CS-470: Machine Learning

Week 4 — Logistic Regression and Softmax Regression

Instructor: Dr. Sajjad Hussain

Department of Electrical and Computer Engineering
SEECS, NUST

October 2nd, 2025

Outline

Week 03 Recap: Model Evaluation and Regularization

- **Underfitting vs. Overfitting:**
  - **Underfitting:** Model too simple → high bias.
  - **Overfitting:** Model too complex → high variance.
  - **Goal:** Find the right balance between bias and variance.
- **Learning Curves:**
  - Plot of training vs. validation error as data increases.
  - Diagnose model performance:
    - Both errors high → Underfitting.
    - Large gap between errors → Overfitting.
- **Regularized Linear Models:**
  - **Ridge Regression:** L2 penalty (shrinks weights).
  - **Lasso Regression:** L1 penalty (feature selection).
  - **Elastic Net:** Combination of L1 and L2 penalties.

Key takeaway: Regularization helps control overfitting and improves model generalization.

Logistic Regression — Introduction

Logistic Regression is one of the most popular algorithms for **binary classification problems** — problems where the output can only be one of two categories (e.g., spam / not spam, pass / fail, yes / no).

Although it has "regression" in its name, Logistic Regression is actually used for **classification**, not prediction of continuous values.

It works by estimating the **probability** that a given input x belongs to a particular class (often labeled as 1, or "positive").

Logistic Regression: Predicting Class Membership

Unlike Linear Regression, which predicts continuous values, **Logistic Regression** predicts whether an instance belongs to one of two classes.

Class Definitions:

$$y = \begin{cases} 1 & \text{Positive class (e.g., "Pass", "Spam", "Admitted")} \\ 0 & \text{Negative class (e.g., "Fail", "Not Spam", "Rejected")} \end{cases}$$

The model computes a weighted sum of the input features:

$$t = \theta^T x$$

But instead of using t directly, Logistic Regression converts it into a probability value that always lies between 0 and 1 using the **sigmoid (logistic) function**.

The Sigmoid (Logistic) Function

The **sigmoid function** maps any real-valued number into the range (0, 1), making it ideal for expressing probabilities.

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

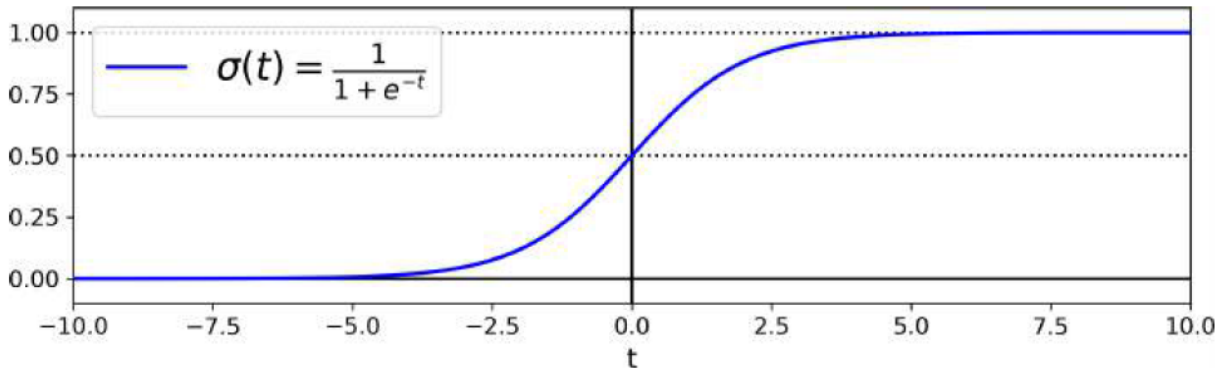


Figure: Sigmoid function $\sigma(t)$ mapping any real input t into (0, 1).

The Sigmoid (Logistic) Function

Interpretation:

- When t is large and positive, $\sigma(t) \approx 1 \rightarrow$ Strong confidence in the **positive class**.
- When t is large and negative, $\sigma(t) \approx 0 \rightarrow$ Strong confidence in the **negative class**.
- When $t = 0$, $\sigma(t) = 0.5 \rightarrow$ Model is uncertain (decision boundary).

Hence, Logistic Regression predicts the probability that an instance belongs to the positive class ($y = 1$):

$$\hat{p} = h_\theta(x) = \sigma(\theta^T x)$$

Why Do We Need a Cost Function?

In Logistic Regression, the model predicts a probability $\hat{p} = h_\theta(x)$ that an instance belongs to the **positive class** $(y = 1)$.

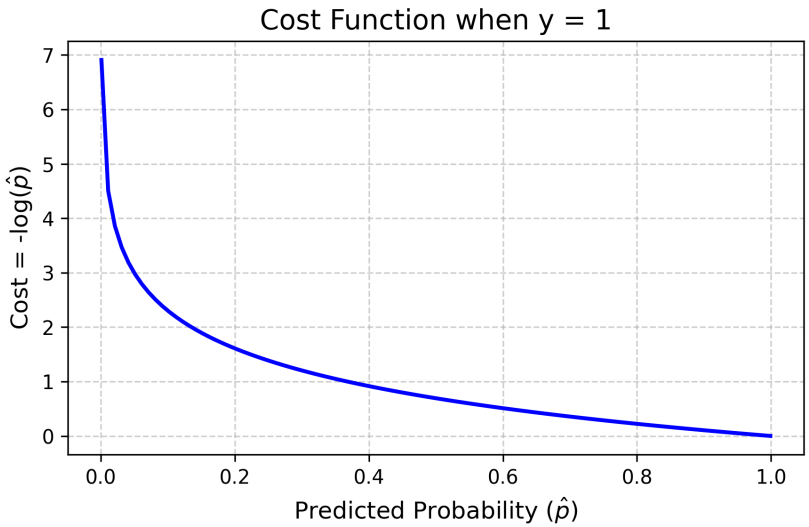
- We need a function to measure how **bad** a prediction is.
- For regression, we used Mean Squared Error (MSE), but it doesn't work well for probabilities (nonlinear scale).
- We want a cost that:
  - Is small when prediction is correct.
  - Grows rapidly when prediction is confidently wrong.

Let's first see what happens when the true class is $y = 1$.

Understanding Cost when Actual Class = 1

When the true label $y = 1$:

- If $\hat{p}$ (predicted probability) is close to 0, the model is very wrong — cost should be very high.
- If $\hat{p}$ is around 0.5, the model is uncertain — cost is moderate.
- If $\hat{p}$ is close to 1, the model is confident and correct — cost should be near 0.



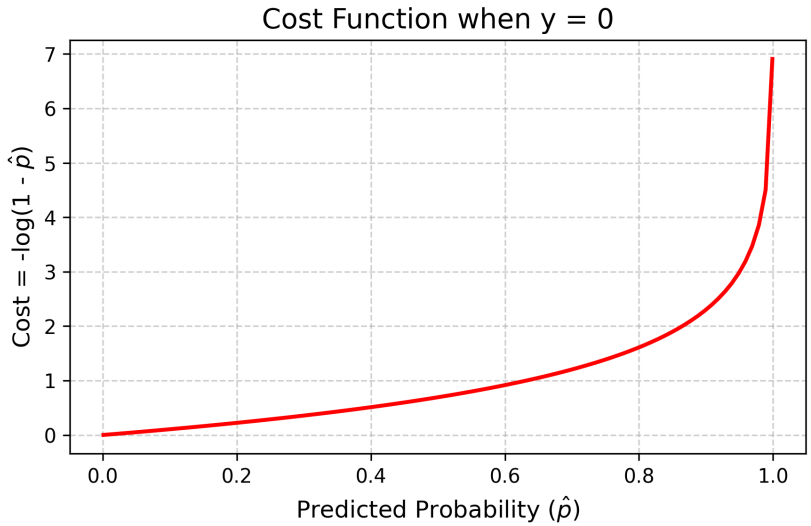
Mathematically, this relationship can be captured by the cost function:

$$\text{Cost}(\hat{p}, y = 1) = -\log(\hat{p})$$

Understanding Cost when Actual Class = 0

When the true label $y = 0$:

- If $\hat{p}$ (predicted probability) is close to 1, the model is very wrong — cost should be very high.
- If $\hat{p}$ is around 0.5, the model is uncertain — cost is moderate.
- If $\hat{p}$ is close to 0, the model is confident and correct — cost should be near 0.



Mathematically, this relationship can be captured by the cost function:

$$\text{Cost}(\hat{p}, y = 0) = -\log(1 - \hat{p})$$

Combining Both Cases

To generalize, we combine both scenarios ($y = 1$ and $y = 0$) into one expression:

$$\text{Cost}(y, \hat{p}) = -\left[y \log(\hat{p}) + (1 - y) \log(1 - \hat{p})\right]$$

Why this works:

- If $y = 1$, the second term becomes 0 $\rightarrow$ Cost $= -\log(\hat{p})$
- If $y = 0$, the first term becomes 0 $\rightarrow$ Cost $= -\log(1 - \hat{p})$

This is called the Binary Cross-Entropy (Log Loss).

Binary Cross-Entropy (Log Loss) Summary

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

Key points:

- Penalizes wrong predictions more when the model is confident.
- Ensures smooth, convex optimization surface.
- Works naturally with probabilities from the sigmoid function.

Training the Logistic Regression Model

Now that we have defined the cost function, our goal is to find the parameters θ that minimize this cost.

Bad news:

- Unlike Linear Regression, Logistic Regression has **no closed-form solution**.
- The sigmoid function makes the cost function non-linear in θ.

Good news:

- The cost function is **convex** — it has a single global minimum.
- We can find the optimal θ values efficiently using **Gradient Descent**.

Goal: Iteratively update θ in the direction that **reduces the cost function**.

Computing the Gradients

For logistic regression:

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

where $\hat{p}^{(i)} = \sigma(\theta^T x^{(i)})$

The partial derivative (gradient) for parameter θ_j is:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(\theta^T x^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

Gradient Descent update rule:

$$\theta_j := \theta_j - \eta\frac{\partial J(\theta)}{\partial \theta_j}$$

where η is the learning rate.

Deriving the Gradient using the Chain Rule

We want to compute the gradient of the cost function with respect to the model parameters θ.

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

where $\hat{p}^{(i)} = \sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$ and $z^{(i)} = \theta^T x^{(i)}$.

We apply the **chain rule**:

$$\frac{\partial J}{\partial \theta_j} = \frac{\partial J}{\partial \hat{p}^{(i)}} \cdot \frac{\partial \hat{p}^{(i)}}{\partial z^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial \theta_j}$$

Simplifying the Gradient Expression

Each term can be computed as:

$$\frac{\partial J}{\partial \hat{p}^{(i)}} = -\frac{y^{(i)}}{\hat{p}^{(i)}} + \frac{1 - y^{(i)}}{1 - \hat{p}^{(i)}}, \quad \frac{\partial \hat{p}^{(i)}}{\partial z^{(i)}} = \hat{p}^{(i)}(1 - \hat{p}^{(i)}), \quad \frac{\partial z^{(i)}}{\partial \theta_j} = x_j^{(i)}$$

Now, substitute the individual terms back into the chain rule:

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \left[\left(\hat{p}^{(i)} - y^{(i)} \right) x_j^{(i)} \right]$$

Thus, the **gradient vector** for all parameters is:

$$\nabla_\theta J(\theta) = \frac{1}{m} X^T \left(\hat{p} - y \right)$$

Gradient Descent Update:

$$\theta := \theta - \eta \nabla_\theta J(\theta)$$

From Binary to Multiclass Classification

So far, **Logistic Regression** predicts only between two classes ($y = 0$ or $y = 1$). But what if we have more than two classes?

Example: Predicting the type of fruit $\{\text{Apple}, \text{Banana}, \text{Mango}\}$

A simple approach:

- Train one Logistic Regression classifier per class.
- Each classifier predicts the probability that the instance belongs to its class vs. all others.
- This is called the **One-vs-Rest (OvR)** or **One-vs-All** strategy.

For 3 classes, we train 3 classifiers:

$$p_1 = P(y = 1|x), \quad p_2 = P(y = 2|x), \quad p_3 = P(y = 3|x)$$

and pick the class with the highest probability:

$$\hat{y} = \arg\max_k p_k$$

Limitation of One-vs-Rest Approach

Although One-vs-Rest works, it has limitations:

- Probabilities from different classifiers may not sum to 1.
- Each model is trained independently — no shared information between classes.
- Inconsistent or conflicting probabilities may occur.

We need a unified model that:

- Handles all classes together.
- Produces valid probability distributions.
- Generalizes the idea of Logistic Regression.

Solution → Softmax Regression (a.k.a. Multinomial Logistic Regression)

Idea of Softmax Regression

Softmax Regression extends Logistic Regression to multiple classes by:

- Assigning a separate weight vector $\theta^{(k)}$ for each class k.
- Computing a score $s_k(x) = \theta^{(k)T} x$ for each class.

Then, the **softmax function** converts these scores into probabilities:

$$P(y = k \mid x) = \frac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{s_j(x)}}$$

These probabilities:

- Are always positive.
- Sum to 1 across all K classes.

Prediction rule:

$$\hat{y} = \arg\max_k \; P(y = k \mid x)$$

Why Exponentials in Softmax?

Each class k produces a score:

$$s_k(x) = \theta^{(k)T} x$$

These scores can be any real numbers — positive, negative, or large.
Problem: We need to convert these scores into probabilities that are:

- Positive
- Sum to 1

Solution: Exponentiate the scores — this makes large scores dominate while keeping all values positive:

$$e^{s_k(x)} > 0$$

Then normalize by dividing by the sum across all classes:

$$P(y = k \mid x) = \frac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{s_j(x)}}$$

Intuition Behind the Softmax Probabilities

Example: Suppose a model produces the following scores for 3 classes:

$$s_1 = 2.0, \quad s_2 = 1.0, \quad s_3 = 0.1$$

Without Softmax: The raw scores are not interpretable as probabilities.

After Softmax:

$$P_1 = \frac{e^{2.0}}{e^{2.0} + e^{1.0} + e^{0.1}} \approx 0.66$$

$$P_2 \approx 0.24, \quad P_3 \approx 0.10$$

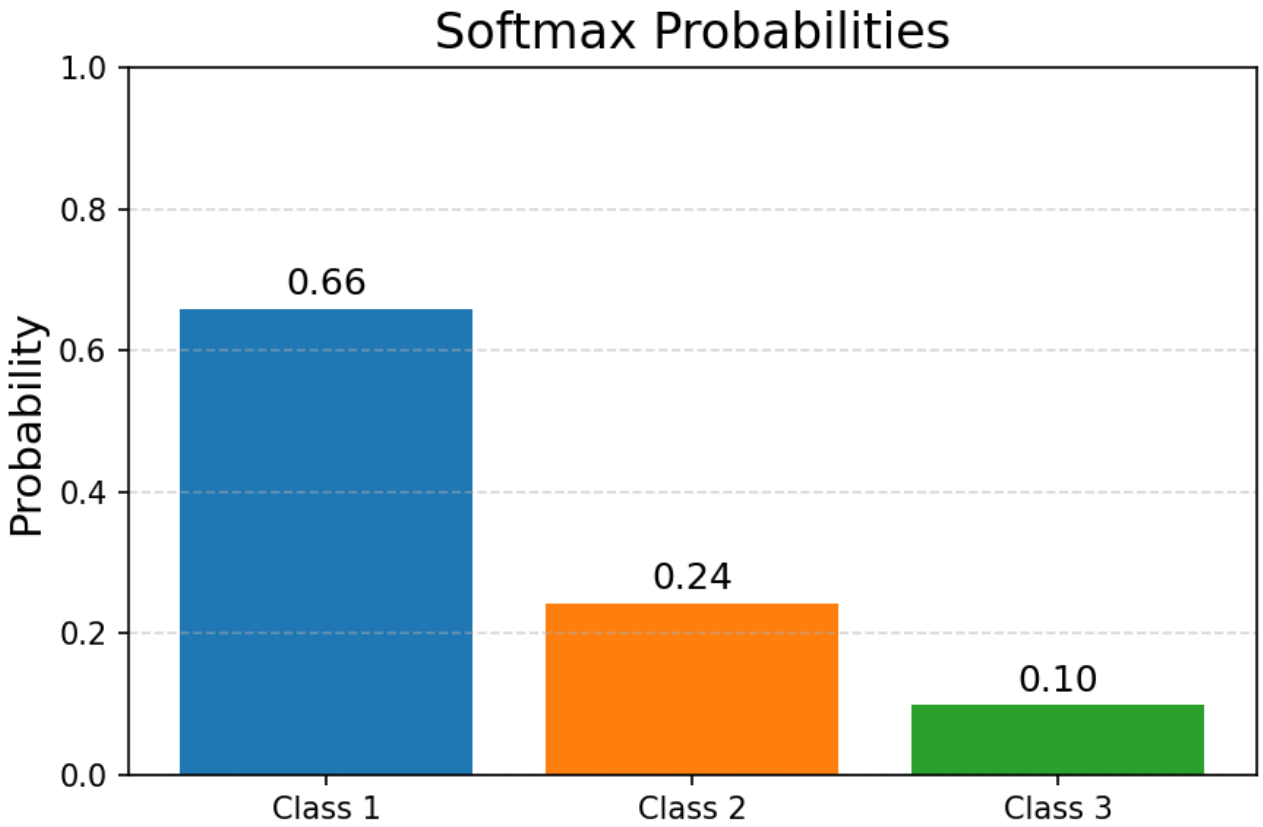
Interpretation:

- Class 1 has the highest probability $\rightarrow$ predicted class.
- Differences in s_k are smoothly transformed into probabilities.

Softmax Output as a Probability Distribution

Key Properties:

- $0 < P(y = k|x) < 1$ for all k
- $\sum_k P(y = k|x) = 1$



Cross-Entropy Cost Function

Since the model estimates probabilities, the goal is to train it to assign:

- High probability to the target class
- Low probability to other classes

The **cross-entropy cost function** achieves this by penalizing wrong predictions:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_i^k \log \hat{p}_i^k$$

- y_i^k: 1 if instance i belongs to class k, else 0
- $\hat{p}_i^k$: predicted probability that instance i belongs to class k
- m: number of training examples
- For binary classification ($K = 2$), this reduces to Logistic Regression's log loss.

Gradient of Cross-Entropy and Parameter Update

To minimize the cross-entropy, we compute the gradient with respect to each class parameter vector $\theta^{(k)}$:

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} \left(p_i^k - y_i^k\right) x_i$$

- x_i: input feature vector of instance i
- $p_i^k - y_i^k$: difference between predicted probability and true label
- Summing over all examples gives the average gradient

Training process:

1. Compute $\nabla_{\theta^{(k)}} J(\Theta)$ for each class k
2. Update parameters using Gradient Descent
3. Repeat iteratively until $J(\Theta)$ is minimized

Key point: Cross-entropy is convex, so iterative optimization will converge to the global minimum.

Week-04 Summary: Logistic and Softmax Regression

Binary Classification: Logistic Regression

- Estimates probability that an instance belongs to the positive class (y=1).
- Uses the **sigmoid function** to map linear combination of features to [0,1].
- **Cost function: Binary Cross-Entropy** (log loss) penalizes wrong predictions.
- Parameters updated iteratively via **Gradient Descent**.

Multi-class Classification: Softmax Regression

- Generalizes logistic regression to K classes.
- Uses **softmax function** to compute probabilities for each class.
- **Cross-Entropy Loss** measures mismatch between predicted probabilities and true labels.
- Gradients computed per class and updated iteratively via Gradient Descent.