UNIVERSITY OF MORATUWA

Faculty of Engineering
Department of Electronic and Telecommunication Engineering

EN3150

Machine Learning for Communication Systems

Assignment 02

Learning from Data and Related Challenges and Classification

Submitted by:

Rivindu Vinsara Kumarage
Index No: 220343B

September 7, 2025

Contents

1 Linear Regression

Question 1 - OLS Line Alignment Issue

Analysis of OLS Misalignment

The Ordinary Least Squares (OLS) fitted line shown in Figure 1 is not aligned with the majority of data points due to the presence of **outliers** in the dataset.
Key Points:

- OLS minimizes the sum of squared residuals: $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- The squared term gives disproportionate weight to large deviations (outliers)
- Outliers have a leverage effect, pulling the regression line towards them
- This results in a line that may not represent the true underlying relationship for the majority of the data

Mathematical Explanation: Since OLS uses squared errors, points far from the true line contribute quadratically to the loss function. For example, if a point has an error of 10 units, it contributes 100 to the loss, while a point with error 1 contributes only 1. This makes the algorithm prioritize reducing large errors over small ones, causing the fitted line to be biased towards outliers.

Question 2 - Modified Loss Function Analysis

Scheme Comparison

Scheme 1 will provide a better fitted line for inliers compared to the original OLS.
Analysis of Both Schemes:
Scheme 1: Outlier weight = 0.01, Inlier weight = 1

- Effectively reduces the influence of outliers by 99%
- Modified loss: $\frac{1}{N}\sum_{i=1}^{N} a_i(y_i - \hat{y}_i)^2$
- For outliers: $a_i = 0.01 \rightarrow$ minimal contribution to loss
- For inliers: $a_i = 1 \rightarrow$ normal contribution
- Result: Line fits primarily to inlier data

Scheme 2: Outlier weight = 5, Inlier weight = 1

- **Increases** the influence of outliers by $5\times$
- Makes the outlier problem worse than original OLS

- Line will be even more biased towards outliers
- Performs worse than standard OLS

Mathematical Justification: In Scheme 1, the optimization becomes:

$$\min_{w} \frac{1}{N}\left[\sum_{i\in\text{inliers}}(y_i-\hat{y}_i)^2 + 0.01\sum_{i\in\text{outliers}}(y_i-\hat{y}_i)^2\right]$$

This effectively ignores outliers and fits primarily to inlier data, resulting in better alignment with the majority of points.

Question 3 - Linear Regression Limitations in Brain Analysis

Why Linear Regression is Unsuitable

Linear regression is not suitable for identifying predictive brain regions due to several critical limitations:

1. High Dimensionality Problem

- Brain images contain thousands of voxels (features)
- Number of features >> number of samples (curse of dimensionality)
- Standard linear regression cannot handle $p > n$ scenarios
- Results in overfitting and poor generalization

2. Multicollinearity Issues

- Adjacent voxels are highly correlated
- Creates unstable coefficient estimates
- Makes interpretation of individual voxel importance unreliable

3. Feature Selection Problem

- Cannot perform automatic feature selection
- All voxels receive non-zero weights
- Cannot identify specific brain regions as predictive
- Lacks sparsity in the solution

4. Interpretability Challenges

- Cannot group voxels into meaningful brain regions
- Individual voxel weights may not reflect regional importance
- Lacks biological/anatomical interpretability

Statistical Consequences: The design matrix $X \in \mathbb{R}^{N\times p}$ with $p >> N$ makes X^TX singular, preventing computation of $(X^TX)^{-1}$ needed for the closed-form solution.

Question 4 & 5 - LASSO vs Group LASSO Comparison

Method Selection and Justification

Group LASSO (Method B) is more appropriate for brain region identification.
Detailed Comparison:
Standard LASSO (Method A):

- Objective: $\min_w \left\{ \frac{1}{N} \sum_{i=1}^{N} (y_i - w^T x_i)^2 + \lambda \|w\|_1 \right\}$
- Performs individual voxel selection
- May select scattered voxels across brain regions
- Lacks spatial/anatomical coherence
- Cannot identify entire brain regions as predictive units

Group LASSO (Method B):

- Objective: $\min_w \left\{ \frac{1}{N} \sum_{i=1}^{N} (y_i - w^T x_i)^2 + \lambda \sum_{g=1}^{G} \|w_g\|_2 \right\}$
- Performs region-level selection (groups of voxels)
- Either selects entire brain regions or excludes them completely
- Maintains spatial/anatomical coherence
- Directly addresses the research question

Why Group LASSO is Superior:
1. Biological Relevance

- Brain functions are localized to specific regions
- Entire regions, not individual voxels, are functionally meaningful
- Group LASSO respects anatomical structure

2. Interpretability

- Directly identifies which brain regions are predictive
- Results are clinically/scientifically interpretable
- Facilitates hypothesis generation for neuroscience research

3. Statistical Advantages

- Reduces effective dimensionality from voxels to regions
- More stable feature selection
- Better handling of within-region correlation
- Improved prediction performance on new subjects

Mathematical Insight: The ℓ_2 norm penalty $\|w_g\|_2$ in Group LASSO creates an "all-or-nothing" selection mechanism for each group g. When λ is sufficiently large, entire groups are set to zero, effectively performing region-level feature selection that aligns with the research objective.

2 Logistic Regression

Question 1 - Data Loading

Data Loading

The provided code loads the penguins dataset and preprocesses it for binary classification between 'Adelie' and 'Chinstrap' species.

Code Analysis:

```python
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

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

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

# Encode labels
le = LabelEncoder()
y_encoded = le.fit_transform(df_filtered['species'])
df_filtered['class_encoded'] = y_encoded
```

Listing 1: Data Loading and Preprocessing

Question 2 - Training Errors with SAGA Solver

SAGA Solver Issues

Expected Errors:

1. Convergence Warning

- `ConvergenceWarning: lbfgs failed to converge`
- SAGA solver struggles with non-numerical features
- Default max_iter may be insufficient

2. Data Type Errors

- Categorical features ('island', 'sex') cause issues
- SAGA expects numerical input only
- String columns cannot be processed directly

Resolution Strategies:
Immediate Fixes:

```
# Remove categorical columns
import numpy as np
X_numeric = df_filtered.select_dtypes(include=[np.number])
X_numeric = X_numeric.drop(['class_encoded'], axis=1)

# Or encode categorical variables
from sklearn.preprocessing import LabelEncoder
le_island = LabelEncoder()
le_sex = LabelEncoder()
df_filtered['island_encoded'] = le_island.fit_transform(
    df_filtered['island'])
df_filtered['sex_encoded'] = le_sex.fit_transform(df_filtered['
    sex'])

# Increase iterations
logreg = LogisticRegression(solver='saga', max_iter=1000)
```

Listing 2: Error Resolution

Comprehensive Preprocessing:

```
# Select only numerical features
numerical_features = ['bill_length_mm', 'bill_depth_mm',
                      'flipper_length_mm', 'body_mass_g']
X = df_filtered[numerical_features]
```

Listing 3: Complete Preprocessing Pipeline

Question 3 - SAGA Solver Performance Issues

Why SAGA Performs Poorly

Technical Reasons for Poor Performance:
1. Algorithm Characteristics

- SAGA is a stochastic gradient method
- Designed for very large datasets (n > 10,000)

- Inefficient for small datasets like penguins (∼300 samples)
- High variance in gradient estimates for small samples

2. Feature Scaling Sensitivity

- SAGA is sensitive to feature scales
- Penguin features have different scales (mm vs grams)
- Unscaled features cause slow/poor convergence
- Gradient updates become imbalanced

3. Convergence Properties

- Requires many iterations to converge
- Default max_iter=100 often insufficient
- Stochastic nature leads to oscillating behavior
- Poor conditioning of the optimization landscape

Mathematical Explanation: SAGA updates follow: $w^{(k+1)} = w^{(k)} - \gamma(\nabla f_i(w^{(k)}) - \alpha_i^{(k)} + \bar{\alpha}^{(k)})$
For small datasets, the stochastic approximation adds unnecessary noise without computational benefits.

Question 4 - LibLinear Solver Performance

LibLinear Accuracy

Expected Classification Accuracy:
With the liblinear solver using only numerical features:

```python
# Using numerical features only
X = df_filtered[['bill_length_mm', 'bill_depth_mm',
                 'flipper_length_mm', 'body_mass_g']]
logreg = LogisticRegression(solver='liblinear')
logreg.fit(X_train, y_train)
accuracy = accuracy_score(y_test, y_pred)
```

Listing 4: LibLinear Implementation

Expected Accuracy: ∼85-95%
The high accuracy is expected because:

- Adelie and Chinstrap penguins have distinct physical characteristics

- Clear separation in feature space
- Logistic regression is well-suited for this binary classification

Question 5 - LibLinear vs SAGA Comparison

Why LibLinear Outperforms SAGA

Algorithmic Advantages of LibLinear:

1. Optimization Method

- Uses coordinate descent algorithm
- Deterministic updates (no stochastic noise)
- Faster convergence for small-medium datasets
- More stable gradient estimates

2. Dataset Size Optimization

- Specifically designed for smaller datasets
- Efficient memory usage
- No overhead from stochastic sampling
- Better suited for n $<$ 10,000 samples

3. Numerical Stability

- More robust to feature scaling issues
- Better conditioned optimization problem
- Consistent convergence behavior
- Less sensitive to hyperparameter choices

4. Implementation Efficiency

- Optimized C++ implementation
- Better cache locality
- Lower computational overhead per iteration
- Faster wall-clock time to convergence

Performance Summary:

Aspect	LibLinear	SAGA
Small datasets	Excellent	Poor
Convergence speed	Fast	Slow
Stability	High	Variable
Scaling sensitivity	Low	High

Question 6 - Random State Variance

Random State Impact on SAGA Accuracy

Sources of Variability:

1. Train-Test Split Randomness

- Different random_state values create different train/test splits
- Some splits may be more/less representative
- Creates baseline variability in performance measurement

2. SAGA Algorithm Stochasticity

- SAGA uses random sampling of gradients
- Different initialization leads to different optimization paths
- Convergence to different local optima possible
- High variance in final weight estimates

3. Convergence Issues

- May not converge within max_iter limit
- Stopping at different iteration counts
- Inconsistent solution quality
- Premature termination effects

Mathematical Explanation: The SAGA update rule introduces stochasticity:

$$w^{(k+1)} = w^{(k)} - \gamma \cdot [\nabla f_{i_k}(w^{(k)}) - \alpha_{i_k}^{(k)} + \bar{\alpha}^{(k)}]$$

Where i_k is randomly sampled, creating path-dependent convergence behavior.

Mitigation Strategies:

- Use cross-validation for robust evaluation

- Increase max_iter for better convergence
- Apply feature scaling
- Consider ensemble methods

Question 7 - Feature Scaling Impact

Scaling Comparison Analysis

Implementation and Results:

```python
from sklearn.preprocessing import StandardScaler

# Without scaling
logreg_saga = LogisticRegression(solver='saga', max_iter=1000)
logreg_liblinear = LogisticRegression(solver='liblinear')

# With scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

logreg_saga_scaled = LogisticRegression(solver='saga', max_iter
    =1000)
logreg_liblinear_scaled = LogisticRegression(solver='liblinear')
```

Listing 5: Feature Scaling Comparison

Expected Results:

Solver	Without Scaling	With Scaling	Improvement
SAGA	60-75%	85-95%	Significant
LibLinear	85-90%	90-95%	Moderate

Reasons for Improvement:

1. SAGA Scaling Sensitivity

- Features have vastly different scales (mm vs grams)
- Gradient components become imbalanced
- Large-scale features dominate updates
- Scaling equalizes gradient contributions

2. Optimization Landscape

- Unscaled: elongated, ill-conditioned ellipsoids
- Scaled: more circular, well-conditioned contours
- Faster convergence along all dimensions
- More stable gradient descent

3. Differential Impact

- SAGA shows dramatic improvement (most sensitive)
- LibLinear shows modest improvement (more robust)
- Confirms algorithmic characteristics

Question 8 - Categorical Feature Encoding

Proper Categorical Feature Handling

The Approach is INCORRECT.
Problems with Label Encoding + Scaling:
1. Artificial Ordinality

- Label encoding: red=0, blue=1, green=2
- Implies: red < blue < green (false ordering)
- Creates meaningless distance relationships
- Scaling preserves these artificial relationships

2. Statistical Issues

- Standard scaling assumes continuous, normally distributed data
- Categorical labels are discrete and nominal
- Mean and standard deviation are meaningless
- Scaled values misrepresent categories

3. Model Interpretation Problems

- Logistic regression assumes linear relationships
- Coefficients become uninterpretable
- May learn spurious patterns from artificial ordering

Proposed Solutions:

Method 1: One-Hot Encoding (Recommended)

```python
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# One-hot encoding for categorical features
encoder = OneHotEncoder(drop='first', sparse=False)
categorical_encoded = encoder.fit_transform(df[['color']].values
    .reshape(-1, 1))

# Create feature names
feature_names = ['color_green', 'color_red']  # blue is dropped
    (reference)

# Scale only numerical features
scaler = StandardScaler()
numerical_scaled = scaler.fit_transform(numerical_features)

# Combine features
X_final = np.hstack([numerical_scaled, categorical_encoded])
```

Listing 6: One-Hot Encoding Approach

Method 2: Mixed Preprocessing Pipeline

```python
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_columns),
        ('cat', OneHotEncoder(drop='first'), categorical_columns
           )
    ]
)

X_processed = preprocessor.fit_transform(X)
```

Listing 7: Column Transformer Approach

Why This Approach is Superior:

- Preserves categorical nature of features
- No artificial ordinality introduced
- Proper statistical treatment for each feature type
- Maintains interpretability
- Standard scaling applied only where appropriate

3 Logistic Regression: First/Second-Order Methods

Question 1 - Data Generation

Data Generation Implementation

The provided code generates synthetic binary classification data using scikit-learn's make_blobs function with specific transformations.

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

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)
```

Listing 8: Data Generation Analysis

This creates a linearly separable binary classification problem with transformed feature space.

Question 2 - Batch Gradient Descent Implementation

Gradient Descent Implementation

Weight Initialization Method: Xavier/Glorot Initialization is recommended for logistic regression.
Reasoning for Xavier Initialization:

- Maintains gradient magnitudes across layers
- Prevents vanishing/exploding gradient problems
- Accounts for number of input features
- Formula: $w \sim \mathcal{N}(0, \frac{1}{\text{n\_features}})$

Implementation:

```python
import numpy as np

def sigmoid(z):
    """Sigmoid␣activation␣function␣with␣numerical␣stability"""
    z = np.clip(z, -250, 250)  # Prevent overflow
    return 1 / (1 + np.exp(-z))

def batch_gradient_descent(X, y, n_iterations=20, learning_rate
   =0.01):
    """
␣␣␣␣Batch␣Gradient␣Descent␣for␣Logistic␣Regression
␣␣␣␣"""
    # Add bias term
    X_bias = np.c_[np.ones((X.shape[0], 1)), X]
    n_samples, n_features = X_bias.shape

    # Xavier initialization
    np.random.seed(42)
    weights = np.random.normal(0, 1/np.sqrt(n_features),
       n_features)

    # Store loss history
    loss_history = []

    for i in range(n_iterations):
        # Forward pass
        z = X_bias.dot(weights)
        predictions = sigmoid(z)

        # Compute loss (cross-entropy)
        epsilon = 1e-15  # Prevent log(0)
        predictions = np.clip(predictions, epsilon, 1 - epsilon)
        loss = -np.mean(y * np.log(predictions) +
                       (1 - y) * np.log(1 - predictions))
        loss_history.append(loss)

        # Compute gradients
        dw = (1/n_samples) * X_bias.T.dot(predictions - y)

        # Update weights
        weights -= learning_rate * dw

        if i % 5 == 0:
            print(f"Iteration␣{i}:␣Loss␣=␣{loss:.6f}")

    return weights, loss_history

# Execute gradient descent
weights_gd, loss_gd = batch_gradient_descent(X, y, n_iterations
   =20)
```

Listing 9: Batch Gradient Descent Implementation

Alternative Initialization Methods:

- **Zero Initialization:** Simple but can cause slow convergence
- **Random Normal:** $w \sim \mathcal{N}(0, 0.01)$ - simple but less principled
- **He Initialization:** Better for ReLU, but Xavier is optimal for sigmoid

Question 3 - Loss Function Selection

Loss Function Choice

Selected Loss Function: Cross-Entropy (Log-Likelihood)
Mathematical Form: $\mathcal{L}(w) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]$
Where $p_i = \sigma(w^T x_i) = \frac{1}{1+e^{-w^T x_i}}$
Reasons for Selection:
1. Statistical Foundation

- Derived from maximum likelihood estimation
- Natural choice for probabilistic binary classification
- Provides proper probability estimates

2. Mathematical Properties

- Convex function (guaranteed global minimum)
- Smooth and differentiable everywhere and Well-behaved gradients for optimization

3. Gradient Characteristics

- Large gradients when predictions are wrong
- Small gradients when predictions are confident and correct
- Self-regulating learning behavior

4. Probabilistic Interpretation

- Outputs meaningful probabilities
- Enables uncertainty quantification
- Compatible with Bayesian inference

Question 4 - Newton's Method Implementation

Newton's Method Implementation

Implementation:

```python
def newtons_method(X, y, n_iterations=20):
    """
    Newton's Method for Logistic Regression
    """
    # Add bias term
    X_bias = np.c_[np.ones((X.shape[0], 1)), X]
    n_samples, n_features = X_bias.shape

    # Xavier initialization
    np.random.seed(42)
    weights = np.random.normal(0, 1/np.sqrt(n_features),
        n_features)

    # Store loss history
    loss_history = []

    for i in range(n_iterations):
        # Forward pass
        z = X_bias.dot(weights)
        predictions = sigmoid(z)

        # Compute loss
        epsilon = 1e-15
        predictions_clipped = np.clip(predictions, epsilon, 1 -
            epsilon)
        loss = -np.mean(y * np.log(predictions_clipped) +
                        (1 - y) * np.log(1 - predictions_clipped)
                           )
        loss_history.append(loss)

        # Compute first derivative (gradient)
        gradient = (1/n_samples) * X_bias.T.dot(predictions - y)

        # Compute second derivative (Hessian)
        # H = (1/n) * X.T * W * X, where W is diagonal weight
            matrix
        W = np.diag(predictions * (1 - predictions))
        hessian = (1/n_samples) * X_bias.T.dot(W).dot(X_bias)

        # Add regularization to prevent singular matrix
        hessian += 1e-8 * np.eye(n_features)

        try:
            # Newton update: w = w - H^(-1) * gradient
            delta_w = np.linalg.solve(hessian, gradient)
            weights -= delta_w
        except np.linalg.LinAlgError:
            print(f"Singular␣matrix␣at␣iteration␣{i},␣stopping")
            break

        if i % 5 == 0:
            print(f"Iteration␣{i}:␣Loss␣=␣{loss:.6f}")

    return weights, loss_history

# Execute Newton's method
weights_newton, loss_newton = newtons_method(X, y, n_iterations
    =20)
```

Listing 10: Newton's Method for Logistic Regression

Key Implementation Details:

1. Hessian Computation

- $H = \frac{1}{n}X^TWX$ where $W_{ii} = p_i(1-p_i)$
- Represents curvature of loss function
- Always positive semi-definite for logistic regression

2. Numerical Stability

- Regularization term added to Hessian
- Using `np.linalg.solve()` instead of explicit inverse
- Clipping predictions to prevent numerical issues

3. Update Rule

- $w^{(k+1)} = w^{(k)} - H^{-1}\nabla\mathcal{L}(w^{(k)})$
- Uses second-order information for faster convergence
- Naturally adaptive step size

Question 5 - Convergence Comparison

Loss Comparison and Analysis

Plotting Implementation:

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
plt.plot(range(len(loss_gd)), loss_gd, 'b-o', linewidth=2,
         markersize=6, label='Batch Gradient Descent', alpha
            =0.8)
plt.plot(range(len(loss_newton)), loss_newton, 'r-s', linewidth
    =2,
         markersize=6, label="Newton's Method", alpha=0.8)

plt.xlabel('Iteration', fontsize=14)
plt.ylabel('Cross-Entropy Loss', fontsize=14)
plt.title('Convergence Comparison: Gradient Descent vs Newton\'s
    Method',
          fontsize=16, fontweight='bold')
plt.legend(fontsize=12)
plt.grid(True, alpha=0.3)
plt.yscale('log')  # Log scale to better show convergence
plt.tight_layout()
plt.show()

# Print final losses
print(f"Final GD Loss: {loss_gd[-1]:.8f}")
print(f"Final Newton Loss: {loss_newton[-1]:.8f}")
print(f"GD Iterations for convergence: {len(loss_gd)}")
print(f"Newton Iterations for convergence: {len(loss_newton)}")
```

Listing 11: Loss Comparison Plot

Expected Results and Analysis:

1. Convergence Speed

- **Newton's Method:** Quadratic convergence (very fast)
- **Gradient Descent:** Linear convergence (slower)
- Newton typically converges in 3-8 iterations
- GD may require 50+ iterations for same precision

2. Loss Reduction Pattern

- **Newton:** Rapid exponential decrease initially
- **GD:** Steady linear decrease on log scale

- Newton shows steeper descent in early iterations
- Both eventually reach similar minimum

3. Mathematical Explanation

- Newton uses curvature information (Hessian)
- Better approximation of optimal step size
- GD uses fixed or simple adaptive learning rates
- Newton naturally handles ill-conditioning better

4. Computational Trade-offs

- **Newton:** $O(p^3)$ per iteration (Hessian inversion)
- **GD:** $O(p^2)$ per iteration (gradient computation)
- Newton faster for small-medium dimensions
- GD preferred for very high dimensions

Theoretical Convergence Rates:

- **Newton:** $\|w^{(k+1)} - w^*\| \leq C\|w^{(k)} - w^*\|^2$ (quadratic)
- **GD:** $\|w^{(k+1)} - w^*\| \leq \rho\|w^{(k)} - w^*\|$ where $\rho < 1$ (linear)

Question 6 - Stopping Criteria

Iteration Decision Approaches

Proposed Approaches for Stopping Criteria:
Approach 1: Convergence-Based Stopping

```python
def convergence_based_stopping(loss_history, tolerance=1e-6):
    """
    Stop when loss change is below threshold
    """
    if len(loss_history) < 2:
        return False

    loss_change = abs(loss_history[-2] - loss_history[-1])
    relative_change = loss_change / abs(loss_history[-2])

    return relative_change < tolerance
```

```python

# Usage in training loop
for i in range(max_iterations):
    # ... training step ...
    loss_history.append(current_loss)

    if convergence_based_stopping(loss_history):
        print(f"Converged at iteration {i}")
        break
```

Listing 12: Convergence-Based Stopping

Advantages:

- Automatically adapts to convergence rate
- Prevents unnecessary computation
- Works for both GD and Newton's method
- Objective and reproducible

Approach 2: Validation-Based Early Stopping

```python
def validation_based_stopping(X_train, y_train, X_val, y_val,
                              patience=5, min_delta=1e-4):
    """
    Stop when validation loss stops improving
    """
    best_val_loss = float('inf')
    patience_counter = 0

    for i in range(max_iterations):
        # Train for one iteration
        weights = train_one_iteration(X_train, y_train, weights)

        # Evaluate on validation set
        val_loss = compute_loss(X_val, y_val, weights)

        if val_loss < best_val_loss - min_delta:
            best_val_loss = val_loss
            patience_counter = 0
        else:
            patience_counter += 1

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

    return weights
```

Listing 13: Validation-Based Early Stopping

Advantages:

- Prevents overfitting
- Better generalization performance
- Robust to noise in training data
- Standard practice in machine learning

Comparison of Approaches:

Criteria	Convergence-Based	Validation-Based
Prevents Overfitting	No	Yes
Computational Efficiency	High	Medium
Generalization	Unknown	Better
Implementation Complexity	Simple	Moderate
Data Requirements	Training only	Train + Validation

Recommendations:

- Use convergence-based for well-conditioned problems
- Use validation-based for real-world applications
- Combine both: convergence as backup, validation as primary
- Consider gradient norm: $\|\nabla\mathcal{L}\| < \epsilon$

Question 7 - Modified Centers Analysis

Convergence Analysis with New Centers

Modified Data Generation:

```
# New centers configuration
centers_new = [[2, 2], [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)

# Apply gradient descent
weights_new, loss_new = batch_gradient_descent(X_new, y_new,
    n_iterations=20)
```

Listing 14: Modified Centers Configuration

Convergence Behavior Analysis:
1. Geometric Changes

- **Original centers:** [[-5, 0], [5, 1.5]] - well separated
- **New centers:** [[2, 2], [5, 1.5]] - closer together
- Reduced inter-class distance
- Increased class overlap after transformation

2. Impact on Separability

- Classes become less linearly separable
- Decision boundary becomes less obvious
- Higher inherent classification difficulty
- More complex optimization landscape

3. Expected Convergence Behavior
Slower Convergence:

- Gradients become smaller near decision boundary
- More iterations required for same precision
- Loss plateaus at higher minimum value
- Less steep descent in loss function

4. Mathematical Explanation
Gradient Magnitude Analysis: For logistic regression, gradient magnitude depends on prediction confidence: $\|\nabla\mathcal{L}\| = \frac{1}{n}\|X^T(p-y)\|$
When classes overlap:

- Predictions p closer to 0.5 (uncertain)
- Smaller values of $|p-y|$ for misclassified points
- Reduced gradient magnitudes
- Slower parameter updates

5. Visualization and Analysis

```python
# Compare convergence rates
plt.figure(figsize=(15, 5))

# Plot 1: Data visualization
plt.subplot(1, 3, 1)
plt.scatter(X[:, 0], X[:, 1], c=y, alpha=0.7, cmap='viridis')
plt.title('Original Data (Well Separated)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

plt.subplot(1, 3, 2)
plt.scatter(X_new[:, 0], X_new[:, 1], c=y_new, alpha=0.7, cmap='
    viridis')
plt.title('Modified Data (Closer Centers)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Plot 3: Loss comparison
plt.subplot(1, 3, 3)
plt.plot(loss_gd, 'b-', label='Original Centers', linewidth=2)
plt.plot(loss_new, 'r--', label='Modified Centers', linewidth=2)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Convergence Comparison')
plt.legend()
plt.yscale('log')

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

Listing 15: Convergence Comparison Analysis

Expected Results:

Metric	**Original**	**Modified**
Final Loss	~ 0.01	$\sim 0.3 - 0.5$
Convergence Rate	Fast	Slow
Iterations to Converge	~ 10	$\sim 20+$
Classification Accuracy	$> 95\%$	$75 - 85\%$

6. Theoretical Implications
Condition Number Impact:

- Closer centers increase condition number of Hessian
- Worse-conditioned optimization problem

- Requires smaller learning rates for stability
- Benefits more from second-order methods (Newton)

Practical Recommendations:

- Use adaptive learning rates (AdaGrad, Adam)
- Consider feature engineering for better separation
- Apply regularization to prevent overfitting
- Use early stopping based on validation performance

4 Conclusion

This assignment demonstrates the fundamental principles of machine learning optimization and classification techniques. Through theoretical analysis and practical implementation, we explored:

Key Learning Outcomes:

- Understanding of outlier impacts on regression methods
- Comparison of regularization techniques (LASSO vs Group LASSO)
- Practical solver selection for logistic regression
- Implementation of first and second-order optimization methods
- Analysis of convergence behavior under different data conditions

The results highlight the importance of proper preprocessing, algorithm selection, and understanding the underlying mathematical principles for successful machine learning applications.

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.