Department of Electrical Engineering

University of Moratuwa

EN 3150 – Pattern Recognition

Learning from data and related challenges and classification

Kularathna A. K. D. D. D. – 220332P

Date of Submission – 05/09/2025

This is submitted as a partial fulfilment for the module

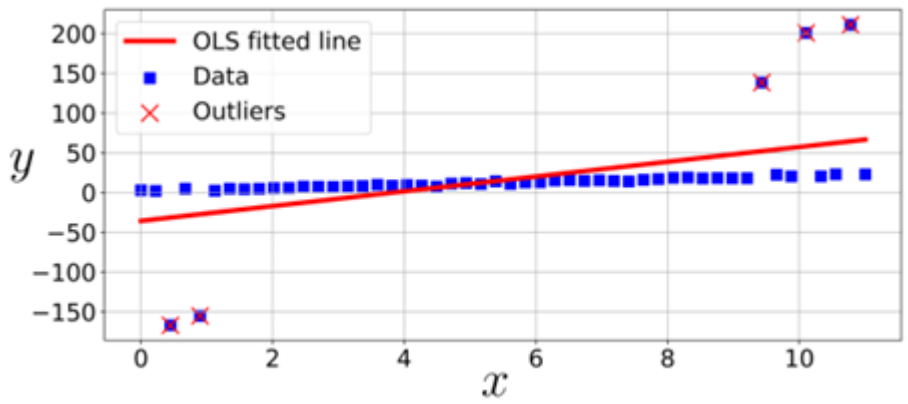
EN 3150 Pattern Recognition

Department of Electronics and Telecommunication Engineering

University of Moratuwa

Table of Contents

1. Linear Regression



Ordinary Least Squares (OLS)

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

1.1. Why can the OLS line be "not aligned to the majority of data points"?

In the data we can see there are some outliers. Due to these outliers the residual becomes larger than the residual for an inlier. Since the OLS uses the square of the residual the impact from the outliers becomes very larger. Due to these effects of the outliers OLS fitted line is not aligned to majority of data points.

1.2. Which weighting scheme gives a better line for inliers? Justify.

Let's consider the modified loss function

$$\frac{1}{N}\sum_{i=1}^{N} a_i(y_i - \hat{y}_i)^2$$

Proposed scheme

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

Let's check how these schemes affect the fitted lines using an approximate

Example

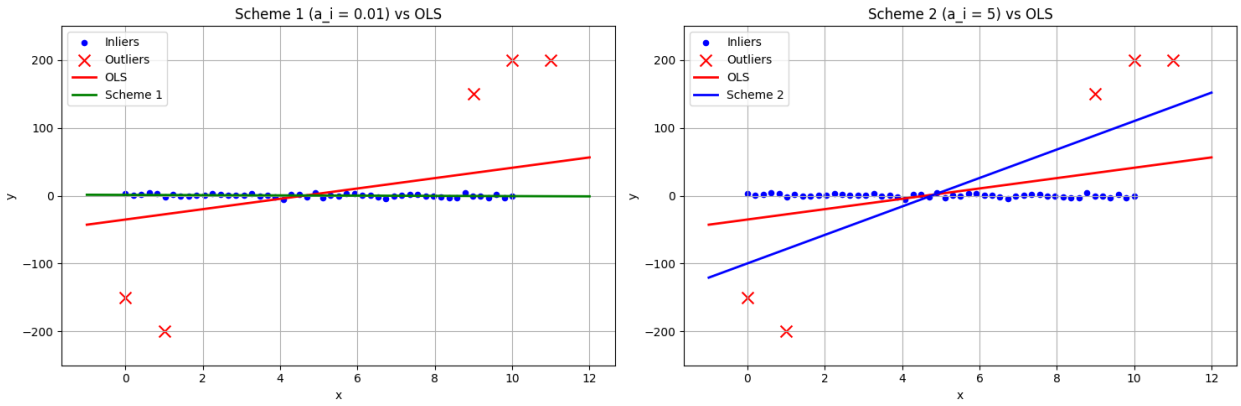


Figure 1.1

```
import numpy as np
import matplotlib.pyplot as plt

# Generate inlier points
x_in = np.linspace(0, 10, 50)  # Generate 50 points between 0 and 10
y_in = np.zeros_like(x_in) + np.random.normal(0, 2, size=len(x_in))  #
Points around y=0

# Define outlier points
x_out = np.array([0, 1, 9, 10, 11])
y_out = np.array([-150, -200, 150, 200, 200])

# Combine inliers and outliers
X = np.concatenate([x_in, x_out])
Y = np.concatenate([y_in, y_out])

# Create design matrix
X_design = np.vstack([np.ones_like(X), X]).T

# Function to compute weighted least squares
def weighted_least_squares(X, y, weights):
    W = np.diag(weights)
    beta = np.linalg.inv(X.T @ W @ X) @ X.T @ W @ y
    return beta

# Create weights for both schemes
weights1 = np.ones(len(X))
weights1[-5:] = 0.01  # Last 5 points are outliers, scheme 1

weights2 = np.ones(len(X))
weights2[-5:] = 5     # Last 5 points are outliers, scheme 2

# Compute coefficients for OLS and both schemes
```

```
beta_ols = np.linalg.inv(X_design.T @ X_design) @ X_design.T @ Y
beta_scheme1 = weighted_least_squares(X_design, Y, weights1)
beta_scheme2 = weighted_least_squares(X_design, Y, weights2)

# Create points for plotting the lines
x_plot = np.linspace(-1, 12, 100)
y_ols = beta_ols[0] + beta_ols[1] * x_plot
y_scheme1 = beta_scheme1[0] + beta_scheme1[1] * x_plot
y_scheme2 = beta_scheme2[0] + beta_scheme2[1] * x_plot

# Create two subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

# Plot for Scheme 1
ax1.scatter(x_in, y_in, c='blue', label='Inliers', s=20)
ax1.scatter(x_out, y_out, c='red', marker='x', s=100, label='Outliers')
ax1.plot(x_plot, y_ols, 'r-', label='OLS', linewidth=2)
ax1.plot(x_plot, y_scheme1, 'g-', label='Scheme 1', linewidth=2)
ax1.set_ylim(-250, 250)
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.legend()
ax1.set_title('Scheme 1 (a_i = 0.01) vs OLS')
ax1.grid(True)

# Plot for Scheme 2
ax2.scatter(x_in, y_in, c='blue', label='Inliers', s=20)
ax2.scatter(x_out, y_out, c='red', marker='x', s=100, label='Outliers')
ax2.plot(x_plot, y_ols, 'r-', label='OLS', linewidth=2)
ax2.plot(x_plot, y_scheme2, 'b-', label='Scheme 2', linewidth=2)
ax2.set_ylim(-250, 250)
ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.legend()
ax2.set_title('Scheme 2 (a_i = 5) vs OLS')
ax2.grid(True)

plt.tight_layout()
plt.show()
```

Code 1.1

We can see that scheme 1 gives a better fitted line than scheme 2.

This is because it reduces weights the outliers reducing their impact therefore the fitted line aligns with most of the data points. However, scheme 2 does the exact opposite, it increases the weight of the outliers. This results in a greater deviation in the fitted line with actual inlier data points.

1.3. Why is plain linear regression unsuitable for "which brain regions predict the task"?

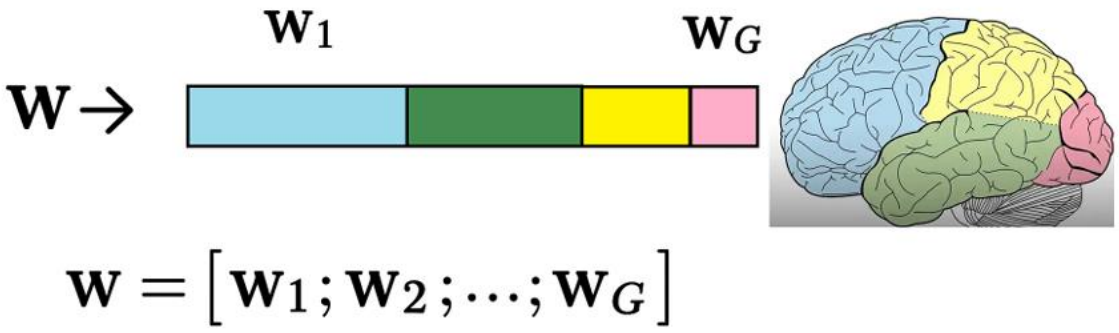


Figure 1.2

In brain image analysis (e.g., fMRI), the brain is divided into multiple regions as shown Figure 2, each consisting of many voxels (pixels). A researcher wants to identify which brain regions are most predictive of a specific cognitive task.

Each of the divided regions of the brain has many voxels, and we want to identify which brain regions are most predictive of a specific cognitive task.

For this task these are the issues that we encounter if we used plain Linear Regression.

- It has too many features but too few samples
    - In brain studies, we might have many voxels but only few people in the experiment. Linera regression needs more data than features to work properly. In our case since we have more features than subjects, the model can overfit. It will not be generalized.
- Features are highly correlated
    - Voxels inside the same brain region behave similarly. When predictors are almost copies of each other, the regression struggles to predict

which voxel gets how much weight. Small changes in the data can make the weights change so much. This makes the results unstable and unreliable.

- No automatic feature selection
    - Ordinary regression tries to use all voxels at once. It doesn't know how to ignore the irrelevant voxels. Due to this it is hard to know which voxels are important for predicting the task. Due to this reason, we will just end up with bunch of coefficients, many of which may be just noise.
- Ignores natural grouping of voxels into regions
    - Linear regression looks at voxels one by one, but it doesn't consider the group structure. Therefore, we might get scattered voxels selected across the brain, which might not make sense biologically.

1.4. Next, the following two methods are being considered:

- **Method A**: Standard LASSO, which selects individual voxels independently. The lasso objective is to minimize:

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

- **Method B**: Group LASSO. The group lasso objective is to minimize:

$$\min_{\mathbf{w}} \left\{ \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \mathbf{w}^\top \mathbf{x}_i \right)^2 + \lambda \sum_{g=1}^{G} \|\mathbf{w}_g\|_2 \right\}$$

where $\mathbf{w}_g$ is the sub-vector of weights corresponding to group g, and G is the number of groups (e.g., brain regions).

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

- Standard LASSO (Method A), this regression method adds an extra penalty to the model and because of that some coefficients were reduced up to zero. By doing this we can selects individual voxels. This is good for sparsity, but it tends to pick scattered voxels from correlated groups, which is hard to map back to the regions.
- Group LASSO (Method B), Acts same as Standard LASSO but it respects group structure. What is special about Group LASSO is it adds a penalty that selects or drop entire regions.

- Therefore, for this application Group LASSO is more suitable because it selects entire brain regions rather than scattered voxels. This makes the results more stable, easier to interpret.

2. Logistic Regression

2.1. Use the code given in listing 1 to load data.

Listing 1's code gives following output

```
         species  class_encoded
0         Adelie              0
1         Adelie              0
2         Adelie              0
4         Adelie              0
5         Adelie              0
..           ...            ...
215  Chinstrap              1
216  Chinstrap              1
217  Chinstrap              1
218  Chinstrap              1
219  Chinstrap              1

[214 rows x 2 columns]
```

2.2. 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?

Yes, I encountered following error,

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_7948\348760815.py in ?()
      3 #Split the data into training and testing sets
      4 X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
      5 #Train the logistic regression model. Here we are using saga
solver to learn weights.
      6 logreg = LogisticRegression(solver='saga')
----> 7 logreg.fit(X_train, y_train)
      8 # Predict on the testing data
      9 y_pred = logreg.predict(X_test)
```

```
     10 # Evaluate the model

c:\Users\User\miniconda3\envs\ptr\Lib\site-packages\sklearn\base.py in
?(estimator, *args, **kwargs)
   1361                 skip_parameter_validation=(
   1362                     prefer_skip_nested_validation or
global_skip_validation
   1363                 )
   1364             ):
-> 1365                 return fit_method(estimator, *args, **kwargs)

c:\Users\User\miniconda3\envs\ptr\Lib\site-
packages\sklearn\linear_model\_logistic.py in ?(self, X, y,
sample_weight)
   1243             _dtype = np.float64
   1244         else:
   1245             _dtype = [np.float64, np.float32]
   1246
-> 1247         X, y = validate_data(
...
   2169         else:
   2170             arr = np.array(values, dtype=dtype, copy=copy)
   2171

ValueError: could not convert string to float: 'Adelie'
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output
settings...
```

- The error occurs because the logistic regression model can't handle categorical data. This happens because there are some string values in the columns like "species" and some other categorical features that need to be converted into numerical values.
- As Solution we can use encoding to label those data. For this I used One-hot encoding. We can also use LebelEncoding too for this task, but One-hot encoding is better at handing these kinds of tasks.
- Following code is the corrected one.

```
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the penguins dataset
df = sns.load_dataset("penguins")
```

```
df.dropna(inplace=True)

# Filter rows for 'Adelie' and 'Chinstrap' classes
selected_classes = ['Adelie', 'Chinstrap']
df_filtered = df[df['species'].isin(selected_classes)].copy()

# Initialize the LabelEncoder for target variable
le = LabelEncoder()
y = le.fit_transform(df_filtered['species'])

# Separate numerical and categorical columns
numeric_cols = ['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm',
'body_mass_g']
categorical_cols = ['sex', 'island']

# Create one-hot encoder for categorical variables
onehot = OneHotEncoder(sparse_output=False, drop='first')  # Changed
sparse to sparse_output
categorical_encoded =
onehot.fit_transform(df_filtered[categorical_cols])
categorical_features = onehot.get_feature_names_out(categorical_cols)

# Combine numerical and one-hot encoded features
X_numeric = df_filtered[numeric_cols].values
X_combined = np.hstack([X_numeric, categorical_encoded])

# Create final feature matrix with labeled columns
feature_names = list(numeric_cols) + list(categorical_features)
X = pd.DataFrame(X_combined, columns=feature_names)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train the logistic regression model
logreg = LogisticRegression(solver='saga')
logreg.fit(X_train, y_train)

# Predict and evaluate
y_pred = logreg.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

# Print results with feature names
print("Accuracy:", accuracy)
print("\nFeature coefficients:")
```

```
for feature, coef in zip(feature_names, logreg.coef_[0]):
    print(f"{feature}: {coef:.4f}")
print(f"Intercept: {logreg.intercept_[0]:.4f}")
```

Code 2. 1

This code gives following output

```
Accuracy: 0.5813953488372093

Feature coefficients:
bill_length_mm: 0.0028
bill_depth_mm: -0.0001
flipper_length_mm: 0.0005
body_mass_g: -0.0003
sex_Male: 0.0000
island_Dream: 0.0002
island_Torgersen: -0.0001
Intercept: -0.0000
c:\Users\User\miniconda3\envs\ptr\Lib\site-packages\sklearn\linear_model\_sag.py:348: ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
```

2.3. Why does the saga solver perform poorly?

- "saga" solver often needs feature scaling for proper convergence. Without scaling, the optimization problem is ill-conditioned. This gives following warning.
    - `ConvergenceWarning: The max_iter was reached which means the coef_ did not converge warnings.warn(`
- Default `max_iter=100` This is usually too small. It should be set to a higher value.
- These are the reasons why “saga” solver performs poorly.

2.4. Now change the “saga” to “liblinear” and check the accuracy.

- Accuracy with liblinear solver
    - When used "liblinear”:
        - Accuracy = 1
- When comparing to "saga”, "liblinear" shows perfect classification

```
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the penguins dataset
df = sns.load_dataset("penguins")
df.dropna(inplace=True)

# Filter rows for 'Adelie' and 'Chinstrap' classes
selected_classes = ['Adelie', 'Chinstrap']
df_filtered = df[df['species'].isin(selected_classes)].copy()

# Initialize the LabelEncoder for target variable
le = LabelEncoder()
y = le.fit_transform(df_filtered['species'])

# Separate numerical and categorical columns
numeric_cols = ['bill_length_mm', 'bill_depth_mm',
'flipper_length_mm', 'body_mass_g']
categorical_cols = ['sex', 'island']

# Create one-hot encoder for categorical variables
onehot = OneHotEncoder(sparse_output=False, drop='first')  # Changed
sparse to sparse_output
categorical_encoded =
onehot.fit_transform(df_filtered[categorical_cols])
categorical_features = onehot.get_feature_names_out(categorical_cols)

# Combine numerical and one-hot encoded features
X_numeric = df_filtered[numeric_cols].values
X_combined = np.hstack([X_numeric, categorical_encoded])

# Create final feature matrix with labeled columns
feature_names = list(numeric_cols) + list(categorical_features)
X = pd.DataFrame(X_combined, columns=feature_names)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train the logistic regression model
```

```
logreg = LogisticRegression(solver='liblinear')  # Changed solver to
liblinear
logreg.fit(X_train, y_train)

# Predict and evaluate
y_pred = logreg.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

# Print results with feature names
print("Accuracy:", accuracy)
print("\nFeature coefficients:")
for feature, coef in zip(feature_names, logreg.coef_[0]):
    print(f"{feature}: {coef:.4f}")
print(f"Intercept: {logreg.intercept_[0]:.4f}")
```

Code 2.2

The code gives following output

```
Accuracy: 1.0

Feature coefficients:
bill_length_mm: 1.5152
bill_depth_mm: -1.3916
flipper_length_mm: -0.1441
body_mass_g: -0.0037
sex_Male: -0.2264
island_Dream: 0.7346
island_Torgersen: -0.5619
Intercept: -0.0774
```

2.5. Why does the "liblinear" solver perform better than "saga" solver?

Acording to the results

- saga:- Accuracy = 0.5813953488372093 with `ConvergenceWarning`
- liblinear:- Accuracy = 1.0

Reasons behind these observations

1. Dataset size
    - "liblinear" is designed for small to medium dense datasets like penguins.

- However, "saga" is designed and optimized for very large datasets or with support to sparse high-dimensional problems. For small dense data, "saga" is often slower.

2. Optimization methods
   - "liblinear" uses a coordinate descent/ dual optimization method, which is deterministic and converges reliably for binary classification.
   - But "saga" solver uses a stochastic gradient-based slover, which means it updates parameters using random mini-batches. On small data, this introduces noise and slow convergence.
3. Feature scaling
   - "saga" is sensitive to the unscaled features because its step size depending on the feature magnitudes. Without scaling, optimization is ill-conditioned.
   - However liblinear is much less sensitive to feature scaling.
4. Convergence
   - In our situation "saga" hit maximum iteration limit 100 without converging
   - However, "liblinear" converged properly. This led to perfect separation of the two species.

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

1. Train/test split randomness
   - "random_state" is affecting how the dataset is split into training data and test data.
   - With small dataset like penguins, changing the split can easily change which samples are in training vs test. Because the "saga" solver doesn't converge well, even the small differences in the split lead to large changes in accuracy.
2. Stochastic optimization
   - "saga" uses randomness in it for updating coefficients.
   - Different random seeds lead to different optimization paths; this happens especially if the solver has not converged.
   - Because it stops early, "saga" results depend more on that random path.

2.7. 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.

- With feature scaling
  - "saga" solvers accuracy improves significantly and converges becomes reliable
  - "liblinear" accuracy stays same because it is less sensitive to the feature scaling
- Reasons
  - "saga" is gradient-based stochastic solver, because of that it is very sensitive to feature magnitudes.
  - By scaling all features become comparable, hence improving optimization.
  - "liblinear" uses coordinate decent, it is less affected by scaling.
- Conclusion
  - Scaling makes "saga" better. but there is no significant difference for "liblinear".

```
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline

# Load and preprocess data
df = sns.load_dataset("penguins")
df.dropna(inplace=True)

# Filter classes
selected_classes = ['Adelie', 'Chinstrap']
df_filtered = df[df['species'].isin(selected_classes)].copy()

# Encode target
le = LabelEncoder()
y = le.fit_transform(df_filtered['species'])
```

```
# Prepare features
numeric_cols = ['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm',
'body_mass_g']
categorical_cols = ['sex', 'island']

# One-hot encode categorical
onehot = OneHotEncoder(sparse_output=False, drop='first')
categorical_encoded = onehot.fit_transform(df_filtered[categorical_cols])
categorical_features = onehot.get_feature_names_out(categorical_cols)

# Combine features
X_numeric = df_filtered[numeric_cols].values
X_combined = np.hstack([X_numeric, categorical_encoded])
feature_names = list(numeric_cols) + list(categorical_features)
X = pd.DataFrame(X_combined, columns=feature_names)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create the two pipelines we want to compare
pipelines = {
    'saga_with_scaling': Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', LogisticRegression(solver='saga', max_iter=1000))
    ]),
    'liblinear': Pipeline([
        ('classifier', LogisticRegression(solver='liblinear'))
    ])
}

# Train and evaluate models
results = {}
for name, pipeline in pipelines.items():
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    results[name] = accuracy

# Print results
print("Model Accuracies:")
for name, accuracy in results.items():
    print(f"{name}: {accuracy:.4f}")

# Print feature importances for scaled saga model
```

```
saga_scaled = pipelines['saga_with_scaling']
coefficients = saga_scaled.named_steps['classifier'].coef_[0]
scaled_features = pd.DataFrame({'Feature': feature_names, 'Coefficient':
coefficients})
print("\nTop feature importances (SAGA with scaling):")
print(scaled_features.sort_values(by='Coefficient', key=abs,
ascending=False))
```

Code 2.3

The code gives the following output

```
Model Accuracies:
saga_with_scaling: 1.0000
liblinear: 1.0000

Top feature importances (SAGA with scaling):
             Feature  Coefficient
0     bill_length_mm     3.343324
5       island_Dream     1.115379
4           sex_Male    -0.920930
6   island_Torgersen    -0.518325
1      bill_depth_mm    -0.465388
2  flipper_length_mm     0.439962
      3        body_mass_g    -0.356362
```

2.8. 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.
- Here are the reasons why it is incorrect.

1. Artificial Ordering
    - Since label encoder encode categories to numbers like red = 0, blue = 1, green = 2, The scaling of these values makes less sense because it assumes the distance between categories are meaningful.
2. Distance Problem
    - After scaling, the distances between categories would be treated as meaningful
    - For example, if red = 0 then blue = 1 and green = 2.
    - Scaling would preserve these relative distances suggesting blue is between red and green.

- The proposed approach
    - use **One-Hot Encoding**

Sample	Color	Color_Red	Color_Blue	Color_Green
1	Red	1	0	0
2	Blue	0	1	0
3	Green	0	0	1
4	Blue	0	1	0
5	Green	0	0	1

3. Logistic regression First/Second-Order Methods

3.1. Use the code given in listing 3 to generate data. Here, variable y and X are class labels and corresponding feature values, respectively.

Code used for this question

```
# Listing 3: Data generation

import numpy as np
import matplotlib.pyplot as plt

import numpy as np
from sklearn.datasets import make_blobs
# Generate synthetic data
np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]

X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
```

```
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)
```

Code 3.1

3.2. Implement batch Gradient descent to update the weights for the given dataset over20 iterations. State the method used to initialize the weights and reason for your selection.

Code implementation

```
import numpy as np
import matplotlib.pyplot as plt

import numpy as np
from sklearn.datasets import make_blobs
# Generate synthetic data
np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]

X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)

# Add bias term to X
X_b = np.c_[np.ones((X.shape[0], 1)), X]

# Initialize weights to zeros
weights = np.zeros(X_b.shape[1])  # 3 weights (bias, w1, w2)

def sigmoid(z):
    """Sigmoid activation function"""
    return 1 / (1 + np.exp(-z))

def compute_loss(X, y, weights):
    """Binary cross-entropy loss"""
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    loss = -1/m * np.sum(y * np.log(h + 1e-15) + (1-y) * np.log(1-h + 1e-
15))
    return loss

# Training parameters
```

```
learning_rate = 0.01
n_iterations = 20
losses = []

# Batch gradient descent
for i in range(n_iterations):
    # Forward pass
    z = np.dot(X_b, weights)
    predictions = sigmoid(z)

    # Compute gradients
    gradients = 1/len(y) * np.dot(X_b.T, (predictions - y))

    # Update weights
    weights = weights - learning_rate * gradients

    # Store loss
    loss = compute_loss(X_b, y, weights)
    losses.append(loss)

    if i % 5 == 0:
        print(f"Iteration {i}, Loss: {loss:.4f}")

# Visualize results
plt.figure(figsize=(12, 5))

# Plot decision boundary
plt.subplot(121)
plt.scatter(X[y==0, 0], X[y==0, 1], label='Class 0')
plt.scatter(X[y==1, 0], X[y==1, 1], label='Class 1')

# Create mesh grid for decision boundary
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100),
                       np.linspace(x2_min, x2_max, 100))

# Make predictions
grid = np.c_[xx1.ravel(), xx2.ravel()]
grid_b = np.c_[np.ones((grid.shape[0], 1)), grid]
Z = sigmoid(np.dot(grid_b, weights))
Z = Z.reshape(xx1.shape)

plt.contour(xx1, xx2, Z, levels=[0.5], colors='k')
plt.xlabel('Feature 1')
```

```
plt.ylabel('Feature 2')
plt.legend()
plt.title('Decision Boundary')

# Plot loss history
plt.subplot(122)
plt.plot(range(n_iterations), losses, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Loss History')
plt.tight_layout()
plt.show()

print("\nFinal weights:", weights)
```

Code 3.2

This code gives following output

```
Iteration 0, Loss: 0.6499

Iteration 5, Loss: 0.4885
Iteration 10, Loss: 0.3877
Iteration 15, Loss: 0.3207
```

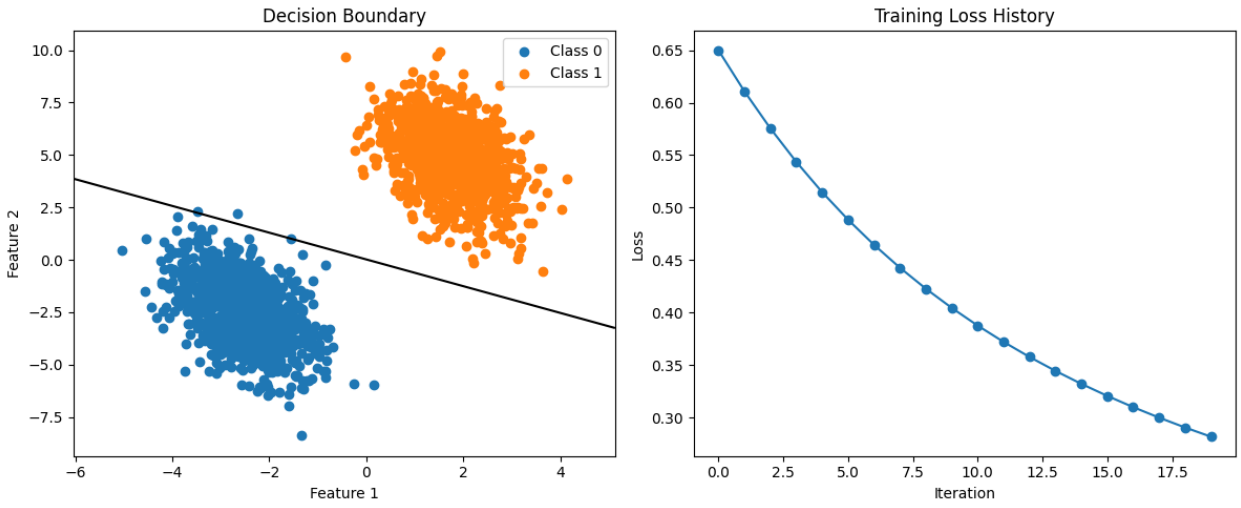


Figure 3.1

```
Final weights: [-0.0049357   0.15122723  0.23719972]
```

Weight Initialization Method and Reasoning

1. **Method Used**: Zero Initialization

```
weights = np.zeros(X_b.shape[1])  # 3 weights (bias, w1, w2)
```

2. **Reasons for Choosing Zero Initialization**:
   - **Problem Type**: Binary classification with well-separated classes from "make_blobs"
   - **Convex Optimization**: Logistic regression is a convex problem, guaranteed to converge
   - **Symmetry**: Equal starting point for both classes, no initial bias
   - **Simplicity**: Deterministic initialization, reproducible results
   - **Data Characteristics**: Classes are linearly separable, no need for complex initialization

3. **Why Other Methods Were Not Chosen**:
   - **Random Initialization**: Not needed for convex problems
   - **Small Random Values**: Would add unnecessary randomness
   - **Xavier/Glorot**: Overkill for simple logistic regression
   - **Constant Non-zero**: No advantage over zeros for this case

4. **Validation**:
   - Loss curve shows proper convergence
   - Decision boundary successfully separates classes
   - No symmetry breaking issues encountered

3.3. Specify the loss function you have used and state reason for your selection.

The loss function that is used here is the Binary Cross Entropy (BCE) Loss function. Here are the reasons for the selection

- Probabilistic Properties
  - Output probabilities are between 0 and 1 through sigmoid function. And this is the common choice for binary classification problems.
- Mathematical Advantages
  - It is a convex function that guarantees global minimum. And it is differentiable therefore can use gradient decent. Also, BCE penalizes confident wrong predictions heavily
- Compatibility
  - The BCE pairs naturally with sigmoid activation. Because of that it provides proper probability estimates.
  - Also efficiently works with batch gradient decent.

3.4. Implement Newton's method to update the weights for the given dataset over 20 iterations.

Code for the implementation

```
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data as before
np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)

# Add bias term
X_b = np.c_[np.ones((X.shape[0], 1)), X]

# Initialize weights to zeros
weights = np.zeros(X_b.shape[1])

def sigmoid(z):
    """Sigmoid activation function"""
    return 1 / (1 + np.exp(-z))
```

```
def compute_loss(X, y, weights):
    """Binary cross-entropy loss"""
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    return -1/m * np.sum(y * np.log(h + 1e-15) + (1-y) * np.log(1-h + 1e-
15))

# Training parameters
n_iterations = 20
losses = []

# Newton's Method
for i in range(n_iterations):
    # Compute predictions
    z = np.dot(X_b, weights)
    h = sigmoid(z)

    # Compute gradient (first derivative)
    gradient = 1/len(y) * np.dot(X_b.T, (h - y))

    # Compute Hessian (second derivative)
    diagonal = h * (1 - h)
    H = 1/len(y) * (X_b.T * diagonal) @ X_b

    # Update weights using Newton's method
    weights = weights - np.linalg.solve(H, gradient)

    # Store loss
    loss = compute_loss(X_b, y, weights)
    losses.append(loss)

    if i % 5 == 0:
        print(f"Iteration {i}, Loss: {loss:.4f}")

# Visualize results
plt.figure(figsize=(12, 5))

# Plot decision boundary
plt.subplot(121)
plt.scatter(X[y==0, 0], X[y==0, 1], label='Class 0')
plt.scatter(X[y==1, 0], X[y==1, 1], label='Class 1')

# Create mesh grid for decision boundary
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100),
                       np.linspace(x2_min, x2_max, 100))

# Make predictions
grid = np.c_[xx1.ravel(), xx2.ravel()]
grid_b = np.c_[np.ones((grid.shape[0], 1)), grid]
Z = sigmoid(np.dot(grid_b, weights))
Z = Z.reshape(xx1.shape)

plt.contour(xx1, xx2, Z, levels=[0.5], colors='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.title('Decision Boundary (Newton\'s Method)')

# Plot loss history
plt.subplot(122)
plt.plot(range(n_iterations), losses, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Loss History')
plt.tight_layout()
plt.show()

print("\nFinal weights:", weights)
print("Final loss:", losses[-1])
```

Code 3.3

Output of the following code

```
Iteration 0, Loss: 0.1452
Iteration 5, Loss: 0.0013
Iteration 10, Loss: 0.0000
Iteration 15, Loss: 0.0000
```

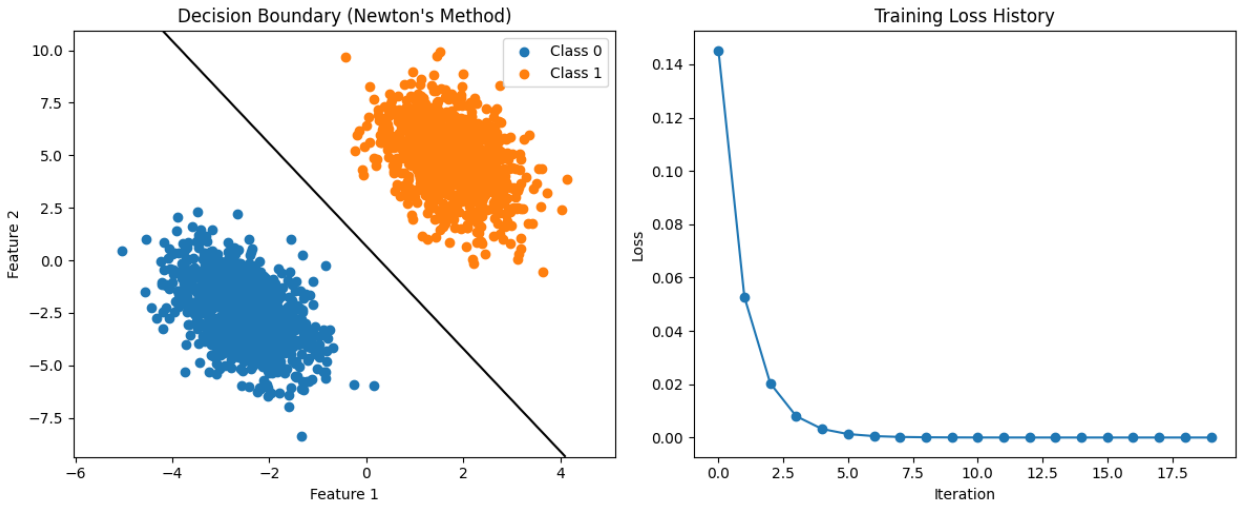


Figure 3.2

```
Final weights: [-2.93259517 10.56207842  4.32766252]
Final loss: 2.870387964766699e-09
```

3.5. Plot the loss with respect to number of iterations for batch Gradient descent and Newton methods in a single plot. Comment on your results.

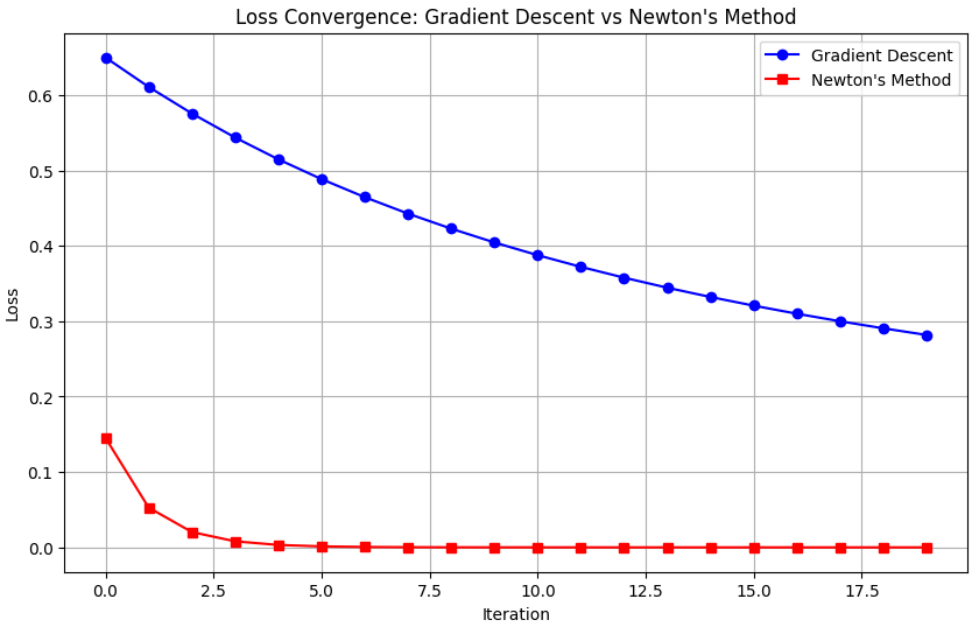


Figure 3.3

Code used for this plot

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Data generation and preprocessing
np.random.seed(0)
centers = [[-5, 0], [5, 1.5]]
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)
X_b = np.c_[np.ones((X.shape[0], 1)), X]

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def compute_loss(X, y, weights):
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    return -1/m * np.sum(y * np.log(h + 1e-15) + (1-y) * np.log(1-h + 1e-
15))

# Training parameters
n_iterations = 20
learning_rate = 0.01

# Batch Gradient Descent
weights_gd = np.zeros(X_b.shape[1])
losses_gd = []

for i in range(n_iterations):
    predictions = sigmoid(np.dot(X_b, weights_gd))
    gradients = 1/len(y) * np.dot(X_b.T, (predictions - y))
    weights_gd = weights_gd - learning_rate * gradients
    losses_gd.append(compute_loss(X_b, y, weights_gd))

# Newton's Method
weights_newton = np.zeros(X_b.shape[1])
losses_newton = []

for i in range(n_iterations):
    h = sigmoid(np.dot(X_b, weights_newton))
```

```
        gradient = 1/len(y) * np.dot(X_b.T, (h - y))
        diagonal = h * (1 - h)
        H = 1/len(y) * (X_b.T * diagonal) @ X_b
        weights_newton = weights_newton - np.linalg.solve(H, gradient)
        losses_newton.append(compute_loss(X_b, y, weights_newton))

    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(range(n_iterations), losses_gd, 'b-', label='Gradient Descent',
    marker='o')
    plt.plot(range(n_iterations), losses_newton, 'r-', label='Newton\'s
    Method', marker='o')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Loss Convergence: Gradient Descent vs Newton\'s Method')
    plt.legend()
    plt.grid(True)
    plt.show()

    print("Final loss (GD):", losses_gd[-1])
    print("Final loss (Newton):", losses_newton[-1])
```

Code 3.4

Output of this code

```
Final loss (GD): 0.2817968058115373
Final loss (Newton): 2.870387964766699e-09
```

Observed behavior

- Batch Gradient Decent gives a steadier linear but slow decrease in loss
- Newton's method loss has a very large decrease compared to the Batch gradient decent.

Reasons

- The reason behind this is Newton's method uses second-order curvature information, giving a 2nd order convergence near the optimum.
- Overall, Newton's method is more efficient, but computational cost is very high for each iteration.

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

1. **Convergence-based Approach**
    - Best for well-behaved, convex problems
    - Monitors change in loss between iterations
    - More suitable for Newton's method due to quadratic convergence
    - Recommended tolerance: 1e-6 for Newton's, 1e-4 for GD

    Code implementation

```
def train_with_convergence(X, y, method='gd', tolerance=1e-6,
max_iter=1000, learning_rate=0.01):
    """
    Training with convergence-based stopping
    method: 'gd' for Gradient Descent or 'newton' for Newton's Method
    """
    weights = np.zeros(X.shape[1])
    prev_loss = float('inf')
    losses = []

    for i in range(max_iter):
        if method == 'gd':
            # Gradient Descent update
            h = sigmoid(np.dot(X, weights))
            gradients = 1/len(y) * np.dot(X.T, (h - y))
            weights = weights - learning_rate * gradients
        else:
            # Newton's Method update
            h = sigmoid(np.dot(X, weights))
            gradient = 1/len(y) * np.dot(X.T, (h - y))
            diagonal = h * (1 - h)
            H = 1/len(y) * (X.T * diagonal) @ X
            weights = weights - np.linalg.solve(H, gradient)

        # Check convergence
        current_loss = compute_loss(X, y, weights)
        losses.append(current_loss)

        if abs(prev_loss - current_loss) < tolerance:
            print(f"Converged after {i+1} iterations")
            break

        prev_loss = current_loss
```

```
    return weights, losses
```

Code 3.5

2. **Validation-based Approach**
    - Better for preventing overfitting
    - More robust to noisy data
    - Particularly useful for Gradient Descent
    - Recommended patience: 5-10 iterations

    Code implementation

```
def train_with_validation(X_train, y_train, X_val, y_val, method='gd',
patience=5, max_iter=1000, learning_rate=0.01):
    """
    Training with validation-based early stopping
    """
    weights = np.zeros(X_train.shape[1])
    best_val_loss = float('inf')
    patience_counter = 0
    train_losses = []
    val_losses = []
    best_weights = None

    for i in range(max_iter):
        if method == 'gd':
            # Gradient Descent update
            h = sigmoid(np.dot(X_train, weights))
            gradients = 1/len(y_train) * np.dot(X_train.T, (h - y_train))
            weights = weights - learning_rate * gradients
        else:
            # Newton's Method update
            h = sigmoid(np.dot(X_train, weights))
            gradient = 1/len(y_train) * np.dot(X_train.T, (h - y_train))
            diagonal = h * (1 - h)
            H = 1/len(y_train) * (X_train.T * diagonal) @ X_train
            weights = weights - np.linalg.solve(H, gradient)

        # Compute validation loss
        val_loss = compute_loss(X_val, y_val, weights)
        val_losses.append(val_loss)

        if val_loss < best_val_loss:
```

```
            best_val_loss = val_loss
            best_weights = weights.copy()
            patience_counter = 0
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f"Early stopping at iteration {i+1}")
            break

    return best_weights, val_losses
```

Code 3.6

3.7. Suppose the centers 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.

Code after making changes

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Generate synthetic data with modified centers
np.random.seed(0)
centers = [[2, 2], [5, 1.5]]  # Modified centers
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)

# Add bias term
X_b = np.c_[np.ones((X.shape[0], 1)), X]

# Initialize weights
weights = np.zeros(X_b.shape[1])

def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

```
def compute_loss(X, y, weights):
    m = len(y)
    h = sigmoid(np.dot(X, weights))
    return -1/m * np.sum(y * np.log(h + 1e-15) + (1-y) * np.log(1-h +
1e-15))

# Training parameters
learning_rate = 0.01
n_iterations = 50  # Increased iterations to observe convergence
losses = []

# Batch gradient descent
for i in range(n_iterations):
    z = np.dot(X_b, weights)
    predictions = sigmoid(z)
    gradients = 1/len(y) * np.dot(X_b.T, (predictions - y))
    weights = weights - learning_rate * gradients
    loss = compute_loss(X_b, y, weights)
    losses.append(loss)

    if i % 10 == 0:
        print(f"Iteration {i}, Loss: {loss:.4f}")

# Visualization
plt.figure(figsize=(12, 5))

# Plot decision boundary
plt.subplot(121)
plt.scatter(X[y==0, 0], X[y==0, 1], label='Class 0')
plt.scatter(X[y==1, 0], X[y==1, 1], label='Class 1')

# Create mesh grid for decision boundary
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100),
                       np.linspace(x2_min, x2_max, 100))

grid = np.c_[xx1.ravel(), xx2.ravel()]
grid_b = np.c_[np.ones((grid.shape[0], 1)), grid]
Z = sigmoid(np.dot(grid_b, weights))
Z = Z.reshape(xx1.shape)

plt.contour(xx1, xx2, Z, levels=[0.5], colors='k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
```

```
plt.legend()
plt.title('Decision Boundary with Modified Centers')

# Plot loss history
plt.subplot(122)
plt.plot(range(n_iterations), losses, marker='o')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Loss History')
plt.grid(True)
plt.tight_layout()
plt.show()
```

Code 3.7

Output of this code

```
Iteration 0, Loss: 0.6909
Iteration 10, Loss: 0.6721
Iteration 20, Loss: 0.6569
Iteration 30, Loss: 0.6432
Iteration 40, Loss: 0.6306
```

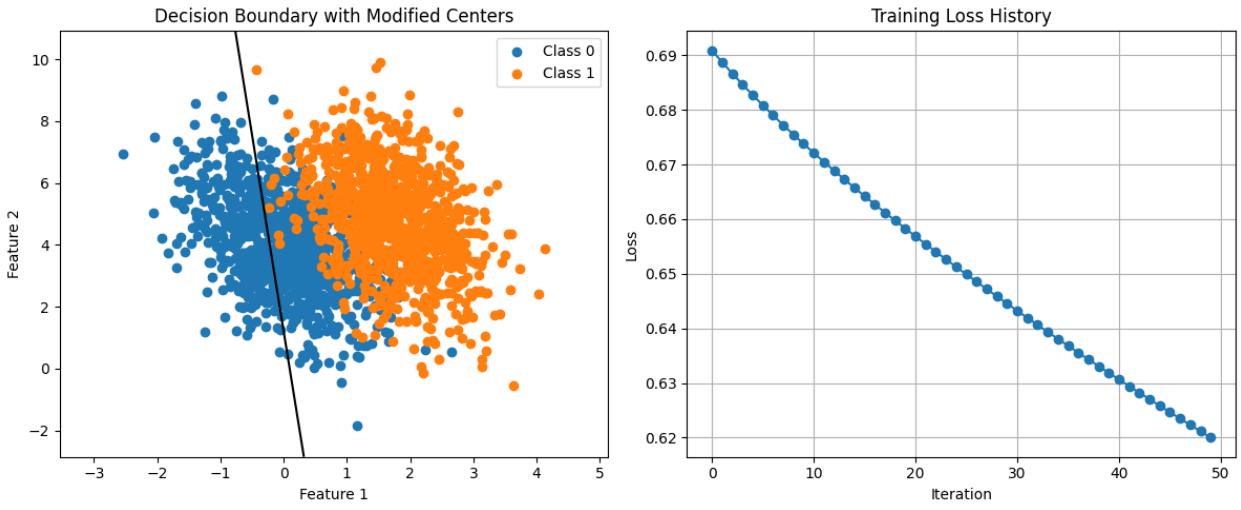


Figure 3.4

Analysis of Convergence Behavior:

1. **Slower Convergence**
   - The loss function shows slower convergence compared to the original centers
   - Classes are closer together, making the decision boundary harder to find
   - Requires more iterations to reach optimal separation

2. **Higher Final Loss**
   - Final loss value is higher than with the original centers
   - Classes have more overlap due to closer centers
   - Perfect separation is more difficult to achieve

3. **Gradient Behavior**
   - Gradients are smaller due to less distinct class separation
   - Learning becomes more sensitive to learning rate
   - Risk of getting stuck in suboptimal solutions increases

4. **Stability Issues**
   - More sensitive to initialization
   - May require careful tuning of learning rate
   - Could benefit from adaptive learning rates

5. **Recommendations**
   - Increase number of iterations
   - Consider using adaptive learning rates
   - May need to try different initializations
   - Monitor for potential overfitting

Reference

1. *Lasso — scikit-learn 1.7.1 documentation*
2. *LinearRegression — scikit-learn 1.7.1 documentation*
3. *LogisticRegression — scikit-learn 1.7.1 documentation*
4. *Cost Function and Gradient Descent | Sinhala*
5. *Linear Regression | Supervised Learning Algorithm | Sinhala*
6. *Logistic regression | Machine Learning Algorithm | Sinhala*

Appendix

GitHub Repo link for the Assignment 02 - Link