EN3150 Assignment 02

Learning from Data and Related Challenges and Classification

Submitted by:
P. G. R. S. U. Bandara
Index Number: **220065A**

Department of Electrical Engineering
University of Moratuwa

September 9, 2025

Contents

1 Linear Regression

1.1 OLS fitted line misalignment

Ordinary Least Squares (OLS): OLS chooses the best fit line by minimizing the squared errors between the predicted values $\hat{y}_i$ and the true values y_i.

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

where,
y_i : actual data point
$\hat{y}_i$: predicted value of the line for x_i

Squaring the errors ensures that all deviations are positive and gives more weight to large errors. The OLS line is chosen to minimize the *total squared residuals*, rather than to pass through the majority of points. Consequently, outliers have a large influence because their squared residuals dominate the loss function. As a result, a outliers can cause the OLS fitted line to deviate from the majority of data points.

Conclusion: The OLS fitted line does not align with most of the data points because it minimizes the sum of squared errors, not the number of points the line passes through. The presence of outliers can cause the fitted line to deviate from the majority of the points.

1.2 Reduce the impact of outliers

Modified Loss Function:

The modified loss function is defined as:

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

where,
y_i : actual data point
$\hat{y}_i$: predicted value of the line for x_i
a_i : weight assigned to each point

The OLS method minimizes the sum of squared differences between y_i and $\hat{y}_i$ for all points. However, outliers can pull the OLS line away from the inliers, resulting in a poor fit for the majority of the data. The modified loss function uses weights a_i to reduce the impact of outliers, allowing a better fit for inliers.

Scheme 1:

$$a_i = \begin{cases} 0.01 & \text{for outliers} \\ 1 & \text{for inliers} \end{cases}$$

In this setting, outliers receive a very small weight ($a_i = 0.01$), so their squared differences have minimal effect on the line's fit. Inliers retain a normal weight ($a_i = 1$), which allows the line to focus on fitting the inliers well. This should make the fitted line align better with the majority of the data points.

Scheme 2:

$$a_i = \begin{cases} 5 & \text{for outliers} \\ 1 & \text{for inliers} \end{cases}$$

Here, outliers are given a large weight ($a_i = 5$), so their squared differences have a much bigger effect on the line. This pulls the line toward the outliers even more than OLS, worsening the fit for inliers.

Justification:
The OLS line is not aligned with the majority of data points due to the influence of outliers. The goal is to improve the fit for inliers. Scheme 1 reduces the influence of outliers by assigning them a tiny weight ($a_i = 0.01$), allowing the model to focus on inliers ($a_i = 1$). Consequently, the fitted line will be closer to the inlier cluster of points. On the other hand, Scheme 2 increases the influence of outliers, pulling the line away from the inliers and providing a worse fit.

Conclusion: Scheme 1 is expected to provide a better fitted line for inliers than the OLS line because it minimizes the impact of outliers, allowing the line to better match the majority of data points.

1.3 Linear regression for identifying brain regions

Brain imaging data (e.g., fMRI or EEG) typically has the following characteristics:

- **High dimensionality:** There are thousands of features (voxels in brain regions), but usually only hundreds of subjects (samples).
- **High correlation:** Neighboring brain regions tend to activate together, which makes features highly collinear.

The researcher's goal is to identify which brain regions are most predictive of a specific cognitive task. While linear regression might seem like a straightforward choice, it is not well-suited for this purpose.

Linear regression assigns a weight to every region, even irrelevant or noisy ones, making interpretation difficult since the model does not automatically remove unimportant features. In addition, the high correlation among brain regions leads to collinearity, which makes the regression coefficients unstable: small changes in the data can cause large variations in the estimated weights, making results unreliable. Furthermore, linear regression generally requires more samples than features, but in brain imaging the opposite is true (many features and few subjects). This leads to severe overfitting and unstable models. As a result, the coefficients obtained from linear regression do not reliably indicate which brain regions are truly important, and large weights may appear on regions that are not predictive.

Conclusion: Linear regression is not suitable because brain imaging data is high-dimensional, noisy, and strongly correlated. It tends to overfit, produce unstable coefficients, and mislead the researcher about which brain regions are truly predictive of the cognitive task.

1.4 Standard LASSO vs Group LASSO

When studying the brain using machine learning, two common methods for feature selection are Standard LASSO and Group LASSO. The choice of method depends on whether the goal is to identify predictive voxels (tiny measurement units in brain scans) or predictive regions (larger groups of voxels corresponding to meaningful brain areas). In this case, the researcher aims to identify which brain regions are most predictive of a cognitive task.

Method A: Standard LASSO

The LASSO objective is to minimize::

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

Standard LASSO performs feature selection at the voxel level, treating each voxel as an independent predictor. This often results in scattered voxel selection across the brain, which is difficult to interpret in terms of brain regions. Moreover, the method can be unstable: small changes in the data may lead to different subsets of voxels being selected.

Method B: Group LASSO

The Group LASSO objective is to minimize:

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

Here, w_g is the sub-vector of weights corresponding to group g, and G is the number of groups (e.g., brain regions). Group LASSO performs feature selection at the region level, selecting or discarding entire sets of voxels together. This approach makes the results more interpretable and biologically meaningful. Group LASSO also tends to produce more stable results, as it treats predefined regions as analysis units.

Justification: The scientific question concerns identifying *brain regions*, not individual voxels. Standard LASSO, which focuses on voxel-level detail, is too fine-grained, unstable, and difficult to interpret. In contrast, Group LASSO aligns directly with the research goal by providing region-level feature selection and producing stable, interpretable outcomes.

Conclusion: Group LASSO is more appropriate for this setting. Since the aim is to identify predictive brain regions, Group LASSO offers region-level selection that is more interpretable, biologically meaningful, and stable. Standard LASSO, while useful for voxel-level sparsity, would lead to scattered and less interpretable findings that do not address the research question.

2 Logistic Regression

2.1 Error in Logistic Regression Training

The task was to train a logistic regression model to predict "Adelie" vs. "Chinstrap" species using the penguins dataset. The workflow included splitting the data, training the model, making predictions, and evaluating the results. However, during training the following error was encountered (see Figure 1):

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-244678139.py in <cell line: 0>()
      4 #Train the logistic regression model. Here we are using saga solver to learn weights.
      5 logreg = LogisticRegression(solver='saga')
----> 6 logreg.fit(X_train, y_train)
      7
      8 # Predict on the testing data

                              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'
```

Figure 1: ValueError encountered while fitting the logistic regression model.

This error occurred because the logistic regression model expects numerical input features, but the feature matrix `X` contained non-numerical (string) values. Logistic regression cannot directly process categorical strings such as "Adelie" or "Male."

```
X.head()
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female
5	Adelie	Torgersen	39.3	20.6	190.0	3650.0	Male

Figure 2: Feature matrix `X` showing both numerical and categorical columns.

From Figure 2, we can see that the feature matrix `X` contains:

- **Numerical columns:** `bill_length_mm`, `bill_depth_mm`, `flipper_length_mm`, `body_mass_g`.
- **Categorical columns:** `species`, `island`, `sex`.

Since categorical columns are strings, they caused the model to fail. In particular, the presence of "Adelie" in the `species` column triggered the error.

How to Resolve the Error

- The `species` column should be **removed from** `X`, since it is already the target variable (`y`) and including it would cause data leakage.
- The columns `island` and `sex` can either be:
  - Dropped entirely if not needed, or
  - Encoded into numerical format using methods such as one-hot encoding if we want to include them as features.
- After removing or encoding these categorical columns, the feature matrix `X` will contain only numerical values, making it suitable for logistic regression.

Conclusion: The error occurred because logistic regression requires numerical features but encountered categorical strings in the dataset. The solution is to preprocess the data by dropping the redundant `species` column and either removing or encoding `island` and `sex`. This ensures that the model trains successfully on a fully numerical feature matrix.

2.2 Why does the saga solver perform poorly?

The `saga` solver is one of the optimization algorithms used by scikit-learn's `LogisticRegression` to estimate model weights. It is based on *Stochastic Average Gradient Descent* , which updates weights in small batches of data and supports both L1 (lasso) and L2 (ridge) regularization. This makes it efficient for large datasets and useful for feature selection when L1 regularization is applied.

After preprocessing the dataset to avoid the earlier `ValueError`, the logistic regression model was trained with the solver hyperparameter set to `saga`. The following results were obtained (see Figure 3):

- Accuracy: 0.58 (close to random guessing for a binary classification task).
- Convergence warning: `The max_iter was reached which means the coef_ did not converge`.

```
Accuracy: 0.5813953488372093
[[ 2.76355051e-03 -8.47540578e-05  4.63083058e-04 -2.86639594e-04]] [-8.52973566e-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(
```

Figure 3: Output after training with the `saga` solver.

Reasons for Poor Performance

The saga solver does not always perform poorly, but in this case several factors contribute to suboptimal performance:

1. **Feature scaling:** The saga solver works best when all features are on a similar scale. In the penguins dataset, features vary widely (e.g., `bill_length_mm` $\approx$ 30–50 vs. `body_mass_g` $\approx$ 3000–5000). Without scaling, larger-valued features dominate the optimization, leading to unstable weight updates and poor accuracy.

2. **Dataset size:** The saga solver is designed for large datasets because it relies on stochastic updates. After filtering for "Adelie" and "Chinstrap," the dataset contains only 214 samples with 7 features (see Figure 4). For such a small dataset, stochastic updates are noisy and unstable, making solvers like `liblinear` more appropriate.

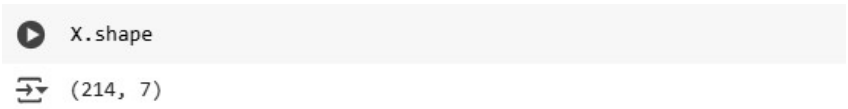

Figure 4: Shape of the feature matrix (`X`): 214 samples, 7 features.

3. **Regularization choice:** By default, the saga solver applies L2 (ridge) regularization, which shrinks weights but keeps all features. If some features are irrelevant, L2 cannot remove them, which may reduce accuracy. L1 (lasso) or elastic net regularization could be better choices, since they allow irrelevant feature weights to shrink to zero, improving interpretability and potentially accuracy.

Conclusion

The saga solver performed poorly in this case due to unscaled features, a relatively small dataset, and the default choice of L2 regularization. For small datasets like this, alternative solvers (e.g., `liblinear`) or appropriate preprocessing (feature scaling and penalty adjustment) would yield more stable and accurate results.

2.3 Classification Accuracy with `liblinear` Solver

After resolving the categorical data issue by dropping the columns `species`, `island`, and `sex`, the logistic regression model was retrained using the `liblinear` solver. Continuing from the previous preprocessing steps, the following results were obtained (see Figure 5):

- Accuracy: 1.0 (100%)
- Coefficients: $[1.4542,\ -0.9394,\ -0.1657,\ -0.0039]$
- Intercept: -0.0479

Figure 5: Classification results with the `liblinear` solver.

The classification accuracy achieved with the `liblinear` solver is **1.0**, meaning the model correctly predicted the species (*Adelie* or *Chinstrap*) for all test samples. In simple terms, the model made no mistakes: every penguin in the test set was classified correctly based on the numerical features provided.

2.4 "Liblinear" solver perform better than "Saga" solver?

The `liblinear` solver is another optimization algorithm, like the previously mentioned `saga` solver, used by scikit-learn's `LogisticRegression` to estimate model weights. It uses a method called *coordinate descent*, which adjusts one weight at a time while keeping the others fixed. This makes small, precise updates and is particularly effective for small datasets. It works especially well for binary classification problems, such as distinguishing between Adelie and Chinstrap penguins. Like `saga`, the `liblinear` solver supports both L1 (lasso) and L2 (ridge) regularization. L1 can set some weights to zero (feature selection), while L2 shrinks weights to prevent overfitting.

For small datasets, `liblinear` is often a better choice because it processes the entire dataset at once and updates weights systematically. This leads to stable and accurate solutions, which explains the perfect accuracy (1.0) obtained in this case. In contrast, the `saga` solver relies on stochastic updates, processing small batches of data. While this is efficient for large datasets, it can introduce noise into weight updates for small datasets, leading to slower convergence or less accurate solutions.

Moreover, the `saga` solver requires features to be on approximately the same scale for fast and reliable convergence. Without proper scaling, large-valued features (e.g., `body_mass_g`, which ranges from 3000–5000) can dominate over smaller-valued features (e.g., `bill_length_mm`, which ranges from 30–50), confusing the algorithm. The `liblinear` solver, however, is less sensitive to unscaled features because it systematically processes the entire dataset. While preprocessing and scaling are still recommended, `liblinear` can achieve strong performance even without them, making it highly effective for this dataset.

2.5 Model's accuracy varies with different random state values

The variation in accuracy with different `random_state` values occurs because the train-testsplit function randomly shuffles the dataset before dividing it into training and testing sets. The parameter `random_state` acts as a seed for this random shuffling. Using the same value reproduces the same split, while changing it (e.g., to 10, 50, 100) generates a different shuffle and split.

In the filtered penguins dataset with 214 samples, the test set contains 43 samples (20%). Each split therefore assigns a different subset of samples to the test set, which directly affects accuracy.

The logistic regression model with the `saga` solver was trained to classify penguins as Adelie or Chinstrap. Running the model with different `random_state` values produced the following results:

- `random_state = 0:` Accuracy = 0.6512, small coefficients, `ConvergenceWarning`.
- `random_state = 10:` Accuracy = 0.6977, small coefficients, `ConvergenceWarning`.
- `random_state = 50:` Accuracy = 0.7674, small coefficients, `ConvergenceWarning`.
- `random_state = 100:` Accuracy = 0.6744, small coefficients, `ConvergenceWarning`.

```
Accuracy: 0.6511627906976745
[[ 2.93042495e-03 -9.49690301e-05  2.01449364e-04 -2.49858584e-04]] [-9.85828649e-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(

Accuracy: 0.6976744186046512
[[ 3.18027257e-03 -7.61524443e-05  1.15344683e-03 -2.95202986e-04]] [-3.47726839e-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(

Accuracy: 0.7674418604651163
[[ 0.00305113 -0.00010503  0.00035362 -0.00022653]] [-9.02023187e-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(

Accuracy: 0.6744186046511628
[[ 2.93028079e-03 -7.03761740e-05  6.93114738e-04 -2.67616619e-04]] [-7.74185306e-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(
```

Figure 6: Classification accuracy results with different `random_state` values using the saga solver.

Each run also generated a `ConvergenceWarning`, indicating that the `saga` solver did not fully converge to optimal weights within the default iteration limit (`max_iter=100`).

The accuracy differences can be explained as follows:

- Each test set $(X_{\text{test}}, y_{\text{test}})$ is unique. Some splits may contain penguins with clearer feature differences, leading to higher accuracy (e.g., 0.7674 for `random_state=50`).
- Other splits may contain penguins with overlapping or similar features, making classification harder and reducing accuracy (e.g., 0.6512 for `random_state=0`).
- The `saga` solver converges quickly only when features are scaled to similar ranges. In this dataset, features such as `body_mass_g` (3000–5000) dominate smaller features such as `bill_length_mm` (30–50). This imbalance causes the solver to assign suboptimal weights, contributing to unstable results across different splits.

Thus, the model's accuracy varies with `random_state` because of (i) differences in which samples end up in the test set, and (ii) solver sensitivity to unscaled features.

2.6 Performance of the solvers with feature scaling

The performance of the `liblinear` and `saga` solvers was compared with and without feature scaling. The results are summarized below:

Solver	**Without Scaling ($X_{\text{preprocess}}$)**	**With Scaling (X_{scaled})**
`saga`	0.5814 (58.14%)	0.9767 (97.67%)
`liblinear`	1.0 (100%)	0.9767 (97.67%)

Table 1: Classification accuracy of solvers with and without feature scaling (random_state = 42).

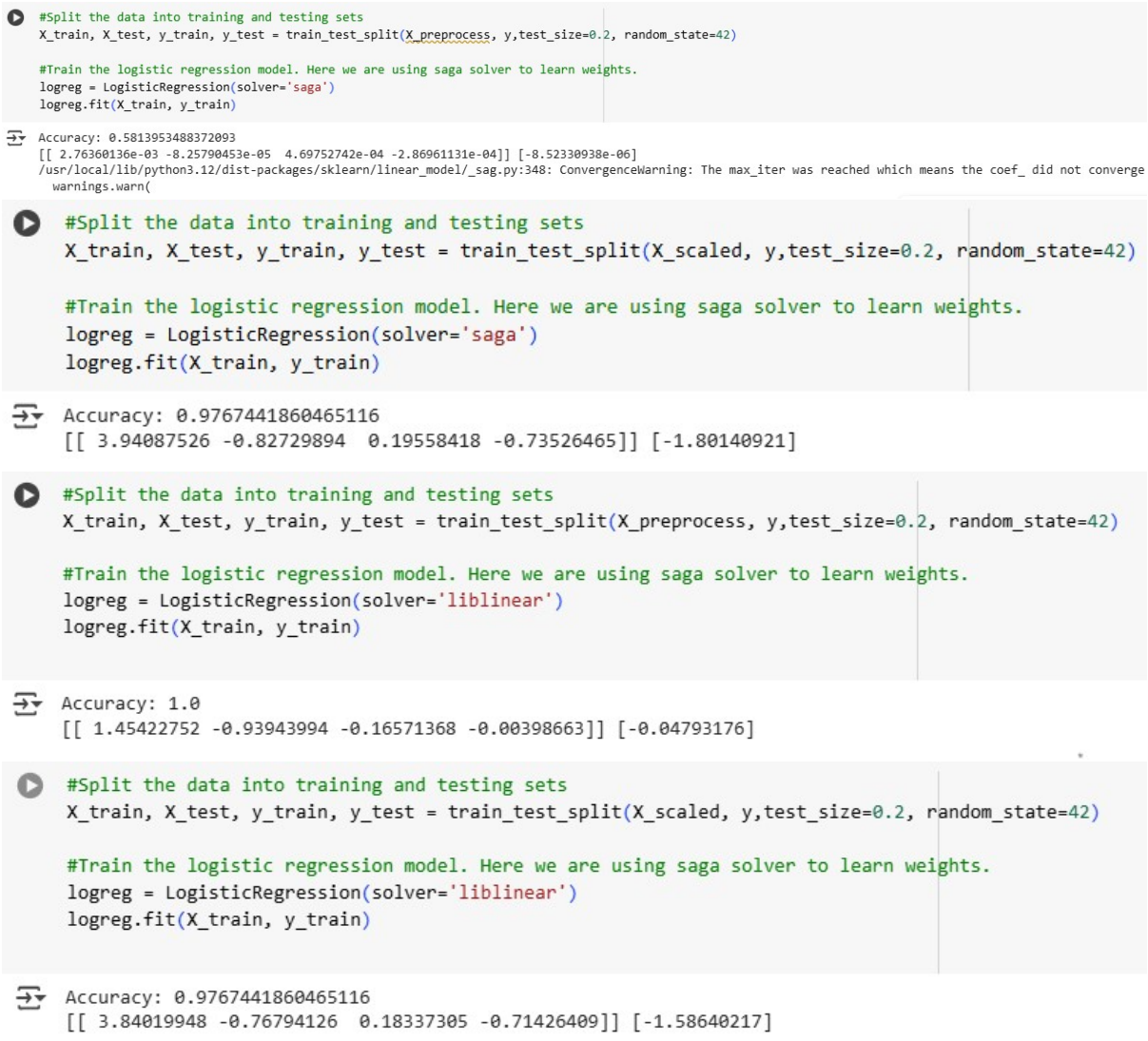

Figure 7: Solver accuracies with and without feature scaling.

Impact of scaling on saga:
The accuracy of the `saga` solver increased significantly from 58.14% to 97.67%, representing a 41.5% improvement after applying feature scaling. The `saga` algorithm uses Stochastic Average Gradient descent, where weight updates depend heavily on feature gradients. When features have very different ranges (for example, `body_mass_g` ranges between 3000–5000 while `bill_length_mm` ranges between 30–50), the larger features dominate the updates. This imbalance leads to poor convergence and inaccurate decision boundaries. Applying feature scaling (using `StandardScaler`, which transforms features to have mean 0 and standard deviation 1) balances the feature ranges, stabilizes gradient updates, and allows the solver to converge more effectively. As a result, the model produces meaningful coefficients and a near-optimal decision boundary. Therefore, feature scaling corrects the sensitivity of `saga` to unscaled features, leading to substantially higher accuracy.

Impact of scaling on liblinear:
Without feature scaling, the `liblinear` solver already achieved 100% accuracy. After applying feature scaling, the accuracy slightly decreased to 97.67%. The `liblinear` algorithm uses coordinate descent, which optimizes one weight at a time using the entire dataset. This makes it less sensitive to feature scales because it directly minimizes the error rather than relying on gradient magnitudes. Scaling can slightly shift the decision boundary, which in this case led to one or two misclassifications in the test set. Overall, `liblinear` remains robust, and feature scaling has only a minor effect on its performance.

Conclusion:
Feature scaling has a **major positive impact** on the `saga` solver, as it corrects feature imbalance and significantly improves accuracy. In contrast, scaling has a **minor effect** on the `liblinear` solver, which is inherently more stable for small datasets. The slight drop in accuracy observed with `liblinear` is due to minor adjustments in the decision boundary rather than limitations of the solver itself.

2.7 Is Scaling Label-Encoded Categorical Features Correct?

Label encoding converts categorical values (text labels) into numbers. For example, using `LabelEncoder` from `sklearn.preprocessing` for ['red', 'blue', 'green', 'blue', 'green'], we might have:

$$\begin{aligned} \text{red} &\to 0 \\ \text{blue} &\to 1 \\ \text{green} &\to 2 \end{aligned}$$

Resulting encoded values: $[0, 1, 2, 1, 2]$. This makes categorical data usable for machine learning models that require numerical input (e.g., logistic regression). However, the numbers 0, 1, 2 are arbitrary and do not represent meaningful order or magnitude. For instance, green (2) is not "twice as much" as blue (1) or "further away" from red (0).

Feature scaling (e.g., Standard Scaling or Min-Max Scaling) adjusts numerical values to a common range to ensure fair treatment by machine learning models. For example:

- **Standard Scaling:** Transforms data to have mean 0 and standard deviation 1.

$$z = \frac{x - \text{mean}}{\text{std}}$$

  For $[0, 1, 2, 1, 2]$, scaled values might be $[-1.434, -0.239, 0.957, -0.239, 0.957]$.

- **Min-Max Scaling:** Scales data to a fixed range (e.g., 0 to 1).

$$x' = \frac{x - \min}{\max - \min}$$

  For $[0, 1, 2, 1, 2]$, scaled values would be $[0, 0.5, 1, 0.5, 1]$.

Applying feature scaling to label-encoded categorical features is **not correct**. The encoded numbers do not have inherent numerical relationships, and scaling implies artificial

distances between categories, which can mislead models like logistic regression. For example, scaling suggests green (0.957) is "further" from red (-1.434) than blue (-0.239), which is meaningless.

Label encoding is appropriate for ordinal features (where order matters, e.g., low, medium, high), but for nominal features (no order, e.g., colors or islands in penguins), one-hot encoding is recommended. One-hot encoding creates separate binary columns for each category:

is_red	**is_blue**	**is_green**
1	0	0
0	1	0
0	0	1
0	1	0
0	0	1

Each category is treated independently, avoiding artificial numerical relationships. Models like logistic regression can use these binary columns directly, with no need for scaling. Scaling them would produce meaningless values (e.g., -0.5, 0.5) and confuse the model.

Conclusion:
Applying feature scaling to label-encoded categorical features is incorrect and can mislead machine learning models. Label encoding assigns arbitrary numbers to categories, which do not represent meaningful order or magnitude, and scaling these numbers creates artificial distances that can distort model decisions. For nominal categorical features, the recommended approach is one-hot encoding, which creates separate binary columns for each category, allowing models like logistic regression to interpret each category independently without introducing false numerical relationships. These binary features do not require scaling, as their values are already on a consistent range (0 or 1). Label encoding is only suitable for ordinal features where the numerical order reflects meaningful relationships, and even then, scaling is rarely needed unless the ordinal values represent meaningful numerical differences that affect the model.

3 Logistic Regression First/Second-Order Methods

3.1 Generates a synthetic dataset

The provided Python code generates a synthetic dataset specifically designed for binary classification tasks. This dataset simulates two distinct groups or *clusters* of data points in a two-dimensional space. The function `make_blobs` is used to create the data, with two specified centers:

- Center 1: $(-5, 0)$ – generates points mostly in the left/lower part of the plot.
- Center 2: $(5, 1.5)$ – generates points mostly in the right/upper part of the plot.

Since the centers are far apart, the two classes are initially easy to separate using a linear decision boundary.

The dataset consists of:

- X: A two-dimensional array of shape $(2000, 2)$, where each row represents a data point with two features (the x- and y-coordinates in 2D space). The points are distributed as Gaussian blobs (normal distributions) around each center, with some random spread (default standard deviation ≈ 1).
- y: A one-dimensional array of shape $(2000, 1)$ containing integer class labels, with 0 assigned to points near the first center and 1 assigned to points near the second center. The dataset is balanced, containing approximately 1000 points per class.

An example of the generated arrays is:

$$X = \begin{bmatrix} -2.1965 & -6.0482 \\ 0.8179 & 4.8793 \\ 2.8314 & 5.8030 \\ \vdots & \vdots \\ -3.2667 & -2.6252 \\ 2.2305 & 4.0832 \\ 0.8170 & 4.8915 \end{bmatrix}, \quad y = [0,\ 1,\ 1,\ \ldots,\ 0,\ 1,\ 1]$$

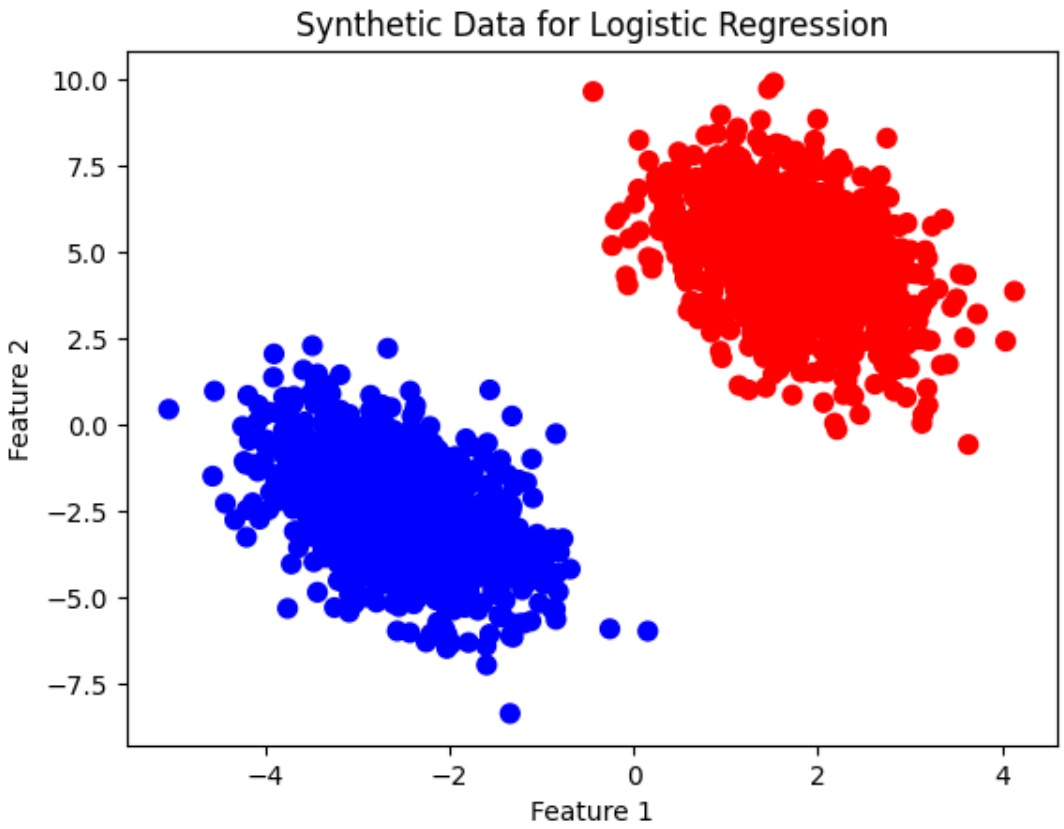

Figure 8: Scatter plot of the synthetic dataset generated by `make_blobs`. Two distinct clusters corresponding to the two classes can be observed.

Thus, the dataset is ideal for testing binary classifiers, as it provides two clearly separable classes in a 2D space.

3.2 Implement Batch Gradient Descent

We implement **logistic regression** using **batch gradient descent** because y is a binary class label.

The prediction is given by:

$$\hat{y} = \sigma(W \cdot X + b), \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

Here, W are the weights, b is the bias, and $\sigma(z)$ is the sigmoid activation function. The loss function is **binary cross-entropy**.

The gradient update rules are:

$$W := W - \eta \cdot \frac{1}{m} X^T (\hat{y} - y), \quad b := b - \eta \cdot \frac{1}{m} \sum (\hat{y} - y)$$

where $m = 2000$ is the number of samples, and η is the learning rate.

The sigmoid function is used because: The raw prediction $z = W \cdot X + b$ can take any real value ranging from $-\infty$ to $+\infty$. The sigmoid function then maps this value into the range $(0, 1)$, which can be interpreted as the probability. After applying the sigmoid, classification is performed using a threshold rule:

$$\hat{y} = \begin{cases} 1, & \text{if } \sigma(z) \geq 0.5 \\ 0, & \text{otherwise} \end{cases}$$

The model was trained for 20 iterations. After the 20 iterations, we can get boundary line (Feature2 = $-0.8920\times$ Feature1 $+0.1039$.) as following

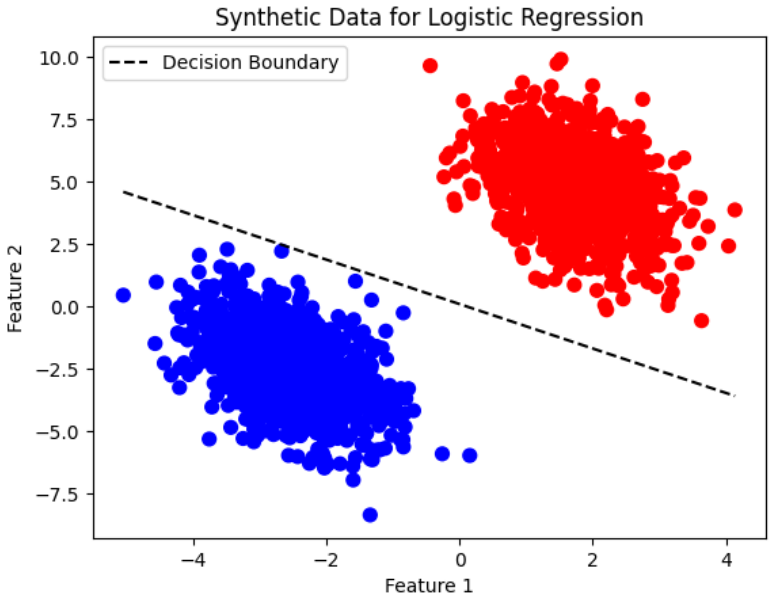

Figure 9: Synthetic dataset with decision boundary learned from logistic regression. The decision boundary equation is: Feature2 = $-0.8920\times$ Feature1 $+0.1039$.

The loss decreased steadily, showing convergence of batch gradient descent.

The results are:

- Iteration 1/20: Loss = 0.6904
- Iteration 2/20: Loss = 0.3539
- Iteration 3/20: Loss = 0.2437
- Iteration 4/20: Loss = 0.1894
- Iteration 5/20: Loss = 0.1564
- Iteration 10/20: Loss = 0.0874
- Iteration 15/20: Loss = 0.0623
- Iteration 20/20: Loss = 0.0490

Final parameters obtained:

$$W = [0.54204219,\ 0.66313971], \quad b = -0.0529956673$$

```
Iteration 1/20, Loss: 0.6904
Iteration 2/20, Loss: 0.3539
Iteration 3/20, Loss: 0.2437
Iteration 4/20, Loss: 0.1894
Iteration 5/20, Loss: 0.1564
Iteration 6/20, Loss: 0.1340
Iteration 7/20, Loss: 0.1178
Iteration 8/20, Loss: 0.1053
Iteration 9/20, Loss: 0.0954
Iteration 10/20, Loss: 0.0874
Iteration 11/20, Loss: 0.0807
Iteration 12/20, Loss: 0.0751
Iteration 13/20, Loss: 0.0702
Iteration 14/20, Loss: 0.0660
Iteration 15/20, Loss: 0.0623
Iteration 16/20, Loss: 0.0590
Iteration 17/20, Loss: 0.0561
Iteration 18/20, Loss: 0.0535
Iteration 19/20, Loss: 0.0511
Iteration 20/20, Loss: 0.0490
Final weights: [0.54204219 0.66313971]
Final bias: -0.05299566730950298
```

Figure 10: Loss values during 20 iterations of batch gradient descent.

Weight Initialization

The weights W were initialized using small random values sampled from a normal distribution with mean 0 and variance 0.01^2, while the bias b was initialized to 0:

$$W \sim \mathcal{N}(0, 0.01^2), \quad b = 0$$

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

# Initialize weights
np.random.seed(42)
W = np.random.randn(X.shape[1]) * 0.01  # small random initialization
b = 0.0

# Hyperparameters
lr = 0.1 #learning-rate
iterations = 20

print(f"Initial weights:{W}, Initial bias:{b}")
```

```
Initial weights:[ 0.00496714 -0.00138264], Initial bias:0.0
```

Figure 11: Weight initialization code and output.

Initialization:

$$W = [0.00496714,\ -0.00138264], \quad b = 0.0$$

Very large initial weights can cause extremely high activations, pushing the sigmoid outputs close to 0 or 1. This slows down learning due to the vanishing gradient problem. In contrast, very small weights (close to zero) keep the output $z = W \cdot X + b$ within the sensitive, non-saturated region of the sigmoid function, which ensures faster convergence.

If all weights were initialized to zero, every parameter would receive identical gradient updates, preventing the model from learning distinct features. To avoid this issue, random initialization is used so that each weight starts with a slightly different value. This diversity allows the model to learn useful feature representations.

Thus, initializing weights with small random values and setting the bias to zero provides a balance between stable training and efficient convergence.

3.3 Loss Function Selection

Since this is a binary classification task, the **Binary Cross-Entropy (Log Loss)** function was used. It is mathematically defined as:

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

where:

- m is the total number of training samples,
- $y_i \in \{0, 1\}$ is the true class label for the i^{th} sample,
- $\hat{y}_i = \sigma(W \cdot X_i + b)$ is the predicted probability for class 1 obtained using the sigmoid function.

Reason for Selection:
Binary cross-entropy was chosen because it is the standard and mathematically sound choice for binary classification tasks. It provides a probabilistic interpretation, numerical stability, and efficient convergence when combined with the sigmoid activation function.

This loss function directly measures the difference between the predicted probabilities and the true binary labels, making it ideal for logistic regression. It penalizes confident but incorrect predictions more severely than less confident predictions, which leads to better-calibrated probability estimates.

Although Mean Squared Error (MSE) could also be applied, it does not align well with the probabilistic nature of classification tasks. In contrast, binary cross-entropy provides stronger gradients when predictions deviate significantly from the target, thereby accelerating the learning process.

3.4 Implement Newton's Method

We implement **logistic regression** using **Newton's method** to update the weights for the given dataset over 20 iterations. Newton's method is a second-order optimization technique that uses both the gradient and the curvature (Hessian) of the loss function to compute updates:

$$\theta^{(t+1)} = \theta^{(t)} - H^{-1}\nabla L(\theta^{(t)})$$

where $\theta = [W, b]$ are the model parameters, ∇L is the gradient of the loss function, and H is the Hessian matrix (matrix of second derivatives of the loss).

For logistic regression:

- **Prediction:**
$$\hat{y} = \sigma(XW + b)$$
- **Gradient:**
$$\nabla L = X^T(\hat{y} - y)$$
- **Hessian:**
$$H = X^T R X, \quad R = \text{diag}(\hat{y}_i(1 - \hat{y}_i))$$

Here, R is a diagonal matrix containing $\hat{y}_i(1-\hat{y}_i)$ for each sample. The initial parameters are set to zeros, $\theta = 0$.

Unlike batch gradient descent, Newton's method computes the exact step size using the inverse Hessian, allowing it to converge much faster. The curvature information helps the method escape symmetry and avoid small gradient issues.

The dataset is generated using `make_blobs` with two distinct clusters, and the weights are updated iteratively for 20 iterations. The loss decreases rapidly due to the second-order updates.

The final weights and bias after 20 iterations are:

$$\theta = [b, W_1, W_2] = [-2.93259517, 10.56207842, 4.32766252]$$

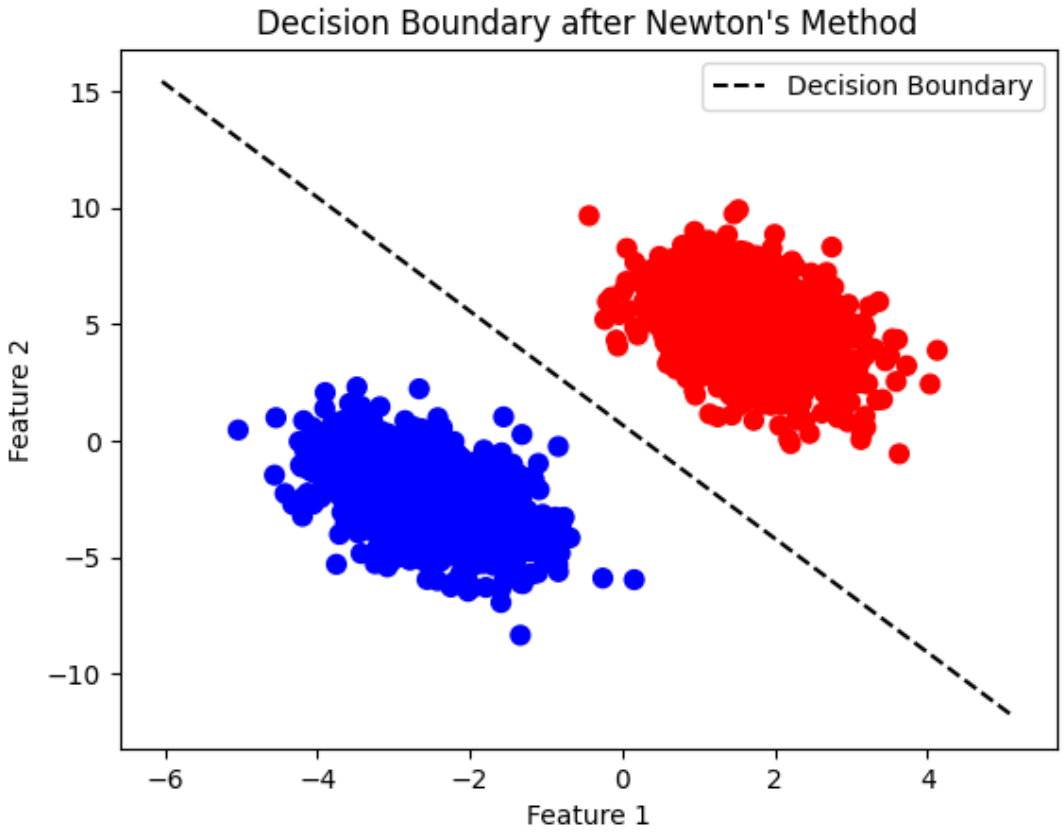

Figure 12: Scatter plot of the dataset with decision boundary obtained after Newton's method.

Decision Boundary Equation: From the final weights, the decision boundary can be expressed as:

$$w_1 x_1 + w_2 x_2 + b = 0 \quad \Rightarrow \quad x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

For example, with the computed weights:

$$x_2 = -0.8920 \cdot x_1 + 0.1039$$

After 20 iterations, the weights stabilize and the loss approaches a minimum quickly, demonstrating the fast convergence of Newton's method compared to gradient descent.

3.5 Loss Comparison: Batch Gradient Descent vs Newton's Method

We implemented **Batch Gradient Descent (BGD)** and **Newton's Method** for logistic regression on the given dataset and plotted the loss (binary cross-entropy) versus the number of iterations for both algorithms.

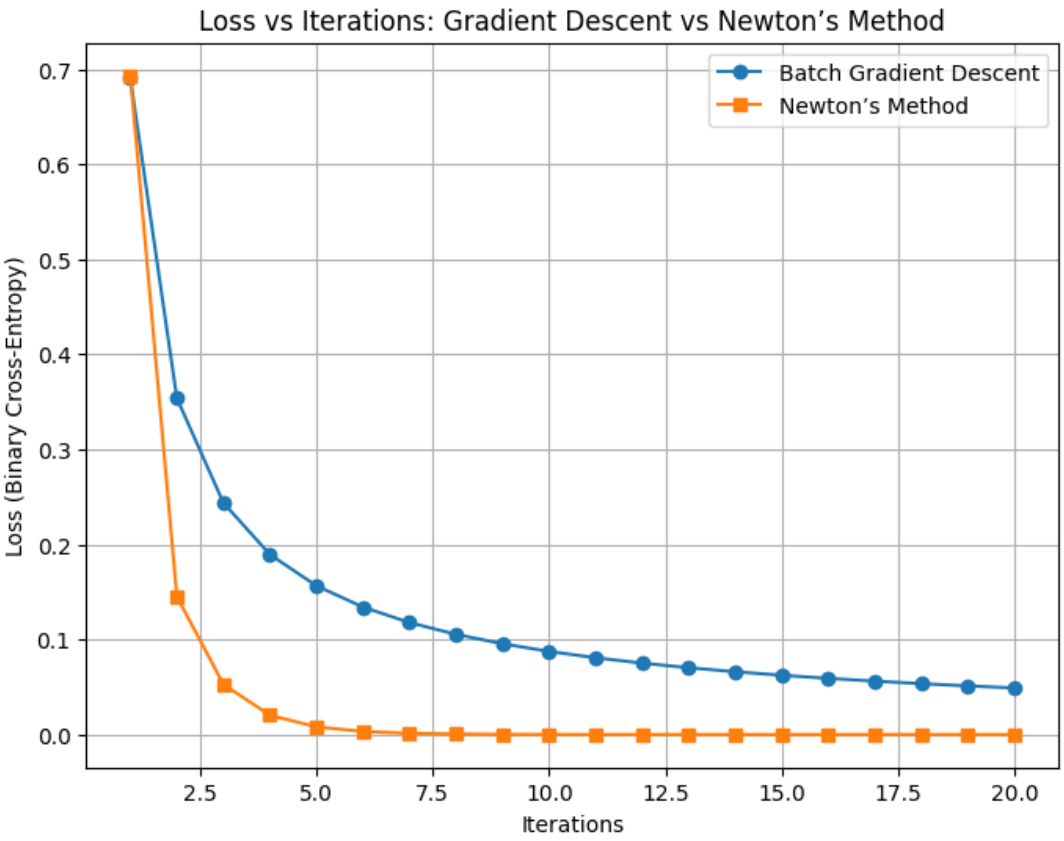

Figure 13: Loss vs Iterations: Batch Gradient Descent vs Newton's Method.

Observations:
Newton's Method converges significantly faster than Gradient Descent. Within just a few iterations (2–5), the loss reaches its minimum and stabilizes, whereas BGD reduces the loss more slowly and requires more iterations to approach a similar minimum.

This difference arises because Newton's Method uses second-order derivative information (the Hessian) to adjust the step size optimally at each iteration, effectively capturing the curvature of the loss function. This allows the algorithm to make more precise steps toward the minimum. In contrast, Gradient Descent relies solely on first-order derivatives and a fixed learning rate, resulting in slower convergence.

However, Newton's Method requires computing and inverting the Hessian matrix, which can be computationally expensive for high-dimensional datasets. Gradient Descent, while slower, is more computationally efficient and scales better to very large datasets, making it a preferred choice for deep learning applications.

Conclusion:
The experiment demonstrates that Newton's Method achieves faster convergence than Batch Gradient Descent by leveraging curvature information, reaching the loss minimum in far fewer iterations. Nevertheless, this speed comes with higher per-iteration computational cost, making Newton's Method suitable for small to medium-sized problems, while Gradient Descent remains more scalable for large datasets and deep neural networks.

3.6 Approaches to Decide Number of Iterations

When training models with optimization algorithms such as **Batch Gradient Descent (BGD)** and **Newton's Method**, deciding the optimal number of iterations is crucial for efficient training and avoiding unnecessary computation. Two practical approaches are commonly used:

1. Convergence-Based Stopping Criterion:

Instead of pre-fixing an arbitrary number of iterations, monitor convergence dynamically. This can be done in two ways:

- **Gradient Norm Monitoring:** Stop iterations when the norm of the gradient becomes very small:
$$\|\nabla L(\theta)\| < \epsilon$$
where ϵ is a small threshold (e.g., 10^{-5}). This indicates that further updates will result in negligible changes in loss.
- **Loss Improvement Threshold:** Stop training when the absolute improvement in loss between iterations is less than a small tolerance:
$$|L^{(t+1)} - L^{(t)}| < \delta$$
where δ is a small constant (e.g., 10^{-6}). This prevents unnecessary iterations once convergence is reached.

This approach works well for both BGD and Newton's Method. It is especially useful for Newton's Method because it converges quadratically, and the loss typically stabilizes after just a few iterations.

2. Validation-Based Early Stopping:

Another approach is to split the data into training and validation sets and monitor the validation loss:

- **Early Stopping:** Stop training when the validation loss stops decreasing for a fixed number of iterations, which also prevents overfitting.
- **Iteration Tuning via Cross-Validation:** Use k-fold cross-validation to determine the iteration count that gives the best generalization performance.

This approach is essential for Gradient Descent since it typically requires many iterations. While Newton's Method generally needs very few iterations, validation-based checks can still help avoid unnecessary computation.

Summary:

Use gradient norm or loss improvement thresholds as a primary stopping rule to avoid hardcoding iteration counts. Combine this with early stopping on a validation set to prevent overfitting and identify the iteration number that generalizes well. Newton's Method typically requires very few iterations due to its second-order optimization, whereas Gradient Descent may need careful tuning of the learning rate and more iterations.

3.7 Batch Gradient Descent with Updated Centers

With the new cluster centers $[[2, 2], [5, 1.5]]$, the two classes become closer and more overlapping compared to the previous configuration. Figure 14 shows the generated dataset, while Figure 15 illustrates the decision boundary obtained from logistic regression using batch gradient descent.

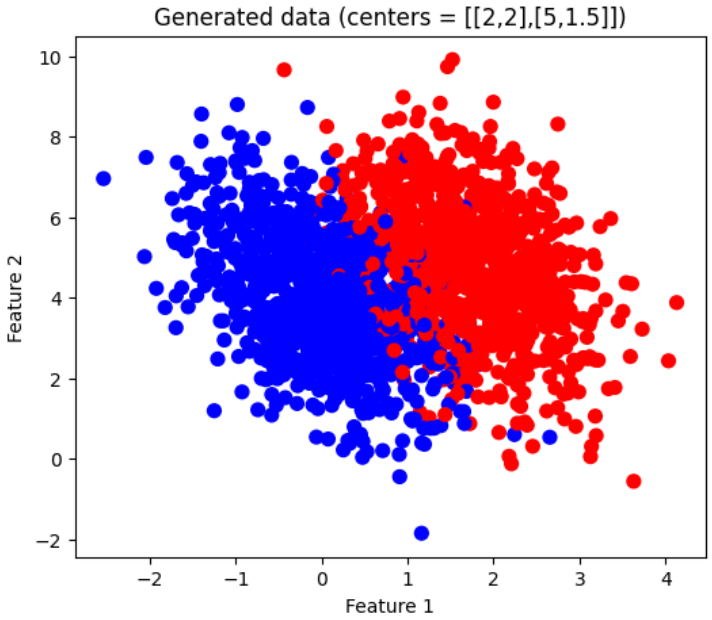

Figure 14: Generated dataset with centers $[2, 2]$ and $[5, 1.5]$.

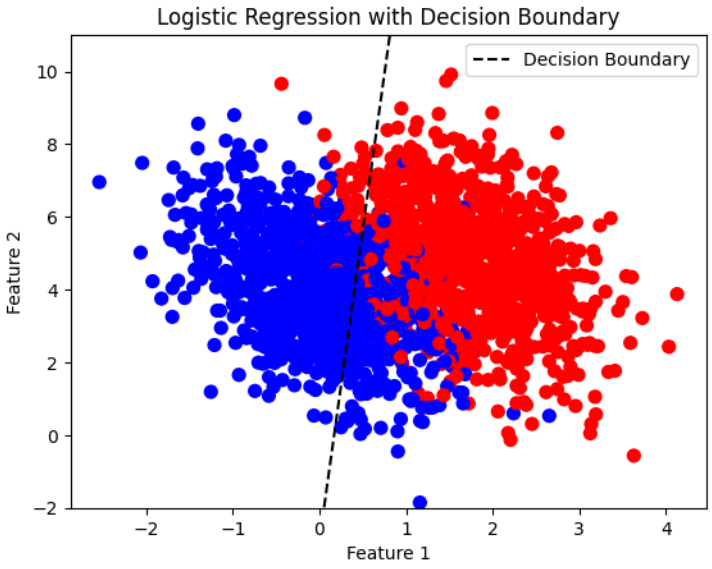

Figure 15: Decision boundary for logistic regression trained with batch gradient descent.

The corresponding loss curves for the previous dataset and the new overlapping dataset are compared in Figure 16. For the well-separated dataset, loss rapidly decreases to a very small value, while for the overlapping dataset, the loss decreases much more slowly and plateaus at a higher value.

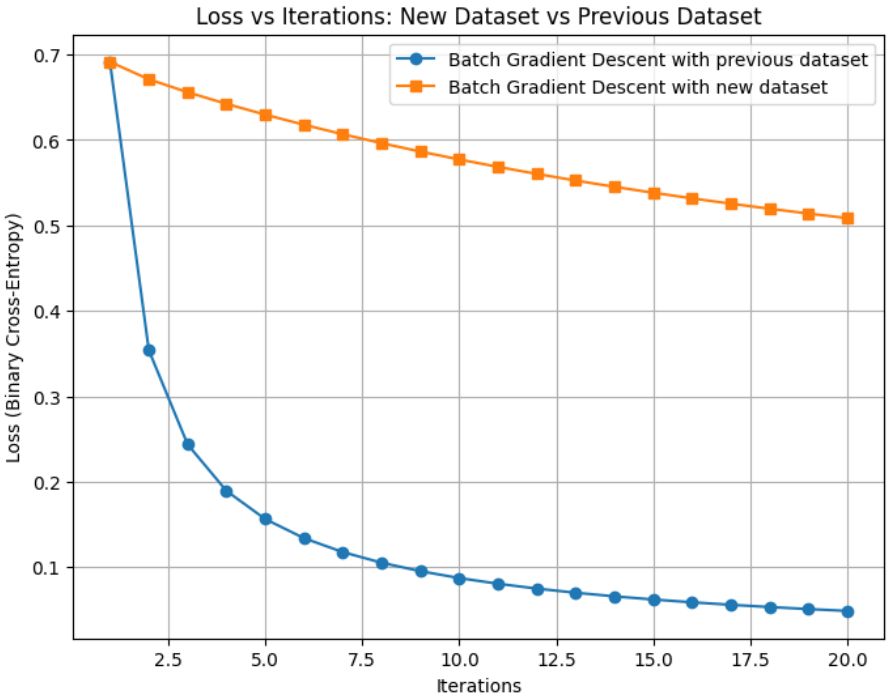

Figure 16: Loss vs iterations for batch gradient descent on previous vs new dataset.

Analysis of Convergence Behavior:

- For linearly separable data (previous dataset), logistic regression can drive the loss close to zero by increasing the magnitude of the weights. Batch gradient descent makes fast progress and achieves low loss within a few iterations.
- When the clusters are closer, as in the new dataset, many points lie near the decision boundary. The optimal linear classifier cannot perfectly separate them, so the minimum logistic loss is strictly greater than zero. The gradients near the optimum are smaller and less consistent, leading to slower convergence and higher final loss.
- Overlap between classes increases effective label noise from the model's perspective, flattening the loss landscape and yielding smaller step sizes per update.
- With an appropriate learning rate, the loss still decreases monotonically but at a reduced rate. Too large a learning rate could cause oscillations or divergence, while too small a value would lead to extremely slow convergence.

Conclusion: Batch gradient descent converges more slowly and stabilizes at a higher loss value for overlapping datasets. This behavior arises because the classes are not perfectly separable, yielding smaller gradients near the optimum.

4 References

1. Sklearn Lasso: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html
2. Sklearn logistic regression: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

5 Appendix

Colab Notebook: https://colab.research.google.com/drive/1Rijrqk5$_O dcFdJRaMMcpV6Ts_9kl$
$vwOIShqw8Cpr$