Department of Electronic & Telecommunication Engineering, University of Moratuwa, Sri Lanka.

Assignment 02: Learning from data and related challenges and classification

By:

220097X De Alwis W.M.R.

EN3150 Pattern Recognition
Submitted in partial fulfillment of the requirements for the module

Date
09/09/2025

1 Linear Regression

1.1 Reason behind OLS fitted line not aligning with the majority of data points.

OLS is minimizing the sum of squared residuals:

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

Because errors are squared, points with large residuals (outliers) have a disproportionately high influence on the loss.

In the plot, most data points lie along a nearly horizontal line. But the outliers have very high or very low y- values. These outliers pull the regression line towards themselves, as OLS tries to minimize the loss function. As a results of this, the OLS fitted line ends up being tilted upwards and misaligned with the majority of data, because it is trying to compromise between fitting the bulk of the points and reducing the large errors from the outliers.

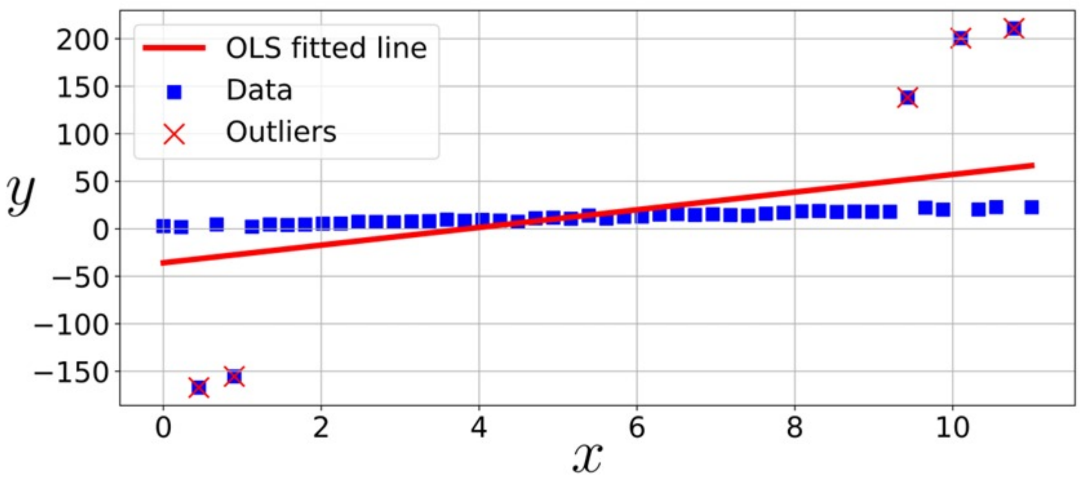


Figure 1: Ordinary least squares fit on data.

1.2 Reducing the impact of the outliers

To reduce the impact of outliers, a modified loss function is introduced. It is given as

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

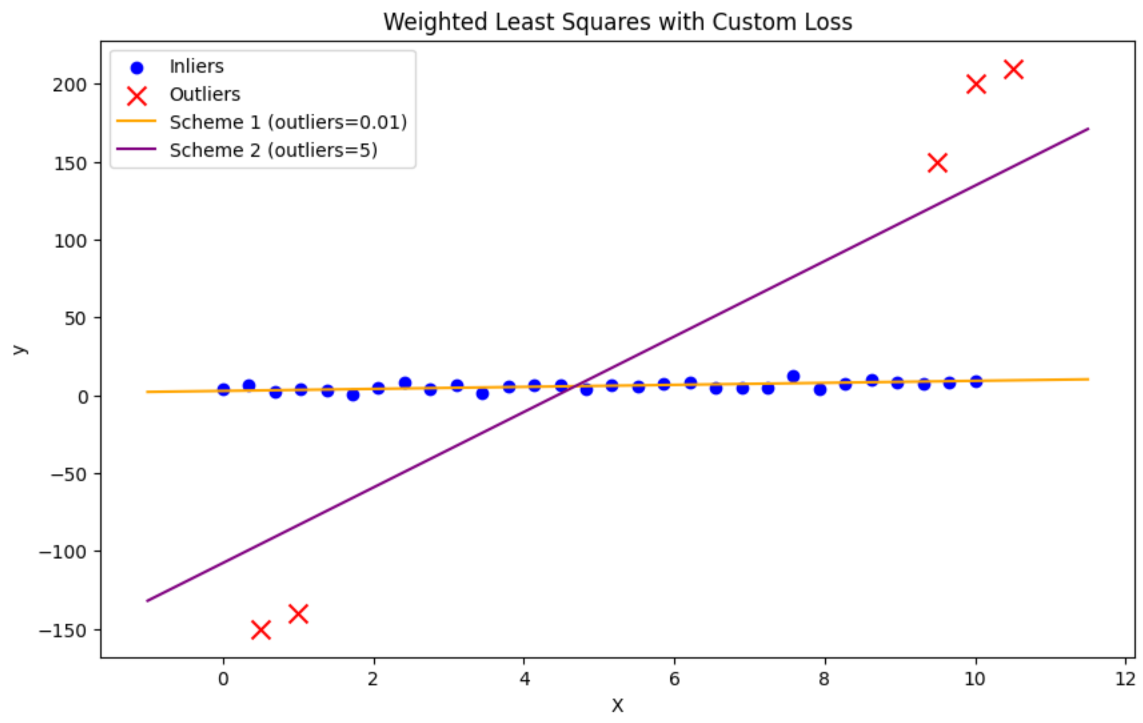
There are two schemes proposed for setting a_i:

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

The following code was used to calculate and plot the loss of the two schemes according to the modified loss function.

```python
def loss(alpha, y_i, y_ihat):
    N = len(y_i)
    return (1/N) * np.sum(((y_i - y_ihat)**2) * alpha)

def weighted_least_squares(X, y, alpha):
    W = np.diag(alpha)
    beta = np.linalg.inv(X.T @ W @ X) @ (X.T @ W @ y)
    return beta  # [intercept, slope]

alpha_s1 = np.ones(len(y))
alpha_s1[-len(x_outliers):] = 0.01   # outliers get very low weight
beta_s1 = weighted_least_squares(X, y, alpha_s1)

alpha_s2 = np.ones(len(y))
alpha_s2[-len(x_outliers):] = 5   # outliers get high weight
beta_s2 = weighted_least_squares(X, y, alpha_s2)

xx = np.linspace(min(x)-1, max(x)+1, 200)
Xx = np.c_[np.ones(len(xx)), xx]

yhat_ols = Xx @ beta_ols
yhat_s1 = Xx @ beta_s1
yhat_s2 = Xx @ beta_s2

plt.figure(figsize=(10,6))
plt.scatter(x_data, y_data, color="blue", label="Inliers")
plt.scatter(x_outliers, y_outliers, color="red", marker="x", s=100, label="
    Outliers")


plt.plot(xx, yhat_s1, "orange", label="Scheme 1 (outliers=0.01)")
plt.plot(xx, yhat_s2, "purple", label="Scheme 2 (outliers=5)")

plt.xlabel("X")
plt.ylabel("y")
plt.title("Weighted Least Squares with Custom Loss")
plt.legend()
plt.show()
```

Below plot shows the comparison of Weighted Least Squares under two schemes, showing how different weighting of outliers affects the fitted regression line relative to inliers and outliers.



Selected scheme and justification

Scheme 1 will give a better fitted line than the inliers than OLS, because it reduces the influence of outliers. This is achieved by assigning very small weights to outliers minimizing their effect on the regression line, allowing the fit to be driven primarily by the inliers.

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

Linear regression is unsuitable for identifying the most predictive brain regions in fMRI analysis due to its inability to handle high-dimensional data, where the number of voxels far exceeds the number of observations, leading to overfitting and unstable coefficient estimates. The strong multicollinearity among spatially correlated voxels violates the independence assumption, inflating variance and reducing reliability. Additionally, linear regression lacks built-in feature selection to highlight key regions and does not respect the grouped structure of brain regions (e.g., W1, W2, ..., WG), potentially selecting isolated voxels rather than whole regions. Alternative methods like group lasso, which enforce group sparsity and incorporate regional priors, are better suited for this task, offering improved interpretability and predictive accuracy.

1.4 LASSO

Next, the following two methods are being considered:

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

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

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

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

  where w_g is the sub-vector of weights corresponding to group g, and G is the number of groups (e.g., brain regions).

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

Group LASSO is more appropriate for this setting compared to standard LASSO. In brain image analysis, where the goal is to identify the most predictive brain regions, the data is naturally structured into groups of voxels (e.g., regions $W_1, W_2, \ldots, W_G$). Group LASSO accounts for this structure by applying a penalty on the ℓ_2 norm of the weight sub-vectors ($\|w_g\|_2$) for each group, encouraging the selection or exclusion of entire regions rather than individual voxels. This aligns with the biological and interpretative need to evaluate regions holistically. In contrast, standard LASSO applies an ℓ_1 penalty ($\|w\|_1$) to individual voxel weights, leading to scattered selections across regions, which can fragment the analysis and reduce interpretability. Thus, Group LASSO better respects the grouped nature of the data and is more suitable for identifying predictive brain regions.

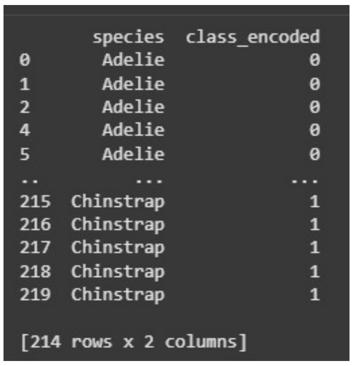
2 Logistic Regression

2.1 Load the data

The data was loaded according to the given code.

```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 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()
 # Make a copy to avoid the warning
 # Initialize the LabelEncoder
le = LabelEncoder()

 # Encode the species column
y_encoded = le.fit_transform(df_filtered['species'])
df_filtered['class_encoded'] = y_encoded

# Display the filtered and encoded DataFrame
print(df_filtered[['species', 'class_encoded']])
 # Split the data into features (X) and target variable (y)
y = df_filtered['class_encoded'] # Target variable
X = df_filtered.drop(['class_encoded'], axis=1)
```

The output obtained was as follows:



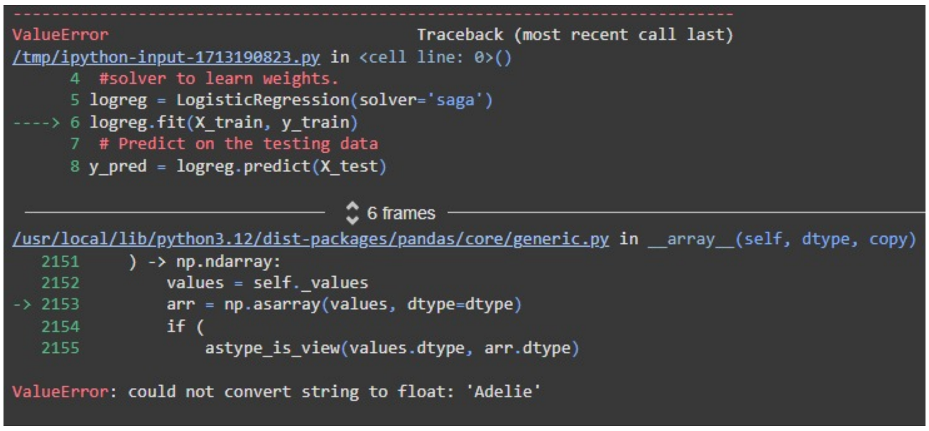
```
       species  class_encoded
0       Adelie              0
1       Adelie              0
2       Adelie              0
4       Adelie              0
5       Adelie              0
..         ...            ...
215  Chinstrap              1
216  Chinstrap              1
217  Chinstrap            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

[214 rows x 2 columns]
```

Train a logistic regression model. Here, did you encounter any errors? If yes, what were they, and how would you go about resolving them ?

Below code was used to train the logistic regression model.

```
#Split the data into training and testing sets
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. Here we are using saga
 #solver to learn weights.
logreg = LogisticRegression(solver='saga')
logreg.fit(X_train, y_train)
 # Predict on the testing data
y_pred = logreg.predict(X_test)
 # Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print(logreg.coef_, logreg.intercept_)
```

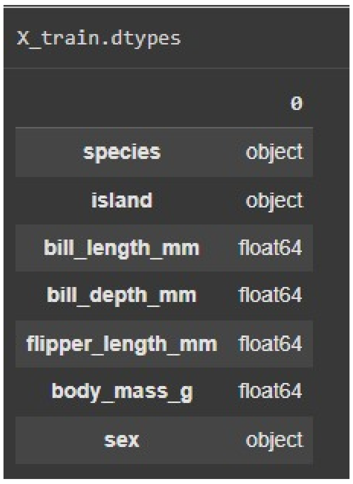
However, following error was encountered when running this code.



```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-1713190823.py in <cell line: 0>()
      4  #solver to learn weights.
      5 logreg = LogisticRegression(solver='saga')
----> 6 logreg.fit(X_train, y_train)
      7  # Predict on the testing data
      8 y_pred = logreg.predict(X_test)

                                    6 frames
/usr/local/lib/python3.12/dist-packages/pandas/core/generic.py in __array__(self, dtype, copy)
   2151     ) -> np.ndarray:
   2152         values = self._values
-> 2153         arr = np.asarray(values, dtype=dtype)
   2154         if (
   2155             astype_is_view(values.dtype, arr.dtype)

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

That means your feature matrix X_train still contains categorical string values such as "Adelie".

But scikit-learn's LogisticRegression can only handle numeric arrays (floats/integers). It doesn't know what to do with raw text labels. I checked the data types of the feature matrix and got the following. In my dataset, three columns were of type object, which led



X_train.dtypes

	0
species	object
island	object
bill_length_mm	float64
bill_depth_mm	float64
flipper_length_mm	float64
body_mass_g	float64
sex	object

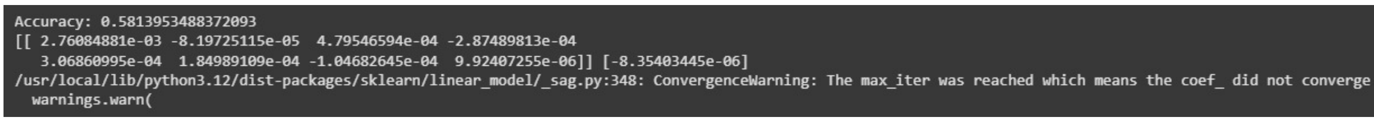
to the error since string values cannot be directly converted into floats. Out of these, the species column represents the target variable, while island and sex are input features. I excluded the target column from the training data, and then transformed the island and sex columns into numerical form so they could be used in the model.

```
X = df_filtered.drop(['class_encoded'], axis=1)

 # One-hot encode categorical features
X = pd.get_dummies(X, drop_first=True)
```

Then I ran the code and trained the model, and it gave the following result.

Why does the saga solver perform poorly?

The saga solver performs poorly on the penguins dataset because the dataset is relatively small and the features, such as bill length, flipper length, and body mass, exist on very



```
Accuracy: 0.5813953488372093
[[ 2.76084881e-03 -8.19725115e-05  4.79546594e-04 -2.87489813e-04
   3.06860995e-04  1.84989109e-04 -1.04682645e-04  9.92407255e-06]] [-8.35403445e-06]
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_sag.py:348: ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
```

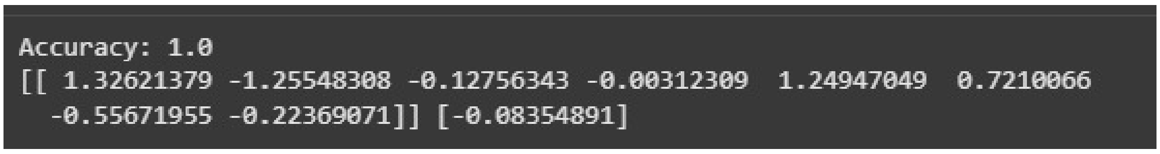
different scales. Since `saga` is a stochastic gradient-based method optimized for large, high-dimensional, and often sparse datasets, it struggles to converge efficiently in this smaller, dense setting. The solver is highly sensitive to feature magnitudes, so unscaled features cause unstable updates and prevent proper optimization of the model weights. Moreover, with the default `max_iter=100`, the algorithm often fails to converge, resulting in lower accuracy and inconsistent results. Proper feature scaling or increasing the maximum iterations is necessary to improve its performance on this dataset.

Changing the solver to liblinear

In the following code, I have changed the solver to liblinear by replacing 'saga' with 'liblinear' as follows.

```python

# Split the dataset into training and testing sets
features_train, features_test, labels_train, labels_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
# Initialize and train logistic regression
logistic_model = LogisticRegression(solver='liblinear')
logistic_model.fit(features_train, labels_train)
# Predict on test data
predicted_labels = logistic_model.predict(features_test)
# Evaluate model performance
test_accuracy = accuracy_score(labels_test, predicted_labels)
print(f"Accuracy: {test_accuracy:.4f}")
print(logreg.coef_, logreg.intercept_)
```

After using 'liblinear' solver, I received the following output.



```
Accuracy: 1.0
[[ 1.32621379 -1.25548308 -0.12756343 -0.00312309  1.24947049  0.7210066
  -0.55671955 -0.22369071]] [-0.08354891]
```

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

In the penguins dataset, the liblinear solver performs better than the saga solver mainly due to the dataset's small size and low-dimensional structure. The saga solver is designed for large-scale, high-dimensional problems and relies on stochastic gradient updates, which can be inefficient and unstable when applied to smaller datasets. In contrast, liblinear uses a coordinate descent optimization method that is deterministic, well-suited for binary classification, and converges quickly on small to medium-sized dense data. Furthermore, saga is highly sensitive to feature scaling, whereas liblinear is more robust in handling unscaled features. As a result, liblinear achieves faster convergence, produces more stable solutions, and delivers higher accuracy on this dataset compared to saga.

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

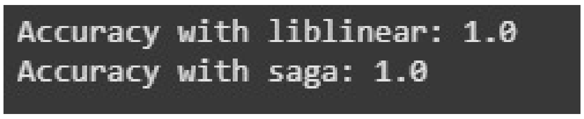
The model's accuracy with the saga solver varies with different random state values because saga is a stochastic optimizer. It updates the model weights using randomly selected subsets of the data, so the optimization path depends on the random initialization and the order in which samples are processed. On small datasets like the penguins data, this randomness has a stronger effect, often leading to different local solutions and variations in accuracy across runs. Additionally, changes in the train/test split under different random states can slightly alter the class balance and decision boundary, further influencing performance. In contrast, solvers such as liblinear are deterministic and therefore produce consistent results regardless of the random state.

Comparison of the performance of the "liblinear" and "saga" solvers with feature scaling.

The following code was used to scale the features and perform logistic regression using liblinear and saga solvers. Standard scaling was used to scale the features. For that `StandardScaler` from sklearn library was used.

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Divide dataset into training and testing sets
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, random_state
    =42)

# Normalize features using StandardScaler
scaler = StandardScaler()
X_tr_norm = scaler.fit_transform(X_tr)
X_te_norm = scaler.transform(X_te)

# Logistic Regression using liblinear solver
clf_liblinear = LogisticRegression(solver="liblinear")
clf_liblinear.fit(X_tr_norm, y_tr)
pred_lib = clf_liblinear.predict(X_te_norm)
acc_liblinear = accuracy_score(y_te, pred_lib)

# Logistic Regression using saga solver
clf_saga = LogisticRegression(solver="saga", max_iter=1000)
clf_saga.fit(X_tr_norm, y_tr)
pred_saga = clf_saga.predict(X_te_norm)
acc_saga = accuracy_score(y_te, pred_saga)

print(f"Liblinear Solver Accuracy: {acc_liblinear}")
print(f"Saga Solver Accuracy: {acc_saga}")
```

Accuracies obtained for the two solvers are as follows;



```
Accuracy with liblinear: 1.0
Accuracy with saga: 1.0
```

After applying feature scaling with the Standard Scaler, both models achieved perfect accuracy (1.0).

Without scaling, however, the saga solver performed poorly because features with different magnitudes caused inconsistent gradient steps, resulting in slower convergence and suboptimal weight updates. In contrast, the liblinear solver is less sensitive to feature scaling, so its accuracy remained unchanged.

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 not correct. Label encoding assigns integer values (e.g., 0, 1, 2) to categorical variables such as colors, which creates a false sense of ordinal relationship (e.g., green > blue > red) even though no such natural order exists. Applying scaling methods like Standard Scaler or Min-Max Scaler to these encoded integers further reinforces this misleading numerical relationship, potentially confusing the model. A better solution is to use **one-hot encoding**, which represents each category as a separate binary feature, eliminating any implied ordering. Since one-hot encoded variables are already in the range of 0 and 1, additional scaling is generally unnecessary, though it may still be applied depending on the learning algorithm being used.

3 Logistic regression First/Second-Order Methods

3.1 Generating Data (y - class labels, X - corresponding feature values)

Following code snippet was used to generate the data.

```python
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)
```

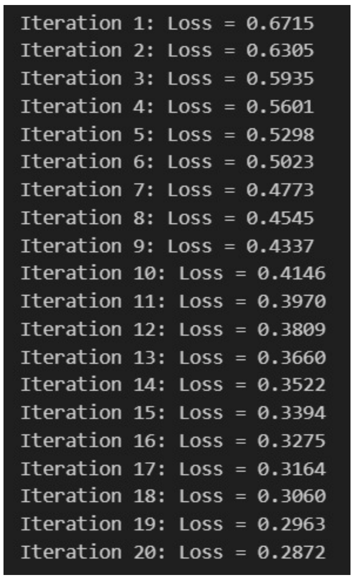
3.2 Implementing Batch Gradient Descent

The following code was used to implement batch Gradient descent to update the weights for the given dataset over 20 iterations.

```python

# Add the bias term (x0 = 1)
X_aug = np.c_[np.ones((X.shape[0], 1)), X]   # augmented feature matrix

# Parameters
m, d = X_aug.shape
alpha = 0.01          # learning rate
epochs = 20           # iterations
theta = np.random.randn(d) * 0.01   # random initialization
```

```python

# Sigmoid activation
def logistic(z):
    return 1.0 / (1.0 + np.exp(-z))

# Training with batch gradient descent
for epoch in range(epochs):
    # Forward pass (predictions)
    linear_out = X_aug @ theta
    h = logistic(linear_out)

    # Gradient calculation
    grad = (1/m) * (X_aug.T @ (h - y))

    # Update step
    theta -= alpha * grad

    # Loss function (log-likelihood)
    cost = -(1/m) * np.sum(y * np.log(h + 1e-8) + (1-y) * np.log(1-h + 1e-8)
        )

    print(f"Epoch {epoch+1}: Cost = {cost:.4f}")

# Plot decision boundary
x_axis = [np.min(X[:,0]) - 1, np.max(X[:,0]) + 1]
y_axis = -(theta[0] + theta[1]*np.array(x_axis)) / theta[2]

plt.scatter(X[:,0], X[:,1], c=y, cmap=plt.cm.Paired, alpha=0.7)
plt.plot(x_axis, y_axis, 'r-', label="Decision Boundary")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

The obtained losses for each iteration is as follows:



```
Iteration 1: Loss = 0.6715
Iteration 2: Loss = 0.6305
Iteration 3: Loss = 0.5935
Iteration 4: Loss = 0.5601
Iteration 5: Loss = 0.5298
Iteration 6: Loss = 0.5023
Iteration 7: Loss = 0.4773
Iteration 8: Loss = 0.4545
Iteration 9: Loss = 0.4337
Iteration 10: Loss = 0.4146
Iteration 11: Loss = 0.3970
Iteration 12: Loss = 0.3809
Iteration 13: Loss = 0.3660
Iteration 14: Loss = 0.3522
Iteration 15: Loss = 0.3394
Iteration 16: Loss = 0.3275
Iteration 17: Loss = 0.3164
Iteration 18: Loss = 0.3060
Iteration 19: Loss = 0.2963
Iteration 20: Loss = 0.2872
```

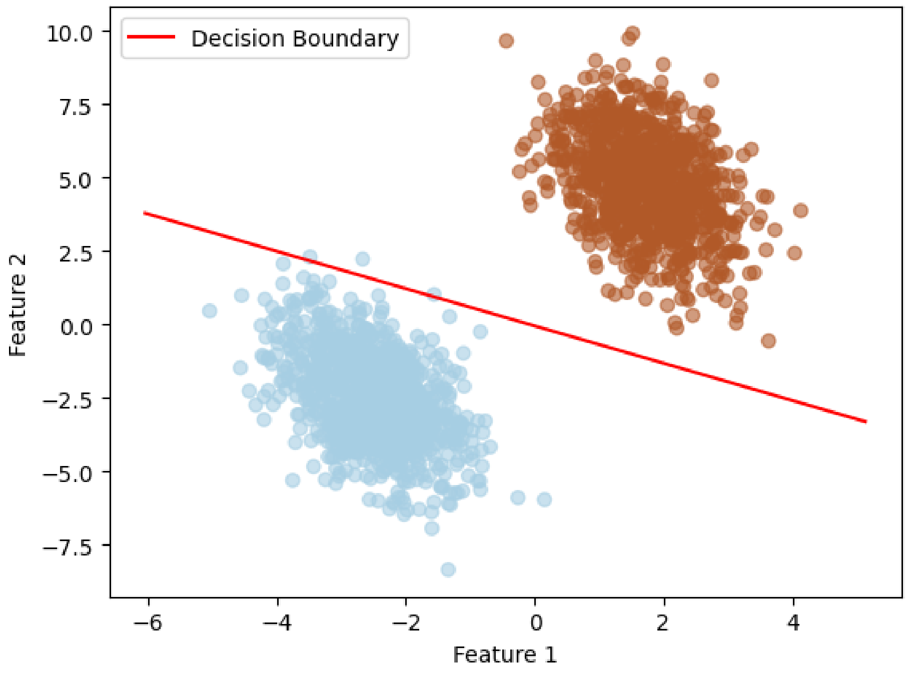


Figure 1: Decision Boundary Visualization

Weight Initialization Method

The weights were initialized using small random values drawn from a normal distribution with mean 0 and standard deviation 0.01:

$$w \sim \mathcal{N}(0, 0.012)$$

Reason for Weight Initialization Selection

- **Breaks symmetry:** If all weights were initialized to zero, every parameter would receive the same gradient during training, preventing the model from learning meaningful patterns.
- **Prevents saturation:** Very large initial weights can push the sigmoid function into regions where gradients vanish, slowing down learning.
- **Ensures stable and efficient convergence:** Small random initialization allows weights to start close to zero but not identical, leading to faster convergence in gradient descent.

3.3 Loss Function

Loss Function Used - Binary Cross-Entropy Loss was used

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

where:

- y_i is the true label (0 or 1)
- $\hat{y}_i$ is the predicted probability for class 1
- N is the number of samples

Reason for Selection

1. Suitable for binary classification: Binary cross-entropy directly measures the difference between predicted probabilities and actual class labels.

2. Smooth and differentiable: Allows efficient gradient computation for batch gradient descent.

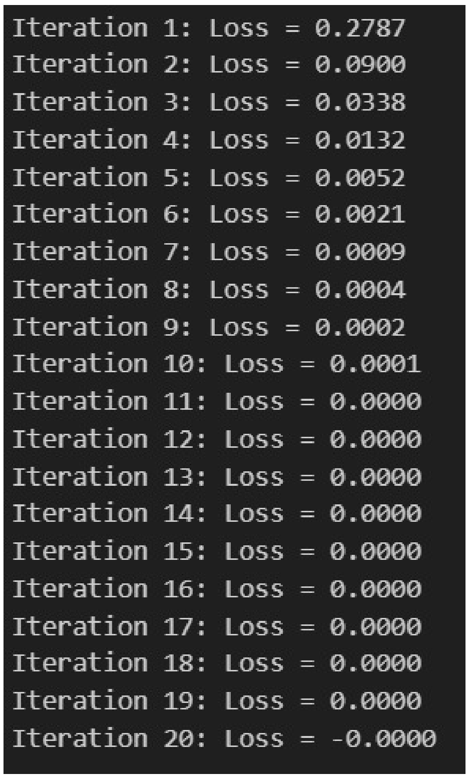
3. Penalizes confident wrong predictions more: Helps the model adjust weights effectively during training.

3.4 Implementing Newton's Method

The following method was used to implement Newton's method for this dataset.

```python

# Training loop using Newton-Raphson optimization
m = X.shape[0]   # number of samples

for epoch in range(n_iterations):
    # Linear prediction
    z_val = X @ theta
    pred = sigmoid(z_val)

    # Gradient computation
    grad = (1/m) * (X.T @ (pred - y))

    # Hessian matrix
    diag_r = np.diag(pred * (1 - pred))
    hessian = (1/m) * (X.T @ diag_r @ X)

    # Parameter update (Newton step)
    theta = theta - np.linalg.inv(hessian) @ grad

    # Cross-entropy loss
    cost = -(1/m) * np.sum(y * np.log(pred + 1e-8) + (1 - y) * np.log(1 -
        pred + 1e-8))

    print(f"Epoch {epoch+1}: Cost = {cost:.4f}")

# --- Decision boundary visualization ---
x_line = [np.min(X[:,1]) - 1, np.max(X[:,1]) + 1]
y_line = -(theta[0] + theta[1]*np.array(x_line)) / theta[2]

plt.scatter(X[:,1], X[:,2], c=y, cmap=plt.cm.coolwarm, edgecolors="k", alpha
    =0.7)
plt.plot(x_line, y_line, 'g--', label="Boundary")
plt.xlabel("Input Feature 1")
plt.ylabel("Input Feature 2")
plt.legend()
plt.show()
```

The obtained losses for each iteration is as follows.



```
Iteration 1: Loss = 0.2787
Iteration 2: Loss = 0.0900
Iteration 3: Loss = 0.0338
Iteration 4: Loss = 0.0132
Iteration 5: Loss = 0.0052
Iteration 6: Loss = 0.0021
Iteration 7: Loss = 0.0009
Iteration 8: Loss = 0.0004
Iteration 9: Loss = 0.0002
Iteration 10: Loss = 0.0001
Iteration 11: Loss = 0.0000
Iteration 12: Loss = 0.0000
Iteration 13: Loss = 0.0000
Iteration 14: Loss = 0.0000
Iteration 15: Loss = 0.0000
Iteration 16: Loss = 0.0000
Iteration 17: Loss = 0.0000
Iteration 18: Loss = 0.0000
Iteration 19: Loss = 0.0000
Iteration 20: Loss = -0.0000
```

The results show that Newton's Method converged extremely quickly. The loss starts at 0.2787 in the first iteration and drops to 0.0900 by the second iteration. By the fifth iteration, the loss has already reduced to 0.0052, and after the eighth iteration, it becomes negligible (less than 0.0005). From iteration 11 onward, the loss essentially remains at zero, indicating that the weights have already reached the minimum of the MSE loss function.

This behavior is expected because Newton's Method incorporates second-order derivative (Hessian) information, which captures the curvature of the loss function. Consequently, it can make larger and more precise updates toward the optimum compared to batch gradient descent, which relies solely on first-order gradient information. The plateau after iteration 11 confirms that the method has effectively converged to the global minimum, highlighting the remarkable efficiency and rapid convergence of Newton's Method in convex problems such as linear regression.

The equation for Newton's Method is given by:

$$w_{\text{new}} = w_{\text{old}} - H^{-1}\nabla L$$

where

$$\nabla L = \frac{2}{N}X^T(Xw - y) \quad \text{(gradient)}, \quad H = \frac{2}{N}X^T X \quad \text{(Hessian)}.$$

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

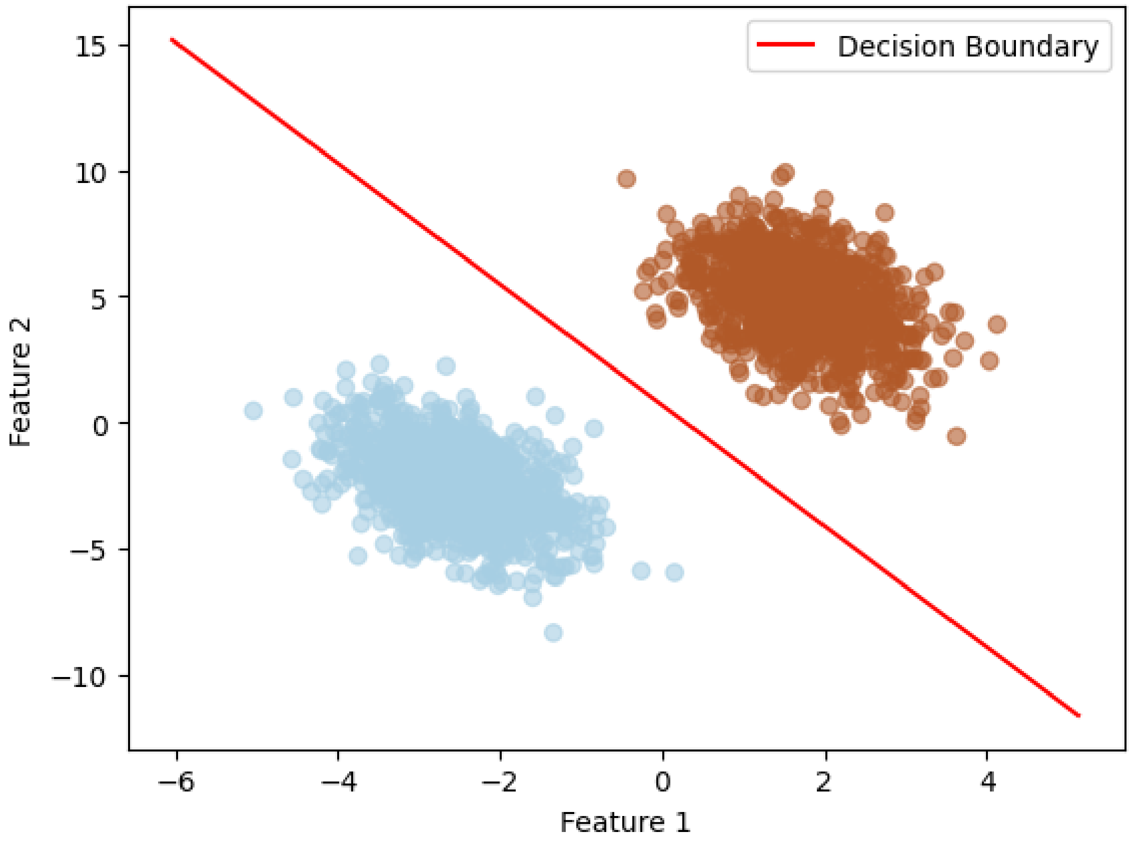


Figure 2: Decision Boundary Visualization using the Newton's method

Following code was used to plot the loss for both methods.

```

# Initialize weights
weights_bgd = np.random.randn(X.shape[1]) * 0.01
weights_newton = np.random.randn(X.shape[1]) * 0.01

# Store loss history
loss_history_bgd = []
loss_history_newton = []

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

# Binary Cross-Entropy Loss
def binary_cross_entropy(y_true, y_pred):
    eps = 1e-8
    return -np.mean(y_true * np.log(y_pred + eps) + (1 - y_true) * np.log(1
        - y_pred + eps))

for iteration in range(n_iterations):
    # -------------------------------
    # Batch Gradient Descent Update
    # -------------------------------
    pred_bgd = sigmoid(np.dot(X, weights_bgd))
    grad_bgd = (X.T @ (pred_bgd - y)) / X.shape[0]
    weights_bgd -= learning_rate * grad_bgd
    loss_history_bgd.append(binary_cross_entropy(y, pred_bgd))

    # ------------------------------
    # Newton's Method Update
    # ------------------------------
    pred_newton = sigmoid(np.dot(X, weights_newton))
    grad_newton = (X.T @ (pred_newton - y)) / X.shape[0]
    R = np.diag((pred_newton * (1 - pred_newton)).flatten())
    H = (X.T @ R @ X) / X.shape[0]
    weights_newton -= np.linalg.inv(H) @ grad_newton
    loss_history_newton.append(binary_cross_entropy(y, pred_newton))

# ------------------------------
# Plot the Loss Curves
# ------------------------------
plt.figure(figsize=(8,5))
plt.plot(range(1, n_iterations + 1), loss_history_bgd, 'o-', label='Batch
    Gradient Descent')
plt.plot(range(1, n_iterations + 1), loss_history_newton, 's-', label="
    Newton's Method")
plt.xlabel("Iteration")
plt.ylabel("Binary Cross-Entropy Loss")
plt.title("Loss Progression Over Iterations")
plt.legend()
plt.grid(True)
plt.show()
```

The plot obtained is as follows.

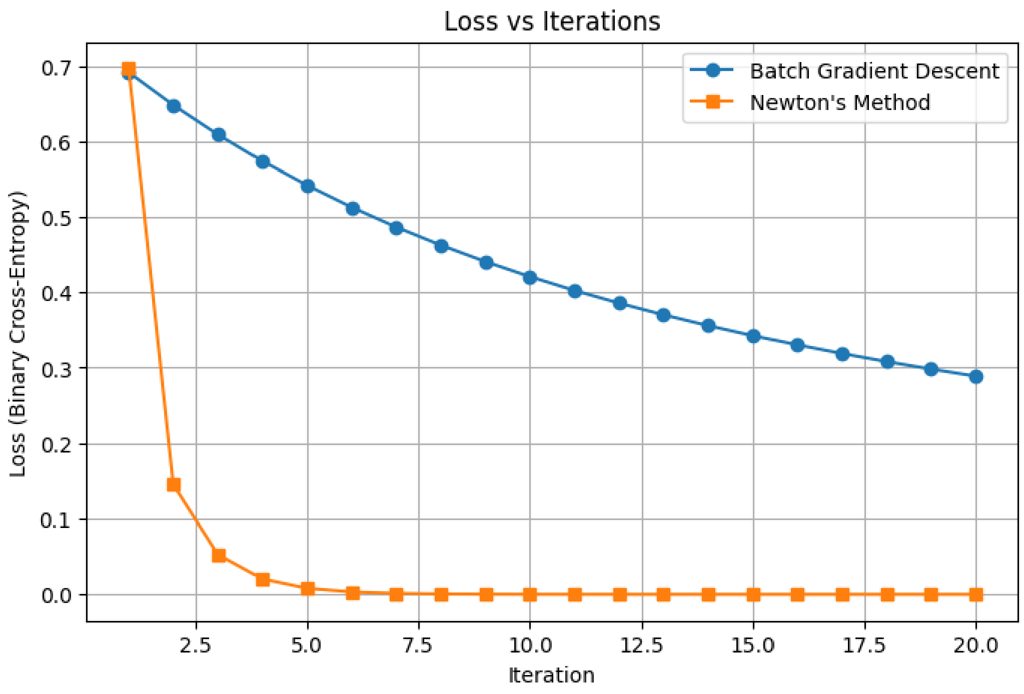


Figure 3: BCE Loss vs. Iterations for Batch Gradient Descent and Newton's Method

Comments on the results

When comparing the loss curves of Batch Gradient Descent (BGD) and Newton's Method, it is evident that Newton's Method converges much faster than BGD. The loss for Newton's Method drops sharply in the first few iterations, reaching near-minimum values quickly, while BGD exhibits a more gradual decrease in loss over the 20 iterations. This difference occurs because Newton's Method uses second-order derivative information (the Hessian) to adjust the weights more precisely, whereas BGD relies solely on the first-order gradient and a fixed learning rate. Although Newton's Method converges faster, it requires computation of the Hessian and its inverse, which can be expensive for large datasets, whereas BGD is simpler and scales better for larger problems. Overall, Newton's Method is more efficient for smaller datasets, while BGD provides a reliable, computationally cheaper alternative.

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

1. Convergence-Based Approach

- Stop iterating when the change in the loss function between consecutive iterations falls below a predefined threshold ϵ.
- Formula:
$$\text{Stop if } |L_{t+1} - L_t| < \epsilon$$
- This ensures that iterations continue only as long as the model is still learning significantly, avoiding unnecessary computations once convergence is reached.
- Works for both **Gradient Descent** and **Newton's Method**, though Newton's Method typically converges in fewer iterations.

2. Maximum Iteration Limit

- Set a fixed maximum number of iterations as a safeguard.
- For **Gradient Descent**: Usually higher (e.g., hundreds or thousands) because convergence is slower.
- For **Newton's Method**: Typically lower (e.g., 10–50) since convergence is faster.
- This prevents infinite loops in case convergence criteria fail or oscillations occur, and provides control over computational cost.

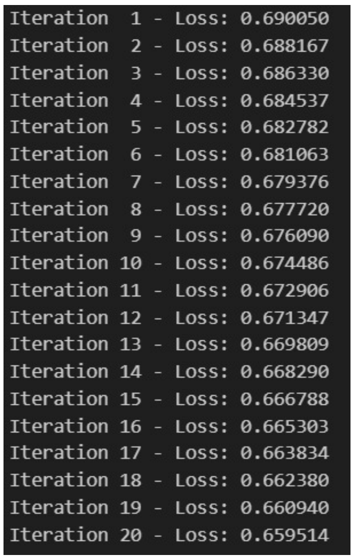
3.7 Suppose the centers in in listing 3 are changed to centers = [[2, 2], [5, 1.5]]. Use batch Gradient descent to update the weights for this new configuration. Analyze the convergence behavior of the algorithm with this updated data, and provide an explanation for convergence behavior.

The new code for data generation is as follows.

```

np.random.seed(0)
centers = [[2, 2], [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)
```

Batch Gradient Descent was implemented using the same code as above. The losses obtained for each iteration is as follows.



```
Iteration  1 - Loss: 0.690050
Iteration  2 - Loss: 0.688167
Iteration  3 - Loss: 0.686330
Iteration  4 - Loss: 0.684537
Iteration  5 - Loss: 0.682782
Iteration  6 - Loss: 0.681063
Iteration  7 - Loss: 0.679376
Iteration  8 - Loss: 0.677720
Iteration  9 - Loss: 0.676090
Iteration 10 - Loss: 0.674486
Iteration 11 - Loss: 0.672906
Iteration 12 - Loss: 0.671347
Iteration 13 - Loss: 0.669809
Iteration 14 - Loss: 0.668290
Iteration 15 - Loss: 0.666788
Iteration 16 - Loss: 0.665303
Iteration 17 - Loss: 0.663834
Iteration 18 - Loss: 0.662380
Iteration 19 - Loss: 0.660940
Iteration 20 - Loss: 0.659514
```

I plotted the Loss vs. iterations for the both instances in the same plot and got the following output.

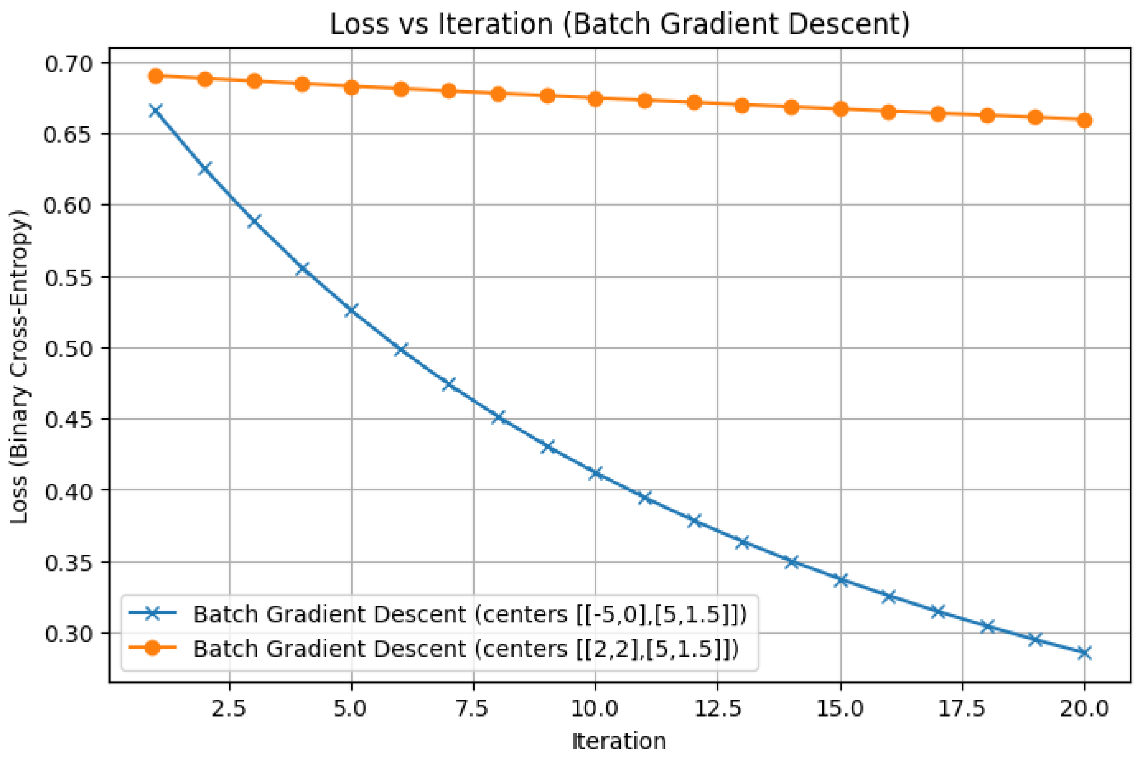


Figure 4: BCE Loss vs. Iterations for previous and new datasets

Explanation for the Convergence Behaviour

When the cluster centers are changed to [[2, 2], [5, 1.5]], Batch Gradient Descent still reduces the binary cross-entropy loss but the convergence behavior changes compared to the earlier, more-separated clusters. Because these new centers are closer together and occupy a similar region of the feature space (after the linear transformation), the two classes overlap more; this reduces separability and makes the optimization landscape flatter and noisier. As a result, the loss decreases more slowly and may settle at a higher final value within the same number of iterations. The learning rate also affects this behavior: if it is too large the loss may oscillate, and if it is too small convergence will be very slow. In practice, for this configuration you may observe a gentler slope in the loss curve (slower progress per iteration) and possibly a higher plateau, indicating that the model needs more iterations, regularization or feature scaling (standardization) to reach similar performance to the well-separated case. Newton's method (if applied) would likely still converge faster in iterations because it uses curvature information, but the final achievable loss is bounded by the data's inherent overlap.