Logistic Regression

September 17, 2024

1 Logistic Regression

2 Detailed Explanation of the Logistic Regression Algorithm

2.1 Introduction to Logistic Regression

Logistic regression is a linear model for binary classification problems. Unlike linear regression, which predicts continuous values, logistic regression predicts the probability that a given input point belongs to a certain class (often labeled as 0 or 1).

2.1.1 1. Sigmoid Function

The sigmoid function is a crucial part of logistic regression. It transforms the linear combination of features into a probability value between 0 and 1. The sigmoid function is defined as:

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

where:

- $z = X\theta$, which is a linear combination of input features X and the parameter vector θ.
- e is the base of the natural logarithm.

The sigmoid function ensures that our output is always between 0 and 1, which can be interpreted as a probability.

2.1.2 2. Hypothesis Function

The hypothesis function for logistic regression uses the sigmoid function to estimate the probability that a given input belongs to class 1. Mathematically, this is represented as:

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

where:

- $h_\theta(x)$ represents the predicted probability of the input belonging to class 1.
- X is the feature matrix (with m samples and n features).
- θ is the parameter vector we aim to learn.

2.1.3 3. Cost Function (Loss Function)

To find the optimal parameters θ, we minimize a cost function, which measures how well our model's predictions match the actual data. The cost function used in logistic regression is the **negative log-likelihood** (also known as the binary cross-entropy loss):

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

where:

- m is the total number of training examples.
- y_i is the actual label (0 or 1) for the i-th training example.
- $\hat{y}_i = h_\theta(x^{(i)}) = \sigma(x^{(i)} \cdot \theta)$ is the predicted probability for the i-th training example.

The cost function penalizes incorrect predictions. For example: - When $y_i = 1$ and $\hat{y}_i$ is close to 0, the cost will be high. - When $y_i = 0$ and $\hat{y}_i$ is close to 1, the cost will also be high.

2.1.4 4. Gradient Descent for Logistic Regression

To minimize the cost function $J(\theta)$, we use an optimization algorithm called **gradient descent**. Gradient descent iteratively updates the parameter vector θ in the direction that reduces the cost function the most.

The update rule for θ is:

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

where:

- α is the learning rate (step size).
- $\nabla J(\theta)$ is the gradient of the cost function with respect to θ.

The gradient of the cost function for logistic regression is:

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

where:

- X is the feature matrix.
- $h_\theta(X)$ is the vector of predicted probabilities for all samples.
- y is the vector of actual labels.

2.1.5 5. Newton's Method for Optimization

Instead of using standard gradient descent, our implementation uses **Newton's method**, which converges faster by incorporating second-order derivative information.

Newton's update rule is:

$$\theta := \theta - H^{-1} \nabla J(\theta)$$

where:

- H is the Hessian matrix, which is the second derivative (or curvature) of the cost function with respect to θ.

Hessian Matrix Calculation The Hessian matrix H for logistic regression is given by:

$$H = \frac{1}{m} X^T D X + \lambda I$$

where:

- $D = \text{diag}(p_i(1 - p_i))$ is a diagonal matrix where each diagonal element is $p_i(1 - p_i)$, the variance of the Bernoulli random variable for the predicted probability p_i.
- λI is a regularization term added to the diagonal to prevent singular matrices, where λ is a small positive constant, and I is the identity matrix.

2.1.6 6. Regularization

To prevent overfitting, we add a regularization term to the Hessian matrix. Regularization helps by penalizing overly complex models (i.e., models with large weights) and encourages simpler models that generalize better.

The regularization term λI controls the magnitude of the penalty:

- A small λ results in minimal regularization.
- A large λ makes the model simpler (i.e., smaller weight values).

2.1.7 7. Stopping Criteria

The gradient descent (or Newton's method) iterations continue until:

1. The change in the parameter vector θ is less than a small threshold ϵ.
2. A maximum number of iterations is reached.

Mathematically, we stop when:

$$\|\alpha \Delta \theta\| < \epsilon$$

where:

- $\| \cdot \|$ denotes the norm (magnitude) of the vector.
- ϵ is a small positive threshold value for convergence.

2.1.8 8. Making Predictions

Once the model is trained (i.e., θ is optimized), we can use it to make predictions on new data points.

To predict the probability that a new data point x belongs to class 1, we compute:

$$\hat{y} = \sigma(x \cdot \theta)$$

To convert this probability into a binary class label (0 or 1), we use a threshold of 0.5:

- If $\hat{y} \geq 0.5$, predict 1 (class 1).
- If $\hat{y} < 0.5$, predict 0 (class 0).

2.1.9 Conclusion

This detailed explanation provides a comprehensive understanding of the logistic regression algorithm, including its mathematical foundation, optimization techniques, and practical considerations. By understanding these details, you will be better equipped to interpret and extend your logistic regression model in the future.

[1]:
```
#Import necessary libraries and preprocess data.

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

# Load the dataset
df = pd.read_csv(r'C:\Users\Machine_Learning\Downloads\archive\Iris.csv')

# Preprocess the dataset
X = df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].
 ↪values
y = df['Species'].values

# Encode labels
le = LabelEncoder()
y = le.fit_transform(y)

# Split the dataset into training and validation sets
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

2.2 Regression

3 Logistic Regression Implementation from Scratch

3.1 Overview

This notebook contains a full implementation of a logistic regression model built from the ground up. The logistic regression algorithm is a popular method used for binary classification tasks. This model uses maximum likelihood estimation to fit a decision boundary between two classes and is trained using gradient descent.

3.1.1 Class Definition

We define a Python class, `Logistic Regression`, with methods for initialization, the sigmoid function, fitting the model to training data, computing the loss function, and predicting new data points.

3.1.2 Sigmoid Function

The sigmoid function, $\sigma(z) = \frac{1}{1+e^{-z}}$, is used to map predicted values to probabilities between 0 and 1.

3.1.3 Fit Method

The fit method implements the core of the logistic regression algorithm using Newton's method for optimization. x: Feature matrix of shape (m, n), where m is the number of samples and n is the number of features. y: Target vector of shape $(m,)$, containing binary labels (0 or 1).

3.1.4 Initialization

```
Adds an intercept term (a column of ones) to x.
Initializes the weight vector theta with small random values if not provided.
```

3.1.5 Gradient Descent Loop

The model is trained using a loop that iteratively adjusts the weights to minimize the cost function. predictedProbability: The predicted probability of class 1 for each sample using the current model. errorTerm: The difference between the actual and predicted probabilities. gradient: The partial derivative of the log-likelihood function with respect to theta, used to update theta in the direction that reduces the loss.

The Hessian matrix is computed as:

$$H = \frac{1}{m} X^T D X + \lambda I$$

where:

- $D = \text{diag}(p_i(1 - p_i))$ is a diagonal matrix with predicted probabilities along the diagonal.
- λI is a small regularization term added to stabilize the matrix inversion.

3.1.6 Updating the Parameters

`delta_theta: Computed change in parameters. We use np.linalg.solve to solve for delta_theta di`
`If the Hessian is singular, we use the pseudoinverse instead.`

3.1.7 Convergence Check

The algorithm stops when the parameter updates are smaller than a given threshold eps

3.1.8 Loss Function

The loss function for logistic regression is the negative log-likelihood:

$$\text{Loss} = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

where: - y_i is the actual label for the i-th example. - $\hat{y}_i$ is the predicted probability for the i-th example. - m is the total number of training examples.

3.1.9 Predict Method

The predict method outputs class labels based on the predicted probabilities. Adds an intercept term to x. Computes predicted probabilities and thresholds them at 0.5 to determine class labels (0 or 1).

3.2 Conclusion

This implementation of logistic regression uses Newton's method for fast convergence by utilizing both gradient and second-order derivative (Hessian) information. It includes handling of singular matrices using the pseudoinverse, and verbose output to monitor training progress.

[2]:
```
class LogisticRegression:
    def __init__(self, step_size=1, max_iter=10000, eps=1e-5, theta_0=None,␣
↪verbose=True):
        self.theta = theta_0
        self.step_size = step_size
        self.max_iter = max_iter
        self.eps = eps
        self.verbose = verbose

    #step_size: The learning rate, which controls the step size during gradient␣
↪descent.
    #max_iter: The maximum number of iterations for the gradient descent␣
↪algorithm.
    #eps: The convergence threshold for stopping the gradient descent.
    #theta_0: Initial parameter values (weights). If not provided, they are␣
↪initialized randomly.
    #verbose: A boolean that determines whether to print progress messages␣
↪during training.
```

```
  def sigmoid(self, z):
      return 1 / (1 + np.exp(-np.clip(z, -500, 500))) #np.clip: Prevents␣
↪overflow by clipping values of z to be within [-500, 500] before applying␣
↪the exponential function.

  def fit(self, x, y):
      m, n = x.shape
      if self.theta is None:
          self.theta= np.random.randn(n + 1) * 0.01
          x = np.concatenate([np.ones((m, 1)), x], axis=1)


      for i in range(self.max_iter):
          predictedProbability = self.sigmoid(np.dot(x, self.theta))
          errorTerm = y-predictedProbability

          gradient = (1/m)*np.dot(x.T,errorTerm)
          hessian = ((1/m) * (x.T @ np.diag(predictedProbability * (1 -␣
↪predictedProbability)) @ x)) + (np.eye(x.shape[1]) * 1e-4)

          try:
              delta_theta = np.linalg.solve(hessian, gradient)
          except np.linalg.LinAlgError:
              print("Singular matrix encountered, using pseudoinverse")
              delta_theta = np.linalg.pinv(hessian).dot(gradient)

          self.theta -= self.step_size * delta_theta

          if np.linalg.norm(self.step_size * delta_theta) < self.eps:
              break

          if self.verbose and i % 100 == 0:
              print(f"Iteration {i}, Loss: {self.loss(x, y)}")

  def loss(self, x, y):
      m = len(y)
      predictions = self.sigmoid(np.dot(x, self.theta))
      return -np.sum(y * np.log(predictions + 1e-8) + (1 - y) * np.log(1 -␣
↪predictions + 1e-8)) / m

  def predict(self, x):
      m,n=x.shape
      x = np.concatenate([np.ones((m, 1)), x], axis=1)
      probabilities = self.sigmoid(np.dot(x,self.theta))
      return (probabilities>=0.5).astype(int)
```

3.3 Training and predicting and evaluating the model

[3]:
```
# Train the logistic regression model
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Predict on the validation set
y_pred = clf.predict(X_valid)

# Evaluate the model
accuracy = accuracy_score(y_valid, y_pred)
print(f"Validation Accuracy: {accuracy:.2f}")
```

```
Iteration 0, Loss: 5.0385672518496545
Iteration 100, Loss: 18.267175071002764
Iteration 200, Loss: 18.267175071002764
Iteration 300, Loss: 18.267175071002764
Iteration 400, Loss: 18.267175071002764
Iteration 500, Loss: 18.267175071002764
Iteration 600, Loss: 18.267175071002764
Iteration 700, Loss: 18.267175071002764
Iteration 800, Loss: 18.267175071002764
Iteration 900, Loss: 18.267175071002764
Iteration 1000, Loss: 18.267175071002764
Iteration 1100, Loss: 18.267175071002764
Iteration 1200, Loss: 18.267175071002764
Iteration 1300, Loss: 18.267175071002764
Iteration 1400, Loss: 18.267175071002764
Iteration 1500, Loss: 18.267175071002764
Iteration 1600, Loss: 18.267175071002764
Iteration 1700, Loss: 18.267175071002764
Iteration 1800, Loss: 18.267175071002764
Iteration 1900, Loss: 18.267175071002764
Iteration 2000, Loss: 18.267175071002764
Iteration 2100, Loss: 18.267175071002764
Iteration 2200, Loss: 18.267175071002764
Iteration 2300, Loss: 18.267175071002764
Iteration 2400, Loss: 18.267175071002764
Iteration 2500, Loss: 18.267175071002764
Iteration 2600, Loss: 18.267175071002764
Iteration 2700, Loss: 18.267175071002764
Iteration 2800, Loss: 18.267175071002764
Iteration 2900, Loss: 18.267175071002764
Iteration 3000, Loss: 18.267175071002764
Iteration 3100, Loss: 18.267175071002764
Iteration 3200, Loss: 18.267175071002764
Iteration 3300, Loss: 18.267175071002764
Iteration 3400, Loss: 18.267175071002764
```

```
Iteration 3500, Loss: 18.267175071002764
Iteration 3600, Loss: 18.267175071002764
Iteration 3700, Loss: 18.267175071002764
Iteration 3800, Loss: 18.267175071002764
Iteration 3900, Loss: 18.267175071002764
Iteration 4000, Loss: 18.267175071002764
Iteration 4100, Loss: 18.267175071002764
Iteration 4200, Loss: 18.267175071002764
Iteration 4300, Loss: 18.267175071002764
Iteration 4400, Loss: 18.267175071002764
Iteration 4500, Loss: 18.267175071002764
Iteration 4600, Loss: 18.267175071002764
Iteration 4700, Loss: 18.267175071002764
Iteration 4800, Loss: 18.267175071002764
Iteration 4900, Loss: 18.267175071002764
Iteration 5000, Loss: 18.267175071002764
Iteration 5100, Loss: 18.267175071002764
Iteration 5200, Loss: 18.267175071002764
Iteration 5300, Loss: 18.267175071002764
Iteration 5400, Loss: 18.267175071002764
Iteration 5500, Loss: 18.267175071002764
Iteration 5600, Loss: 18.267175071002764
Iteration 5700, Loss: 18.267175071002764
Iteration 5800, Loss: 18.267175071002764
Iteration 5900, Loss: 18.267175071002764
Iteration 6000, Loss: 18.267175071002764
Iteration 6100, Loss: 18.267175071002764
Iteration 6200, Loss: 18.267175071002764
Iteration 6300, Loss: 18.267175071002764
Iteration 6400, Loss: 18.267175071002764
Iteration 6500, Loss: 18.267175071002764
Iteration 6600, Loss: 18.267175071002764
Iteration 6700, Loss: 18.267175071002764
Iteration 6800, Loss: 18.267175071002764
Iteration 6900, Loss: 18.267175071002764
Iteration 7000, Loss: 18.267175071002764
Iteration 7100, Loss: 18.267175071002764
Iteration 7200, Loss: 18.267175071002764
Iteration 7300, Loss: 18.267175071002764
Iteration 7400, Loss: 18.267175071002764
Iteration 7500, Loss: 18.267175071002764
Iteration 7600, Loss: 18.267175071002764
Iteration 7700, Loss: 18.267175071002764
Iteration 7800, Loss: 18.267175071002764
Iteration 7900, Loss: 18.267175071002764
Iteration 8000, Loss: 18.267175071002764
Iteration 8100, Loss: 18.267175071002764
Iteration 8200, Loss: 18.267175071002764
```

```
Iteration 8300, Loss: 18.267175071002764
Iteration 8400, Loss: 18.267175071002764
Iteration 8500, Loss: 18.267175071002764
Iteration 8600, Loss: 18.267175071002764
Iteration 8700, Loss: 18.267175071002764
Iteration 8800, Loss: 18.267175071002764
Iteration 8900, Loss: 18.267175071002764
Iteration 9000, Loss: 18.267175071002764
Iteration 9100, Loss: 18.267175071002764
Iteration 9200, Loss: 18.267175071002764
Iteration 9300, Loss: 18.267175071002764
Iteration 9400, Loss: 18.267175071002764
Iteration 9500, Loss: 18.267175071002764
Iteration 9600, Loss: 18.267175071002764
Iteration 9700, Loss: 18.267175071002764
Iteration 9800, Loss: 18.267175071002764
Iteration 9900, Loss: 18.267175071002764
Validation Accuracy: 0.33
```