Experiment Report

Title: Analysis of Logistic Regression Model for ICU Patient Classification

Introduction

Logistic regression is the process whereby we create probabilities instead of using specific values. We can achieve this by putting data values into the sigmoid function. This will generate a probability distribution between 0 and 1.

$$\hat{y} = \frac{1}{1 + e^{-wx+b}}$$

$$s(x) = \frac{1}{1 + e^{-x}}$$

Fig 1. Sigmoid function and regression equation

In logistic regression, we will be using cross-entropy method to obtain the error function. Then we will perform gradient descent, which means we need to calculate the gradient of the error function in terms of the weight and bias. Gradient descent strategy shows at a specific point of the graphic function, the direction we should go to minimize the error. The updates of the parameters (weight and bias) will be done by subtracting the value of gradient multiply by the learning rate from the current value of the weight. Learning rate is a parameter which decides how fast we should go during gradient descend. A learning rate that is too small might cause the training to be to slow and reach the minimum error on time. If it is too large, then it might skip past the minimum error. Thus, deciding a decent learning rate is important for our training.

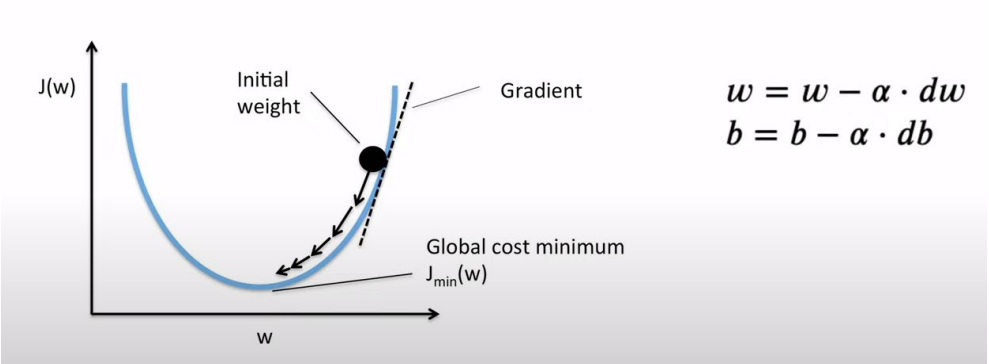

Fig 2. Gradient descent

Methodology

Training:

1. We will first initialize both weight and bias as zero.
2. Then we predict result by using our function in figure 1.
3. Calculate the error.
4. Use gradient descent to figure out new weight and bias.
5. Repeat for n times.

Testing:

1. Put the values from the data point into the equation in figure 1.
2. Choose the label based on the probability.

Experimental Setup

We will be using Jupyter Notebook (Python 3) to implement this.

We will first initialize all variables. Variable “lr” denotes the learning rate, “n_iters” denotes the number of iterations.

```
class LogisticRegression(BaseEstimator, ClassifierMixin):
    def __init__(self, lr=0.001, n_iters=1000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
```

Figure 3. Initialization

The fit function will be training the actual model. Basically, it is all the equations we listed before.

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)
    self.bias = 0

    for _ in range(self.n_iters):
        linear_pred = np.dot(X, self.weights) + self.bias
        predictions = sigmoid(linear_pred)

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

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

Figure 4. Fit function

The predict function runs the prediction on the test set.

```
def predict(self, X):
    linear_pred = np.dot(X, self.weights) + self.bias
    y_pred = sigmoid(linear_pred)
    class_pred = [0 if y <= 0.5 else 1 for y in y_pred]
    return class_pred
```

Figure 5. Predict function

Results and conclusion

For the learning rate, I have used a for loop to train the model repeatedly with increasing learning rate and found out the maximum accuracy is 79.1% on the training dataset with learning rate set to 0.5. (I have removed the code for this section). Just to add on, we could also not preset a learning rate and the accuracy was just 1-2% lower.

The final accuracy we had on the data set is about 78.85%

```
Training Error: 20.26%
Cross-Validation Errors: 21.50%
Test Error: 21.15%
Accuracy on the test dataset: 78.85%
```

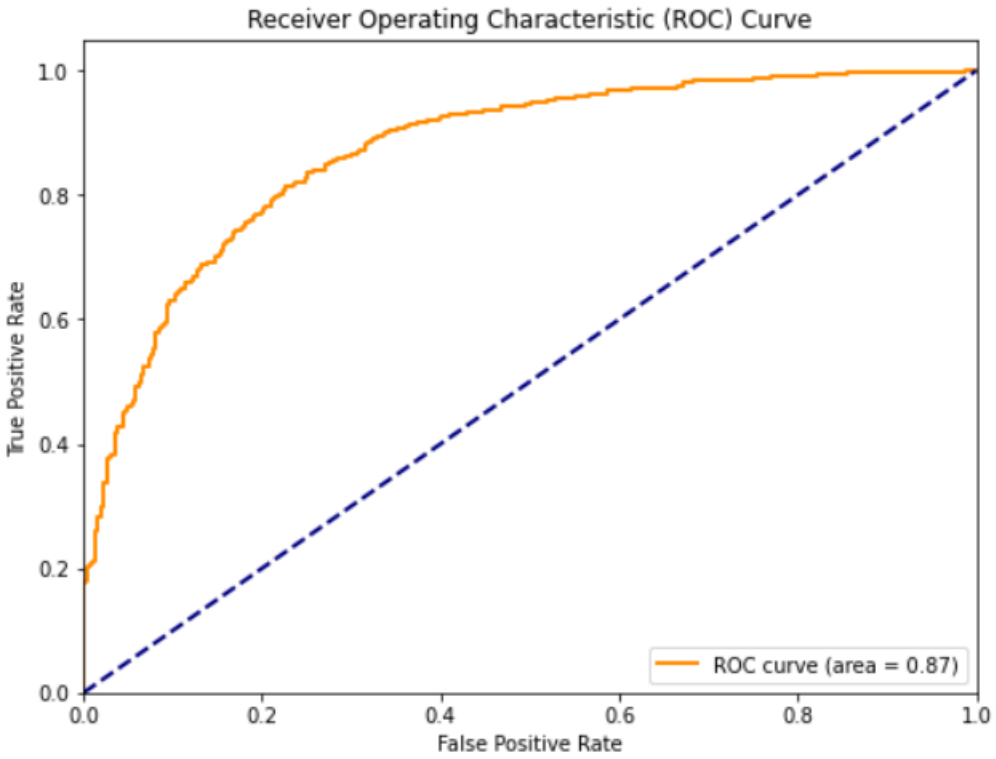

Fig 6. Experiment result