Logistic Regression with Rust

Arpit Bhat *501808* Archit Bhonsle *501810* Shalmali Kulkarni *501835*

1 Problem Statement

We aim to create a desktop application using GTK, a toolkit to create GUI apps that guides use through the a simplified version of the machine learning pipeline. The algorithm we'll be using is logistic regression which is used in various simple classification tasks.

2 Dataset

The dataset we've chosen is reduced version of a larger dataset with 76 attributes found here: `https://archive.ics.uci.edu/ml/datasets/Heart+Disease`. This reduced dataset has 14 attributes (described in detail in table 1) and one target variable which indicates whether heart disease is present or not. The reduced dataset has numeric, discrete, continuous and binary attributes all of which can be represented as numbers which in our case are 64-bit floating point values. This is an assumption that helps us optimize our model and skip certain steps like mapping nominal attributes to unique numbers.

Column	Attribute
age	Age in years
sex	Sex of person: 0 is female, 1 is male
cp	Chest pain type
trestbps	Resting blood pressure (in mm Hg)
chol	Serum cholesterol in mg/dl
fbs	1 if fasting blood sugar $>$ 120 mg/dl else 0
restecg	Resting electrocardiograph results
thalch	Maximum heart rate achieved (in mm Hg)
exang	1 if exercise induced angina else 0
oldpeak	ST depression induced by exercise relative to rest
slope	The slope of the peak exercise ST segment
ca	Number of major vessels (0-3) colored by fluoroscope
thal	3 = normal; 6 = fixed defect; 7 = reversable defect
target	1 if heart disease is present 0 otherwise

Table 1: Attributes in the heart dataset

3 Model

The model we use here is logistic regression. It is a model used for binary classification problems like the heart disease problem we are tackling here. A fundamental part of

this model is the sigmoid function which maps an arbitary number of real values to a probability between 0 and 1:

$$\frac{1}{1+e^{-x}}$$

The cost function we use is:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(h_\theta(x^{(i)})\right) + \left(1-y^{(i)}\right)\log\left(1-h_\theta(x^{(i)})\right)\right)$$

And thus in gradient descent, which is basically applying this repeatedly:

$$\theta_j := \theta_j - \alpha\frac{\delta}{\delta\theta_j}J(\theta)$$

becomes after resolving the derivative using calculus:

$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}x_j^{(i)}\left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

4 Creating the model and testing

The core code for the model is encapsulated inside the following `forward_backward` function:

```
fn forward_backward(
    weights: &Array2<f64>,
    bias: &f64,
    x_train: &Array2<f64>,
    y_train: &Array2<f64>,
) -> (f64, Array2<f64>, f64) {
    // forward
    let y_head: Array2<f64> = sigmoid(weights.t().dot(x_train)
        .mapv(|x| x + bias));
    let loss = (y_train * &(y_head.mapv(|z| z.ln()))
        + &((y_train.mapv(|z| 1. - z)) * &y_head.mapv(|z| (1. - z).ln())))
        .mapv(|z| -1. * z);
    let cost = loss.sum() / x_train.ncols() as f64;

    // backward
    let d_weights = (x_train.dot(&(&y_head - y_train).t()))
        .mapv(|z| z / x_train.ncols() as f64);
    let d_bias = (&y_head - y_train).sum() / x_train.ncols() as f64;

    (cost, d_weights, d_bias)
}
```

Here, `y_head` is $h_\theta(x^{(i)})$ or the hypothesis. `loss` is $J(\theta)$ and `cost` is the average loss. This comprises the "forward" step. In the backward step we calculate the change in the weights, `d_weights` and the change in bias as `d_bias`.

These values are then used by the `update` function which tweaks the weights and the biases for the specified `iterations`. Thus, the model is trained.

```
fn update(
    weights: Array2<f64>,
    bias: f64,
    x_train: Array2<f64>,
    y_train: Array2<f64>,
    learning_rate: f64,
    iterations: usize,
) -> (Vec<f64>, Array2<f64>, f64) {
    let mut costs: Vec<f64> = Vec::new();
    let mut weights = weights;
    let mut bias = bias;

    for _ in 0..iterations {
        let (cost, d_weight, d_bias) = forward_backward(
            &weights, &bias, &x_train, &y_train);
        weights -= &d_weight.mapv(|x| learning_rate * x);
        bias -= learning_rate * d_bias;

        costs.push(cost);
    }

    (costs, weights, bias)
}
```

5 Running the application

The application has three pages. Each page has represents one section of the machine learning pipeline.

1. In the first step, as shown in figure 1, we select our dataset. We can we review it in the tabular view and inspect it's various attributes.

2. After that we perform preprocessing on this data. As one may observe in the tabular view some attributes go in the high 100s while some are simple binary values, 0 and 1. If we give this data to the model it may cause problems. To counter this we normalize the data. This is demonstrated in figure 2.

3. In the third step, as shown in the figure 3, we actually train our model. We specify the learning rate and the number of iterations. Once trained a graph of the model's cost as the it goes through various iterations is graphed. Based on this we can evaluate the model. Once trained we can test the model against the testing data. Then the various metrics like accuracy, precision, recall and f1-score are shown.

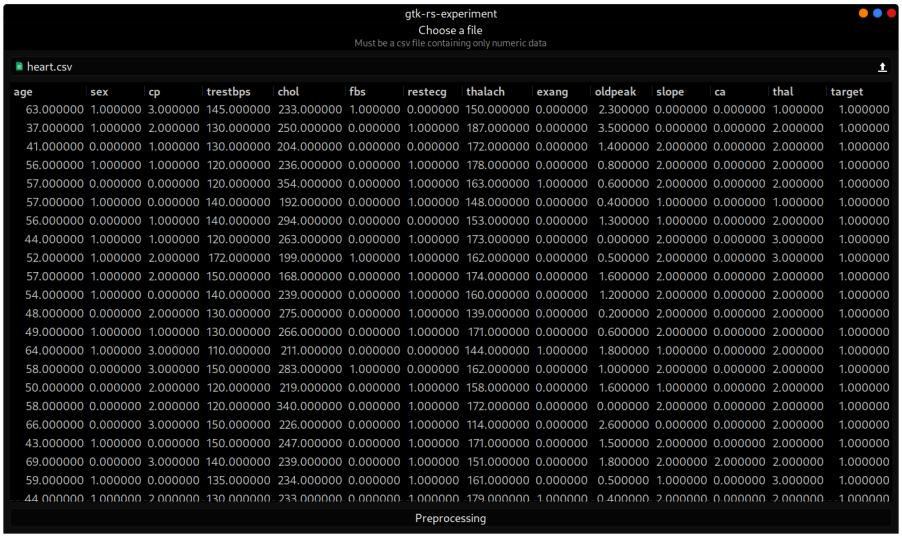

Figure 1: Choose Page

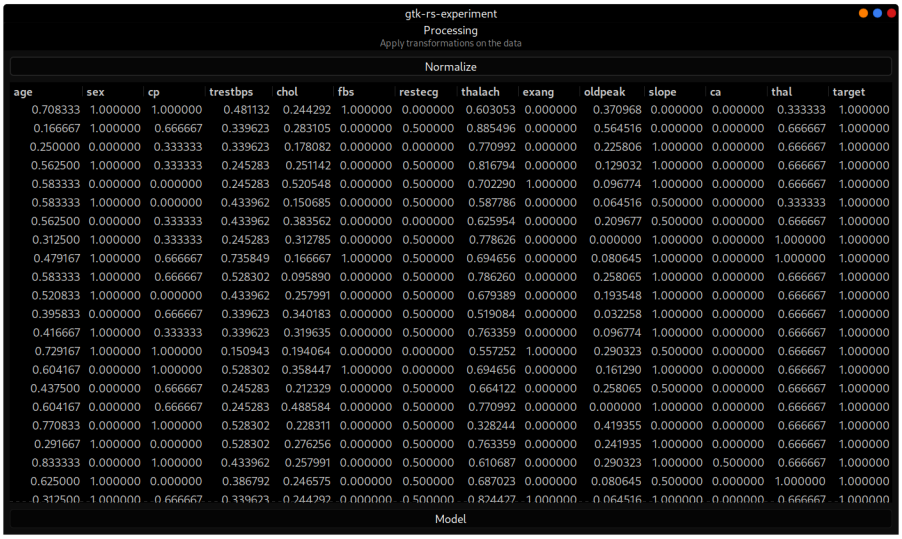

Figure 2: Preproccessing Page

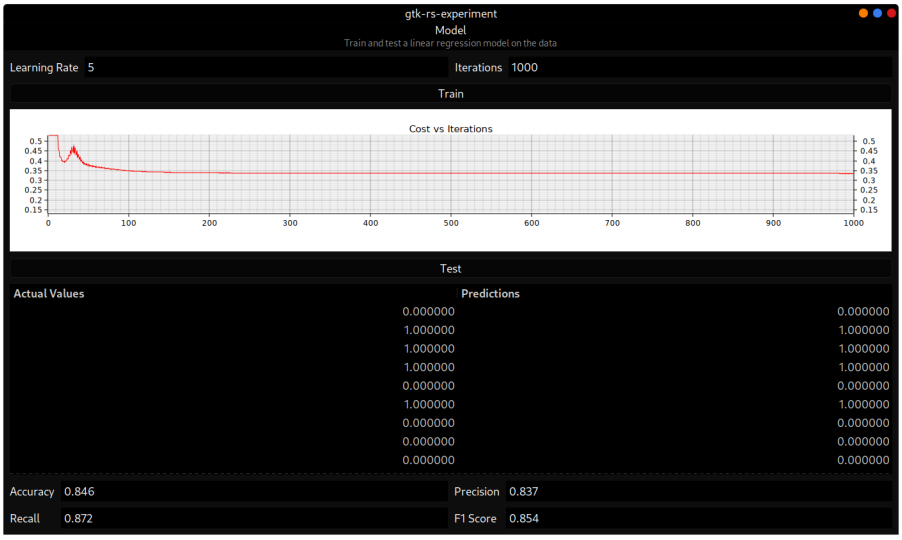

Figure 3: Model Page

6 Analyzing the results

The results of various values of the hyperparameters, learning rate and iterations are show in table 2.

LR	Iterations	Accuracy	Precision	Recall	F1-Score
0.1	100	79.1%	90.2%	76.7%	82.9%
0.1	1000	80.2%	90.2%	78.0%	83.6%
1	100	80.1%	90.2%	78.0%	83.6%
1	1000	83.5%	92.2%	81.0%	86.2%
10	100	74.2%	68.6%	83.3%	75.3%
10	1000	76.9%	70.6%	85.7%	77.4%

Table 2: Metrics for various hyperparameters

As we can see, the model with the learning rate of 1 and 1000 iterations performs the best. The learning rate of 0.1 is too slow and it takes 1000 to match the performance of the (1, 1000) model. The learning rate of 10 is too fast which is also apparent from it's cost vs iterations graph. Ideally, we'd like a mechanism that reduces the learning rate whenever the cost stops decreasing. This is known as an "adaptive learning rate".

7 Setup

Regardless of the operating system you're on you might need a dataset to test this on. The app only supports datasets with numeric columns so other datasets might crash the app.

1. Go to this link.
2. Right click on the page and select "Save as" and save it in the location of you choice.

7.1 Linux

1. Go to this link.
2. Right click on the page and select "Save as" and save it in the location of you choice.
3. Double click on the executable.

7.2 Windows

1. Go here.
2. Input this link: `https://github.com/ArchitBhonsle/gtk-rs-experiment/windows-package` and press Enter. This will download the folder.
3. In this folder double click on the `gtk-rs-experiment.exe`.