Binary Classification using Logistic Regression

Ankit Saha
AI21BTECH11004

29 March 2022

Report

I. Problem Statement

We are given a red wine data set. This is a classification problem. The goal is to find a model that can classify wine as of:

1) high quality (1) or
2) low quality (0)

II. Exploratory Data Analysis

We first count the number of samples that are labelled as ‘good quality’ and ‘bad quality’ in the entire data set.

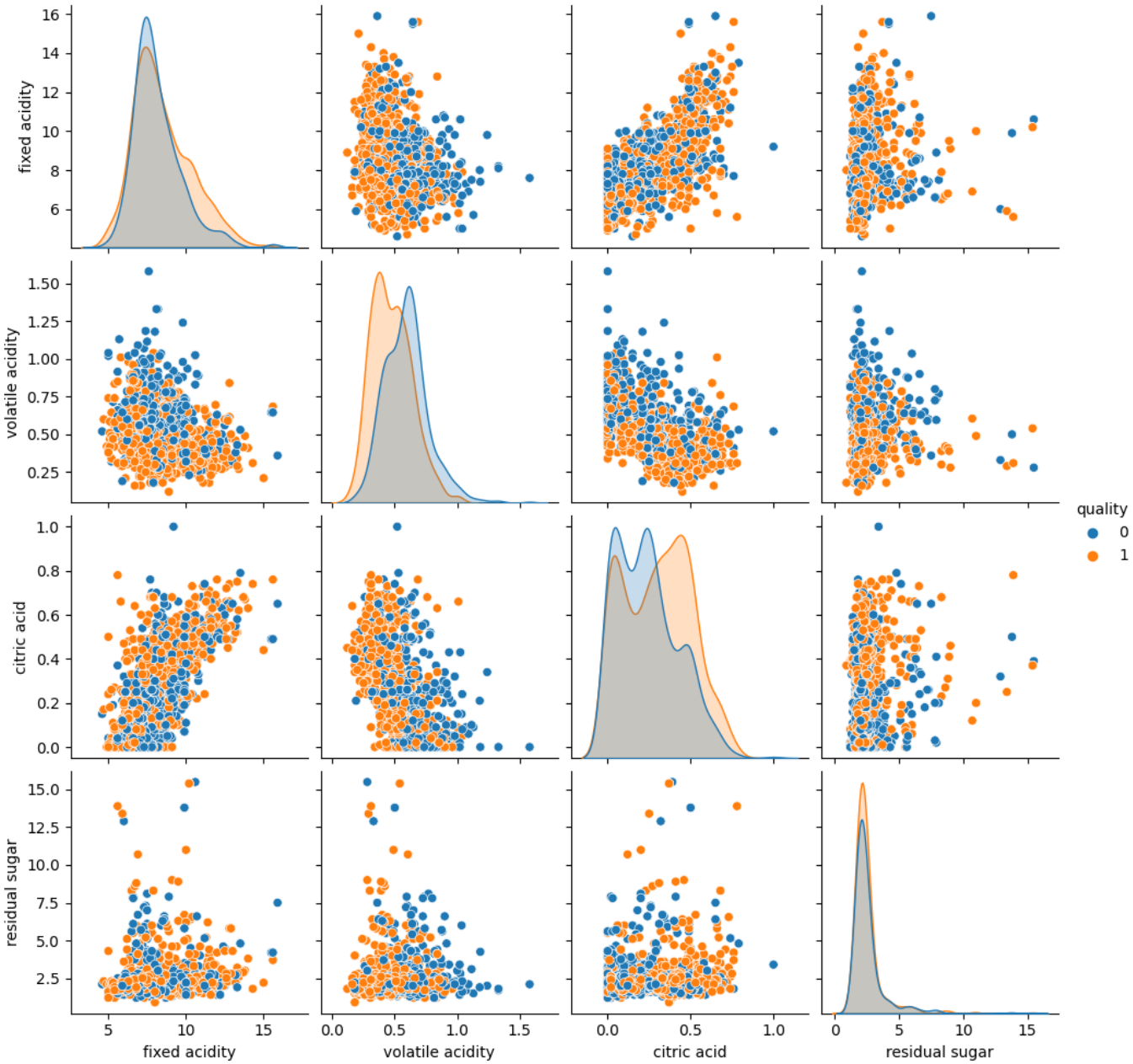
```
good_count = np.sum(df['quality'])
bad_count = total_count - good_count
```

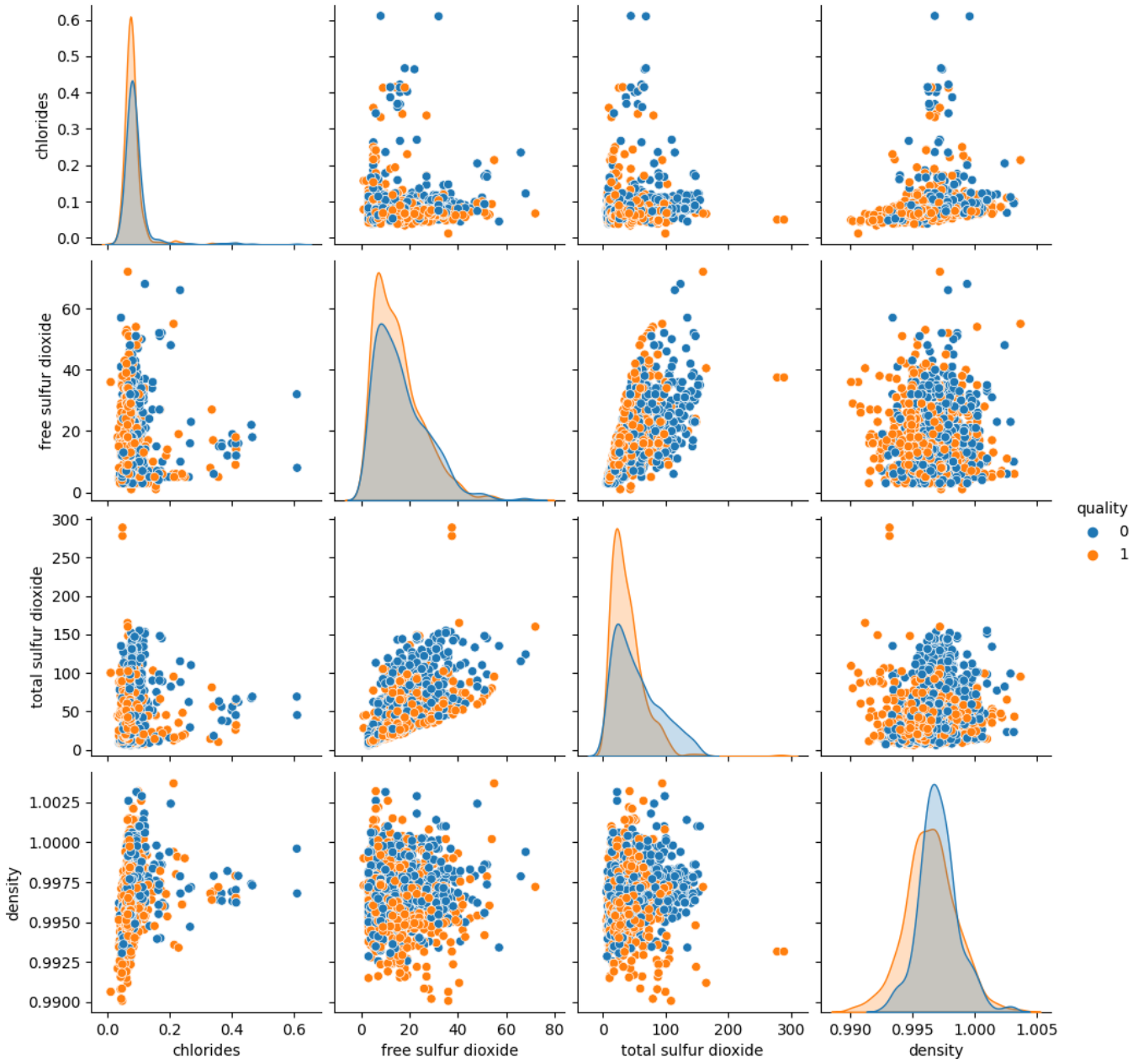
where `total_count` is the total number of samples present in the data set.

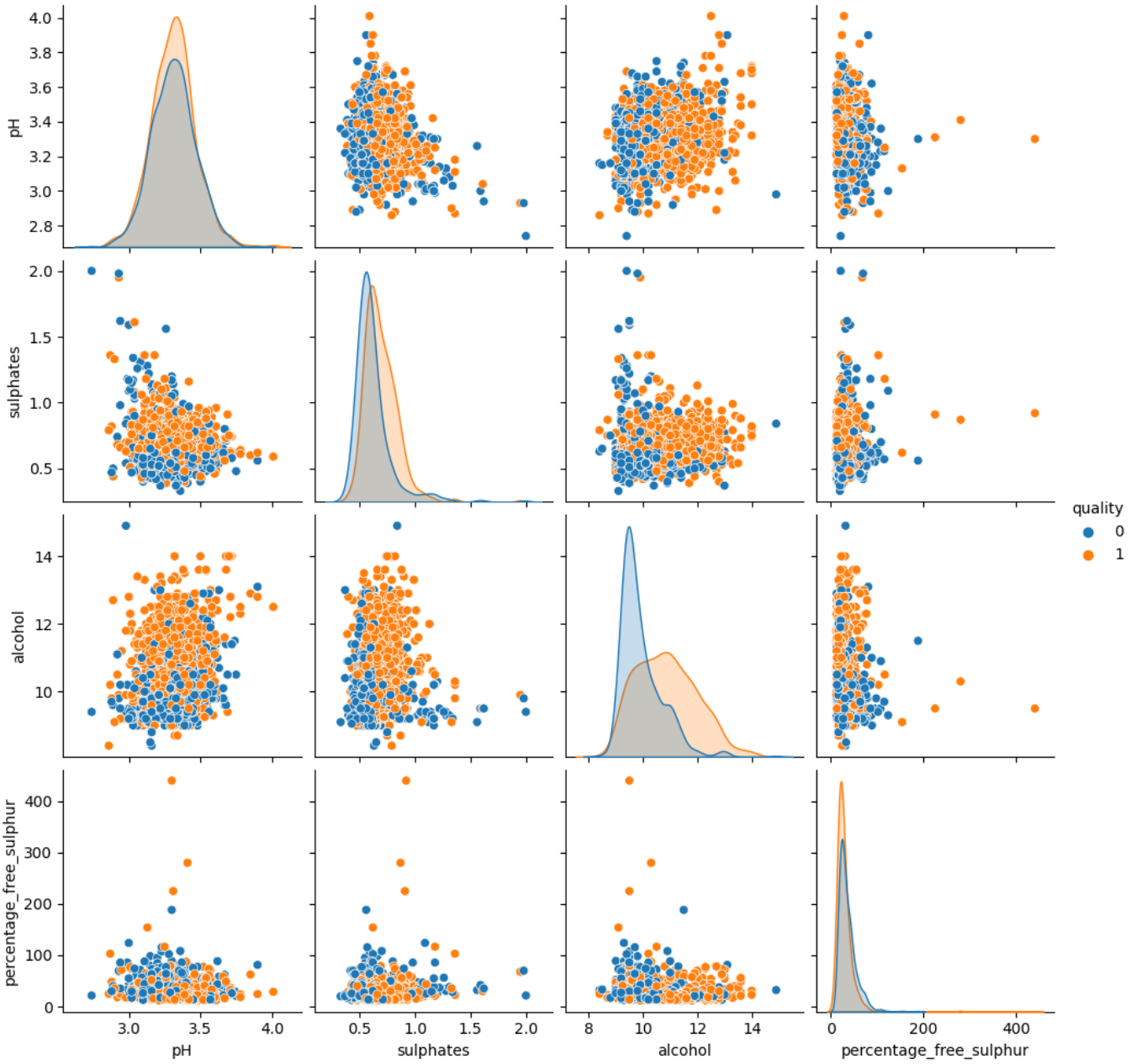
This works because the values in the `'quality'` column are 1 for ’good quality’ and 0 for ’bad quality’. Thus, only the good quality samples contribute to the sum.

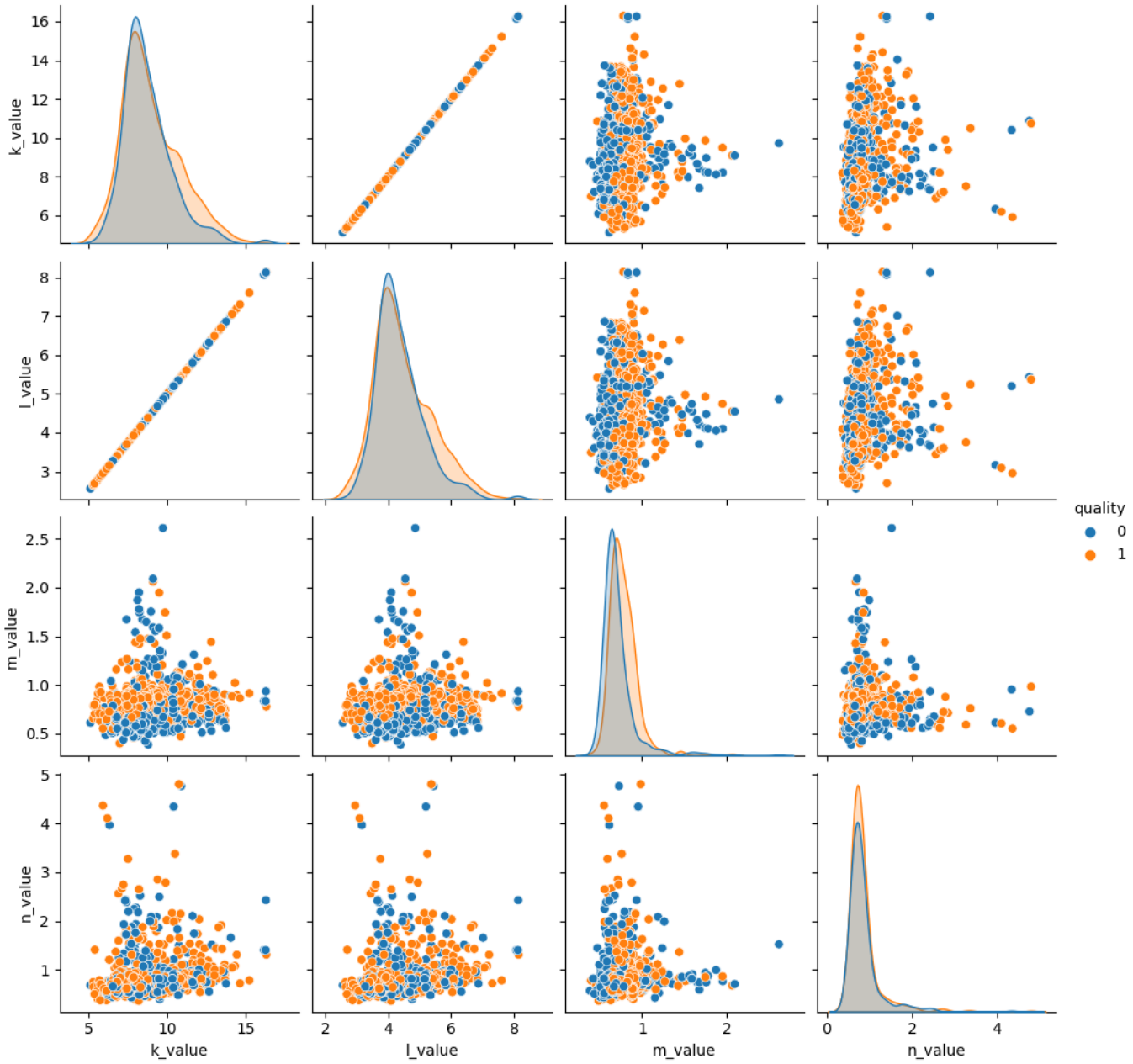
We then use the `pairplot` function from the `seaborn` module to plot pairwise relations between a set of 4 variables at a time. We also set the keyword argument `hue` to `'quality'` so that we can visualize the contrast between good and bad samples for these comparisons.

```
ls1 = list(df)[:4] # ['volatile acidity', 'citric acid', 'residual sugar', 'chlorides']
ls1.append('quality')
sc1 = sns.pairplot(df.loc[:, ls1], hue='quality')
plt.savefig('figs/fig-1.png')
plt.show()
```


fixed acidity
volatile acidity
citric acid
residual sugar
quality
0
1


chlorides
free sulfur dioxide
total sulfur dioxide
density
quality
0
1


pH
sulphates
alcohol
percentage_free_sulphur
quality
0
1



Observe that the plot of `'k_value'` vs `'l_value'` is a straight line, which suggests that they are linearly dependent on each other.

III. Training the model

First, we randomly split our data set into training and test data sets in the ratio 80 : 20

```
training_indices = random.sample(range(total_count), int(0.8 * total_count))
```

`random.sample` chooses a random subset (of indices) of size = $\lfloor 0.8 \times 1598 \rfloor = 1279$ from the set $\{0, 1, 2, \ldots, 1599\}$. We consider the samples (without the `'quality'` column) of the data set corresponding to these indices to be our training data set and the rest as our test data set. We store the values of the 'quality' column corresponding to these indices as `'training_labels'` and `'test_labels'` respectively.

The Logistic Regression Model

We use a logistic regression model for this classification problem.
We are given $N = 1599$ labelled samples $\{(\mathbf{x}_i, y_i)\}, \quad i = 1, 2, \ldots, N$
For this problem, we choose the sigmoid function to model our probability.

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

$$1 - S(x) = 1 - \frac{1}{1 + e^{-x}} \tag{2}$$

$$= \frac{1 + e^{-x} - 1}{1 + e^{-x}} \tag{3}$$

$$= \frac{e^{-x}}{1 + e^{-x}} \tag{4}$$

$$= \frac{1}{1 + e^{x}} \tag{5}$$

$$= S(-x) \tag{6}$$

Our goal is to find a weight vector $\mathbf{w}$ that maximizes the log-likelihood

$$J = \frac{1}{N} \sum_{i=1}^{N} y_i \log(S(z)) + (1 - y_i) \log(1 - S(z)) \tag{7}$$

where $z_i = \mathbf{w} \cdot \mathbf{x}_i$

$$J = \frac{1}{N} \sum_{i=1}^{N} y_i \log(S(z_i)) + (1 - y_i) \log(1 - S(z_i)) \tag{8}$$

$$= \frac{1}{N} \sum_{i=1}^{N} y_i \log\left(\frac{1}{1 + e^{-z_i}}\right) + (1 - y_i) \log\left(\frac{1}{1 + e^{z_i}}\right) \tag{9}$$

$$= -\frac{1}{N} \sum_{i=1}^{N} y_i \log\left(1 + e^{-z_i}\right) + (1 - y_i) \log\left(1 + e^{z_i}\right) \tag{10}$$

$$\implies -J = \frac{1}{N} \sum_{i=1}^{N} y_i \ \mathrm{log1p}\left(e^{-z_i}\right) + (1 - y_i) \ \mathrm{log1p}\left(e^{z_i}\right) \tag{11}$$

$$\tag{12}$$

where log1p(x) = log(1 + x)
Maximizing J is equivalent to minimizing J
log1p(x) is being used, because in Python, `log1p(x)` is more precise than `log(1 + x)` when $x \ll 1$

Gradient of the log-likelihood function

We will now compute the gradient of the negative of the log-likelihood function, which we are going to need later

For the sake of simplicity, let $N = 2$ and $\mathbf{x}_i$ be two-dimensional vectors $\begin{pmatrix} x_{ia} & x_{ib} \end{pmatrix}$

$$-J = \frac{1}{N}\sum_{i=1}^{N} y_i \log\left(1 + e^{-z}\right) + (1 - y_i)\log\left(1 + e^{z}\right) \tag{13}$$

$$= \frac{1}{2}\left(y_1 \log\left(1 + e^{-z_1}\right) + (1 - y_1)\log\left(1 + e^{z_1}\right) + y_2 \log\left(1 + e^{-z_2}\right) + (1 - y_2)\log\left(1 + e^{z_2}\right)\right) \tag{14}$$

Now, let $\mathbf{w} = \begin{pmatrix} w_a & w_b \end{pmatrix}$

$$\nabla(-J) = \begin{pmatrix} \frac{\partial(-J)}{\partial w_a} & \frac{\partial(-J)}{\partial w_b} \end{pmatrix} \tag{15}$$

$$= \frac{1}{2}\left(y_1 \frac{1}{1 + e^{-z_1}}(-e^{-z_1})\frac{\partial z_1}{\partial w_a} + (1 - y_1)\frac{1}{1 + e^{z_1}} e^{z_1}\frac{\partial z_1}{\partial w_a} + \cdots \quad \cdots\right) \tag{16}$$

Now, $z_1 = \mathbf{w} \cdot \mathbf{x}_1 = w_a x_{1a} + w_b x_{1b}$

$$\implies \frac{\partial z_1}{\partial w_a} = x_{1a}, \qquad \frac{\partial z_1}{\partial w_b} = x_{1b} \tag{17}$$

Similarly,

$$\implies \frac{\partial z_2}{\partial w_a} = x_{2a}, \qquad \frac{\partial z_2}{\partial w_b} = x_{2b} \tag{18}$$

$$\nabla(-J) = \frac{1}{2}\left(-y_1 S(-z_1)x_{1a} + (1 - y_1)S(z_1)x_{1a} + \cdots \quad \cdots\right) \tag{19}$$

Now,

$$-y_1 S(-z_1)x_{1a} + (1 - y_1)S(z_1)x_{1a} = -y_1(1 - S(z_1))x_{1a} + (1 - y_1)S(z_1)x_{1a} \tag{20}$$

$$= -y_1 x_{1a} + y_1 x_{1a} S(z_1) + x_{1a} S(z_1) - y_1 x_{1a} S(z_1) \tag{21}$$

$$= x_{1a}(S(z_1) - y_1) \tag{22}$$

Thus,

$$\nabla(-J) = \frac{1}{2}\begin{pmatrix} x_{1a}(S(z_1) - y_1) + x_{2a}(S(z_2) - y_2) & x_{1b}(S(z_1) - y_1) + x_{2b}(S(z_2) - y_2) \end{pmatrix} \tag{23}$$

$$= \frac{1}{2}\begin{pmatrix} x_{1a} & x_{2a} \\ x_{1b} & x_{2b} \end{pmatrix}\begin{pmatrix} S(z_1) - y_1 \\ S(z_2) - y_2 \end{pmatrix} \tag{24}$$

$$= \frac{1}{2}\mathbf{X}^T\begin{pmatrix} S(z_1) - y_1 \\ S(z_2) - y_2 \end{pmatrix} \tag{25}$$

where $\mathbf{X} = \begin{pmatrix} x_{1a} & x_{1b} \\ x_{2a} & x_{2b} \end{pmatrix}$ is the training data set

Therefore, the code for calculating the gradient is given by

```
def gradient(X, y, W):
      z = np.dot(X, W)
      return np.dot(X.T, sigmoid(z) - y) / len(X)
```

Steepest Descent Algorithm

We use the steepest descent algorithm to compute the optimal $\mathbf{w}$ that minimizes the cost function C (in our case $C = -J$)

The algorithm is given by:

1) Initialize $\mathbf{w} = \mathbf{w}^{(0)}$
2) Compute the gradient $g^{(k)} = \nabla C(\mathbf{w}^{(k)})$
3) At the next iteration, set $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - t_k g^{(k)}$ where t_k is the step size (constant in our model)
4) Keep iterating over k until $\nabla C(\mathbf{w}^{(k)})$ is sufficiently small

The code for this is:

```
def steepest_descent(X, y):
      W = np.zeros(len(X.columns)) # Initial guess
      epsilon = 1e-6
      step_size = 1e-2
      g_prev = gradient(X, y, W)
      W -= step_size * g_prev
      g_curr = gradient(X, y, W)

      while (abs(np.linalg.norm(g_curr) - np.linalg.norm(g_prev)) > epsilon):
          g_prev = g_curr
          W -= step_size * g_curr
          g_curr = gradient(X, y, W)

      return W
```

For a fixed step size, it is possible that $\nabla C(\mathbf{w}^{(k)})$ will just keep oscillating around the minima. In order to prevent that, we have set the break condition so that if the norm of $\nabla C(\mathbf{w}^{(k)})$ doesn't change significantly, i.e., is negligibly small, we break out of the loop.

This is because, when $\nabla C(\mathbf{w}^{(k)})$ oscillates around the minima, $\nabla C(\mathbf{w}^{(k)})$ doesn't approach zero, but its norm remains roughly the same.

All that's left now is training this model on our training data set and finding the optimal weight vector. We do that by executing the following code:

```
      weight_vector = steepest_descent(training_set, training_labels)
```

The following are the evaluation metrics obtained after running this model on our test data set:

$$\text{Accuracy} = 0.603125 \tag{26}$$

$$\text{Precision} = 0.581081081081081 \tag{27}$$

$$\text{Recall/Sensitivity} = 0.9828571428571429 \tag{28}$$

$$\text{Specificity} = 0.14482758620689656 \tag{29}$$