Regression Analysis

Linear and Logistic Regression

Converted from Markdown

AI Course Material

May 10, 2025

Outline

0-1 What is Regression Analysis?

Core Concept

Regression analysis is a cornerstone of supervised machine learning and statistics. Its primary goal is to model the relationship between:

- A **dependent variable** (target)
- One or more **independent variables** (features or predictors)

Applications

- Predicting continuous values (e.g., house prices, stock prices).
- Understanding the influence of different factors on an outcome.
- Forming the basis for more complex models.

1-0 Linear Regression: Overview

Modelling Linear Relationships

Definition

Linear Regression is one of the simplest and most widely used regression algorithms. It assumes a linear relationship between the input features and the continuous target variable.

Key Strengths

- Simplicity and interpretability.
- Computationally efficient.
- Powerful tool for prediction and understanding feature importance.
- Foundation for many other statistical and machine learning models.

1-1 Data Representation

Notation and Structure

- m: Number of training examples (samples).
- n: Number of features (dimensionality of input).
- $\mathbf{X}$: Input features matrix of shape (m, n).

$$\mathbf{X} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}$$

- $\mathbf{y}$: Target variable vector of shape $(m, 1)$.

$$\mathbf{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

- $\mathbf{x}^{(i)}$: Feature vector for the i-th example (column vector):
  $[\quad x^{(i)} \quad]^T$

1-2 Model Representation

The Hypothesis Function

The linear regression model hypothesizes that the target y is a linear combination of features $\mathbf{x}$ and parameters θ.

- For a single example $\mathbf{x}^{(i)}$ (an $(n+1)$-dim column vector):

$$h_\theta(\mathbf{x}^{(i)}) = \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)} = \theta^T \mathbf{x}^{(i)}$$

  Where:
  - $\theta = [\theta_0, \theta_1, \ldots, \theta_n]^T$ is the $(n+1)$-dim parameter column vector.
  - θ_0 is the intercept (bias).
  - $\theta_1, \ldots, \theta_n$ are feature coefficients.

Vectorized Form (All Examples)

$$\hat{\mathbf{y}} = \mathbf{X}\theta$$

Where:

- $\hat{\mathbf{y}}$: $(m, 1)$ vector of predictions.
- $\mathbf{X}$: $(m, n+1)$ augmented feature matrix.
- θ: $(n+1, 1)$ parameter vector.

1-3 Learning: Cost Function (MSE)

Quantifying Prediction Error

"Learning" means finding optimal θ to minimize the difference between predictions $\hat{\mathbf{y}}$ and actuals $\mathbf{y}$.

Cost Function $J(\theta)$ - Mean Squared Error (MSE)

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2$$

- $\frac{1}{m}$: Averages squared error.
- $\frac{1}{2}$: Simplifies derivative calculation.

Vectorized Cost Function

$$J(\theta) = \frac{1}{2m} (\mathbf{X}\theta - \mathbf{y})^T (\mathbf{X}\theta - \mathbf{y})$$

Goal: Find θ that minimizes $J(\theta)$.

1-4 Learning: Closed-form Solution (Normal Equation)

Direct Analytical Solution

For linear regression with MSE, we can find θ by setting $\nabla_\theta J(\theta) = \mathbf{0}$.

- Gradient: $\nabla_\theta J(\theta) = \frac{1}{m}\mathbf{X}^T(\mathbf{X}\theta - \mathbf{y})$
- Setting to zero: $\mathbf{X}^T(\mathbf{X}\theta - \mathbf{y}) = \mathbf{0}$
- Solving for θ:

$$\mathbf{X}^T\mathbf{X}\theta = \mathbf{X}^T\mathbf{y}$$

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This is the **Normal Equation**.

Considerations

- Requires $\mathbf{X}^T\mathbf{X}$ to be invertible (if features are linearly independent and $m \geq n + 1$).
- Non-invertible if: redundant features, $m < n + 1$.
- Time Complexity: Dominated by $(\mathbf{X}^T\mathbf{X})^{-1}$, which is $O(n^3)$ for an $n \times n$ matrix.
- **Pros**: No learning rate, no iterations.
- **Cons**: Expensive for large n (e.g., $n > 10,000$), $\mathbf{X}^T\mathbf{X}$ might be

1-5 Learning: Iterative Optimization (Gradient Descent)

Iteratively Finding the Minimum

Gradient Descent is a first-order iterative optimization algorithm to find a local minimum.

General Update Rule

$$\theta := \theta - \alpha \nabla J(\theta)$$

Where:

- θ: Parameters to update.
- α: **Learning rate** (scalar hyperparameter controlling step size).
- $\nabla J(\theta)$: Gradient of the cost function.

Iteratively take steps in the direction opposite to the gradient.

Three main variants based on data used for gradient computation.

1-6 Gradient Descent: Batch Gradient Descent (BGD)

Using the Entire Dataset for Each Update

Algorithm

1. Initialize θ.
2. Repeat until convergence:
   1. Compute gradient $\nabla J(\theta)$ using all m training examples:

$$\nabla J(\theta) = \frac{1}{m}\mathbf{X}^T(\mathbf{X}\theta - \mathbf{y})$$

   2. Update parameters: $\theta := \theta - \alpha \nabla J(\theta)$

Pros:

- Smooth, stable convergence.
- Deterministic updates.
- Guaranteed convergence to global min for convex loss (like MSE).

Cons:

- Computationally expensive for large datasets.
- Entire dataset must fit in memory.
- Not for online learning.

1-7 Gradient Descent: Stochastic Gradient Descent (SGD)

Using a Single Example for Each Update

Algorithm

1. Initialize θ.
2. Repeat for number of epochs:
   1. Randomly shuffle dataset $(\mathbf{X}, \mathbf{y})$.
   2. For each example $(\mathbf{x}^{(i)}, y^{(i)})$:
      1. Compute gradient $\nabla J_i(\theta)$ using only this single example:

$$\nabla J_i(\theta) = (\theta^T \mathbf{x}^{(i)} - y^{(i)})\mathbf{x}^{(i)}$$

      (Note: not averaged here usually, or $\nabla J_i(\theta) = (\mathbf{X}^{(i)}\theta - y^{(i)})(\mathbf{x}^{(i)})^T$ if $\mathbf{X}^{(i)}$ is row)
      2. Update parameters: $\theta := \theta - \alpha \nabla J_i(\theta)$

Pros:

- Much faster updates.
- Handles very large datasets (online learning).

Cons:

- High variance in updates (noisy convergence).
- May oscillate around minimum.

1-8 Gradient Descent: Mini-batch SGD

A Balance Between BGD and SGD

Algorithm

1. Initialize θ.
2. Repeat for number of epochs:
   1. Randomly shuffle dataset $(\mathbf{X}, \mathbf{y})$.
   2. Divide into mini-batches of size b.
   3. For each mini-batch $(\mathbf{X}_{\text{batch}}, \mathbf{y}_{\text{batch}})$:
      1. Compute gradient $\nabla J_{\text{batch}}(\theta)$ using current mini-batch:

$$\nabla J_{\text{batch}}(\theta) = \frac{1}{b}\mathbf{X}_{\text{batch}}^T(\mathbf{X}_{\text{batch}}\theta - \mathbf{y}_{\text{batch}})$$

      2. Update parameters: $\theta := \theta - \alpha \nabla J_{\text{batch}}(\theta)$

Pros:

- Good balance: stability & speed.
- Efficient computation via vectorization on GPUs/TPUs.
- Smoother convergence than

Cons:

- Extra hyperparameter: mini-batch size b.
- Performance can be sensitive to b.

1-9 Why MSE? (Probabilistic Interpretation)

Connection to Maximum Likelihood Estimation (MLE)

MSE is popular due to:

1. **Mathematical Convenience**: Differentiable, convex (single global minimum).
2. **Penalizes Larger Errors More**: Due to squaring.
3. **Probabilistic Interpretation (MLE)**:

MLE Assumption for Linear Regression

Assume errors $\epsilon^{(i)} = y^{(i)} - h_\theta(\mathbf{x}^{(i)})$ are Independent and Identically Distributed (IID) according to a **Gaussian (Normal) distribution** with mean 0 and variance σ^2:

$$\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$$

This implies $y^{(i)}|\mathbf{x}^{(i)}; \theta \sim \mathcal{N}(\theta^T \mathbf{x}^{(i)}, \sigma^2)$. The probability of $y^{(i)}$ given $\mathbf{x}^{(i)}, \theta$:

$$P(y^{(i)}|\mathbf{x}^{(i)}; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T \mathbf{x}^{(i)})^2}{2\sigma^2}\right)$$

1-10 Bias-Variance Tradeoff

Decomposing Generalization Error

The generalization error of a model can be decomposed into bias, variance, and irreducible error.

- **Bias**: Error from approximating a complex real-life problem with a simpler model. High bias $\implies$ model systematically misses true relationship (underfitting).
- **Variance**: How much the learned model h_θ would change if trained on different data. High variance $\implies$ model captures noise (overfitting).
- **Irreducible Error (Noise)**: Error due to inherent randomness or unmeasured variables. Cannot be reduced.

The Tradeoff

- Simple models (e.g., few features): **High Bias, Low Variance**.
- Complex models (e.g., many features, high-degree polynomial): **Low Bias, High Variance**.

Goal: Find a model that balances bias and variance for lowest total error on unseen data.

1-11 Overfitting and Underfitting

Diagnosing Model Performance

Underfitting (High Bias)

- Model is too simple.
- Fails to capture underlying data structure.
- Performs poorly on both training and test sets.
- *Example*: Fitting a line to quadratic data.

Overfitting (High Variance)

- Model learns training data too well, including noise.
- Performs very well on training set.
- Performs poorly on unseen test set.
- *Example*: Fitting a high-degree polynomial to noisy linear data.

(A conceptual plot showing underfit, good fit, and overfit curves would be illustrative here.)

1-12 Regularization: Preventing Overfitting

Penalizing Model Complexity

Regularization techniques add a penalty term to the cost function for large parameter values. This discourages model complexity.

General Regularized Cost Function

$$J_{reg}(\theta) = J_{MSE}(\theta) + \lambda P(\theta)$$

Where:

- $J_{MSE}(\theta)$: Original MSE cost function.
- $\lambda \geq 0$: **Regularization parameter** (controls penalty strength).
- $P(\theta)$: Penalty term (function of parameters).

Conventionally, the bias term θ_0 is not regularized.

1-13 Ridge Regression (L2 Regularization)

Shrinking Coefficients Towards Zero

- **Penalty Term**: $P(\theta) = \sum_{j=1}^{n} \theta_j^2 = ||\theta_{1:n}||_2^2$ (sum of squared parameters, excluding θ_0).
- **Cost Function**:

$$J_{Ridge}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

- **Effect**: Shrinks coefficients θ_j towards zero. Does not set them exactly to zero (unless $\lambda \to \infty$). Useful for multicollinearity.

Closed-form Solution (Ridge)

$$\theta = (\mathbf{X}^T\mathbf{X} + \lambda \mathbf{l}')^{-1}\mathbf{X}^T\mathbf{y}$$

Where $\mathbf{l}'$ is an $(n+1) \times (n+1)$ identity with $l'_{0,0} = 0$.

Gradient Descent Update (for $\theta_j, j > 0$)

Shows "weight decay":

1-14 Lasso Regression (L1 Regularization)

Promoting Sparsity (Feature Selection)

- **Penalty Term**: $P(\theta) = \sum_{j=1}^{n} |\theta_j| = ||\theta_{1:n}||_1$ (sum of absolute parameters, excluding θ_0).
- **Cost Function**:

$$J_{Lasso}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{m} \sum_{j=1}^{n} |\theta_j|$$

- **Effect**: Shrinks coefficients θ_j towards zero and can set some exactly to zero. Useful for automatic **feature selection**.

Solving Lasso

- No simple closed-form solution (L1 penalty not differentiable at $\theta_j = 0$).
- Requires iterative algorithms: Coordinate Descent, LARS, proximal gradient methods (ISTA, FISTA).
- For Gradient Descent-like methods, use subgradient for L1 term: $\frac{\partial |\theta_j|}{\partial \theta_j} = \text{sgn}(\theta_j)$ if $\theta_j \neq 0$.

1-15 Regularization: Connection to MAP Estimation

Bayesian Interpretation

Regularization can be seen as Maximum A Posteriori (MAP) estimation.

- **MLE**: $\theta_{MLE} = \arg\max_\theta P(\text{Data}|\theta)$ (corresponds to minimizing MSE, no regularization).
- **MAP**: $\theta_{MAP} = \arg\max_\theta P(\theta|\text{Data})$. Using Bayes' theorem: $P(\theta|\text{Data}) \propto P(\text{Data}|\theta)P(\theta)$. This is equivalent to maximizing $\log P(\text{Data}|\theta) + \log P(\theta)$.
  - $\log P(\text{Data}|\theta)$: Log-likelihood (related to MSE).
  - $\log P(\theta)$: Log of the prior distribution over parameters.

Priors and Regularization

- **Ridge (L2)** $\iff$ Gaussian prior on $\theta_j \sim \mathcal{N}(0, \tau^2)$.
  $\log P(\theta) = \text{Const} - \frac{1}{2\tau^2}\sum \theta_j^2$. Minimizing MSE $+ \lambda' \sum \theta_j^2$.
- **Lasso (L1)** $\iff$ Laplace prior on $\theta_j \sim \text{Laplace}(0, b)$.
  $\log P(\theta) = \text{Const} - \frac{1}{b}\sum |\theta_j|$. Minimizing MSE $+ \lambda'' \sum |\theta_j|$.

Choosing λ: Hyperparameter, typically via cross-validation.

1-16 Polynomial Regression

Modeling Non-linear Relationships

Extends Linear Regression to model non-linear relationships by transforming features.

Model (single feature x, degree p)

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_p x^p$$

Implementation

1. Transform original feature(s) x into polynomial features $[x, x^2, \ldots, x^p]$. Let this new feature vector be $\mathbf{z}$.
2. Apply standard Linear Regression to these transformed features $\mathbf{z}$.

Example: For $x^{(i)}$, degree $p = 2$, transformed features: $[1, x^{(i)}, (x^{(i)})^2]$.

Considerations

- **Degree** p: Hyperparameter. Higher $p \implies$ more complex, risk of overfitting.
- Feature Scaling: Very important due to different scales of

1-17 Polynomial Regression: Linear or Non-linear?

A Dialectical View

Argument for Non-linear

The relationship between the *original input* x and the *output* y is non-linear.

- Plotting y vs original x shows a curve.
- $h_\theta(x)$ is a non-linear function of x.

Argument for Linear

The model is *linear in its parameters* θ_j.

- Define new features $z_1 = x, z_2 = x^2, \ldots$.
- $h_\theta(\mathbf{z}) = \theta_0 + \theta_1 z_1 + \cdots + \theta_p z_p$.
- This is a standard linear model w.r.t. transformed features $\mathbf{z}$.
- "Linear" in "linear model" usually means linear in parameters.

Synthesis/Conclusion

Polynomial regression models non-linear relationships in the original feature space. It achieves this by transforming features so the learning

2-0 Logistic Regression: Overview

From Regression to Classification

Core Idea

While Linear Regression predicts continuous values, **Logistic Regression** is a fundamental algorithm for classification tasks.

- Models the probability of a binary outcome (e.g., 0 or 1, Spam/Not Spam).
- Can be extended for multi-class classification.

Despite "Regression" in its name, it's a classification algorithm.

2-1 Relationship to Linear Regression

Building on Linear Foundations

Logistic Regression adapts Linear Regression for classification:

1. **Linear Combination (Logit):** Starts by computing a weighted sum of features (same as Linear Regression):

$$z = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n = \theta^T \mathbf{x}$$

   Here, z is the **logit** or log-odds. It can range from $-\infty$ to $+\infty$.

2. **Transformation for Probability (Sigmoid Function):** To get a probability (0 to 1), z is passed through the **logistic function** (or **sigmoid function** $\sigma(\cdot)$):

$$h_\theta(\mathbf{x}) = \sigma(z) = \sigma(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-(\theta^T \mathbf{x})}}$$

   $h_\theta(\mathbf{x})$ is interpreted as $P(y = 1|\mathbf{x}; \theta)$. $P(y = 0|\mathbf{x}; \theta) = 1 - h_\theta(\mathbf{x})$.

Logistic Regression uses a linear model internally but squashes its output to $[0, 1]$ using sigmoid.

2-2 Logit, Logistic, and Regression Explained

Understanding the Terminology

- **Regression**: Term used because the underlying model for log-odds is linear: log-odds $= \theta^T \mathbf{x}$.
- **Logistic (Sigmoid) Function**: $\sigma(z) = \frac{1}{1+e^{-z}}$
  - S-shaped curve, maps $\mathbb{R} \to (0, 1)$.
  - $\sigma(0) = 0.5$, $\sigma(z) \to 1$ as $z \to \infty$, $\sigma(z) \to 0$ as $z \to -\infty$.
  - Derivative: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.
- **Logit Function (Log-odds)**: Inverse of logistic function. If $p = P(y = 1|\mathbf{x}; \theta)$:

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

$\frac{p}{1-p}$ is the **odds**. Logit is the **logarithm of the odds**. In Logistic Regression:

$$\log\left(\frac{P(y = 1|\mathbf{x}; \theta)}{1 - P(y = 1|\mathbf{x}; \theta)}\right) = \theta^T \mathbf{x}$$

2-3 Sigmoid (Binary) vs. Softmax (Multi-class)

Handling Different Classification Scenarios

Sigmoid (Binary Classification)

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

- Output $h_\theta(\mathbf{x})$ is $P(y = 1|\mathbf{x})$.
- Decision boundary: $P(y = 1) = 0.5 \implies z = \theta^T \mathbf{x} = 0$.
- $\theta^T \mathbf{x} > 0 \implies P(y = 1) > 0.5 \implies$ predict 1.
- $\theta^T \mathbf{x} < 0 \implies P(y = 1) < 0.5 \implies$ predict 0.

Softmax (Multi-class Classification)

For K classes, input $\mathbf{x}$:

1. Compute K linear scores (logits): $z_k = \theta_k^T \mathbf{x}$ for $k = 1, \ldots, K$.
2. Convert scores to probabilities:

$$P(y = k|\mathbf{x}; \mathbf{\Theta}) = \text{softmax}(z_k) =$$

Properties:

- Outputs are in $[0, 1]$.
- $\sum_{k=1}^{K} P(y = k|\mathbf{x}; \mathbf{\Theta}) = 1$.

(Generalization of logistic to

2-4 Data Representation (Classification)

Inputs and Targets

- m: Number of training examples.
- n: Number of features.
- $\mathbf{X}$: Input features matrix $(m, n+1)$ (augmented with $x_0 = 1$).
- $\mathbf{y}$: Target variable vector.
  - **Binary Classification**: $\mathbf{y}$ is $(m, 1)$, $y^{(i)} \in \{0, 1\}$.
  - **Multi-class Classification (K classes)**:
    - $\mathbf{y}$ as $(m, 1)$, $y^{(i)} \in \{0, 1, \ldots, K-1\}$ (class indices).
    - OR $\mathbf{y}$ as (m, K) matrix using **one-hot encoding**.
- $\mathbf{x}^{(i)}$: $(n+1)$-dim feature vector for i-th example.

2-5 Model Representation (Binary Logistic)

Hypothesis for Two Classes

Estimates $P(y = 1|\mathbf{x})$.

1. **Linear Combination (Logit)** for example $\mathbf{x}^{(i)}$:

$$z^{(i)} = \theta^T \mathbf{x}^{(i)}$$

   θ is $(n+1)$-dim parameter vector.

2. **Hypothesis (Sigmoid Function)**:

$$h_\theta(\mathbf{x}^{(i)}) = \sigma(z^{(i)}) = \sigma(\theta^T \mathbf{x}^{(i)}) = \frac{1}{1 + e^{-(\theta^T \mathbf{x}^{(i)})}}$$

Vectorized Form

1. Logits: $\mathbf{z} = \mathbf{X}\theta$ (vector of m logits)
2. Hypotheses (Probabilities): $\mathbf{h}_\theta(\mathbf{X}) = \sigma(\mathbf{X}\theta)$ (element-wise sigmoid)

2-6 Model Representation (Softmax Regression)

Hypothesis for Multiple Classes (Multinomial Logistic Regression)

For $K > 2$ classes. Each class $k \in \{0, \ldots, K-1\}$ has parameter vector θ_k.

1. **Linear Combinations (Logits per Class)** for example $\mathbf{x}^{(i)}$:

$$z_k^{(i)} = \theta_k^T \mathbf{x}^{(i)} \quad \text{for } k = 0, \ldots, K-1$$

2. **Hypothesis (Softmax Function)**: Probability of class c:

$$P(y^{(i)} = c | \mathbf{x}^{(i)}; \mathbf{\Theta}) = \text{softmax}(z_c^{(i)}) = \frac{e^{z_c^{(i)}}}{\sum_{j=0}^{K-1} e^{z_j^{(i)}}}$$

$\mathbf{\Theta} = \{\theta_0, \ldots, \theta_{K-1}\}$ is the set of all parameter vectors.

Vectorized Form

Let $\mathbf{\Theta}$ be an $(n+1) \times K$ matrix (columns are θ_k).

1. Logits: $\mathbf{Z} = \mathbf{X}\mathbf{\Theta}$ (matrix (m, K) of logits).
2. Hypotheses (Probabilities): $\mathbf{H}_\Theta(\mathbf{X}) = \text{softmax}(\mathbf{Z})$ (softmax applied row-wise).

2-7 Learning: Cost Function (Cross-Entropy)

Why Not MSE? Deriving from MLE

- Using MSE for Logistic Regression results in a non-convex cost function (many local minima).
- Instead, Logistic Regression uses **Cross-Entropy Loss** (Log Loss).

Probabilistic Motivation (MLE for Binary Logistic Regression)

Assume IID examples. $h_\theta(\mathbf{x}^{(i)}) = P(y^{(i)} = 1|\mathbf{x}^{(i)}; \theta)$. Probability of observing $y^{(i)}$:

$$P(y^{(i)}|\mathbf{x}^{(i)}; \theta) = (h_\theta(\mathbf{x}^{(i)}))^{y^{(i)}}(1 - h_\theta(\mathbf{x}^{(i)}))^{1-y^{(i)}}$$

Log-likelihood $\ell(\theta)$ for all m examples:

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

MLE: Find θ maximizing $\ell(\theta)$. This is equivalent to minimizing the **average negative log-likelihood**.

2-8 Learning: Cross-Entropy Cost Function Details

Properties and Intuition

Cross-Entropy Cost Function $J(\theta)$ (Binary)

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

This cost function is convex for Logistic Regression.

Cost for a single example: $\text{Cost}(h_\theta(\mathbf{x}), y)$

- If $y = 1$: $\text{Cost} = -\log(h_\theta(\mathbf{x}))$.
  - $h_\theta(\mathbf{x}) \to 1$ (correct): $\text{Cost} \to 0$.
  - $h_\theta(\mathbf{x}) \to 0$ (incorrect): $\text{Cost} \to \infty$.
- If $y = 0$: $\text{Cost} = -\log(1-h_\theta(\mathbf{x}))$.
  - $h_\theta(\mathbf{x}) \to 0$ (correct): $\text{Cost} \to 0$.
  - $h_\theta(\mathbf{x}) \to 1$ (incorrect): $\text{Cost} \to \infty$.

Penalizes confident wrong predictions heavily.

2-9 Learning: Gradient of Cross-Entropy

For Parameter Updates

The gradient of the Cross-Entropy cost function $J(\theta)$ (for binary logistic regression) w.r.t. θ_j is:

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

Derivation Sketch (for one example J_i)

$J_i = -[y^{(i)} \log(h_i) + (1 - y^{(i)}) \log(1 - h_i)]$, $h_i = \sigma(z^{(i)})$, $z^{(i)} = \theta^T \mathbf{x}^{(i)}$.

Chain rule: $\frac{\partial J_i}{\partial \theta_j} = \frac{\partial J_i}{\partial h_i} \cdot \frac{\partial h_i}{\partial z^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial \theta_j}$

- $\frac{\partial J_i}{\partial h_i} = \frac{h_i - y^{(i)}}{h_i(1-h_i)}$
- $\frac{\partial h_i}{\partial z^{(i)}} = \sigma'(z^{(i)}) = h_i(1 - h_i)$ (property of sigmoid)
- $\frac{\partial z^{(i)}}{\partial \theta_j} = x_j^{(i)}$

Combining gives: $\frac{\partial J_i}{\partial \theta_j} = (h_\theta(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$

2-10 Learning: Cross-Entropy for Multi-class (Softmax)

Generalizing the Loss

For multi-class classification (K classes) with Softmax.

- $y^{(i)}$: True class, often one-hot encoded vector (e.g., $[0, 1, 0]$ for class 1 of 3). $y_k^{(i)} = 1$ if true class is k.
- $P(y_k^{(i)} = 1 | \mathbf{x}^{(i)}; \mathbf{\Theta})$: Predicted probability from Softmax for class k.

Cross-Entropy Loss (Single Example i)

$$\text{Cost}_i = -\sum_{k=1}^{K} y_k^{(i)} \log(P(y_k^{(i)} = 1 | \mathbf{x}^{(i)}; \mathbf{\Theta}))$$

Since $y^{(i)}$ is one-hot, if true class is c ($y_c^{(i)} = 1$):

$$\text{Cost}_i = -\log(P(y_c^{(i)} = 1 | \mathbf{x}^{(i)}; \mathbf{\Theta}))$$

(Penalizes low probability for the true class).

Total Cost $J(\mathbf{\Theta})$

2-11 Evaluation Metrics for Classification (Part 1)

Assessing Model Performance

Common metrics, using TP (True Positive), TN (True Negative), FP (False Positive), FN (False Negative):

- **Accuracy**: $\frac{TP+TN}{TP+TN+FP+FN}$
  - Overall correctness. Can be misleading for imbalanced datasets.
- **Precision (Positive Predictive Value)**: $\frac{TP}{TP+FP}$
  - Of those predicted positive, how many actually were?
  - High precision $\implies$ fewer false positives. (e.g., spam detection)
- **Recall (Sensitivity, True Positive Rate)**: $\frac{TP}{TP+FN}$
  - Of actual positives, how many did we identify?
  - High recall $\implies$ fewer false negatives. (e.g., medical diagnosis)
- **F1-Score**: $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
  - Harmonic mean of Precision and Recall. Balances both. Useful for uneven class distribution.

2-12 Evaluation Metrics for Classification (Part 2)

More Ways to Assess Performance

- **Confusion Matrix**: Table visualizing TP, TN, FP, FN.

		Predicted Class	
		Negative	Positive
Actual Class	Negative	TN	FP
	Positive	FN	TP

- **ROC Curve (Receiver Operating Characteristic)**:
  - Plots True Positive Rate (Recall) vs. False Positive Rate ($FPR = \frac{FP}{FP+TN}$) at various probability thresholds.
  - Top-left corner = perfect model. Diagonal = random guessing.
- **AUC (Area Under the ROC Curve)**:
  - Single scalar for overall performance across thresholds.
  - AUC=1: Perfect. AUC=0.5: Random.
  - Measures separability.
- **Log Loss (Cross-Entropy Loss)**: Value of the cost function. Lower is better. Penalizes confident wrong predictions.

Choice of metric depends on problem, objectives, and class distribution.

3-0 Appendix Introduction

Illustrative Code Snippets

The following slides show excerpts from PyTorch implementations for Linear and Logistic Regression, based on the provided markdown.

- These are simplified examples for demonstration.
- Full scripts would include data loading, more complete training loops, and visualizations.
- Focus is on model definition, loss, and optimizer setup for different GD strategies.

Note: Code is typeset small to fit. Refer to original markdown for complete, runnable code.

3-1 PyTorch Linear Regression: Setup

Data Generation and Model

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from torch.utils.data import TensorDataset, DataLoader

# 1. Data Generation (Synthetic)
num_samples = 200
X_numpy = np.random.rand(num_samples, 1) * 10
true_W = np.array([[2.5]]); true_b = np.array([5.0])
y_numpy = X_numpy @ true_W + true_b + np.random.randn(num_samples, 1) * 2
X_tensor = torch.from_numpy(X_numpy.astype(np.float32))
y_tensor = torch.from_numpy(y_numpy.astype(np.float32))
dataset = TensorDataset(X_tensor, y_tensor)

# 2. Model Definition
def create_model(): # nn.Linear implements y = Wx + b
    return nn.Linear(input_features=1, output_features=1)

# 3. Loss Function
criterion = nn.MSELoss()
```

```
# 4. Training Function (Generalized)
def train_model(model, dataloader, criterion, optimizer, num_epochs=100, strategy_name=""
    ):
    # ... (loop through epochs and batches) ...
    # for inputs, targets in dataloader:
    #     predictions = model(inputs)
    #     loss = criterion(predictions, targets)
    #     optimizer.zero_grad(); loss.backward(); optimizer.step()
    # ... (logging) ...
```

3-2 PyTorch Linear Regression: Batching Strategies

BGD, SGD, Mini-batch SGD via DataLoader

Batch Gradient Descent (BGD)

```
batch_gd_model = create_model()
# DataLoader with batch_size = total number of samples
batch_gd_dataloader = DataLoader(dataset, batch_size=num_samples, shuffle=False)
batch_gd_optimizer = optim.SGD(batch_gd_model.parameters(), lr=0.005)
# train_model(batch_gd_model, batch_gd_dataloader, ...)
```

Stochastic Gradient Descent (SGD)

```
sgd_model = create_model()
# DataLoader with batch_size = 1 (shuffle is important)
sgd_dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
sgd_optimizer = optim.SGD(sgd_model.parameters(), lr=0.001) # Often smaller LR
# train_model(sgd_model, sgd_dataloader, ...)
```

Mini-batch Stochastic Gradient Descent

```
minibatch_model = create_model()
MINIBATCH_SIZE = 32
minibatch_dataloader = DataLoader(dataset, batch_size=MINIBATCH_SIZE, shuffle=True)
minibatch_optimizer = optim.SGD(minibatch_model.parameters(), lr=0.01)
```

3-3 PyTorch Binary Logistic Regression: Setup

Data and Model for Binary Classification

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 1. Data Generation (Binary Classification)
X_numpy, y_numpy = make_classification(n_samples=200, n_features=2, ...)
y_numpy = y_numpy.reshape(-1, 1) # Column vector
# ... (train_test_split, StandardScaler) ...
X_train = torch.from_numpy(X_train_np.astype(np.float32))
y_train = torch.from_numpy(y_train_np.astype(np.float32)) # Target is float for
     BCEWithLogitsLoss

# 2. Model Definition
class BinaryLogisticRegression(nn.Module):
    def __init__(self, n_input_features):
        super(BinaryLogisticRegression, self).__init__()
        self.linear = nn.Linear(n_input_features, 1) # Output 1 logit
    def forward(self, x): # Outputs raw logits
        return self.linear(x)

model_binary = BinaryLogisticRegression(n_features=X_train.shape[1])

# 3. Loss Function and Optimizer
# BCEWithLogitsLoss is numerically stable (combines Sigmoid + BCELoss)
criterion_binary = nn.BCEWithLogitsLoss()
optimizer_binary = optim.SGD(model_binary.parameters(), lr=0.1)

# 4. Training Loop: Similar to linear regression, using new model and criterion
# ... train_model(...) ...
```

3-4 PyTorch Binary Logistic Regression: Evaluation

Making Predictions

```
# 5. Evaluation
with torch.no_grad(): # No gradients needed for evaluation
    test_outputs_binary = model_binary(X_test) # Get logits
    test_probabilities = torch.sigmoid(test_outputs_binary) # Apply sigmoid for probs
    y_predicted_binary = (test_probabilities >= 0.5).float() # Threshold at 0.5

    accuracy_binary = (y_predicted_binary == y_test).sum().item() / y_test.shape[0]
    print(f'Binary Logistic Regression - Test Accuracy: {accuracy_binary:.4f}')

# 6. Visualization (Decision Boundary for 2D data)
# def plot_decision_boundary_binary(X, y, model, scaler):
    # ... (meshgrid, model predictions, contourf plot) ...
    # plt.scatter(X[:, 0].numpy(), X[:, 1].numpy(), c=y.squeeze().numpy(), ...)
    # plt.show()
```

The plot function would show data points colored by class, overlaid with the decision boundary learned by the model.

3-5 PyTorch Multinomial Logistic Regression: Setup

Data and Model for Multi-class Classification

```
# 1. Data Generation (Multi-class, e.g., 3 classes)
X_numpy_multi, y_numpy_multi = make_classification(..., n_classes=3)
# ... (train_test_split, StandardScaler) ...
X_train_multi = torch.from_numpy(X_train_np_multi.astype(np.float32))
# Target is LongTensor of class indices (0, 1, 2...) for CrossEntropyLoss
y_train_multi = torch.from_numpy(y_train_np_multi.astype(np.int64))

# 2. Model Definition
class MultinomialLogisticRegression(nn.Module):
    def __init__(self, n_input_features, n_classes):
        super(MultinomialLogisticRegression, self).__init__()
        self.linear = nn.Linear(n_input_features, n_classes) # Output K logits
    def forward(self, x): # Outputs raw logits
        return self.linear(x)

model_multi = MultinomialLogisticRegression(n_features_multi, n_classes_multi)

# 3. Loss Function and Optimizer
# CrossEntropyLoss combines LogSoftmax and NLLLoss. Expects raw logits.
criterion_multi = nn.CrossEntropyLoss()
optimizer_multi = optim.SGD(model_multi.parameters(), lr=0.1)

# 4. Training Loop: Similar structure
# ... train_model(...) ...
```

3-6 PyTorch Multinomial Logistic Regression: Evaluation

Making Predictions for Multiple Classes

```
# 5. Evaluation
with torch.no_grad():
    test_outputs_multi = model_multi(X_test_multi) # Get K logits per sample
    # Get predicted class by finding index of max logit
    _, y_predicted_multi = torch.max(test_outputs_multi, 1) # dim=1 for max over classes

    accuracy_multi = (y_predicted_multi == y_test_multi).sum().item() / y_test_multi.
        shape[0]
    print(f'Multinomial Logistic Regression - Test Accuracy: {accuracy_multi:.4f}')

# For getting probabilities from softmax model:
# with torch.no_grad():
#     test_logits = model_multi(X_test_multi)
#     test_probabilities_softmax = torch.softmax(test_logits, dim=1)
#     print(test_probabilities_softmax[:5])

# 6. Visualization (Decision Boundary for 2D data)
# def plot_decision_boundary_multi(X, y, model, scaler):
    # ... (meshgrid, model predictions with argmax, contourf plot) ...
    # plt.scatter(X[:, 0].numpy(), X[:, 1].numpy(), c=y.numpy(), ...)
    # plt.show()
```

The plot function would show data points colored by their respective classes, with regions indicating the model's decision boundaries for each class.

Thank You!

Questions?