University of Moratuwa, Sri Lanka

Faculty of Engineering

Department of Electronics and Telecommunication Engineering
Semester 5 (Intake 2021)

EN3150 - Pattern Recognition

Assignment 02
Learning from data and related challenges and classification

Sehara G.M.M - 210583B

This report is submitted as a partial fulfillment for the module EN3150 - Pattern Recognition, Department of Electronic and Telecommunication Engineering, University of Moratuwa.

Contents

1 Logistic regression

Question 1

1.1 Loading the data

The given code in Listing 1 imports the `penguins` dataset, drops any rows with missing values, and filters the data to include only two species: 'Adelie' and 'Chinstrap'. It then uses `LabelEncoder` to encode the species into numerical values, creating a new column called `class_encoded` to store the encoded labels. After that, it prepares the dataset for machine learning by selecting the target variable (`class_encoded`) and dropping irrelevant or non-numeric columns like `species`, `island`, and `sex` from the feature set. This prepares the data for further modeling.

```
        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]
```

Figure 1: Output from the code in listing 1

1.2 What is the purpose of `y_encoded = le.fit_transform(df_filtered['species'])`?

The line `y_encoded = le.fit_transform(df_filtered['species'])` encodes the categorical values of the `'species'` column into numerical values using the `LabelEncoder`. Each species is assigned a unique integer, and the result is stored in `y_encoded`.

1.3 What is the purpose of `X = df.drop(['species', 'island', 'sex'], axis=1)`?

The line `X = df.drop(['species', 'island', 'sex'], axis=1)` removes the columns `'species'`, `'island'`, and `'sex'` from the DataFrame, keeping only the remaining columns in `X`.

1.4 Why we cannot use "island" and "sex" features?

The features "island" and "sex" are excluded because they are either non-numeric. Especially algorithms like Logistic Regression, typically require numeric input. Categorical features like "island" and "sex" would need to be encoded (e.g., using one-hot encoding) before they can be used in the model. Additionally, these features might not be relevant for predicting the target variable, so they are dropped to avoid adding noise to the model.

1.5 Training a Logistic Regression Model

This code in Listing 2 splits the dataset into training and testing sets, with 80% used for training and 20% for testing. It then trains a logistic regression model using the `saga` solver, which is optimized for large datasets. After fitting the model to the training data, it predicts the target values for the testing set. Finally, it evaluates the model's accuracy and prints both the accuracy score and the model's learned coefficients and intercept.

```
Accuracy: 0.5813953488372093
[[ 2.76807632e-03 -8.12284854e-05  4.60571732e-04 -2.86524099e-04]] [-8.48516252e-06]
```

Figure 2: Result of the code in listing 2

1.6 What is the usage of `random_state=42`?

The `random_state=42` in `train_test_split` is used to ensure reproducibility of the results. It sets a seed for the random number generator, so that we'll get the same sequence of random numbers each time we run the code. This can be useful for debugging, sharing results, and comparing different models. The specific value of 42 is arbitrary, but it's commonly used in examples. Without setting `random_state`, the split would vary with each run.

1.7 Why is accuracy low? why does the saga solver perform poorly?

The accuracy of 0.581 could be low for several reasons:

- **Imbalanced Classes:** The dataset has an unequal distribution of the two species.(152 in Adelie class and 68 in Chinstrap class) This could impact the accuracy. The model may be biased toward the majority class, leading to lower accuracy for the minority class.
- **Hyperparameter Settings:** The default settings of the saga solver might not be optimal for my dataset. Tuning hyperparameters, such as the regularization strength, could help improve the model's performance.
- **Feature Selection:** We have dropped the island and sex columns when creating the feature set. These features might provide important information that helps distinguish between the 'Adelie' and 'Chinstrap' species. Including these features could enhance the model's performance.
- **Data Scaling:** The features we used have different scales, which can affect the model's performance. Logistic regression, including the saga solver, typically works better when the data is scaled. Implementing feature scaling (like using StandardScaler) might improve the accuracy.

1.8 Changing the solver to "liblinear"

When the solver is changed to liblinear, the model outputs a 1.00 accuracy.

```python
#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 liblinear to learn weights.
logreg = LogisticRegression(solver='liblinear')
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_)
```

✓ 0.0s

```
Accuracy: 1.0
[[ 1.59665154 -1.42501103 -0.15238046 -0.003951  ]] [-0.0755452]
```

Figure 3: When liblinear is used

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

The `liblinear` solver may perform better than the `saga` solver in this case for several reasons:

- **Small Dataset:** If the dataset is relatively small, `liblinear` is often more efficient and can find a good solution quickly. It is specifically designed for smaller datasets and performs well with binary classification problems.
- **Regularization:** The `liblinear` solver uses coordinate descent to optimize the loss function, which can be effective for certain data distributions. It handles L1 and L2 regularization well, which might fit the characteristics of the data better.
- **Convergence:** `liblinear` might converge more reliably for the specific problem at hand, especially if the feature space is well-behaved. In contrast, `saga` is generally better for larger datasets but may require more iterations to converge.

In summary, for small datasets or specific data distributions, `liblinear` can outperform `saga`, particularly in terms of speed and convergence.

1.10 Performance after feature scaling using Standard Scalar

1.10.1 For ”saga” solver

After feature scaling using the Standard Scalar, the model accuracy has been increased from 0.5814 to 0.8837 when the ”saga” solver is used.

```
#From sklearn import standard scaler
from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler
scaler.fit(X_train)

# Transform the features
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train the logistic regression model
logreg = LogisticRegression(solver='saga')
logreg.fit(X_train, y_train)

# Predict on the testing data
y_pred = logreg.predict(X_test_scaled)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print(logreg.coef_, logreg.intercept_)

✓ 0.0s

Accuracy: 0.8837209302325582
[[ 2.76370504e-03 -8.22223618e-05  4.79333675e-04 -2.87473810e-04]] [-8.43945427e-06]
```

Figure 4: Code and output for "saga" solver

1.10.2 For "liblinear" solver

After feature scaling using the Standard Scalar, the model accuracy remained at 1.000 when the "liblinear" solver is used.

```
# Train the logistic regression model. Here we are using liblinear to learn weights.
logreg = LogisticRegression(solver='liblinear')
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_)
✓ 0.0s

Accuracy: 1.0
[[ 1.59665154 -1.42501103 -0.15238046 -0.003951  ]] [-0.0755452]
```

Figure 5: Code and output for "liblinear" solver

1.11 Problem of the given code and how to solve this?

The code snippet contains the line `X = df_filtered.drop(['species', 'class_encoded'], axis=1)`. However, this only removes the 'species' and 'class_encoded' columns, leaving the 'sex' and 'island' columns, which contain categorical values. This could potentially cause an error. To fix this, either drop the 'sex' and 'island' columns as well using `X = df_filtered.drop(['sex', 'island', 'species', 'class_encoded'], axis=1)`, or encode these categorical variables. Nevertheless, as

we discussed earlier, these two columns do not have a significant impact on the classification, so it is preferable to drop them.

```
Accuracy: 0.8604651162790697
[[ 0.15634391 -0.01051094 -0.02007102 -0.00092967  0.01187741 -0.00686766
   0.00071824]] [-0.00086854]
```

Figure 6: Output after running listing 3 code

1.12 Applying a feature scaling method after encoding

Using label encoding for categorical features like 'red', 'blue', and 'green' can cause issues, especially when followed by feature scaling methods such as Standard or Min-Max Scaling.

Issues with Label Encoding

- **Arbitrary Numeric Representation:** Label encoding assigns arbitrary numbers to categories (e.g., 'red' = 0, 'blue' = 1, 'green' = 2), which can mislead machine learning models into interpreting these values as having a numerical order, even though they don't.
- **Impact of Scaling:** After label encoding, scaling methods treat encoded values as continuous, which can distort data interpretation.

Recommended Approaches

- **One-Hot Encoding:** For nominal features like colors, one-hot encoding is preferred. It creates binary columns for each category, avoiding any implied ordinal relationship.

Row	Red	Blue	Green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0
5	0	0	1

Table 1: One-Hot Encoded Table

- **Avoid Scaling Post-Label Encoding:** When label encoding is necessary, avoid scaling directly after. Alternatively, use models that handle categorical data natively or apply one-hot encoding before scaling.
- **Ordinal Encoding:** If a categorical feature has a meaningful order (e.g., 'low', 'medium', 'high'), ordinal encoding should be used to preserve that order.

Question 2

The dataset contains the following variables:

- x_1: Number of hours studied.
- x_2: Undergraduate GPA.
- y: Whether the student received an A+ in the class.

The estimated logistic regression coefficients are:

$$w_0 = -5.9, \quad w_1 = 0.06, \quad w_2 = 1.5$$

1. Estimated Probability Calculation

We use the logistic regression model to estimate the probability $P(y = 1)$, which is defined as:

$$P(y = 1) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$

Given $x_1 = 50$ (hours studied) and $x_2 = 3.6$ (GPA), substitute the values into the model:

$$P(y = 1) = \frac{1}{1 + e^{-(-5.9 + 0.06(50) + 1.5(3.6))}}$$

First, calculate the linear combination:

$$\begin{aligned} \text{Linear combination} &= -5.9 + 0.06(50) + 1.5(3.6) \\ &= -5.9 + 3 + 5.4 = 2.5 \end{aligned}$$

Now, substitute this into the logistic function:

$$P(y = 1) = \frac{1}{1 + e^{-2.5}}$$

Approximating $e^{-2.5} \approx 0.0821$, we calculate the probability:

$$P(y = 1) = \frac{1}{1 + 0.0821} \approx \frac{1}{1.0821} \approx 0.923$$

Thus, the estimated probability that the student will receive an A+ is approximately 92.3%.

2. Required Study Hours for 60% Probability

To achieve a 60% probability of receiving an A+ ($P(y = 1) = 0.6$), we solve for the number of study hours x_1.

Starting with the logistic regression equation:

$$P(y = 1) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}} = 0.6$$

Rearranging the equation:

$$\frac{1}{0.6} - 1 = e^{-(w_0 + w_1 x_1 + w_2 x_2)}$$

Calculating gives:

$$\frac{1}{0.6} = 1.6667 \quad \Rightarrow \quad 1.6667 - 1 = 0.6667$$

Taking the natural logarithm:

$$\begin{aligned} \ln(0.6667) &= -(w_0 + w_1 x_1 + w_2 x_2) \\ \ln(0.6667) &\approx -0.4055 \end{aligned}$$

Substituting $w_0 = -5.9$, $w_2 = 1.5$, and $x_2 = 3.6$:

$$\begin{aligned} -0.4055 &= -(-5.9 + 0.06x_1 + 1.5(3.6)) \\ -0.4055 &= -(-0.5 + 0.06x_1) \end{aligned}$$

Multiplying both sides by -1 and solving for x_1:

$$\begin{aligned} 0.4055 &= -0.5 + 0.06x_1 \\ 0.4055 + 0.5 &= 0.06x_1 \\ 0.9055 = 0.06x_1 \quad &\Rightarrow \quad x_1 = \frac{0.9055}{0.06} \approx 15.1 \end{aligned}$$

Thus, the student needs to study approximately 15.1 hours to achieve a 60% probability of receiving an A+.

2 Logistic regression on real world data

2.1 Choose a data set from UCI Machine Learning Repository

I have selected the breast cancer dataset.

Listing 1: Loading breast cancer dataset

```
import pandas as pd
import numpy as np

from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
```

2.2 Correlation matrix and Pair plots

Listing 2: Plotting Correlation Matrix Using Seaborn and Matplotlib

```
import matplotlib.pyplot as plt
import seaborn as sns

# Define the selected features
selected_features = ['mean radius', 'mean texture', 'fractal dimension
   error', 'mean area', 'worst fractal dimension']

corr_matrix = df[selected_features].corr()

#Plot the correlation matrix using Seaborns heatmap
plt.figure(figsize=(10, 8))  # Adjust the figure size as needed
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)

# Add title
plt.title("Correlation Matrix of Selected Features", fontsize=16)

# Show the heatmap
plt.show()
```

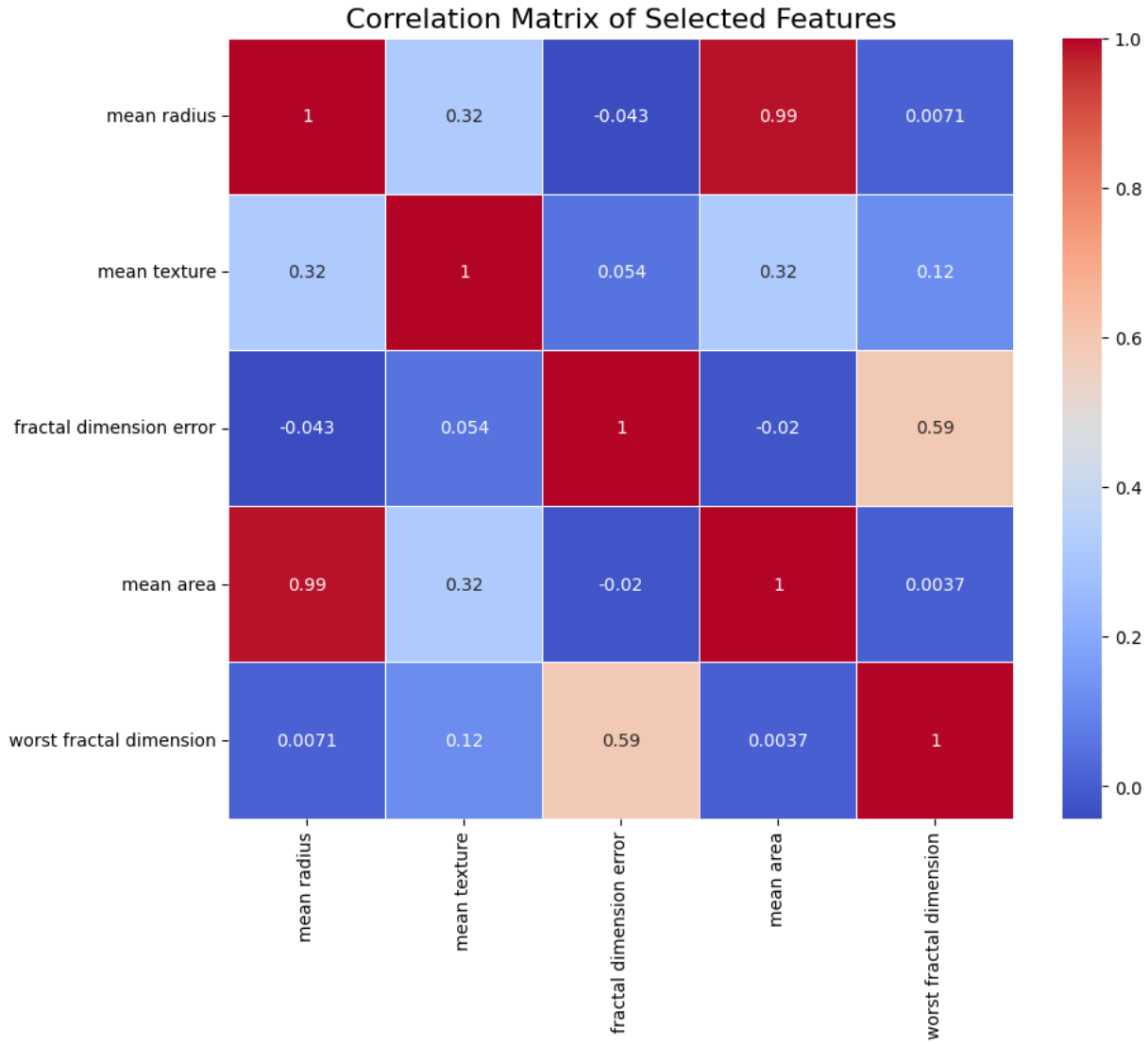

Figure 7: Correlation Matrix

Listing 3: Pair Plot

```
# Create a subset of the DataFrame that includes the selected features + target
pairplot_data = df[selected_features + ['target']]

# Generate pair plots using seaborn (with the subset of features and target)
sns.pairplot(pairplot_data, hue='target', diag_kind='kde')

# Show the pair plot
plt.show()
```

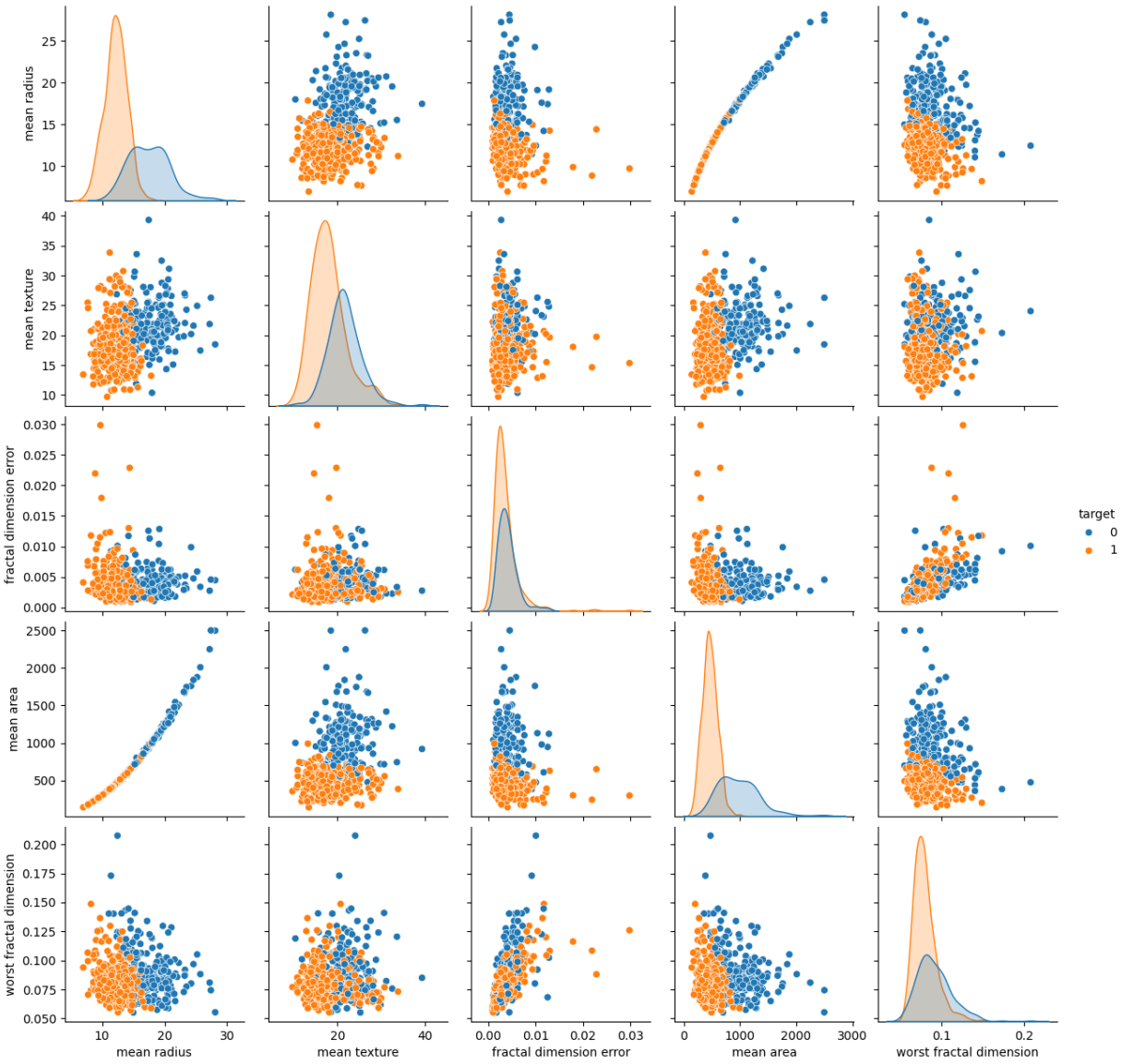

Figure 8: Pair Plots

2.2.1 Observing Results

Class Distribution Differences (Target 0 vs 1)

From the pair plot, several key differences between the target classes are noted:

- **Mean radius** and **mean area** exhibit clear class separation. Target 1 (orange) tends to have higher values than target 0 (blue).
- **Fractal dimension error** shows minimal separation between classes, indicating limited relevance for classification.
- **Mean texture** and **worst fractal dimension** display moderate separation, though not as distinct as mean radius and area.

Feature Relationships

The pair plot and correlation matrix reveal important feature relationships:

- **Mean radius** and **mean area** are highly correlated ($r = 0.99$), suggesting redundancy.
- A moderate positive correlation ($r = 0.59$) exists between **fractal dimension error** and **worst fractal dimension**.
- **Mean radius** and **fractal dimension error** have no significant relationship ($r = -0.043$).

Key Insights

- **Key Features**: **Mean radius** and **mean area** are strong indicators for distinguishing classes and should be prioritized in feature selection.
- **Redundancy**: Due to high correlation, either mean radius or mean area could be excluded to simplify the model.
- **Less Predictive Features**: **Fractal dimension error** offers little predictive value and can be deprioritized.
- **Moderately Correlated Features**: **Mean texture** and **worst fractal dimension** provide moderate class separation and may still contribute value.

2.3 Fit a Linear Regression Model

Listing 4: Logistic Regression Model

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report

X = df[selected_features] # Features
y = df['target'] # Target variable

X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.2,
    random_state=42)

#Fit the Logistic Regression model
model = LogisticRegression() # Increase max_iter if needed
model.fit(X_train, y_train)

#Make predictions
y_pred = model.predict(X_test)

#Evaluate the m o d e l s performance

accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(class_report)
```

```
Accuracy: 0.91
Confusion Matrix:
[[37  6]
 [ 4 67]]
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.86      0.88        43
           1       0.92      0.94      0.93        71

    accuracy                           0.91       114
   macro avg       0.91      0.90      0.91       114
weighted avg       0.91      0.91      0.91       114
```

Figure 9: Model Parameters

The model achieved an accuracy of 0.91, correctly predicting the target class 91% of the time. The confusion matrix indicates 37 true positives for class 0 and 67 for class 1, with 6 false positives for class 0 and 4 false negatives for class 1.

In the classification report, precision for class 0 is 0.90, meaning 90% of predicted class 0 instances were accurate, while recall is 0.86, indicating 86% of actual class 0 instances were identified. For class 1, precision is 0.92 and recall is 0.94, reflecting strong performance in predicting and capturing this class. Overall, the model demonstrates effective differentiation between the two classes.

2.4 Using statsmodels.Logit

Listing 5: Logistic Regression Model

```python
# Add a constant to the model (intercept)
X = sm.add_constant(X)

# Fit the Logistic Regression model using statsmodels
try:

    model = sm.Logit(y, X)
    result = model.fit(maxiter=1000)
    # Increase max_iter if needed
    # Obtain and interpret the summary
    print(result.summary())
    # Check p-values
    p_values = result.pvalues

    print("\nP-values for each predictor:")
    print(p_values)

    # Determine which features can be discarded (e.g., p > 0.05)
    discarded_features = p_values[p_values > 0.05].index.tolist()
    print("\nFeatures that can be discarded (p > 0.05):")
    print(discarded_features)

except Exception as e:
    print(f"An error occurred: {e}")
```

```
Optimization terminated successfully.
         Current function value: 0.138667
         Iterations 11
                           Logit Regression Results
==============================================================================
Dep. Variable:                 target   No. Observations:                  569
Model:                          Logit   Df Residuals:                      563
Method:                           MLE   Df Model:                            5
Date:                Fri, 27 Sep 2024   Pseudo R-squ.:                  0.7900
Time:                        06:34:39   Log-Likelihood:                -78.901
converged:                       True   LL-Null:                       -375.72
Covariance Type:            nonrobust   LLR p-value:                4.793e-126
==============================================================================
```

Figure 10

```
==============================================================================================
                                coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------------------
const                        13.3091      8.318      1.600      0.110      -2.995      29.613
mean radius                   2.1222      1.169      1.816      0.069      -0.168       4.413
mean texture                 -0.3003      0.055     -5.479      0.000      -0.408      -0.193
fractal dimension error     641.9064    155.194      4.136      0.000     337.732     946.081
mean area                    -0.0413      0.014     -2.981      0.003      -0.069      -0.014
worst fractal dimension    -154.6579     20.169     -7.668      0.000    -194.188    -115.128
==============================================================================================
```

Figure 11

```
Possibly complete quasi-separation: A fraction 0.14 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.

P-values for each predictor:
const                       1.096117e-01
mean radius                 6.940043e-02
mean texture                4.275974e-08
fractal dimension error     3.531769e-05
mean area                   2.874224e-03
worst fractal dimension     1.743968e-14
dtype: float64

Features that can be discarded (p > 0.05):
['const', 'mean radius']
```

Figure 12

3 Logistic regression First/Second-Order Methods

3.1 Use the code given in listing 4 to generate data.

3.2 Implement batch gradient descent to update the weights

Listing 6: Implement batch gradient descent

```
# 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)

# Add a bias term to X
X = np.c_[np.ones(X.shape[0]), X]
# Adding a column of ones for the bias
# Initialize weights randomly
np.random.seed(1)
weights = np.random.normal(0, 0.01, X.shape[1])
# Small random values

# Hyperparameters
learning_rate = 0.01
n_iterations = 20

# Batch Gradient Descent

for iteration in range(n_iterations):
    # Predictions

    predictions = X.dot(weights)
    # Calculate the loss (Mean Squared Error)
    loss = np.mean((predictions - y) ** 2)
    # Calculate the gradients
    gradients = (2 / X.shape[0]) * X.T.dot(predictions - y)
    # Update weights

    weights -= learning_rate * gradients


    # Print the loss for every iteration
    print(f"Iteration {iteration + 1}/{n_iterations},Loss: {loss:.4f}")
```

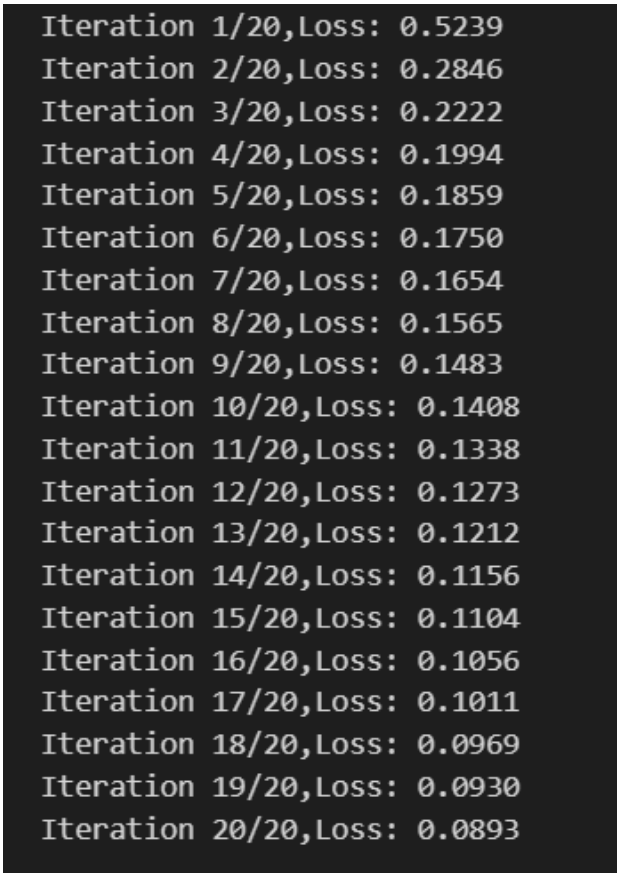

Figure 13: Output of batch gradient descent

Weight Initialization: Weights are initialized by drawing small random values from a normal distribution with a mean of 0 and a standard deviation of 0.01. **Reasons:**

- **Breaking Symmetry:** Small random weights prevent neurons from learning the same features, thus avoiding symmetry issues.
- **Stability:** Small values help keep gradients stable during early iterations of gradient descent, ensuring smoother convergence and avoiding instability.

3.3 Loss function

The loss function used is the Mean Squared Error (MSE):

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

Where:

- $\hat{y}_i$: Predicted value
- y_i: Actual target value
- n: Number of data points

Reasons for Selection:

- **Regression Task:** MSE is ideal for regression tasks as it measures the average squared difference between predictions and actual values.
- **Sensitivity to Large Errors:** Squared errors give more weight to large errors, encouraging the model to focus on reducing significant deviations.
- **Differentiability:** MSE is smooth and differentiable, which facilitates gradient-based optimization algorithms like gradient descent.

3.4 Loss with respect to number of iterations

Listing 7: Loss with respect to number of iterations

```
# Hyperparameters
learning_rate = 0.01
iterations = 20
m = len(y) # number of samples
# List to store the loss at each iteration
loss_history_gd = []
# Batch Gradient Descent
for i in range(iterations):
    #Calculate predictions
    predictions = np.dot(X, weights)
    # Compute the error
    error = predictions- y
    # Calculate the loss (MSE)
    loss = (1 / (2 * m)) * np.sum(error ** 2)
    # Store the loss in the list
    loss_history_gd.append(loss)
    # Compute the gradient
    gradient = (1 / m) * np.dot(X.T, error)
    # Update the weights
    weights = weights- learning_rate * gradient
# Plot the loss vs iterations
plt.plot(range(1, iterations + 1), loss_history_gd, marker='o')
plt.xlabel('Iterations')
plt.ylabel('loss (MSE)')
plt.title('loss Reduction over Iterations')
plt.grid(True)
plt.show()
```

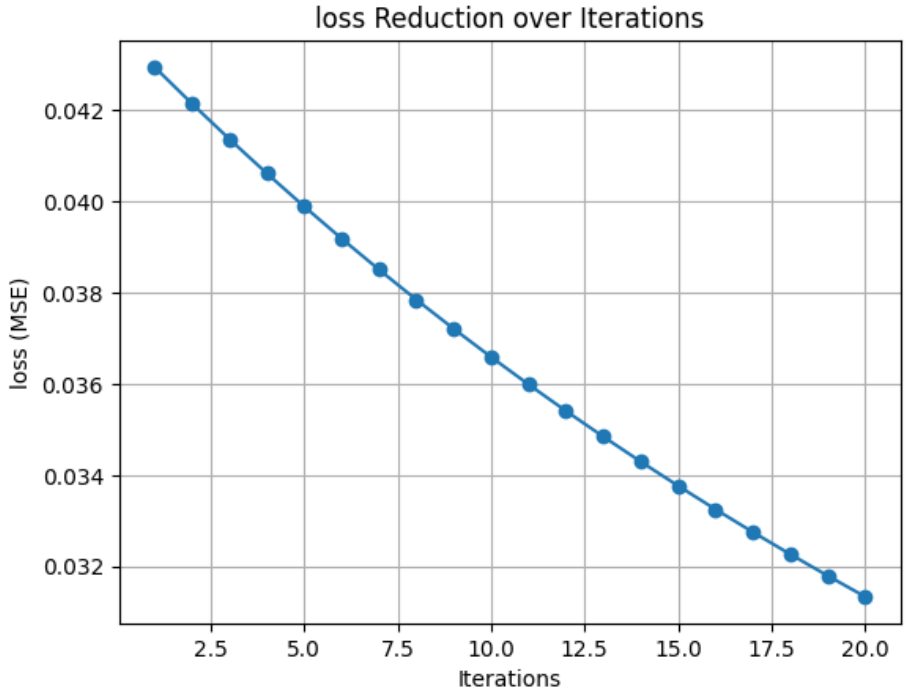

Figure 14

3.5 Repeat step 2 for stochastic Gradient descent

3.5.1 Loss with respect to number of iterations

Listing 8: Stochastic Gradient descent.

```
loss_history_sgd = []

# Stochastic Gradient Descent
for i in range(iterations):
    total_loss = 0

    for j in range(m):
        # Select one data point
        X_j = X[j, :].reshape(1,-1)
        y_j = y[j]
        # Calculate prediction
        prediction = np.dot(X_j, weights)
        # Compute the error
        error = prediction- y_j
        # Compute the loss (MSE for this data point)
        loss = (1 / 2) * (error ** 2)
        total_loss += loss
        # Compute the gradient
        gradient = np.dot(X_j.T, error)
        # Update the weights
        weights = weights- learning_rate * gradient.flatten()
    # Store the average loss for the epoch
    loss_history_sgd.append(total_loss / m)
    # Print the loss at each iteration
```

```
    average_loss = total_loss / m
    print(f"Iteration {i+1}, Loss: {average_loss}")
# Plot the loss vs iterations
plt.plot(range(1, iterations + 1), loss_history_sgd, marker='o')
plt.xlabel('Iterations')
plt.ylabel('loss (MSE)')
plt.title('loss Reduction over Iterations (SGD)')
plt.grid(True)
plt.show()
```

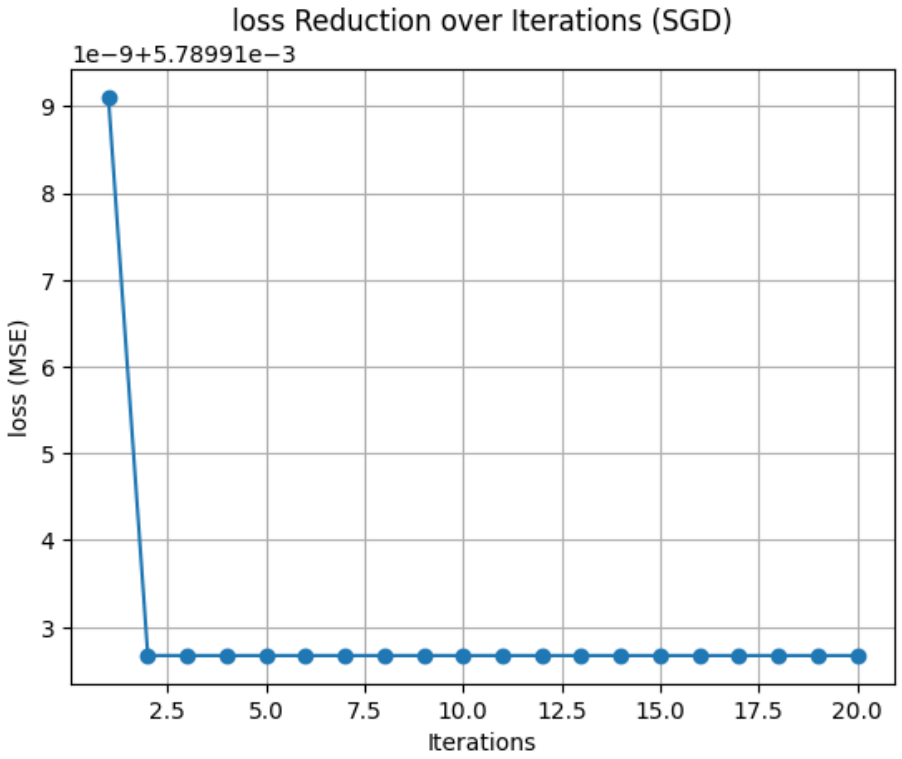

Figure 15

3.6 Implement Newton's method to update the weights

Listing 9: Newton's Method

```
loss_history_newton = []

# Newtons Method
for i in range(iterations):
    # Calculate predictions
    predictions = np.dot(X, weights)
    # Compute the error
    error = predictions- y
    # Compute the loss (MSE)
    loss = (1 / (2 * m)) * np.sum(error ** 2)
    loss_history_newton.append(loss)
    # Compute the gradient (first derivative)
    gradient = (1 / m) * np.dot(X.T, error)
```

```
    # Compute the Hessian (second derivative)
    hessian = (1 / m) * np.dot(X.T, X)
    # Update weights using Newtons method: W = W- H^-1 * gradient
    weights = weights- np.dot(np.linalg.inv(hessian), gradient)
    # Print the loss at each iteration
    print(f"Iteration {i+1}, Loss: {loss}")
```

3.7 Loss respective to the number of iterations

Listing 10: Newton's Method

```
 # Plot the loss vs iterations
plt.plot(range(1, iterations + 1), loss_history_newton, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Loss (MSE)')
plt.title('Loss Reduction over Iterations (Newton\ s  Method)')
plt.grid(True)
plt.show()
```

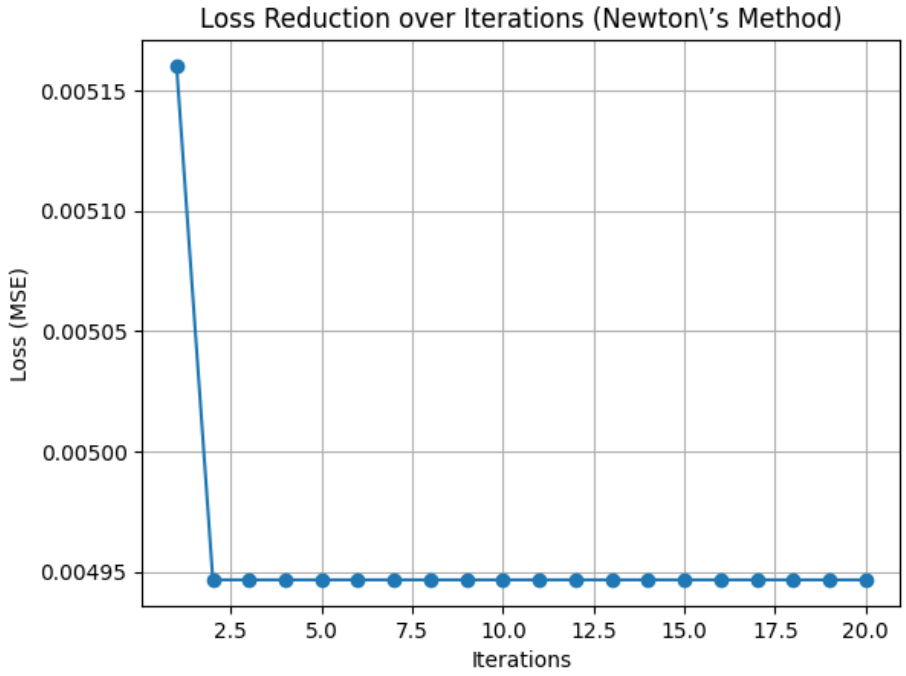

Figure 16

3.8 Loss with respect to number of iterations for Gradient descent, stochastic Gradient descent and Newton method's in a single plot

Listing 11: Newton's Method

```
 # Plot the loss vs iterations for each method
plt.plot(range(1, iterations + 1), loss_history_gd,
marker='o', label='Gradient Descent')
plt.plot(range(1, iterations + 1), loss_history_sgd,
```

```
5
6  marker='x', label='Stochastic Gradient Descent')
7  plt.plot(range(1, iterations + 1), loss_history_newton,
8  marker='s', label='Newton\'s Method')
9
10 # Adding labels and title
11 plt.xlabel('Iterations')
12 plt.ylabel('Loss (MSE)')
13 plt.title('Loss Reduction over Iterations for Different Methods')
14 plt.legend()
15 plt.grid(True)
16 plt.show()
```

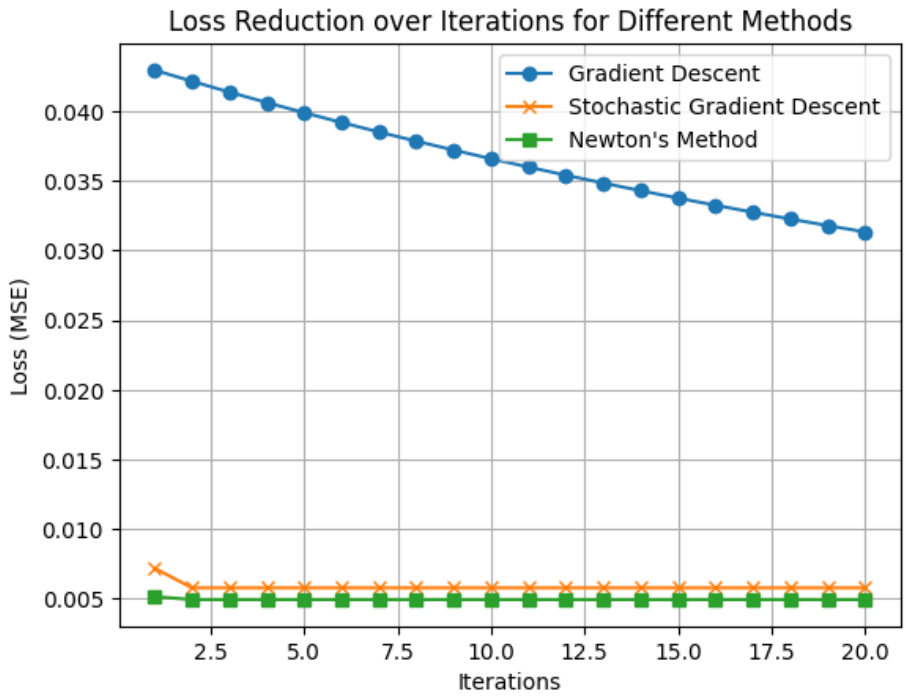

Figure 17: Comparison

3.9 Twoapproaches to decide number of iterations for Gradient descent and Newton's method.

3.9.1 Gradient Descent

Convergence Criteria

Define a stopping criterion based on the variation in the loss function or gradient. For instance: - **Loss Threshold**: Terminate iterations when the absolute difference in loss between successive iterations is less than a specified value:

$$|J(W_{t+1}) - J(W_t)| < \epsilon$$

- **Gradient Norm**: Halt when the gradient norm becomes sufficiently small:

$$\|\nabla J(W)\| < \delta$$

This suggests that additional updates will have negligible effect on the weights.

Early Stopping with Validation Set

Use early stopping with a validation set to track model performance:

- **Training**: Train the model for a maximum number of iterations.
- **Validation**: After each iteration, compute the validation loss. If the validation loss does not decrease for a predefined number of consecutive iterations (patience), stop the training.

This technique helps prevent over fitting and identifies the optimal number of iterations for convergence.

Newton's Method

- **Hessian Condition**: Use the Hessian matrix to attentively decide when to stop: - Calculate the condition number of the Hessian. If this number exceeds a certain threshold, it might signal that the optimization is diverging, which can trigger early stopping.
- **Weight Change**: Terminate when the change in weights between iterations becomes insignificant:

$$\|W_{t+1} - W_t\| < \epsilon$$

- **- Maximum Iterations with Improvement Check**: Set a limit on the number of iterations but ensure meaningful progress:
  - After a defined number of iterations, check the loss. If the improvement is below a predefined percentage (e.g., less than 1%), consider stopping the iterations. This helps avoid unnecessary computations when further updates have little effect.

3.10 Change of Centers

3.10.1 Update the Data Generation

Change the centers to [[3, 0], [5, 1.5]] in the data generation code.

Listing 12: Data Generation

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

# Generate synthetic data with new centers
np.random.seed(0)
centers = [[3, 0], [5, 1.5]]  # Updated centers
X, y = make_blobs(n_samples=2000, centers=centers, random_state=5)

# Apply transformation
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)
```

3.10.2 Initialize Weights and Batch Gradient Descent

Listing 13: Implementing batch gradient descent

```
# Initialize weights
W = np.random.randn(2)  # Assuming 2 features
learning_rate = 0.01
max_iters = 1000
epsilon = 1e-6

# Gradient descent loop
losses = []
for i in range(max_iters):
    # Prediction: using a linear model
    y_pred = X @ W

    # Loss: mean squared error
    loss = np.mean((y_pred - y) ** 2)
    losses.append(loss)

    # Compute gradients
    grad = (2 / len(X)) * X.T @ (y_pred - y)

    # Update weights
    W -= learning_rate * grad

    # Check convergence
    if np.linalg.norm(grad) < epsilon:
        print(f"Converged after {i} iterations")
        break

# Plot loss over iterations
plt.plot(losses)
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Convergence of Batch Gradient Descent')
plt.show()
```

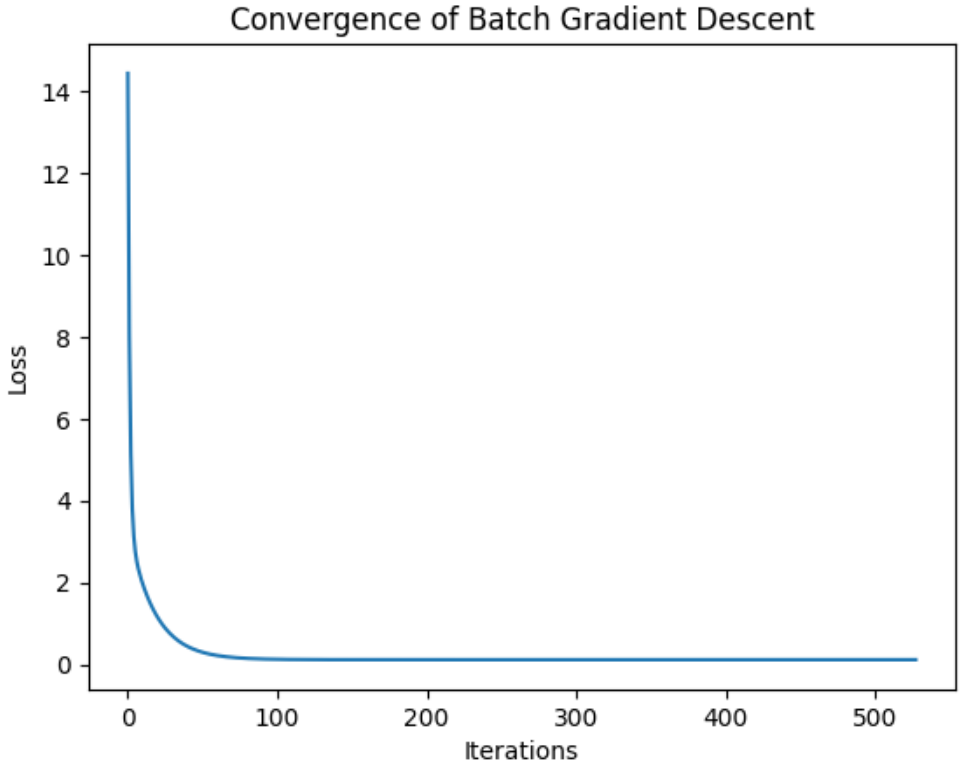

Figure 18

3.10.3 Convergence Behavior Analysis

- **Convergence Speed**: The new centers impact convergence by altering the data distribution. Greater separation leads to steeper gradients and faster updates.

- **Loss Reduction**: Monitor how quickly the loss decreases. A rapid drop followed by a plateau suggests the model is approaching optimal weights.

- **Impact of Centers**: Changing the centers to $[[3, 0], [5, 1.5]]$ affects cluster separation, influencing the difficulty of classification and convergence speed.

3.10.4 Explanation

- **Fast Convergence**: Likely due to better separability between clusters, allowing faster gradient updates and clearer decision boundaries.

- **Slow Convergence**: Likely caused by overlapping clusters, requiring more iterations for the weights to converge.

By plotting the loss over iterations, assess if further tuning, such as adjusting the learning rate or adding regularization, is necessary.