James Zhao
HW4
A15939512

1 1

1.1 a

Maximum likelihood estimate is the average y: $\frac{1+3+4+6}{4} = 3.5$
MSE: $\frac{(1-3.5)^2+(3-3.5)^2+(4-3.5)^2+(6-3.5)^2}{4} = \frac{13}{4} = 3.25$

1.2 b

Predictions: $(1,1),(1,1),(4,4),(4,4)$
Truths: $(1,1),(1,3),(4,4),(4,6)$
MSE: $\frac{(1-1)^2+(1-3)^2+(4-4)^2+(4-6)^2}{4} = \frac{8}{4} = 2$

1.3 c

From lecture:

$$y = ax + b, b = \bar{y} - a\bar{x}, a = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$$

$$\bar{x} = \frac{1+1+4+4}{4} = 2.5, \bar{y} = \frac{1+3+4+6}{4} = 3.5$$

$$a = \frac{(1-2.5)(1-3.5) + (1-2.5)(3-3.5) + (4-2.5)(4-3.5) + (4-2.5)(6-3.5)}{(1-2.5)^2 + (1-2.5)^2 + (4-2.5)^2 + (4-2.5)^2} = \frac{9}{9} = 1$$

$$b = \bar{y} - a\bar{x} = 3.5 - 1 * 2.5 = 1$$

Answer:

$$y = x + 1$$

2 2

2.1 a

$$\frac{d}{ds}L(s) = \frac{d}{ds}\frac{1}{n}\sum_{i=1}^{n}(x_i - s)^2$$

$$= \frac{1}{n}\sum_{i=1}^{n}\frac{d}{ds}(x_i - s)^2$$

$$= \frac{1}{n}\sum_{i=1}^{n} -2(x_i - s)$$

$$= -\frac{1}{n}\sum_{i=1}^{n} 2(x_i - s)$$

2.2 b

$$0 = -\frac{1}{n}\sum_{i=1}^{n} 2(x_i - s)$$

$$0 = -\frac{1}{n}\sum_{i=1}^{n} 2x_i + \frac{1}{n}\sum_{i=1}^{n} 2s$$

$$\frac{1}{n}\sum_{i=1}^{n} 2x_i = \frac{1}{n}\sum_{i=1}^{n} 2s$$

$$\sum_{i=1}^{n} 2x_i = \sum_{i=1}^{n} 2s$$

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} s$$

$$\sum_{i=1}^{n} x_i = ns$$

$$\frac{\sum_{i=1}^{n} x_i}{n} = s = E[x]$$

3 3

Total penalty is the penalty over every item. The loss functions can optionally be mean losses (multiply by $\frac{1}{n}$) too, which are functionally equivalent to optimize.

$$L(y, \hat{y}) = \sum_i |y_i - \hat{y}_i|$$

$$L(y, x, \theta) = \sum_i |y_i - \theta \cdot x_i|$$

$$L(y, x, w, b) = \sum_i |y_i - (w \cdot x_i + b)|$$

4 4

Let us assume that y and 1 are column-vectors (that is, they both have dimension $(n \times 1)$), and X has dimension $n \times d$

4.1 a

Sum of a vector is same as dot-product with a 1 vector

$$\frac{1}{n} 1^T y$$

4.2 b

$x^i \cdot x^j$ is the same as $x^i x^{jT}$, which is done automatically in standard matrix multiplication if we transpose the second matrix:

$$X X^T$$

4.3 c

This is similar to (a)

$$\frac{1}{n} 1^T X$$

4.4 d

Matrix multiplication (AB) computes the inner product of row i of matrix A and column j of matrix B in element (i, j). If we want element (i, j) to be the dot product of the i'th feature of x with the j'th feature of x (along dimension with size n), then we need row i of matrix A be an n-vector, and column j of matrix B to be an n vector. This is only achieved by transposing X:

$$\frac{1}{n} X^T X$$

5 5

Because $w \cdot 0 + b = c_0$, we must have it so $b = c_0$. $x^{(i)}$ is the ith standard basis vector, then we just need the i'th element of w to be $c_i - c_0$, since all the other terms are zero-ed anyways (the $-c_0$ is there because we need to "undo" the bias term). Repeating this for all positions, we get $w = (c_1 - c_0, ..., c_d - c_0), b = c_0$.

6 6

6.1 a

When the λ term is zero, the data is fit perfectly. Thus, the training loss is 0.

6.2 b

As λ increases, the norm of w_λ decreases because the model is "punished" for having a large weight vector.

6.3 c

As λ increases, $L(\lambda)$ increases. The model will learn a smaller parameterization of w due to the regularization, and emphasize more information the bias term, leading to predictions that are no longer 100% accurate. This causes the SSE error (without the Ridge regularization term) to increase.

6.4 d

As λ goes to infinity, the weight vector drops to zero. Thus, the model is entirely parameterized by bias, in which the value that yields the lowest loss is the mean (which we proved earlier). Thus, the loss approaches:

$$L(\infty) = \sum_{i=0}^{d} (y^i - b)^2, b = \frac{\sum_i^{d+1} y^i}{d+1}$$

$$= \sum_i y^{i2} - 2\sum_i by^i + \sum_i b^2$$

$$= \sum_i c^{i2} - 2(\frac{\sum_i c_i}{d+1})\sum_i c^i + (d+1)(\frac{(\sum_i c_i)^2}{(d+1)^2})$$

$$= \sum_{i=0}^{d} [c^{(i)2}] - \frac{(\sum_{i=0}^{d} c^{(i)})^2}{d+1}$$

7 7

7.1 a

My strategy is to do perform a Ridge Regression with strong regularization, and then sort by the absolute value of the weights descending. There should be a clearly observed 'gap' in the absolute value of the weights between the 10th highest and 11th highest weight.

7.2 b

Using 0-indexing: $[1, 2, 4, 6, 10, 12, 16, 18, 22, 26]$
Using 1-indexing: $[2, 3, 5, 7, 11, 13, 17, 19, 23, 27]$

8 8

8.1 a, b

To perform my partition, I generated a random permutation of indexes in [0,303] using a fixed seed of (123). The first 200 indexes correspond to indexes of the training points, and the remaining ones are the test points.

Afterwards, I trained 3 separate Logistic models on all 200 training points (with no regularization), each with different normalization methods. These were all trained using sklearn.linear_model.LogisticRegression(). Normalization statistics were only obtained from training data, and the same normalization statistics were used for normalizing training and test data.

To determine influential features, I sorted the weight values by their absolute value, and chose the 3 largest (and the feature those weights correspond to)

Method	**Formula**	**Test Error**
Z-score	$x_{i,j} = \frac{x_{i,j} - mean(x_{:,j})}{std(x_{:,j})}$	$13/103 = 0.1262$
Min-Max 0-1	$x_{i,j} = \frac{x_{i,j} - min(x_{:,j})}{max(x_{:,j}) - min(x_{:,j})}$	$13/103 = 0.1262$
None	$x_{i,j} = x_{i,j}$	$14/103 = 0.1359$

Weights:

- **Z-score**:
  - w: [-0.0366 -0.9783 0.8047 -0.4959 -0.3468 0.1503 0.1725 0.691 -0.4286 -0.7565 0.5111 -0.8314 -0.4907]
  - b: [-0.0234]
  - Influential features: [sex, ca, cp]
- **Min-Max 0-1**:
  - w: [-0.6315 -1.3791 1.6426 -1.1085 -0.6228 0.1735 0.5641 1.4912 -0.867 -1.7894 1.3386 -1.8582 -1.3364]
  - b: [1.4089]
  - Influential features: [ca, oldpeak, cp]
- **None**:
  - w: [0.0184 -1.5697 0.7578 -0.0242 -0.0055 0.1327 0.4142 0.0374 -0.6592 -0.6019 0.7472 -0.7584 -0.7095]
  - b: [0.0501]
  - Influential features: [sex, ca, cp]

8.2 c

When performing cross-validation, I first split the data, and then used the resulting training set to obtain statistics for normalizing both the validation and training sets. Using a random-seed state of (123) (which determines the permutation used for train/test splits), the following results are obtained:

Method	Fold1	Fold2	Fold3	Fold4	Fold5	Average-Fold Error	Q8b Test Error
Z-score	9/40	8/40	10/40	10/40	4/40	0.2050	0.1262
Min-Max 0-1	9/40	9/40	10/40	8/40	2/40	0.1975	0.1262
None	10/40	7/40	9/40	9/40	3/40	0.1950	0.1359

Random Seed State	Z-Score Avg-Error	Min-Max Avg-Error	No Normalization Avg-Error
111	0.1550	0.1575	0.1550
9999	0.1800	0.1725	0.1700
42	0.1850	0.1900	0.1900
122	0.1500	0.1500	0.1517

On average, the estimated test error use 5-fold cross validation tends to be higher than the test error in part 8b. This makes sense since we are training on less data (160 vs 200) which could potentially explain the difference in performance. It also has a high degree of variance based on what train/test split is used and how the folds are divided, which is exacerbated by the fact that we have a very small number of non-training samples (40 validation samples for 8c versus 103 test samples in 8b). This means that smaller variations in performance are essentially amplified by the law of large numbers. It is possible that the result I got in 8b (using seed 123) was exceptionally lucky, whereas many of the trials i tried in part 8c are closer to more typical results.

Additionally, it seems there is no perfect normalization method either; the permutation for the train/test split sometimes favors one method over the other.

In [27]:
```
import numpy as np, os
from pathlib import Path
```

Q7

(b)

- Using 0-indexing: [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
- Using 1-indexing: [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]

In [28]:
```
with open("data/HW4/mystery.dat", "r") as f:
  lines = np.array([list(map(float, l.strip().split(","))) for l in f.readlines() if no
print(lines.shape)

X = lines[:,:-1]
y = lines[:,[-1]]

print(X.shape, y.shape)
```

```
(101, 101)
(101, 100) (101, 1)
```

In [29]:
```
from sklearn import linear_model
import operator


for i in range(0, 40, 5):
  reg = linear_model.Ridge(alpha=i).fit(X, y)

  w, b = reg.coef_, reg.intercept_

  out = sorted(zip(list(w.flatten()), range(len(w.flatten()))), key = lambda x : -abs(x
  res = sorted(map(operator.itemgetter(1), out[:10]))

  print(f"Ridge, alpha={i}")
  print("\tOutput indexes: ", res)
  print("\tOutput indexes (1-indexed): ", [v + 1 for v in res])

  print("\tStrongest 10: ", np.round(np.array([v[0] for v in out[:10]]), 3))
  print("\tNext Strongest 5: ", np.round(np.array([v[0] for v in out[10:15]]), 3))
  print(f"\tGap: [{abs(out[9][0]) - abs(out[10][0])}]")

for i in np.linspace(0.01, 1, 10):
  reg = linear_model.Lasso(alpha=i).fit(X, y)

  w, b = reg.coef_, reg.intercept_

  out = sorted(zip(list(w.flatten()), range(len(w.flatten()))), key = lambda x : -abs(x
  res = sorted(map(operator.itemgetter(1), out[:10]))

  print(f"Lasso, alpha={i}")
  print("\tOutput indexes: ", res)
  print("\tOutput indexes (1-indexed): ", [v + 1 for v in res])
```

```
    print("\tStrongest 10: ", np.round(np.array([v[0] for v in out[:10]]), 3))
    print("\tNext Strongest 5: ", np.round(np.array([v[0] for v in out[10:15]]), 3))
    print(f"\tGap: [{abs(out[9][0]) - abs(out[10][0])}]")
    # for i, v in enumerate(out[:20]):
    #   if i == 10:
    #     print("VV== Should be sharp drop below ==VV")
    #   print(v)
```

```
Ridge, alpha=0
        Output indexes:  [1, 10, 20, 23, 26, 33, 43, 57, 67, 70]
        Output indexes (1-indexed):  [2, 11, 21, 24, 27, 34, 44, 58, 68, 71]
        Strongest 10:  [ 3.347  3.204  2.866 -2.501  2.248  2.22  -2.145  2.087 -2.073 -
2.024]
        Next Strongest 5:  [ 1.922  1.889  1.859 -1.83  -1.8  ]
        Gap: [0.10115185856595121]
Ridge, alpha=5
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [1.094 1.073 0.936 0.928 0.839 0.824 0.815 0.777 0.737 0.644]
        Next Strongest 5:  [0.374 0.308 0.304 0.277 0.271]
        Gap: [0.270319002169907]
Ridge, alpha=10
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.992 0.961 0.865 0.791 0.759 0.753 0.722 0.708 0.705 0.566]
        Next Strongest 5:  [ 0.355  0.264  0.241  0.237 -0.221]
        Gap: [0.211190224943121]
Ridge, alpha=15
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.928 0.879 0.807 0.736 0.712 0.697 0.678 0.655 0.642 0.518]
        Next Strongest 5:  [ 0.342  0.247  0.211 -0.206  0.204]
        Gap: [0.17677514297840408]
Ridge, alpha=20
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.877 0.82  0.76  0.713 0.656 0.656 0.65  0.606 0.598 0.483]
        Next Strongest 5:  [ 0.33   0.23   0.213 -0.199 -0.198]
        Gap: [0.1523785553397372]
Ridge, alpha=25
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.834 0.774 0.721 0.692 0.625 0.624 0.614 0.567 0.564 0.454]
        Next Strongest 5:  [ 0.32   0.216 -0.215  0.213 -0.192]
        Gap: [0.13364157993843162]
Ridge, alpha=30
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.797 0.735 0.687 0.671 0.602 0.596 0.579 0.537 0.536 0.429]
        Next Strongest 5:  [ 0.311 -0.226  0.213  0.204 -0.191]
        Gap: [0.11859582413173275]
Ridge, alpha=35
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.765 0.703 0.658 0.651 0.581 0.572 0.551 0.513 0.51  0.408]
        Next Strongest 5:  [ 0.302 -0.234  0.212  0.193 -0.19 ]
        Gap: [0.10615841736401493]
Lasso, alpha=0.01
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [1.183 1.178 1.177 1.084 0.992 0.988 0.985 0.905 0.882 0.765]
        Next Strongest 5:  [ 0.325 -0.307 -0.302 -0.287  0.277]
        Gap: [0.44000164237035727]
```

```
Lasso, alpha=0.12
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [1.025 1.021 0.936 0.931 0.87  0.847 0.81  0.806 0.734 0.715]
        Next Strongest 5:  [-0.084 -0.066 -0.061 -0.034  0.026]
        Gap: [0.6303411282607317]
Lasso, alpha=0.23
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.973 0.906 0.868 0.844 0.768 0.761 0.721 0.666 0.63  0.574]
        Next Strongest 5:  [-0.034 -0.027 -0.     0.     0.   ]
        Gap: [0.5401516469191754]
Lasso, alpha=0.34
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.92  0.802 0.785 0.743 0.666 0.655 0.631 0.53  0.526 0.419]
        Next Strongest 5:  [-0.001 -0.     0.     0.     0.   ]
        Gap: [0.41718710149151095]
Lasso, alpha=0.45
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.863 0.738 0.671 0.643 0.566 0.535 0.533 0.423 0.389 0.25 ]
        Next Strongest 5:  [-0.  0.  0.  0. -0.]
        Gap: [0.2497038902728777]
Lasso, alpha=0.56
        Output indexes:  [1, 2, 4, 6, 10, 12, 16, 18, 22, 26]
        Output indexes (1-indexed):  [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
        Strongest 10:  [0.807 0.674 0.556 0.544 0.465 0.438 0.411 0.317 0.253 0.08 ]
        Next Strongest 5:  [-0.  0.  0.  0. -0.]
        Gap: [0.0803492240049078]
Lasso, alpha=0.67
        Output indexes:  [0, 1, 2, 4, 6, 10, 12, 18, 22, 26]
        Output indexes (1-indexed):  [1, 2, 3, 5, 7, 11, 13, 19, 23, 27]
        Strongest 10:  [ 0.751  0.6    0.445  0.429  0.357  0.352  0.298  0.215  0.134 -
0.   ]
        Next Strongest 5:  [-0.  0.  0. -0.  0.]
        Gap: [0.0]
Lasso, alpha=0.78
        Output indexes:  [0, 1, 2, 4, 6, 10, 12, 18, 22, 26]
        Output indexes (1-indexed):  [1, 2, 3, 5, 7, 11, 13, 19, 23, 27]
        Strongest 10:  [ 0.696  0.517  0.349  0.291  0.276  0.243  0.193  0.117  0.031 -
0.   ]
        Next Strongest 5:  [-0.  0.  0. -0.  0.]
        Gap: [0.0]
Lasso, alpha=0.89
        Output indexes:  [0, 1, 2, 3, 4, 6, 10, 18, 22, 26]
        Output indexes (1-indexed):  [1, 2, 3, 4, 5, 7, 11, 19, 23, 27]
        Strongest 10:  [ 0.625  0.421  0.264  0.204  0.163  0.127  0.079  0.035 -0.    -
0.   ]
        Next Strongest 5:  [ 0.  0. -0.  0.  0.]
        Gap: [0.0]
Lasso, alpha=1.0
        Output indexes:  [0, 1, 2, 3, 4, 5, 6, 7, 22, 26]
        Output indexes (1-indexed):  [1, 2, 3, 4, 5, 6, 7, 8, 23, 27]
        Strongest 10:  [ 0.533  0.318  0.179  0.13   0.04   0.014 -0.    -0.     0.
0.   ]
        Next Strongest 5:  [-0.  0.  0.  0.  0.]
        Gap: [0.0]
```

Q8

In [65]:

```
import pandas as pd
```

```
df = pd.read_csv("data/HW4/heart.csv")
df.head()
X = np.array(df.iloc[:,:-1])
y = np.array(df.iloc[:,[-1]])


np.random.seed(123)

idx = np.random.choice(303, 303, replace=False)
tr_idx, te_idx = idx[:200], idx[200:]

X_tr, y_tr = X[tr_idx], y[tr_idx]
X_te, y_te = X[te_idx], y[te_idx]

def normalize(mode, X_tr, X_te):
  if mode == "z":
    # z-score normalize
    X_tr_mu = np.mean(X_tr, axis=0, keepdims=True)
    X_tr_sig = np.std(X_tr, axis=0, keepdims=True)

    X_tr_ = (X_tr - X_tr_mu) / X_tr_sig
    X_te_ = (X_te - X_tr_mu) / X_tr_sig
    return X_tr_, X_te_
  elif mode == "min-max":
    # min-max 0-1 normalization
    X_tr_mi = np.min(X_tr, axis=0, keepdims=True)
    X_tr_ma = np.max(X_tr, axis=0, keepdims=True)
    X_tr_ = (X_tr - X_tr_mi) / (X_tr_ma - X_tr_mi)
    X_te_ = (X_te - X_tr_mi) / (X_tr_ma - X_tr_mi)
    return X_tr_, X_te_
  else:
    return X_tr, X_te

print(X_tr.shape, y_tr.shape, X_te.shape, y_te.shape)
```

```
(200, 13) (200, 1) (103, 13) (103, 1)
```

Q8a

The three most most significant features (based on magnitude of weight, after normalizing the features) are 'sex', 'ca', and 'cp'

In [66]:

```
np.set_printoptions(linewidth=200)
col_names = list(df.columns)[:-1]

def most_influential(w):
  feats = sorted(list(zip(w.flatten(), col_names)), key = lambda x : -np.abs(x[0]))
  print(f"\t{[(v[1], np.round(v[0], 4)) for v in feats[:5]]}")

def test_err(reg: linear_model.LogisticRegression, X, y):
  y_hat = reg.predict(X)
  err = np.not_equal(y_hat, y.flatten())
  return np.sum(err)

def experiment(mode):
  print(f"*** Using {mode} Normalization")
  X_tr_, X_te_ = normalize(mode, X_tr, X_te)
```

```
  for C in [0]:
    print(f"L2 Regularization = {C}")
    if C == 0:
      reg = linear_model.LogisticRegression().fit(X_tr_, y_tr.flatten())
    else:
      reg = linear_model.LogisticRegression(C=C).fit(X_tr_, y_tr.flatten())
    w, b = reg.coef_, reg.intercept_
    print("\tw: ", np.round(w, 4).flatten())
    print("\tb: ", np.round(b, 4))
    most_influential(w)

    print(f"=== Testing === (Q8b)")
    err = test_err(reg, X_te_, y_te)
    print(f"\tError Rate: {np.sum(err)}/{len(y_te)}={np.sum(err)/len(y_te):0.4f}")
  print()
  print("-"*50)
  print()

experiment("z")
experiment("min-max")
experiment(None)
```

```
*** Using z Normalization
L2 Regularization = 0
        w:  [-0.0366 -0.9783  0.8047 -0.4959 -0.3468  0.1503  0.1725  0.691  -0.4286 -0.
7565  0.5111 -0.8314 -0.4907]
        b:  [-0.0234]
        [('sex', -0.9783), ('ca', -0.8314), ('cp', 0.8047), ('oldpeak', -0.7565), ('thal
ach', 0.691)]
=== Testing === (Q8b)
        Error Rate: 13/103=0.1262

--------------------------------------------------

*** Using min-max Normalization
L2 Regularization = 0
        w:  [-0.6315 -1.3791  1.6426 -1.1085 -0.6228  0.1735  0.5641  1.4912 -0.867  -1.
7894  1.3386 -1.8582 -1.3364]
        b:  [1.4089]
        [('ca', -1.8582), ('oldpeak', -1.7894), ('cp', 1.6426), ('thalach', 1.4912), ('s
ex', -1.3791)]
=== Testing === (Q8b)
        Error Rate: 13/103=0.1262

--------------------------------------------------

*** Using None Normalization
L2 Regularization = 0
        w:  [ 0.0187 -1.5203  0.7477 -0.0241 -0.0053  0.1293  0.4061  0.0371 -0.6407 -0.
5994  0.7219 -0.7591 -0.7095]
        b:  [0.0472]
        [('sex', -1.5203), ('ca', -0.7591), ('cp', 0.7477), ('slope', 0.7219), ('thal',
-0.7095)]
=== Testing === (Q8b)
        Error Rate: 14/103=0.1359

--------------------------------------------------

c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\sklearn\linear_model\_logistic.py:
458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Q8c

In [62]:

```
n = len(X_tr)
folds = 5
stats_rate = []
stats_wrong = []

def k_exp(mode):
  print(f"=== Mode: {mode} ===")
  for fold in range(folds):
    b1 = np.round(n / folds * fold).astype(int)
    b2 = np.round(n / folds * (fold + 1)).astype(int)
    X_trr = np.vstack((X_tr[:b1,:], X_tr[b2:,:]))
    y_trr = np.vstack((y_tr[:b1,:], y_tr[b2:,:]))
    X_va = X_tr[b1:b2]
    y_va = y_tr[b1:b2]

    # X_tr_, X_va_ = X_trr, X_va
    X_tr_, X_va_ = normalize(mode, X_trr, X_va)

    print(f"Fit: {X_tr_.shape}, {y_trr.shape}")
    reg = linear_model.LogisticRegression().fit(X_tr_, y_trr.flatten())
    err = test_err(reg, X_va_, y_va)

    print(f"\t[Fold {fold}] Error Rate: {np.sum(err)}/{len(y_va)}={np.sum(err)/len(y_va
    stats_rate.append(err / len(y_va))
    stats_wrong.append(err)
  print(f"sum(rate)/5 = {np.mean(stats_rate).item():0.4f}")
  print(f"sum(wrong)/5 = {np.mean(stats_wrong).item():0.4f}")
k_exp("z")
k_exp("min-max")
k_exp(None)
```

```
=== Mode: z ===
Fit: (160, 13), (160, 1)
        [Fold 0] Error Rate: 6/40=0.1500
Fit: (160, 13), (160, 1)
        [Fold 1] Error Rate: 6/40=0.1500
Fit: (160, 13), (160, 1)
        [Fold 2] Error Rate: 7/40=0.1750
Fit: (160, 13), (160, 1)
        [Fold 3] Error Rate: 7/40=0.1750
Fit: (160, 13), (160, 1)
        [Fold 4] Error Rate: 4/40=0.1000
sum(rate)/5 = 0.1500
sum(wrong)/5 = 6.0000
=== Mode: min-max ===
Fit: (160, 13), (160, 1)
        [Fold 0] Error Rate: 7/40=0.1750
Fit: (160, 13), (160, 1)
        [Fold 1] Error Rate: 5/40=0.1250
Fit: (160, 13), (160, 1)
        [Fold 2] Error Rate: 8/40=0.2000
```

```
Fit: (160, 13), (160, 1)
        [Fold 3] Error Rate: 6/40=0.1500
Fit: (160, 13), (160, 1)
        [Fold 4] Error Rate: 4/40=0.1000
sum(rate)/5 = 0.1500
sum(wrong)/5 = 6.0000
=== Mode: None ===
Fit: (160, 13), (160, 1)
        [Fold 0] Error Rate: 6/40=0.1500
Fit: (160, 13), (160, 1)
        [Fold 1] Error Rate: 8/40=0.2000
Fit: (160, 13), (160, 1)
        [Fold 2] Error Rate: 7/40=0.1750
Fit: (160, 13), (160, 1)
        [Fold 3] Error Rate: 6/40=0.1500
Fit: (160, 13), (160, 1)
        [Fold 4] Error Rate: 4/40=0.1000
sum(rate)/5 = 0.1517
sum(wrong)/5 = 6.0667
c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\sklearn\linear_model\_logistic.py:
458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\sklearn\linear_model\_logistic.py:
458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\sklearn\linear_model\_logistic.py:
458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\sklearn\linear_model\_logistic.py:
458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\sklearn\linear_model\_logistic.py:
458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```