Classification: Logistic Regression - A Comprehensive Guide with Mathematical Derivation and Python Code

Rafiq Islam

2024-10-07

Table of contents

Introduction

Logistic Regression is a popular classification algorithm used for binary and multi-class classification problems. Unlike Linear Regression, which is used for regression problems, Logistic Regression is used to predict categorical outcomes. In binary classification, the output is either 0 or 1, and the relationship between the input features and the outcome is modeled using a logistic function (also called the sigmoid function).

What is Logistic Regression?

Logistic Regression is a type of regression analysis used when the dependent variable is categorical. In binary logistic regression, the output can have only two possible outcomes (e.g., 0 or 1, pass or fail, spam or not spam). Logistic Regression works by modeling the probability of an event occurring based on one or more input features. It estimates the probability that a

given input belongs to a particular category (0 or 1) using the **logistic function (sigmoid function)**.

The Sigmoid Function

The sigmoid function maps any real-valued number to a value between 0 and 1, making it ideal for modeling probabilities.

The sigmoid function is given by the formula:

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

Where:

- z is the input to the sigmoid function (in logistic regression, $z = \mathbf{x} \cdot \theta$)
- e is the base of the natural logarithm

The output of the sigmoid function is interpreted as the probability $P(y = 1|X)$.

Logistic Regression Model

In Logistic Regression, the hypothesis is modeled as:

$$h_\theta(X) = \frac{1}{1 + e^{-\theta^T X}}$$

Where:

- X is the input feature vector
- θ is the parameter vector (weights)

Cost Function for Logistic Regression

Unlike Linear Regression, which uses the Mean Squared Error (MSE) as the cost function, Logistic Regression uses **log loss** or **binary cross-entropy** as the cost function, as the output is binary (0 or 1).

So, basically we model probability from the given data. In other words, we can write

$$\mathbb{P}(y = 1 \text{ or } 0| \text{ given } X) = p(\mathbf{x}) = \sigma(\mathbf{x} \cdot \theta) = \frac{1}{1 + e^{-\mathbf{x} \cdot \theta}}$$

$$\implies p_\theta(\mathbf{x}) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d)}}$$

$$\implies p_\theta(\mathbf{x}) = \begin{cases} p_\theta(\mathbf{x}) & \text{if } y = 1 \\ 1 - p_\theta(\mathbf{x}) & \text{if } y = 0 \end{cases}$$

Where, $\theta, \mathbf{x} \in \mathbb{R}^{d+1}$ and d is the dimension of the data. For single data vector $\mathbf{x}$ the binary cross-entropy function can be written as

$$l(\theta) = y p_\theta(\mathbf{x}) + (1 - y)(1 - p_\theta(\mathbf{x}))$$

Since we have n of those i.i.d data vectors therefore, we can write

$$L(\theta) = \prod_{i=1}^{n} (y_i p_\theta(\mathbf{x_i}) + (1 - y_i)(1 - p_\theta(\mathbf{x_i})))$$

Since our goal is to minimize the loss, we need to perform derivatives of the loss function. Therefore, to change from the product form to addition form we take negative log of the above expression

$$\ell(\theta) = -\log L(\theta) = -\sum_{i=1}^{n} y_i \log p_\theta(\mathbf{x}) + (1 - y_i) \log (1 - p_\theta(\mathbf{x}))$$

For the ease of calculation, let's rewrite the above equation in terms of m and b where $m \in \mathbb{R}^d = (\theta_1, \theta_2, \cdots, \theta_d)^T$ and $b \in \mathbb{R}$.

$$\ell(\theta) = -\sum_{i=1}^{n} y_i \log p_{m,b}(\mathbf{x}) + (1 - y_i) \log (1 - p_{m,b}(\mathbf{x}))$$

Where:

- n is the number of training examples

- m is the number of features
- $y^{(i)}$ is the true label of the i^{th} example
- b is the bias for the i^{th} example

Gradient Descent

To minimize the cost function and find the optimal values for θ, we use **gradient descent**. We start from the last form of the loss function and convert this to a form that is easy to take the partial dervivatives.

$$
\begin{aligned}
\ell(\theta) &= -\sum_{i=1}^{n} y_i \log p_{m,b}(\mathbf{x}) + (1-y_i)\log\left(1-p_{m,b}(\mathbf{x})\right) \\
&= -\sum_{i=1}^{n} y_i \log\left(\sigma(mx_i+b)\right) + (1-y_i)\log\left(1-\sigma(mx_i+b)\right) \\
&= -\sum_{i=1}^{n} y_i \log\left(\sigma(mx_i+b)\right) + (1-y_i)\log\left(\sigma(-(mx_i+b))\right); \quad \text{Since } 1-\sigma(x)=\sigma(-x) \\
&= -\sum_{i=1}^{n} y_i \left[\log\left(\sigma(mx_i+b)\right) - \log\left(\sigma(-(mx_i+b))\right)\right] + \log\left(-\sigma(mx_i+b)\right) \\
&= -\sum_{i=1}^{n} y_i \log\left(\frac{\sigma(mx_i+b)}{\sigma(-(mx_i+b))}\right) + \log\left(-\sigma(mx_i+b)\right) \\
&= -\sum_{i=1}^{n} y_i (mx_i+b) + \log\left(\sigma(-(mx_i+b))\right); \quad \text{Since } \frac{\sigma(x)}{-\sigma(x)} = e^x
\end{aligned}
$$

Now we again use the beautiful features of the sigmoid function

$$
\begin{aligned}
\frac{d\sigma(x)}{dx} = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) &= \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\cdot\frac{e^{-x}}{1+e^{-x}} \\
&= \sigma(x)\left(1-\frac{1}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x)) \\
&= \sigma(x)\sigma(-x)
\end{aligned}
$$

Finally, we are ready to take the partial derivatives of the loss function with respect to m and b,

$$\begin{aligned}
\frac{\partial \ell}{\partial m} &= -\sum_{i=1}^{n} y_i x_i + \frac{1}{\sigma(-(mx_i+b))} \frac{d}{dx}(\sigma(-(mx_i+b))) \\
&= -\sum_{i=1}^{n} y_i x_i + \frac{1}{\sigma(-(mx_i+b))} \sigma(-(mx_i+b))\sigma(mx_i+b)(-x_i) \\
&= -\sum_{i=1}^{n} x_i(y_i - \sigma(mx_i+b)) \\
&= \sum_{i=1}^{n} x_i(p_{m,b}(x_i) - y_i) = \sum_{i=1}^{n} x_i(\hat{y}_i - y_i) \\
&= \mathbf{x_i} \cdot (\hat{\mathbf{y}}_\mathbf{i} - \mathbf{y_i})
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial \ell}{\partial b} &= -\sum_{i=1}^{n} y_i + \frac{1}{\sigma(-(mx_i+b))} \frac{d}{dx}(\sigma(-(mx_i+b))) \\
&= -\sum_{i=1}^{n} y_i - \frac{1}{\sigma(-(mx_i+b))} \sigma(-(mx_i+b))\sigma(mx_i+b) \\
&= \sum_{i=1}^{n} p_{m,b}(x_i) - y_i = \sum_{i=1}^{n} \hat{y}_i - y_i \\
&= \hat{\mathbf{y}}_i - \mathbf{y}_i
\end{aligned}$$

Using this gradient, we update the parameter vector θ iteratively:

$$\theta_{j+1} := \theta_j - \alpha \nabla \ell(\theta_j)$$

Where:

- α is the learning rate
- $\nabla \ell(\theta_j)$ is the partial derivative of the cost function with respect to θ_j and

$$\nabla \ell(\theta) = \begin{bmatrix} \sum_{i=1}^{n} \hat{y}_i - y_i \\ \sum_{i=1}^{n} x_i(\hat{y}_i - y_i) \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{y}}_i - \mathbf{y}_i \\ \mathbf{x_i} \cdot (\hat{\mathbf{y}}_\mathbf{i} - \mathbf{y_i}) \end{bmatrix} = X^T(\hat{\mathbf{y}}_i - \mathbf{y}_i) = X^T(\sigma(X\vec{\theta}) - \vec{y})$$

Python Code Implementation from Scratch

Here's how to implement Logistic Regression from scratch in Python. We will use two different forms for our class

```
import numpy as np

class LogisticRegression1:
    def __init__(self, learning_rate = 0.1, n_iterations = 1000):
        """
        Hyper Parameters
        - learning_rate: learning rate; float; default 0.01
        - n_itearations: number of iterations; int; default 1000
        Model Parameters
        - weights: weights of the features; float or int
        - bias: bias of the model; float or int
        """
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def _sigmoid(self, x):
        return 1/(1+np.exp(-x))

    def fit(self, X,y):
        """
        n_sample = number of samples in the data set: the value n
        n_features = number of features or the dimension of the data set: the value d
        """
        n_sample,n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iterations):
            linear = np.dot(X, self.weights) + self.bias
            pred = self._sigmoid(linear)

            dw = (1/n_sample)* np.dot(X.T,(pred-y))
            db = (1/n_sample) * np.sum(pred-y)

            self.weights = self.weights - self.learning_rate * dw
            self.bias = self.bias - self.learning_rate * db
```

```
    def predict(self, X):
        linear = np.dot(X, self.weights) + self.bias
        predicted_y = self._sigmoid(linear)
        class_of_y = [0 if y<=0.5 else 1 for y in predicted_y]
        return class_of_y
```

Now let's use this using the `scikit-learn` breast cancer data set.

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

b_cancer = load_breast_cancer()
X, y = b_cancer.data, b_cancer.target
X_train, X_test, y_train, y_test = train_test_split(X,y, random_state=123, stratify=y, test_s

clf1 = LogisticRegression1(learning_rate=0.01)
clf1.fit(X_train, y_train)
predicted_y = clf1.predict(X_test)
print(np.round(accuracy_score(predicted_y, y_test),2))
```

```
0.92
```

Now lets compare this with the standard `scikit-learn` library

```
from sklearn.linear_model import LogisticRegression

clf2 = LogisticRegression()
clf2.fit(X_train, y_train)
predicted_y = clf2.predict(X_test)
print(np.round(accuracy_score(predicted_y, y_test),2))
```

```
0.96
```

References

- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.
- Gradient descent is a widely used optimization technique in machine learning.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Nocedal, J., & Wright, S. (2006). *Numerical Optimization* (2nd ed.). Springer.
- Regularization techniques like L2 (Ridge) and L1 (Lasso) are commonly used in logistic regression to prevent overfitting.
- Ng, A. (2004). *Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance*. ICML Proceedings.
- Friedman, J., Hastie, T., & Tibshirani, R. (2010). *Regularization Paths for Generalized Linear Models via Coordinate Descent*. Journal of Statistical Software, 33(1), 1-22.
- The extension of logistic regression to multiclass classification via the softmax function is part of the core material for understanding classification tasks.
- Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
- VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media.
- Raschka, S., & Mirjalili, V. (2017). *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*. Packt Publishing.
