Chapter 5

Logistic Regression

Logistic regression is a foundational algorithm for classification tasks, where the goal is to predict the class or category of a given input, such as determining whether an email is spam or not, or predicting whether a patient has a certain disease based on clinical features. Because of its simplicity, interpretability, and effectiveness, it is widely used as a baseline model against which more complex classifiers are compared.

Originally developed for binary classification, logistic regression models the probability that an input belongs to the "positive" class rather than the "negative" class by applying the logistic (sigmoid) function to a linear combination of the input features. This function maps real-valued inputs to the interval [0, 1], allowing the output to be interpreted as a probability. The term "regression" reflects the fact that the model's output is a continuous value between 0 and 1; however, in machine learning it is typically used for classification by thresholding the predicted probability (typically at 0.5) to produce a binary label.

When extended to multi-class problems, logistic regression is known as multinomial logistic regression or softmax regression. In this case, the softmax function replaces the sigmoid function so that the predicted probabilities across all classes sum to one, allowing the model to predict the probability of the input belonging to one of several mutually exclusive classes.

This chapter provides a comprehensive treatment of logistic regression, covering both its theoretical underpinnings and practical applications. The chapter is organized as follows. Section 5.1 introduces different types of classification problems. Section 5.2 formally defines the logistic regression model and its computational process. Section 5.3 covers optimization techniques for training logistic regression models, including the implementation of gradient-based optimization in Python. Section 5.4 demonstrates how to use Scikit-Learn's built-in classes to efficiently train logistic regression models

in practice. Section 5.5 discusses common metrics for evaluating classifiers. Section 5.6 presents methods for dealing with imbalanced datasets with highly skewed class distributions. Section 5.7 introduces general strategies for extending binary classifiers to multi-class problems. Section 5.8 introduces multinomial logistic regression and demonstrates its application to the well-known MNIST handwritten digits dataset. Section 5.9 broadens the discussion to generalized linear models (GLMs), a flexible framework that extends linear regression and logistic regression to handle a wide range of response distributions. Finally, Section 5.10 concludes with a summary.

5.1 Classification Problems

In classification problems, we are given a set of n labeled samples: $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, where $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})^T$ is the feature vector of the i-th sample, and $y_i \in 1, \ldots, k$ is its class label. The goal is to learn a function that maps input vectors to their corresponding classes.

Classification problems can be categorized based on the number and structure of the class labels:

1. **Binary classification**: Each input belongs to one of two classes ($k = 2$), such as spam ($y = 1$) or not spam ($y = 0$). The less frequent or more informative class (e.g., spam) is typically designated as the **positive** class and the other as the **negative** class.

2. **Multi-class classification**: In this setting, there are more than two possible classes ($k > 2$), and each input is assigned to exactly one of them. For example, in digit recognition, each image is classified into one of ten classes, corresponding to the digits 0, 1, ..., 9.

3. **Multi-label classification**: Each input may be associated with multiple classes simultaneously. For example, a news article may be tagged as belonging to both the "Finance" and "Technology" categories.

4. **Multi-output classification**: Each input has multiple outputs, each corresponding to a distinct classification task. For example, an object in an image might be classified by both its shape (e.g., circle, square) and its color (e.g., red, blue).

Some algorithms, such as logistic regression and support vector machines (SVMs), were originally designed for binary classification. However, there are general techniques

that can extend any binary classifier to multi-class and multi-label settings (see Section 5.7).

5.2 The Logistic Regression Model

Logistic regression is a binary classification model that estimates the probability that a given input belongs to the positive class, based on one or more predictor variables [182, 384].

Recall that in linear regression, the output is modeled as a linear function of the input: $y = \mathbf{w}^T\mathbf{x}$, which produces continuous values in $(-\infty, \infty)$. However, classification tasks require predicting discrete labels or probabilities that lie within [0, 1]. To achieve this, we apply a nonlinear transformation f to the linear predictor, producing a valid probability:

$$p = f(\mathbf{w}^T\mathbf{x}). \tag{5.1}$$

This approach defines a broad family of models known as **generalized linear models (GLMs)** (see Section 5.9), of which logistic regression is a special case. Before introducing the specific form of f used in logistic regression, we first define the concepts of odds and log-odds.

The **odds** (or **odds ratio**) of an event expresses the likelihood that the event occurs relative to it not occurring. If p denotes the probability that an event occurs, then the odds of that event are defined as the ratio of p to $1 - p$:

$$\text{odds}(p) = \frac{p}{1-p}. \tag{5.2}$$

For example, if the probability of rain tomorrow is 0.75, the odds of rain are $\frac{0.75}{1-0.75} = 3$, meaning rain is three times more likely than no rain.

In binary classification, the odds represent the ratio between the probability that a sample belongs to the positive class and the probability that it belongs to the negative class. Unlike probabilities, which range between 0 and 1, odds can take values from 0 to ∞, providing a more sensitive measure of likelihood, especially when probabilities approach 0 or 1.

The **log-odds**, or **logit**, is the natural logarithm of the odds:

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right). \tag{5.3}$$

The logit function, shown in Figure 5.1, transforms probabilities from the interval $[0, 1]$ to the entire real line $(-\infty, +\infty)$.

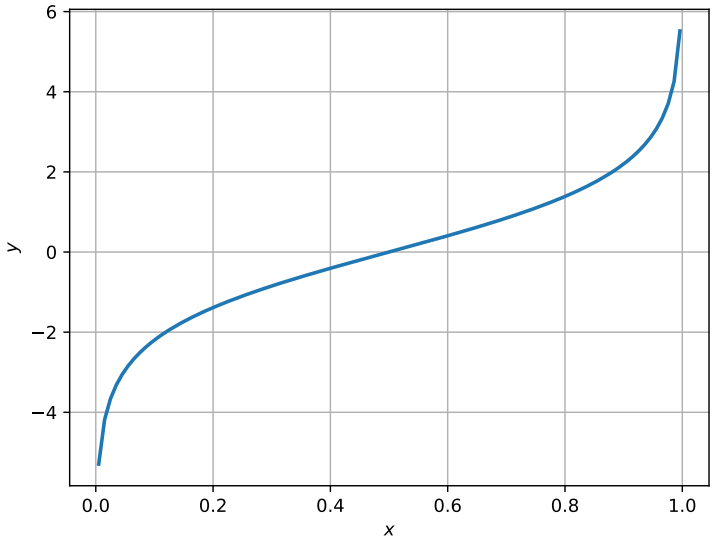

Figure 5.1: The logit function maps probabilities in [0, 1] to the real number line, transforming low probabilities into large negative values and high probabilities into large positive values.

Logistic regression assumes that the log-odds of the positive class is a linear combination of the input features. This formulation allows us to model classification using the familiar linear form, applied to the log-odds rather than directly to the output probability:

$$z = \text{logit}(p) = w_0 + w_1 x_1 + \cdots + w_d x_d = \mathbf{w}^T \mathbf{x}. \tag{5.4}$$

Here, $\mathbf{x} = (1, x_1, \ldots, x_d)^T$ is the input vector including the bias term, $\mathbf{w} = (w_0, \ldots, w_d)^T$ is the weight vector, and $p = P(y = 1|\mathbf{x})$ is the predicted probability of the positive class.

To convert the linear prediction z into a probability, we apply the **sigmoid function** (or the **logistic function**), denoted by $\sigma(z)$, which is the inverse of the logit:

$$p = \sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}}. \tag{5.5}$$

To see that the sigmoid function is the inverse of the logit, we start from the logit

equation and solve for p:

$$\log\left(\frac{p}{1-p}\right) = z$$
$$\frac{p}{1-p} = e^z$$
$$p = (1-p)e^z$$
$$p + pe^z = e^z$$
$$p(1+e^z) = e^z$$
$$p = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}} = \sigma(z).$$

The sigmoid function maps real numbers to the interval [0, 1], ensuring that the model's output is a valid probability. It has a characteristic "S"-shape curve (see Figure B.1).

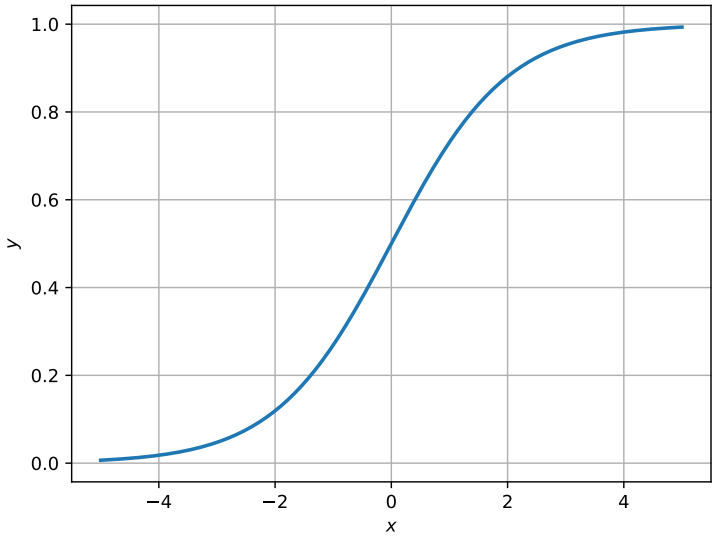

Figure 5.2: The sigmoid function: An "S"-shaped curve that maps real-valued inputs to probabilities in [0, 1]. It is used in logistic regression to convert log-odds into probabilities.

The sigmoid function also satisfies the following useful properties, which simplify

the derivation of the gradient during model training (see Exercise 5.13):

$$\sigma(-z) = 1 - \sigma(z), \tag{5.6}$$
$$\sigma'(z) = \sigma(z)(1 - \sigma(z)). \tag{5.7}$$

In summary, given an input vector $\mathbf{x}$, the logistic regression model computes the probability that the target label is 1 using:

$$p = P(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}}. \tag{5.8}$$

If the predicted probability p exceeds a certain threshold (commonly set at 0.5), the model classifies the sample as belonging to class 1; otherwise, it is classified as class 0. The computational process of logistic regression, from the input features to the final classification, is depicted in Figure 5.3.

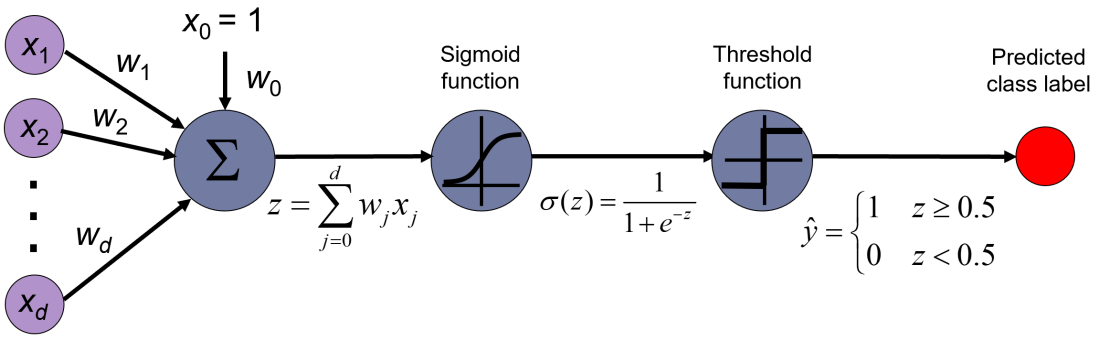

Figure 5.3: The computational process of the logistic regression model. Input features $x_1, x_2, \ldots, x_d$ are combined in a weighted sum, followed by the application of the sigmoid function to produce a probability. A threshold is then applied to assign the input into one of the two classes.

5.2.1 Numerical Example

Suppose we are studying the likelihood of individuals choosing to bike to work based on the temperature on a given day. The predictor variable x is the temperature in degrees Celsius, and the binary outcome variable y equals 1 if the individual bikes to work and 0 otherwise. We have collected five data samples, as shown in Table 5.1.

Temperature (°C)	Bikes to work
10	0
15	0
20	1
25	1
30	1

Table 5.1: Sample data for examining the relationship between temperature and the decision to bike to work

We can use logistic regression to predict the probability that someone will bike to work based on the temperature. The logistic regression model is given by:

$$P(y = 1|x) = \frac{1}{1 + e^{-(w_0 + w_1 x)}},$$

where w_0 is the intercept and w_1 is the coefficient for temperature.

Let's assume that through some estimation process (described in Section 5.3), we obtain the parameter values $w_0 = -12.94$ and $w_1 = 0.74$. The resulting model becomes:

$$P(y = 1|x) = \frac{1}{1 + e^{-(-12.94 + 0.74x)}}.$$

Using this model, we can compute the predicted probability of biking to work at various temperatures:

$$P(y = 1|x = 10) = \frac{1}{1 + e^{-(-12.94 + 0.74 \cdot 10)}} = 0.0039,$$

$$P(y = 1|x = 20) = \frac{1}{1 + e^{-(-12.94 + 0.74 \cdot 20)}} = 0.8640,$$

$$P(y = 1|x = 25) = \frac{1}{1 + e^{-(-12.94 + 0.74 \cdot 25)}} = 0.9961.$$

At 10°C, there is only a 0.39% chance of biking to work. At 20°C, the probability increases sharply to 86.4%, and by 25°C, it rises to nearly 100%.

Figure 5.4 shows the logistic regression curve based on this model. The sigmoid shape reflects the smooth transition from low to high probability as temperature increases. The horizontal dashed line at $y = 0.5$ represents the classification threshold used to distinguish between the two outcomes.

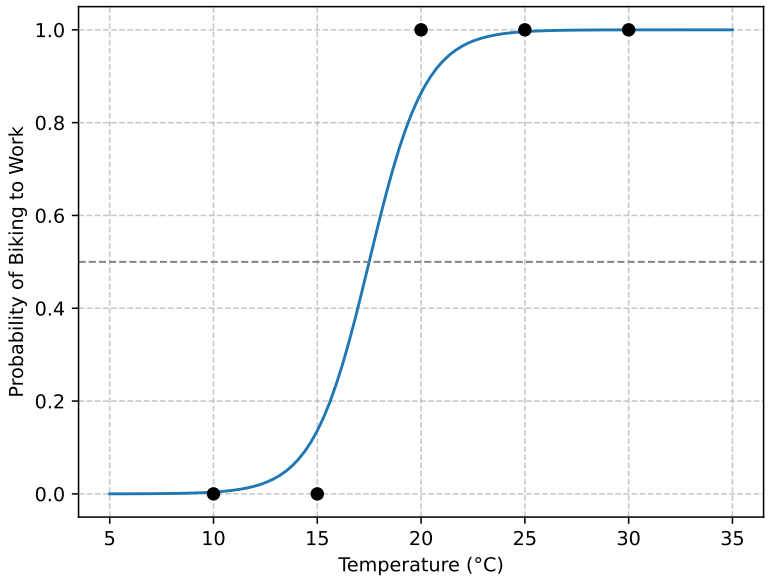

Figure 5.4: The sigmoidal curve of the logistic regression model predicting the probability of biking to work as a function of temperature. The curve transitions smoothly from low to high probability, with the decision boundary indicated by the horizontal line at $y = 0.5$.

5.2.2 Decision Boundaries

In classification problems, **decision boundaries** refer to the set of points in the feature space where a model assigns equal probability (or score) to two or more classes. Understanding these boundaries can provide insight into the model's ability to distinguish between classes and helps identify regions of uncertainty, where the model's confidence is low.

In logistic regression, the decision boundary between the two classes is defined by the set of points where $p = 0.5$, which corresponds to a log-odds value of zero: $\mathbf{w}^T\mathbf{x} = 0$. This equation describes a hyperplane in the feature space that is orthogonal to the weight vector $\mathbf{w}$ (see Figure 5.5). Points on one side of this hyperplane ($\mathbf{w}^T\mathbf{x} > 0$) are classified as positive, while those on the other side ($\mathbf{w}^T\mathbf{x} < 0$) are classified as negative.

The linearity of this boundary makes logistic regression a **linear classifier**, meaning it separates the classes using a linear decision surface. Linear models are known for their simplicity and interpretability, making them a popular choice in many practical

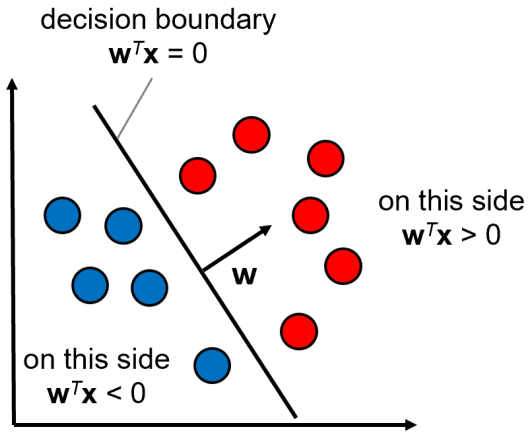

Figure 5.5: The decision boundary in logistic regression is defined by the hyperplane $\mathbf{w}^T\mathbf{x} = 0$, which separates the feature space into two regions. Points for which $\mathbf{w}^T\mathbf{x} > 0$ are classified as positive (red), while those for which $\mathbf{w}^T\mathbf{x} < 0$ are classified as negative (blue). The weight vector $\mathbf{w}$ is orthogonal to the decision boundary.

applications.

5.3 Training a Logistic Regression Model

When training logistic regression models, the objective is to find a parameter vector $\mathbf{w}$ that makes the predicted probabilities $p = \sigma(\mathbf{w}^T\mathbf{x})$ as close as possible to the actual class labels y. To achieve this, we define a suitable loss function that measures the discrepancy between the model's predictions and the true labels, and then apply an optimization algorithm to minimize this function.

A natural candidate for binary classification is the **0–1 loss function**, which simply checks whether a prediction is correct or not:

$$L_{0\text{–}1}(y, \hat{y}) = \begin{cases} 1 & \hat{y} \neq y, \\ 0 & \hat{y} = y, \end{cases} \tag{5.9}$$

where y is the true label of the sample and $\hat{y}$ is the predicted label. However, this function is non-differentiable and it ignores the model's predicted probabilities, making it unsuitable for optimizing logistic regression models.

Another common loss function—the squared loss, which we used in linear regression—is also inappropriate for logistic regression, for several reasons: it assumes a continuous

target and normally distributed errors; it leads to a non-convex objective when combined with the sigmoid function; and it lacks a meaningful probabilistic interpretation for classification.

5.3.1 Log Loss

Given the limitations of the 0-1 loss and squared loss for logistic regression, we need a loss function that reflects the probabilistic nature of the model and can be optimized using gradient-based methods.

As discussed in Section 2.5, the **negative log-likelihood (NLL)** provides a principled way to define loss functions in probabilistic models. To apply it in logistic regression, we first need to specify a distribution for the target variable. In binary classification, a natural choice is the Bernoulli distribution, where a random variable takes the value 1 with probability p and 0 with probability $1 - p$ (see Section C.5.6.1).

Under this model, for a given sample $(\mathbf{x}, y)$, the likelihood of observing a positive label is $P(y = 1|\mathbf{x}) = p$, where p is the probability estimated by the model. Similarly, the likelihood of observing a negative sample is $P(y = 0|\mathbf{x}) = 1 - p$. These two cases can be combined into a single expression:

$$P(y|\mathbf{x}) = p^y(1 - p)^{1-y}. \tag{5.10}$$

This formula compactly represents both possibilities: when $y = 1$, it simplifies to $P(y|\mathbf{x}) = p$; and when $y = 0$, it yields $P(y|\mathbf{x}) = 1 - p$.

Thus, the log-likelihood of observing the true label under the model is:

$$\log P(y|\mathbf{x}) = y \log p + (1 - y) \log(1 - p). \tag{5.11}$$

Negating this log-likelihood gives us the **log loss**—also known as the **logistic loss** or **binary cross-entropy loss**—defined as:

$$L_{\log}(y, p) = -y \log p - (1 - y) \log(1 - p). \tag{5.12}$$

Figure 5.6 shows the log loss as a function of p when $y = 1$. For $y = 0$, the curve is mirrored, exhibiting the same behavior from the perspective of the opposite class. As shown, the log loss reaches its minimum value of 0 when the prediction is perfect ($p = y$) and approaches infinity as the model becomes increasingly confident in the wrong class (i.e., $p \to 0$ when $y = 1$, and vice versa).

To evaluate the model's performance on the entire training set, we define a cost function that averages the log loss across all the training samples:

$$J(\mathbf{w}) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log p_i + (1 - y_i) \log(1 - p_i) \right]. \tag{5.13}$$

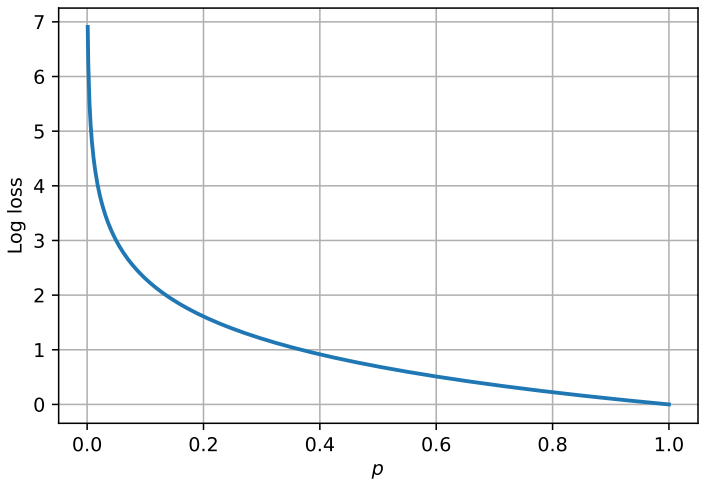

Figure 5.6: The log loss as a function of the predicted probability p for when the true label is $y = 1$. The loss increases sharply as p approaches 0, penalizing incorrect predictions more heavily when the model is confident in them.

For computational efficiency, this function can be written in vectorized form as:

$$J(\mathbf{w}) = -\frac{1}{n}\left(\mathbf{y}^T \log \mathbf{p} + (\mathbf{1} - \mathbf{y}^T) \log(\mathbf{1} - \mathbf{p})\right). \tag{5.14}$$

Here, $\mathbf{y} = (y_1, \ldots, y_n)$ and $\mathbf{p} = (p_1, \ldots, p_n)$ are vectors of the true labels and predicted probabilities for all training samples, respectively, and $\mathbf{1}$ is a vector of ones of size n. The $\log(\mathbf{x})$ operation is applied element-wise on the vector $\mathbf{x}$.

This cost function is convex, meaning that any local minimum is also a global minimum. However, due to the nonlinearities introduced by the sigmoid and logarithm functions, there is no closed-form solution for the optimal parameters $\mathbf{w}^*$. As a result, we must use iterative optimization techniques, such as gradient descent or Newton's method, to minimize the cost. These methods are explored in the following subsections.

5.3.2 Gradient Descent

To minimize the cost function $J(\mathbf{w})$ using gradient descent, we compute its partial derivatives with respect to each weight w_j as follows:

1. Start with the definitions of the cost function and the predicted probabilities

$p_i = \sigma(\mathbf{w}^T x_i)$:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \frac{\partial}{\partial w_j} \left[-\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\sigma(\mathbf{w}^T \mathbf{x}_i)) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] \right]$$

2. Move the derivative inside the summation:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \frac{\partial}{\partial w_j} \log(\sigma(\mathbf{w}^T \mathbf{x}_i)) + (1 - y_i) \frac{\partial}{\partial w_j} \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right]$$

3. Apply the chain rule to differentiate the logarithmic terms:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = -\frac{1}{n} \sum_{i=1}^{n} \left[\left(\frac{y_i}{\sigma(\mathbf{w}^T \mathbf{x}_i)} - \frac{1 - y_i}{1 - \sigma(\mathbf{w}^T \mathbf{x}_i)} \right) \frac{\partial}{\partial w_j} \sigma(\mathbf{w}^T \mathbf{x}_i) \right]$$

4. Differentiate the sigmoid function using $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, and note that $\frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}_i = x_{ij}$:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = -\frac{1}{n} \sum_{i=1}^{n} \left[\left(\frac{y_i}{\sigma(\mathbf{w}^T \mathbf{x}_i)} - \frac{1 - y_i}{1 - \sigma(\mathbf{w}^T \mathbf{x}_i)} \right) \sigma(\mathbf{w}^T \mathbf{x}_i) \left(1 - \sigma(\mathbf{w}^T \mathbf{x}_i) \right) x_{ij} \right]$$

5. Simplify the expression:

$$\begin{aligned} \frac{\partial}{\partial w_j} J(\mathbf{w}) &= -\frac{1}{n} \sum_{i=1}^{n} \left[\left(y_i \left(1 - \sigma(\mathbf{w}^T \mathbf{x}_i) \right) - (1 - y_i) \sigma(\mathbf{w}^T \mathbf{x}_i) \right) x_{ij} \right] \\ &= -\frac{1}{n} \sum_{i=1}^{n} \left[\left(y_i - \sigma(\mathbf{w}^T \mathbf{x}_i) \right) x_{ij} \right] \\ &= \frac{1}{n} \sum_{i=1}^{n} \left[(p_i - y_i) x_{ij} \right]. \end{aligned} \tag{5.15}$$

That is, the partial derivative of the cost function with respect to w_j is the average over the training samples of the product of the prediction error $(p_i - y_i)$ and the corresponding feature value x_{ij}.

For computational efficiency, the full gradient of the cost function can be written in vectorized form as follows:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{n} X^T (\mathbf{p} - \mathbf{y}), \tag{5.16}$$

where X^T is the transpose of the feature matrix X, $\mathbf{p}$ contains the model's predicted probabilities for all samples, and $\mathbf{y}$ contains the corresponding true labels. The difference $\mathbf{p} - \mathbf{y}$ represents the prediction errors for all the training examples.

In batch gradient descent, the weights are updated iteratively according to the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{1}{n} X^T (\mathbf{p} - \mathbf{y}), \tag{5.17}$$

where α is a learning rate that controls the step size of each update.

In stochastic gradient descent (SGD), the weights are updated after each individual training sample. The update rule for SGD is:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha (p - y)\mathbf{x}, \tag{5.18}$$

where p is the predicted probability, y is the true label, and $\mathbf{x}$ is the feature vector of the current sample.

5.3.3 Implementation in Python

In this section, we implement the logistic regression model in Python from scratch. This includes computing the cost function and its gradient, optimizing the model using gradient descent, evaluating the model's performance, and visualizing the resulting decision boundaries.[1]

For this demonstration, we use the Iris dataset introduced in Section 3.5. We simplify the original three-class problem by considering only setosa and versicolor flowers, reducing it to a binary classification task. In addition, we use only the first two features: sepal width and sepal length, to allow for easier visualization of the decision boundary in a two-dimensional space.

We start by importing the required libraries and fixing the random seed for reproducibility:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

np.random.seed(42)
```

[1] Implementing machine learning algorithms from scratch is a highly valuable exercise that can deepen your understanding of how they work and may spark ideas for algorithmic improvements or even entirely new methods. Throughout this book, you will find several opportunities to do so.

5.3.3.1 Data Loading and Visualization

We proceed by loading the Iris dataset, selecting only the first two features (sepal length and sepal width), and filtering the samples to include only setosa and versicolor flowers (class 0 and class 1):

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data[:, :2] # Use only the first two features
y = iris.target
class_labels = iris.target_names

# Filter for setosa and versicolor flowers
X = X[(y == 0) | (y == 1)]
y = y[(y == 0) | (y == 1)]
```

To gain an initial understanding of the data distribution, we visualize the dataset using the following function:

```
def plot_data(X, y):
    sns.scatterplot(x=X[:, 0], y=X[:, 1],
                    hue=class_labels[y], style=class_labels[y],
                    palette=['r','b'], markers=('s','o'), edgecolor='k')
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])
    plt.legend()

plot_data(X, y)
```

The function `sns.scatterplot` from Seaborn creates a scatter plot of the samples, with color and marker style determined by their class label. (Reproducing the same styling in Matplotlib would require significantly more code.) As illustrated in Figure 5.7, the two classes are linearly separable, making this dataset well-suited for a linear classification model such as logistic regression.

5.3.3.2 Data Preparation

Although both features are numeric, they have different ranges: sepal length ranges from 4.3 to 7.0, while sepal width ranges from 2.0 to 4.4. Although the feature ranges are relatively similar, standardizing them can still improve the convergence of gradient

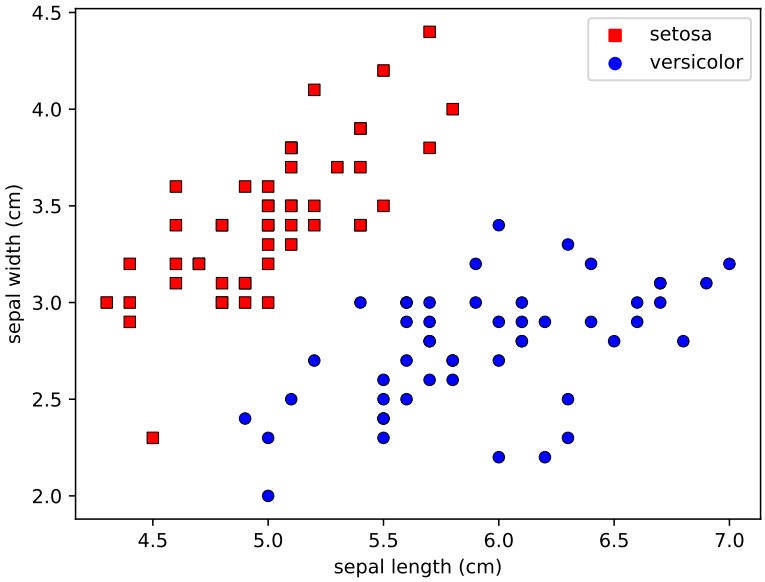

Figure 5.7: Scatter plot of the Iris dataset showing sepal width and sepal length for two species: setosa (squares) and versicolor (circles). Since the data is linearly separable, it is well-suited for linear classifiers such as logistic regression.

descent.

Additionally, we need to add a bias term to the model by appending a column of ones to the feature matrix X. This step should be performed after feature scaling, as the bias term must remain a column of ones (standardizing it would incorrectly convert it into a column of zeros).

The complete data preparation process is shown below:

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

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Add a column of ones for the bias term
n_train = X_train_scaled.shape[0]
n_test = X_test_scaled.shape[0]
X_train_b = np.hstack((np.ones((n_train, 1)), X_train_scaled))
X_test_b = np.hstack((np.ones((n_test, 1)), X_test_scaled))
```

5.3.3.3 Model Implementation

We are now ready to implement the logistic regression model. First, we define a utility function for computing the sigmoid function:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Next, we introduce a function for computing both the logistic regression cost function and its gradient, as defined in Equations (5.14) and (5.16):

```
def cost_function(X, y, w):
    """Compute the logistic regression cost and its gradient.
    Args:
        X: Feature matrix (with bias term)
        y: Target labels
        w: Model parameters
    Returns: Tuple containing the cost and gradient
    """
    n = len(y) # Number of training samples
    p = sigmoid(np.dot(X, w)) # Predicted probabilities for y = 1
    cost = -(1/n) * (y @ np.log(p) + (1 - y) @ np.log(1 - p)) # Average log
        loss
    grad = (1/n) * X.T @ (p - y) # Gradient of the cost

    return cost, grad
```

We now implement batch gradient descent to optimize the model parameters (see Algorithm E.1):

```
def optimize_model(X, y, alpha=0.1, max_iter=1000, tol=0.0001):
    """Optimize the model parameters using batch gradient descent.
    Args:
        X: Feature matrix (with bias term)
```

```
        y: Target labels
        alpha: Learning rate
        max_iter: Maximum number of iterations
        tol: Tolerance for the stopping criterion
    Returns:
        w: The optimized parameters
        cost_history: List of cost values at each iteration
    """
    w = np.random.rand(X.shape[1]) # Random parameter initialization
    cost_history = [] # Track cost at each iteration

    for i in range(max_iter):
        cost, grad = cost_function(X, y, w) # Compute cost and gradient
        w = w - alpha * grad # Update parameters
        cost_history.append(cost)

        # Check for convergence
        if i > 0 and (cost_history[-2] - cost) < tol:
            print(f'Convergence reached after {i} iterations.')
            return w, cost_history

    print('Reached maximum iterations without convergence.')
    return w, cost_history
```

This function initializes the model parameters randomly and iteratively updates them in the direction of the negative gradient. This process continues until either the maximum number of iterations is reached or the cost improvement falls below the specified tolerance. The function returns both the optimized parameters and the history of cost values, which can be used later to analyze the learning process.

We now apply this function to find the optimal parameters for our logistic regression model:

```
w_opt, cost_history = optimize_model(X_train_b, y_train)
print('Optimized weights:', w_opt)
```

The output is:

```
Convergence reached after 395 iterations.
Optimized weights: [ 0.42026863  2.88499483 -2.2992438 ]
```

We now plot the learning curve using the cost history:

```
plt.plot(cost_history)
plt.xlabel('Iteration')
plt.ylabel('Cost')
```

The learning curve shown in Figure 5.8 demonstrates a rapid decrease in cost during the early iterations, followed by a gradual leveling off as the model converges to the optimal parameter values.

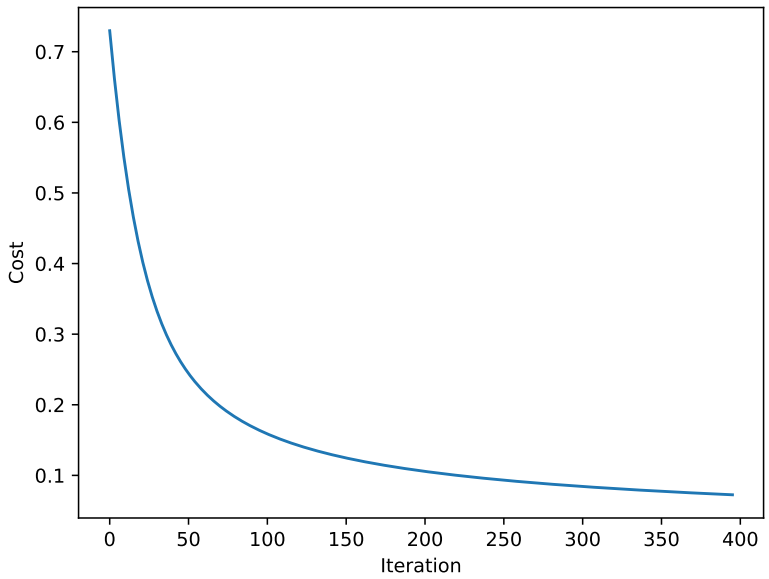

Figure 5.8: Learning curve of logistic regression on the simplified Iris dataset. The cost steadily decreases throughout the iterations, with a sharper drop at the beginning, and gradually levels off as the model approaches convergence.

5.3.3.4 Making Predictions

Now that we have optimized the model's parameters, we can use them to make predictions on new data. Since the model was trained on standardized features, any new input must be scaled using the same transformation applied during training. Additionally, we need to add a bias term (a column of ones) to the scaled feature matrix before computing the predicted probabilities.

The following function computes the probability that a given sample (or group of samples) belongs to the positive class, after scaling the input features and adding the bias term:

```
def predict_prob(X_new, w_opt, scaler):
    """Compute the probability of the positive class for samples in X_new."""
    X_new_scaled = scaler.transform(X_new)
    X_new_b = np.hstack((np.ones((X_new_scaled.shape[0], 1)), X_new_scaled))
    p = sigmoid(X_new_b @ w_opt)
    return p
```

For example, we can use this function to estimate the probability that a new flower with sepal length of 6 cm and sepal width of 2 cm is a versicolor (the positive class):

```
prob = predict_prob([[6, 2]], w_opt, scaler)
print(prob)
```

```
[0.9996311]
```

The output indicates a 99.96% chance that the flower is versicolor, demonstrating the model's strong confidence in this prediction, as the sample is located well within the versicolor region. Conversely, for a sample with sepal length of 5.5 cm and sepal width of 3.1 cm, which lies closer to the decision boundary, we have:

```
prob = predict_prob([[5.5, 3.1]], w_opt, scaler)
print(prob)
```

```
[0.59593864]
```

This produces a lower probability (59.59%), reflecting the model's greater uncertainty near the decision boundary.

To classify samples based on these probabilities, we apply a threshold (by default 0.5) to determine the class assignment. The following function implements this thresholding:

```
def predict(X_new, w_opt, scaler, threshold=0.5):
    """Classify samples as 0 or 1 using a probability threshold."""
    p = predict_prob(X_new, w_opt, scaler)
    y_pred = (p >= threshold).astype(int)
    return y_pred
```

Applying this function to our earlier examples:

```
y_pred = predict([[6, 2], [5.5, 3]], w_opt, scaler)
y_pred
```

The output is:

```
array([1, 1])
```

This result indicates that both samples are classified as versicolor.

5.3.3.5 Model Evaluation

We now evaluate the performance of our trained model. A key evaluation metric for classification tasks is **accuracy**, which measures the proportion of correctly predicted instances:

$$\text{accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}(y_i = \hat{y}_i), \tag{5.19}$$

where $\mathbb{1}$ is the indicator function that returns 1 if the condition inside is true and 0 otherwise. A related metric is the **error rate**, which represents the proportion of misclassified instances. It is simply the complement of accuracy:

$$\text{error rate} = 1 - \text{accuracy}. \tag{5.20}$$

The following function computes the model's accuracy on a given dataset:

```
def evaluate_model(X, y, w_opt, scaler):
    """Calculate the model's accuracy on the given dataset."""
    y_pred = predict(X, w_opt, scaler) # Predict class labels
    accuracy = np.mean(y == y_pred) # Proportion of correct predictions
    return accuracy
```

Using this function, we now assess the model's accuracy on the training and test sets:

```
train_accuracy = evaluate_model(X_train, y_train, w_opt, scaler)
print(f'Train accuracy: {train_accuracy:.4f}')

test_accuracy = evaluate_model(X_test, y_test, w_opt, scaler)
print(f'Test accuracy: {test_accuracy:.4f}')
```

The output is:

```
Train accuracy: 1.0000
Test accuracy: 1.0000
```

These scores indicate that the model perfectly separates the two classes on both the training and test sets.

5.3.3.6 Plotting the Decision Boundary

Since our dataset is two-dimensional, we can visualize the decision boundary that separates the two classes. This boundary corresponds to the region where the model assigns equal probabilities to both classes, i.e., where $\sigma(\mathbf{w}^T\mathbf{x}) = 0.5$, which implies $\mathbf{w}^T\mathbf{x} = 0$. For a two-dimensional feature space, this is a line defined by:

$$w_0 + w_1 x_1 + w_2 x_2 = 0.$$

Rearranging this equation gives an explicit form for plotting:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2},$$

where $-\frac{w_1}{w_2}$ is the slope and $-\frac{w_0}{w_2}$ is the y-intercept. The following function plots this decision boundary:

```
def plot_decision_boundary(X, y, w_opt):
    """Plot the decision boundary between the classes."""
    # Extract the min and max values of the first feature (x1)
    x1_min, x1_max = X[:, 0].min(), X[:, 0].max()

    # Create a range of values for x1
    line_x = np.array([x1_min, x1_max])

    # Compute the corresponding y values (x2) using the decision boundary
        equation
    line_y = -1 / w_opt[2] * (w_opt[1] * line_x + w_opt[0])

    # Plot the decision boundary as a dashed line
    plt.plot(line_x, line_y, c='k', ls='--')
```

We now plot the decision boundary over the scaled feature space:

```
X_scaled = scaler.transform(X)
plot_data(X_scaled, y)
plot_decision_boundary(X_scaled, y, w_opt)
```

Figure 5.9 illustrates the decision boundary. As shown, the logistic regression model perfectly separates the two classes.

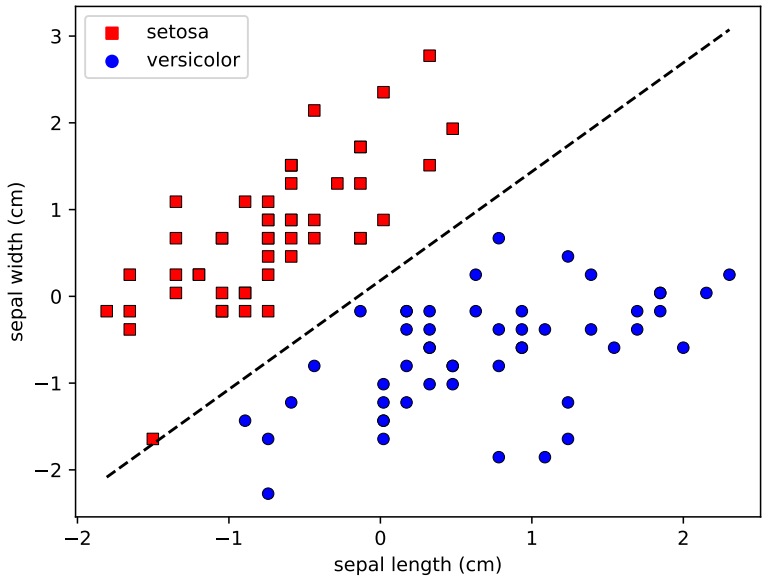

Figure 5.9: The decision boundary (dashed line) produced by the logistic regression model, perfectly separating the two flower species: setosa (squares) and versicolor (circles), based on sepal length and sepal width. The boundary represents the threshold where the model assigns equal probabilities to both classes.

5.3.4 Newton's Method

An alternative and often more efficient optimization technique for logistic regression is **Newton's method**. Unlike gradient descent, which uses only first-order information (the gradient), Newton's method also incorporates second-order information via the Hessian matrix. This enables it to adjust both the search direction and the step size at each iteration based on the local curvature of the cost function. As a result, Newton's method often converges much faster—with a quadratic convergence rate, compared to the linear rate of gradient descent—particularly near the minimum. For a detailed discussion of Newton's method, see Section E.5.1.

When applied to logistic regression, Newton's method updates the weight vector

using the following rule:

$$\mathbf{w} \leftarrow \mathbf{w} - H^{-1}(\mathbf{w})\nabla_{\mathbf{w}} J(\mathbf{w}), \tag{5.21}$$

where:

- $\nabla_{\mathbf{w}} J(\mathbf{w})$ is the gradient of the cost function at $\mathbf{w}$, as given in Equation 5.16.
- $H^{-1}(\mathbf{w})$ is the inverse of the Hessian matrix of the cost function at $\mathbf{w}$. In Exercise 5.15, you will derive the explicit form of the Hessian and examine when it is invertible.

Computing and inverting the Hessian can be computationally expensive, especially for large datasets or high-dimensional models. To address this, **quasi-Newton methods** such as L-BFGS and Newton-CG (see Section E.5.2) offer scalable alternatives by approximating the Hessian rather than computing it directly.

5.3.5 Regularized Logistic Regression

As in linear regression, regularization can be applied to logistic regression models to reduce overfitting, especially in high-dimensional settings [725]. Common types of regularization include:

- **L2 regularization (ridge)** adds a penalty equal to the sum of the squared weights to the cost function, discouraging large coefficients and leading to a smoother decision boundary. The cost function becomes:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda \|\mathbf{w}\|_2^2, \tag{5.22}$$

  where λ is the regularization coefficient controlling the strength of the penalty.

- **L1 regularization (lasso)** adds a penalty equal to the sum of absolute values of the weights to the cost function, encouraging sparsity by driving some coefficients exactly to zero. This allows the model to automatically select a relevant subset of features for the classification task. The cost function becomes:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda \|\mathbf{w}\|_1. \tag{5.23}$$

- **Elastic net regularization** combines the L1 and L2 penalties, balancing sparsity and coefficient shrinkage. This is especially useful when there is multicollinearity or when both regularization and feature selection are desired:

$$J(\mathbf{w}) = \left[-\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda_1 \|\mathbf{w}\|_1 + \lambda_2 \|\mathbf{w}\|_2^2, \tag{5.24}$$

  where λ_1 and λ_2 control the relative weights of the two penalties.

Optimization techniques such as gradient descent and quasi-Newton methods can also be applied to minimize these regularized cost functions.

5.4 Logistic Regression in Scikit-Learn

Scikit-Learn provides the following classes for training logistic regression models:

- `LogisticRegression` offers a highly optimized implementation that supports several solvers, including L-BFGS, Newton-CG, and SAG (Stochastic Average Gradient). These solvers provide fast and reliable convergence for a wide range of problem sizes.
- `SGDClassifier` fits linear classifiers using stochastic gradient descent (SGD), making it particularly suitable for very large datasets and online learning. While more scalable, it typically converges more slowly than `LogisticRegression` and requires careful tuning of hyperparameters such as the learning rate.

In this section, we explore both classes and demonstrate how to use them on the Iris dataset.

5.4.1 The `LogisticRegression` Class

Scikit-learn provides the `LogisticRegression` class for fitting logistic regression models. This class offers several efficient solvers, including L-BFGS (the default), Newton-CG, and SAG, which are generally more effective than gradient descent for this task. See Appendix E for more details on these algorithms.

Additionally, `LogisticRegression` supports various regularization techniques, including L1, L2, and elastic net, as well as multinomial logistic regression for multi-class problems (see Section 5.8). Table 5.2 summarizes the key parameters of the class.

Parameter	Description	Default
`solver`	Optimization algorithm to use. Options include: `'lbfgs'`, `'liblinear'`, `'newton-cg'`, `'newton-cholesky'`, `'sag'`, or `'saga'`.	`'lbfgs'`
`penalty`	Type of regularization to apply. Options are: `'l1'`, `'l2'`, `'elasticnet'`, or `None`.	`'l2'`
`C`	Inverse of the regularization coefficient ($1/\lambda$); smaller values specify stronger regularization.	`1.0`
`l1_ratio`	Elastic net mixing parameter that controls the balance between L1 and L2 penalties. Used only when `penalty='elasticnet'`.	`None`
`max_iter`	Maximum number of iterations for the solver to converge.	`100`
`tol`	Tolerance for the stopping criterion.	`0.0001`
`class_weight`	Weights associated with classes, given as `{class:weight}`. useful for handling imbalanced datasets.	`None`

Table 5.2: Key parameters of the `LogisticRegression` class

For example, let's apply the `LogisticRegression` class to the simplified Iris dataset from Section 5.3.2. First, we load the dataset and split it into training and test sets, using the same random seed to allow comparison with our previous results:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
X = iris.data[:, :2] # Use only the first two features
y = iris.target
class_labels = iris.target_names

# Filter for setosa and versicolor flowers
X = X[(y == 0) | (y == 1)]
y = y[(y == 0) | (y == 1)]

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

Note that `LogisticRegression` automatically handles the addition of the bias term to the feature matrix. We now build a pipeline consisting of a `StandardScaler` and a `LogisticRegression` classifier:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

model = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression(random_state=42))
])
model.fit(X_train, y_train)
```

The accuracy of the model on the training and test sets is:

```
train_accuracy = model.score(X_train, y_train)
print(f'Train accuracy: {train_accuracy:.4f}')

test_accuracy = model.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
```

```
Train accuracy: 1.0000
Test accuracy: 1.0000
```

The model achieves perfect accuracy on both sets. Let's also check how many iterations were required for convergence:

```
print('Number of iterations:', model['clf'].n_iter_[0])
```

```
Number of iterations: 8
```

The L-BFGS solver converged in just 8 iterations, highlighting its efficiency compared to the basic gradient descent implementation from Section 5.3.2, which required about 400 iterations to converge.

5.4.2 The `SGDClassifier` Class

Another way to perform logistic regression in Scikit-Learn is by using the `SGDClassifier` class, which solves classification tasks using stochastic gradient descent (SGD).

This class operates similarly to `SGDRegressor` that solves regression tasks using SGD (see Section 4.8.2.1) and shares many of the same hyperparameters. By default,

`SGDClassifier` uses the `'hinge'` loss, which corresponds to a linear SVM. To perform logistic regression, the `loss` parameter must be set to `'log_loss'`, as shown below:

```
from sklearn.linear_model import SGDClassifier

model = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', SGDClassifier(loss='log_loss', random_state=42))
])
model.fit(X_train, y_train)
```

The number of iterations needed for this classifier to converge is:

```
print('Number of iterations:', model['clf'].n_iter_)
```

```
Number of iterations: 12
```

In this case, convergence using SGD took slightly longer than the quasi-Newton method used by `LogisticRegression`.

5.5 Classification Evaluation Metrics

Classification tasks often involve **imbalanced datasets**, where the classes are not equally represented in the data [362, 455]. These datasets are typically dominated by a large number of "normal" (negative) examples, with only a small fraction of "interesting" (positive) ones. For example, medical imaging datasets for cancer detection—such as chest X-rays or mammograms—often exhibit severe class imbalance, with most images labeled as normal and only a small fraction (e.g., less than 1%) containing cancerous findings.

Class imbalance is also common in domains such as fraud detection, network intrusion detection, rare disease diagnosis, and natural disaster prediction. In all of these cases, the minority class is typically the one of greatest interest, despite its rarity.

Imbalanced datasets pose two main challenges to classification algorithms:

- **Standard metrics can be misleading.** Accuracy, in particular, often fails to reflect the true model performance in imbalanced settings. For example, in a fraud detection task with only 1% fraudulent transactions, a model that always predicts "legitimate" achieves 99% accuracy—giving a false impression of strong performance, even though it fails to detect any fraud. In such cases, identifying the minority class (e.g., fraudulent transactions) is far more important than

maximizing overall accuracy. This highlights the need for evaluation metrics that place greater emphasis on minority class performance.

- **Algorithms tend to favor the majority class.** Most classification algorithms are designed to minimize the overall error rate, which inherently biases them toward the majority class. The limited number of minority class examples makes it difficult for the model to learn their patterns, often resulting in poor performance—sometimes with minority class accuracy as low as 0–10%.

  In high-stakes domains such as healthcare, this imbalance can have serious consequences. A model that fails to detect rare conditions, despite correctly classifying the majority of healthy cases, may lead to missed diagnoses, delayed treatment, and adverse patient outcomes.

  Section 5.6 explores strategies for improving model performance on imbalanced data, including resampling techniques and cost-sensitive learning algorithms.

5.5.1 The Credit Card Fraud Detection Dataset

To illustrate the challenges posed by imbalanced datasets, we will use the Credit Card Fraud Detection dataset hosted on Kaggle. This dataset contains anonymized credit card transactions made by European cardholders over a span of two days in September 2013. It exhibits a severe class imbalance, with only 492 fraudulent transactions among 284,807 in total.

The dataset consists of 30 numerical features, with 28 of them (`V1` to `V28`) derived from applying Principal Component Analysis (PCA) to the original features (used both for anonymization and dimensionality reduction). The remaining features are `Time`, indicating the elapsed time from the first transaction, and `Amount`, representing the transaction value. The target variable `Class` is binary, with 1 indicating fraud and 0 indicating a legitimate transaction.

We begin by loading the dataset from its CSV file and examining the first and last few rows. For clarity, we display only a selected subset of the columns (see Figure 5.10):

```
import pandas as pd

df = pd.read_csv('data/creditcard.csv')
df = df[['Time', 'Amount', 'V1', 'V2', 'V3', 'V4', 'V5', 'Class']]
df
```

Let's examine the class distribution:

	Time	Amount	V1	V2	V3	V4	V5	Class
0	0.0	149.62	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0
1	0.0	2.69	1.191857	0.266151	0.166480	0.448154	0.060018	0
2	1.0	378.66	-1.358354	-1.340163	1.773209	0.379780	-0.503198	0
3	1.0	123.50	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	0
4	2.0	69.99	-1.158233	0.877737	1.548718	0.403034	-0.407193	0
...	...	...	...	...	...	...	...	...
284802	172786.0	0.77	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	0
284803	172787.0	24.79	-0.732789	-0.055080	2.035030	-0.738589	0.868229	0
284804	172788.0	67.88	1.919565	-0.301254	-3.249640	-0.557828	2.630515	0
284805	172788.0	10.00	-0.240440	0.530483	0.702510	0.689799	-0.377961	0
284806	172792.0	217.00	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	0

Figure 5.10: A preview of the credit card fraud detection dataset, showing the first and last five rows for selected columns: `Time`, `Amount`, PCA-transformed features `V1` through `V5`, and the target label `Class`.

```
df['Class'].value_counts(normalize=True)
```

```
Class
0    0.998273
1    0.001727
```

As expected, the dataset is highly imbalanced: class 0 (legitimate transactions) makes up 99.82% of the data, while class 1 (fraudulent transactions) constitutes only 0.17%. This imbalance poses a major challenge—models can easily achieve high accuracy by predicting only the majority class while failing to detect any fraud.

As a baseline, we train a logistic regression model on the dataset. First, we need to separate the features and labels:

```
X = df.drop('Class', axis=1)
y = df['Class']
```

Next, we split the dataset into training and test sets, ensuring they maintain the same class proportions using the `stratify=y` parameter:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42,
    stratify=y)
```

We then define a pipeline with `StandardScaler` and `LogisticRegression` and fit it to the training data:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

model = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression(random_state=42))
])
model.fit(X_train, y_train)
```

The model's accuracy on the training and test sets is:

```
train_accuracy = model.score(X_train, y_train)
print(f'Train accuracy: {train_accuracy:.4f}')

test_accuracy = model.score(X_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
```

```
Train accuracy: 0.9992
Test accuracy: 0.9992
```

While these accuracy values appear excellent, they are misleading in the context of such a heavily imbalanced dataset. A model can achieve high accuracy by simply predicting all transactions as legitimate. To better assess the model's performance on the minority class, we explore alternative evaluation metrics in the following subsections.

5.5.2 Confusion Matrix

A **confusion matrix** (also known as a **contingency table**) is a pivotal tool for evaluating classification models and forms the basis for many commonly used performance

metrics. The rows of the matrix represent the actual classes, while the columns represent the predicted classes.[2] Each cell (i, j) indicates the number of samples that belong to class i but were predicted as class j. The values along the diagonal represent correct classifications, whereas the off-diagonal values correspond to errors—hence the term "confusion" between classes.

In binary classification, the confusion matrix is a 2×2 table (see Table 5.3), summarizing the model's performance in terms of four components:

- **True Positives (TP)**: Positive samples correctly classified as positive.
- **True Negatives (TN)**: Negative samples correctly classified as negative.
- **False Positives (FP)**: Negative samples incorrectly classified as positive (also known as Type I error).
- **False Negatives (FN)**: Positive samples incorrectly classified as negative (also known as Type II error).

		Predicted Class	
		+	−
Actual Class	+	True Positives (TP)	False Negatives (FN)
	−	False Positives (FP)	True Negatives (TN)

Table 5.3: Confusion matrix for binary classification, illustrating the relationship between the actual and predicted class labels.

A useful way to remember these components is to note that the second letter in the component's name (P or N) refers to the model's prediction (positive or negative), while the first letter (T or F) indicates whether the prediction was correct (true) or incorrect (false). For example, FP refers to cases where the model predicted positive, but the prediction was incorrect (i.e., the actual label was negative).

In Scikit-Learn, the confusion matrix for a trained model can be computed using the `confusion_matrix` function, which takes the true labels and the model's predicted labels as input. For example, we can compute the confusion matrix for our logistic regression model on the test set as follows:

[2]Some sources use the opposite convention, where rows represent predicted classes and columns represent actual classes. Always check the labeling when interpreting a confusion matrix.

```
from sklearn.metrics import confusion_matrix

y_test_pred = model.predict(X_test)
conf_mat = confusion_matrix(y_test, y_test_pred)
print(conf_mat)
```

The output is:

```
[[71065    14]
 [   46    77]]
```

In Scikit-Learn's confusion matrix, the first row corresponds to the negative class (true negatives and false positives), while the second row corresponds to the positive class (false negatives and true positives). In this case, the matrix shows: 71,065 true negatives (TN), 14 false positives (FP), 46 false negatives (FN), and 77 true positives (TP).

The overall accuracy of the model can be computed from the confusion matrix as:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{77 + 71{,}065}{77 + 71{,}065 + 14 + 46} = 99.92\%, \tag{5.25}$$

which confirms the result obtained earlier using Scikit-Learn's `score` method. Despite this high accuracy, the confusion matrix reveals that the model correctly identifies only $77/(46 + 77) = 62.6\%$ of the fraudulent transactions in the test set. This value, known as **recall** (see Section 5.5.3), highlights a significant shortcoming in the model's ability to detect the minority class.

For a better visualization of the confusion (especially in multi-class settings), we can use the `ConfusionMatrixDisplay` class:

```
from sklearn.metrics import ConfusionMatrixDisplay

disp = ConfusionMatrixDisplay(confusion_matrix=conf_mat)
disp.plot(cmap='Blues')
```

Figure 5.11 provides a graphical representation of the confusion matrix, where color intensity reflects the number of samples in each cell.

5.5.3 Precision and Recall

When working with imbalanced datasets, precision and recall are key evaluation metrics, as they provide a more meaningful assessment than accuracy alone.

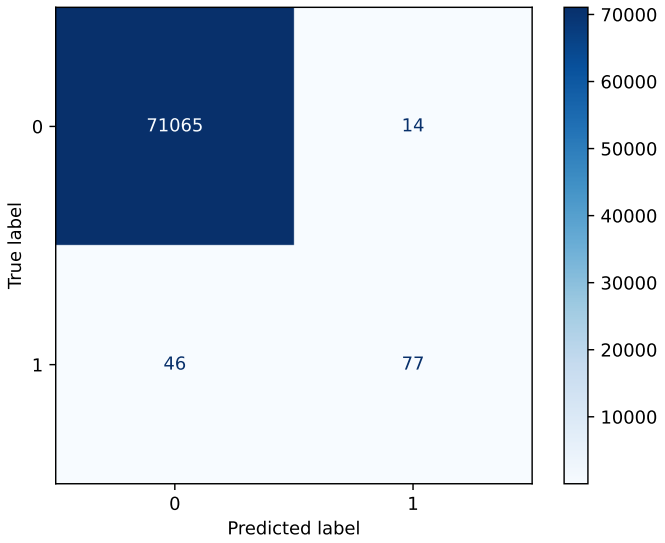

Figure 5.11: Confusion matrix for the logistic regression model on the credit card fraud detection test set.

Precision measures how many of the instances predicted as positive are actually positive. It is defined as the ratio of true positives to all predicted positives (i.e., true positives plus false positives):

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{5.26}$$

Recall (also known as **sensitivity** or **true positive rate**) measures how many of the actual positive instances were correctly identified by the model. It is defined as the ratio of true positives to all actual positives (i.e., true positives plus false negatives):

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{5.27}$$

In many classification tasks, there is an inherent tradeoff between precision and recall. A model that favors precision tries to avoid false positives by being cautious and only predicting positive when it is very confident. In contrast, a model that favors recall tries to avoid false negatives, often by predicting positive more aggressively to ensure that most actual positives are captured.

At the extremes of the tradeoff, a model that classifies every instance as positive would achieve perfect recall but very low precision, since most of its positive predictions

would be incorrect. Conversely, a model that classifies every instance as negative would have perfect precision (since there are no false positives) but zero recall.

The relative importance of precision versus recall depends on the application. For example, in spam detection, precision is typically more important, since false positives (legitimate emails flagged as spam) are less desirable than false negatives (spam emails that remain in the inbox). On the other hand, in medical diagnostics, recall is often prioritized, as failing to detect a disease (false negative) can be far more harmful than mistakenly flagging a healthy patient (false positive).

Precision and recall can be computed in Scikit-Learn using the functions `precision_score` and `recall_score`, respectively:

```
from sklearn.metrics import precision_score, recall_score

precision = precision_score(y_test, y_test_pred)
print(f'Precision: {precision:.4f}')

recall = recall_score(y_test, y_test_pred)
print(f'Recall: {recall:.4f}')
```

```
Precision: 0.8462
Recall: 0.6260
```

These results indicate that when the model predicts a transaction as fraudulent, it is correct 84.62% of the time (precision). However, it identifies only 62.6% of the actual fraudulent transactions (recall), highlighting the model's difficulty in detecting all positive cases.

5.5.4 F-Scores

Precision and recall can be combined into a single metric known as the **F1 score**, which is defined as their harmonic mean:

$$F_1 = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{5.28}$$

The harmonic mean penalizes large discrepancies between precision and recall, ensuring that both metrics must be high for the F1 score to be high. This makes the F1 score particularly useful when precision and recall are equally important. In Scikit-Learn, the F1 score can be computed using the `f1_score` function:

```
from sklearn.metrics import f1_score

f1 = f1_score(y_test, y_test_pred)
print(f'F1 score: {f1:.4f}')
```

```
F1 score: 0.7196
```

When precision and recall are not equally important, the more general F_β score introduces a weighting factor β to control the relative importance of recall with respect to precision:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}. \tag{5.29}$$

Here, $\beta > 1$ gives more weight to recall, while $\beta < 1$ gives more weight to precision. To compute the F_β score in Scikit-Learn, you can use the `fbeta_score` function.

Although these metrics are typically reported on the test set, computing them on the training set as well can provide insight into potential overfitting.

5.5.5 Thresholding

In classification models that produce a continuous output (such as a probability or score), the tradeoff between precision and recall can be adjusted by modifying the **decision threshold**. This threshold determines the value above which a sample is classified as belonging to the positive class.

For example, the `LogisticRegression` class provides a `predict_proba` method that returns the predicted probabilities for each class. By default, a threshold of 0.5 is applied to these probabilities to classify a sample as either positive or negative. Adjusting this threshold allows us to balance precision and recall according to the needs of the application.

In fraud detection, for instance, recall is often prioritized because missing a positive instance (a fraudulent transaction) can have severe consequences. To improve the recall of our model, we can lower the threshold to 0.3:

```
threshold = 0.3 # Lower the threshold to increase recall
probs = model.predict_proba(X_test)[:, 1] # Probabilities for the positive
    class
y_test_pred = (probs > threshold).astype(int)

precision = precision_score(y_test, y_test_pred)
print(f'Precision: {precision:.4f}')
```

```
recall = recall_score(y_test, y_test_pred)
print(f'Recall: {recall:.4f}')
```

```
Precision: 0.7636
Recall: 0.6829
```

Lowering the threshold increases recall from 62.6% to 68.3%, but this comes at the cost of reducing precision from 84.6% to 76.4%.

5.5.6 Precision–Recall Curve

A precision–recall (PR) curve is a useful tool for visualizing the tradeoff between precision and recall [117]. It can also be used to evaluate a classifier's overall performance and to compare different classifiers across all possible decision thresholds.

In Scikit-Learn, the `precision_recall_curve` function computes precision and recall values for every possible threshold and returns them along with the corresponding thresholds. These values can then be used to plot the precision–recall curve:

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train, probs)
plt.plot(recalls, precisions)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
```

Figure 5.12 shows the precision–recall curve for the logistic regression classifier. The plot demonstrates the inherent tradeoff between precision and recall: as recall increases, precision tends to decrease. This occurs because achieving higher recall requires lowering the classification threshold, which leads to more positive predictions, including more false positives. Notably, there is a sharp drop in precision around a recall of 0.8, indicating that any further gains in recall beyond this point result in a significant loss in precision. This highlights the difficulty in maintaining high precision while maximizing recall.

When comparing two classifiers, the one whose PR curve consistently dominates (i.e., lies above and to the right of the other) is generally considered superior. This means that, for the same recall, the superior classifier achieves higher precision—or equivalently, for the same precision, it achieves higher recall.

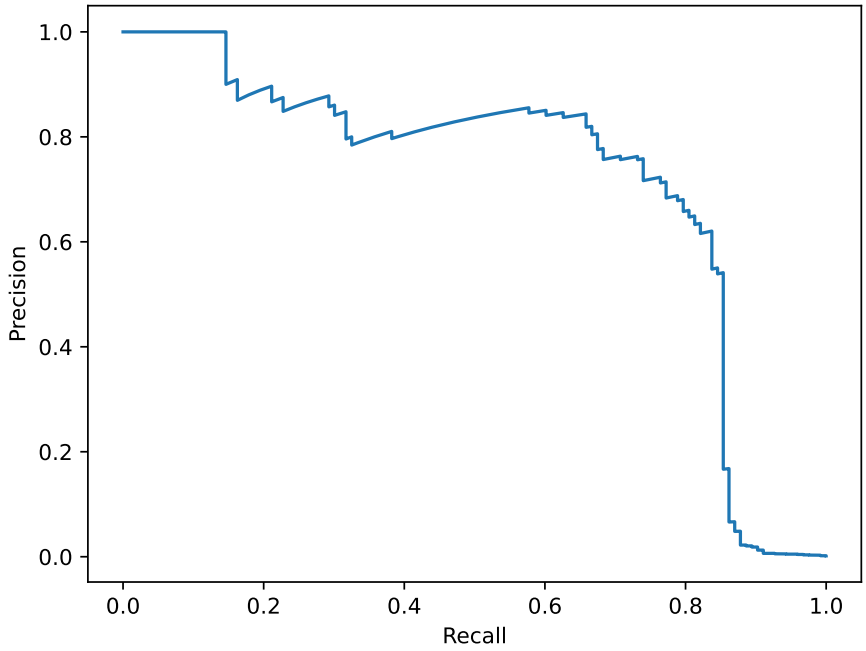

Figure 5.12: Precision–recall curve for the logistic regression classifier on the credit card fraud detection dataset. As recall increases, precision generally decreases, reflecting the challenge of maintaining high precision while achieving high recall.

A related plot shows how precision and recall change as a function of the decision threshold. Such a plot provides insight into how adjusting the threshold affects performance and assists in selecting an appropriate threshold based on the application's priorities. The following code generates this plot:

```
plt.plot(thresholds, precisions[:-1], label='Precision')
plt.plot(thresholds, recalls[:-1], label='Recall')
plt.xlabel('Threshold')
plt.ylabel('Metric')
plt.legend()
```

The slicing `[:-1]` is used because `precision_recall_curve` returns one more precision–recall pair than there are thresholds. The final pair corresponds to a theoretical threshold of infinity, which has no associated threshold value.

Figure 5.13 shows how both precision and recall vary with the decision threshold. As the threshold increases, recall decreases because the classifier becomes more selective,

leading to fewer positive predictions. Conversely, precision tends to increase with a higher threshold, although it may fluctuate depending on the balance between true positives and false positives at specific thresholds.

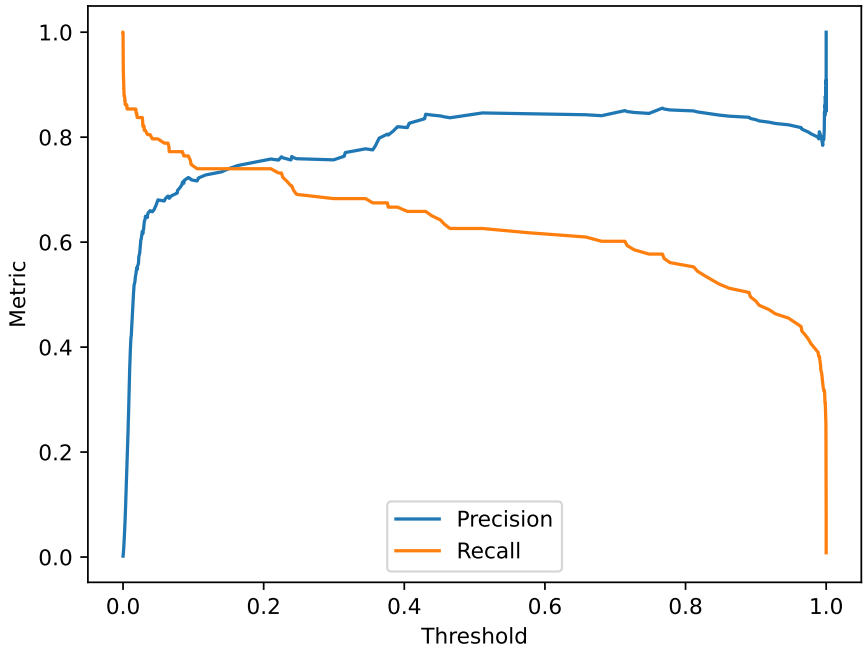

Figure 5.13: Precision and recall as functions of the decision threshold for the logistic regression classifier. As the threshold increases, recall decreases while precision tends to increase. The optimal threshold depends on the specific application and whether higher precision or higher recall is prioritized.

By examining both the PR curve and the precision/recall-vs.-threshold plot, we gain a more complete understanding of how the classifier behaves and can adjust the threshold to achieve the desired balance between precision and recall for a given application.

5.5.7 ROC Curve

The **receiver operating characteristic (ROC)** curve is another valuable tool for evaluating a classifier's performance across different decision thresholds and for comparing multiple classifiers [255].

The x-axis of an ROC curve corresponds to the **false positive rate (FPR)**, which

measures the proportion of negative samples incorrectly classified as positive:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}. \tag{5.30}$$

The y-axis corresponds to the **true positive rate (TPR)** (also known as **recall**), which measures the proportion of positive samples correctly classified as positive:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{5.31}$$

Similar to the tradeoff between precision and recall, there is a tradeoff between FPR and TPR. Models that are more conservative in predicting the positive class produce fewer false positives but may miss more true positives, while more aggressive models capture more true positives at the cost of a higher false positive rate. The ROC curve visualizes this tradeoff by plotting the TPR against the FPR as the decision threshold varies. Because increasing the threshold can never increase the TPR without also increasing (or maintaining) the FPR, the ROC curve is monotonically increasing.

Several key points and lines in the ROC space are worth noting (see Figure 5.14):

- The bottom-left point (0, 0) corresponds to a model that always predicts the negative class, resulting in no false positives but also no true positives.
- The top-right point (1, 1) corresponds to a model that always predicts the positive class, resulting in both the FPR and the TPR being equal to 1.
- The top-left point (0, 1) corresponds to a perfect classifier, with no false positives and all true positives correctly identified.
- The diagonal line $y = x$ represents a random guesser. For example, a model that guesses the positive class 50% of the time will lie at the point (0.5, 0.5). As the positive-guess rate increases, both TPR and FPR rise together along the diagonal.
- In general, one point in ROC space represents a better classifier than another if it lies further to the northwest (i.e., has a higher TPR and a lower FPR).

Figure 5.14 shows an example of an ROC curve with five classifiers (A, B, C, D, E) plotted according to their true positive rates and false positive rates. The classifiers in the figure exhibit different performance characteristics:

- Classifier A is more conservative than B, prioritizing a lower false positive rate at the expense of missing more true positives.

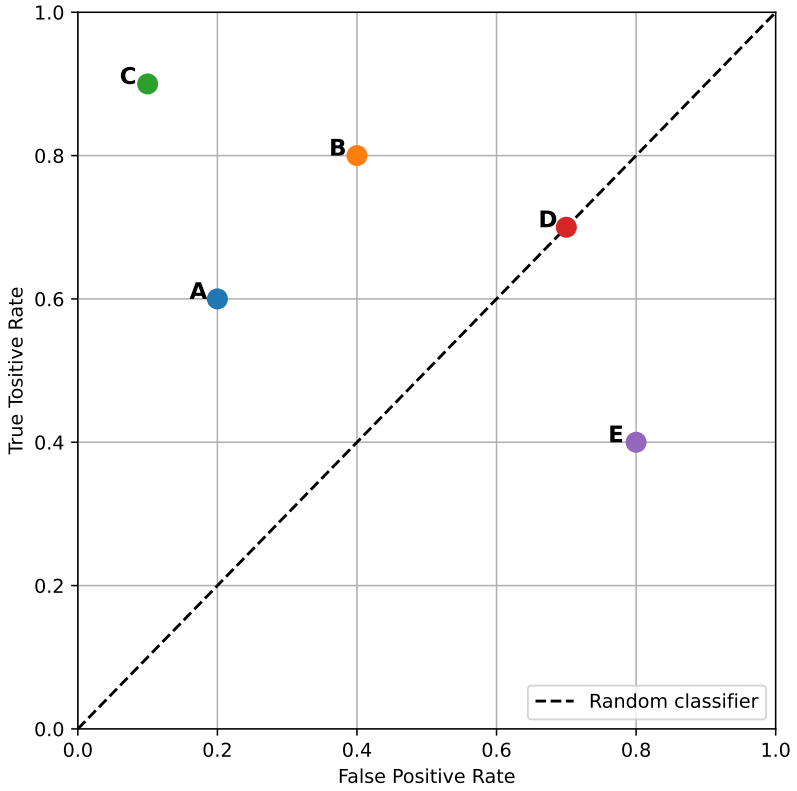

Figure 5.14: ROC curve illustrating five classifiers (A, B, C, D, E). Classifier C outperforms A and B by having both lower FPR and higher TPR. Classifier D demonstrates performance akin to random guessing. Classifier E, which is in the lower-right triangle, performs worse than random but can be negated to achieve a point like B.

- Classifier C outperforms both A and B, achieving a lower false positive rate and a higher true positive rate.
- Classifier D behaves like a random classifier that predicts the positive class 70% of the time, resulting in the point (0.7, 0.7).
- Classifier E is located at the point (0.8, 0.4) in the lower-right triangle of the ROC curve, indicating performance worse than random guessing. However, inverting its predictions on every instance would reflect this point across the diagonal, moving it to (0.8, 0.4) and yielding performance identical to that of Classifier B.

An efficient algorithm for generating the ROC curve is shown in Algorithm 5.1.

This algorithm exploits the monotonicity of thresholded classifications: any example classified as positive at a given threshold remains classified as positive for all lower thresholds. Therefore, we can sort the examples by their predicted probabilities in descending order and iterate through this sorted list, updating the true positive (TP) and false positive (FP) counts as we go. For each positive example, TP is incremented; for each negative example, FP is incremented. After processing each example, the corresponding point (FPR, TPR) is added to the list of ROC points. These points can then be used to plot the ROC curve.

Algorithm 5.1 Build ROC Curve

Input:

L: set of labeled examples
$f(i)$: classifier's predicted score for example i
P: number of positive examples
N: number of negative examples

Output:

R: list of ROC points sorted by false positive rate

$L_{\text{sorted}} \leftarrow L$ sorted by decreasing f scores
$FP \leftarrow 0, TP \leftarrow 0$
$R \leftarrow []$
for $i = 1$ **to** $|L_{\text{sorted}}|$ **do**
    Add $\left(\frac{FP}{N}, \frac{TP}{P}\right)$ to R
    if $L_{\text{sorted}}[i]$ is a positive example **then**
        $TP \leftarrow TP + 1$
    else
        $FP \leftarrow FP + 1$
Add $\left(\frac{FP}{N}, \frac{TP}{P}\right)$ to R ▷ Final point (1, 1)
return R

In Scikit-Learn, the ROC curve can be generated using the `roc_curve` function:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_test, probs)

plt.plot(fpr, tpr, linewidth=2)
plt.plot([0, 1], [0, 1], 'k--') # Diagonal reference line
```

```
plt.axis([0, 1, 0, 1])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
```

Figure 5.15 shows the ROC curve of the logistic regression classifier on the credit card fraud dataset. The curve starts at (0, 0) and ends at (1, 1), illustrating how the true positive rate and false positive rate vary as the decision threshold changes.

The fact that the ROC curve lies well above the diagonal indicates that the logistic regression model performs significantly better than random guessing. The steep initial rise shows that the classifier is able to identify many true positives while maintaining a low false positive rate—a particularly desirable property in imbalanced settings where correctly identifying the minority class is critical.

However, a low FPR does not necessarily imply high precision. For example, consider a case in which the model is very conservative, predicting the positive class only rarely. In this case, there will be few false positives overall (leading to a low FPR), but those few positive predictions may still be mostly incorrect. As a result, the precision remains low even though the FPR is low. This highlights a limitation of ROC curves: on highly imbalanced datasets, they can present an overly optimistic view of a model's performance.

The **area under the ROC curve (AUC)** provides an overall measure of a classifier's performance across all decision thresholds by calculating the area under the ROC curve [353, 100]. It enables performance comparison between classifiers using a single scalar value. Since the ROC curve lies within a unit square, the AUC ranges from 0 to 1: an AUC of 1 represents a perfect classifier, while an AUC of 0.5 corresponds to random guessing.

Statistically, the AUC represents the probability that the classifier assigns a higher score to a randomly chosen positive instance than to a randomly chosen negative instance. Thus, a classifier with a higher AUC is more likely, on average, to rank positive instances above negative ones.

In Scikit-Learn, the AUC can be computed using the `roc_auc_score` function:

```
from sklearn.metrics import roc_auc_score

auc = roc_auc_score(y_test, probs)
print(f'AUC: {auc:.4f}')
```

```
AUC: 0.9587
```

This result indicates that the logistic regression classifier performs well at distin-

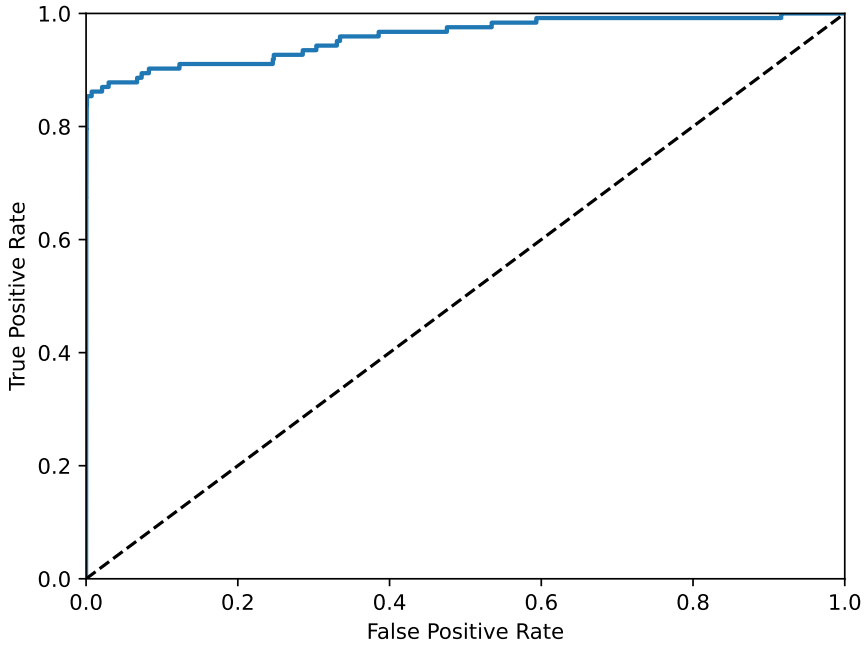

Figure 5.15: ROC curve for the logistic regression classifier on the credit card fraud detection dataset. The curve shows how the true positive rate (TPR) varies with the false positive rate (FPR) across different thresholds. The steep rise at the beginning suggests that the model captures many true positives while generating relatively few false positives.

guishing fraudulent from non-fraudulent transactions, with a 95.87% probability of ranking a randomly chosen positive instance higher than a randomly chosen negative instance. However, a high AUC does not necessarily imply high recall (or high precision) at a particular threshold.

To summarize, the main advantages of ROC curves for classifier evaluation are:

- ROC curves visualize classifier performance across all decision thresholds, allowing for a more holistic evaluation than metrics calculated at a single threshold. Additionally, the area under the ROC curve (AUC) summarizes performance across all thresholds into a single scalar value, making it easy to compare classifiers.
- Unlike precision–recall curves, ROC curves are less affected by changes in the class distribution. This is because both TPR and FPR are calculated independently of the class distribution: TPR is the proportion of true positives among all actual

positives, while FPR is the proportion of false positives among all actual negatives. As a result, ROC curves tend to remain stable even when the prevalence of the classes changes.

However, ROC curves can be misleading in highly imbalanced datasets due to how the FPR is computed. When the number of negative instances is very large, even a substantial number of false positives may produce a small FPR. In contrast, precision—defined as the proportion of true positives among all positive predictions—is more sensitive to false positives. As a result, precision–recall (PR) curves are often more informative than ROC curves on highly imbalanced datasets [195].

Overall, there is no single perfect metric for evaluating classification models, as each metric reduces the confusion matrix to a single value and inevitably loses information. A more comprehensive assessment is achieved by considering multiple metrics—such as accuracy, precision, recall, F1 score, and AUC—which together capture different aspects of model performance, including its behavior across decision thresholds and under class imbalance.

5.6 Learning from Imbalanced Data

Imbalanced datasets, in which one class (the minority class) is significantly underrepresented, pose a major challenge for machine learning algorithms [404, 853, 362]. Models trained on such data often become biased toward the majority class, as they have far fewer examples from which to learn the characteristics of the minority class. This imbalance can severely hinder the model's ability to correctly identify minority class instances.

Several strategies have been developed to address class imbalance:

- **Sampling techniques**: These techniques modify the training data in order to balance the class distribution:
  - **Oversampling the minority class**: The representation of the minority class is increased by duplicating existing samples or generating synthetic ones. While simple random oversampling may cause overfitting [569], a more effective approach is SMOTE [145], which synthesizes new samples by interpolating between a minority instance and its k nearest neighbors (see Section 5.6.1).

  - **Undersampling the majority class**: The number of majority class instances is reduced to achieve a balanced dataset [461]. While this can improve performance on the minority class, it risks losing useful information about the majority class, potentially degrading the overall model performance.
  - **Hybrid approaches**: Combining oversampling and undersampling can mitigate the drawbacks of each method, improving the minority class representation without significantly reducing the data available from the majority class [47].

- **Cost-sensitive learning**: Instead of altering the dataset, cost-sensitive methods modify the loss function to penalize errors on minority class instances more heavily (see Section 5.6.2). This approach is especially useful when resampling is impractical or undesirable due to risks of information loss or overfitting, as it directly encourages the model to focus on the minority class.

- **Ensemble methods**: Ensemble techniques combine multiple models to improve performance (see Chapter 9) and are often very effective for imbalanced datasets [288]. By combining predictions from models trained on different data subsets or with different class weights, ensembles mitigate the bias toward the majority class and improve generalization.

- **Anomaly detection techniques**: When the minority class is extremely rare, the classification task can be reframed as an anomaly detection problem, where the goal is to identify instances that deviate significantly from the majority class (this topic is covered in Volume II).

5.6.1 Synthetic Minority Oversampling (SMOTE)

SMOTE (Synthetic Minority Oversampling Technique) [145] is a widely used method for addressing class imbalance by generating synthetic examples of the minority class rather than simply duplicating existing ones. By creating new samples through interpolation, SMOTE increases the diversity of the minority class and reduces the risk of overfitting typically associated with naive oversampling.

The algorithm proceeds as follows:

1. Randomly select a minority class instance $\mathbf{x}_i$.

2. Identify the k nearest neighbors of $\mathbf{x}_i$ within the minority class (typically $k = 5$).

3. Randomly select one of these neighbors, denoted $\mathbf{x}_p$.

4. Generate a synthetic sample $\mathbf{x}_s$ at a random point along the line segment between $\mathbf{x}_i$ and $\mathbf{x}_p$ (see Figure 5.16). For each feature j, compute the synthetic feature value as follows:

   (a) $d \leftarrow x_{pj} - x_{ij}$

   (b) $r \leftarrow$ random number between 0 and 1

   (c) $x_{sj} \leftarrow x_{ij} + r \cdot d$

   This procedure linearly interpolates between the original instance and its neighbor, generating each feature of the synthetic sample at a random position along the corresponding dimension. This ensures diversity among synthetic samples while preserving the local structure of the minority class.

5. Repeat this process until the desired number of minority samples is reached.

The Python library `imbalanced-learn` provides an implementation of SMOTE, along with many other algorithms and tools for handling imbalanced datasets. It can be easily installed using `pip install imbalanced-learn`.

Let's use the SMOTE class from this library to oversample the fraudulent class in our fraud detection dataset. The key parameters of this class include:

- `sampling_strategy`: Controls the resampling ratio, defined as the desired ratio of the number of minority-class samples after resampling to the number of majority-class samples. For example, a value of 0.1 means that the minority class will be resampled to 10% of the size of the majority class. The default value `'auto'` balances the classes so they have the same number of samples. This parameter can strongly affect the balance between recall and precision, so it should be tuned according to the application's objectives.

- `k_neighbors`: Specifies how many nearest neighbors are considered as reference points when creating synthetic samples. The default is 5.

For example, the following code increases the proportion of fraudulent transactions to 1% of the legitimate transactions (up from about 0.17% in the original dataset), improving minority representation without creating an unrealistically balanced dataset, which could distort the true class distribution and lead the model to assume that fraud is much more common than it actually is:

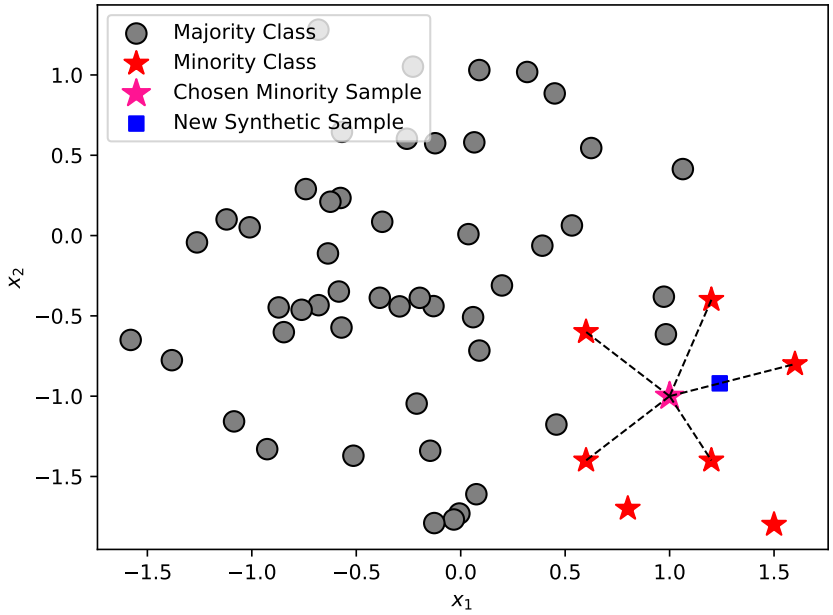

Figure 5.16: Illustration of the SMOTE (Synthetic Minority Oversampling Technique) algorithm. Majority class instances are represented by circles, and minority class instances by stars. The chosen minority instance is highlighted using a lighter star, and its nearest neighbors are connected by dashed lines. A new synthetic sample (represented by a square) is generated at a random position between the selected instance and one of its neighbors.

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42, sampling_strategy=0.01)
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)
```

It is crucial to apply SMOTE only to the training set to ensure that model evaluation on the test set remains valid. Applying it to the test set could introduce synthetic data, biasing performance metrics and undermining the assessment of generalization to truly unseen examples.

After applying SMOTE, we can inspect the size of the training set:

```
print(f'Before SMOTE: {X_train.shape}')
```

```
print(f'After SMOTE: {X_train_res.shape}')
```

```
Before SMOTE: (213605, 30)
After SMOTE: (215368, 30)
```

SMOTE added 1,763 synthetic samples to the minority class. The new class distribution is:

```
y_res.value_counts(normalize=True)
```

```
Class
0    0.990101
1    0.009899
```

As expected, the minority class now makes up about 1% of the majority class. We can now retrain our logistic regression model on the resampled training set:

```
model.fit(X_train_res, y_train_res)
```

The performance metrics of the model on the test set are:

```
y_test_pred = model.predict(X_test)

print(f'Precision: {precision_score(y_test, y_test_pred):.4f}')
print(f'Recall: {recall_score(y_test, y_test_pred):.4f}')
print(f'F1 score: {f1_score(y_test, y_test_pred):.4f}')
```

```
Precision: 0.7500
Recall: 0.8049
F1 score: 0.7765
```

Recall has improved significantly, from 62.6% to 80.49%, while precision decreased from 84.62% to 75%. Overall, the F1 score increased from 71.96% to 77.65%, indicating a better balance between precision and recall.

For comparison, we also evaluate a simple random oversampling approach using the `RandomOverSampler` class also from the `imbalanced-learn` library:

```
from imblearn.over_sampling import RandomOverSampler

ros = RandomOverSampler(random_state=42, sampling_strategy=0.01)
X_train_res, y_train_res = ros.fit_resample(X_train, y_train)

model.fit(X_train_res, y_train_res)
```

```
y_test_pred = model.predict(X_test)
print(f'Precision: {precision_score(y_test, y_test_pred):.4f}')
print(f'Recall: {recall_score(y_test, y_test_pred):.4f}')
print(f'F1 score: {f1_score(y_test, y_test_pred):.4f}')
```

The results are:

```
Precision: 0.7481
Recall: 0.7967
F1 score: 0.7717
```

In this case, random oversampling yields slightly worse results than SMOTE. The difference is small, partly because logistic regression is a linear model and thus cannot fully exploit the diversity introduced by SMOTE's synthetic samples. Since it can only learn linear decision boundaries, the additional synthetic points do not lead to a more expressive class separation.

While SMOTE can be a powerful tool for addressing class imbalance, its effectiveness depends heavily on the specific dataset and the problem context. The synthetic samples it generates may not accurately reflect the true underlying distribution of the minority class, potentially leading to overlap with the majority class or introducing noise.

To address these issues, several extensions of SMOTE have been proposed, many of which are implemented in the `imbalanced-learn` library:

- **Borderline SMOTE**: Focuses on minority samples near the decision boundary (the borderline) [350]. It generates synthetic samples in directions where the nearest neighbors belong to the majority class, reinforcing the minority presence in ambiguous areas and helping the model learn a more discriminative boundary. Implemented by the `BorderlineSMOTE` class.
- **SMOTE-ENN (SMOTE + Edited Nearest Neighbors)**: This hybrid method combines SMOTE with the Edited Nearest Neighbor (ENN) cleaning technique [47]. After generating the synthetic samples, ENN removes instances that differ from the majority of their neighbors, thereby reducing noise and improving the decision boundary. Implemented by the `SMOTEENN` class.
- **SMOTE-Tomek**: Combines SMOTE with the removal of Tomek links [47]. Tomek links are pairs of samples from opposite classes that are each other's nearest neighbors, often indicating borderline cases or noise. Removing these

pairs sharpens the decision boundary and improves the effectiveness of SMOTE. Implemented by the SMOTETomek class.

- **K-Means SMOTE**: Incorporates k-means clustering into the oversampling process [237]. Synthetic samples are generated within minority-class clusters, making the oversampling process more adaptive to complex and heterogeneous class distributions. Implemented by the `KMeansSMOTE` class.

- **ADASYN (Adaptive Synthetic Sampling)**: Generates more synthetic samples in regions where the minority class is sparse [361]. By adapting the number of synthetic samples to local data density, ADASYN focuses oversampling on harder-to-learn areas of the feature space. Implemented by the `ADASYN` class.

SMOTE remains an active area of research, with ongoing work aimed at improving its effectiveness and scalability [86, 45], integrating it with deep learning models [187], and developing adaptive sampling techniques that respond to real-time model performance [227]. For a comprehensive review of recent progress and open challenges, see Fernández et al. [258].

5.6.2 Cost-Sensitive Learning

Cost-sensitive learning enables models to assign different penalties to different types of errors, allowing mistakes on minority-class instances to be weighted more heavily than those on majority-class instances [249, 561].

Central to this approach is the **cost matrix**, which specifies the cost incurred for each combination of true and predicted classes. Its structure resembles a confusion matrix: rows correspond to actual classes and columns to predicted classes. The entry $C(i, j)$ represents the cost of predicting class j when the true class is i.

Table 5.4 illustrates a cost matrix for a binary classification problem. Typically, correct classifications incur no cost (i.e., $C(0, 0) = C(1, 1) = 0$), and the cost of misclassifying a minority-class instance (false negative) is higher than misclassifying a majority-class instance (false positive), i.e., $C(1, 0) > C(0, 1)$.

In some applications, the entries of the cost matrix are not fixed but depend on the specific instance. For example, in credit card fraud detection, the cost of approving a fraudulent transaction depends on the size of the transaction, as the bank must cover the fraudulent charges. Table 5.5 shows a cost matrix for this scenario, where x denotes the transaction amount in dollars. In this example, approving a fraudulent transaction costs the bank x, while refusing a legitimate transaction incurs a fixed customer dissatisfaction cost. The entry $-0.01x$ in the "Approve/Legitimate" cell represents a small

		Predicted Class	
		$-$	$+$
Actual Class	$-$	$C(0,0)$	$C(0,1)$
	$+$	$C(1,0)$	$C(1,1)$

Table 5.4: Cost matrix for a binary classification problem

benefit (1% of the transaction value) from approving a legitimate transaction, corresponding to the bank's profit or long-term customer satisfaction.

	Legitimate	Fraudulent
Approve	$-0.01x$	x
Refuse	\$10	0

Table 5.5: Example of a cost matrix for credit card transactions

In cost-sensitive learning, the optimal classification for an instance $\mathbf{x}$ is the one that minimizes the expected misclassification cost, also known as the **Bayes conditional risk**. Specifically, the expected cost of predicting class j is given by

$$R(j|\mathbf{x}) = \sum_i P(i|\mathbf{x})C(i,j), \tag{5.32}$$

where $P(i|\mathbf{x})$ is the probability that $\mathbf{x}$ belongs to class i, and $C(i,j)$ is the cost of predicting class j when the true class is i.

There are several ways to incorporate cost-sensitive learning into machine learning algorithms:

- **Modifying the objective function**: Many algorithms can incorporate misclassification costs directly into their loss function [906, 463, 14]. For example, in logistic regression, the log loss can be weighted according to the cost matrix so that errors on minority-class instances receive higher penalties.
- **Adjusting class weights**: Some algorithms allow assigning different weights to classes, which can affect various aspects of the learning process such as split decisions in decision trees [508] or sample selection in random forests [146]. Assigning a higher weight to the minority class encourages the model to focus more on correctly classifying these instances.

- **Adapting learning rates**: In gradient-based optimization, the learning rate can be adapted based on misclassification costs [463]. A higher learning rate for minority-class instances encourages the model to update its parameters more aggressively on these instances.
- **Post-processing decision thresholds**: After training, decision thresholds can be shifted to reflect misclassification costs. For instance, in binary classification, the threshold for predicting the positive class can be lowered when the cost of misclassifying a minority-class instance is higher.

In Scikit-Learn, many algorithms support class weighting through the `class_weight` parameter, which can influence the objective function or other parts of the learning process, depending on the implementation. This parameter accepts either a dictionary mapping class labels to weights, or the string `'balanced'`, which automatically sets weights inversely proportional so that each class is equally represented. The default value `None` means that no class weighting is applied, so all samples are treated equally.

For example, in our logistic regression model, we can assign a higher weight to the minority class (e.g., 0.8) as follows:

```
model = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression(class_weight={0: 0.2, 1: 0.8},
       random_state=42))
])
model.fit(X_train, y_train)
```

The performance of the model is:

```
y_test_pred = model.predict(X_test)

print(f'Precision: {precision_score(y_test, y_test_pred):.4f}')
print(f'Recall: {recall_score(y_test, y_test_pred):.4f}')
print(f'F1 score: {f1_score(y_test, y_test_pred):.4f}')
```

```
Precision: 0.7520
Recall: 0.7642
F1 score: 0.7581
```

The F1 score of 75.81% is higher than that of the baseline model with equal class weights (71.96%), but slightly lower than the score achieved with SMOTE (77.65%).

Additionally, many algorithms in Scikit-Learn support assigning weights to individual samples using the `sample_weight` argument in the `fit` method:

```
sample_weights = [1, 1, 5, 1, 3, 2] # Example of individual weights
model.fit(X_train, y_train, sample_weight=sample_weights)
```

This can be useful when the cost depends on specific instances, allowing you to give more influence to costly or difficult examples.

SMOTE and class-weight adjustment can also be combined: SMOTE increases the representation of the minority class, while class weighting emphasizes these instances during training. Together, they can leverage the diversity of synthetic samples and the cost-awareness of the learning algorithm to potentially achieve better results.

5.7 Multi-Class Extensions for Binary Classifiers

Multi-class classification involves assigning instances to one of several classes, rather than just two. While some machine learning algorithms, such as decision trees and neural networks, natively support multi-class classification, others like logistic regression and support vector machines (SVMs) are designed for binary classification and must be adapted to handle multiple classes. Two general strategies for extending binary classifiers to multi-class problems are:

- **One-vs-Rest (OvR)** (also known as **One-vs-All (OvA)**): This method trains a separate binary classifier for each class, treating that class as the positive class and all other classes combined as the negative class (see Figure 5.17). At prediction time, all classifiers are evaluated and the class with the highest score (e.g., predicted probability) is chosen. If the algorithm only outputs hard labels, we choose one of the classes whose OvR classifier predicts positive, breaking ties with a simple rule (e.g., choosing the class with the smallest index).

- **One-vs-One (OvO)**: In this strategy, a separate binary classifier is trained for every pair of classes (see Figure 5.18). For n classes, this results in $\frac{n(n-1)}{2}$ classifiers. During prediction, each classifier votes for one of its two classes, and the class with the most votes overall is selected. If there is a tie, the decision scores (when available) are aggregated across classifiers to break the tie.

Both OvR and OvO have their strengths and limitations. OvR is computationally more efficient since it trains only n classifiers, and it can make use of prediction scores to select the class with the highest confidence. However, it may suffer from class imbalance: each binary classifier must separate one class from all the others, leading to a large imbalance between positive and negative examples.

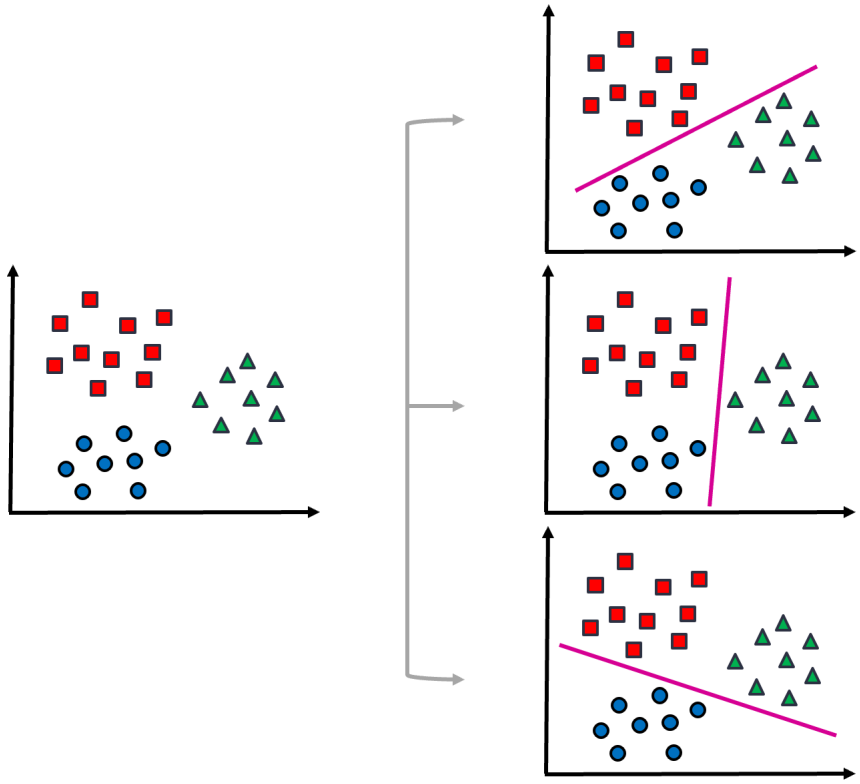

Figure 5.17: Illustration of the one-vs-rest (OvR) strategy for multi-class classification. The left panel shows a dataset with three classes, represented by squares, triangles, and circles. In the right panels, a separate classifier is trained for each class to distinguish it from all others. The final prediction is made by selecting the class with the highest score among all classifiers.

Conversely, OvO requires a quadratic number of classifiers, which can be costly to train and evaluate when there are many classes. On the other hand, each classifier only needs to distinguish between two classes, making the subproblems simpler and the individual classifiers faster to train. In practice, OvO often achieves better performance, but at the cost of increased training and prediction time.

Scikit-Learn provides two classes that implement these strategies:

- `OneVsRestClassifier` for the one-vs-rest strategy.
- `OneVsOneClassifier` for the one-vs-one strategy.

These classes can wrap any binary classifier to enable multi-class classification. For

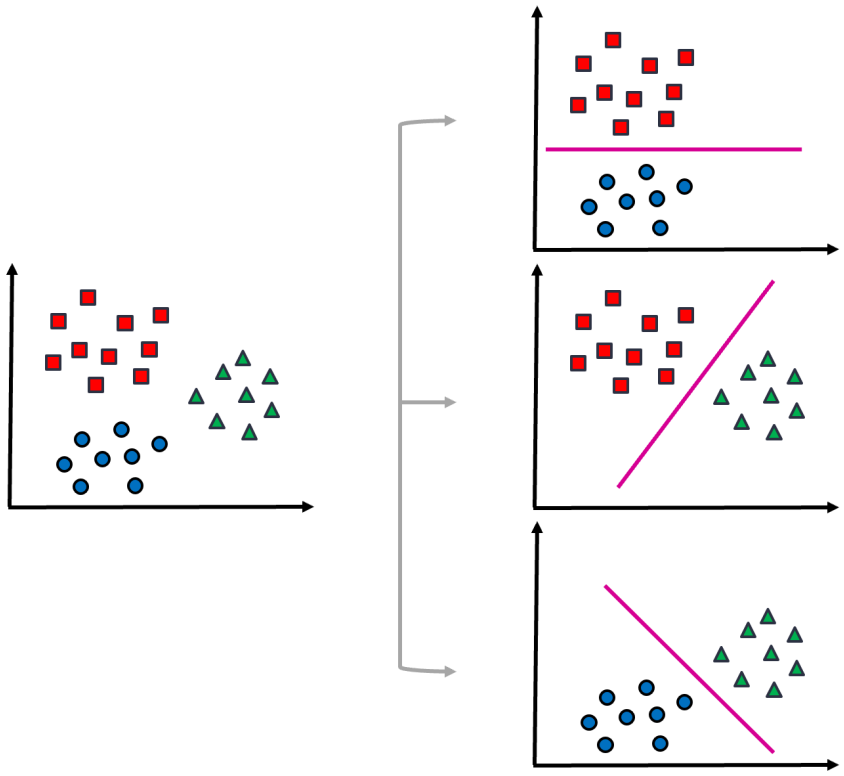

Figure 5.18: Illustration of the one-vs-one (OvO) strategy for multi-class classification. The left panel shows a dataset with three classes: squares, triangles, and circles. The right panels illustrate how a separate classifier is trained for each pair of classes. The final prediction is based on the total number of votes each class receives.

example, you can use logistic regression with the one-vs-rest strategy as follows:

```
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

ovr_classifier = OneVsRestClassifier(LogisticRegression())
```

In addition, some estimators (such as SVMs) automatically select an appropriate multi-class strategy when their `fit` method is called on data with more than two classes.

5.8 Multinomial Logistic Regression

Multinomial logistic regression (also known as **softmax regression**) is an extension of logistic regression for multi-class classification problems [565, 384]. It models the probability of each class as a function of the input features, subject to the constraint that all predicted probabilities are nonnegative and sum to one. For example, in an image classification task, a multinomial logistic regression model can predict the probabilities that an image contains a cat, a dog, or a car, based on features extracted from the image.

This joint modeling approach is preferred over fitting multiple binary logistic regression models when the class labels are *mutually exclusive*. For example, in the one-vs-rest strategy, a separate binary classifier is trained for each class, making independent predictions. This can lead to probabilities that do not sum to one, even though only one class should be chosen. Multinomial logistic regression addresses this by modeling all classes jointly, ensuring that the predicted probabilities form a valid distribution and capturing shared patterns across classes—for example, how the same image features affect the likelihood of different object categories.

However, when samples can belong to multiple classes simultaneously (for example, a movie labeled as both a comedy and a romance), the softmax assumption of mutually exclusive outcomes no longer applies. In such multi-label classification settings, it is more appropriate to build a separate binary classifier for each class, enabling independent predictions for each label.

5.8.1 Formal Definition

Given an input sample $(\mathbf{x}, y)$, multinomial logistic regression outputs a probability vector $\mathbf{p} = (p_1, \ldots, p_k)^T$, where each component $p_i = P(y = i|\mathbf{x})$ represents the predicted probability that the sample belongs to class i. These probabilities sum to one, i.e., $\sum_{i=1}^{k} p_i = 1$.

As discussed, binary logistic regression models the log-odds of the positive class as a linear function of the input features. Multinomial logistic regression generalizes this idea by modeling the log-odds of each class relative to a **reference class** (typically the last class, k) as a linear function of the input features. For each class $1 \leq i < k$, the log-odds of class i versus the reference class k are expressed as:

$$\log\left(\frac{p_i}{p_k}\right) = \mathbf{w}_i^T \mathbf{x}, \tag{5.33}$$

where $\mathbf{w}_i$ is the parameter vector associated with class i.

As a result, $k-1$ parameter vectors must be estimated from the data: $\mathbf{w}_1, \ldots, \mathbf{w}_{k-1}$. To enable efficient computation of class scores via matrix multiplication, these vectors are typically stacked as the rows of a parameter matrix $W \in \mathbb{R}^{(k-1)\times(d+1)}$:

$$W = \begin{pmatrix} — & \mathbf{w}_1^T & — \\ — & \mathbf{w}_2^T & — \\ & \vdots & \\ — & \mathbf{w}_{k-1}^T & — \end{pmatrix} = \begin{pmatrix} w_{10} & w_{11} & w_{12} & \cdots & w_{1d} \\ w_{20} & w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{k-1,0} & w_{k-1,1} & w_{k-1,2} & \cdots & w_{k-1,d} \end{pmatrix}. \tag{5.34}$$

To derive an expression for the predicted probability p_i of each class, we begin by exponentiating both sides of the log-odds equation:

$$\frac{p_i}{p_k} = e^{\mathbf{w}_i^T \mathbf{x}}$$
$$p_i = p_k e^{\mathbf{w}_i^T \mathbf{x}}$$

Since the class probabilities sum to 1, we can express p_k as

$$p_k = 1 - \sum_{i=1}^{k-1} p_i = 1 - \sum_{i=1}^{k-1} \left(p_k e^{\mathbf{w}_i^T \mathbf{x}} \right)$$
$$p_k + p_k \left(\sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}} \right) = 1$$
$$p_k \left(1 + \sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}} \right) = 1$$
$$p_k = \frac{1}{1 + \sum_{i=1}^{k-1} e^{\mathbf{w}_i^T \mathbf{x}}}$$

Substituting this back into the expression for p_i, we obtain for $1 \leq i < k$

$$p_i = p_k e^{\mathbf{w}_i^T \mathbf{x}} = \frac{e^{\mathbf{w}_i^T \mathbf{x}}}{1 + \sum_{j=1}^{k-1} e^{\mathbf{w}_j^T \mathbf{x}}}. \tag{5.35}$$

To simplify notation, we set $\mathbf{w}_k = 0$ for the reference class. This has no effect on the model because only the log-odds relative to the reference class are used in computing the probabilities. With this convention, $e^{\mathbf{w}_k^T \mathbf{x}} = e^0 = 1$, which allows us to rewrite the class probabilities uniformly as:

$$p_i = \frac{e^{\mathbf{w}_i^T \mathbf{x}}}{\sum_{j=1}^{k} e^{\mathbf{w}_j^T \mathbf{x}}}. \tag{5.36}$$

The expression on the right-hand side corresponds to the **softmax function** applied to the class logits $\mathbf{w}_i^T\mathbf{x}$. The softmax function $\sigma(\mathbf{z})$ maps a vector of real values $\mathbf{z} = (z_1, \ldots, z_k)^T$ to a probability distribution over k classes, defined as:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}. \tag{5.37}$$

It can be easily verified that the components of $\sigma(\mathbf{z})$ lie in the range (0,1) and sum to 1, which confirms that they form a valid probability distribution. The name "softmax" reflects the fact that the function acts as a smooth, differentiable approximation of the argmax function. For example, for the vector $\mathbf{z} = (1, 2, 6)$, the softmax output is $\sigma(\mathbf{z}) = (0.007, 0.018, 0.976)$, indicating a strong preference for the largest input value.

A common issue when computing the softmax function is numerical instability caused by the exponentiation of large logits. This can be mitigated by subtracting the maximum logit $M = \max_{1 \le i \le k} z_i$ from all components of the input vector before exponentiation:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i - M}}{\sum_{j=1}^{k} e^{z_j - M}}. \tag{5.38}$$

This transformation leaves the resulting probabilities unchanged while ensuring that all exponents are ≤ 0, which prevents overflow. Most machine learning libraries implement softmax with this shift internally.

Using the softmax function, the output of the multinomial logistic regression model can be written as

$$\mathbf{p} = \begin{pmatrix} P(y=1|\mathbf{x}) \\ \vdots \\ P(y=k|\mathbf{x}) \end{pmatrix} = \begin{pmatrix} \dfrac{e^{\mathbf{w}_1^T\mathbf{x}}}{\sum_{j=1}^{k} e^{\mathbf{w}_j^T\mathbf{x}}} \\ \vdots \\ \dfrac{e^{\mathbf{w}_k^T\mathbf{x}}}{\sum_{j=1}^{k} e^{\mathbf{w}_j^T\mathbf{x}}} \end{pmatrix} = \sigma(W\mathbf{x}), \tag{5.39}$$

where W is the matrix of coefficients (with a row of zeros corresponding to the reference class). Figure 5.19 illustrates the computational process of the multinomial logistic regression model.

5.8.2 Training the Model

Similar to binary logistic regression, we first define a suitable loss function based on the negative log-likelihood and then use iterative optimization methods such as gradient descent to minimize this function.

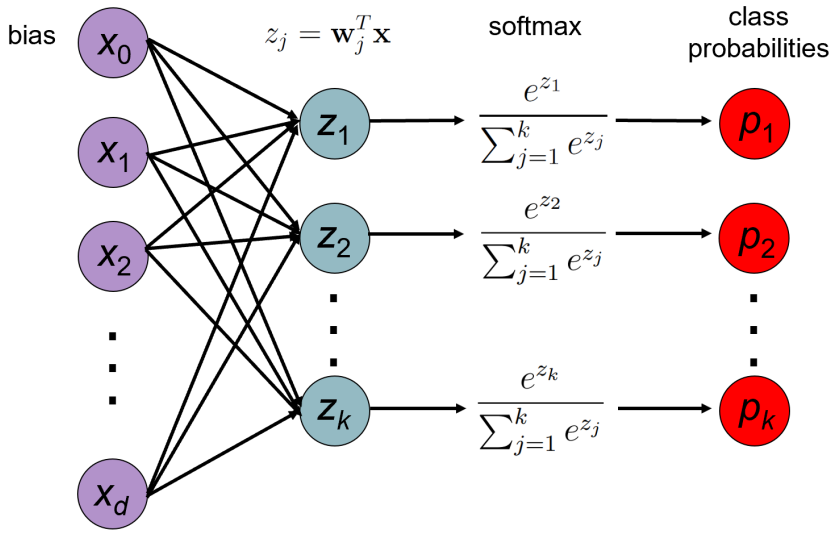

Figure 5.19: The computational process of multinomial logistic regression. The input vector $\mathbf{x} = (x_0, x_1, \ldots, x_d)$ is linearly combined with the weight vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_k$ to produce the logits $z_1, z_2, \ldots, z_k$. These logits are passed through the softmax function, which transforms them into class probabilities $p_1, p_2, \ldots, p_k$ that sum to 1.

5.8.2.1 The Cross-Entropy Loss

In multinomial logistic regression, the model outputs a probability vector $\mathbf{p} = (p_1, \ldots, p_k)$, where p_i represents the predicted probability of class i. However, the true label y for each example is typically stored as an integer index $y \in \{1, \ldots, k\}$. To align the predicted probabilities with the true label, we one-hot encode the label into a binary vector $\mathbf{y} = (y_1, \ldots, y_k)$, where $y_i = 1$ if i is the true class and 0 otherwise.

Our goal is to learn the parameters W that maximize the likelihood of the true labels under the model. We assume that the target variable y follows a **categorical distribution** (see Section C.5.6.7), which generalizes the Bernoulli distribution to $k > 2$ outcomes. In this setting, the model assigns a probability $P(y = i|\mathbf{x}) = p_i$ to each class i, and these probabilities satisfy $\sum_{i=1}^{k} p_i = 1$.

For a given sample $(\mathbf{x}, \mathbf{y})$, where $\mathbf{y} = (y_1, \ldots, y_k)$ is the one-hot representation of the true label, the likelihood of observing the true label under the model (parameterized by W) is:

$$P(\mathbf{y}|\mathbf{x}, W) = p_1^{y_1} \cdot p_2^{y_2} \ldots p_k^{y_k} = \prod_{i=1}^{k} p_i^{y_i}. \tag{5.40}$$

Since only one component $y_j = 1$ (for the true class j) and all others are zero, the likelihood reduces $P(\mathbf{y}|\mathbf{x}, W) = p_j$, the predicted probability of the correct class.

The log-likelihood is therefore

$$\log P(\mathbf{y}|\mathbf{x}, W) = \sum_{i=1}^{k} y_i \log p_i. \tag{5.41}$$

Negating this expression gives the **cross-entropy loss**:

$$L_{\text{CE}}(\mathbf{y}, \mathbf{p}) = -\sum_{i=1}^{k} y_i \log p_i. \tag{5.42}$$

When $k = 2$, this reduces to the log loss, which is why log loss is often called binary cross-entropy. The term *cross-entropy* comes from information theory, where it measures the difference between two probability distributions [179]. In this context, it quantifies the difference between the true label distribution $\mathbf{y}$ and the predicted distribution $\mathbf{p}$.

For example, consider a classification problem with three classes ($k = 3$). Suppose the true class is 2, represented as $\mathbf{y} = (0, 1, 0)$, and the model's predicted probabilities are $\mathbf{p} = (0.3, 0.6, 0.1)$. The cross-entropy loss for this instance is:

$$L_{\text{CE}} = -(0 \cdot \log 0.3 + 1 \cdot \log 0.6 + 0 \cdot \log 0.1) = 0.5108.$$

5.8.2.2 Gradient Descent

Our goal is to minimize the cross-entropy loss in order to find the optimal parameters of the softmax regression model. As in binary logistic regression, this minimization problem does not have a closed-form solution, thus we rely on iterative optimization methods such as gradient descent.

We now derive the gradient of the cross-entropy loss with respect to the model parameters. For class j ($1 \leq j \leq k$), the gradient with respect to the weight vector $\mathbf{w}_j$ is:

$$\frac{\partial L}{\partial \mathbf{w}_j} = (p_j - y_j)\mathbf{x}, \tag{5.43}$$

where p_j is the predicted probability of class j for the input $\mathbf{x}$, and y_j is the j-th element of the one-hot encoded true label vector $\mathbf{y}$.

To see why this formula holds, we start from the cross-entropy loss:

$$L = -\sum_{i=1}^{k} y_i \log p_i = -\sum_{i=1}^{k} y_i \log \left(\frac{e^{z_i}}{\sum_{d=1}^{k} e^{z_d}} \right) = -\sum_{i=1}^{k} y_i \log \left(\frac{e^{\mathbf{w}_i^T \mathbf{x}}}{\sum_{d=1}^{k} e^{\mathbf{w}_d^T \mathbf{x}}} \right), \tag{5.44}$$

where $z_i = \mathbf{w}_i^T \mathbf{x}$ is the logit for class i.

Using the chain rule, the gradient of L with respect to $\mathbf{w}_j$ is:

$$\frac{\partial L}{\partial \mathbf{w}_j} = \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial \mathbf{w}_j}. \tag{5.45}$$

Since for any vector $\mathbf{u}$, $\partial(\mathbf{u}^T\mathbf{x})/\partial \mathbf{u} = \mathbf{x}$ (see Section B.9.1.1), it follows that

$$\frac{\partial z_j}{\partial \mathbf{w}_j} = \frac{\partial \mathbf{w}_j^T \mathbf{x}}{\partial \mathbf{w}_j} = \mathbf{x}. \tag{5.46}$$

It remains to compute $\partial L/\partial z_j$. Again applying the chain rule, we differentiate with respect to z_j:

$$\frac{\partial L}{\partial z_j} = -\frac{\partial}{\partial z_j} \sum_{i=1}^{k} y_i \log p_i = -\sum_{i=1}^{k} \frac{y_i}{p_i} \cdot \frac{\partial p_i}{\partial z_j}. \tag{5.47}$$

The partial derivatives of the softmax outputs p_i with respect to the logits z_j are:

1. For $i = j$:

$$\frac{\partial p_i}{\partial z_i} = \frac{\partial}{\partial z_i} \frac{e^{z_i}}{\sum_{d=1}^{k} e^{z_d}} = \frac{\sum_{d=1}^{k} e^{z_d} e^{z_i} - e^{z_i} e^{z_i}}{\left(\sum_{d=1}^{k} e^{z_d}\right)^2} = \frac{e^{z_i}}{\sum_{d=1}^{k} e^{z_d}} \left(\frac{\sum_{d=1}^{k} e^{z_d} - e^{z_i}}{\sum_{d=1}^{k} e^{z_d}} \right) = p_i(1-p_i).$$

2. For $i \neq j$:

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial}{\partial z_j} \frac{e^{z_i}}{\sum_{d=1}^{k} e^{z_d}} = \frac{\sum_{d=1}^{k} e^{z_d} \cdot 0 - e^{z_i} e^{z_j}}{\left(\sum_{d=1}^{k} e^{z_d}\right)^2} = -\frac{e^{z_i}}{\sum_{d=1}^{k} e^{z_d}} \cdot \frac{e^{z_j}}{\sum_{d=1}^{k} e^{z_d}} = -p_i p_j.$$

Substituting into the expression for $\partial L/\partial z_j$, we obtain:

$$\frac{\partial L}{\partial z_j} = -\sum_{i=1}^{k} \frac{y_i}{p_i} \frac{\partial p_i}{\partial z_j} = \sum_{i=1}^{k} \begin{cases} y_i(p_i - 1) & i = j \\ y_i p_j & i \neq j \end{cases}.$$

Let us analyze the sum on the right-hand side. Because the label vector $\mathbf{y}$ is one-hot encoded, there are only two possibilities:

- If j is the true class, $y_j = 1$ and all other components of the vector are zero. In this case, the only nonzero term in the sum corresponds to $i = j$, and the result simplifies to $p_j - 1$, which is equal to $p_j - y_j$.

- Otherwise, $y_j = 0$ and the only nonzero term in the sum comes from the true class $i \neq j$, for which $y_i = 1$. In this case, the sum simplifies to p_j, which also equals $p_j - y_j$ (since $y_j = 0$).

Therefore,

$$\frac{\partial L}{\partial z_j} = p_j - y_j. \tag{5.48}$$

Finally, substituting Equations (5.46) and (5.48) into (5.45) yields:

$$\frac{\partial L}{\partial \mathbf{w}_j} = (p_j - y_j)\mathbf{x},$$

which matches the expression for the gradient given in Equation (5.43).

With the gradient of the loss function in hand, we can now apply (stochastic) gradient descent to iteratively update the weights and minimize the loss. The process proceeds as follows:

1. Randomly initialize the weight matrix W.
2. For each training sample:
   (a) Compute the predicted class probabilities $\mathbf{p}$ using Equation (5.39).
   (b) Calculate the gradient of the cross-entropy loss with respect to each weight vector $\mathbf{w}_j$ using Equation (5.43).
   (c) Optionally, compute the cross-entropy loss (Equation (5.42)) to monitor convergence.
   (d) Update the weights $\mathbf{w}_j$ by moving in the opposite direction of the gradient, scaled by the learning rate α:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j - \alpha(p_j - y_j)\mathbf{x}. \tag{5.49}$$

3. Repeat until a fixed number of iterations is reached, or until the change in the cross-entropy loss falls below a predefined threshold.

You will implement this algorithm in Python in Exercise 5.28.

5.8.3 Example: Classifying Handwritten Digits

The MNIST dataset [213] is a widely used benchmark in image classification, often regarded as the "Hello World" of machine learning due to its simplicity and accessibility. It has been used over the years to showcase the progress of machine learning—ranging from early linear models to deep convolutional neural networks.

The dataset contains 60,000 training images and 10,000 test images of handwritten digits. Each image is a 28×28 grayscale image, with pixel intensities ranging from 0 (black) to 255 (white). To make these images compatible with standard machine learning models, they are typically flattened into 784-dimensional vectors ($28 \times 28 = 784$). In this flattened form, each component x_j of the feature vector $\mathbf{x}$ corresponds to a single pixel.

The goal is to classify each image into one of 10 classes, representing the digits 0 to 9. This is a multi-class classification problem with mutually exclusive classes, making multinomial logistic regression a natural choice since it produces class probabilities that sum to 1.

Scikit-Learn's `LogisticRegression` class supports multi-class classification through its `multi_class` parameter, which defaults to `'auto'`. In this mode, Scikit-Learn automatically selects the multi-class strategy: it uses multinomial logistic regression (`'multinomial'`) if the dataset has more than two classes and the chosen solver supports multinomial loss (all solvers except `'liblinear'`); otherwise, it falls back to the one-vs-rest strategy (`'ovr'`).

5.8.3.1 Loading and Exploring the Dataset

We begin by fetching the MNIST dataset using the `fetch_openml` function:

```
from sklearn.datasets import fetch_openml

X, y = fetch_openml('mnist_784', return_X_y=True, as_frame=False)
```

The parameter `as_frame=False` ensures that the features and labels are returned as NumPy arrays rather than as Pandas objects.

Let's inspect the shape of the feature matrix:

```
print(X.shape)
```

```
(70000, 784)
```

The dataset contains 70,000 samples, each represented by a 784-dimensional feature vector corresponding to the 28×28 pixel grid of the images. Next, let's examine the

distribution of the digit classes:

```
np.unique(y, return_counts=True)
```

The output shows that the classes are relatively balanced:

```
(array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object),
 array([6903, 7877, 6990, 7141, 6824, 6313, 6876, 7293, 6825, 6958],
       dtype=int64))
```

To get a sense of what the digits look like, we can plot the first 50 images:

```
fig, axes = plt.subplots(5, 10, figsize=(10, 5))
for i, ax in enumerate(axes.flat):
    ax.imshow(X[i].reshape(28, 28), cmap='binary')
    ax.axis('off')
```

The resulting visualization is shown in Figure 5.20.

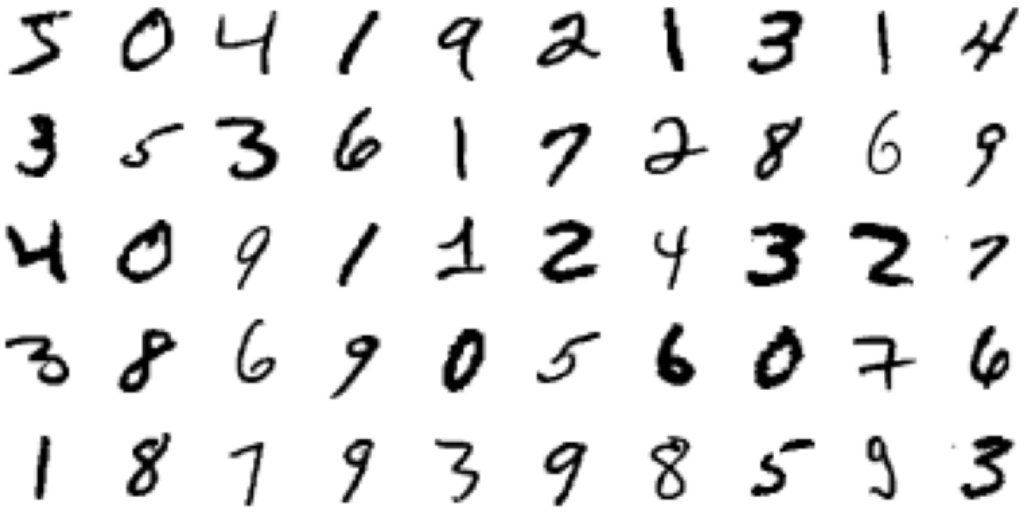

Figure 5.20: The first 50 handwritten digits from the MNIST dataset, arranged in a grid. Each digit is a 28 × 28 pixel image with pixel intensities ranging from 0 (black) to 255 (white).

5.8.3.2 Data Preprocessing

Although all pixel values are in the range [0, 255], it is common practice to scale the pixel intensities to the range [0, 1] when using gradient-based optimization methods such as gradient descent:

```
X = X / 255.0 # Scale features to [0, 1]
```

This scaling keeps the input values within a smaller range, which improves the numerical stability of the optimization and prevents the gradients of the softmax function from becoming extremely small (similar to the sigmoid, the softmax function saturates quickly as the input grows), leading to more effective learning.

This scaling reduces the dynamic range of the input features, which improves numerical stability during training and helps prevent vanishing gradients. As with the sigmoid function, the softmax function saturates for large input values, causing its gradients to become very small. Scaling the inputs helps mitigate this effect and leads to more efficient and stable learning.

Since the MNIST dataset consists of 60,000 training images followed by 10,000 test images, we can simply use array slicing to create the training and test sets:

```
train_size = 60000
X_train, y_train = X[:train_size], y[:train_size]
X_test, y_test = X[train_size:], y[train_size:]
```

5.8.3.3 Building the Model

We now instantiate a `LogisticRegression` classifier with default settings and fit it to the training set:

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(random_state=42)
clf.fit(X_train, y_train)
```

During training, you may see a warning indicating that the model did not converge within the default limit of 100 iterations:

```
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

To resolve this, we can increase the maximum number of iterations to 1000:

```
clf = LogisticRegression(max_iter=1000, random_state=42)
clf.fit(X_train, y_train)
```

After increasing the iteration limit, the model converges successfully. To check how many iterations were actually used, we can inspect the `n_iter_` attribute:

```
print(clf.n_iter_)
```

```
[265]
```

The output shows that it took 265 iterations for the solver to converge.

5.8.3.4 Evaluating the Model

Since the MNIST dataset is relatively balanced across classes, accuracy can provide a useful first assessment of model performance. We can compute it on both the training and test sets:

```
print(f'Train accuracy: {clf.score(X_train, y_train):.4f}')
print(f'Test accuracy: {clf.score(X_test, y_test):.4f}')
```

```
Train accuracy: 0.9388
Test accuracy: 0.9259
```

These results show that the model generalizes well, achieving good accuracy on both sets. However, more advanced models—particularly deep convolutional neural networks (CNNs)—perform significantly better on MNIST, achieving up to 99.8% accuracy on the test set [167, 375]. This performance gap is expected, as multinomial logistic regression is equivalent to a very shallow neural network with a single linear layer followed by a softmax output.

Although the MNIST class distribution is relatively balanced, additional evaluation metrics can provide deeper insights into the model's behavior and help identify where its errors occur.

A **classification report** summarizes key evaluation metrics for each class, including precision, recall, and F1 score, as well as macro- and weighted-average statistics across all classes. You can generate this report using Scikit-Learn's `classification_report` function:

```
from sklearn.metrics import classification_report

y_test_pred = clf.predict(X_test)
```

```
report = classification_report(y_test, y_test_pred, digits=4)
print(report)
```

This produces the following output:

```
              precision    recall  f1-score   support

           0     0.9532    0.9765    0.9647       980
           1     0.9619    0.9789    0.9703      1135
           2     0.9291    0.9012    0.9149      1032
           3     0.9038    0.9119    0.9078      1010
           4     0.9360    0.9389    0.9375       982
           5     0.8954    0.8733    0.8842       892
           6     0.9421    0.9509    0.9465       958
           7     0.9331    0.9232    0.9281      1028
           8     0.8850    0.8768    0.8809       974
           9     0.9095    0.9167    0.9131      1009

    accuracy                         0.9259     10000
   macro avg     0.9249    0.9248    0.9248     10000
weighted avg     0.9257    0.9259    0.9257     10000
```

The report shows for each class:

- **Precision**: The proportion of predictions for the class that are correct.
- **Recall**: The proportion of actual instances of the class that are correctly classified.
- **F1 score**: The harmonic mean of precision and recall.
- **Support**: The number of instances of the class in the dataset (in this case, the test set).

For example, the slightly lower precision and recall for digits 5 and 8 suggest that the model has more difficulty classifying these digits correctly.

In addition, the report includes two averaged metrics:

- **Macro average**: The unweighted average of the metric across all classes. This is useful for imbalanced datasets, as each class contributes equally to the average, ensuring that minority classes are not dominated by the majority classes.

- **Weighted average**: The average of the metric weighted by the number of instances in each class (its support). This reflects overall performance on the dataset but can obscure poor performance on minority classes.

Another useful metric for imbalanced datasets is the **micro average**, which aggregates the true positives, false positives, and false negatives across all classes and then computes the metric globally. For example, the micro-averaged precision is given by

$$\frac{\sum \text{TPs over all classes}}{\sum \text{TPs over all classes} + \sum \text{FPs over all classes}}.$$

Micro averaging reflects overall performance at the instance level, treating every prediction equally regardless of class. Like weighted averaging, it is influenced by the majority classes, as those classes naturally contribute more instances to the total count.

In Scikit-Learn, the averaging method can be specified in the `precision_score`, `recall_score`, and `f1_score` functions using the `average` parameter (for example, `average='micro'`):

```
from sklearn.metrics import precision_score, recall_score, f1_score

micro_precision = precision_score(y_test, y_test_pred, average='micro')
micro_recall = recall_score(y_test, y_test_pred, average='micro')
micro_f1 = f1_score(y_test, y_test_pred, average='micro')

print(f'Micro Precision: {micro_precision:.4f}')
print(f'Micro Recall: {micro_recall:.4f}')
print(f'Micro F1 Score: {micro_f1:.4f}')
```

```
Micro Precision: 0.9259
Micro Recall: 0.9259
Micro F1 Score: 0.9259
```

For single-label multi-class problems, both micro-precision and micro-recall are equal to the overall accuracy. This is because when aggregating over all classes, every false positive for one class is also a false negative for another, making $\sum \text{FP} = \sum \text{FN}$. In this example, the micro scores are very close to the macro and weighted averages shown in the classification report, as MNIST is relatively balanced and the model performs similarly across classes.

5.8.3.5 Error Analysis

To better understand where the model makes mistakes, we can examine the confusion matrix for the test set predictions:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_test, y_test_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap='Blues')
```

Figure 5.21 shows that most errors occur between digits with similar shapes, such as 5 vs. 3 and 2 vs. 8.

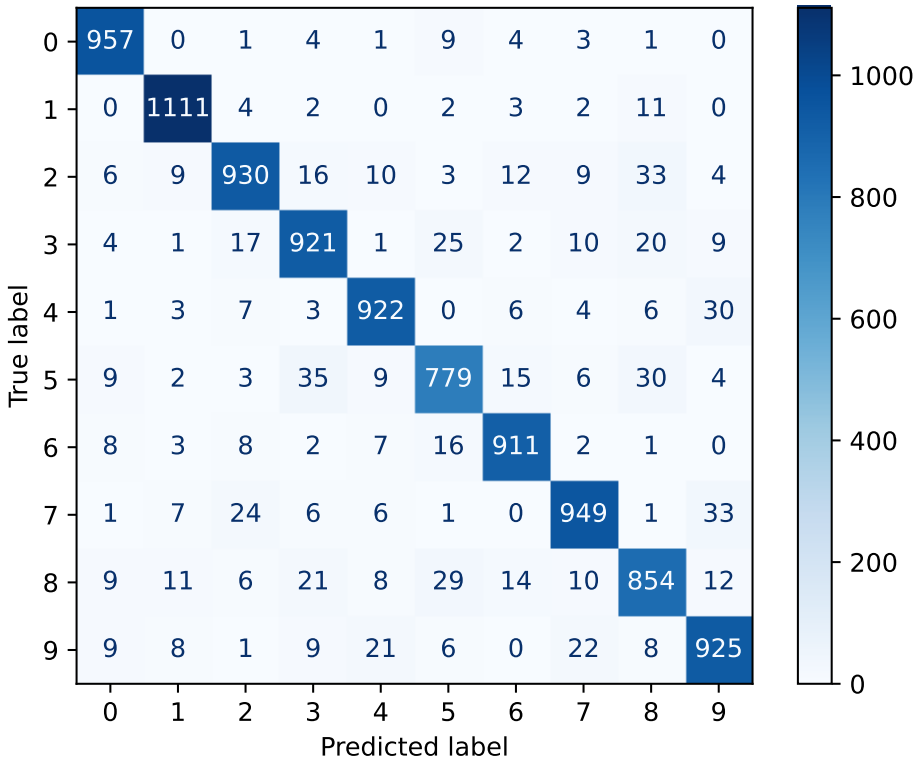

Figure 5.21: Confusion matrix for the MNIST test set predictions. Each entry shows how many instances of a given digit (true label) were predicted as another digit (predicted label). Most errors occur between visually similar digits, such as 3 vs. 5, and 5 vs. 8.

To explore specific errors, we can extract cases where the model predicted 3 instead of 5:

```
X_5conf3 = X_test[(y_test == '5') & (y_test_pred == '3')]

# Display these misclassified images in a grid
num_images = len(X_5conf3)
rows = 3
cols = (num_images + rows - 1) // rows

fig, axes = plt.subplots(rows, cols, figsize=(cols, rows))
axes = axes.flatten()
for i in range(num_images):
    axes[i].imshow(X_5conf3[i].reshape(28, 28), cmap='binary')
    axes[i].axis('off')

# Hide any unused subplots
for i in range(num_images, len(axes)):
    axes[i].axis('off')
```

Figure 5.22 presents the 35 examples of digit 5 that were misclassified as 3. It is evident that some of these handwritten 5s closely resemble a 3, explaining why the model struggled to distinguish them.

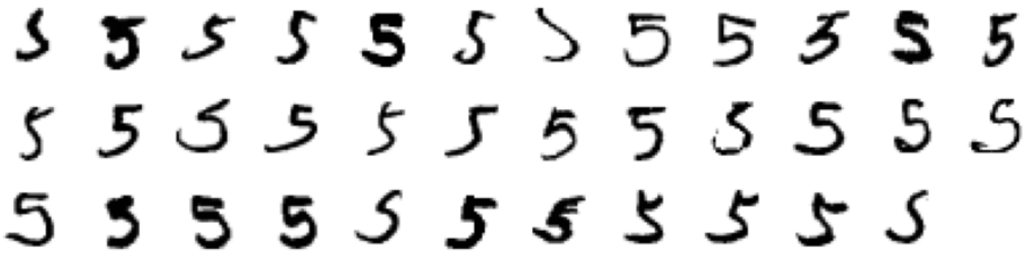

Figure 5.22: Examples of digit 5 from the MNIST test set that were misclassified as 3. Several of these errors appear to result from handwriting styles where the shape of 5 resembles that of a 3.

To improve the model's ability to distinguish between similar-looking digits, several strategies can be employed:

- **Data augmentation**: Expand the training set with additional examples of frequently confused digits, either by collecting new samples or by generating synthetic variations of existing images using operations such as rotations, scaling, or adding noise. Such augmentation helps the model learn subtle differences between similar digits, such as 5 and 3.

- **Feature engineering**: Extract higher-level features from the raw pixel values, such as the number of loops in a digit or the presence of specific curves, to enhance the model's ability to distinguish challenging cases.

5.8.3.6 Model Interpretation

A key advantage of logistic regression over other classification models is its high interpretability: the weight assigned to each feature indicates its contribution to the classification decision.

For the MNIST dataset, we can visualize the learned weights assigned for each pixel in each digit class. This can help us identify which regions of the image contribute most to recognizing each digit.

The weight matrix of the trained model is stored in the `coef_` attribute, whose shape is (10, 784). Each row i corresponds to the weight vector $\mathbf{w}_i$ for class i. By reshaping each row into a 28×28 grid, we can visualize these weights as images:

```
fig, axes = plt.subplots(2, 5, figsize=(15, 5))

for i, ax in enumerate(axes.flat):
    # Reshape the i-th row of the weight matrix and plot it
    img = ax.imshow(clf.coef_[i].reshape(28, 28), cmap='gray', vmin=-0.5,
        vmax=0.5)
    ax.axis('off')
    ax.set_title(f'Digit {i}')

# Add a colorbar to show the weight scale
fig.colorbar(img, ax=axes.flat)
```

Figure 5.23 visualizes the weight patterns learned by the model. Lighter areas (positive weights) correspond to pixels that support the prediction of that digit, while darker areas (negative weights) correspond to pixels that inhibit the prediction. Pixels shaded in neutral gray (weights close to 0) have little influence on the prediction, often corresponding to background regions near the image borders.

These visualizations provide insight into how the model distinguishes between digits by highlighting the most informative areas of each image. Such understanding can guide future improvements, for example, by augmenting images in those regions or by extracting features from those specific areas.

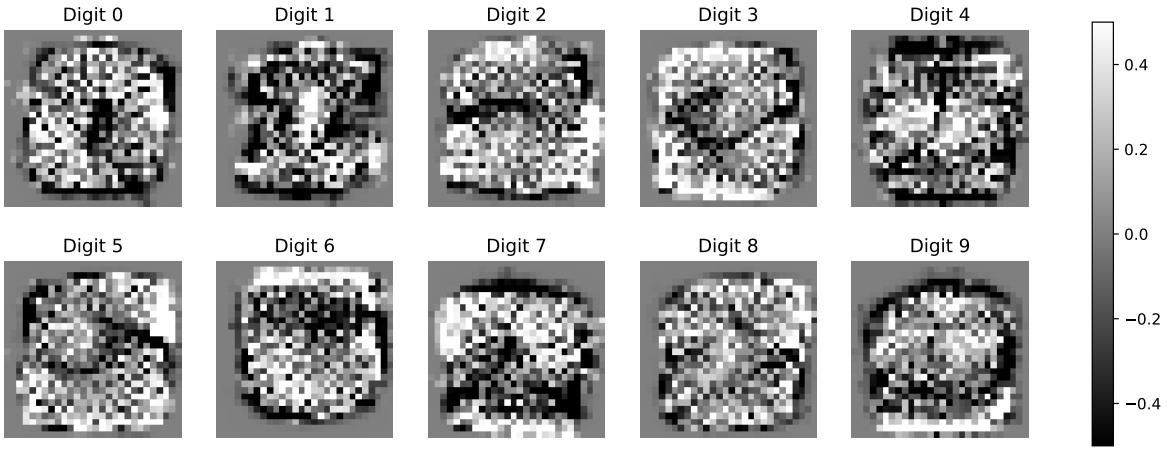

Figure 5.23: Visualization of the weight vectors for each digit class in the multinomial logistic regression model trained on MNIST. Lighter regions represent positive weights that support the prediction of that digit, darker regions represent negative weights that suppress it, and neutral gray regions indicate weights close to zero with minimal impact on the prediction. Notably, the pattern of white pixels in each image roughly outlines the central shape of the digit.

5.9 (*) Generalized Linear Models

Up to this point, we have studied two important models: linear regression, where the target variable is assumed to follow a normal distribution with a mean equal to a linear combination of the input features; and logistic regression, where the target variable is assumed to follow a Bernoulli distribution whose mean is given by applying the sigmoid function to a linear combination of the features. These models, while seemingly different, share the same underlying structure and are, in fact, special cases of a broader family of models known as **generalized linear models (GLMs)** [609, 230].

Generalized linear models extend the familiar structure of linear models to handle a wide variety of response variable types within a unified framework. They preserve the idea of combining the input features linearly but generalize how the mean of the response variable relates to this predictor and allow the response to follow various probability distributions. This allows GLMs to handle binary, count, categorical, and other types of data, while retaining the advantages of linear models such as computational efficiency and interpretability.

A generalized linear model consists of two main components:

1. **Link function**: This function connects the linear predictor (a linear combination

of the input features) to the mean of the response variable y. Formally, a link function g maps the expected value of y, denoted by $\mu = \mathbb{E}[y]$, to the linear predictor η:

$$\eta = g(\mu) = \mathbf{w}^T\mathbf{x}. \tag{5.50}$$

For example, in logistic regression, the link function is the *logit function*, which maps the probability of the positive class $p = P(y = 1)$ (the mean of the Bernoulli-distributed response variable) to the log-odds, expressed as a linear combination of the features:

$$g(p) = \log\left(\frac{p}{1-p}\right) = \mathbf{w}^T\mathbf{x} = \eta.$$

The **inverse link function** $h = g^{-1}$ maps the linear predictor back to the mean response: $\mu = h(\eta)$. In logistic regression, this inverse link is the *sigmoid function*, which maps η to a probability:

$$h(\eta) = \frac{1}{1+e^{-\eta}}.$$

2. **Exponential family distributions**: In a generalized linear model, the response variable is assumed to follow a distribution from the **exponential family** (not to be confused with the exponential distribution). This family includes many common distributions such as Bernoulli, Poisson, Gaussian, Gamma, and others, allowing GLMs to model a wide range of data types.

   The general form of a probability distribution in the exponential family is

   $$p(y|\theta,\phi) = \exp\left(\frac{y\theta - b(\theta)}{a(\phi)} + c(y,\phi)\right), \tag{5.51}$$

   where θ is the natural parameter of the distribution (which in general is not the same as the mean), ϕ is a dispersion (or scale) parameter, and $a(\phi)$, $b(\theta)$, and $c(y,\phi)$ are known functions that define the specific distribution.

   For example, consider the Bernoulli distribution used in logistic regression. Its probability mass function is:

   $$p(y|\eta) = p^y(1-p)^{1-y},$$

   where $p = P(y = 1)$. In logistic regression, this probability is modeled using the sigmoid function:

   $$p = \frac{1}{1+e^{-\eta}} = \frac{e^\eta}{1+e^\eta}, \quad 1-p = \frac{1}{1+e^\eta},$$

where $\eta = \mathbf{w}^T\mathbf{x}$ is the linear predictor. Substituting these expressions into the PMF gives

$$p(y|\eta) = \left(\frac{e^\eta}{1+e^\eta}\right)^y \left(\frac{1}{1+e^\eta}\right)^{1-y} = \frac{e^{y\eta}}{1+e^\eta}.$$

This can be written in exponential-family form as

$$p(y|\eta) = \exp\left(y\eta - \log(1+e^\eta)\right).$$

Here, $\theta = \eta$, $b(\theta) = \log(1+e^\eta)$, $a(\phi) = 1$, and $c(y,\phi) = 0$.

This derivation shows that the Bernoulli distribution belongs to the exponential family when parameterized by the log-odds (the natural parameter).

An important consequence of this formulation is that the maximum likelihood estimation (MLE) procedure for GLMs has a unified structure. Given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, the log-likelihood under a GLM is

$$\ell(\mathbf{w}) = \sum_{i=1}^n \left[\frac{y_i\theta_i - b(\theta_i)}{a(\phi)} + c(y_i,\phi)\right]. \tag{5.52}$$

When the link function is chosen such that the linear predictor equals the natural parameter, $\theta = \eta$, it is called the **canonical link**. This is the default choice in most GLMs, as it leads to simpler expressions. For example, the logit link is canonical for a Bernoulli response, and the identity link is canonical for a normal response.

With a canonical link, the log-likelihood (5.52) simplifies to

$$\ell(\mathbf{w}) = \sum_{i=1}^n \left[y_i\eta_i - b(\eta_i)\right], \tag{5.53}$$

and its gradient with respect to $\mathbf{w}$ takes the particularly simple form:

$$\frac{\partial \ell}{\partial \mathbf{w}} = \sum_{i=1}^n (y_i - \mu_i)\ \mathbf{x}_i, \tag{5.54}$$

where $\mu_i = h(\eta_i)$ is the predicted mean response. Comparing this result with Equations (4.27) and (5.16) shows that the gradients for both linear and logistic regression are special cases of this general GLM gradient.

In Scikit-Learn, a subset of generalized linear models known as Tweedie distributions [821] is implemented by the `TweedieRegressor` class. Key parameters of the class include:

- `power`: Specifies the distribution of the response variable. For example, `power=0` corresponds to a normal distribution, `power=1` to a Poisson distribution, and `power=2` to a gamma distribution.
- `link`: Specifies the link function of the GLM. By default, it is set to `'auto'`, which selects the canonical link corresponding to the chosen `power` parameter (for example, `'identity'` for the normal distribution and `'log'` for Poisson).

The following example demonstrates how to use `TweedieRegressor` to model sample data that follow a Poisson distribution, such as the number of bike rentals occurring in a fixed time interval given the average temperature:

```
from sklearn.linear_model import TweedieRegressor

X = [[5], [10], [15], [18], [20], [25]] # Daily average temperature (Celsius)
y = [50, 80, 130, 180, 220, 300] # Number of bike rentals

reg = TweedieRegressor(power=1, link='log') # power=1 for Poisson
reg.fit(X, y)
print(f'R2 score: {reg.score(X, y):.4f}')
```

```
R2 score: 0.9884
```

Here, `X` contains the predictor values (daily average temperatures) and `y` is the response (the number of bike rentals for each day). Since we are modeling count data, we assume that each y_i follows a Poisson distribution with rate parameter λ_i, which represents the expected number of rentals on day i. The model relates this rate parameter to the input features via the log link function: $\log \lambda_i = \mathbf{w}^T \mathbf{x}_i$, which implies that $\mathbb{E}[y_i] = \lambda_i = \exp(\mathbf{w}^T \mathbf{x}_i)$. This formulation ensures that the predicted rate λ_i is always positive, while preserving the interpretability of a linear model on the log scale.

5.10 Summary

In this chapter, we explored logistic regression, a foundational method for modeling binary outcomes from a set of input features. By applying the logistic (sigmoid) function to a linear combination of the features, the model produces probabilistic predictions that are easy to interpret. Its simplicity and computational efficiency make logistic regression a strong baseline for many classification problems, particularly when the decision boundary is approximately linear.

We have also discussed extensions of logistic regression to multi-class problems and various strategies for handling imbalanced datasets, including sampling techniques and cost-sensitive learning.

Advantages of logistic regression compared to other classification models:

- It efficiently finds a separating hyperplane when the classes are (approximately) linearly separable.
- It scales well to large datasets, as the number of parameters grows linearly with the number of features.
- It has a convex optimization objective, allowing gradient-based methods to converge efficiently to the global optimum.
- It is typically resistant to overfitting, especially when regularization techniques like L1 or L2 are applied.
- It produces probability estimates rather than only class labels.
- It is highly interpretable, with feature coefficients indicating the influence of each feature on the prediction.
- It requires only a few hyperparameters, making it relatively easy to tune.

Disadvantages of logistic regression:

- It can only represent linear decision boundaries, which may lead to underfitting when the relationship between the features and the target is nonlinear.
- It relies on the assumption of a linear relationship between the predictors and the log-odds of the positive class, which may not hold in real-world data.
- It is often outperformed by more flexible models (e.g., decision trees, neural networks) on data with nonlinear patterns or complex interactions.
- It is sensitive to outliers, which can disproportionately affect the decision boundary.
- High multicollinearity among predictors can hurt the model's effectiveness and interpretability.
- It requires feature scaling for efficient convergence of gradient-based optimization.
- It does not inherently handle missing data; imputation or exclusion of incomplete samples is necessary.

5.11 Exercises

5.11.1 Multiple-Choice Questions

5.1 Which of the following statements is true about logistic regression?

(a) Logistic regression models the odds ratio as a linear combination of the input features.

(b) Logistic regression minimizes the cross-entropy loss function.

(c) Logistic regression has a closed-form analytical solution.

(d) Logistic regression assumes that the class labels are sampled from a binomial distribution.

(e) Logistic regression is highly susceptible to multicollinearity among features.

5.2 If a logistic regression classifier predicts a probability of 0.75 for the positive class, what is the corresponding logit value?

(a) 0.75

(b) $\dfrac{0.75}{0.25}$

(c) $\log\left(\dfrac{0.75}{0.25}\right)$

(d) $\dfrac{1}{1+e^{-0.75}}$

5.3 You have built a logistic regression model to predict whether a financial transaction is fraudulent ($y = 1$) or legitimate ($y = 0$). The model includes two features: the amount of the transaction (x_1) and the number of transactions made by the user in the last 24 hours (x_2). The coefficients of the logistic regression model are: $w_0 = -3$ (bias), $w_1 = 0.01$, and $w_2 = 0.5$. Which of the following transactions has the highest predicted probability of being fraudulent?

(a) A transaction of \$100 with 2 transactions in the last 24 hours

(b) A transaction of \$200 with 1 transaction in the last 24 hours

(c) A transaction of \$80 with 3 transactions in the last 24 hours

(d) A transaction of \$50 with 4 transactions in the last 24 hours

5.4 You are developing a logistic regression model to classify data into two categories. The training set consists of three one-dimensional points:

- $x_1 = 3$ with label $y_1 = 1$
- $x_2 = 1$ with label $y_2 = 1$
- $x_3 = -1$ with label $y_3 = 0$

Which of the following are possible values for b (bias) and w (coefficient of x) in the logistic regression model?

(a) $w = 1, b = 2$

(b) $w = 0, b = 1$

(c) $w = -1, b = -2$

(d) $w = 2, b = 1$

5.5 Which of the following are advantages of Newton's method over gradient descent?

(a) It converges faster on non-convex functions.

(b) It has a lower computational cost per iteration.

(c) It has a faster convergence rate near the optimal solution.

(d) It is more stable when working with high-dimensional data.

(e) It automatically adjusts the learning rate at each iteration based on the curvature of the function.

5.6 You have trained a logistic regression model but observed very low accuracy on the training set. Which of the following could be possible reasons for this?

(a) The number of iterations in gradient descent was too small.

(b) The number of iterations in gradient descent was too large.

(c) The learning rate in gradient descent was too high.

(d) The learning rate in gradient descent was too low.

(e) The regularization parameter λ was too high.

(f) The features were not scaled properly.

5.7 You are evaluating a spam filter designed to classify emails as either spam ($y = 1$) or ham ($y = 0$). After testing your model on a dataset, you obtain the following confusion matrix values:

- True Positives (TP): 85
- False Positives (FP): 45
- True Negatives (TN): 100
- False Negatives (FN): 15

What is the F1 score of the model?

(a) 0.68

(b) 0.71

(c) 0.74

(d) 0.77

5.8 Which of the following statements about the ROC curve is true?

(a) The ROC curve is a monotonically non-decreasing function.

(b) The ROC curve illustrates the tradeoff between recall and false positive rate at various thresholds.

(c) If the ROC curve passes through the point (1, 1), the model correctly classifies all the positives.

(d) An area under the ROC curve (AUC) of 0.5 suggests that the model's predictions are random.

(e) The ROC curve is sensitive to changes in the class distribution.

5.9 What are the drawbacks of using SMOTE for imbalanced datasets?

(a) The synthetic samples generated by SMOTE might overlap with the majority class, leading to increased misclassification.

(b) SMOTE assumes that synthetic instances created between two minority class instances are always representative of the minority class.

(c) SMOTE can introduce noise into the majority class data.

(d) SMOTE can cause overfitting by making the decision regions for the minority class become overly specific.

(e) SMOTE can cause the model to rely too much on interpolated data, which might affect the generalization ability.

5.10 After training a multinomial logistic regression model on a dataset with four classes, you obtain the following class scores (logits) for a test sample: [2, 1, -1, 0]. What is the probability that this sample belongs to the second class?

(a) 0.2447

(b) 0.2369

(c) 0.5761

(d) 0.1223

5.11.2 Theoretical Exercises

5.11 Consider a logistic regression model developed to predict the likelihood of patients having diabetes based on two health indicators: body mass index (BMI) and age. The model's estimated coefficients are as follows:

- Intercept (w_0): -1.0
- Coefficient for BMI (w_1): 0.05
- Coefficient for age (w_2): 0.03

Calculate the probability that a patient with BMI 35 and age 50 has diabetes according to this model. Show your calculations.

5.12 Consider the dataset shown in Table 5.6, where x is a predictor variable and y is a binary outcome variable.

x	y
1	0
2	0
10	1

Table 5.6: Sample dataset used for logistic regression analysis.

(a) Write the log-likelihood function for the logistic regression model based on the given dataset.

(b) Provide an expression for the gradient of the log-likelihood function with respect to the coefficients w_0 and w_1.

(c) Perform three iterations of batch gradient descent by hand with a learning rate of $\alpha = 0.5$ to estimate the coefficients w_0 and w_1. Begin with $w_0 = 0$ and $w_1 = 0$, and show all intermediate calculations for each step.

(d) Based on the final estimated coefficients, what are the predicted probabilities for each sample in the dataset?

5.13 Prove the following properties of the sigmoid function $\sigma(z) = \dfrac{1}{1+e^{-z}}$:

(a) $\sigma(-z) = 1 - \sigma(z)$.

(b) $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

5.14 The goal of this exercise is to analyze the gradient-based optimization process of logistic regression and interpret the influence of each model component on the update steps and the resulting decision boundary.

(a) Show how the binary logistic regression model can be written in terms of the conditional probability distribution as follows:

$$p(y|\mathbf{x};\mathbf{w}) = \sigma\left((-1)^{y+1} f(\mathbf{x};\mathbf{w})\right),$$

where σ is the sigmoid function and $f(\mathbf{x};\mathbf{w}) = \mathbf{w}^T\mathbf{x}$ is the linear predictor.

(b) Consider the gradient descent algorithm, which seeks to find the maximum likelihood estimation (MLE) solution for logistic regression. Show how to write the update step of the algorithm in the following form:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \sum_{i=1}^{n} \left(1 - p(y_i|\mathbf{x}_i;\mathbf{w}^{(t)})\right)(-1)^{y_i} \nabla_{\mathbf{w}} f(\mathbf{x}_i;\mathbf{w}^{(t)}).$$

(c) Try to give an intuitive interpretation of the role of the different variables in the previous update step. Start by ignoring the term $(1 - p(y_i|\mathbf{x}_i;\mathbf{w}^{(t)})$, and obtain the following simplified update step:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \sum_{i=1}^{n} (-1)^{y_i} \nabla_{\mathbf{w}} f(\mathbf{x}_i;\mathbf{w}^{(t)}).$$

Explain how each update step affects the function $f(\mathbf{x};\mathbf{w})$. Specifically, explain how the value of the function is updated at points $\mathbf{x}_i$, and how it changes depending on whether $y_i = 1$ or $y_i = 0$.

(d) Consider the term $\left(1 - p(y_i|\mathbf{x}_i; \mathbf{w}^{(t)})\right)$ once again. What does this term represent when the model assigns a high probability versus a low probability to a specific sample $(\mathbf{x}_i, y_i)$?

Now, treat this term as a weighting factor that assigns different weights to each example. How does this weighting scheme affect the updates in each step of the gradient descent?

5.15 In this exercise, you will derive the Hessian of the logistic regression cost function, analyze its properties, and formulate the Newton update step.

(a) Starting from the gradient of the logistic regression cost function (Equation (5.16)), derive the Hessian of the cost function with respect to $\mathbf{w}$. Show that the Hessian matrix has the form:

$$H(\mathbf{w}) = \frac{1}{n} X^T R X,$$

where $X \in \mathbb{R}^{n \times d}$ is the design matrix, and $R \in \mathbb{R}^{n \times n}$ is a diagonal matrix with entries $r_i = \sigma(\mathbf{w}^T \mathbf{x}_i)(1 - \sigma(\mathbf{w}^T \mathbf{x}_i))$.

(b) Prove that the logistic regression cost function is convex by showing that the Hessian matrix is positive semidefinite.

(c) Discuss the conditions under which the Hessian $H(\mathbf{w}) = X^T R X$ is invertible. What properties of the data matrix X and the predicted probabilities $\mathbf{p}$ ensure invertibility?

(d) Use your result to derive the Newton update step:

$$\mathbf{w} \leftarrow \mathbf{w} - (X^T R X)^{-1} X^T (\mathbf{p} - \mathbf{y}).$$

5.16 In this exercise we will explore the impact of different regularization strategies on the training error of a logistic regression model. Figure 5.24 shows a two-dimensional training set, where '+' corresponds to class $y = 1$ and 'x' to class $y = 0$. The dataset is linearly separable.

The logistic regression model used for this classification task is:

$$P(y = 1|\mathbf{x}; \mathbf{w}) = \sigma(w_0 + w_1 x_1 + w_2 x_2) = \frac{1}{1 + \exp(-w_0 - w_1 x_1 - w_2 x_2)}.$$

(a) Consider a regularized logistic regression model that maximizes the following penalized log-likelihood:

$$\sum_{i=1}^{n} \log(P(y_i|\mathbf{x}_i; w_0, w_1, w_2)) - \lambda w_j^2,$$

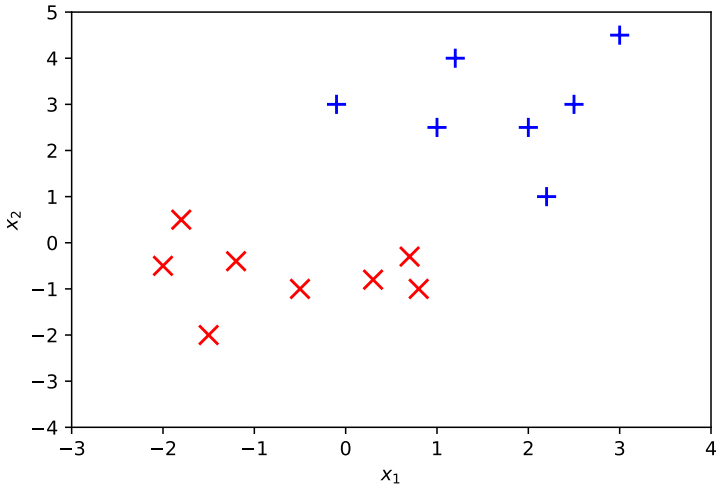

Figure 5.24: Two-dimensional labeled training set, where '+' corresponds to class $y = 1$ and 'x' corresponds to class $y = 0$.

where λ is very large and regularization is applied to one parameter w_j at a time (for $j = 0, 1, 2$). For each of the following cases, analyze how the training error is affected. Would you expect the error to increase, decrease, or stay the same? Justify your answer.

i. Regularizing w_2
ii. Regularizing w_1
iii. Regularizing w_0 (the bias term)

(b) Now consider an L1-norm penalty applied to w_1 and w_2 (but not w_0). The penalized log-likelihood is given by:

$$\sum_{i=1}^{n} \log(P(y_i|\mathbf{x}_i; w_0, w_1, w_2)) - \lambda(|w_1| + |w_2|).$$

i. As λ increases, which coefficient will reach zero first: w_1, w_2, or will they reach zero at the same time?
ii. For very large λ, what value might you expect w_0 to take? (Note that both classes have the same number of data points.)
iii. For very large λ, what range of values might you expect w_0 to take if the class labels become unbalanced (e.g., more '+' instances)? Explain how the class imbalance might affect this outcome.

5.17 Suppose that you are working on a spam detection system. You formulated the problem as a classification task where "Spam" is the positive class and "Ham" (non-spam) is the negative class. Your training set contains 1000 emails, 99% of these are Ham and 1% are Spam.

(a) What is the accuracy of a classifier that always predicts Ham?

(b) Suppose you trained a classifier on this training set, and you obtained the confusion matrix shown in Table 5.7.

		Predicted class	
		Spam	Ham
Actual class	Spam	8	2
	Ham	16	974

Table 5.7: Confusion matrix for a spam detection system

What are the accuracy, precision, recall, and F1 score of the classifier?

5.18 Consider a binary classification problem where the positive class constitutes $\alpha\%$ of the dataset, and a classifier that predicts randomly with probability β that a sample belongs to the positive class. What is the expected precision and recall of such a classifier?

5.19 Does increasing recall always lead to a decrease in precision? If true, provide a mathematical proof. If not, provide a counterexample.

5.20 Consider a binary classification problem with an imbalanced dataset, where the positive class is the minority. Suppose you apply the Synthetic Minority Oversampling Technique (SMOTE) to balance the dataset.

(a) Prove that applying SMOTE does not change the decision boundary of a classifier whose predictions depend only on a linear combination of the features and are invariant to linear transformations (e.g., logistic regression). For simplicity, assume that SMOTE generates synthetic points using a single interpolation factor $\lambda \in [0, 1]$ across all dimensions.

(b) Compare this to cost-sensitive learning, where the classifier's loss function is modified using a cost matrix. Derive the new decision boundary for a linear classifier under cost-sensitive learning, assuming different misclassification costs for the minority and majority classes. Show that the decision boundary in cost-sensitive learning is not invariant to linear scaling of the input data.

5.21 Prove that the softmax function used in multinomial logistic regression reduces to the sigmoid function when there are only two classes. In addition, show that the cross-entropy loss in this case reduces to the log loss used in binary logistic regression.

5.22 You are given a dataset of images and you need to classify them along two dimensions: dogs vs. cats and indoor vs. outdoor. Should you implement two separate logistic regression classifiers or a single multinomial logistic regression classifier? Justify your answer.

5.23 (*) Show that ordinary least squares (OLS) linear regression is a special case of a generalized linear model (GLM) with a normal distribution and the identity link function.

5.11.3 Programming Exercises

5.24 In this exercise, you will use the Pima Indians Diabetes dataset to train and evaluate a logistic regression model. This dataset includes diagnostic measurements from a large sample of individuals of Pima Indian heritage. The goal is to predict whether the patient shows signs of diabetes based on several predictor variables such as the number of pregnancies the patient has had, BMI, age, insulin level, and other clinical features.

(a) Load the dataset using the `fetch_openml` function:

```
from sklearn.datasets import fetch_openml
X, y = fetch_openml('diabetes', version=1, return_X_y=True,
    as_frame=True)
```

(b) Exploratory data analysis (EDA):

i. Conduct an initial analysis to understand the structure of the data (e.g., examine the types of features, check for missing values, etc.).

ii. Visualize the feature distributions and correlations.

iii. What is the class distribution of the target variable?

(c) Data preprocessing:

i. Scale the continuous features appropriately (e.g., using standard scaling) to prepare the data for logistic regression.

ii. Split the dataset into training and test sets (e.g., 80–20 split).

(d) Model training:

i. Train a logistic regression model with default settings on the training data.

ii. How many iterations were needed for the optimization algorithm to converge?

(e) Model evaluation:

i. Evaluate the model using several metrics: accuracy, precision, recall, F1 score, and the area under the ROC curve (AUC).

ii. Plot the ROC curve for the model.

(f) Hyperparameter tuning:

i. Use grid search with cross-validation to find the optimal settings for the hyperparameters, such as the regularization coefficient `C` and the solver used.

ii. Evaluate the performance of the model with the optimized parameters on the test set and compare it to the performance of the initial model.

(g) Discuss the overall impact of handling class imbalance on the model's generalization and usefulness in a real-world scenario.

5.25 Implement Newton's method for unconstrained optimization and apply it to logistic regression.

(a) Write a Python function that implements Newton's method for minimizing a scalar-valued function $f\colon \mathbb{R}^n \to \mathbb{R}$ that is twice-differentiable. Your function should take the following inputs:

- A function $f(\mathbf{x})$ that returns the value of the objective function at a given point $\mathbf{x}$.
- A function $\nabla f(\mathbf{x})$ that returns the gradient vector at a given point $\mathbf{x}$.
- A function $H(\mathbf{x})$ that returns the Hessian matrix at a given point $\mathbf{x}$.
- An initial guess $\mathbf{x}_0$.
- A tolerance value for convergence.
- A maximum number of iterations.

(b) Use the following Newton update rule:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - H(\mathbf{x}_n)^{-1}\nabla f(\mathbf{x}_n).$$

Terminate the iterations when either the change in $\mathbf{x}$ or the norm of the gradient is smaller than the given tolerance.

(c) Test your implementation by a minimizing a simple convex function such as:

$$f(\mathbf{x}) = (x_1 - 1)^2 + 2(x_2 + 2)^2.$$

Verify that your implementation converges to the global minimum.

(d) Now, apply your Newton optimizer to train a logistic regression model on a binary classification dataset (e.g., the Iris dataset with two classes or a synthetic dataset). Use the log loss as the objective function and compute its gradient and Hessian analytically (see Exercise 5.15).

(e) Compare your implementation with Scikit-Learn's `LogisticRegression(solver='newton-cg`. Report differences in convergence speed, objective value, and classification accuracy.

(f) Discuss the advantages and limitations of using Newton's method for logistic regression. In which situations might gradient descent or quasi-Newton methods be preferable?

5.26 Implement the algorithm for building an ROC curve for a binary classifier (Algorithm 5.1). Use the following steps:

(a) Write a function that takes the true labels (`y_true`) and predicted probabilities (`y_scores`) from a binary classifier and returns the true positive rates (TPR) and false positive rates (FPR) at all possible classification thresholds.

(b) Write another function that uses these rates to plot the ROC curve using Matplotlib.

(c) Write a function that calculates the Area Under the Curve (AUC) by approximating the integral using the trapezoidal rule. Specifically, if (x_i, y_i) and (x_{i+1}, y_{i+1}) are two consecutive points on the ROC curve, the area of the trapezoid between these points is given by:

$$\text{Area} = \frac{1}{2}(y_{i+1} + y_i) \cdot (x_{i+1} - x_i)$$

Summing the areas of all trapezoids yields the AUC.

(d) Train a binary logistic regression model using the Pima Indians Diabetes dataset (or another binary dataset of your choice).

(e) Use your functions to plot the ROC curve and calculate the AUC score.

(f) Compare the AUC score obtained from your implementation with Scikit-learn's built-in `roc_auc_score` function to ensure correctness.

5.27 In this exercise, you will explore the effectiveness of various imbalanced learning techniques on the Bank Marketing Dataset, which contains data from phone calls made by a Portuguese banking institution. The goal is to predict whether clients will subscribe to a term deposit.

(a) Dataset preparation:

i. Download the `bank.zip` file from the UCI Machine Learning Repository at this link.

ii. Extract the file `bank.csv` from the downloaded zip file.

iii. Load it into your notebook using the `pd.read_csv` function as follows:

```
import pandas as pd
df = pd.read_csv('bank.csv', delimiter=';')
```

iv. Conduct a preliminary data analysis to explore the dataset and assess the degree of class imbalance.

v. Preprocess the data (e.g., handle missing values, scale continuous features, and encode categorical variables as necessary).

(b) Initial model training:

i. Train a logistic regression model using the original, imbalanced dataset.

ii. Evaluate the model on the test set using metrics such as accuracy, precision, recall, F1 score, and AUC. Additionally, plot both the precision–recall curve and the ROC curve for the test set.

iii. Optionally, evaluate the model on the training set to check for potential overfitting.

(c) Class balancing with SMOTE:

i. Apply SMOTE (Synthetic Minority Over-sampling Technique) to generate synthetic samples and balance the class distribution.

ii. Visualize the class distribution before and after applying SMOTE (e.g., using a bar plot of class counts).

iii. Retrain the logistic regression model on the balanced dataset.

iv. Evaluate the new model on the test set using the same metrics (accuracy, precision, recall, F1 score, and AUC) and compare the results with the original imbalanced model.

v. Optionally, evaluate the new model on the training set to monitor for overfitting after SMOTE.

(d) Cost-sensitive learning:

i. Modify the logistic regression model to handle the imbalance using cost-sensitive learning by adjusting the `class_weight` parameter in Scikit-Learn.

ii. Train the cost-sensitive logistic regression model on the original imbalanced dataset.

iii. Evaluate this model on the test set using the same metrics and compare the performance of the cost-sensitive model with the baseline and SMOTE-enhanced models.

iv. Optionally, evaluate the model on the training set to detect potential overfitting or underfitting.

(e) Bonus (other sampling techniques):

i. Experiment with other oversampling techniques, such as Borderline-SMOTE or ADASYN, using the `imbalanced-learn` package.

ii. Experiment with under-sampling techniques or combinations of over-sampling and under-sampling.

iii. Compare the results with SMOTE and cost-sensitive learning to determine the most effective technique for this dataset.

(f) Write a short report summarizing your findings. Your report should include:

i. A brief description of each class-balancing technique you applied.

ii. A comparison of model performance across techniques using evaluation metrics (e.g., accuracy, precision, recall, F1 score, and AUC).

iii. Plots of the ROC and precision–recall curves, with commentary on what they reveal about model behavior.

iv. A discussion of which technique(s) performed best and under what conditions.

v. A discussion of how each technique affected the precision–recall tradeoff, and the implications for model selection.

5.28 Implement multinomial logistic regression in Python from scratch using the following steps:

(a) Define the softmax function, which converts raw logits (class scores) into probabilities.

(b) Implement the cross-entropy loss function to measure the difference between the predicted probabilities and the true labels.

(c) Compute the gradient of the loss function with respect to the model parameters.

(d) Implement gradient descent (either batch or stochastic) to optimize the model parameters.

(e) Train the model on the MNIST dataset and plot both the cross-entropy loss and accuracy on the validation set during the learning process.

(f) Evaluate the model's performance on the test set.

(g) Compare the performance of your implementation with Scikit-Learn's multinomial logistic regression implementation (using `LogisticRegression` with `multi_class='multinomial'`).

5.29 Compare three different strategies for multi-class classification using logistic regression: one-vs-rest, one-vs-one, and multinomial logistic regression using the Wine Quality dataset. The target variable in the dataset represents wine quality on a scale from 3 to 8. Use the following steps:

(a) Load the dataset using the `fetch_openml` function:

```
from sklearn.datasets import fetch_openml
X, y = fetch_openml('wine-quality-red', return_X_y=True,
    as_frame=True)
```

(b) Conduct exploratory data analysis (EDA) to understand the distribution of features and classes.

(c) Split the dataset into training and test sets (e.g., 80–20 split).

(d) Standardize the continuous features for better performance in logistic regression.

(e) Train three different logistic regression models:

- One-vs-rest (OvR) logistic regression.
- One-vs-one (OvO) logistic regression.
- Multinomial logistic regression.

Record the training time for each model.

(f) Evaluate each model's performance on the test set using accuracy. Record the prediction times for each model.

(g) Use Matplotlib to create comparative charts showing:

i. Training times for the three models.
ii. Prediction times for the three models.
iii. Accuracies of the three models on the test set.

(h) Analyze the results:

i. Which model had the fastest training and prediction times?
ii. Which model achieved the highest test accuracy?
iii. Discuss the tradeoffs between model complexity, speed, and predictive performance.

5.30 In this exercise, you will explore different techniques for handling class imbalance in a multi-class setting using the Wine Quality dataset.

(a) Load the dataset using the `fetch_openml` function (see previous exercise).

(b) Conduct exploratory data analysis (EDA) to visualize the class distribution. What do you observe about the class imbalance?

(c) Baseline model:

i. Train a multinomial logistic regression model using the original imbalanced dataset.
ii. Evaluate the performance using precision, recall, F1 score, and AUC. Pay particular attention to the precision and recall for the minority classes.

(d) SMOTE (Synthetic Minority Oversampling Technique):

i. Apply SMOTE to the training data to generate synthetic samples for the minority classes. Experiment with different resampling ratios.
ii. Train a multinomial logistic regression model on the balanced dataset.
iii. Evaluate the model using the same metrics as before. Compare the performance of the SMOTE-enhanced model with the baseline.

(e) Cost-sensitive learning:

i. Implement a cost-sensitive logistic regression model by adjusting the class weights (using the `class_weight` parameter).
ii. Train the cost-sensitive model on the original dataset.
iii. Evaluate the model using the same metrics as before. Compare the results with both the baseline and SMOTE-enhanced models.

(f) Bonus (other SMOTE variations):

i. Experiment with other variations of SMOTE, such as Borderline SMOTE or SMOTE-ENN, using the `imbalanced-learn` package.

ii. Try under-sampling techniques or combining over-sampling and under-sampling.

iii. Compare the results with SMOTE and cost-sensitive learning.

(g) Analysis:

i. Analyze how each technique affected the performance of the model on the minority classes.

ii. Which technique gave the best tradeoff between precision and recall for the minority classes?

iii. Write a short report summarizing your findings.

5.31 In this exercise, you will explore how variations in the parameters of softmax regression affect the decision boundaries between the classes in a multi-class setting. Consider a softmax regression model used for classifying data points into three classes $(0, 1, 2)$. The class scores (logits) are computed as:

$$\begin{aligned} z_0 &= w_{0,0} + w_{0,1}x_1 + w_{0,2}x_2, \\ z_1 &= w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2, \\ z_2 &= w_{2,0} + w_{2,1}x_1 + w_{2,2}x_2. \end{aligned}$$

The softmax function is then applied to (z_0, z_1, z_2) to obtain the predicted class probabilities. Assume a fixed initial weight matrix $w_{i,j}$ (e.g., small random values), and use this initialization consistently throughout the exercise.

(a) Single parameter variation:

i. Choose one parameter from each score function (e.g., $w_{0,1}, w_{1,2}, w_{2,0}$). Systematically increase and decrease its value while keeping the other parameters constant.

ii. For each variation, plot the decision boundaries using a grid over the input space $x_1, x_2 \in [-5, 5]$ with a step size of 0.1.

iii. Describe how the decision boundary changes as a result of the parameter variation. Specifically, note any shifts, rotations, or changes in the shape of the boundary lines.

(b) Simultaneous parameter variation:

i. Select a pair of parameters from different score functions (e.g., $w_{0,1}$ and $w_{2,1}$). Vary them simultaneously, considering cases where both parameters increase together, decrease together, or change in opposite directions.

ii. Plot the resulting decision boundaries as above, and analyze how these combined changes affect the shape, orientation, or location of the boundaries.

iii. Explain how these parameter interactions might affect overall accuracy, precision, and recall, especially in regions near the boundaries.

(c) Geometric interpretation:

i. Based on your observations from the above experiments, provide a geometric interpretation of the softmax decision boundaries. Explain how the parameters $w_{i,j}$ influence the orientation, slope, and position of the these boundaries in the feature space.

ii. Discuss how changing these parameters affects the relative spacing between boundaries and what this implies for the model's classification behavior.

(d) Practical recommendations:

i. Based on your findings, provide practical recommendations for adjusting the model parameters to achieve specific classification goals (e.g., maximizing separation between two specific classes).

ii. Discuss when manual parameter tuning may lead to overfitting or underfitting and how techniques such as regularization can mitigate these risks.