



## Contents

<b>1</b>	<b>Linear Regression</b>	<b>2</b>
<b>2</b>	<b>Logistic regression</b>	<b>5</b>
<b>3</b>	<b>Logistic regression First/Second-Order Methods</b>	<b>12</b>

The Jupyter Notebook is available on GitHub

<https://github.com/HimethW/Learning-from-data-and-related-challenges.git>

# 1 Linear Regression

1) A set of data points  $(x_i, y_i)$  are known to form a line. The ordinary least squares(OLS) is performed on this dataset. The OLS minimizes a loss function which is defined as  $\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$  with  $y_i$  and  $\hat{y}_i$  are true and OLS outputs, respectively. The fitted OLS line and data points are shown in Figure 1. It is observed that the OLS fitted line is not aligned to majority of data points. What is the reason behind this?

The poor fitting of the OLS line with the data points is due to the presence of several outliers.

**Reason:**

- The loss function that is used in the OLS algorithm which is  $\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$ , is really sensitive towards outliers. This is because it takes the residual error between the predicted and true values(error) and squares it. as we can see, in the dataset there are several outliers and the error introduced from them is large. by squaring it, the impact of those outliers becomes even larger.
- So in order to minimize the loss, the OLS line is pulled towards the outliers as shown in the figure. This results in the OLS line not fitting with the majority of data points.

2) To reduce the impact of outliers, a modified loss function is introduced. It is given as  $L = \frac{1}{N} \sum_{i=1}^N a_i (y_i - \hat{y}_i)^2$  There are two schemes proposed for setting  $a_i$ :

- **Scheme 1:** For outliers  $a_i = 0.01$  and for inliers  $a_i = 1$ ,
- **Scheme 2:** For outliers  $a_i = 5$  and for inliers  $a_i = 1$ .

Under which scheme do you expect a better fitted line for inliers than the OLS fitted line in Figure 1. Justify your answer.

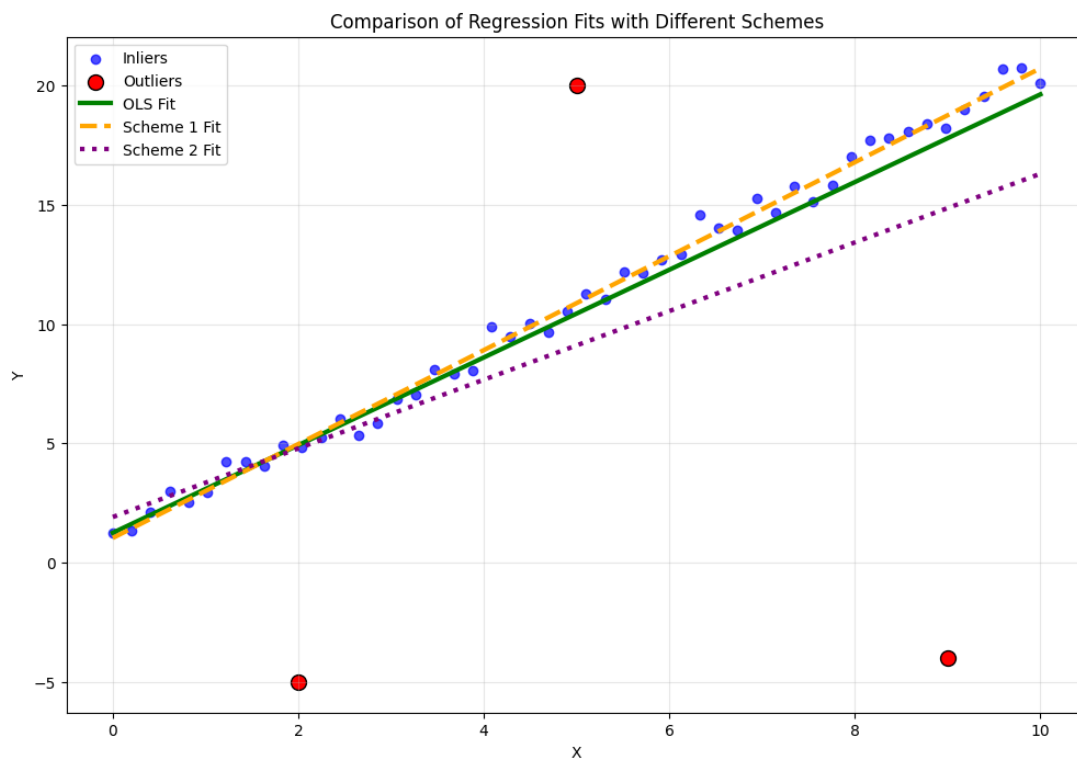
Let's consider a random dataset with several outliers and plot the OLS line using both schemes as well as with the original OLS algorithm

```
1 np.random.seed(42)
2 n_inliers = 50
3 x_inliers = np.linspace(0, 10, n_inliers)
4
5 # y = 2*x + 1 + noise
6 y_inliers = 2 * x_inliers + 1 + np.random.normal(0, 0.5, n_inliers)
7
8 #Generate a few outliers
9 n_outliers = 3
10 x_outliers = np.array([2, 5, 9])
11 y_outliers = np.array([-5, 20, -4])
12
```

```

13 #Combine into the full dataset. Last 3 elements are the outliers
14 X = np.concatenate([x_inliers, x_outliers]).reshape(-1, 1)
15 y = np.concatenate([y_inliers, y_outliers])
16
17 #Create the sample weights array for each scheme
18 # First 50 points are inliers (weight=1), last 3 are outliers
19 weights_scheme1 = np.ones(len(X))
20 weights_scheme1[-n_outliers:] = 0.01 # Downweight outliers
21
22 weights_scheme2 = np.ones(len(X))
23 weights_scheme2[-n_outliers:] = 5 # Overweight outliers
24
25 #Fit the models
26 #Standard OLS
27 model_ols = LinearRegression()
28 model_ols.fit(X, y)
29
30 #Scheme 1
31 model_scheme1 = LinearRegression()
32 model_scheme1.fit(X, y, sample_weight=weights_scheme1)
33
34 #Scheme 2
35 model_scheme2 = LinearRegression()
36 model_scheme2.fit(X, y, sample_weight=weights_scheme2)
37
38 #create x axis for plotting the models
39 x_plot = np.linspace(0, 10, 100).reshape(-1, 1)
40 y_plot_ols = model_ols.predict(x_plot)
41 y_plot_scheme1 = model_scheme1.predict(x_plot)
42 y_plot_scheme2 = model_scheme2.predict(x_plot)

```



A better fitted line for in-liners can be expected by using **Scheme 1**:

(For outliers  $a_i = 0.01$  and for inliers  $a_i = 1$ )

**Reason:**

- The  $a_i$  value is used to determine how much weight we give to the error of the  $i^{th}$  data point.
- In scheme 1, by giving the outliers a weight of 0.01, we are down-weighting their contribution by a factor of 100. This way we are able to neglect the large penalty they introduce. As for the inliers we are keeping the weight at 1, which is the normal value used in the original OLS algorithm so the inlier error contribution is not affected. This way we can get a better fit for the inliers. This can also be seen by the yellow dashed line in the above figure.
- In scheme 2, we overweight the outliers by giving them a larger weight than the inliers ( $a_i = 5$ ). The outliers already give a large error term and by over-weighting them, the outlier contribution increases even further. The model will now try more to fit the outliers than before, leading to the OLS line being more skewed towards the outliers. This can also be seen in the above figure using the purple dotted line.
- We can conclude that a better fit for the inliers can be expected by using **scheme 1**.

**3) In brain image analysis (eg: fMRI), the brain is divided into multiple regions as shown in Figure 2, each consisting of many voxels (pixels). A researcher wants to identify which brain regions are most predictive of a specific cognitive task. Why linear regression is not suitable algorithm for the above task?**

- In this dataset, the number of features, which is the number of voxels, can be very large (several hundreds of thousands). However, the number of observations that we can get will be very much lower than the number of features. So there can be many weight vectors that can be used to fit the regression line to our dataset, making it less generalized and unstable. This will lead to the model not working properly with new data.
- If linear regression is used, it will give a weight for each individual voxel. The resulting weight vector will be at the voxel level, making it hard to identify which region is most predictive (For example, some voxels in region 'A' will get a higher weight while some from the same region will get a lower weight). The goal is not to identify each voxel but to find out the region. Linear regression does not give a clear answer at the region level.

**5) Which method (LASSO or group LASSO) is more appropriate in this setting, and why?**

**Method A: Standard LASSO**

$$\min_w \left( \frac{1}{N} \sum_{i=1}^N (y_i - w^\top x_i)^2 + \lambda \|w\|_1 \right)$$

## Method B: Group LASSO

$$\min_w \left( \frac{1}{N} \sum_{i=1}^N (y_i - w^\top x_i)^2 + \lambda \sum_{g=1}^G \|w_g\|_2 \right)$$

The best method is **Group Lasso**

### Reason:

- The Standard Lasso method adds a penalty term that is used to reduce large weights making the model more simple. This is done by the Lasso method taking the less important features and forcing their weight to be zero. The result is a sparse weight vector. However, this method removes features independently without caring about the regions. For example, some voxels from region 'A' might be kept as important while some voxels from the same region might be deemed unimportant.
- Group Lasso also has a penalty term but it is applied to entire groups rather than individual voxels. The  $L_2$  norm  $\|w_g\|_2$  measures the weight of the  $\mathbf{w}$  vector for that entire group. So, as a result, it either keeps all the features in a particular group or deletes all the features in a group, instead of treating the voxels individually. If a region is predictive, all the voxels in that region are kept, providing a clear answer on which brain regions are most predictive.
- In conclusion we can say that Group Lasso is most suitable for the task.

## 2 Logistic regression

2) Now, use the code given in listing 2 to train a logistic regression model. Here, did you encounter any errors? If yes, what were they, and how would you go about resolving them?

During training, the following error was encountered.

ValueError: could not convert string to float: 'Adelie'

```
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_55188\484805931.py in ?()
      2 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
      3
      4 # Train the logistic regression model. Here we are using saga solver to learn weights.
      5 logreg = LogisticRegression(solver='saga')
----> 6 logreg.fit(X_train, y_train)
      7
      8 # Predict on the testing data
      9 y_pred = logreg.predict(X_test)
```

Figure 1: Encountered Error

### Reason:

The feature matrix X is taken after dropping the "class encoded" column from df\_filtered. X still contains categorical columns such as Species, Island and Sex. However, the logistic

regression model in scikit-learn can only handle data with a numerical value so these features will result in a `ValueError` as shown above.

### Solution:

```
1 #drop species column
2 X = X.drop(['species'], axis=1)
3
4 #encode the categorical columns using One-Hot encoding
5 X = pd.get_dummies(X, columns=['island', 'sex'])
6
7 # Split the data into training and testing sets
8 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
9             random_state=42)
10
11 # Train the logistic regression model. Here we are using saga solver to
12   learn weights.
13
14 logreg = LogisticRegression(solver='saga')
15 logreg.fit(X_train, y_train)
16
17 # Predict on the testing data
18 y_pred = logreg.predict(X_test)
19
20 # Evaluate the model
21 accuracy = accuracy_score(y_test, y_pred)
22 print("Accuracy:", accuracy)
23 print(logreg.coef_, logreg.intercept_)
```

- To get rid of the error, we need to encode the categorical columns.
- First remove the species columns as that is our target and not a feature.  
`X = X . drop (['species'] , axis =1)`
- Then use One-Hot encoding to convert 'island' and 'sex' to numbers.  
`X = pd.get_dummies(X, columns=['island', 'sex'])`

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	\
0	Adelie	Torgersen	39.1	18.7	181.0	
1	Adelie	Torgersen	39.5	17.4	186.0	
2	Adelie	Torgersen	40.3	18.0	195.0	
4	Adelie	Torgersen	36.7	19.3	193.0	
5	Adelie	Torgersen	39.3	20.6	190.0	
..	...	...	...	...	...	
215	Chinstrap	Dream	55.8	19.8	207.0	
216	Chinstrap	Dream	43.5	18.1	202.0	
217	Chinstrap	Dream	49.6	18.2	193.0	
218	Chinstrap	Dream	50.8	19.0	210.0	
219	Chinstrap	Dream	50.2	18.7	198.0	
	body_mass_g	sex				
0	3750.0	Male				
1	3800.0	Female				
2	3250.0	Female				
4	3450.0	Female				
5	3650.0	Male				
..	...	...				
215	4000.0	Male				
216	3400.0	Female				
217	3775.0	Male				
218	4100.0	Male				
219	3775.0	Female				

Figure 2: Before encoding

	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	\
0	39.1	18.7	181.0	3750.0	
1	39.5	17.4	186.0	3800.0	
2	40.3	18.0	195.0	3250.0	
4	36.7	19.3	193.0	3450.0	
5	39.3	20.6	190.0	3650.0	
..	...	...	...	...	
215	55.8	19.8	207.0	4000.0	
216	43.5	18.1	202.0	3400.0	
217	49.6	18.2	193.0	3775.0	
218	50.8	19.0	210.0	4100.0	
219	50.2	18.7	198.0	3775.0	
	island_Biscoe	island_Dream	island_Torgersen	sex_Female	sex_Male
0	False	False	True	False	True
1	False	False	True	True	False
2	False	False	True	True	False
4	False	False	True	True	False
5	False	False	True	False	True
..	...	...	...	...	...
215	False	True	False	False	True
216	False	True	False	True	False
217	False	True	False	False	True
218	False	True	False	False	True
219	False	True	False	True	False

Figure 3: After encoding

**Result**  $\Rightarrow$  Accuracy: 0.5813953488372093

### 3) Why does the saga solver perform poorly?

- As mentioned above, the accuracy is about 0.581, which indicates poor performance. The main reason for this is that the "saga" solver is sensitive towards feature scaling.
- Our dataset has features that are on different scales. For example the mass is in thousands of grams while the bill length is around 40mm.
- The "saga" solver is based on stochastic gradient descent, so larger scales try to dominate the updating and learning process of the model. This will lead to poor performance and slow convergence.

### 4) Now change the solver to "liblinear" by using `logreg = LogisticRegression(solver='liblinear')`. What is the classification accuracy with this configuration?

Updated code:

```
1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
2 # Train the logistic regression model.
3 logreg = LogisticRegression(solver='liblinear')
4 logreg.fit(X_train, y_train)
5
6 # Predict on the testing data
7 y_pred = logreg.predict(X_test)
8
9 # Evaluate the model
10 accuracy = accuracy_score(y_test, y_pred)
11 print("Accuracy:", accuracy)
12 print(logreg.coef_, logreg.intercept_)
```

```
Accuracy: 1.0
[[ 1.49676343 -1.38121095 -0.1435575  -0.00353557 -0.23127486  0.72387861
 -0.57084769  0.1303921  -0.20863605]] [-0.07824394]
```

Figure 4: Accuracy when using liblinear solver

Result  $\Rightarrow$  Accuracy: 1.0

Solver	Accuracy
saga solver	0.5813953488372093
liblinear solver	1.0

### 5) Why does the "liblinear" solver perform better than "saga" solver?

- Our data is unscaled and the liblinear solver is better at handling unscaled data. This is because it internally used different optimization algorithms, making it more robust.
- Moreover, liblinear solver is optimized to handle small datasets (our dataset has only 9 features and 214 data items) making it more stable.



6) Explain why the model's accuracy (with saga solver) varies with different random state values ?

Random state value	Accuracy
10	0.6976744186046512
20	0.627906976744186
30	0.6744186046511628
42	0.5813953488372093
50	0.7674418604651163
60	0.6976744186046512
70	0.6744186046511628

Table 1: Accuracy comparison with different random state values

**Reason:**

- The saga solver uses the Stochastic gradient descent algorithm as mentioned above.
- So when the dataset is unscaled, the landscape where the gradient descent takes place becomes poorly shaped (very steep or narrow) and the starting point of the algorithm greatly affects the outcome. The random\_state determines this starting point.
- Different random\_states can cause the algorithm to converge to different local minima or even get stuck at some point, leading to different accuracy levels as shown in the above table

7) Compare the performance of the "liblinear" and "saga" solvers with feature scaling. If there is a significant difference in the accuracy with and without feature scaling, what is the reason for that. You may use Standard Scaler available in sklearn library.

• saga solver with feature scaling

```
1 from sklearn.preprocessing import StandardScaler
2 scaler = StandardScaler()
3
4 random_state_vals = [10,20,30,42,50,60,70]
5 for rs in random_state_vals:
6     # Split the data into training and testing sets
7     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=rs)
8
9     #scale the data
10    X_train_scaled = scaler.fit_transform(X_train)
11    X_test_scaled = scaler.transform(X_test)
12
```

```

13 # Train the logistic regression model. Here we are using saga
    solver to learn weights.
14 logreg = LogisticRegression(solver='saga')
15 logreg.fit(X_train_scaled, y_train)
16
17 # Predict on the testing data
18 y_pred = logreg.predict(X_test_scaled)
19
20 # Evaluate the model
21 accuracy = accuracy_score(y_test, y_pred)
22 print(f"Accuracy: {accuracy}          random state value: {rs}")
23 print(logreg.coef_, logreg.intercept_)
24 print()
25

```

```

Accuracy: 1.0          random state value: 10
[[ 3.43987734 -0.57144326  0.43043397 -0.40348266 -0.45489502  0.86741243
 -0.59984691  0.46510701 -0.46510701]] [-2.29978478]

Accuracy: 1.0          random state value: 20
[[ 3.36597164 -0.41429405  0.44539521 -0.24671521 -0.36848432  0.85458601
 -0.65501971  0.49738649 -0.49738649]] [-2.50893234]

Accuracy: 0.9767441860465116          random state value: 30
[[ 3.41950719 -0.47226229  0.57662858 -0.50476074 -0.40534909  0.8524272
 -0.62943684  0.48143853 -0.48143853]] [-2.34483569]

Accuracy: 1.0          random state value: 42
[[ 3.3394185  -0.42444243  0.45868914 -0.29610856 -0.44263423  0.89167189
 -0.62628753  0.5352215  -0.5352215  ]] [-2.57333491]

Accuracy: 1.0          random state value: 50
[[ 3.46694884 -0.47930744  0.4196324  -0.33168718 -0.48386059  0.88190859
 -0.60326942  0.52573396 -0.52573396]] [-2.05178922]

Accuracy: 1.0          random state value: 60
[[ 3.31714155 -0.3616647  0.46972869 -0.33946006 -0.47073203  0.95673766
 -0.67191829  0.53939887 -0.53939887]] [-2.32229919]

Accuracy: 1.0          random state value: 70
[[ 3.48202814 -0.38286735  0.43670146 -0.27893331 -0.42037874  0.82776828
 -0.59249266  0.50888196 -0.50888196]] [-2.23824651]

```

Figure 5: Accuracy of the saga solver after feature scaling

- As we can see the accuracy of the saga solver has improved significantly upto 1. Even after changing the random state, the accuracy did not decrease much (even when `random_state = 30`, the reduction in accuracy is insignificant) indicating that the model is stable. It has matched the accuracy of the liblinear solver.
- The reason for this is that after scaling, the dataset has a mean of 0 and a

standard deviation of 1. This gives a smooth landscape that is easier for the saga solver to perform stochastic gradient descent. It is no longer dominated by large scale features such as `body_mass` resulting in a optimal solution as shown above

## • liblinear solver with feature scaling

```

1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)
2
3 #scale the data
4 X_train_scaled = scaler.fit_transform(X_train)
5 X_test_scaled = scaler.transform(X_test)
6
7 # Train the logistic regression model.
8 logreg = LogisticRegression(solver='liblinear')
9 logreg.fit(X_train_scaled, y_train)
10
11 # Predict on the testing data
12 y_pred = logreg.predict(X_test_scaled)
13
14 # Evaluate the model
15 accuracy = accuracy_score(y_test, y_pred)
16 print("Accuracy:", accuracy)
17 print(logreg.coef_, logreg.intercept_)
18

```

```

Accuracy: 1.0
[[ 3.26974452 -0.3637722  0.40189087 -0.29907748 -0.3029009  0.67927884
 -0.50944667  0.48969306 -0.48969306]] [-1.96200479]

```

Figure 6: Accuracy of the liblinear solver after feature scaling

- As we can see the accuracy of the liblinear solver hasn't changed.
- This is due to the liblinear solver being more robust towards unscaled small datasets. Even before scaling, it had found a optimal solution giving 100% accuracy.

8) Suppose you have a categorical feature with the categories 'red', 'blue', 'green', 'blue', 'green'. After encoding this feature using label encoding, you then apply a feature scaling method such as Standard Scaling or Min-Max Scaling. Is this approach correct? or not?. What do you propose

**This approach is incorrect Reason:**

- Label encoding assigns an integer value for the colours. For example 'red' = 0, 'blue' = 1, 'green' = 2
- when we apply standard scaling or min-max scaling to this we are implying a mathematical order between colours ( green is the highest and red is the lowest) even though

there is no such relation. This will lead to the model learning patterns that are not present.

- The correct method is to use One-Hot encoding. This represents the columns without introducing any order between categories. After encoding, scaling is not needed as the data is already on the binary scale (0 or 1).

Class label	One-Hot encoding
red	100
blue	010
green	001

Table 2: One-Hot encoding

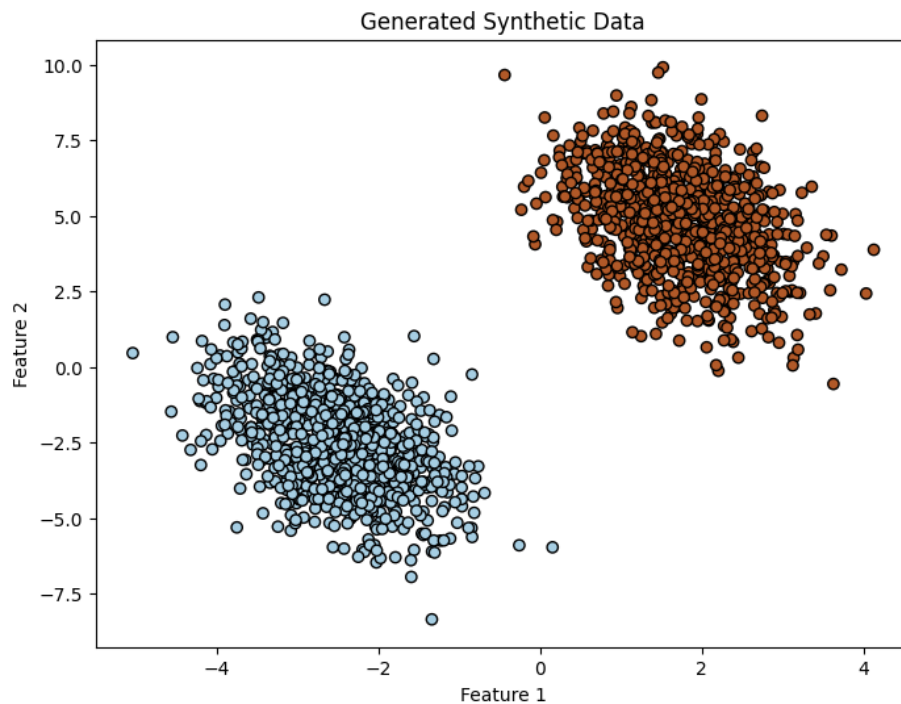
Data Sample	Colour
1	red
2	blue
3	green
4	blue
5	green

Table 3: Before encoding

Data Sample	Is_red	Is_blue	Is_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0
5	0	0	1

Table 4: After One-Hot encoding

### 3 Logistic regression First/Second-Order Methods



2) Implement batch Gradient descent to update the weights for the given dataset over 20 iterations. State the method used to initialize the weights and reason for your selection

```
1 def sigmoid(z):
2     #define the sigmoid function
3     return 1 / (1 + np.exp(-z))
4
5 def loss_calc(y, y_hat):
6     # Binary Cross-Entropy Loss
7     return -np.mean(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat))
8
9 # Add a column of ones to X (bias term)
10 X_b = np.c_[np.ones((X.shape[0], 1)), X] # X_b is now [1, feature1,
11     feature2] with 2000 rows (2000x3 matrix)
12
13 n_in = X_b.shape[1] # Number of input features including the bias term
14 limit = np.sqrt(6 / (n_in + 1)) #n_out is 1 for logistic regression(output
15     features)
16
17 # Initialize weights from the Xavier Uniform distribution
18 w_ini = np.random.uniform(-limit, limit, size=(n_in, 1))
19 w_grad = w_ini #this is a 3x1 matrix
20
21 #make y a column vector
22 y = y.reshape(-1, 1)
```

```

22 learning_rate = 0.1
23 iterations = 20
24
25 # store the loss for plotting
26 loss_history_gradient = []
27
28 # Implement Batch Gradient Descent
29 for i in range(iterations):
30     z = np.dot(X_b, w_grad)
31     y_hat = sigmoid(z)      #this is a 2000x1 matrix
32
33     # store the loss
34     loss = loss_calc(y, y_hat)
35     loss_history_gradient.append(loss)
36
37     # Compute the gradients and update w
38     gradient = np.dot(X_b.T, (y_hat - y)) / len(y)
39     w_grad = w_grad - learning_rate * gradient
40
41 print("Final GD weights:\n", w_grad.ravel())

```

**Result:** Final GD weights: [0.08534997 0.78405749 0.5748131 ]

### Calculation explanation

- $X_b$  is a new feature matrix with a column of 1 added to the front (bias term).
- The weight matrix  $w_{grad}$  is initialized.
- After that the learning rate and no. of iteration are defined.
- `loss_history_gradient` is an array used to store the loss values for plotting.
- $z$  is the value we get when the weights are used on the features. After that we get the predicted value  $\hat{y}$  by passing  $z$  through a sigmoid function defined by  $\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$
- The loss is calculated using the Binary Cross Entropy Loss
- The the gradient was calculated using

$$\frac{\partial L}{\partial w_j} = \sum_{i=1}^m (\hat{y}_i - y_i) x_{j,i} \quad (1)$$

$$gradient = \frac{1}{m} X_b^\top (\hat{y} - y) \quad (2)$$

- The the new weights are calculated using  $w_{grad} = w_{grad} - \alpha \times gradient$

### Method used to initialize the weights:

The weights were initialized using the Xavier Uniform Initialization. It's taken using the following formula

$$w_{ini} \sim U(-limit, limit) \quad (3)$$

where,

$$limit = \sqrt{\frac{6}{n_{in} + n_{out}}} \quad (4)$$

$n_{in}$  = number of input features (3 in this case with bias)

$n_{out}$  = number of output units (1 in this case)

- Using this method we can avoid too small weights(vanishing gradients) and too large weights(exploding gradients).
- This formula ensures that when the number of input features are increased, the uniform distribution limits gets shrunk. So the initial guessing range also changes dynamically based on the inputs.
- An alternate would be zero initialization (simple). However, this is a symmetric starting point but the importance of different features are different. So it is not the best solution.

### 3) Specify the loss function you have used and state reason for your selection.

$$L = -\frac{1}{N} \sum_{i=1}^N (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \quad (5)$$

The loss is calculated using the Binary Cross Entropy Loss. **Reason:**

- Our target variable is a binary value of 0 or 1. BCE is the best loss function for these type of applications as it uses log terms in the loss calculation and gives a heavy penalty if the prediction is wrong. This gives a strong gradient.
- On the other hand, MSE is used when the output is a continuous value. In that case, predicting a 0 as 1 is considered as a small error as they are close. But in classification, this is a major error and MSE does not give a heavy penalty. So MSE is not suitable.
- So we chose the Binary Cross Entropy Loss function.

### 4) Implement Newton's method to update the weights for the given dataset over 20 iterations

```
1 # Initialize weights
2 w_newton = w_ini
3
4 loss_history_newton = []
5
6 # Newton's Method
7 for i in range(iterations):
8
9     z = np.dot(X_b, w_newton)
10    y_hat = sigmoid(z)
11
12
13    loss = loss_calc(y, y_hat)
14    loss_history_newton.append(loss)
15
16 # Compute the gradient
17 gradient = np.dot(X_b.T, (y_hat - y)) / len(y)
18
```

```

19 # Compute the Hessian matrix
20 R = np.diagflat(y_hat * (1 - y_hat)) # R is a diagonal matrix
21 hessian = np.dot(X_b.T, np.dot(R, X_b)) / len(y)
22
23 # Update the weights: w = w - H^{-1} * gradient
24 w_newton = w_newton - np.dot(np.linalg.inv(hessian), gradient)
25
26 print("Final Newton's weights:\n", w_newton.ravel())

```

### Calculation explanation

- Similar to before, first the weights are initialized.
- Then, for each iteration, the prediction  $\hat{y}$  is calculated similar to before and the loss is calculated using the BCE.
- The gradient vector is then calculated using the previous algorithm.
- The weights are updated using the following formula

$$w_{newton} = w_{newton} - H^{-1} \nabla L \quad (6)$$

Where H is the hessian matrix of L

5) Plot the loss with respect to number of iterations for batch Gradient descent and Newton method's in a single plot. Comment on your results.

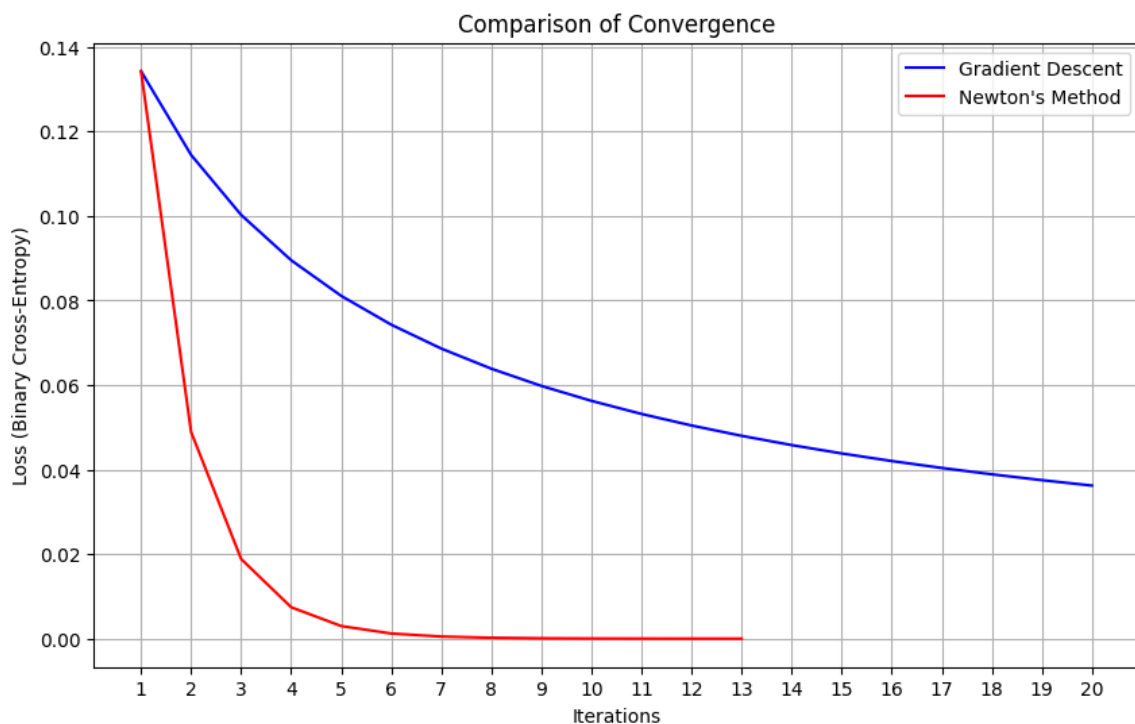


Figure 7: Loss comparison



- It can be seen that the Newton's method converges much faster than the batch gradient descent. The loss of the Newton's method drops quickly while the loss of the gradient descent method decreases slowly.
- The reason for this is that the Newton's method is a second order method. It uses the Hessian matrix to get the curvature of the loss. This way, rather than following the steepest path towards the minima, it can get the exact step needed to get there. This results in quadratic convergence  
] rather than the linear convergence shown in batch gradient descent.

## 6) Propose two approaches to decide number of iterations for Gradient descent and Newton's method.

### 1. Provide a threshold to the gradient magnitude:

Rather than fixing the number of iterations, we can keep on updating the weights until the gradient of the loss has approached a specific threshold. When this happens we can decide that the algorithm has arrived at a local minima point where the slope is almost flat.

```

1 # Add a column of ones to X (bias term)
2 X_b = np.c_[np.ones((X.shape[0], 1)), X] # X_b is now [1, feature1,
      feature2] with 2000 rows (2000x3 matrix)
3
4 # Initialize starting weights to 0.
5 w_grad = w_ini #this is a 3x1 matrix
6
7 #make y a column vector
8 y = y.reshape(-1, 1)
9
10 learning_rate = 0.1
11 max_iterations = 1000
12 threshold = 0.05
13
14 # store the loss for plotting
15 loss_history_gradient = []
16
17 # Implement Batch Gradient Descent
18 for i in range(1,max_iterations+1):
19     z = np.dot(X_b, w_grad)
20     y_hat = sigmoid(z) #this is a 2000x1 matrix
21
22     # store the loss
23     loss = loss_calc(y, y_hat)
24     loss_history_gradient.append(loss)
25
26     # Compute the gradients and update w
27     gradient = np.dot(X_b.T, (y_hat - y)) / len(y)
28     if(np.linalg.norm(gradient) < threshold):
29         break
30     w_grad = w_grad - learning_rate * gradient
31 print("Number of iterations = ",i-1)
32 print("Final GD weights:\n", w_grad.ravel())
33

```

For the Newton's Method also we can calculate the step value. The step is calculated using  $H^{-1}gradient$ . When this step size is very small we can stop the iterations since

```
Number of iterations = 262
Final GD weights:
[-1.3662698  2.06823884 -0.01349061]
```

the algorithm as reached a minimum point.

## 2. Early stopping based on validation:

First, we can split the data into training and validation sets. Then after updating the weights in each iteration, calculate the loss for the validation set. Keep track of the best set of weights that gave the lowest loss for the validation set. At one point the loss for the validation set will start to increase while the training loss keeps on decreasing. This is when the models starts to overfit to the dataset. At this point revert the weights back to the previous weights, which gave the lowest loss. This way we can get the optimal number of iterations needed to train our model for generalized data

```
1
2 #split the data
3 X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
4         random_state=42, stratify=y)
5 X_train_b = np.c_[np.ones((X_train.shape[0], 1)), X_train] # Training
6         data with bias
7 X_val_b = np.c_[np.ones((X_val.shape[0], 1)), X_val] #
8         Validation data with bias
9
10
11 w_grad = w_ini # this is a 3x1 matrix
12
13
14 y_train = y_train.reshape(-1, 1)
15 y_val = y_val.reshape(-1, 1)
16
17 learning_rate = 0.1
18 max_iterations = 1000
19 no_improvement = 10 # How many iterations to wait after no
20         improvement
21 tolerance = 0.001
22
23 best_val_loss = np.inf
24 best_weights = w_grad.copy() # Save a copy of the best weights
25 no_improvement_counter = 0 # Counter for how long we've waited
26         without improvement
27
28 loss_history_gradient_3 = []
29 val_loss_history = [] # New list to track validation loss
30
31 # Batch Gradient Descent with Early Stopping
32 for i in range(max_iterations):
33     # Use the training set to calculate gradient and update weights
34     z_train = np.dot(X_train_b, w_grad)
35     y_hat_train = sigmoid(z_train)
36
37     train_loss = loss_calc(y_train, y_hat_train)
```

```

33     loss_history_gradient_3.append(train_loss)
34
35     gradient = np.dot(X_train_b.T, (y_hat_train - y_train)) / len(
36         y_train)
37     w_grad = w_grad - learning_rate * gradient
38
39     # Use the validation dataset to check the validation loss
40     z_val = np.dot(X_val_b, w_grad)
41     y_hat_val = sigmoid(z_val)
42     current_val_loss = loss_calc(y_val, y_hat_val)
43     val_loss_history.append(current_val_loss)
44
45     if best_val_loss - current_val_loss > tolerance:
46         best_val_loss = current_val_loss
47         best_weights = w_grad.copy() # Save a copy of these best
48         # weights if the loss has reduced
49         no_improvement_counter = 0 # Reset the counter
50     else:
51         # did not improve.
52         no_improvement_counter += 1
53
54     #check if there is no improvement for several iterations
55     if no_improvement_counter >= no_improvement:
56         print(f"\nEarly stopping triggered at iteration {i+1}")
57         print(f"Best validation loss was {best_val_loss:.6f} at
58         iteration {i+1 - no_improvement_counter}")
59         # Revert our weights to the best ones we found
60         w_grad = best_weights
61         break

```

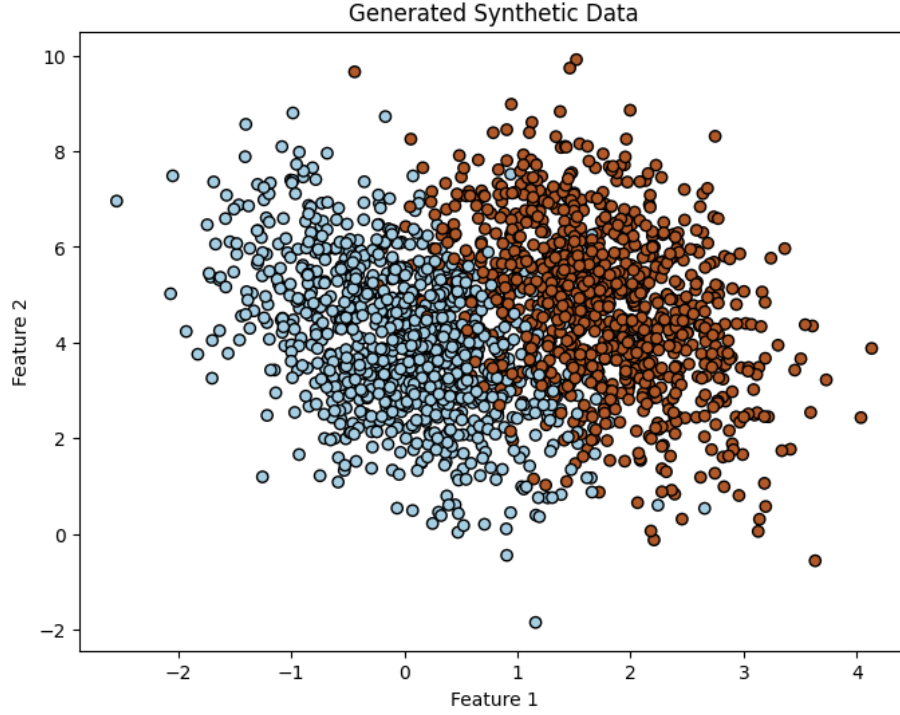
```

Early stopping triggered at iteration 102
Best validation loss was 0.010479 at iteration 92
Final GD weights:
[0.02631071 1.07526056 0.82649732]

```

7) Suppose the centers in in listing 3 are changed to `centers = [[2, 2], [5, 1.5]]`. Use batch Gradient descent to update the weights for this new configuration. Analyze the convergence behavior of the algorithm with this updated data, and provide an explanation for convergence behavior

Now the centers are close together so there will be some overlapping between the classes.



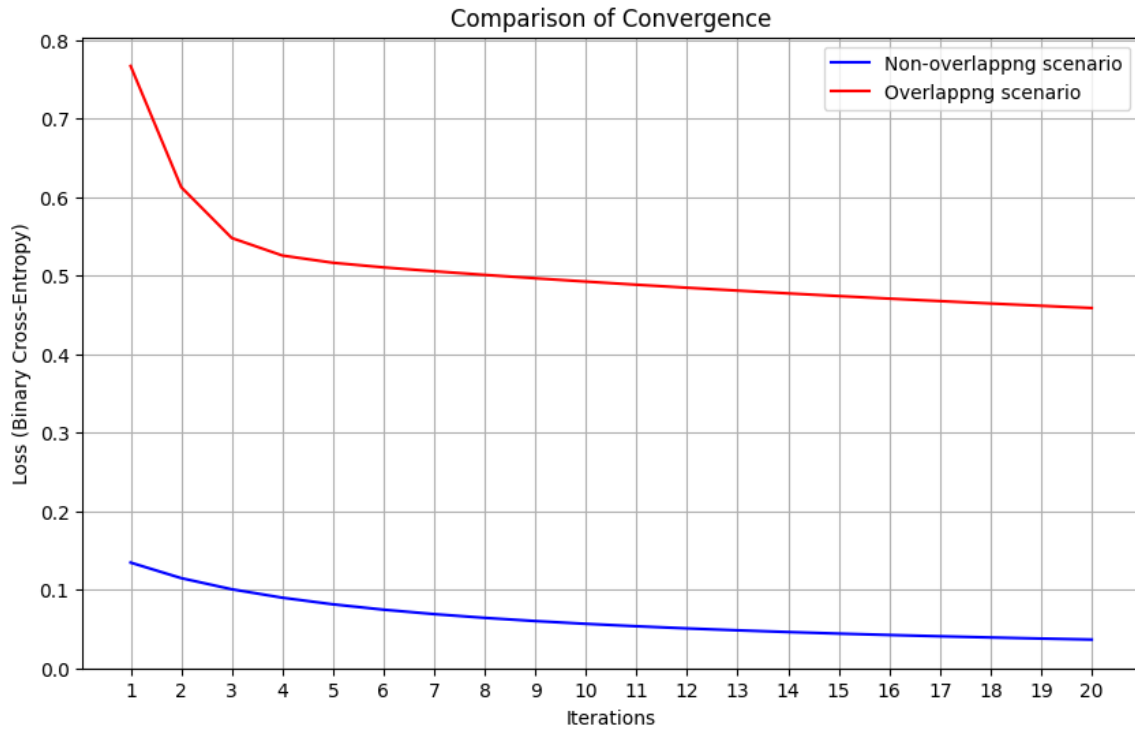
After using the batch gradient descent for 20 iterations, the following results were obtained.

**Result:** Final GD weights:  $[-0.06405436 \ 0.8672853 \ -0.0861103]$

These were different from the previous results when using batch gradient descent on the non overlapping classes. The loss comparison is as follows.

Weights	Non-overlapping Weights	Overlapping Weights
$w_0$	0.08534997	-0.06405436
$w_1$	0.78405749	0.8672853
$w_2$	0.5748131	-0.0861103

Table 5: Weights after 20 iterations



- Previously the classes were far apart so we can find a line that separate them easily. But now there is no line that can perfectly separate them.
- Now the convergence is happening more slowly compared to the non overlapping classes scenario. The reason for this is that, in the overlapping region the model will be more uncertain and the predicted probability will be around 0.5.
- So, for uncertain points the gradient will be a smaller value resulting in the weight updates being smaller. Also the loss function can become more flat around the optimal solution reducing the gradient and slowing the convergence.
- Hence we can see that the convergence speed has significantly reduced.