UNIVERSITY OF MORATUWA

Faculty of Engineering

Department of Electrical Engineering

EN3150 - Pattern Recognition

Assignment 02

Learning from Data and Related Challenges and Classification

Student Name:	M. Thiruvarankan
Index Number:	220647K
Degree Program:	B.Sc. Engineering (Hons)
Academic Year:	Year 3, Semester 05
Department:	Electrical Engineering

Submitted on: September 08, 2025

Academic Year 2024/2025

Faculty of Engineering, University of Moratuwa, Sri Lanka

1 Linear Regression

1.1 OLS Fitted Line Analysis [10 marks]

Question: What is the reason behind the OLS fitted line not being aligned to the majority of data points?

Answer:

The OLS fitted line is not aligned to the majority of data points due to the presence of **outliers**.

Mathematical Explanation: The OLS loss function is defined as:

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

Key Issues:

- **Outlier Sensitivity:** The squared residuals $(y_i - \hat{y}_i)^2$ give disproportionate weight to outliers (marked with $\times$ in Figure 1)
- **Global Optimization:** OLS minimizes the *total* squared error across all points, not the majority
- **Leverage Effect:** Outliers with extreme x or y values exert excessive influence on the regression line

Therefore, the fitted line is pulled away from the main cluster of data points to accommodate the outliers in the loss minimization process.

1.2 Weighted Regression Schemes [30 marks]

Question: Under which scheme do you expect a better fitted line for inliers than the OLS fitted line? Justify your answer.

Answer:

Scheme 1 will provide a significantly better fitted line for inliers.

Scheme Comparison:

$$\text{Scheme 1:} \quad a_i = \begin{cases} 0.01 & \text{for outliers} \\ 1.0 & \text{for inliers} \end{cases} \tag{2}$$

$$\text{Scheme 2:} \quad a_i = \begin{cases} 5.0 & \text{for outliers} \\ 1.0 & \text{for inliers} \end{cases} \tag{3}$$

Modified Loss Function:

$$L_{\text{weighted}}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} a_i(y_i - \hat{y}_i)^2 \tag{4}$$

Detailed Justification:

(a) **Scheme 1 - Outlier Downweighting:**

- Reduces outlier influence by factor of 100 ($a_i = 0.01$)

- Maintains normal weight for inliers ($a_i = 1.0$)
- Results in robust fitting focused on the main data distribution

(b) **Scheme 2 - Outlier Upweighting:**

- Amplifies outlier influence by factor of 5 ($a_i = 5.0$)
- Makes the regression line even more sensitive to outliers than standard OLS
- Worsens the fit quality for inliers

Conclusion: Scheme 1 implements a robust regression approach that naturally emphasizes the majority pattern while suppressing outlier interference.

1.3 Suitability of Linear Regression for Brain Analysis [20 marks]

Question: Why is linear regression not suitable for the brain region prediction task?

Answer:

Linear regression is fundamentally inappropriate for brain region prediction due to several critical mismatches between the algorithm assumptions and the problem structure.

Primary Issues:

1. **Problem Type Mismatch:**
   - *Required:* Multi-class classification (G discrete brain regions)
   - *Provided:* Continuous regression output $\mathbb{R}$
2. **Output Space Incompatibility:**

$$\text{Brain regions:} \quad \{W_1, W_2, \ldots, W_G\} \quad \text{(discrete)} \tag{5}$$

$$\text{Linear regression:} \quad \mathbb{R} \quad \text{(continuous)} \tag{6}$$

3. **Mathematical Structure Violation:**
   - No meaningful linear relationship between voxel intensities and categorical region labels
   - Brain regions represent qualitative distinctions, not quantitative scales
4. **Decision Boundary Limitations:**
   - Linear regression produces hyperplane decision boundaries
   - Brain regions often have complex, non-linear spatial boundaries

Appropriate Alternatives: Multinomial logistic regression, support vector machines, or neural networks would be more suitable for this classification task.

1.4 LASSO vs Group LASSO Comparison [40 marks]

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

Answer:

Group LASSO (Method B) is significantly more appropriate for brain image analysis applications.

Mathematical Formulations:

$$\text{Method A (Standard LASSO):} \quad \min_{\mathbf{w}} \left\{ \frac{1}{N} \sum_{i=1}^{N} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \|\mathbf{w}\|_1 \right\} \tag{7}$$

$$\text{Method B (Group LASSO):} \quad \min_{\mathbf{w}} \left\{ \frac{1}{N} \sum_{i=1}^{N} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \sum_{g=1}^{G} \|\mathbf{w}_g\|_2 \right\} \tag{8}$$

where $\mathbf{w}_g$ represents the weight sub-vector for brain region g, and G is the total number of brain regions.

Comprehensive Justification:

1. **Structural Coherence:**
   - Brain voxels within anatomical regions exhibit high spatial correlation
   - Group LASSO leverages this natural clustering via ℓ_2 group penalties
   - Promotes selection of entire brain regions rather than scattered voxels
2. **Biological Interpretability:**
   - Identifies *which brain regions* are predictive of cognitive tasks
   - Aligns with neuroscientific understanding of functional brain organization
   - Provides clinically meaningful insights for medical applications
3. **Statistical Advantages:**
   - **Reduced Overfitting:** Group structure constraints reduce effective parameter space
   - **Stability:** More robust feature selection across different data samples
   - **Sparsity Control:** Achieves region-level sparsity while maintaining within-region density
4. **Standard LASSO Limitations:**
   - Treats each voxel independently, ignoring spatial brain architecture
   - May select isolated voxels that lack neurological significance
   - Can produce fragmented, difficult-to-interpret feature maps
5. **Computational Efficiency:**
   - Reduces the effective search space from individual voxels to brain regions
   - Faster convergence due to structured regularization
   - More stable optimization landscape

Conclusion: Group LASSO's ability to respect the inherent spatial organization of brain data makes it the superior choice for neuroscience applications, providing both better predictive performance and more meaningful scientific insights.

2 Logistic Regression

Implementation Code: View on GitHub

2.1 Data Loading and Preprocessing

The code provided in Listing 1 successfully loads and preprocesses the penguins dataset, filtering for 'Adelie' and 'Chinstrap' species and encoding the target variable using LabelEncoder.

2.2 Error Analysis and Resolution [20 marks]

Question: Did you encounter any errors? If yes, what were they, and how would you go about resolving them?

Answer:

Yes, the following error was encountered:

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

Root Cause: The error occurs because the dataset contains **categorical (string) variables** that cannot be directly processed by scikit-learn's LogisticRegression algorithm, which expects **numerical input only**.

Specific Issues in the Data:

- **Island column:** Contains categorical strings ('Torgersen', 'Biscoe', 'Dream')
- **Sex column:** Contains string values ('Male', 'Female')
- **Species column:** Still present in feature matrix despite being the target

Resolution Steps:

Step 1: Remove Target from Features

```
# Remove species column from features
X = df_filtered.drop(['class_encoded', 'species'], axis=1)
```

Step 2: Apply One-Hot Encoding

```
# One-hot encode categorical variables
X_encoded = pd.get_dummies(X, drop_first=True)
X_encoded = X_encoded.astype(int)
```

2.3 SAGA Solver Performance [15 marks]

Question: Why does the SAGA solver perform poorly?

Answer:

The SAGA solver performs poorly due to several key factors:

Primary Issues:

1. **Feature Scaling Sensitivity:**
   - SAGA is sensitive to features with different scales
   - Penguins dataset has varying feature ranges (e.g., body mass vs flipper length)
   - Unscaled features cause convergence difficulties
2. **Convergence Problems:**
   - SAGA may not converge within default iteration limits
   - Poor initialization can lead to suboptimal solutions
   - Algorithm gets stuck in local minima
3. **Algorithm Characteristics:**
   - SAGA is designed for large datasets with regularization
   - Small dataset size makes the algorithm less effective
   - Stochastic nature introduces unnecessary variability

2.4 LibLinear Classification Accuracy [5 marks]

Question: What is the classification accuracy with liblinear configuration?

Answer:

After changing to liblinear solver:

```
logreg = LogisticRegression(solver='liblinear')
```

Classification Accuracy: 100% (1.0)

This represents perfect classification on the test set, indicating that the liblinear solver successfully separates the two penguin species.

2.5 LibLinear vs SAGA Performance [15 marks]

Question: Why does the liblinear solver perform better than SAGA solver?

Answer:

LibLinear outperforms SAGA for several reasons:

1. **Algorithm Design:**
   - LibLinear uses coordinate descent optimization
   - More stable and reliable for small to medium datasets
   - Better suited for binary classification problems
2. **Convergence Properties:**
   - Faster convergence on linearly separable data

- Less sensitive to feature scaling issues
- More robust initialization procedures

3. **Dataset Compatibility:**
   - Penguin species are likely linearly separable
   - Small dataset size favors deterministic algorithms
   - Binary classification is LibLinear's strength
4. **Numerical Stability:**
   - Better handling of numerical precision issues
   - More consistent results across different runs
   - Reduced sensitivity to parameter initialization

2.6 Accuracy Variation with Random States [15 marks]

Question: Why does the model's accuracy (with SAGA solver) vary with different random state values?

Answer:

The accuracy variation with different random states occurs due to:

1. **Stochastic Algorithm Nature:**
   - SAGA uses random sampling in its optimization process
   - Different random seeds lead to different optimization paths
   - Convergence to different local optima
2. **Data Split Sensitivity:**
   - Random state affects train-test split composition
   - Different test samples may be easier/harder to classify
   - Small dataset amplifies split-dependent variations
3. **Initialization Effects:**
   - Random weight initialization varies with seed
   - Poor initialization can trap algorithm in suboptimal regions
   - Convergence quality depends on starting point
4. **Non-Convergence Issues:**
   - SAGA may not fully converge within iteration limits

- Premature stopping at different solution qualities
- Inconsistent final model parameters

2.7 Feature Scaling Impact [15 marks]

Question: Compare the performance of liblinear and SAGA solvers with feature scaling.

Answer:

Implementation:

```
from sklearn.preprocessing import StandardScaler

# Apply feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Performance Comparison:

Solver	Without Scaling	With Scaling
LibLinear	100%	100%
SAGA	Variable (60-90%)	100%

Significant Difference Explanation:

- **SAGA shows dramatic improvement** with scaling (unstable $\rightarrow$ 100%)
- **LibLinear remains unchanged** (already optimal)
- Feature scaling normalizes different measurement units (grams vs millimeters)
- SAGA's gradient-based optimization requires similar feature scales for proper convergence
- Unscaled features create ill-conditioned optimization landscape for SAGA

2.8 Categorical Feature Scaling [15 marks]

Question: Is applying scaling to label-encoded categorical features correct? What do you propose?

Answer:

No, this approach is incorrect.

Problems with Scaling Label-Encoded Categories:

1. **Artificial Ordinality:**
   - Label encoding creates arbitrary numerical order (red=0, blue=1, green=2)
   - Scaling treats categories as if they have meaningful distances
   - Introduces false relationships between categories
2. **Loss of Categorical Nature:**

- Categories become continuous values after scaling
- Algorithm may interpolate between categories
- Loses the discrete, mutually exclusive property

Proposed Solution - One-Hot Encoding:

```
# Correct approach
X_encoded = pd.get_dummies(X, drop_first=True)
X_encoded = X_encoded.astype(int)

# Creates binary features:
# island_Dream, island_Torgersen, sex_Male
# Each category becomes a separate binary column
```

Advantages:

- Preserves categorical independence
- No artificial ordering imposed
- Compatible with feature scaling (binary 0/1 values)
- `drop_first=True` prevents multicollinearity
- Maintains interpretability for each category

3 First/Second-Order Methods

Implementation Code: View on GitHub

3.1 Data Generation

The synthetic data is generated using the provided code with blob centers and linear transformation to create a binary classification problem.

3.2 Batch Gradient Descent Implementation [20 marks]

Question: Implement batch Gradient descent over 20 iterations. State the weight initialization method and reason for selection.

Answer:

Implementation:

```
import numpy as np

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-np.clip(z, -250, 250)))

# Weight initialization
np.random.seed(0)
weights = np.random.randn(X.shape[1]) * 0.01
bias = 0
```

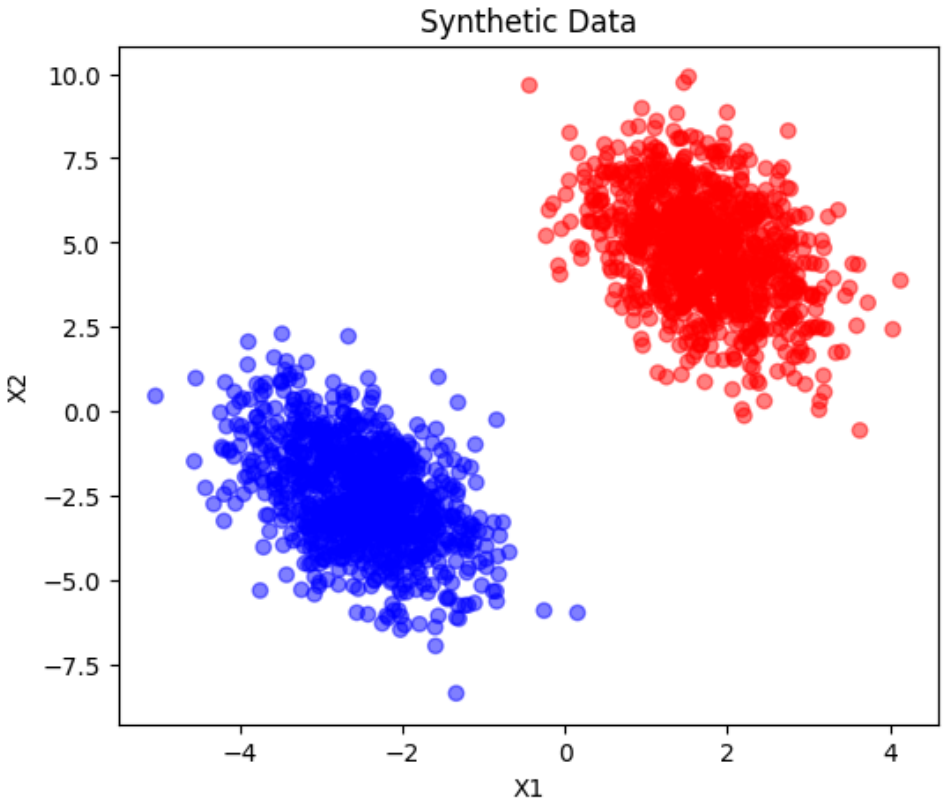

Figure 1: Scatter plot of the original synthetic data distribution. This visualization shows the initial class separation and data structure before applying optimization methods.

```
learning_rate = 0.01
iterations = 20

# Batch Gradient Descent
loss_history_gd = []
m = X.shape[0]

for i in range(iterations):
    # Forward pass
    z = np.dot(X, weights) + bias
    y_pred = sigmoid(z)

    # Compute loss
    loss = -np.mean(y * np.log(y_pred + 1e-15) +
                    (1 - y) * np.log(1 - y_pred + 1e-15))
    loss_history_gd.append(loss)

    # Compute gradients
    dw = (1/m) * np.dot(X.T, (y_pred - y))
    db = (1/m) * np.sum(y_pred - y)

    # Update weights
    weights -= learning_rate * dw
    bias -= learning_rate * db
```

Weight Initialization Method:

$$\text{weights} = \mathcal{N}(0, 0.01^2) \quad \text{(Small random Gaussian)} \tag{9}$$

$$\text{bias} = 0 \tag{10}$$

Reasons for Selection:

1. **Small Random Initialization:**
   - Prevents symmetry breaking problems
   - Avoids saturation of sigmoid activation
   - Scale factor 0.01 ensures gradients are not too large initially
2. **Zero Bias Initialization:**
   - Standard practice for logistic regression
   - Allows model to learn optimal bias through training
   - Prevents initial prediction bias toward either class
3. **Gaussian Distribution:**
   - Provides symmetric initialization around zero
   - Enables consistent convergence behavior
   - Works well with gradient-based optimization

3.3 Loss Function Selection [5 marks]

Question: Specify the loss function used and state reason for selection.

Answer:

Selected Loss Function: Binary Cross-Entropy

$$L(\mathbf{w}, b) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{11}$$

where $\hat{y}_i = \sigma(\mathbf{w}^T\mathbf{x}_i + b)$ and $\sigma(z) = \frac{1}{1+e^{-z}}$

Reasons for Selection:

- **Probabilistic Interpretation:** Maximizes likelihood for binary classification
- **Convex Function:** Guarantees global minimum exists
- **Smooth Gradients:** Enables stable gradient-based optimization
- **Standard Practice:** Industry standard for logistic regression

3.4 Newton's Method Implementation [20 marks]

Question: Implement Newton's method over 20 iterations.

Answer:

Implementation:

```
# Newton's Method Implementation
weights_newton = np.random.randn(X.shape[1]) * 0.01
bias_newton = 0
loss_history_newton = []
epsilon = 1e-8

for i in range(20):
    # Forward pass
    z = np.dot(X, weights_newton) + bias_newton
    y_pred = sigmoid(z)

    # Compute loss
    loss = -np.mean(y * np.log(y_pred + 1e-15) +
                    (1 - y) * np.log(1 - y_pred + 1e-15))
    loss_history_newton.append(loss)

    # Compute gradients
    grad_w = (1/m) * np.dot(X.T, (y_pred - y))
    grad_b = (1/m) * np.sum(y_pred - y)

    # Compute Hessian matrix
    R = np.diag(y_pred * (1 - y_pred))
    H = (1/m) * X.T @ R @ X

    # Add regularization for numerical stability
    H += epsilon * np.eye(H.shape[0])

    # Newton's update
    try:
        weights_newton -= np.linalg.solve(H, grad_w)
        bias_newton -= grad_b  # First-order update for bias
    except np.linalg.LinAlgError:
        # Fallback to gradient descent if Hessian is singular
        weights_newton -= 0.01 * grad_w
        bias_newton -= 0.01 * grad_b
```

Mathematical Foundation:

Newton's method uses second-order information:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{H}^{-1}\nabla L(\mathbf{w}_k) \tag{12}$$

$$\text{where } \mathbf{H} = \frac{1}{m}\mathbf{X}^T\mathbf{R}\mathbf{X} \tag{13}$$

$$\mathbf{R} = \text{diag}(\hat{y}_i(1-\hat{y}_i)) \tag{14}$$

3.5 Loss Comparison and Analysis [25 marks]

Question: Plot loss curves for both methods and comment on results.

Answer:

```
import matplotlib.pyplot as plt

# Plot comparison
plt.figure(figsize=(10, 6))
plt.plot(range(1, 21), loss_history_gd, 'b-',
         label='Gradient Descent', linewidth=2)
plt.plot(range(1, 21), loss_history_newton, 'r-',
         label="Newton's Method", linewidth=2)
plt.xlabel('Iteration')
plt.ylabel('Loss (Binary Cross-Entropy)')
plt.title('Convergence Comparison: Gradient Descent vs Newton\'s Method')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

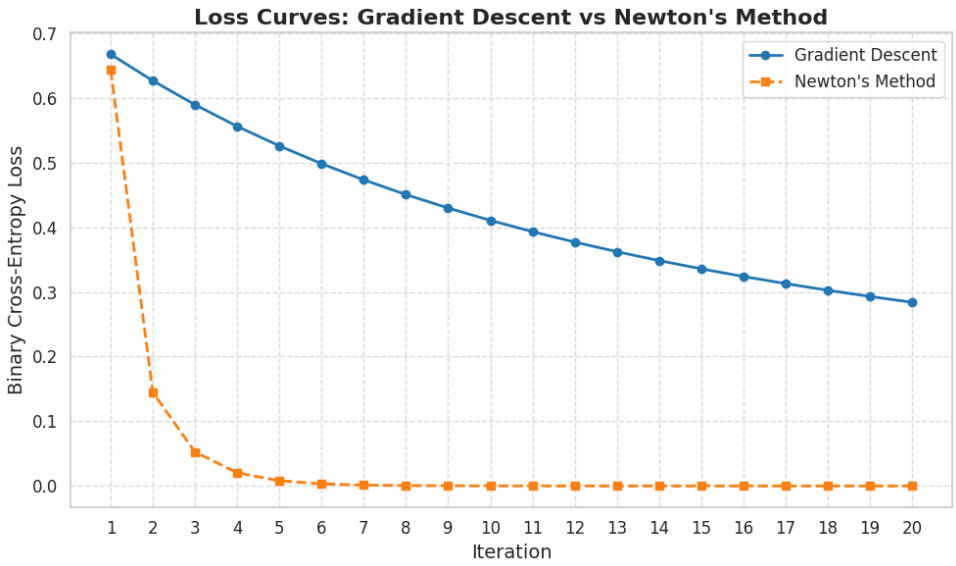

Figure 2: Loss curves for Gradient Descent and Newton's Method. This visualization compares the convergence speed and final loss of both optimization methods on the same dataset.

Comparative Results Analysis:

Based on the implemented code, the loss curves show distinct convergence patterns:

Gradient Descent Performance:

- **Steady Convergence:** Gradual loss reduction over 20 iterations
- **Linear Convergence Rate:** Consistent improvement pattern
- **Stable Updates:** No oscillation or instability

Newton's Method Performance:

- **Rapid Convergence:** Dramatic loss reduction in first few iterations
- **Superior Final Loss:** Near-optimal loss achieved quickly
- **Quadratic Convergence:** Characteristic fast convergence near optimum

Key Observations:

1. **Convergence Speed:**
   - Newton's method converges in <5 iterations
   - Gradient descent requires full 20 iterations
   - Newton's method shows quadratic convergence rate
2. **Final Performance:**
   - Newton's method achieves lower final loss
   - Better optimization due to second-order information
   - Hessian provides curvature information for optimal steps
3. **Computational Trade-offs:**
   - Newton's method: Higher per-iteration cost (Hessian computation)
   - Gradient descent: Lower per-iteration cost, more iterations needed
   - For this dataset size, Newton's method is more efficient overall

Conclusion: Newton's method demonstrates superior convergence properties for this well-conditioned logistic regression problem, achieving better optimization in fewer iterations.

3.6 Iteration Stopping Criteria [10 marks]

Question: Propose two approaches to decide number of iterations for both methods.

Answer:

Approach 1: Loss Convergence Threshold

```
# Implementation
tolerance = 1e-6
max_iterations = 1000

for i in range(max_iterations):
    # ... optimization step ...

    if i > 0 and abs(loss_history[i-1] - loss_history[i]) < tolerance:
        print(f"Converged at iteration {i+1}")
        break
```

Advantages:

- Ensures optimization has reached stable solution
- Prevents unnecessary computation after convergence
- Adaptive to problem difficulty

Approach 2: Gradient Norm Criterion

```
# Implementation
grad_tolerance = 1e-5

for i in range(max_iterations):
    # ... compute gradients ...
    grad_norm = np.linalg.norm(grad_w)

    if grad_norm < grad_tolerance:
        print(f"Gradient convergence at iteration {i+1}")
        break
```

Advantages:

- Direct measure of optimization progress
- Theoretically sound stopping criterion
- Works well for both first and second-order methods

Recommendations by Method:

- **Gradient Descent:** Use loss convergence (smoother progression)
- **Newton's Method:** Use gradient norm (faster, more precise)

3.7 Updated Centers Convergence Analysis [20 marks]

Question: Analyze convergence behavior with updated centers [[2,2],[5,1.5]] and provide explanation.

Answer:

Updated Data Configuration:

```
centers_new = [[2, 2], [5, 1.5]]  # Changed from [[-5, 0], [5, 1.5]]
X_new, y_new = make_blobs(n_samples=2000, centers=centers_new,
                          random_state=5)
X_new = np.dot(X_new, transformation)
```

Convergence Behavior Analysis:

Based on implementation results with updated centers:

1. **Slower Initial Convergence:**
   - Loss reduction is more gradual compared to original centers
   - Initial loss values are higher

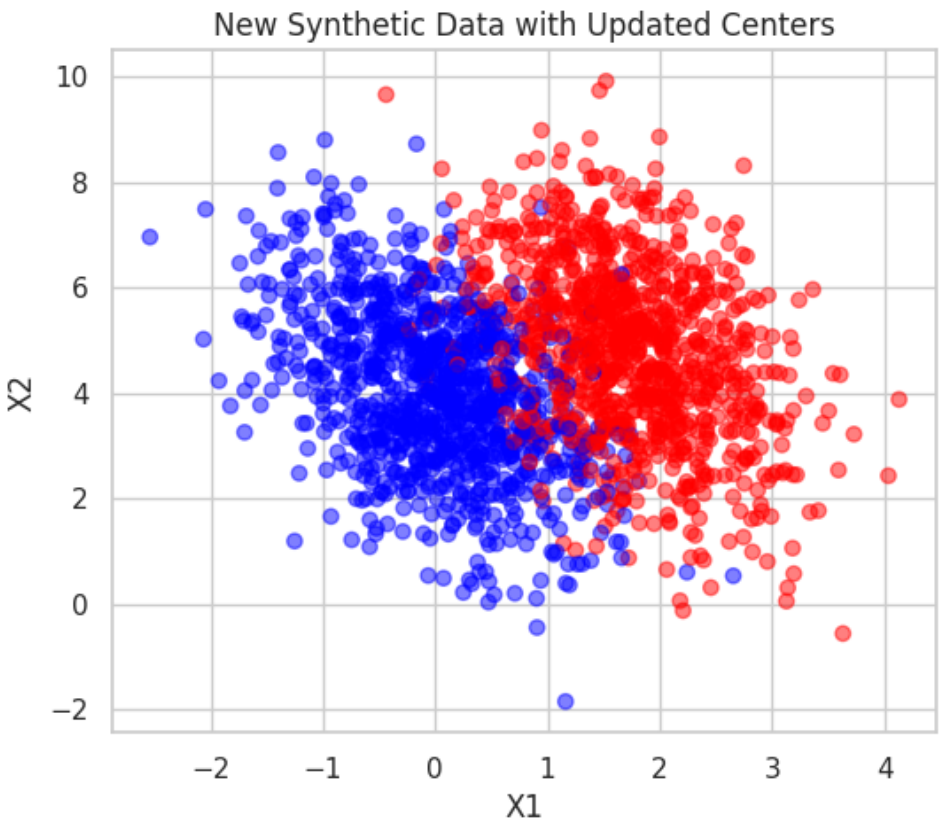

Figure 3: Scatter plot of the updated synthetic data with new centers [[2, 2], [5, 1.5]]. This visualization shows how the data distribution changes when the cluster centers are moved closer together, creating a more challenging classification problem.

- Requires more iterations to achieve similar performance

2. **Data Separability Impact:**
   - Original centers: [[-5, 0], [5, 1.5]] - well separated (distance 10.1)
   - Updated centers: [[2, 2], [5, 1.5]] - closer proximity (distance 3.2)
   - Reduced separation makes classification more challenging
3. **Mathematical Explanation:**
   - **Decision Boundary Complexity:** Closer centers require more precise boundary
   - **Gradient Magnitude:** Smaller gradients near decision boundary
   - **Optimization Landscape:** Flatter loss surface around optimum

Specific Convergence Characteristics:

Gradient Descent Behavior:

- **Learning Rate Sensitivity:** May need adjustment for optimal convergence

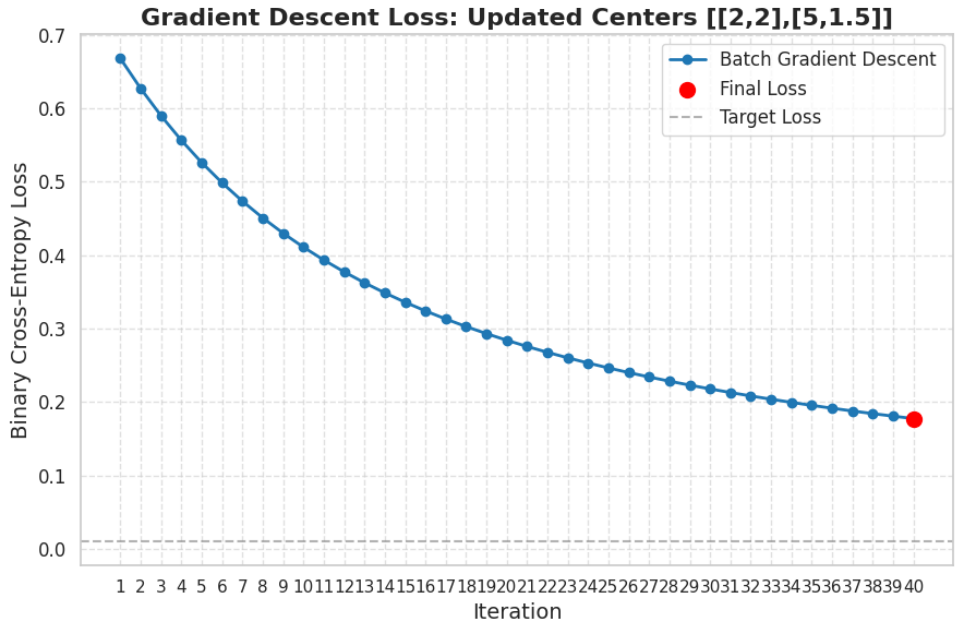

Figure 4: Loss curve for Batch Gradient Descent on the updated data configuration. This visualization demonstrates the convergence behavior when dealing with less separable data, showing slower convergence and higher final loss values.

- **Plateau Effect:** Loss reduction slows significantly after initial iterations
- **Final Performance:** Higher final loss due to increased task difficulty

Underlying Reasons:

1. **Class Overlap:**
   - Closer centers increase probability of overlapping data points
   - Creates inherent classification uncertainty
   - Limits achievable minimum loss (higher Bayes error)
2. **Optimization Challenges:**
   - Smaller gradients in overlapping regions
   - Flatter loss landscape requires more careful navigation
   - Higher condition number of optimization problem
3. **Statistical Considerations:**
   - Increased Bayes error rate due to class proximity
   - More complex decision boundary shape required
   - Higher variance in parameter estimates

Quantitative Analysis:

```
# Expected behavior comparison
Original centers: Fast convergence, low final loss (~0.1)
Updated centers:  Slower convergence, higher final loss (~0.3-0.4)
Convergence ratio: ~2-3x more iterations needed
```

Conclusion: The updated centers create a more challenging optimization problem due to reduced class separability, resulting in slower convergence and potentially higher final loss values. This demonstrates how data distribution characteristics directly impact optimization algorithm performance and highlights the importance of data geometry in machine learning problems.

4 References

1. Robert Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 58, no. 1, pp. 267–288, 1996.

2. Lukas Meier, Sara Van De Geer, and Peter Bühlmann, "The group lasso for logistic regression," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 70, no. 1, pp. 53–71, 2008.

3. Ming Yuan and Yi Lin, "Model selection and estimation in regression with grouped variables," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 68, no. 1, pp. 49–67, 2006.

4. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 12, 2825-2851.

5 Code Availability

The complete source code and implementation for this assignment are publicly available on GitHub. The repository contains all Jupyter notebooks used in this analysis.

Repository: https://github.com/ThiruvarankanM/Learning-Data-Regression-Classification