Comparison of Custom Logistic Regression Model with scikit-learn Model

Pavan Sekhar Naidu Routhu

22B1028

1 Mathematics

Logistic regression is used for binary classification. The probability that a given input $\mathbf{x}$ belongs to a certain class is modeled using the logistic function:

$$h(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b) \tag{1}$$

where $\sigma(z)$ is the sigmoid function defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

1.1 Cost Function

The cost function for logistic regression is derived from the likelihood function. The negative log-likelihood (cross-entropy) for binary classification is:

$$J(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h(\mathbf{x}^{(i)}) + (1 - y^{(i)})\log(1 - h(\mathbf{x}^{(i)}))\right] \tag{3}$$

where m is the number of training examples, $y^{(i)}$ is the actual label, and $h(\mathbf{x}^{(i)})$ is the predicted probability.

1.2 Gradient Descent

To minimize the cost function, we use gradient descent. The gradients of the cost function with respect to $\mathbf{w}$ and b are:

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)\mathbf{x}^{(i)} \tag{4}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right) \tag{5}$$

2 Custom Logistic Regression Model Implementation

The custom logistic regression model is implemented in Python using NumPy for matrix operations. The model is trained using gradient descent as detailed above.

2.1 Model Code

Listing 1: Custom Logistic Regression Model Implementation

```
import numpy as np

class custom_model():
  def __init__(self, X, learning_rate = 0.1, num_iters = 10000):
    self.lr = learning_rate
    self.num_iters = num_iters
    self.w = np.zeros(X.shape[1])
    self.b = 0
    self.m, self.n = X.shape

  def train(self, X, y):
    for i in range(self.num_iters):
      z = np.dot(X, self.w) + self.b
      a = self.sigmoid(z)
      dw = (1/self.m) * np.dot(X.T, (a - y))
      self.w -= self.lr * dw
      db = (1/self.m) * np.sum(a - y)
      self.b -= self.lr * db

  def predict(self, X):
    z = np.dot(X, self.w) + self.b
    a = self.sigmoid(z)
    y_pred = np.where(a > 0.5, 1, 0)
    return y_pred

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

  def accuracy(self, y_pred, y):
    return np.sum(y_pred == y) / len(y)
```

3 Dataset Description

A synthetic dataset is generated using the `make_blobs` function from scikit-learn, consisting of two features and two centers.

Listing 2: Dataset Generation

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=100, n_features=2, centers=2, random_state=0)
```

The dataset is split into training and test sets with an 80-20 ratio.

4 Training and Prediction with the Custom Model

The custom model is trained on the training dataset and used to make predictions on the test set.

Listing 3: Training and Prediction with Custom Model

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random

model = custom_model(X_train)
model.train(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy score from custom model:", model.accuracy(y_pred, y_test))
```

5 Training and Prediction with scikit-learn Model

The scikit-learn logistic regression model is trained and tested for comparison.

Listing 4: Training and Prediction with scikit-learn Model

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy score from scikit-learn model:", model.score(X_test, y_test))
```

6 Results and Comparison

The accuracy of both models is calculated and compared. The custom model achieved an accuracy of **90%**, while the scikit-learn model achieved an accuracy of **90%**.