

**Department of Electrical Engineering**

**University of Moratuwa**

**EN 3150 – Pattern Recognition**

**Learning from data and related challenges and classification**

Kularathna A. K. D. D. D. – 220332P

Date of Submission – 05/09/2025

This is submitted as a partial fulfilment for the module

EN 3150 Pattern Recognition

Department of Electronics and Telecommunication Engineering

University of Moratuwa

Table of Contents

[1. Linear Regression 3](#_Toc208078978)

[1.1. Why can the OLS line be “not aligned to the majority of data points”? 3](#_Toc208078979)

[1.2. Which weighting scheme gives a better line for inliers? Justify. 3](#_Toc208078980)

[1.3. Why is plain linear regression unsuitable for “which brain regions predict the task”? 6](#_Toc208078981)

[1.4. Next, the following two methods are being considered: 7](#_Toc208078982)

[1.5. Which method (LASSO or group LASSO) is more appropriate in this setting, and why? 7](#_Toc208078983)

[2. Logistic Regression 8](#_Toc208078984)

[2.1. Use the code given in listing 1 to load data. 8](#_Toc208078985)

[2.2. Now, use the code given in listing 2 to train a logistic regression model. Here, did you encounter any errors? If yes, what were they, and how would you go about resolving them? 8](#_Toc208078986)

[2.3. Why does the saga solver perform poorly? 11](#_Toc208078987)

[2.4. Now change the “saga” to “liblinear” and check the accuracy. 11](#_Toc208078988)

[2.5. Why does the "liblinear" solver perform better than "saga" solver? 13](#_Toc208078989)

[2.6. Explain why the model’s accuracy (with saga solver) varies with different random state values? 14](#_Toc208078990)

[2.7. Compare the performance of the "liblinear" and "saga" solvers with feature scaling. If there is a significant difference in the accuracy with and without feature scaling, what is the reason for that. You may use Standard Scaler available in sklearn library. 15](#_Toc208078991)

[2.8. Suppose you have a categorical feature with the categories ’red’, ’blue’, ’green’, ’blue’, ’green’. After encoding this feature using label encoding, you then apply a feature scaling method such as Standard Scaling or Min-Max Scaling. Is this approach correct? or not? What do you propose? 17](#_Toc208078992)

[3. Logistic regression First/Second-Order Methods 18](#_Toc208078993)

[3.1. Use the code given in listing 3 to generate data. Here, variable y and X are class labels and corresponding feature values, respectively. 18](#_Toc208078994)

[3.2. Implement batch Gradient descent to update the weights for the given dataset over20 iterations. State the method used to initialize the weights and reason for your selection. 19](#_Toc208078995)

[3.3. Specify the loss function you have used and state reason for your selection. 23](#_Toc208078996)

[3.4. Implement Newton’s method to update the weights for the given dataset over 20 iterations. 23](#_Toc208078997)

[3.5. Plot the loss with respect to number of iterations for batch Gradient descent and Newton methods in a single plot. Comment on your results. 26](#_Toc208078998)

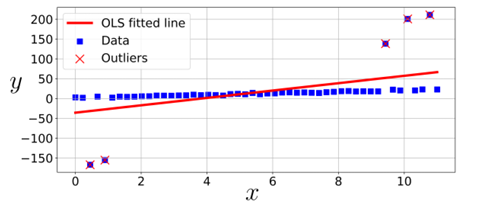
[3.6. Propose two approaches to decide number of iterations for Gradient descent and Newton’s method. 29](#_Toc208078999)

[3.7. Suppose the centers in listing 3 are changed to centers = [2, 2], [5, 1.5]]. Use batch Gradient descent to update the weights for this new configuration. Analyze the convergence behavior of the algorithm with this updated data and provide an explanation for convergence behavior. 31](#_Toc208079000)

[Reference 35](#_Toc208079001)

[Appendix 35](#_Toc208079002)

# Linear Regression

****

A black text on a white background

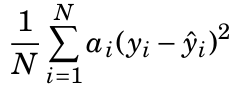
AI-generated content may be incorrect.**Ordinary Least Squares (OLS)**

## Why can the OLS line be “not aligned to the majority of data points”?

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

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

Let’s consider the modified loss function



Proposed scheme

* Scheme 1: For outliers ai = 0.01 and for inliers ai = 1
* Scheme 2: For outliers ai = 5 and for inliers ai = 1

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

Example

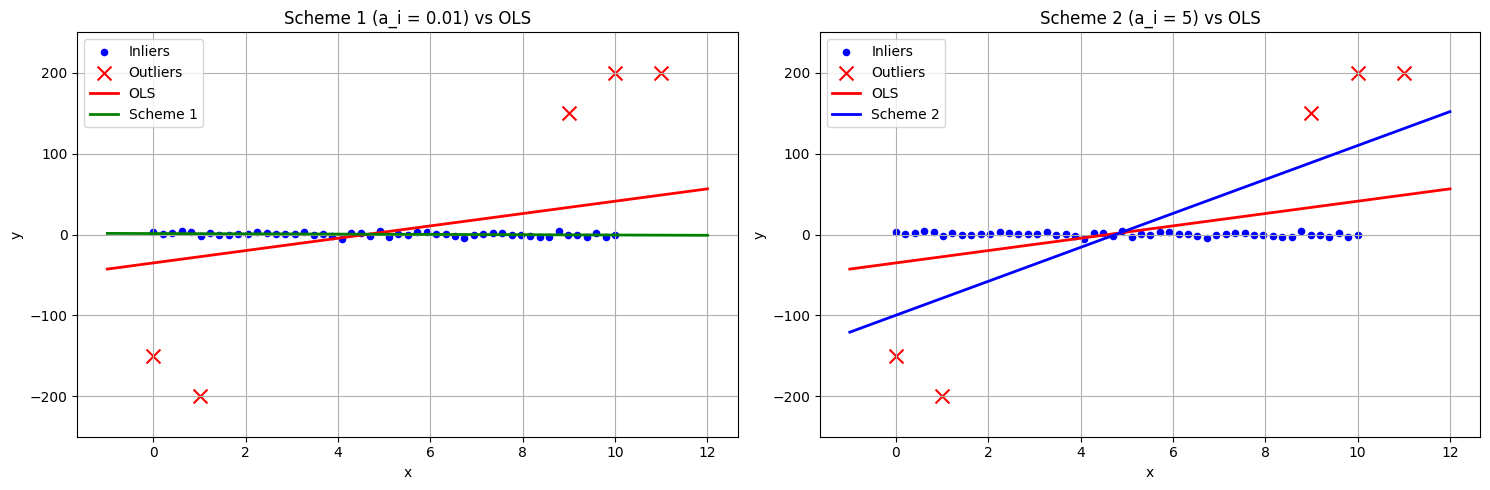


Figure 1.1

Code for the plots

import numpy as np

import matplotlib.pyplot as plt

# Generate inlier points

x\_in = np.linspace(0, 10, 50)  # Generate 50 points between 0 and 10

y\_in = np.zeros\_like(x\_in) + np.random.normal(0, 2, size=len(x\_in))  # Points around y=0

# Define outlier points

x\_out = np.array([0, 1, 9, 10, 11])

y\_out = np.array([-150, -200, 150, 200, 200])

# Combine inliers and outliers

X = np.concatenate([x\_in, x\_out])

Y = np.concatenate([y\_in, y\_out])

# Create design matrix

X\_design = np.vstack([np.ones\_like(X), X]).T

# Function to compute weighted least squares

def weighted\_least\_squares(X, y, weights):

    W = np.diag(weights)

    beta = np.linalg.inv(X.T @ W @ X) @ X.T @ W @ y

    return beta

# Create weights for both schemes

weights1 = np.ones(len(X))

weights1[-5:] = 0.01  # Last 5 points are outliers, scheme 1

weights2 = np.ones(len(X))

weights2[-5:] = 5     # Last 5 points are outliers, scheme 2

# Compute coefficients for OLS and both schemes

beta\_ols = np.linalg.inv(X\_design.T @ X\_design) @ X\_design.T @ Y

beta\_scheme1 = weighted\_least\_squares(X\_design, Y, weights1)

beta\_scheme2 = weighted\_least\_squares(X\_design, Y, weights2)

# Create points for plotting the lines

x\_plot = np.linspace(-1, 12, 100)

y\_ols = beta\_ols[0] + beta\_ols[1] \* x\_plot

y\_scheme1 = beta\_scheme1[0] + beta\_scheme1[1] \* x\_plot

y\_scheme2 = beta\_scheme2[0] + beta\_scheme2[1] \* x\_plot

# Create two subplots

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

# Plot for Scheme 1

ax1.scatter(x\_in, y\_in, c='blue', label='Inliers', s=20)

ax1.scatter(x\_out, y\_out, c='red', marker='x', s=100, label='Outliers')

ax1.plot(x\_plot, y\_ols, 'r-', label='OLS', linewidth=2)

ax1.plot(x\_plot, y\_scheme1, 'g-', label='Scheme 1', linewidth=2)

ax1.set\_ylim(-250, 250)

ax1.set\_xlabel('x')

ax1.set\_ylabel('y')

ax1.legend()

ax1.set\_title('Scheme 1 (a\_i = 0.01) vs OLS')

ax1.grid(True)

# Plot for Scheme 2

ax2.scatter(x\_in, y\_in, c='blue', label='Inliers', s=20)

ax2.scatter(x\_out, y\_out, c='red', marker='x', s=100, label='Outliers')

ax2.plot(x\_plot, y\_ols, 'r-', label='OLS', linewidth=2)

ax2.plot(x\_plot, y\_scheme2, 'b-', label='Scheme 2', linewidth=2)

ax2.set\_ylim(-250, 250)

ax2.set\_xlabel('x')

ax2.set\_ylabel('y')

ax2.legend()

ax2.set\_title('Scheme 2 (a\_i = 5) vs OLS')

ax2.grid(True)

plt.tight\_layout()

plt.show()

Code 1.1

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

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

## Why is plain linear regression unsuitable for “which brain regions predict the task”?

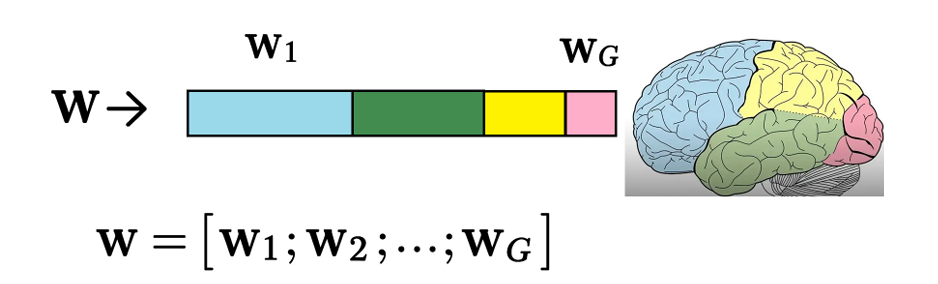


Figure 1.2

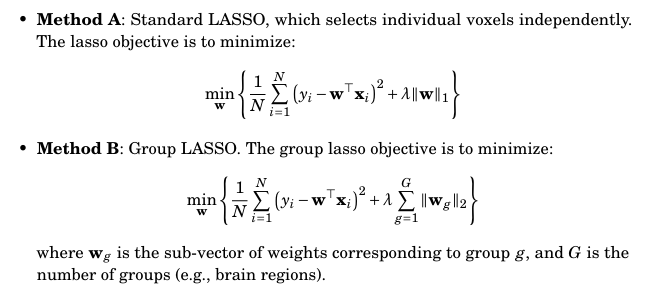
In brain image analysis (e.g., fMRI), the brain is divided into multiple regions as shown Figure 2, each consisting of many voxels (pixels). A researcher wants to identify which brain regions are most predictive of a specific cognitive task.

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

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

* It has too many features but too few samples
  + In brain studies, we might have many voxels but only few people in the experiment. Linera regression needs more data than features to work properly. In our case since we have more features than subjects, the model can overfit. It will not be generalized.
* Features are highly correlated
  + Voxels inside the same brain region behave similarly. When predictors are almost copies of each other, the regression struggles to predict which voxel gets how much weight. Small changes in the data can make the weights change so much. This makes the results unstable and unreliable.
* No automatic feature selection
  + Ordinary regression tries to use all voxels at once. It doesn’t know how to ignore the irrelevant voxels. Due to this it is hard to know which voxels are important for predicting the task. Due to this reason, we will just end up with bunch of coefficients, many of which may be just noise.
* Ignores natural grouping of voxels into regions
  + Linear regression looks at voxels one by one, but it doesn’t consider the group structure. Therefore, we might get scattered voxels selected across the brain, which might not make sense biologically.

## Next, the following two methods are being considered:



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

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

# Logistic Regression

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

Listing 1’s code gives following output

species class\_encoded

0 Adelie 0

1 Adelie 0

2 Adelie 0

4 Adelie 0

5 Adelie 0

.. ... ...

215 Chinstrap 1

216 Chinstrap 1

217 Chinstrap 1

218 Chinstrap 1

219 Chinstrap 1

[214 rows x 2 columns]

## Now, use the code given in listing 2 to train a logistic regression model. Here, did you encounter any errors? If yes, what were they, and how would you go about resolving them?

Yes, I encountered following error,

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

~\AppData\Local\Temp\ipykernel\_7948\348760815.py in ?()

3 #Split the data into training and testing sets

4 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

5 #Train the logistic regression model. Here we are using saga solver to learn weights.

6 logreg = LogisticRegression(solver='saga')

----> 7 logreg.fit(X\_train, y\_train)

8 # Predict on the testing data

9 y\_pred = logreg.predict(X\_test)

10 # Evaluate the model

c:\Users\User\miniconda3\envs\ptr\Lib\site-packages\sklearn\base.py in ?(estimator, \*args, \*\*kwargs)

1361 skip\_parameter\_validation=(

1362 prefer\_skip\_nested\_validation or global\_skip\_validation

1363 )

1364 ):

-> 1365 return fit\_method(estimator, \*args, \*\*kwargs)

c:\Users\User\miniconda3\envs\ptr\Lib\site-packages\sklearn\linear\_model\\_logistic.py in ?(self, X, y, sample\_weight)

1243 \_dtype = np.float64

1244 else:

1245 \_dtype = [np.float64, np.float32]

1246

-> 1247 X, y = validate\_data(

...

2169 else:

2170 arr = np.array(values, dtype=dtype, copy=copy)

2171

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

*Output is truncated. View as a* [*scrollable element*](command:cellOutput.enableScrolling?5879ea12-0c53-43aa-af07-2084b360e482) *or open in a* [*text editor*](command:workbench.action.openLargeOutput?5879ea12-0c53-43aa-af07-2084b360e482)*. Adjust cell output* [*settings*](command:workbench.action.openSettings?%5B%22%40tag%3AnotebookOutputLayout%22%5D)*...*

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

import seaborn as sns

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import OneHotEncoder

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# Load the penguins dataset

df = sns.load\_dataset("penguins")

df.dropna(inplace=True)

# Filter rows for 'Adelie' and 'Chinstrap' classes

selected\_classes = ['Adelie', 'Chinstrap']

df\_filtered = df[df['species'].isin(selected\_classes)].copy()

# Initialize the LabelEncoder for target variable

le = LabelEncoder()

y = le.fit\_transform(df\_filtered['species'])

# Separate numerical and categorical columns

numeric\_cols = ['bill\_length\_mm', 'bill\_depth\_mm', 'flipper\_length\_mm', 'body\_mass\_g']

categorical\_cols = ['sex', 'island']

# Create one-hot encoder for categorical variables

onehot = OneHotEncoder(sparse\_output=False, drop='first')  # Changed sparse to sparse\_output

categorical\_encoded = onehot.fit\_transform(df\_filtered[categorical\_cols])

categorical\_features = onehot.get\_feature\_names\_out(categorical\_cols)

# Combine numerical and one-hot encoded features

X\_numeric = df\_filtered[numeric\_cols].values

X\_combined = np.hstack([X\_numeric, categorical\_encoded])

# Create final feature matrix with labeled columns

feature\_names = list(numeric\_cols) + list(categorical\_features)

X = pd.DataFrame(X\_combined, columns=feature\_names)

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train the logistic regression model

logreg = LogisticRegression(solver='saga')

logreg.fit(X\_train, y\_train)

# Predict and evaluate

y\_pred = logreg.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

# Print results with feature names

print("Accuracy:", accuracy)

print("\nFeature coefficients:")

for feature, coef in zip(feature\_names, logreg.coef\_[0]):

    print(f"{feature}: {coef:.4f}")

print(f"Intercept: {logreg.intercept\_[0]:.4f}")

Code 2. 1

This code gives following output

Accuracy: 0.5813953488372093

Feature coefficients:

bill\_length\_mm: 0.0028

bill\_depth\_mm: -0.0001

flipper\_length\_mm: 0.0005

body\_mass\_g: -0.0003

sex\_Male: 0.0000

island\_Dream: 0.0002

island\_Torgersen: -0.0001

Intercept: -0.0000

[c:\Users\User\miniconda3\envs\ptr\Lib\site-packages\sklearn\linear\_model\\_sag.py:348](file:///C:\Users\User\miniconda3\envs\ptr\Lib\site-packages\sklearn\linear_model\_sag.py:348): ConvergenceWarning: The max\_iter was reached which means the coef\_ did not converge

warnings.warn(

## Why does the saga solver perform poorly?

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

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

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

import seaborn as sns

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import OneHotEncoder

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# Load the penguins dataset

df = sns.load\_dataset("penguins")

df.dropna(inplace=True)

# Filter rows for 'Adelie' and 'Chinstrap' classes

selected\_classes = ['Adelie', 'Chinstrap']

df\_filtered = df[df['species'].isin(selected\_classes)].copy()

# Initialize the LabelEncoder for target variable

le = LabelEncoder()

y = le.fit\_transform(df\_filtered['species'])

# Separate numerical and categorical columns

numeric\_cols = ['bill\_length\_mm', 'bill\_depth\_mm', 'flipper\_length\_mm', 'body\_mass\_g']

categorical\_cols = ['sex', 'island']

# Create one-hot encoder for categorical variables

onehot = OneHotEncoder(sparse\_output=False, drop='first')  # Changed sparse to sparse\_output

categorical\_encoded = onehot.fit\_transform(df\_filtered[categorical\_cols])

categorical\_features = onehot.get\_feature\_names\_out(categorical\_cols)

# Combine numerical and one-hot encoded features

X\_numeric = df\_filtered[numeric\_cols].values

X\_combined = np.hstack([X\_numeric, categorical\_encoded])

# Create final feature matrix with labeled columns

feature\_names = list(numeric\_cols) + list(categorical\_features)

X = pd.DataFrame(X\_combined, columns=feature\_names)

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Train the logistic regression model

logreg = LogisticRegression(solver='liblinear')  # Changed solver to liblinear

logreg.fit(X\_train, y\_train)

# Predict and evaluate

y\_pred = logreg.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

# Print results with feature names

print("Accuracy:", accuracy)

print("\nFeature coefficients:")

for feature, coef in zip(feature\_names, logreg.coef\_[0]):

    print(f"{feature}: {coef:.4f}")

print(f"Intercept: {logreg.intercept\_[0]:.4f}")

Code 2.2

The code gives following output

Accuracy: 1.0

Feature coefficients:

bill\_length\_mm: 1.5152

bill\_depth\_mm: -1.3916

flipper\_length\_mm: -0.1441

body\_mass\_g: -0.0037

sex\_Male: -0.2264

island\_Dream: 0.7346

island\_Torgersen: -0.5619

Intercept: -0.0774

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

Acording to the results

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

Reasons behind these observations

1. Dataset size
   * + "liblinear" is designed for small to medium dense datasets like penguins.
     + However, "saga" is designed and optimized for very large datasets or with support to sparse high-dimensional problems. For small dense data, "saga" is often slower.
2. Optimization methods
   * + "liblinear" uses a coordinate descent/ dual optimization method, which is deterministic and converges reliably for binary classification.
     + But "saga" solver uses a stochastic gradient-based slover, which means it updates parameters using random mini-batches. On small data, this introduces noise and slow convergence.
3. Feature scaling
   * + "saga" is sensitive to the unscaled features because its step size depending on the feature magnitudes. Without scaling, optimization is ill-conditioned.
     + However liblinear is much less sensitive to feature scaling.
4. Convergence
   * + In our situation "saga" hit maximum iteration limit 100 without converging
     + However, "liblinear" converged properly. This led to perfect separation of the two species.

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

1. Train/test split randomness

* "random\_state" is affecting how the dataset is split into training data and test data.
* With small dataset like penguins, changing the split can easily change which samples are in training vs test. Because the "saga" solver doesn’t converge well, even the small differences in the split lead to large changes in accuracy.

1. Stochastic optimization

* "saga" uses randomness in it for updating coefficients.
* Different random seeds lead to different optimization paths; this happens especially if the solver has not converged.
* Because it stops early, "saga" results depend more on that random path.

## Compare the performance of the "liblinear" and "saga" solvers with feature scaling. If there is a significant difference in the accuracy with and without feature scaling, what is the reason for that. You may use Standard Scaler available in sklearn library.

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

import seaborn as sns

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import OneHotEncoder, StandardScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

from sklearn.pipeline import Pipeline

# Load and preprocess data

df = sns.load\_dataset("penguins")

df.dropna(inplace=True)

# Filter classes

selected\_classes = ['Adelie', 'Chinstrap']

df\_filtered = df[df['species'].isin(selected\_classes)].copy()

# Encode target

le = LabelEncoder()

y = le.fit\_transform(df\_filtered['species'])

# Prepare features

numeric\_cols = ['bill\_length\_mm', 'bill\_depth\_mm', 'flipper\_length\_mm', 'body\_mass\_g']

categorical\_cols = ['sex', 'island']

# One-hot encode categorical

onehot = OneHotEncoder(sparse\_output=False, drop='first')

categorical\_encoded = onehot.fit\_transform(df\_filtered[categorical\_cols])

categorical\_features = onehot.get\_feature\_names\_out(categorical\_cols)

# Combine features

X\_numeric = df\_filtered[numeric\_cols].values

X\_combined = np.hstack([X\_numeric, categorical\_encoded])

feature\_names = list(numeric\_cols) + list(categorical\_features)

X = pd.DataFrame(X\_combined, columns=feature\_names)

# Split data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create the two pipelines we want to compare

pipelines = {

    'saga\_with\_scaling': Pipeline([

        ('scaler', StandardScaler()),

        ('classifier', LogisticRegression(solver='saga', max\_iter=1000))

    ]),

    'liblinear': Pipeline([

        ('classifier', LogisticRegression(solver='liblinear'))

    ])

}

# Train and evaluate models

results = {}

for name, pipeline in pipelines.items():

    pipeline.fit(X\_train, y\_train)

    y\_pred = pipeline.predict(X\_test)

    accuracy = accuracy\_score(y\_test, y\_pred)

    results[name] = accuracy

# Print results

print("Model Accuracies:")

for name, accuracy in results.items():

    print(f"{name}: {accuracy:.4f}")

# Print feature importances for scaled saga model

saga\_scaled = pipelines['saga\_with\_scaling']

coefficients = saga\_scaled.named\_steps['classifier'].coef\_[0]

scaled\_features = pd.DataFrame({'Feature': feature\_names, 'Coefficient': coefficients})

print("\nTop feature importances (SAGA with scaling):")

print(scaled\_features.sort\_values(by='Coefficient', key=abs, ascending=False))

Code 2.3

The code gives the following output

Model Accuracies:

saga\_with\_scaling: 1.0000

liblinear: 1.0000

Top feature importances (SAGA with scaling):

Feature Coefficient

0 bill\_length\_mm 3.343324

5 island\_Dream 1.115379

4 sex\_Male -0.920930

6 island\_Torgersen -0.518325

1 bill\_depth\_mm -0.465388

2 flipper\_length\_mm 0.439962

3 body\_mass\_g -0.356362

## Suppose you have a categorical feature with the categories ’red’, ’blue’, ’green’, ’blue’, ’green’. After encoding this feature using label encoding, you then apply a feature scaling method such as Standard Scaling or Min-Max Scaling. Is this approach correct? or not? What do you propose?

* This approach is incorrect.
* Here are the reasons why it is incorrect.

1. Artificial Ordering

* Since label encoder encode categories to numbers like red = 0, blue = 1, green = 2, The scaling of these values makes less sense because it assumes the distance between categories are meaningful.

1. Distance Problem

* After scaling, the distances between categories would be treated as meaningful
* For example, if red = 0 then blue = 1 and green = 2.
* Scaling would preserve these relative distances suggesting blue is between red and green.
  + - The proposed approach
      * + use **One-Hot Encoding**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sample** | **Color** | **Color\_Red** | **Color\_Blue** | **Color\_Green** |
| 1 | Red | 1 | 0 | 0 |
| 2 | Blue | 0 | 1 | 0 |
| 3 | Green | 0 | 0 | 1 |
| 4 | Blue | 0 | 1 | 0 |
| 5 | Green | 0 | 0 | 1 |

# Logistic regression First/Second-Order Methods

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

Code used for this question

# Listing 3: Data generation

import numpy as np

import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make\_blobs

# Generate synthetic data

np.random.seed(0)

centers = [[-5, 0], [5, 1.5]]

X, y = make\_blobs(n\_samples=2000, centers=centers, random\_state=5)

transformation = [[0.5, 0.5], [-0.5, 1.5]]

X = np.dot(X, transformation)

Code 3.1

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

Code implementation

import numpy as np

import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import make\_blobs

# Generate synthetic data

np.random.seed(0)

centers = [[-5, 0], [5, 1.5]]

X, y = make\_blobs(n\_samples=2000, centers=centers, random\_state=5)

transformation = [[0.5, 0.5], [-0.5, 1.5]]

X = np.dot(X, transformation)

# Add bias term to X

X\_b = np.c\_[np.ones((X.shape[0], 1)), X]

# Initialize weights to zeros

weights = np.zeros(X\_b.shape[1])  # 3 weights (bias, w1, w2)

def sigmoid(z):

    """Sigmoid activation function"""

    return 1 / (1 + np.exp(-z))

def compute\_loss(X, y, weights):

    """Binary cross-entropy loss"""

    m = len(y)

    h = sigmoid(np.dot(X, weights))

    loss = -1/m \* np.sum(y \* np.log(h + 1e-15) + (1-y) \* np.log(1-h + 1e-15))

    return loss

# Training parameters

learning\_rate = 0.01

n\_iterations = 20

losses = []

# Batch gradient descent

for i in range(n\_iterations):

    # Forward pass

    z = np.dot(X\_b, weights)

    predictions = sigmoid(z)

    # Compute gradients

    gradients = 1/len(y) \* np.dot(X\_b.T, (predictions - y))

    # Update weights

    weights = weights - learning\_rate \* gradients

    # Store loss

    loss = compute\_loss(X\_b, y, weights)

    losses.append(loss)

    if i % 5 == 0:

        print(f"Iteration {i}, Loss: {loss:.4f}")

# Visualize results

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

# Plot decision boundary

plt.subplot(121)

plt.scatter(X[y==0, 0], X[y==0, 1], label='Class 0')

plt.scatter(X[y==1, 0], X[y==1, 1], label='Class 1')

# Create mesh grid for decision boundary

x1\_min, x1\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx1, xx2 = np.meshgrid(np.linspace(x1\_min, x1\_max, 100),

                       np.linspace(x2\_min, x2\_max, 100))

# Make predictions

grid = np.c\_[xx1.ravel(), xx2.ravel()]

grid\_b = np.c\_[np.ones((grid.shape[0], 1)), grid]

Z = sigmoid(np.dot(grid\_b, weights))

Z = Z.reshape(xx1.shape)

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

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend()

plt.title('Decision Boundary')

# Plot loss history

plt.subplot(122)

plt.plot(range(n\_iterations), losses, marker='o')

plt.xlabel('Iteration')

plt.ylabel('Loss')

plt.title('Training Loss History')

plt.tight\_layout()

plt.show()

print("\nFinal weights:", weights)

Code 3.2

This code gives following output

Iteration 0, Loss: 0.6499

Iteration 5, Loss: 0.4885

Iteration 10, Loss: 0.3877

Iteration 15, Loss: 0.3207

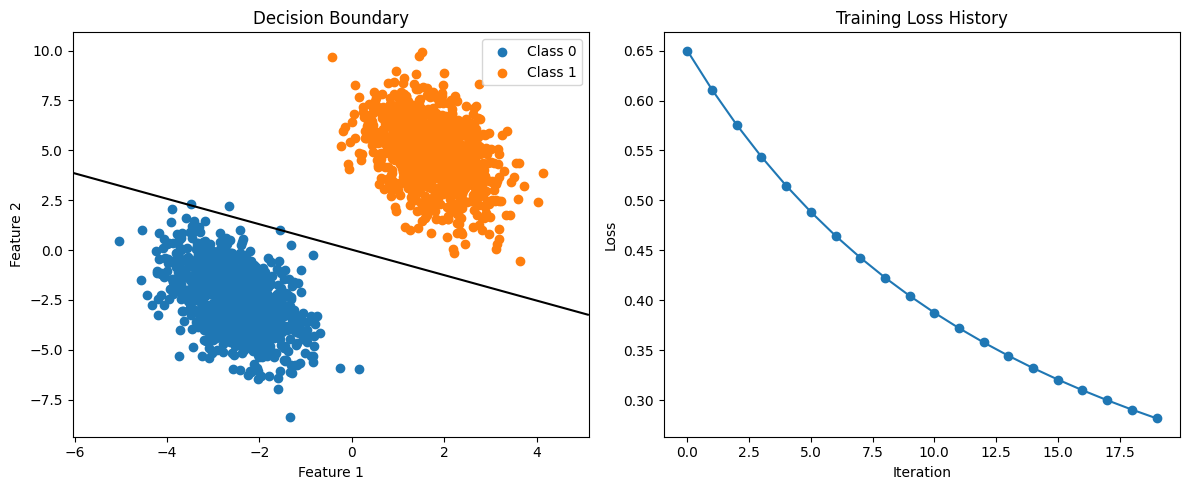


Figure 3.1

Final weights: [-0.0049357 0.15122723 0.23719972]

**Weight Initialization Method and Reasoning**

1. **Method Used**: Zero Initialization

  weights = np.zeros(X\_b.shape[1])  # 3 weights (bias, w1, w2)

1. **Reasons for Choosing Zero Initialization**:

* **Problem Type**: Binary classification with well-separated classes from “make\_blobs”
* **Convex Optimization**: Logistic regression is a convex problem, guaranteed to converge
* **Symmetry**: Equal starting point for both classes, no initial bias
* **Simplicity**: Deterministic initialization, reproducible results
* **Data Characteristics**: Classes are linearly separable, no need for complex initialization

1. **Why Other Methods Were Not Chosen**:

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

1. **Validation**:

* Loss curve shows proper convergence
* Decision boundary successfully separates classes
* No symmetry breaking issues encountered

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

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

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

## Implement Newton’s method to update the weights for the given dataset over 20 iterations.

Code for the implementation

import numpy as np

import matplotlib.pyplot as plt

# Generate synthetic data as before

np.random.seed(0)

centers = [[-5, 0], [5, 1.5]]

X, y = make\_blobs(n\_samples=2000, centers=centers, random\_state=5)

transformation = [[0.5, 0.5], [-0.5, 1.5]]

X = np.dot(X, transformation)

# Add bias term

X\_b = np.c\_[np.ones((X.shape[0], 1)), X]

# Initialize weights to zeros

weights = np.zeros(X\_b.shape[1])

def sigmoid(z):

    """Sigmoid activation function"""

    return 1 / (1 + np.exp(-z))

def compute\_loss(X, y, weights):

    """Binary cross-entropy loss"""

    m = len(y)

    h = sigmoid(np.dot(X, weights))

    return -1/m \* np.sum(y \* np.log(h + 1e-15) + (1-y) \* np.log(1-h + 1e-15))

# Training parameters

n\_iterations = 20

losses = []

# Newton's Method

for i in range(n\_iterations):

    # Compute predictions

    z = np.dot(X\_b, weights)

    h = sigmoid(z)

    # Compute gradient (first derivative)

    gradient = 1/len(y) \* np.dot(X\_b.T, (h - y))

    # Compute Hessian (second derivative)

    diagonal = h \* (1 - h)

    H = 1/len(y) \* (X\_b.T \* diagonal) @ X\_b

    # Update weights using Newton's method

    weights = weights - np.linalg.solve(H, gradient)

    # Store loss

    loss = compute\_loss(X\_b, y, weights)

    losses.append(loss)

    if i % 5 == 0:

        print(f"Iteration {i}, Loss: {loss:.4f}")

# Visualize results

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

# Plot decision boundary

plt.subplot(121)

plt.scatter(X[y==0, 0], X[y==0, 1], label='Class 0')

plt.scatter(X[y==1, 0], X[y==1, 1], label='Class 1')

# Create mesh grid for decision boundary

x1\_min, x1\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx1, xx2 = np.meshgrid(np.linspace(x1\_min, x1\_max, 100),

                       np.linspace(x2\_min, x2\_max, 100))

# Make predictions

grid = np.c\_[xx1.ravel(), xx2.ravel()]

grid\_b = np.c\_[np.ones((grid.shape[0], 1)), grid]

Z = sigmoid(np.dot(grid\_b, weights))

Z = Z.reshape(xx1.shape)

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

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend()

plt.title('Decision Boundary (Newton\'s Method)')

# Plot loss history

plt.subplot(122)

plt.plot(range(n\_iterations), losses, marker='o')

plt.xlabel('Iteration')

plt.ylabel('Loss')

plt.title('Training Loss History')

plt.tight\_layout()

plt.show()

print("\nFinal weights:", weights)

print("Final loss:", losses[-1])

Code 3.3

Output of the following code

Iteration 0, Loss: 0.1452

Iteration 5, Loss: 0.0013

Iteration 10, Loss: 0.0000

Iteration 15, Loss: 0.0000

A graph and diagram of a graph

AI-generated content may be incorrect.

Figure 3.2

Final weights: [-2.93259517 10.56207842 4.32766252]

Final loss: 2.870387964766699e-09

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

A graph with a blue line and red line

AI-generated content may be incorrect.

Figure 3.3

Code used for this plot

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_blobs

# Data generation and preprocessing

np.random.seed(0)

centers = [[-5, 0], [5, 1.5]]

X, y = make\_blobs(n\_samples=2000, centers=centers, random\_state=5)

transformation = [[0.5, 0.5], [-0.5, 1.5]]

X = np.dot(X, transformation)

X\_b = np.c\_[np.ones((X.shape[0], 1)), X]

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

def compute\_loss(X, y, weights):

    m = len(y)

    h = sigmoid(np.dot(X, weights))

    return -1/m \* np.sum(y \* np.log(h + 1e-15) + (1-y) \* np.log(1-h + 1e-15))

# Training parameters

n\_iterations = 20

learning\_rate = 0.01

# Batch Gradient Descent

weights\_gd = np.zeros(X\_b.shape[1])

losses\_gd = []

for i in range(n\_iterations):

    predictions = sigmoid(np.dot(X\_b, weights\_gd))

    gradients = 1/len(y) \* np.dot(X\_b.T, (predictions - y))

    weights\_gd = weights\_gd - learning\_rate \* gradients

    losses\_gd.append(compute\_loss(X\_b, y, weights\_gd))

# Newton's Method

weights\_newton = np.zeros(X\_b.shape[1])

losses\_newton = []

for i in range(n\_iterations):

    h = sigmoid(np.dot(X\_b, weights\_newton))

    gradient = 1/len(y) \* np.dot(X\_b.T, (h - y))

    diagonal = h \* (1 - h)

    H = 1/len(y) \* (X\_b.T \* diagonal) @ X\_b

    weights\_newton = weights\_newton - np.linalg.solve(H, gradient)

    losses\_newton.append(compute\_loss(X\_b, y, weights\_newton))

# Plotting

plt.figure(figsize=(10, 6))

plt.plot(range(n\_iterations), losses\_gd, 'b-', label='Gradient Descent', marker='o')

plt.plot(range(n\_iterations), losses\_newton, 'r-', label='Newton\'s Method', marker='o')

plt.xlabel('Iteration')

plt.ylabel('Loss')

plt.title('Loss Convergence: Gradient Descent vs Newton\'s Method')

plt.legend()

plt.grid(True)

plt.show()

print("Final loss (GD):", losses\_gd[-1])

print("Final loss (Newton):", losses\_newton[-1])

Code 3.4

Output of this code

Final loss (GD): 0.2817968058115373

Final loss (Newton): 2.870387964766699e-09

**Observed behavior**

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

**Reasons**

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

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

1. **Convergence-based Approach**

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

Code implementation

def train\_with\_convergence(X, y, method='gd', tolerance=1e-6, max\_iter=1000, learning\_rate=0.01):

    """

    Training with convergence-based stopping

    method: 'gd' for Gradient Descent or 'newton' for Newton's Method

    """

    weights = np.zeros(X.shape[1])

    prev\_loss = float('inf')

    losses = []

    for i in range(max\_iter):

        if method == 'gd':

            # Gradient Descent update

            h = sigmoid(np.dot(X, weights))

            gradients = 1/len(y) \* np.dot(X.T, (h - y))

            weights = weights - learning\_rate \* gradients

        else:

            # Newton's Method update

            h = sigmoid(np.dot(X, weights))

            gradient = 1/len(y) \* np.dot(X.T, (h - y))

            diagonal = h \* (1 - h)

            H = 1/len(y) \* (X.T \* diagonal) @ X

            weights = weights - np.linalg.solve(H, gradient)

        # Check convergence

        current\_loss = compute\_loss(X, y, weights)

        losses.append(current\_loss)

        if abs(prev\_loss - current\_loss) < tolerance:

            print(f"Converged after {i+1} iterations")

            break

        prev\_loss = current\_loss

    return weights, losses

Code 3.5

1. **Validation-based Approach**

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

Code implementation

def train\_with\_validation(X\_train, y\_train, X\_val, y\_val, method='gd', patience=5, max\_iter=1000, learning\_rate=0.01):

    """

    Training with validation-based early stopping

    """

    weights = np.zeros(X\_train.shape[1])

    best\_val\_loss = float('inf')

    patience\_counter = 0

    train\_losses = []

    val\_losses = []

    best\_weights = None

    for i in range(max\_iter):

        if method == 'gd':

            # Gradient Descent update

            h = sigmoid(np.dot(X\_train, weights))

            gradients = 1/len(y\_train) \* np.dot(X\_train.T, (h - y\_train))

            weights = weights - learning\_rate \* gradients

        else:

            # Newton's Method update

            h = sigmoid(np.dot(X\_train, weights))

            gradient = 1/len(y\_train) \* np.dot(X\_train.T, (h - y\_train))

            diagonal = h \* (1 - h)

            H = 1/len(y\_train) \* (X\_train.T \* diagonal) @ X\_train

            weights = weights - np.linalg.solve(H, gradient)

        # Compute validation loss

        val\_loss = compute\_loss(X\_val, y\_val, weights)

        val\_losses.append(val\_loss)

        if val\_loss < best\_val\_loss:

            best\_val\_loss = val\_loss

            best\_weights = weights.copy()

            patience\_counter = 0

        else:

            patience\_counter += 1

        if patience\_counter >= patience:

            print(f"Early stopping at iteration {i+1}")

            break

    return best\_weights, val\_losses

Code 3.6

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

Code after making changes

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_blobs

# Generate synthetic data with modified centers

np.random.seed(0)

centers = [[2, 2], [5, 1.5]]  # Modified centers

X, y = make\_blobs(n\_samples=2000, centers=centers, random\_state=5)

transformation = [[0.5, 0.5], [-0.5, 1.5]]

X = np.dot(X, transformation)

# Add bias term

X\_b = np.c\_[np.ones((X.shape[0], 1)), X]

# Initialize weights

weights = np.zeros(X\_b.shape[1])

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

def compute\_loss(X, y, weights):

    m = len(y)

    h = sigmoid(np.dot(X, weights))

    return -1/m \* np.sum(y \* np.log(h + 1e-15) + (1-y) \* np.log(1-h + 1e-15))

# Training parameters

learning\_rate = 0.01

n\_iterations = 50  # Increased iterations to observe convergence

losses = []

# Batch gradient descent

for i in range(n\_iterations):

    z = np.dot(X\_b, weights)

    predictions = sigmoid(z)

    gradients = 1/len(y) \* np.dot(X\_b.T, (predictions - y))

    weights = weights - learning\_rate \* gradients

    loss = compute\_loss(X\_b, y, weights)

    losses.append(loss)

    if i % 10 == 0:

        print(f"Iteration {i}, Loss: {loss:.4f}")

# Visualization

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

# Plot decision boundary

plt.subplot(121)

plt.scatter(X[y==0, 0], X[y==0, 1], label='Class 0')

plt.scatter(X[y==1, 0], X[y==1, 1], label='Class 1')

# Create mesh grid for decision boundary

x1\_min, x1\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx1, xx2 = np.meshgrid(np.linspace(x1\_min, x1\_max, 100),

                       np.linspace(x2\_min, x2\_max, 100))

grid = np.c\_[xx1.ravel(), xx2.ravel()]

grid\_b = np.c\_[np.ones((grid.shape[0], 1)), grid]

Z = sigmoid(np.dot(grid\_b, weights))

Z = Z.reshape(xx1.shape)

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

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend()

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

# Plot loss history

plt.subplot(122)

plt.plot(range(n\_iterations), losses, marker='o')

plt.xlabel('Iteration')

plt.ylabel('Loss')

plt.title('Training Loss History')

plt.grid(True)

plt.tight\_layout()

plt.show()

Code 3.7

Output of this code

Iteration 0, Loss: 0.6909

Iteration 10, Loss: 0.6721

Iteration 20, Loss: 0.6569

Iteration 30, Loss: 0.6432

Iteration 40, Loss: 0.6306

A graph and diagram with numbers

AI-generated content may be incorrect.

Figure 3.4

Analysis of Convergence Behavior:

1. **Slower Convergence**

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

1. **Higher Final Loss**

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

1. **Gradient Behavior**

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

1. **Stability Issues**

* More sensitive to initialization
* May require careful tuning of learning rate
* Could benefit from adaptive learning rates

1. **Recommendations**

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

# Reference

1. [***Lasso — scikit-learn 1.7.1 documentation***](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html)
2. [***LinearRegression — scikit-learn 1.7.1 documentation***](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)
3. [***LogisticRegression — scikit-learn 1.7.1 documentation***](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)
4. [***Cost Function and Gradient Descent | Sinhala***](https://www.youtube.com/watch?v=dGVdGJlhjCw&list=PL495mke12zYDHN9ONfcal1eQfo8VqmOgu&index=7)
5. [***Linear Regression | Supervised Learning Algorithm | Sinhala***](https://www.youtube.com/watch?v=yl1psqJf-3Q&list=PL495mke12zYDHN9ONfcal1eQfo8VqmOgu&index=5)
6. [***Logistic regression | Machine Learning Algorithm | Sinhala***](https://www.youtube.com/watch?v=WUeMWfim3GE&list=PL495mke12zYDHN9ONfcal1eQfo8VqmOgu&index=10)

# Appendix

**GitHub Repo link for the Assignment 02 -** [**Link**](https://github.com/DilshanKularathna/EN3150-Assignment02)