence. For example, if our input string is "Hello Sir", "Hello" is the $t-1$-st input and "Sir" is the t-th input. Then, it is obvious to determine, we can compute the next derivative w.r.t. W_{xh} at time step $t-1$.



Note: The derivative we computed for W_{xh} at time step t doesn't work here. We again have to compute:

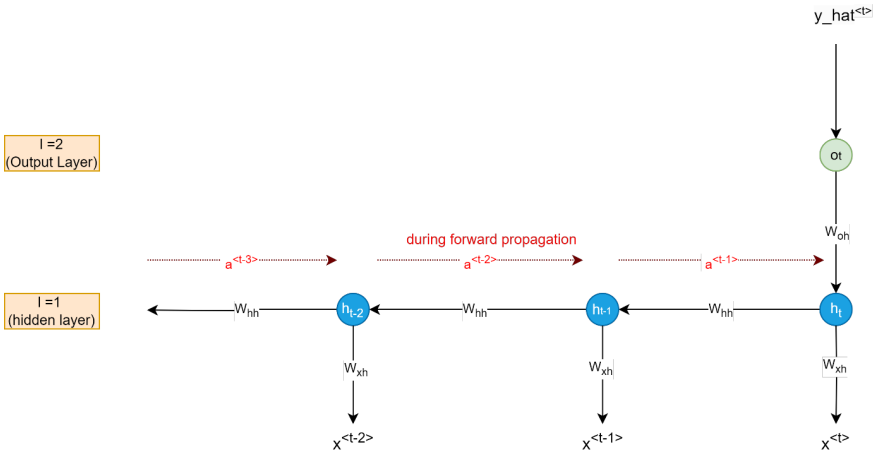
$$dW_{xh} = \frac{\partial E}{\partial W_{xh}} = \frac{\partial E}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial a^{(t)}} \cdot \frac{\partial a^{(t)}}{\partial a^{(t-1)}} \cdot \frac{\partial a^{(t-1)}}{\partial W_{xh}}$$

But, we can accumulate the derivative W_{xh} from all the sequences and write as:

$$dW_{xh} = \frac{\partial E}{\partial W_{xh}} = \frac{\partial E}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial a^{(t)}} \cdot \frac{\partial a^{(t)}}{\partial a^{(t-1)}} \cdot \frac{\partial a^{(t-1)}}{\partial W_{xh}} + \frac{\partial E}{d\hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial a^{(t)}} \cdot \frac{\partial a^{(t)}}{\partial W_{xh}}$$

Now, similarly, if we consider three time steps, we get:

$$dW_{xh} = \frac{\partial E}{\partial W_{xh}} = \frac{\partial E}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial a^{(t)}} \cdot \frac{\partial a^{(t)}}{\partial a^{(t-1)}} \cdot \frac{\partial a^{(t-1)}}{\partial a^{(t-2)}} \cdot \frac{\partial a^{(t-2)}}{\partial W_{xh}} + \frac{\partial E}{\partial \hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial a^{(t)}} \cdot \frac{\partial a^{(t)}}{\partial a^{(t-1)}} \cdot \frac{\partial a^{(t-1)}}{\partial W_{xh}} + \frac{\partial E}{d\hat{y}^{(t)}} \cdot \frac{\partial \hat{y}^{(t)}}{\partial a^{(t)}} \cdot \frac{\partial a^{(t)}}{\partial W_{xh}}$$



Generalization

Backward Propagation Through Time is written as:

Thus, for $t = 1, \ldots, T$, from above we can write for W_{xh}:

$$dW_{xh} = \frac{\partial E}{\partial W_{xh}} = \sum_{t=0}^{T} \frac{\partial E}{\partial \hat{y}^{(T-t)}} \cdot \frac{\partial \hat{y}^{(T-t)}}{\partial h^{(T)}} \left(\prod_{j=T-t+1}^{T} \frac{\partial h^{(T-j+1)}}{\partial h^{(T-j)}} \right) \cdot \frac{\partial h^{(T-(t+1))}}{\partial W_{xh}}$$

Similarly, for W_{hh}:

$$dW_{hh} = \frac{\partial E}{\partial W_{hh}} = \sum_{t=0}^{T} \frac{\partial E}{\partial \hat{y}^{(T-t)}} \cdot \frac{\partial \hat{y}^{(T-t)}}{\partial h^{(T)}} \left(\prod_{j=T-t+1}^{T} \frac{\partial h^{(T-j+1)}}{\partial h^{(T-j)}} \right) \cdot \frac{\partial h^{(T-(t+1))}}{\partial W_{hh}}$$

And for W_{oh}:

$$dW_{oh} = \frac{\partial E}{\partial W_{oh}} = \sum_{t=0}^{T} \frac{\partial E}{\partial \hat{y}^{(T-t)}} \cdot \frac{\partial \hat{y}^{(T-t)}}{\partial h^{(T)}} \left(\prod_{j=T-t+1}^{T} \frac{\partial h^{(T-j+1)}}{\partial h^{(T-j)}} \right) \cdot \frac{\partial h^{(T-(t+1))}}{\partial W_{oh}}$$

Taking into account the bias in each neuron, for b_h:

$$db_h^{(t)} = \sum_{t=0}^{T} \frac{\partial E}{\partial \hat{y}^{(T-t)}} \cdot \frac{\partial \hat{y}^{(T-t)}}{\partial h^{(T-t)}} \cdot \frac{\partial h^{(T-t)}}{\partial b_h}$$

And for b_o:

$$db_o^{(t)} = \sum_{t=0}^{T} \frac{\partial E}{\partial \hat{y}^{(T-t)}} \cdot \frac{\partial \hat{y}^{(T-t)}}{\partial b_o}$$

6.6 Types of RNN

Based on inputs and outputs, there are the following four types of RNN:

1. **One to One**
   RNN with one input and one output.
   E.g. Sequence Prediction.
2. **One to Many**
   RNN with one input but many outputs.
   E.g. Text Generation, Music Generation, etc.
3. **Many to One**
   RNN with many inputs but one output.
   E.g. Sentiment Classification, etc.
4. **Many to Many**
   RNN with many inputs and many outputs.
   E.g. Language Translation, Text Summarization, etc.

6.7 Vanishing/Exploding Gradient

The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long-term dependencies because of multiplicative gradients that can be exponentially decreasing or increasing with respect to the number of layers.

For example, consider the formula derived above. We can see the product of derivatives involved. If the derivative value happens to be very small, between 0 and 1 (e.g., 0.025), and is multiplied

Figure 6.15: One to One RNN

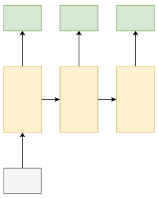


Figure 6.16: One to Many RNN

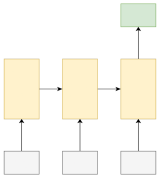


Figure 6.17: Many to One RNN

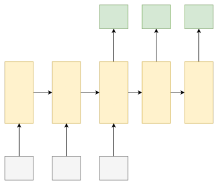


Figure 6.18: Many to Many RNN

by a similar next derivative value (e.g., 0.1), the result is a smaller value. If this is the case, then during the backpropagation from the T-th sequence to the 1st sequence, there will be so many multiplications that the ultimate product value may approach zero. This means the derivative value of the cost C with respect to the weight W will be zero, resulting in no update applied during the update rule:

$$w = w - \alpha \cdot \frac{\partial E}{\partial W} = w \quad [\text{since } \frac{\partial E}{\partial W} = 0]$$

This is called the vanishing gradient, also known as the decay of information over time. Similarly, if

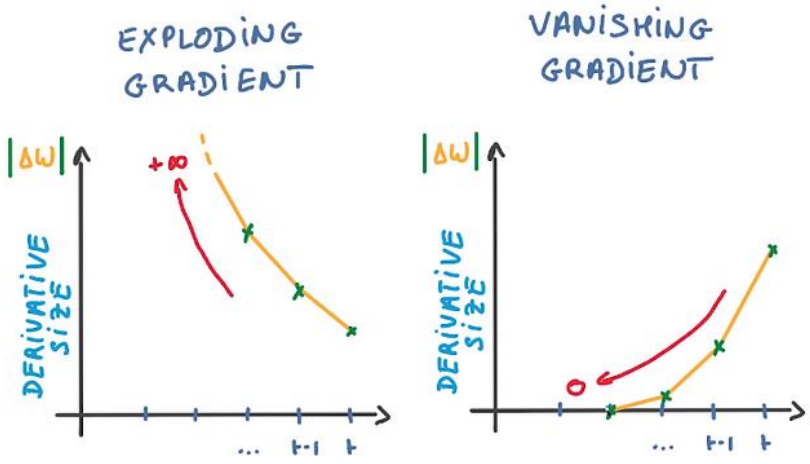


Figure 6.19: Vanishing Gradient

the value of the derivative is greater than 1, the resulting multiplication value will be too large, and the weight update will be significantly influenced by this large value. This is known as the exploding gradient.

Since the sigmoid function results in outputs between 0 and 1, it often leads to vanishing gradients. Therefore, it is not recommended to use the sigmoid function with RNNs. Instead, the tangent function is used with RNNs, which results in outputs between -1 and 1.

6.8 Dropout

Dropout is the concept of temporarily dropping or replacing random neurons from the hidden layer of the network. Dropping a neuron will disable the to and from paths of that neuron. Dropout

is performed only during the training phase but not during testing and prediction. The reason is that dropout is performed in the network to avoid overfitting (memorizing the pattern from data by specific neurons) during the training phase. During dropout, a random number of neurons are dropped, usually 10 or 20% of neurons. Thus, dropout is also known as a type of regularization in deep learning.