IN402

Machine Learning

Chapter
2

Logistic Regression and Classification

Author: Abbas El-Hajj Youssef

University: Lebanese University

Department: Computer Science Department

These notes extend course materials taught by Prof. Ahmad Faour with additional content from textbooks and supplementary resources.

Disclaimer: *This is not an official course document.*

Contents

Essential Notation (Quick Reference)

m	Number of training examples
n	Number of features (excluding bias)
$\boldsymbol{x}^{(i)}$	Feature vector for i-th example ($\boldsymbol{x} \in \mathbb{R}^{n+1}$ with bias)
$y^{(i)}$	Label for i-th example ($y \in \{0, 1\}$ for binary, $y \in \{1, \dots, K\}$ for multi-class)
$\boldsymbol{\theta}$	Parameter vector (weights and bias)
$h_{\boldsymbol{\theta}}(\boldsymbol{x})$	Hypothesis function (model prediction)
$J(\boldsymbol{\theta})$	Cost function (average loss over dataset)
α	Learning rate for gradient descent

See section A for a complete notation reference.

1 Introduction to Classification Problems

In our previous discussions on linear regression, we focused on predicting continuous-valued outputs (e.g., house prices, temperatures, stock values). We now turn our attention to a fundamentally different type of supervised learning problem: **classification**.

1.1 What is Classification?

Classification is the task of predicting a discrete category or class label for a given input. Unlike regression, where the output is a number on a continuous scale, classification assigns inputs to one of a finite set of categories.

Real-World Classification Examples:

- **Medical Diagnosis:** Is a tumor malignant or benign? (2 classes)
- **Email Filtering:** Is an email spam or not spam? (2 classes)
- **Handwriting Recognition:** Which digit (0-9) is written? (10 classes)
- **Weather Prediction:** Will it be sunny, cloudy, rainy, or snowy? (4 classes)
- **Document Classification:** Is this article about sports, politics, or technology? (3 classes)

1.2 Binary vs. Multi-Class Classification

Classification problems fall into two main categories based on the number of possible output classes:

Binary Classification

A problem where the output variable y can take only two possible values, typically labeled as:

- $y = 1$ (positive class): The event of interest occurs
- $y = 0$ (negative class): The event of interest does not occur

Examples: Spam detection (spam vs. not spam), medical diagnosis (disease vs. no disease).

Multi-Class Classification

A problem where the output variable y can take one of $K > 2$ possible discrete values, typically labeled as $y \in \{1, 2, 3, \ldots, K\}$.
Examples: Handwritten digit recognition (0-9), weather forecasting (sunny, cloudy, rainy).

In this first part, we'll thoroughly develop binary classification using logistic regression, and later extend these concepts to handle multi-class problems.

1.3 Why Linear Regression Fails for Classification

A natural first question is: "Can't we just use linear regression for classification?" Let's explore why this seemingly simple approach fails.

Problems with this approach:

1. **Unbounded outputs:** Linear regression can produce predictions like 5.7 or −2.3, which don't correspond to meaningful class labels.
2. **No probability interpretation:** The outputs are not constrained to the $[0, 1]$ range required for probabilities.
3. **Sensitivity to outliers:** A single extreme data point can drastically shift the regression line and the decision threshold, as shown in fig. 1.

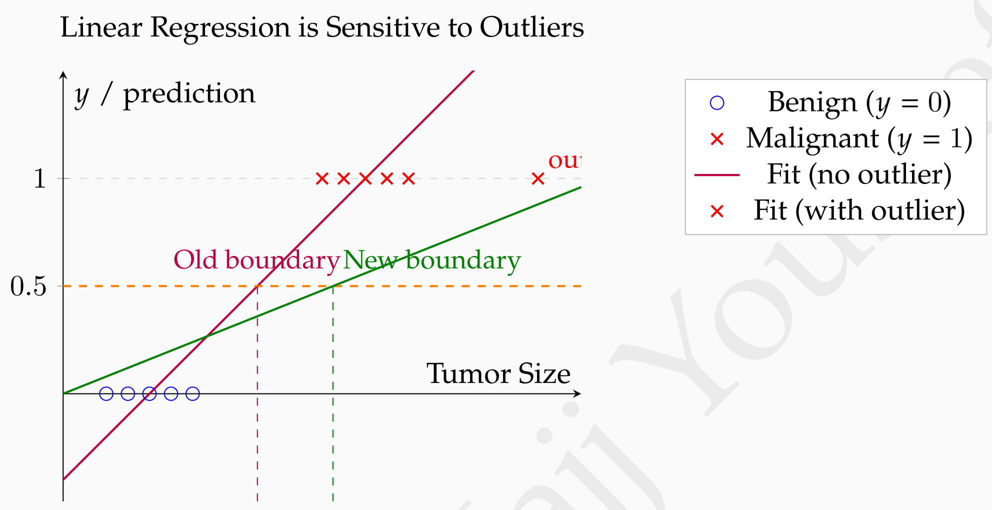


Figure 1: *The purple line fits the original data well. The green line shows how a single outlier shifts the fit and misclassifies a malignant tumor at size=6.*

2 The Logistic Regression Model

2.1 From Linear Output to Probabilities: The Sigmoid Function

A linear regression model, $\boldsymbol{\theta}^T\boldsymbol{x}$, can produce any real number from $-\infty$ to $+\infty$. However, for classification, we need a probability, which must be between 0 and 1. This creates a fundamental mismatch.

We need a function that maps the unbounded output of the linear model to the $[0, 1]$ range of probabilities. The **sigmoid function** (or **logistic function**) does this perfectly. But its choice is not arbitrary–it arises naturally from the statistical concept of **log-odds**.

2.2 The Statistical Foundation: From Probability to Log-Odds

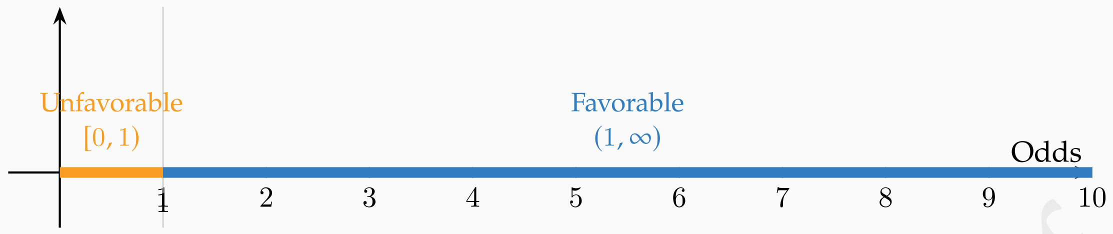
The structure of logistic regression emerges from a key assumption: the **log-odds** of the outcome are a linear function of the features. To understand this, let's build from probability to odds, and finally to log-odds.

First, let $p = P(y = 1 \mid \boldsymbol{x})$ be the probability of the positive class. The **Odds** are the ratio of the probability of an event happening to it not happening.

$$\text{Odds} = \frac{p}{1-p}$$

For example, if $p = 0.8$ (80% chance of winning), the odds are $\frac{0.8}{0.2} = 4$, or "4 to 1 in favor." While intuitive, the scale of odds is asymmetric: the range for unfavorable events ($p < 0.5$) is squashed into $[0, 1)$, while the range for favorable events ($p > 0.5$) stretches from $(1, \infty)$.

The Asymmetric Scale of Odds

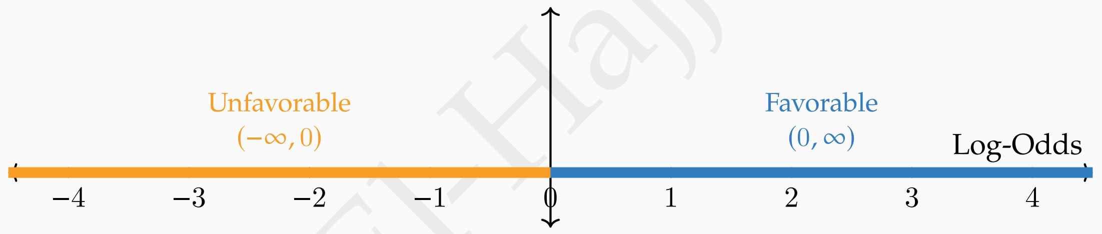


This asymmetry makes it difficult to model odds with a linear function. To fix this, we take the natural logarithm of the odds. This transformation is called the **Log-Odds** or **Logit**.

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

This simple step creates a symmetric, unbounded scale that perfectly matches the output of a linear model. For instance, odds of 4 and 0.25 become $\log(4) \approx +1.39$ and $\log(0.25) \approx -1.39$, which are perfectly symmetric around 0. The log-odds maps a probability from $[0, 1]$ to the entire real number line $(-\infty, +\infty)$.

The Symmetric Scale of Log-Odds



This transformation provides the crucial link: we can now model the **log-odds** with a linear equation.

Deriving the Sigmoid Function from Log-Odds

The core assumption of logistic regression is that the log-odds are a linear function of the features $\boldsymbol{x}$:

$$\log\left(\frac{p}{1-p}\right) = \boldsymbol{\theta}^T\boldsymbol{x}$$

By solving this equation for p, we can recover the probability. This process naturally gives us the sigmoid function.

$$\frac{p}{1-p} = e^{\boldsymbol{\theta}^T\boldsymbol{x}} \tag{1}$$

$$p = (1-p)e^{\boldsymbol{\theta}^T\boldsymbol{x}} \tag{2}$$

$$p(1 + e^{\boldsymbol{\theta}^T\boldsymbol{x}}) = e^{\boldsymbol{\theta}^T\boldsymbol{x}} \tag{3}$$

$$p = \frac{e^{\boldsymbol{\theta}^T\boldsymbol{x}}}{1 + e^{\boldsymbol{\theta}^T\boldsymbol{x}}} = \frac{1}{1 + e^{-\boldsymbol{\theta}^T\boldsymbol{x}}} \tag{4}$$

This final expression, $p = \sigma(\boldsymbol{\theta}^T\boldsymbol{x})$, is exactly the sigmoid function! Thus, the sigmoid is not an arbitrary choice; it is the direct consequence of modeling log-odds linearly.

Interpreting the Coefficients

From eq. (1), the odds of the positive class are

$$\text{Odds} = e^{\boldsymbol{\theta}^T \boldsymbol{x}}.$$

Each coefficient θ_j indicates how its feature x_j influences the prediction:

- Increasing x_j by one unit (holding others fixed) adds θ_j to the **log-odds**.
- Equivalently, it multiplies the **odds** by e^{θ_j} — the **odds ratio**.

Example: Exam Success Predicting whether a student passes an exam ($y = 1$) from the number of **study hours** (x):

$$\log\left(\frac{p}{1-p}\right) = -2 + 0.8x$$

Each extra study hour increases the log-odds by 0.8, or multiplies the odds of passing by $e^{0.8} \approx 2.2$ — roughly **doubling the odds of success**.

The Sigmoid Function

The sigmoid function, $\sigma(z)$, is defined as

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

where $z \in (-\infty, +\infty)$ is the output of the linear model, $z = \boldsymbol{\theta}^T \boldsymbol{x}$.

Key Properties:

- Maps any real number to the interval $(0, 1)$.
- $\sigma(0) = 0.5$, a natural decision threshold.
- Symmetric around $(0, 0.5)$: $\sigma(-z) = 1 - \sigma(z)$.
- Simple derivative for optimization: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

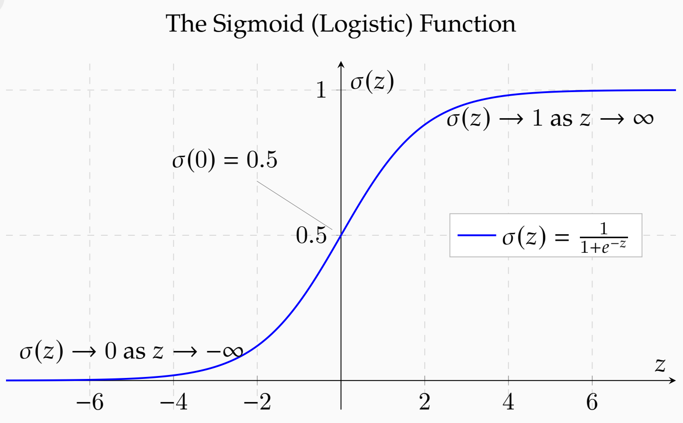


Figure 2: *The sigmoid function smoothly maps any real number to the interval $(0, 1)$, making it ideal for representing probabilities.*

2.3 The Hypothesis Function

We combine the linear model with the sigmoid function to create our hypothesis for logistic regression.

Logistic Regression Hypothesis

The hypothesis function $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ maps a feature vector $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$ to the estimated probability that $y = 1$:

$$\begin{aligned} h_{\boldsymbol{\theta}}(\boldsymbol{x}) &= \sigma(\theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n), \quad (x_0 = 1) \qquad (5) \\ &= \sigma(\boldsymbol{\theta}^T \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \boldsymbol{x}}}. \qquad (6) \end{aligned}$$

This represents the probability of the positive class:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = P(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}),$$

Read as: "probability that y equals 1, given input $\boldsymbol{x}$, parameterized by $\boldsymbol{\theta}$"
Consequently, the probability of the negative class is

$$P(y = 0 \mid \boldsymbol{x}; \boldsymbol{\theta}) = 1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}).$$

2.4 Decision Boundaries

To make a discrete prediction, we set a threshold on the probability. The standard threshold is 0.5.

$$\begin{aligned} &\text{Predict } y = 1 \quad \text{if } h_{\boldsymbol{\theta}}(\boldsymbol{x}) \geq 0.5 \quad \Longleftrightarrow \quad \boldsymbol{\theta}^T \boldsymbol{x} \geq 0 \\ &\text{Predict } y = 0 \quad \text{if } h_{\boldsymbol{\theta}}(\boldsymbol{x}) < 0.5 \quad \Longleftrightarrow \quad \boldsymbol{\theta}^T \boldsymbol{x} < 0 \end{aligned}$$

This leads to the concept of a decision boundary.

Decision Boundary

The decision boundary separates regions where we predict $y = 1$ from regions where we predict $y = 0$. It is defined by the set of points where the model is maximally uncertain:

$$\boldsymbol{\theta}^T \boldsymbol{x} = 0 \quad \text{or} \quad \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n = 0.$$

At this boundary, the hypothesis satisfies

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(0) = 0.5,$$

meaning the model assigns equal probability to both classes. Points with $\boldsymbol{\theta}^T \boldsymbol{x} > 0$ predict $y = 1$, while points with $\boldsymbol{\theta}^T \boldsymbol{x} < 0$ predict $y = 0$.
The boundary depends only on the learned parameters $\boldsymbol{\theta}$, not on the data itself.

Examples of Decision Boundaries

Example 1: Linear Decision Boundary

Suppose we have learned parameters $\boldsymbol{\theta} = [-3, 1, 1]^T$ for features x_1, x_2 (with $x_0 = 1$ for the bias). The decision boundary is:

$$-3 + 1 \cdot x_1 + 1 \cdot x_2 = 0 \implies x_1 + x_2 = 3$$

This is a straight line. Points where $x_1 + x_2 \geq 3$ will be classified as $y = 1$, while points where $x_1 + x_2 < 3$ will be classified as $y = 0$.

Example 2: Non-linear Decision Boundary

By engineering polynomial features, we can create non-linear boundaries. Suppose we use features $[1, x_1, x_2, x_1^2, x_2^2]$ and learn parameters $\boldsymbol{\theta} = [-1, 0, 0, 1, 1]^T$. The decision boundary is:

$$-1 + 0 \cdot x_1 + 0 \cdot x_2 + 1 \cdot x_1^2 + 1 \cdot x_2^2 = 0 \implies x_1^2 + x_2^2 = 1$$

This is a circle with radius 1 centered at the origin. Points outside the circle ($x_1^2 + x_2^2 \geq 1$) will be classified as $y = 1$, while points inside will be classified as $y = 0$.

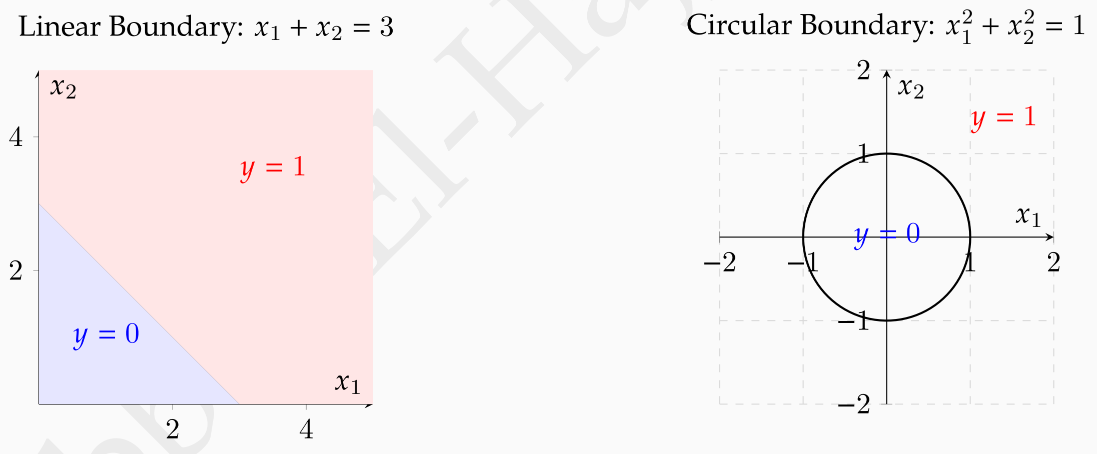


Figure 3: *Examples of decision boundaries. Left: Linear boundary from linear features. Right: Circular boundary from polynomial features.*

3 The Cost Function and Optimization

To find the best parameters $\boldsymbol{\theta}$, we need a cost function to minimize. For classification, we use the **Cross-Entropy** cost function. We'll first build intuition for how it works, then rigorously derive it from statistical principles in section 5.

3.1 Intuition for the Cross-Entropy Cost Function

For a single example $(\boldsymbol{x}, y)$, the loss is:

$$\ell(\hat{y}, y) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$

where $\hat{y} = h_{\boldsymbol{\theta}}(\boldsymbol{x})$ is the predicted probability.

How this loss behaves:

- If $y = 1$ and we predict $\hat{y} \to 1$ (correct), the loss $-\log(1) \to 0$.
- If $y = 1$ and we predict $\hat{y} \to 0$ (incorrect), the loss $-\log(0) \to \infty$.

The function heavily penalizes confident but incorrect predictions, which is exactly what we want.

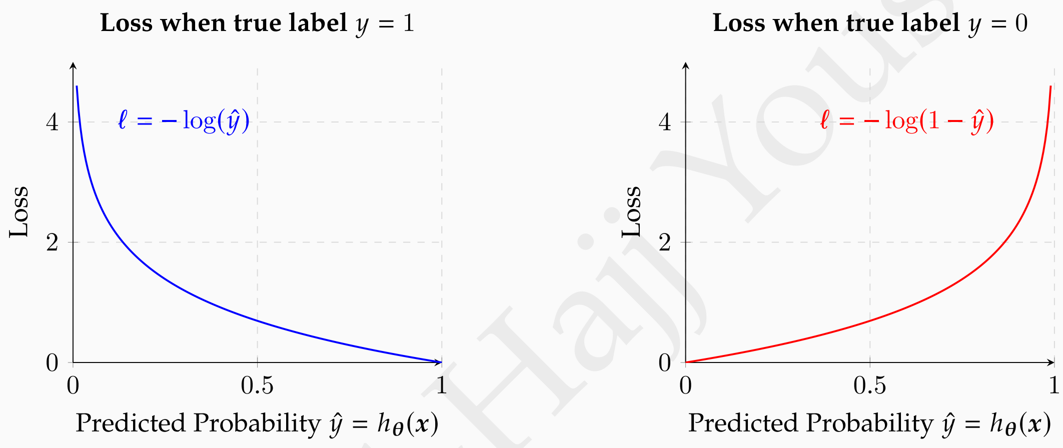


Figure 4: *The Log Loss function penalizes confident wrong predictions exponentially.*

We can write the loss for a single example in a single, compact equation:

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

The total cost over all m training examples is the average of these individual losses.

Cross-Entropy Cost Function

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) + (1-y^{(i)})\log(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))\right] \tag{7}$$

Our goal is to find the parameters $\boldsymbol{\theta}$ that minimize this cost function.

A Note on Terminology

The cost function used in logistic regression goes by several names, which are all related:

- **Cross-Entropy Loss:** This is the most general term, originating from information theory, measuring the difference between two probability distributions (the true labels and the model's predictions).

- **Log Loss:** This is a common name for cross-entropy loss specifically in the context of binary classification.
- **Negative Log-Likelihood:** As we will see in section 5, this cost function is derived from the principle of Maximum Likelihood Estimation (MLE). Minimizing the cost is equivalent to maximizing the likelihood of the observed data.

In this course, we will primarily use the term "Cross-Entropy."

3.2 Optimization with Gradient Descent

We use Gradient Descent to find the $\boldsymbol{\theta}$ that minimizes $J(\boldsymbol{\theta})$. This involves repeatedly updating each parameter by taking a step in the direction opposite to the gradient. The derivation of these formulas will be presented in section 6.

Gradient Descent Update Rule

Repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}) x_j^{(i)} \tag{8}$$

for all $j = 0, 1, \ldots, n$ simultaneously.
Vectorized form:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{m} \mathbf{X}^T (\boldsymbol{h} - \boldsymbol{y}) \tag{9}$$

A Major Advantage: Convexity

A crucial property of the cross-entropy cost function for logistic regression is that it is **convex**. A convex function has only one minimum—a global minimum. This means that gradient descent, if run long enough, is **guaranteed** to find the optimal parameter values.
This is a major theoretical advantage over deep neural networks and other non-convex models, whose cost functions may have many local minima where an optimization algorithm can get stuck. The principled choice of the cross-entropy loss function is what gives us this guarantee.

4 Making Predictions with Logistic Regression

Before diving into how to train a logistic regression model, let's first understand how to use a trained model to make predictions. This will help us appreciate what we're trying to achieve when we train the model.

4.1 Prediction with a Trained Model

Problem: Tumor Classification

Task: Predict if a tumor is malignant ($y = 1$) or benign ($y = 0$).

Model: Logistic regression with trained parameters $\boldsymbol{\theta} = [-5, 1.5, 0.8]^T$.

Features: $\boldsymbol{x} = [x_0, x_1, x_2]^T$ where $x_0 = 1$ (bias), x_1 is tumor size, x_2 is patient age.

Input: New patient — tumor size $x_1 = 4$, age $x_2 = 0.6$.

Solution Steps:

Step 1: Compute Linear Score

$$z = \boldsymbol{\theta}^T \boldsymbol{x} = (-5)(1) + (1.5)(4) + (0.8)(0.6) = -5 + 6 + 0.48 = 1.48$$

Step 2: Apply Sigmoid Function

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(z) = \frac{1}{1 + e^{-1.48}} \approx 0.814$$

The model estimates an **81.4% probability** that the tumor is malignant.

Step 3: Make Discrete Prediction

Since $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = 0.814 \geq 0.5$, we predict **Malignant ($y = 1$)**.

Step 4: Evaluate Prediction Quality (if true label known)

- If true label $y = 1$ (correct): Loss $= -\log(0.814) \approx 0.206$ — low penalty
- If true label $y = 0$ (incorrect): Loss $= -\log(0.186) \approx 1.68$ — high penalty

From Predictions to Training

We've seen how to make predictions with a trained model. But how do we find the optimal parameters $\boldsymbol{\theta}$ in the first place?
The next two sections build the mathematical foundation:

- **Section 5:** Derives why cross-entropy is the "correct" cost function (via MLE)
- **Section 6:** Derives how to compute gradients for optimization
- **Section 7:** Synthesizes everything into a complete training example

5 Deriving the Cross-Entropy Cost Function

The cross-entropy cost function is fundamental to logistic regression, yet its mathematical form may initially appear arbitrary. In this section, we demonstrate that this cost function emerges naturally from first principles through the framework of **Maximum Likelihood Estimation (MLE)**. This derivation not only justifies our choice mathematically but also reveals deep connections between logistic regression and statistical theory.

5.1 The Statistical Foundation: Modeling Binary Outcomes

To understand why cross-entropy is the natural cost function for logistic regression, we begin by establishing a probabilistic framework for binary classification.

The Bernoulli Distribution for Binary Classification

In binary classification, each label $y \in \{0, 1\}$ can be modeled as a realization of a Bernoulli random variable. The probability mass function is:

$$P(y; p) = p^y (1-p)^{1-y} \tag{10}$$

This compact formulation elegantly captures both outcomes:

- When $y = 1$: $P(y = 1; p) = p$
- When $y = 0$: $P(y = 0; p) = 1 - p$

In the context of logistic regression, the hypothesis function $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ represents our model's estimate of $P(y = 1 \mid \boldsymbol{x})$—the probability that the positive class occurs given input features $\boldsymbol{x}$. Therefore, the probability of observing a specific label y given input $\boldsymbol{x}$ and parameters $\boldsymbol{\theta}$ becomes:

$$P(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = h_{\boldsymbol{\theta}}(\boldsymbol{x})^y \left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x})\right)^{1-y} \tag{11}$$

This formulation quantifies prediction quality: high probability indicates good alignment between prediction and true label, while low probability signals poor prediction.

5.2 Maximum Likelihood Estimation: A Principled Approach

Having established our probabilistic model, we now address the central question: *How should we select the optimal parameters $\boldsymbol{\theta}$?* Maximum Likelihood Estimation provides a statistically principled answer.

The Maximum Likelihood Principle

MLE selects parameters that maximize the probability of observing the training data. Formally, given a dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^m$, we seek:

$$\boldsymbol{\theta}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} P(\mathcal{D} \mid \boldsymbol{\theta}) \tag{12}$$

This approach answers the question: *"Which parameter values make our observed data most probable?"*

5.3 Constructing the Likelihood Function

To implement MLE, we must first express the probability of observing our entire dataset. Under the standard assumption that training examples are independently drawn from the same distribution, this probability factorizes into a product:

The Likelihood Function

For m independent training examples, the likelihood function is:

$$\mathcal{L}(\boldsymbol{\theta}) = P(\mathcal{D} \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} P(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{13}$$

$$= \prod_{i=1}^{m} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})^{y^{(i)}} \left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)^{1-y^{(i)}} \tag{14}$$

Maximizing this likelihood function yields the parameters that best explain our observed data according to our model.

5.4 The Log-Likelihood: Mathematical Convenience and Numerical Stability

While conceptually straightforward, directly optimizing $\mathcal{L}(\boldsymbol{\theta})$ presents practical challenges:

1. **Numerical underflow**: Products of many small probabilities can exceed machine precision limits
2. **Mathematical complexity**: Products are more difficult to differentiate than sums

The logarithm transformation elegantly addresses both issues. Since log is strictly monotonically increasing, maximizing $\log \mathcal{L}(\boldsymbol{\theta})$ yields identical optimal parameters while converting products to sums:

The Log-Likelihood Function

Applying the logarithm to the likelihood function:

$$\log \mathcal{L}(\boldsymbol{\theta}) = \log \prod_{i=1}^{m} P(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{15}$$

$$= \sum_{i=1}^{m} \log P(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{16}$$

$$= \sum_{i=1}^{m} \left[y^{(i)} \log h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) + (1 - y^{(i)}) \log \left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) \right] \tag{17}$$

This transformation preserves the optimization objective while providing computational advantages.

5.5 Deriving the Cross-Entropy Cost Function

The final step connects maximum likelihood to the cross-entropy cost function through a simple but important transformation. Machine learning optimization algorithms conventionally minimize cost functions rather than maximize objective functions.

$$\boxed{\textbf{Maximizing } \log \mathcal{L}(\boldsymbol{\theta}) \textbf{ is equivalent to minimizing } -\log \mathcal{L}(\boldsymbol{\theta})}$$

We achieve this by:

1. **Negating** the log-likelihood (converting maximization to minimization)
2. **Normalizing** by the number of examples (ensuring scale invariance)

$$\boxed{J(\boldsymbol{\theta}) = -\frac{1}{m}\log \mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) + (1-y^{(i)})\log\left(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)\right]} \tag{18}$$

This is precisely the cross-entropy cost function introduced earlier. The normalization factor $1/m$ serves dual purposes:

- Makes the cost magnitude independent of dataset size
- Ensures gradient magnitudes remain consistent across different batch sizes

5.6 Synthesis: The Theoretical Foundation of Cross-Entropy

Key Insight: Cross-Entropy as Maximum Likelihood

The cross-entropy cost function is not arbitrary—it is the direct consequence of:

1. **Statistical Modeling**: Treating binary classification as Bernoulli trials
2. **Maximum Likelihood**: Selecting parameters that maximize data probability
3. **Mathematical Transformation**: Converting to log-space for computational efficiency
4. **Optimization Convention**: Minimizing cost rather than maximizing likelihood

When we minimize cross-entropy via gradient descent, we are performing maximum likelihood estimation—finding the parameters most likely to have generated our observed data.

5.7 Connection to the Broader Statistical Framework

Unified Framework: Generalized Linear Models (GLMs)

The similarity between logistic and linear regression is not coincidental—both belong to the family of **Generalized Linear Models (GLMs)**. This framework reveals their shared structure:

Model	**Linear Core**	**Link Function**
Linear Regression	$\boldsymbol{\theta}^T\boldsymbol{x}$	Identity: $f(z) = z$
Logistic Regression	$\boldsymbol{\theta}^T\boldsymbol{x}$	Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$

Both models:

- Compute a linear combination of features: $\boldsymbol{\theta}^T\boldsymbol{x}$
- Apply a link function appropriate to their output domain
- Derive their cost functions via maximum likelihood estimation

This unified perspective explains why their mathematical foundations and optimization procedures share remarkable similarities despite addressing different problem types.

Checkpoint: What We Just Learned

- Cross-entropy is not arbitrary–it's the negative log-likelihood.
- Minimizing cross-entropy = maximizing likelihood of observing our data
- This is why logistic regression is a *principled* statistical model
- The convexity guarantee comes from this MLE foundation

6 The Mathematics of Optimization

Now we will derive the gradient descent update rule that we introduced in Section 3. This requires applying the chain rule to our cost function.

6.1 Deriving the Gradient of the Cost Function

Our goal is to compute the partial derivative of the cost function $J(\boldsymbol{\theta})$ with respect to a single parameter θ_j.

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

We can focus on a single training example first and then sum the results. The loss for one example $(\boldsymbol{x}, y)$ is $\ell = -[y\log h + (1-y)\log(1-h)]$, where $h = \sigma(\boldsymbol{\theta}^T\boldsymbol{x})$.

Step 1: Apply the Chain Rule

We need to compute $\frac{\partial \ell}{\partial \theta_j}$. Using the chain rule, we can write:

$$\frac{\partial \ell}{\partial \theta_j} = \frac{\partial \ell}{\partial h}\cdot\frac{\partial h}{\partial z}\cdot\frac{\partial z}{\partial \theta_j}$$

where $z = \boldsymbol{\theta}^T\boldsymbol{x}$. Let's compute each part.

Step 2: Compute the components

1. **Derivative of Loss w.r.t. Hypothesis (ℓ w.r.t. h):**

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

2. **Derivative of Hypothesis w.r.t. Linear Score (h w.r.t. z):** This is the derivative of the sigmoid function, $\sigma(z)$:

$$\frac{\partial h}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z)) = h(1-h)$$

3. **Derivative of Linear Score w.r.t. Parameter (z w.r.t. θ_j):**

$$\frac{\partial z}{\partial \theta_j} = \frac{\partial}{\partial \theta_j}(\theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n) = x_j$$

Step 3: Combine the results

Now, we multiply the three parts together:

$$\frac{\partial \ell}{\partial \theta_j} = \left(\frac{h-y}{h(1-h)}\right)\cdot(h(1-h))\cdot(x_j) = (h-y)x_j$$

This is a remarkably simple and elegant result. The gradient for a single example is just the prediction error times the feature value.

Step 4: Average over all examples

To get the full gradient of the cost function $J(\boldsymbol{\theta})$, we average this result over all m training examples:

$$\boxed{\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}} \tag{19}$$

This is precisely the gradient term used in the gradient descent update rule from section 3.

Intuition Behind the Gradient: $(h - y)x_j$

This beautifully simple result shows us exactly how the algorithm learns:

- $(h - y)$ **is the prediction error.**
  - If we predict $h = 0.9$ but $y = 0$, we were too confident ($h - y = 0.9$).
  - If we predict $h = 0.2$ but $y = 1$, we weren't confident enough ($h - y = -0.8$).
- x_j **scales the update by how much this feature was present.**
  - Large $x_j \to$ large update (this feature contributed more to the error).
  - $x_j = 0 \to$ no update (this feature wasn't involved).

Gradient descent therefore updates parameters in proportion to two things: how wrong the prediction was, and how strongly each feature contributed to that mistake. It's exactly what a learning algorithm should do—learn more from bigger errors and focus on the parameters that matter most.

The Elegant Gradient Formula

The gradient has a remarkably simple and beautiful structure:

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_i (h^{(i)} - y^{(i)})\, x_j^{(i)}$$

$$\textbf{Gradient} = \textbf{Average of } \big[(\text{Prediction Error}) \times (\text{Feature Value})\big]$$

This same form appears in:

- Linear regression: $(h - y) \cdot x$
- Logistic regression: $(h - y) \cdot x$
- Softmax regression: $(\hat{y}_k - y_k) \cdot x$ for each class k

This is not a coincidence—it's a consequence of using MLE with exponential-family distributions.

7 The Complete Training Process

Now that we understand *what* to optimize (Section 5: cross-entropy via MLE) and *how* to compute gradients (Section 6), let's walk through a complete training cycle from scratch. This section synthesizes everything we've learned into a practical example.

7.1 Complete Training Iteration

7.1.1 Problem Setup and Initialization

Dataset: Student Exam Performance

We have $m = 4$ students with $n = 2$ features: hours studied (x_1) and hours slept (x_2). The target y is binary: 1 (passed) or 0 (failed).

Sample i	x_1 (Study)	x_2 (Sleep)	y (Pass)
1	2	5	0
2	3	7	1
3	5	6	1
4	1	4	0

Initial Parameters:

- Weights: $\theta_1 = 0.1$, $\theta_2 = 0.2$
- Bias: $\theta_0 = -1.0$
- Learning rate: $\alpha = 0.1$

Our initial hypothesis: $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(0.1x_1 + 0.2x_2 - 1.0)$

7.1.2 Step 1: Forward Propagation

For each training sample, we compute the model's prediction $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$.

Process:

1. Calculate logit: $z^{(i)} = \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \theta_0$
2. Apply sigmoid: $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) = \sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$

Calculations for Each Sample:

Sample 1 ($x_1 = 2, x_2 = 5, y = 0$):

$$z^{(1)} = 0.1(2) + 0.2(5) - 1.0 = 0.2 + 1.0 - 1.0 = 0.2$$

$$h^{(1)} = \sigma(0.2) = \frac{1}{1 + e^{-0.2}} \approx 0.5498$$

Sample 2 ($x_1 = 3, x_2 = 7, y = 1$):

$$z^{(2)} = 0.1(3) + 0.2(7) - 1.0 = 0.3 + 1.4 - 1.0 = 0.7$$

$$h^{(2)} = \sigma(0.7) \approx 0.6682$$

Sample 3 ($x_1 = 5, x_2 = 6, y = 1$):

$$z^{(3)} = 0.1(5) + 0.2(6) - 1.0 = 0.5 + 1.2 - 1.0 = 0.7$$

$$h^{(3)} = \sigma(0.7) \approx 0.6682$$

Sample 4 ($x_1 = 1, x_2 = 4, y = 0$):

$$\begin{aligned} z^{(4)} &= 0.1(1) + 0.2(4) - 1.0 = 0.1 + 0.8 - 1.0 = -0.1 \\ h^{(4)} &= \sigma(-0.1) \approx 0.4750 \end{aligned}$$

7.1.3 Step 2: Compute the Cost (Cross-Entropy Loss)

The cross-entropy cost is:

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

We compute the individual loss for each sample:

Compute Individual Losses and Total Cost

Individual Losses:

$$\begin{aligned} L_1\,(y=0): &\quad -\log(1-0.5498) = -\log(0.4502) = 0.7981 \\ L_2\,(y=1): &\quad -\log(0.6682) = 0.4032 \\ L_3\,(y=1): &\quad -\log(0.6682) = 0.4032 \\ L_4\,(y=0): &\quad -\log(1-0.4750) = -\log(0.5250) = 0.6444 \end{aligned}$$

Total Cost:

$$J(\boldsymbol{\theta}) = \frac{1}{4}(0.7981 + 0.4032 + 0.4032 + 0.6444) = \frac{2.2489}{4} = \boxed{0.5622}$$

7.1.4 Step 3: Backward Propagation (Compute Gradients)

To update our parameters via gradient descent, we need the partial derivatives of J with respect to each parameter. The gradient formula is:

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}$$

The gradient is the average of **(prediction error)** × **(feature value)**.

Backward Propagation

Step 3a: Compute Prediction Errors

Sample i	$y^{(i)}$	$h^{(i)}$	**Error** $(h^{(i)} - y^{(i)})$
1	0	0.5498	0.5498
2	1	0.6682	-0.3318
3	1	0.6682	-0.3318
4	0	0.4750	0.4750

Step 3b: Compute Gradients
For θ_1 (weight for study hours):

$$\begin{aligned}\frac{\partial J}{\partial \theta_1} &= \frac{1}{4}[(0.5498)(2) + (-0.3318)(3) + (-0.3318)(5) + (0.4750)(1)]\\ &= \frac{1}{4}[1.0996 - 0.9954 - 1.659 + 0.4750] = \frac{-1.0798}{4} = \boxed{-0.2700}\end{aligned}$$

For θ_2 (weight for sleep hours):

$$\begin{aligned}\frac{\partial J}{\partial \theta_2} &= \frac{1}{4}[(0.5498)(5) + (-0.3318)(7) + (-0.3318)(6) + (0.4750)(4)]\\ &= \frac{1}{4}[2.749 - 2.3226 - 1.9908 + 1.9] = \frac{0.3356}{4} = \boxed{0.0839}\end{aligned}$$

For θ_0 (bias):

$$\frac{\partial J}{\partial \theta_0} = \frac{1}{4}[0.5498 - 0.3318 - 0.3318 + 0.4750] = \frac{0.3612}{4} = \boxed{0.0903}$$

Interpreting the Gradients

- $\frac{\partial J}{\partial \theta_1} = -0.27 < 0$ The cost *decreases* when we increase θ_1 — study hours should have more weight.
- $\frac{\partial J}{\partial \theta_2} = +0.08 > 0$ The cost *increases* when we increase θ_2 — sleep hours should have less weight.
- $\frac{\partial J}{\partial \theta_0} = +0.09 > 0$ The cost increases with the bias, so we should decrease it slightly.

These gradients indicate exactly how to adjust each parameter to reduce prediction error!

7.1.5 Step 4: Update Parameters

Using the gradient descent update rule: $\theta_{\text{new}} = \theta_{\text{old}} - \alpha \frac{\partial J}{\partial \theta}$

Parameter Updates

$$\theta_1^{\text{new}} = 0.1 - 0.1(-0.2700) = 0.1 + 0.027 = \boxed{0.1270}$$

$$\theta_2^{\text{new}} = 0.2 - 0.1(0.0839) = 0.2 - 0.00839 = \boxed{0.1916}$$
$$\theta_0^{\text{new}} = -1.0 - 0.1(0.0903) = -1.0 - 0.00903 = \boxed{-1.0090}$$

Result: After one iteration, our parameters are $\boldsymbol{\theta} = [-1.0090, 0.1270, 0.1916]^T$.

This process repeats for many iterations (typically hundreds to thousands) until the cost converges to a minimum.

7.1.6 Making Predictions After Training

After many iterations, suppose our parameters have converged to:

$$\theta_0 = -3.5, \quad \theta_1 = 0.5, \quad \theta_2 = 0.3$$

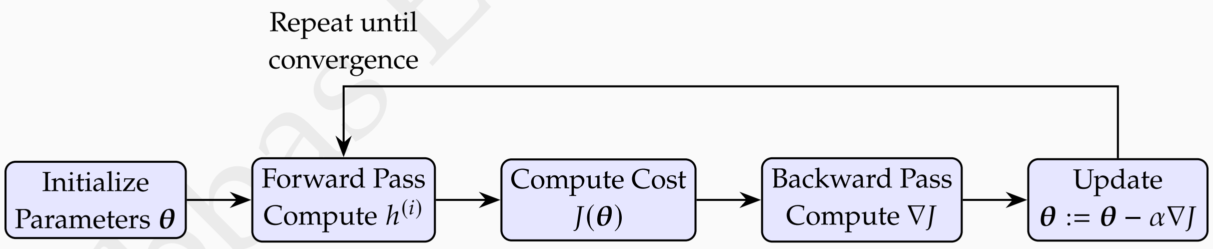
Predict outcome for a new student

Student studied $x_1 = 4$ hours and slept $x_2 = 6$ hours.

$$z = (0.5)(4) + (0.3)(6) - 3.5 = 2.0 + 1.8 - 3.5 = 0.3$$
$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(0.3) = \frac{1}{1 + e^{-0.3}} \approx 0.5744$$

Since $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = 0.5744 > 0.5$, we predict **Pass** $(y = 1)$ with 57.4% confidence.

7.2 Summary: The Training Cycle


Repeat until convergence
Initialize Parameters $\boldsymbol{\theta}$
Forward Pass Compute $h^{(i)}$
Compute Cost $J(\boldsymbol{\theta})$
Backward Pass Compute ∇J
Update $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla J$

8 Handling More Than Two Classes

Many real-world problems require classifying inputs into multiple categories, such as recognizing digits (0–9) or grouping news articles by topic. We'll examine two main strategies for extending binary logistic regression to handle multi-class classification.

8.1 Strategy 1: One-vs-All (One-vs-Rest)

The One-vs-All (OvA) approach, also called One-vs-Rest (OvR), is an intuitive strategy that breaks a multi-class problem down into multiple binary classification problems.

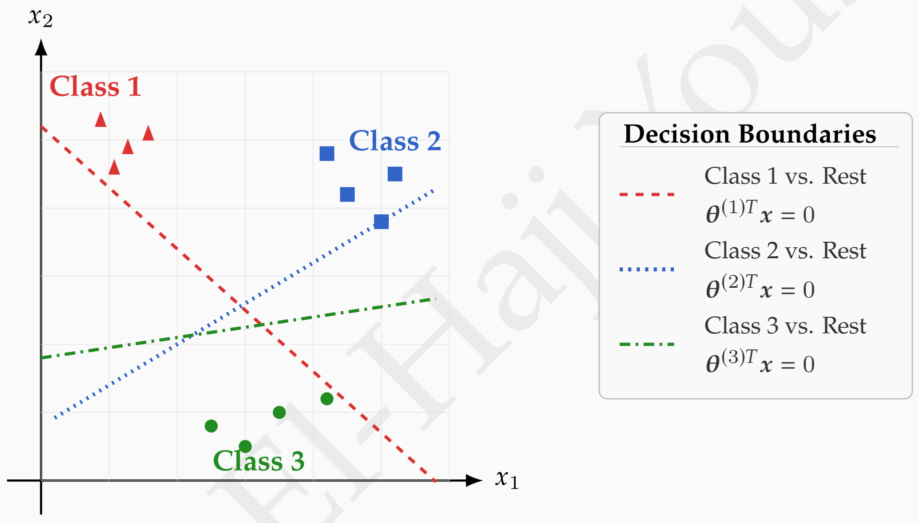
One-vs-All: The Core Idea

For a problem with K classes, we train K separate binary logistic regression classifiers.

- The k-th classifier is trained to distinguish class k (as the positive class) from all other classes combined (as the negative class).
- This gives us K parameter vectors: $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \dots, \boldsymbol{\theta}^{(K)}$.
- Each classifier k outputs the probability $h_{\boldsymbol{\theta}^{(k)}}(\boldsymbol{x}) = P(y = k \mid \boldsymbol{x})$.

To make a prediction for a new input $\boldsymbol{x}$, we compute the probability from all K classifiers and choose the class with the highest score:

$$\hat{y} = \arg \max_{k \in \{1,\dots,K\}} h_{\boldsymbol{\theta}^{(k)}}(\boldsymbol{x})$$



x_2
x_1
Class 1
Class 2
Class 3
Decision Boundaries
Class 1 vs. Rest
$\boldsymbol{\theta}^{(1)T}\boldsymbol{x} = 0$
Class 2 vs. Rest
$\boldsymbol{\theta}^{(2)T}\boldsymbol{x} = 0$
Class 3 vs. Rest
$\boldsymbol{\theta}^{(3)T}\boldsymbol{x} = 0$

Figure 5: *One-vs-All ($K = 3$): each $h_{\boldsymbol{\theta}^{(k)}}$ separates class k; predict $\hat{y} = \arg\max_k h_{\boldsymbol{\theta}^{(k)}}$.*

Advantages of One-vs-All:

- Simple to implement and understand.
- Can use any binary classifier, not just logistic regression.
- Often works well in practice.

Disadvantages of One-vs-All:

- Less efficient as it requires training K models.
- The resulting probabilities from each classifier are not calibrated against each other and will not sum to 1.

8.2 Strategy 2: Softmax Regression

Softmax regression (or Multinomial Logistic Regression) is a more principled generalization of logistic regression to multiple classes. Instead of training K independent models, it trains a single, unified model.

8.2.1 The Softmax Function

The model first computes a linear score $z_k = (\boldsymbol{\theta}^{(k)})^T \boldsymbol{x}$ for each class k. Then, it uses the **softmax function** to convert these scores into a valid probability distribution.

The Softmax Function

The softmax function takes a vector of K real-valued scores $\boldsymbol{z} = [z_1, z_2, \dots, z_K]^T$ and converts it into a probability distribution. The probability for class k is:

$$P(y = k \mid \boldsymbol{x}; \boldsymbol{\Theta}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} = \frac{e^{(\boldsymbol{\theta}^{(k)})^T \boldsymbol{x}}}{\sum_{j=1}^{K} e^{(\boldsymbol{\theta}^{(j)})^T \boldsymbol{x}}} \tag{20}$$

Key properties:

- Each output $P(y = k \mid \boldsymbol{x})$ is in the range $(0, 1)$
- All outputs sum to exactly 1: $\sum_{k=1}^{K} P(y = k \mid \boldsymbol{x}) = 1$
- The full parameter set is $\boldsymbol{\Theta} = \{\boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(K)}\}$

Why "Softmax"?

The softmax function is a smooth, differentiable approximation of the 'argmax' function:

- **Argmax** (hard): Outputs 1 for the highest score, 0 for all others
- **Softmax** (soft): The highest score gets the highest probability, but other scores get non-zero probabilities

This smoothness is essential for gradient-based optimization. The "soft" probabilities allow the model to express uncertainty and enable gradient descent to work effectively.

Numerical Stability: The LogSumExp Trick

Direct computation of softmax can overflow when logits z_k are large (e.g., $e^{1000} = \infty$).
Stable implementation: Subtract the maximum logit before exponentiating:

$$P(y = k|\boldsymbol{x}) = \frac{e^{z_k - z_{\max}}}{\sum_j e^{z_j - z_{\max}}}$$

where $z_{\max} = \max_j z_j$.
This transformation doesn't change the result (cancels out in numerator/denominator) but prevents overflow. Modern ML libraries (NumPy, TensorFlow, PyTorch) implement this automatically.

8.2.2 Representing Multi-Class Labels: One-Hot Encoding

To train softmax regression, we need a way to represent class labels that matches the probability distribution output. This is done through **one-hot encoding**.

One-Hot Encoding

For a classification problem with K classes, each label $y \in \{1, 2, \ldots, K\}$ is represented as a vector $\boldsymbol{y} \in \mathbb{R}^K$ where:

- Position k contains a 1 if the example belongs to class k
- All other positions contain 0

Mathematical form:

$$y_k = \mathbb{1}\{y = k\} = \begin{cases} 1 & \text{if the true class is } k \\ 0 & \text{otherwise} \end{cases}$$

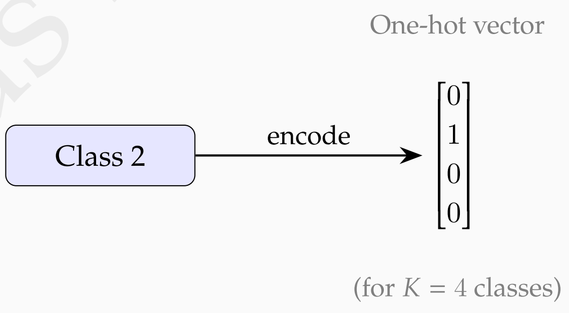
One-Hot Encoding Examples

For a 4-class problem ($K = 4$):

Class Label	One-Hot Vector	Interpretation
$y = 1$	$[1, 0, 0, 0]^T$	Class 1
$y = 2$	$[0, 1, 0, 0]^T$	Class 2
$y = 3$	$[0, 0, 1, 0]^T$	Class 3
$y = 4$	$[0, 0, 0, 1]^T$	Class 4

For IRIS classification ($K = 3$):

- Setosa (class 1): $\boldsymbol{y} = [1, 0, 0]^T$
- Versicolor (class 2): $\boldsymbol{y} = [0, 1, 0]^T$
- Virginica (class 3): $\boldsymbol{y} = [0, 0, 1]^T$



Why One-Hot Encoding?

One-hot encoding serves two critical purposes:

1. **Matches model output format**: The softmax function outputs a probability vector of length K. The one-hot encoded label is also a vector of length K, making them directly comparable.
2. **No ordinal relationship**: Classes often have no inherent ordering (e.g., "cat" vs. "dog" vs. "bird"). Using labels like 1, 2, 3 would incorrectly imply that class 3 is

"larger" than class 1. One-hot encoding treats all classes equally.

8.2.3 The Cost Function: Categorical Cross-Entropy

The cost function for softmax regression is a natural generalization of binary cross-entropy to multiple classes.

Categorical Cross-Entropy Cost Function

For a dataset with m samples and K classes, the cost function is:

$$J(\mathbf{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log P(y = k \mid \boldsymbol{x}^{(i)}; \mathbf{\Theta}) \tag{21}$$

where $y_k^{(i)} = \mathbb{1}\{y^{(i)} = k\}$ is the one-hot encoded label.
Simplified form: Due to one-hot encoding, only one term in the inner sum is non-zero (the true class). This simplifies to:

$$J(\mathbf{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m} \log P(y = c^{(i)} \mid \boldsymbol{x}^{(i)}; \mathbf{\Theta}) \tag{22}$$

where $c^{(i)}$ is the true class label for example i.

Intuition: The loss for a single example is simply the negative log-probability assigned to the correct class. If the model assigns high probability to the correct class, the loss is low. If it assigns low probability, the loss is high.

Understanding the Loss

Consider a 3-class problem where the true class is 2 (one-hot: $[0, 1, 0]^T$).
Scenario 1: Good prediction

- Model output: $[0.1, 0.8, 0.1]^T$
- Loss: $-\log(0.8) \approx 0.22 \quad$ (low penalty)

Scenario 2: Poor prediction

- Model output: $[0.5, 0.2, 0.3]^T$
- Loss: $-\log(0.2) \approx 1.61 \quad$ (high penalty)

Scenario 3: Very poor prediction

- Model output: $[0.6, 0.05, 0.35]^T$
- Loss: $-\log(0.05) \approx 3.00 \quad$ (very high penalty)

Concept

MLE Derivation for Softmax Cost
Just as binary cross-entropy derives from Maximum Likelihood Estimation assuming a Bernoulli distribution, categorical cross-entropy has the same statistical foundation:

- We model multi-class labels as following a **Categorical distribution** (the generalization of Bernoulli to K outcomes)
- The categorical cross-entropy is the **negative log-likelihood** of the training data
- Minimizing this cost is equivalent to maximizing the likelihood of observing our data

This provides a principled, unified framework: cross-entropy is the ”correct” cost function for both binary and multi-class classification, derived from first principles via MLE.

8.2.4 Gradient Descent for Softmax Regression

To optimize the parameters via gradient descent, we need the gradients of the cost function with respect to each class's parameters.

Definition

Gradient of Categorical Cross-Entropy
The gradient of $J(\boldsymbol{\Theta})$ with respect to the parameters of class k is:

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}^{(k)}} = \frac{1}{m}\sum_{i=1}^{m}\left(P(y=k \mid \boldsymbol{x}^{(i)};\boldsymbol{\Theta}) - y_k^{(i)}\right)\boldsymbol{x}^{(i)} \tag{23}$$

This can be written more compactly as:

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}^{(k)}} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}_k^{(i)} - y_k^{(i)}\right)\boldsymbol{x}^{(i)} \tag{24}$$

where $\hat{y}_k^{(i)}$ is the predicted probability for class k on example i.

Elegant Gradient Form

Notice the beautiful similarity to binary logistic regression:

Gradient = (Prediction - True Label) × Features

This same elegant structure appears in:

- **Linear regression**: $(h-y)\cdot x$
- **Binary logistic regression**: $(h-y)\cdot x$
- **Softmax regression**: $(\hat{y}_k - y_k)\cdot x$ for each class k

This is no coincidence—all three models belong to the GLM family and derive their cost functions from maximum likelihood estimation, leading to this unified gradient form.

Gradient Descent Update Rule

For each class k, repeat until convergence:

$$\boldsymbol{\theta}^{(k)} := \boldsymbol{\theta}^{(k)} - \alpha \frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}^{(k)}} \tag{25}$$

Expanded form:

$$\boldsymbol{\theta}^{(k)} := \boldsymbol{\theta}^{(k)} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(\hat{y}_k^{(i)} - y_k^{(i)} \right) \boldsymbol{x}^{(i)} \tag{26}$$

Key point: All K parameter vectors must be updated simultaneously in each iteration.

8.2.5 Complete Softmax Training Algorithm

Softmax Regression Training Algorithm

Input:

- Training set: $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$
- Number of classes: K
- Learning rate: α

Initialize: All parameter vectors $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(K)}$ (typically to small random values or zeros)
Repeat until convergence:

1. **Forward pass**: For each training example i:
   - Compute logits: $z_k^{(i)} = (\boldsymbol{\theta}^{(k)})^T \boldsymbol{x}^{(i)}$ for all k
   - Apply softmax: $\hat{y}_k^{(i)} = \frac{e^{z_k^{(i)}}}{\sum_{j=1}^{K} e^{z_j^{(i)}}}$ for all k
2. **Compute cost**:
$$J = -\frac{1}{m} \sum_{i=1}^{m} \log \hat{y}_{c^{(i)}}^{(i)}$$
   where $c^{(i)}$ is the true class for example i
3. **Backward pass**: For each class k:
$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(k)}} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_k^{(i)} - y_k^{(i)}) \boldsymbol{x}^{(i)}$$
4. **Update parameters**: For each class k:
$$\boldsymbol{\theta}^{(k)} := \boldsymbol{\theta}^{(k)} - \alpha \frac{\partial J}{\partial \boldsymbol{\theta}^{(k)}}$$

Output: Trained parameters $\boldsymbol{\Theta} = \{\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(K)}\}$

Parameter Redundancy in Softmax

An interesting mathematical property: the softmax model has redundant parameters. If you subtract the same constant vector $\boldsymbol{v}$ from all parameter vectors $\boldsymbol{\theta}^{(k)}$, the output probabilities remain unchanged:

$$P(y = k \mid \boldsymbol{x}) = \frac{e^{(\boldsymbol{\theta}^{(k)}-\boldsymbol{v})^T\boldsymbol{x}}}{\sum_{j=1}^{K} e^{(\boldsymbol{\theta}^{(j)}-\boldsymbol{v})^T\boldsymbol{x}}} = \frac{e^{-\boldsymbol{v}^T\boldsymbol{x}} \cdot e^{(\boldsymbol{\theta}^{(k)})^T\boldsymbol{x}}}{e^{-\boldsymbol{v}^T\boldsymbol{x}} \cdot \sum_{j=1}^{K} e^{(\boldsymbol{\theta}^{(j)})^T\boldsymbol{x}}} = \frac{e^{(\boldsymbol{\theta}^{(k)})^T\boldsymbol{x}}}{\sum_{j=1}^{K} e^{(\boldsymbol{\theta}^{(j)})^T\boldsymbol{x}}}$$

Practical implications:

- Some implementations fix one parameter vector (e.g., $\boldsymbol{\theta}^{(K)} = \mathbf{0}$) to ensure a unique solution
- Gradient descent will converge to a valid solution even without this constraint
- This redundancy means the maximum likelihood solution is not unique—there are infinitely many equivalent parameter sets

9 Worked Examples for Multi-Class Classification

This section demonstrates softmax regression through two complementary examples: making predictions with a trained model, and walking through complete numerical calculations of a full training iteration.

9.1 Example 1: Prediction with a Trained Softmax Model

Problem Setup: IRIS Flower Classification

- **Classes:** 1: Setosa, 2: Versicolor, 3: Virginica
- **Features:** x_1: Sepal length, x_2: Sepal width
- **Trained Softmax Parameters:**

$$\boldsymbol{\theta}^{(1)} = \begin{bmatrix} 5.0 \\ -1.0 \\ 2.5 \end{bmatrix}, \quad \boldsymbol{\theta}^{(2)} = \begin{bmatrix} -1.0 \\ 0.5 \\ -0.5 \end{bmatrix}, \quad \boldsymbol{\theta}^{(3)} = \begin{bmatrix} -2.0 \\ 0.8 \\ -0.3 \end{bmatrix}$$

- **Task:** Classify a new flower with sepal length $x_1 = 5.0$ and sepal width $x_2 = 3.5$. The feature vector is $\boldsymbol{x} = [1, 5.0, 3.5]^T$.

Step 1: Compute scores (logits) for each class

$$z_1 = (\boldsymbol{\theta}^{(1)})^T\boldsymbol{x} = 5.0 - 5.0 + 8.75 = 8.75$$
$$z_2 = (\boldsymbol{\theta}^{(2)})^T\boldsymbol{x} = -1.0 + 2.5 - 1.75 = -0.25$$
$$z_3 = (\boldsymbol{\theta}^{(3)})^T\boldsymbol{x} = -2.0 + 4.0 - 1.05 = 0.95$$

Step 2: Exponentiate the scores

$$e^{z_1} \approx 6310.7, \quad e^{z_2} \approx 0.78, \quad e^{z_3} \approx 2.59$$

Step 3: Normalize to get probabilities
The sum is $\sum e^{z_j} \approx 6310.7 + 0.78 + 2.59 = 6314.07$.

$$P(y = 1 \mid \boldsymbol{x}) = \frac{6310.7}{6314.07} \approx 0.9995$$
$$P(y = 2 \mid \boldsymbol{x}) = \frac{0.78}{6314.07} \approx 0.0001$$
$$P(y = 3 \mid \boldsymbol{x}) = \frac{2.59}{6314.07} \approx 0.0004$$

Note that the probabilities sum to 1.

Step 4: Make prediction
The highest probability is for class 1.
Prediction: Setosa. The model is 99.95% confident in its prediction.

9.2 Example 2: Complete Softmax Training Iteration

Now let's walk through a complete training cycle for softmax regression: forward propagation, loss calculation, backward propagation (gradient calculation), and parameter update.

9.2.1 Problem Setup and Initialization

Dataset: Simplified IRIS Classification

We have a simplified Iris dataset with $m = 3$ samples, $K = 3$ classes (0, 1, 2), and $n = 2$ features.

Sample i	x_1	x_2	y **(Class)**
1	1	2	0
2	2	3	1
3	3	1	2

Initial Parameters (including bias):

- Class 0: $\boldsymbol{\theta}^{(0)} = [0, 0.1, 0.2]^T$ (bias, $\theta_1^{(0)}$, $\theta_2^{(0)}$)
- Class 1: $\boldsymbol{\theta}^{(1)} = [0, 0.2, 0.1]^T$ (bias, $\theta_1^{(1)}$, $\theta_2^{(1)}$)
- Class 2: $\boldsymbol{\theta}^{(2)} = [0, 0.1, 0.1]^T$ (bias, $\theta_1^{(2)}$, $\theta_2^{(2)}$)
- Learning rate: $\alpha = 0.1$

Feature vectors (with bias):

$$\boldsymbol{x}^{(1)} = [1, 1, 2]^T, \quad \boldsymbol{x}^{(2)} = [1, 2, 3]^T, \quad \boldsymbol{x}^{(3)} = [1, 3, 1]^T$$

9.2.2 Step 1: Forward Propagation

We compute the predictions for all three samples using the softmax function.

Forward Pass for Sample 1

Sample 1 input: $(\boldsymbol{x}^{(1)} = [1, 1, 2]^T, y^{(1)} = 0)$:

1. Calculate logits z_k for each class:

$$\begin{aligned} z_0^{(1)} &= (\boldsymbol{\theta}^{(0)})^T \boldsymbol{x}^{(1)} = 0(1) + 0.1(1) + 0.2(2) = 0.5 \\ z_1^{(1)} &= (\boldsymbol{\theta}^{(1)})^T \boldsymbol{x}^{(1)} = 0(1) + 0.2(1) + 0.1(2) = 0.4 \\ z_2^{(1)} &= (\boldsymbol{\theta}^{(2)})^T \boldsymbol{x}^{(1)} = 0(1) + 0.1(1) + 0.1(2) = 0.3 \end{aligned}$$

Logit vector: $\boldsymbol{z}^{(1)} = [0.5, 0.4, 0.3]^T$

2. Apply Softmax function:

$$\sum_{j=0}^{2} e^{z_j^{(1)}} = e^{0.5} + e^{0.4} + e^{0.3} = 1.6487 + 1.4918 + 1.3499 = 4.4904$$

$$\begin{aligned} \hat{y}_0^{(1)} &= \frac{e^{0.5}}{4.4904} = \frac{1.6487}{4.4904} = 0.3672 \\ \hat{y}_1^{(1)} &= \frac{e^{0.4}}{4.4904} = \frac{1.4918}{4.4904} = 0.3322 \\ \hat{y}_2^{(1)} &= \frac{e^{0.3}}{4.4904} = \frac{1.3499}{4.4904} = 0.3006 \end{aligned}$$

Prediction vector: $\hat{\boldsymbol{y}}^{(1)} = [0.3672, 0.3322, 0.3006]^T$

Forward Pass for Sample 2

Sample 2 input: $(\boldsymbol{x}^{(2)} = [1, 2, 3]^T, y^{(2)} = 1)$:

1. Logits:

$$\begin{aligned} z_0^{(2)} &= 0(1) + 0.1(2) + 0.2(3) = 0.8 \\ z_1^{(2)} &= 0(1) + 0.2(2) + 0.1(3) = 0.7 \\ z_2^{(2)} &= 0(1) + 0.1(2) + 0.1(3) = 0.5 \end{aligned}$$

2. Probabilities:

$$\sum_{j=0}^{2} e^{z_j^{(2)}} = e^{0.8} + e^{0.7} + e^{0.5} = 2.2255 + 2.0138 + 1.6487 = 5.8880$$

$$\hat{\boldsymbol{y}}^{(2)} = \left[\frac{2.2255}{5.8880}, \frac{2.0138}{5.8880}, \frac{1.6487}{5.8880}\right]^T = [0.3780, 0.3420, 0.2799]^T$$

Forward Pass for Sample 3

Sample 3 input: $(\boldsymbol{x}^{(3)} = [1, 3, 1]^T, y^{(3)} = 2)$.

1. Logits:

$$\begin{aligned} z_0^{(3)} &= 0(1) + 0.1(3) + 0.2(1) = 0.5 \\ z_1^{(3)} &= 0(1) + 0.2(3) + 0.1(1) = 0.7 \\ z_2^{(3)} &= 0(1) + 0.1(3) + 0.1(1) = 0.4 \end{aligned}$$

2. Probabilities:

$$\sum_{j=0}^{2} e^{z_j^{(3)}} = e^{0.5} + e^{0.7} + e^{0.4} = 1.6487 + 2.0138 + 1.4918 = 5.1543$$

$$\hat{\boldsymbol{y}}^{(3)} = \left[\frac{1.6487}{5.1543}, \frac{2.0138}{5.1543}, \frac{1.4918}{5.1543}\right]^T = [0.3199, 0.3907, 0.2894]^T$$

9.2.3 Step 2: Calculating the Categorical Cross-Entropy Loss

The true labels in one-hot format are:

$$\boldsymbol{y}^{(1)} = [1, 0, 0]^T, \quad \boldsymbol{y}^{(2)} = [0, 1, 0]^T, \quad \boldsymbol{y}^{(3)} = [0, 0, 1]^T$$

The cross-entropy loss for each sample is the negative log of the predicted probability for the *correct* class:

Loss Calculation

Individual Losses:

$$\begin{aligned} L_1 &= -\log(\hat{y}_0^{(1)}) = -\log(0.3672) \approx 1.0018 \quad \text{(true class: 0)} \\ L_2 &= -\log(\hat{y}_1^{(2)}) = -\log(0.3420) \approx 1.0729 \quad \text{(true class: 1)} \\ L_3 &= -\log(\hat{y}_2^{(3)}) = -\log(0.2894) \approx 1.2400 \quad \text{(true class: 2)} \end{aligned}$$

Total Cost:

$$J(\boldsymbol{\Theta}) = \frac{1}{3}(1.0018 + 1.0729 + 1.2400) = \frac{3.3147}{3} = \boxed{1.1049}$$

9.2.4 Step 3: Backward Propagation (Compute Gradients)

Recall from Section 8.3 that the gradient for class k is:

$$\frac{\partial J(\boldsymbol{\Theta})}{\partial \boldsymbol{\theta}^{(k)}} = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}_k^{(i)} - y_k^{(i)}\right) \boldsymbol{x}^{(i)}$$

This elegant formula tells us that the gradient for class k is the average of **(prediction error for class k) $\times$ (feature vector)**.

We'll compute this for each class by:

1. Computing the prediction error $(\hat{y}_k^{(i)} - y_k^{(i)})$ for each sample
2. Multiplying by the feature vector $\boldsymbol{x}^{(i)}$
3. Averaging across all samples

Computing Gradients for Class 0: $\frac{\partial J}{\partial \boldsymbol{\theta}^{(0)}}$

Step 1: Compute prediction errors for class 0:

$$\begin{aligned}
\text{Sample 1:}\quad & \hat{y}_0^{(1)} - y_0^{(1)} = 0.3672 - 1 = -0.6328 \\
\text{Sample 2:}\quad & \hat{y}_0^{(2)} - y_0^{(2)} = 0.3780 - 0 = 0.3780 \\
\text{Sample 3:}\quad & \hat{y}_0^{(3)} - y_0^{(3)} = 0.3199 - 0 = 0.3199
\end{aligned}$$

Step 2: Multiply each error by its feature vector:

$$\begin{aligned}
(-0.6328) \cdot \boldsymbol{x}^{(1)} &= (-0.6328) \cdot [1, 1, 2]^T = [-0.6328, -0.6328, -1.2656]^T \\
(0.3780) \cdot \boldsymbol{x}^{(2)} &= (0.3780) \cdot [1, 2, 3]^T = [0.3780, 0.7560, 1.1340]^T \\
(0.3199) \cdot \boldsymbol{x}^{(3)} &= (0.3199) \cdot [1, 3, 1]^T = [0.3199, 0.9597, 0.3199]^T
\end{aligned}$$

Step 3: Average across all samples:

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{\theta}^{(0)}} &= \frac{1}{3}\left([-0.6328, -0.6328, -1.2656]^T + [0.3780, 0.7560, 1.1340]^T + [0.3199, 0.9597, 0.3199]^T\right) \\
&= \frac{1}{3}[0.0651, 1.0829, 0.1883]^T \\
&= \boxed{[0.0217, 0.3610, 0.0628]^T}
\end{aligned}$$

This gives us: $\frac{\partial J}{\partial \theta_0^{(0)}} = 0.0217$ (bias), $\frac{\partial J}{\partial \theta_1^{(0)}} = 0.3610$, $\frac{\partial J}{\partial \theta_2^{(0)}} = 0.0628$.

Computing Gradients for Class 1: $\frac{\partial J}{\partial \boldsymbol{\theta}^{(1)}}$

Prediction errors for class 1:

$$\begin{aligned}
\text{Sample 1:}\quad & \hat{y}_1^{(1)} - y_1^{(1)} = 0.3322 - 0 = 0.3322 \\
\text{Sample 2:}\quad & \hat{y}_1^{(2)} - y_1^{(2)} = 0.3420 - 1 = -0.6580 \\
\text{Sample 3:}\quad & \hat{y}_1^{(3)} - y_1^{(3)} = 0.3907 - 0 = 0.3907
\end{aligned}$$

Multiply by feature vectors and average:

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{\theta}^{(1)}} &= \frac{1}{3}\left((0.3322)[1, 1, 2]^T + (-0.6580)[1, 2, 3]^T + (0.3907)[1, 3, 1]^T\right) \\
&= \frac{1}{3}\left([0.3322, 0.3322, 0.6644]^T + [-0.6580, -1.3160, -1.9740]^T + [0.3907, 1.1721, 0.3907]^T\right) \\
&= \frac{1}{3}[0.0649, 0.1883, -0.9189]^T \\
&= \boxed{[0.0216, 0.0628, -0.3063]^T}
\end{aligned}$$

Computing Gradients for Class 2: $\frac{\partial J}{\partial \boldsymbol{\theta}^{(2)}}$

Prediction errors for class 2:

$$\begin{aligned}
\text{Sample 1:} \quad & \hat{y}_2^{(1)} - y_2^{(1)} = 0.3006 - 0 = 0.3006 \\
\text{Sample 2:} \quad & \hat{y}_2^{(2)} - y_2^{(2)} = 0.2799 - 0 = 0.2799 \\
\text{Sample 3:} \quad & \hat{y}_2^{(3)} - y_2^{(3)} = 0.2894 - 1 = -0.7106
\end{aligned}$$

Multiply by feature vectors and average:

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{\theta}^{(2)}} &= \frac{1}{3}\left((0.3006)[1,1,2]^T + (0.2799)[1,2,3]^T + (-0.7106)[1,3,1]^T\right) \\
&= \frac{1}{3}\left([0.3006, 0.3006, 0.6012]^T + [0.2799, 0.5598, 0.8397]^T + [-0.7106, -2.1318, -0.7106]^T\right) \\
&= \frac{1}{3}[-0.1301, -1.2714, 0.7303]^T \\
&= \boxed{[-0.0434, -0.4238, 0.2434]^T}
\end{aligned}$$

Understanding the Gradient Structure

Notice how the elegant formula $\frac{\partial J}{\partial \boldsymbol{\theta}^{(k)}} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_k^{(i)} - y_k^{(i)})\boldsymbol{x}^{(i)}$ captures everything:

- For **Class 0**: Sample 1 has a large *negative* error (-0.6328) because we under-predicted the correct class. This pulls the gradient in the direction to *increase* the probability for class 0.
- For **Class 1**: Sample 2 has a large *negative* error (-0.6580) for the same reason—we need to increase probability for class 1 on this sample.
- For **Class 2**: Sample 3 has a large *negative* error (-0.7106) because we under-predicted the correct class 2.
- Samples where the class is *incorrect* have *positive* errors, pushing the gradient to *decrease* those probabilities.

This is the same beautiful structure we saw in binary logistic regression: **gradient = average of (error × features)**, now generalized to K classes simultaneously!

9.2.5 Step 4: Parameter Update

Finally, we update all parameters using gradient descent with learning rate $\alpha = 0.1$. The update rule is $\boldsymbol{\theta}^{(k)}_{\text{new}} = \boldsymbol{\theta}^{(k)}_{\text{old}} - \alpha\frac{\partial J}{\partial \boldsymbol{\theta}^{(k)}}$.

Parameter Updates for All Classes

Class 0 Update:

$$\boldsymbol{\theta}^{(0)}_{\text{new}} = [0, 0.1, 0.2]^T - 0.1 \cdot [0.0217, 0.3610, 0.0628]^T$$

$$
\begin{aligned}
&= [0, 0.1, 0.2]^T - [0.0022, 0.0361, 0.0063]^T \\
&= \boxed{[-0.0022, 0.0639, 0.1937]^T}
\end{aligned}
$$

Class 1 Update:

$$
\begin{aligned}
\boldsymbol{\theta}^{(1)}_{\text{new}} &= [0, 0.2, 0.1]^T - 0.1 \cdot [0.0216, 0.0628, -0.3063]^T \\
&= [0, 0.2, 0.1]^T - [0.0022, 0.0063, -0.0306]^T \\
&= \boxed{[-0.0022, 0.1937, 0.1306]^T}
\end{aligned}
$$

Class 2 Update:

$$
\begin{aligned}
\boldsymbol{\theta}^{(2)}_{\text{new}} &= [0, 0.1, 0.1]^T - 0.1 \cdot [-0.0434, -0.4238, 0.2434]^T \\
&= [0, 0.1, 0.1]^T - [-0.0043, -0.0424, 0.0243]^T \\
&= \boxed{[0.0043, 0.1424, 0.0757]^T}
\end{aligned}
$$

Summary of new parameters:

Class	Bias	θ_1	θ_2	Vector Form
0	-0.0022	0.0639	0.1937	$[-0.0022, 0.0639, 0.1937]^T$
1	-0.0022	0.1937	0.1306	$[-0.0022, 0.1937, 0.1306]^T$
2	0.0043	0.1424	0.0757	$[0.0043, 0.1424, 0.0757]^T$

Result: These updated parameters would be used for the next iteration of training, repeating Steps 1-4 until convergence.

9.3 Key Insights from Multi-Class Examples

What We Learned

1. **Softmax naturally extends sigmoid**: The softmax function creates a valid probability distribution over K classes, generalizing the sigmoid's binary output.
2. **One-hot encoding is essential**: It allows us to compare predicted probability distributions with true labels in a meaningful way.
3. **Categorical cross-entropy generalizes binary cross-entropy**: The loss structure remains elegant—we penalize low probability assigned to the correct class.
4. **The gradient structure is preserved**: $(\hat{y}_k - y_k) \cdot x$ for each class k maintains the beautiful "error times feature" form we saw in binary classification.
5. **All classes train simultaneously**: Unlike One-vs-All, softmax updates all K parameter vectors together, ensuring they learn relative to each other.
6. **Probabilities sum to 1**: Unlike OvA, softmax outputs represent a true probability distribution, making them directly interpretable.

10 Summary: Logistic vs. Softmax Regression

Table 1: *Comparison of Multi-Class Classification Strategies*

Aspect	**Logistic Regression (One-vs-All)**	**Softmax Regression**
Number of Models	K independent binary models	1 unified multi-class model
Training Method	K separate training processes	Joint training of all parameters
Probabilities Sum to 1?	✗ No	✓ Yes
Calibrated Probabilities?	✗ No	✓ Yes
Foundation	Heuristic extension of binary case	Principled multi-class MLE
Best For	▶ Simple problems ▶ Reusing existing binary classifiers	▶ When true probabilities are needed ▶ Foundation for neural networks

Key Takeaways

1. **Logistic regression** extends linear models to classification using the sigmoid function, which emerges naturally from modeling log-odds.
2. The **cross-entropy cost** derives from Maximum Likelihood Estimation on Bernoulli (binary) or Categorical (multi-class) distributions.
3. The cost function is **convex**, guaranteeing global minimum convergence.
4. The gradient update $(h - y) \cdot x$ has elegant interpretation and motivates **feature scaling**.
5. **Softmax regression** provides principled multi-class extension with true probability distributions.

A Complete Notation Reference

Dataset and Features

m Number of training examples in the dataset.

n Number of features (excluding bias term).

$\boldsymbol{x}^{(i)}$ Feature vector of the i-th example, $\boldsymbol{x} \in \mathbb{R}^{n+1}$ (includes bias $x_0 = 1$).

$x_j^{(i)}$ Value of the j-th feature for the i-th example.

$y^{(i)}$ Label for the i-th example: $y \in \{0, 1\}$ (binary) or $y \in \{1, \ldots, K\}$ (multi-class).

Model Parameters

$\boldsymbol{\theta}$ Weight vector for a single classifier, including bias θ_0.

θ_j j-th weight in $\boldsymbol{\theta}$.

$\boldsymbol{\Theta}$ Collection of all weight vectors in multi-class softmax: $\{\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(K)}\}$.

$\boldsymbol{\theta}^{(k)}$ Weight vector for class k.

Functions & Hypotheses

$h_{\boldsymbol{\theta}}(x)$ Hypothesis function: model prediction for input $\boldsymbol{x}$ ($P(y = 1|\boldsymbol{x})$ for binary classification).

$\sigma(z)$ Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$.

z Linear score (logit): $z = \boldsymbol{\theta}^T \boldsymbol{x}$.

z_k Linear score for class k: $z_k = (\boldsymbol{\theta}^{(k)})^T \boldsymbol{x}$.

Cost & Loss

$J(\boldsymbol{\theta})$ Binary logistic regression cost function.

$J(\boldsymbol{\Theta})$ Multi-class softmax cost function.

$\ell(\hat{y}, y)$ Loss for a single training example.

$\mathcal{L}(\boldsymbol{\theta})$ Likelihood of observing data given $\boldsymbol{\theta}$.

$\log \mathcal{L}(\boldsymbol{\theta})$ Log-likelihood function.

Optimization

α Learning rate for gradient descent.

λ Regularization parameter controlling overfitting (weight decay strength).

Multi-class Classification

K Total number of classes.

c Index of the true class for a given example.

$P(y = k \mid x)$ Probability that input $\boldsymbol{x}$ belongs to class k.

$\mathbb{1}\{\cdot\}$ Indicator function: equals 1 if condition is true, 0 otherwise.

Probability & Statistics

p Probability of the positive class: $p = P(y = 1 \mid \boldsymbol{x})$.

logit(p) Log-odds: $\log \frac{p}{1-p}$.

e Euler's number (base of natural logarithm), approximately 2.71828.

Matrices & Vectors

$\mathbf{X}$ Design matrix containing all training examples (each row is one example).

$\boldsymbol{h}$ Vector of hypothesis outputs for all training examples.

$\boldsymbol{y}$ Vector of all true labels in the training set.

$\boldsymbol{v}$ Constant vector (used in discussion of parameter redundancy).

Miscellaneous

$\hat{y}$ Predicted class label or probability for an input.

OvA / OvR One-vs-All (One-vs-Rest) classification strategy.

MLE Maximum Likelihood Estimation.

GLM Generalized Linear Model.