Logistic Regression: A Practical Tutorial

Your Name

September 7, 2025

Abstract

Logistic regression is a fundamental model for binary classification, widely used due to its interpretability and probabilistic outputs. This tutorial reviews the theory, provides practical tips, and offers a Python script to generate illustrative figures.

1 Introduction

Logistic regression models the conditional probability of a binary label given features. Unlike linear regression, it uses the sigmoid link to ensure outputs lie in $[0, 1]$, making it well-suited for classification and calibrated probability estimation. Common applications include risk prediction, medical diagnosis, and click-through-rate modeling.

2 Theory and Formulas

Let $\boldsymbol{x} \in \mathbb{R}^d$ and $y \in \{0, 1\}$. The model is

$$p(y = 1 \mid \boldsymbol{x}) = \sigma(z), \quad z = w_0 + \boldsymbol{w}^\top \boldsymbol{x}, \quad \sigma(t) = \frac{1}{1 + e^{-t}}. \tag{1}$$

Equivalently, the log-odds (logit) is linear: $\log \frac{p}{1-p} = w_0 + \boldsymbol{w}^\top \boldsymbol{x}$.

Given n i.i.d. samples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, the negative log-likelihood (binary cross-entropy) is

$$\mathcal{L}(\boldsymbol{w}, w_0) = -\sum_{i=1}^{n} \left[y_i \log p_i + (1 - y_i) \log(1 - p_i) \right], \quad p_i = \sigma(w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i). \tag{2}$$

The gradient is

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \sum_{i=1}^{n} (p_i - y_i)\, \boldsymbol{x}_i, \qquad \frac{\partial \mathcal{L}}{\partial w_0} = \sum_{i=1}^{n} (p_i - y_i). \tag{3}$$

With ℓ_2-regularization (ridge), add $\frac{\lambda}{2} \|\boldsymbol{w}\|^2$ to the loss to reduce overfitting.

The decision boundary for threshold 0.5 is $\sigma(z) = 0.5 \iff z = 0$, i.e., the hyperplane $w_0 + \boldsymbol{w}^\top \boldsymbol{x} = 0$.

3 Applications and Tips

- Feature scaling helps optimization and interpretability of coefficients.
- Consider class imbalance: adjust decision threshold, use class weights, or resampling.
- Regularize to mitigate multicollinearity; ℓ_2 is standard; ℓ_1 promotes sparsity.
- Calibrated probabilities enable ranking and decision-making under costs.
- Inspect coefficients and odds ratios e^{w_j} for interpretability.

4 Python Practice

Run the following script to generate figures used in this document. It has no exotic dependencies beyond NumPy and Matplotlib; it includes a simple logistic regression implementation to avoid version issues.

Listing 1: Python script to generate figures

You can also include the source directly:

Listing 2: gen_logistic_regression_figures.py

```python
"""
Generate figures for the Logistic Regression chapter.

Figure list (saved under ./figures/):
  - sigmoid_curve.png : Sigmoid function curve
  - logistic_loss_curves.png : Per-sample logistic losses for y=0 and y=1 vs logit z
  - decision_boundary.png : 2D synthetic data with learned decision boundary
  - probability_contours.png : Predicted probability contours over a grid
  - confusion_matrix.png : Confusion matrix heatmap on a held-out split

Dependencies:
  - numpy, matplotlib

Notes on compatibility:
  - Avoids optional or newer Matplotlib parameters; uses standard pyplot API.
  - Implements a simple Logistic Regression via gradient descent to avoid external
    deps.

Usage:
  python gen_logistic_regression_figures.py
"""

from __future__ import annotations

import os
import numpy as np
import matplotlib.pyplot as plt


def sigmoid(z: np.ndarray) -> np.ndarray:
    """Numerically stable sigmoid."""
    # For large negative z, exp(-z) can overflow; using np.clip is a simple safeguard.
    z = np.clip(z, -50, 50)
    return 1.0 / (1.0 + np.exp(-z))


def binary_cross_entropy(z: np.ndarray, y: np.ndarray) -> np.ndarray:
    """Per-sample logistic loss as a function of logit z and label y in {0,1}."""
    p = sigmoid(z)
    # Clip for numerical stability in log
    eps = 1e-12
    p = np.clip(p, eps, 1.0 - eps)
    return -(y * np.log(p) + (1 - y) * np.log(1 - p))


def make_gaussian_2class(n_per_class: int = 200, seed: int = 42):
    """Generate a linearly separable-ish 2D dataset of two Gaussian blobs."""
```

```python
    rng = np.random.RandomState(seed)

    mean0 = np.array([-1.0, -1.0])
    mean1 = np.array([+1.2, +1.2])
    cov = np.array([[0.6, 0.2], [0.2, 0.6]])

    X0 = rng.multivariate_normal(mean0, cov, size=n_per_class)
    X1 = rng.multivariate_normal(mean1, cov, size=n_per_class)
    y0 = np.zeros(n_per_class, dtype=int)
    y1 = np.ones(n_per_class, dtype=int)

    X = np.vstack([X0, X1])
    y = np.concatenate([y0, y1])

    # Shuffle
    idx = rng.permutation(X.shape[0])
    X, y = X[idx], y[idx]
    return X, y


def train_logreg_gd(X: np.ndarray, y: np.ndarray, lr: float = 0.1, n_iter: int = 1000,
     reg_l2: float = 0.0, seed: int = 42):
    """Train a simple logistic regression via batch gradient descent.

    Parameters
    ----------
    X : (n_samples, n_features)
    y : (n_samples,) in {0,1}
    lr : learning rate
    n_iter : number of iterations
    reg_l2 : L2 regularization strength (applied to weights, not bias)
    seed : random seed for initialization

    Returns
    -------
    w0 : bias (float)
    w : weights (n_features,)
    history : dict with loss per iteration
    """
    rng = np.random.RandomState(seed)
    n, d = X.shape

    # Initialize small random weights for symmetry breaking
    w = rng.normal(scale=0.01, size=d)
    w0 = 0.0

    hist_loss = []
    for _ in range(n_iter):
        z = w0 + X.dot(w)
        p = sigmoid(z)
        # Gradients
        err = (p - y)
        grad_w = X.T.dot(err) / n + reg_l2 * w
        grad_b = err.mean()
        # Update
        w -= lr * grad_w
        w0 -= lr * grad_b

```

```python
        # Track loss
        loss = binary_cross_entropy(z, y).mean() + 0.5 * reg_l2 * np.dot(w, w)
        hist_loss.append(loss)

    return w0, w, {"loss": np.array(hist_loss)}


def plot_sigmoid(out_path: str):
    t = np.linspace(-10, 10, 500)
    s = sigmoid(t)
    plt.figure(figsize=(6, 4))
    plt.plot(t, s, color="tab:blue", lw=2)
    plt.axhline(0.5, color="gray", lw=1, ls="--")
    plt.axvline(0.0, color="gray", lw=1, ls="--")
    plt.title("Sigmoid Function")
    plt.xlabel("t")
    plt.ylabel("sigma(t)")
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig(out_path, dpi=300, bbox_inches="tight")
    plt.close()


def plot_logistic_losses(out_path: str):
    z = np.linspace(-10, 10, 500)
    loss_y1 = binary_cross_entropy(z, np.ones_like(z))
    loss_y0 = binary_cross_entropy(z, np.zeros_like(z))

    plt.figure(figsize=(6.5, 4.2))
    plt.plot(z, loss_y1, label="y=1", color="tab:blue", lw=2)
    plt.plot(z, loss_y0, label="y=0", color="tab:orange", lw=2)
    plt.title("Logistic Loss vs Logit z")
    plt.xlabel("z")
    plt.ylabel("Per-sample loss")
    plt.legend(frameon=False)
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig(out_path, dpi=300, bbox_inches="tight")
    plt.close()


def plot_decision_boundary_and_data(X: np.ndarray, y: np.ndarray, w0: float, w: np.
    ndarray, out_path: str):
    plt.figure(figsize=(6.8, 5.2))

    # Scatter points
    m0 = y == 0
    m1 = y == 1
    plt.scatter(X[m0, 0], X[m0, 1], s=20, c="tab:orange", alpha=0.8, label="Class 0")
    plt.scatter(X[m1, 0], X[m1, 1], s=20, c="tab:blue", alpha=0.8, label="Class 1")

    # Decision boundary w0 + w1 x + w2 y = 0
    if abs(w[1]) > 1e-12:
        xs = np.linspace(X[:, 0].min() - 0.5, X[:, 0].max() + 0.5, 200)
        ys = -(w0 + w[0] * xs) / w[1]
        plt.plot(xs, ys, color="k", lw=2, label="Decision boundary (z=0)")
    else:
        # Vertical boundary
```

```python
        x_b = -w0 / (w[0] + 1e-12)
        plt.axvline(x_b, color="k", lw=2, label="Decision boundary (z=0)")

    plt.title("Logistic Regression Decision Boundary")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.legend(frameon=False)
    plt.grid(alpha=0.25)
    plt.tight_layout()
    plt.savefig(out_path, dpi=300, bbox_inches="tight")
    plt.close()


def plot_probability_contours(X: np.ndarray, w0: float, w: np.ndarray, out_path: str):
    # Grid covering the data extent
    x_min, x_max = X[:, 0].min() - 0.8, X[:, 0].max() + 0.8
    y_min, y_max = X[:, 1].min() - 0.8, X[:, 1].max() + 0.8
    xx, yy = np.meshgrid(
        np.linspace(x_min, x_max, 200),
        np.linspace(y_min, y_max, 200),
    )
    grid = np.c_[xx.ravel(), yy.ravel()]
    z = w0 + grid.dot(w)
    p = sigmoid(z).reshape(xx.shape)

    plt.figure(figsize=(6.8, 5.2))
    cs = plt.contourf(xx, yy, p, levels=21, cmap="RdBu_r", alpha=0.8)
    cbar = plt.colorbar(cs)
    cbar.set_label("p(y=1|x)")
    # Decision contour at p=0.5 (z=0)
    plt.contour(xx, yy, p, levels=[0.5], colors=["k"], linewidths=2)
    plt.title("Predicted Probability Contours")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.tight_layout()
    plt.savefig(out_path, dpi=300, bbox_inches="tight")
    plt.close()


def plot_confusion_matrix(y_true: np.ndarray, y_prob: np.ndarray, threshold: float,
     out_path: str):
    y_pred = (y_prob >= threshold).astype(int)
    # Compute confusion matrix counts
    tp = int(((y_true == 1) & (y_pred == 1)).sum())
    tn = int(((y_true == 0) & (y_pred == 0)).sum())
    fp = int(((y_true == 0) & (y_pred == 1)).sum())
    fn = int(((y_true == 1) & (y_pred == 0)).sum())
    cm = np.array([[tn, fp], [fn, tp]], dtype=float)

    plt.figure(figsize=(4.8, 4.2))
    im = plt.imshow(cm, interpolation="nearest", cmap="Blues")
    plt.title("Confusion Matrix (thr=%.2f)" % threshold)
    plt.colorbar(im, fraction=0.046, pad=0.04)
    tick_marks = np.arange(2)
    plt.xticks(tick_marks, ["Pred 0", "Pred 1"])
    plt.yticks(tick_marks, ["True 0", "True 1"])

    # Annotate counts
```

```
    for i in range(2):
        for j in range(2):
            plt.text(j, i, "%d" % cm[i, j], ha="center", va="center", color="black")

    plt.tight_layout()
    plt.ylabel("True label")
    plt.xlabel("Predicted label")
    plt.savefig(out_path, dpi=300, bbox_inches="tight")
    plt.close()


def main():
    # Ensure output directory exists
    out_dir = os.path.join(os.path.dirname(__file__), "figures")
    if not os.path.isdir(out_dir):
        os.makedirs(out_dir)

    # 1) Sigmoid curve
    plot_sigmoid(os.path.join(out_dir, "sigmoid_curve.png"))

    # 2) Logistic losses
    plot_logistic_losses(os.path.join(out_dir, "logistic_loss_curves.png"))

    # 3) Synthetic data + split
    X, y = make_gaussian_2class(n_per_class=250, seed=42)
    # Simple train/test split
    n = X.shape[0]
    split = int(0.7 * n)
    X_train, y_train = X[:split], y[:split]
    X_test, y_test = X[split:], y[split:]

    # 4) Train simple logistic regression
    w0, w, hist = train_logreg_gd(X_train, y_train, lr=0.15, n_iter=800, reg_l2=0.01,
        seed=42)

    # 5) Decision boundary with training data
    plot_decision_boundary_and_data(X_train, y_train, w0, w, os.path.join(out_dir, "
        decision_boundary.png"))

    # 6) Probability contours over full extent
    plot_probability_contours(X, w0, w, os.path.join(out_dir, "probability_contours.
        png"))

    # 7) Confusion matrix on test set
    z_test = w0 + X_test.dot(w)
    p_test = sigmoid(z_test)
    plot_confusion_matrix(y_test, p_test, threshold=0.5, out_path=os.path.join(out_dir
        , "confusion_matrix.png"))

    print("Figures written to:", out_dir)


if __name__ == "__main__":
    main()
```

5 Result

Core illustrations are shown below.

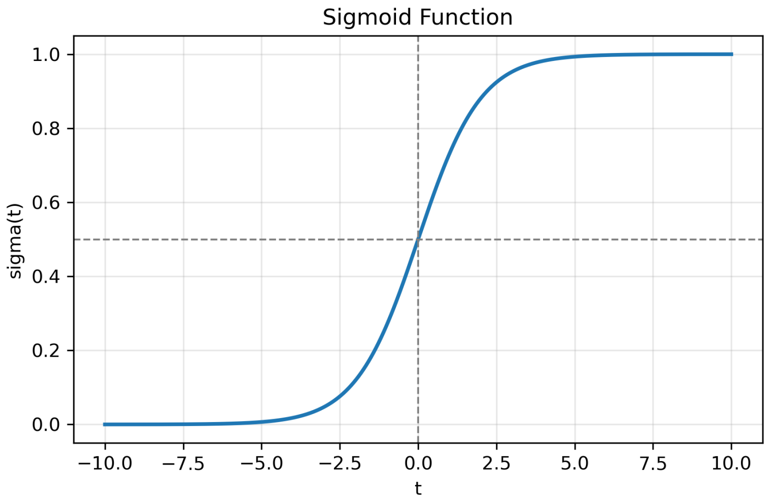

Figure 1: Sigmoid function $\sigma(t) = 1/(1 + e^{-t})$.

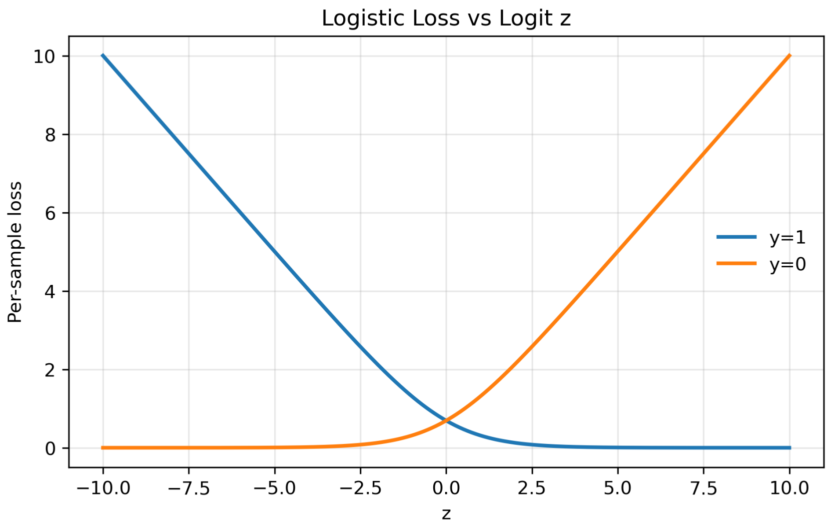

Figure 2: Binary cross-entropy per-sample loss versus logit z for $y = 0$ and $y = 1$.

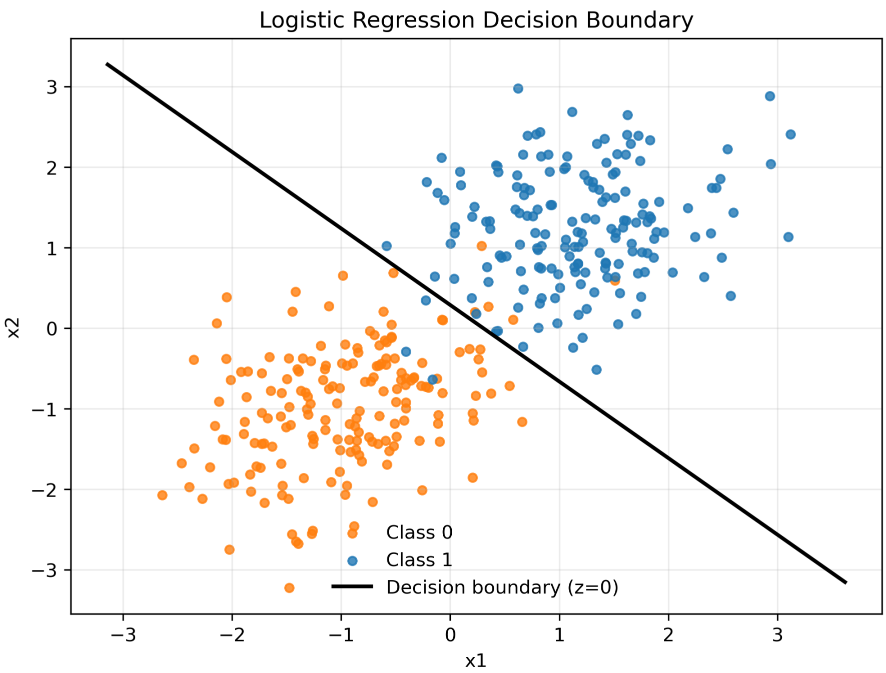

Figure 3: Synthetic 2D data and learned logistic regression decision boundary.

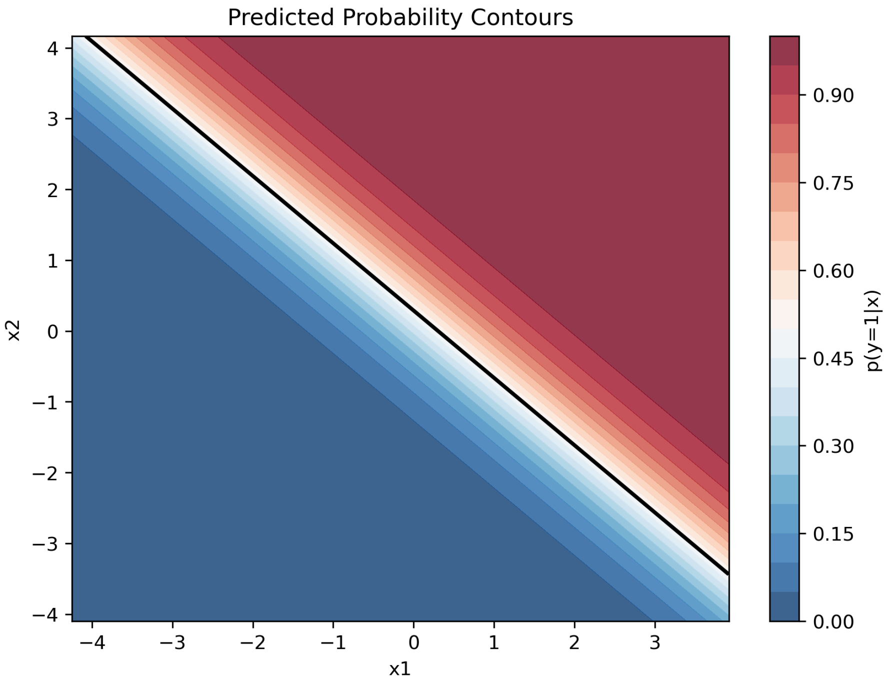

Figure 4: Predicted probability contours $p(y = 1 \mid \boldsymbol{x})$ on a grid.

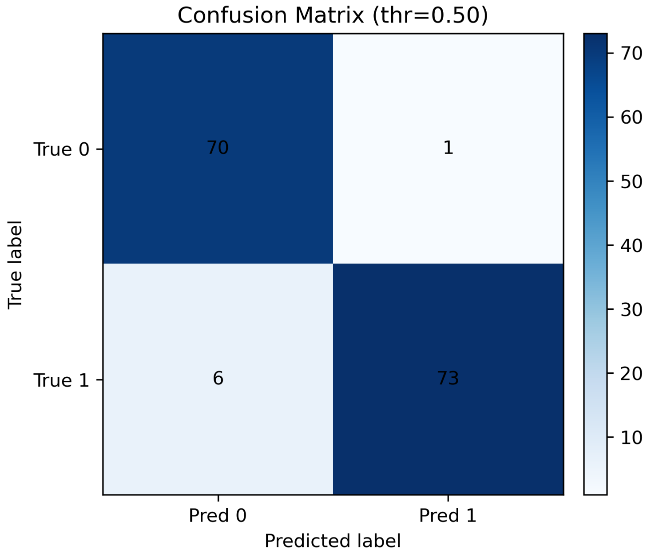

Figure 5: Confusion matrix on a held-out split at threshold 0.5.

6 Summary

Logistic regression provides a simple, interpretable, and effective classifier for many problems. Its probabilistic nature, convex training objective, and linear decision boundary make it a staple in the machine learning toolbox, often serving as a strong baseline.