Module 2

Overfitting :Regularization Techniques -LASSO and RIDGE:

Overfitting is when your model gets too fit to the training data. Rather than learn patterns it just memorizes the training data. Error during training will be near zero but testing error will be high.

To fix this problem, we need to stop training the model before the test error graph goes up again. To do this we can either stop the model, or penalize the model when it becomes overfit by implementing Generalization using Regularization techniques such as LASSO and RIDGE.

Term	Meaning	Effect
Bias	Error from incorrect assumptions in the learning algorithm (i.e., the model is *too simple*).	High bias → underfitting.
Variance	Error from sensitivity to small fluctuations in the training set (i.e., the model is *too complex*).	High variance → overfitting.

Minimize Loss + Penalty (regularization terms)

In regularized Regression,

```
$$Minimize: \sum(y-{y_i})^2+λ×PenaltyOnβ$$
```

Think of β as *volume knobs* for each feature:

- Turn a knob up → that feature has more influence on prediction.
- Regularization (Ridge/LASSO) gently pushes these knobs down — and in LASSO, some knobs get turned **all the way off** (β = 0).
- **Ridge Regression**:
  - $$Loss = \sum(y-yi)2 + \lambda\sum\beta j^2$$
- **Lasso Regression**:
  - $$Loss = \sum(y-yi)^2 + \lambda\sum|\beta j|$$

Aspect	Ridge (L2)	LASSO (L1)
Penalty term	$(\lambda\sum\beta_j^2)$	$(\lambda\sum\|\beta\|)$
Effect on coefficients	Shrinks but doesn't eliminate	Can shrink to zero
Handles multicollinearity	Very effective	May randomly choose one variable among correlated ones
Feature selection	❌ No	✅ Yes
Use case	Many small/medium effects	Few large effects

Classification Algorithms- Linear and Non linear algorithms

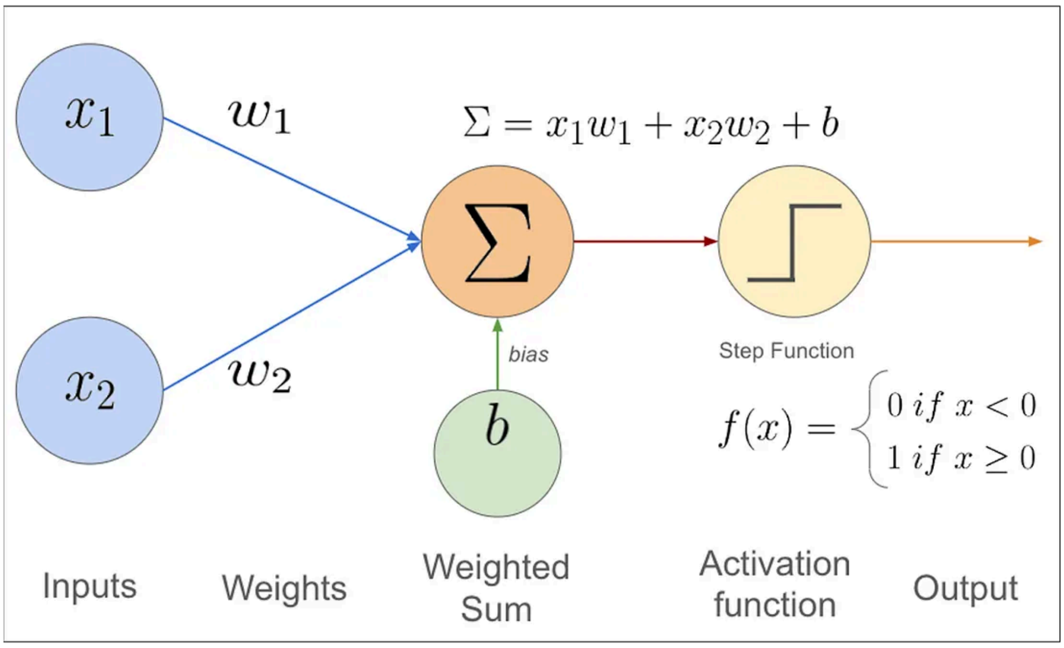
Linear:

Algorithm	Description
Logistic Regression	Uses a sigmoid function to output probabilities; suitable for binary or multinomial classification.
Linear Discriminant Analysis (LDA)	Assumes features follow a normal distribution; seeks to find a linear combination of features that best separates classes.
Support Vector Machine (Linear Kernel)	Finds the hyperplane that maximizes the margin between classes.
Perceptron	A basic single-layer neural network that updates weights based on misclassified samples.

Non Linear:

Algorithm	Description
Kernel SVM (e.g., RBF, Polynomial)	Uses kernel functions to map data into higher-dimensional space where a linear boundary can separate the classes.
Decision Tree	Splits data recursively based on feature values; naturally handles non-linear decision boundaries.
Random Forest	Ensemble of decision trees; reduces overfitting and improves generalization.
k-Nearest Neighbors (kNN)	Classifies based on the majority label of nearest neighbors; decision boundary adapts to data shape.
Neural Networks (Multi-layer Perceptron)	Uses multiple layers of non-linear activations to learn complex patterns.
Naive Bayes (with non-linear features)	Though linear in log-space, it can handle non-linear boundaries if features are non-linearly transformed.

Perceptron



x_1
w_1
$\Sigma = x_1w_1 + x_2w_2 + b$
x_2
w_2
bias
b
Step Function
$f(x) = \begin{cases} 0 \; if \; x < 0 \\ 1 \; if \; x \geq 0 \end{cases}$
Inputs
Weights
Weighted Sum
Activation function
Output

Perceptron is inspired by a biological neuron.

- Take Input
- Apply weight to input
- Sum them up
- Add bias
- Pass them through an activation function
- Produce Output

  Mathematical representation: $f(\sum wixi + b)$

Logistic Regression

- They're tryna fool u, despite its name being regression, this thing is actually used for classification
- It models the probability that a given input belongs to a certain class (e.g., yes/no, spam/not spam, disease/no disease).

The key idea is to predict the probability that y=1y = 1y=1 (the positive class) given an input vector xx.

$$P(y = 1 \mid x) = \sigma(w \cdot x + b)$$

Where:

- $\mathbf{w} = weights$
- b = bias
- $\sigma(z)\sigma(z)\sigma(z) = sigmoid(logistic) function$

The sigmoid squashes any real-valued number into the range (0, 1):

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

Naive Bayes

Naïve Bayes is a **classification algorithm based on probability (Bayes' theorem)**, with the **"naïve" assumption** that all features are **independent** given the class label

Type	When to Use	How it Works
Multinomial Naïve Bayes	Discrete counts (e.g., word counts in text)	Models word frequency; popular for text classification.
Bernoulli Naïve Bayes	Binary features (e.g., word presence/absence)	Useful when features are boolean.
Gaussian Naïve Bayes	Continuous features	Assumes features follow a normal (Gaussian) distribution.
Complement Naïve Bayes	Text data with class imbalance	A variation of Multinomial NB that's more robust for imbalanced datasets.

If a feature value never appears in training for a class, probability becomes 0 → handled using **Laplace smoothing**

Decision Tree

1. Basics

What a **Decision Tree** is and how it works (recursive splitting of data).
- A decision tree splits data recursively based on feature values to create **homogeneous subsets**.
- **Goal:** Minimize impurity / maximize information gain.

Term	Meaning
Node	A decision point (test on an attribute).

Term	Meaning
Edge/Branch	Outcome of a test.
Leaf Node	Final class/decision.
Root Node	Top of the tree (first split).

2. Entropy (Measure of Impurity)

$Entropy(S) = -\sum_{i=1}^{k} p_i \log_2(p_i)$

Range: 0 (pure) → 1 (maximally impure for binary classification).

- Types: Classification trees vs Regression trees.
- Advantages, disadvantages, and when to use them.
-

3. Information Gain (ID3)

$$IG(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|Sv|}{|S|} Entropy(Sv)$$

Choose attribute A with the highest Information Gain for splitting.

Gini Index (CART)

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

2. Splitting Criteria (Math Core)

- Entropy and Information Gain (ID3 algorithm).
- Gini Index (used in CART).
- Gain Ratio (used in C4.5).
- Example calculations — how to pick the best attribute for a split.

3. Algorithms

- ID3 Algorithm: Based on Information Gain.
-
- CART (Classification and Regression Trees): Uses Gini index or variance reduction.

4. Pruning Techniques

- Pre-pruning (early stopping).
- Post-pruning (Reduced Error, Cost Complexity pruning).
- Why pruning helps avoid overfitting-
  - Makes the tree more generalized otherwise it becomes too complex and sensitive to noise during the training process.

Neural Networks - Concept of artificial neuron

1. Basic Idea

An artificial neuron (also called a *perceptron* or *node*) is a computational model inspired by biological neurons.

It:

- Takes inputs
- Multiplies them by weights
- Adds a bias
- Passes the result through an activation function

2. Neuron Model

Inputs: `x₁, x₂, ..., xₙ`
Weights: `w₁, w₂, ..., wₙ`
Bias: `b`

Equations:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$y = f(z)$$

where $f(z)$ is the activation function.

3. Activation Functions

Function	Formula	Output Range	Notes
Step	$f(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$	{0,1}	Simple perceptron
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	(0,1)	Probabilistic output
Tanh	$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	(-1,1)	Zero-centered
ReLU	$f(z) = \max(0, z)$	[0, ∞)	Fast, avoids vanishing gradient
Leaky ReLU	$f(z) = \max(0.01z, z)$	(-∞, ∞)	Prevents dead ReLU

Function	Formula	Output Range	Notes
Softmax	$f_i(z) = \frac{e^{z_i}}{\sum_j e^{z_j}}$	(0,1), Σ=1	Multi-class output layer

4. Perceptron Learning Rule

For each training example:

$$w_i^{new} = w_i^{old} + \eta(t - y)x_i$$

$$b^{new} = b^{old} + \eta(t - y)$$

where:

- η = learning rate
- t = target output
- y = predicted output

Goal: minimize $(t - y)$.

5. Error Function

Mean Squared Error (MSE):

$$E = \frac{1}{2}\sum(t - y)^2$$

6. Gradient Descent Update

General rule:

$$w_i^{new} = w_i^{old} - \eta\frac{\partial E}{\partial w_i}$$

→ Move weights in the direction that reduces error.

7. Key Concepts

Term	Meaning
Weight (w)	Strength of connection between input and neuron
Bias (b)	Shifts activation threshold
Activation (y)	Output after applying activation function
Learning Rate (η)	Controls step size in weight updates
Epoch	One full pass through the dataset
Loss Function	Measures how wrong the model is

8. Advantages

- Learns **non-linear** decision boundaries
- Universal function approximator
- Foundation for **Deep Learning**

9. Limitations

- Needs **large data** and tuning
- **Slow** to train
- Acts as a **black box**

Tldr byheart this

$$z = \sum_i w_i x_i + b$$

$$y = f(z)$$

$$w \leftarrow w + \eta(t - y)x$$

Feed Forward Neural Network

1. Concept Overview

A Feed Forward Neural Network (FFNN) is an artificial neural network where information moves in one direction — from input layer → hidden layer(s) → output layer.
There are no cycles or loops.

2. Architecture

Layers:

1. **Input layer:** Receives input features $(x_1, x_2, \ldots, x_n)$.
2. **Hidden layer(s):** Perform weighted transformations and apply activation functions.
3. **Output layer:** Produces final predictions.

Information flow:

$$x \to h^{(1)} \to h^{(2)} \to \cdots \to y$$

3. Forward Propagation (Mathematical Model)

For each layer l:

Weighted sum:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

Activation:

$$a^{(l)} = f^{(l)}(z^{(l)})$$

where:

- $W^{(l)}$ = weight matrix for layer l
- $b^{(l)}$ = bias vector for layer l
- $a^{(l-1)}$ = activations (outputs) from the previous layer
- $f^{(l)}$ = activation function (Sigmoid, ReLU, etc.)

For the **output layer:**

$$\hat{y} = f^{(L)}(z^{(L)})$$

4. Activation Functions (Same thing if u read it already ignore)

Function	Formula	Output Range	Notes
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	(0,1)	Probabilities
Tanh	$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	(-1,1)	Zero-centered
ReLU	$f(z) = \max(0, z)$	[0,∞)	Speeds up training
Softmax	$f_i(z) = \frac{e^{z_i}}{\sum_j e^{z_j}}$	(0,1), Σ=1	Used in multi-class output

5. Loss (Cost) Functions

Task	Common Loss Function	Formula
Regression	Mean Squared Error (MSE)	$E = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$
Binary Classification	Binary Cross-Entropy	$E = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$
Multi-class Classification	Categorical Cross-Entropy	$E = -\sum_i y_i \log(\hat{y}_i)$

6. Backpropagation (Weight Update Rule)

Goal: minimize the loss E by adjusting weights using gradient descent.

For each weight w:

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

where:

- η = learning rate
- $\frac{\partial E}{\partial w}$ = gradient of error w.r.t. weight

Gradients are computed using the chain rule:

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

Error term for output layer:

$$\delta^{(L)} = (\hat{y} - y) \odot f'^{(L)}(z^{(L)})$$

Error term for hidden layers:

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot f'^{(l)}(z^{(l)})$$

7. Training Algorithm (Summary)

1. **Initialize** weights and biases randomly
2. **Forward Pass:** Compute activations for all layers
3. **Compute Loss:** Compare $\hat{y}$ and true y
4. **Backward Pass:** Compute gradients (backpropagation)
5. **Update Weights:** $W \leftarrow W - \eta \frac{\partial E}{\partial W}$
6. **Repeat** for all epochs

8. Bias-Variance Tradeoff

- **Small network:** High bias, underfits
- **Large/deep network:** High variance, overfits
- Use **regularization** (L2, dropout, early stopping) to balance.

9. Advantages

- Learns **complex, non-linear** relationships
- Flexible architecture
- Basis for deep learning models (CNNs, RNNs, etc.)

10. Limitations

- Computationally expensive
- Prone to overfitting
- Requires tuning (learning rate, layers, etc.)

Quick Recall

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = f^{(l)}(z^{(l)})$$
$$E = \frac{1}{2} \sum (y - \hat{y})^2$$
$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Backpropagation Algorithms

1. Concept Overview

Backpropagation is the **learning algorithm** used to train feed-forward neural networks.
It computes how much each weight contributed to the overall error and updates weights to **minimize the loss** using **Gradient Descent**.

Key idea:

> Propagate the error backward from the output layer → hidden layers → input layer.

2. Network Setup

For each layer l:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = f^{(l)}(z^{(l)})$$

where:

- $a^{(0)} = x$ (input layer)
- $a^{(L)} = \hat{y}$ (output layer prediction)

3. Loss (Error) Function

For a single training sample:

$$E = \frac{1}{2}\sum_k (y_k - \hat{y}_k)^2$$

or, for classification tasks:

$$E = -\sum_k y_k \log(\hat{y}_k)$$

---

4. Steps in Backpropagation

Step 1: Forward Pass

Compute activations layer by layer from input → output.

Step 2: Compute Output Error

For the output layer L:

$$\delta^{(L)} = (\hat{y} - y) \odot f'^{(L)}(z^{(L)})$$

where $\odot$ = element-wise multiplication.

Step 3: Propagate Error Backward

For each hidden layer $l = L-1, L-2, \ldots, 1$:

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot f'^{(l)}(z^{(l)})$$

This computes how much each neuron contributed to the total error.

Step 4: Compute Gradients

For each weight:

$$\frac{\partial E}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

$$\frac{\partial E}{\partial b^{(l)}} = \delta^{(l)}$$

Step 5: Update Weights (Gradient Descent)

Update all weights and biases:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial E}{\partial W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial E}{\partial b^{(l)}}$$

where η = learning rate.

---

5. Summary (Algorithm Pseudocode)

Initialize weights W, biases b randomly
Repeat for each epoch:

1. Forward propagate inputs to compute outputs
2. Compute error E = Loss(y, ŷ)
3. Backward propagate errors:

   $\delta(L) = (\hat{y} - y) \odot f'(z(L))$

   $\delta(l) = (W(l+1))^T \delta(l+1) \odot f'(z(l))$
4. Compute gradients:

   $\partial E / \partial W(l) = \delta(l)(a(l-1))^T$
5. Update weights:

   ```
   $W(l) ← W(l) − η ∂E/∂W(l)$
   ```

   Until convergence

---

6. Intuition

- **Forward Pass:** Compute predictions.
- **Backward Pass:** Measure blame for each weight.
- **Update:** Move weights slightly in the direction that reduces total error.

Think of it as **error correction** flowing backward.

---

7. Common Variants

Variant	Description
Batch Gradient Descent	Update after entire dataset
Stochastic Gradient Descent (SGD)	Update after each sample
Mini-batch Gradient Descent	Update after small subsets
Momentum / Adam	Use previous gradients for faster convergence

---

8. Key Formulas (unfortunately u gotta byheart this)

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = f^{(l)}(z^{(l)})$$

$$\delta^{(L)} = (\hat{y} - y) \odot f'^{(L)}(z^{(L)})$$

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot f'^{(l)}(z^{(l)})$$

$$W^{(l)} \leftarrow W^{(l)} - \eta \delta^{(l)} (a^{(l-1)})^T$$
