University of Moratuwa, Sri Lanka

Faculty of Engineering

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

EN3150 - Pattern Recognition

Assignment 02 : Learning from data and related challenges and classification

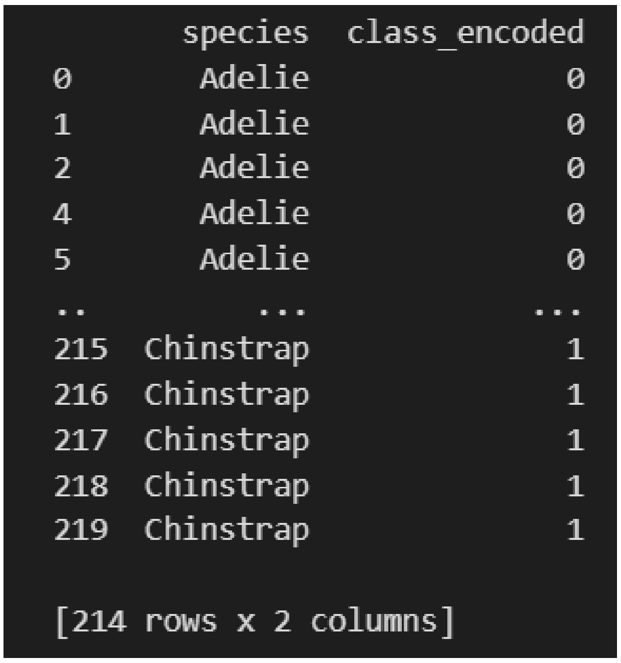
Kodikara U. S. S.
210293K

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

1.1 Load Data



```
       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: Loading data

1.2 Purpose of species encoding

In the penguins dataset, the species of penguins (*'Adelie'*, *'Chinstrap'*, and *'Gentoo'*) represents a categorical target variable used in classification tasks. Machine learning models, such as logistic regression, cannot work directly with categorical data, as they require numerical input. Therefore, encoding the species is necessary to transform these non-numeric labels into numerical values.

By using `LabelEncoder`, the species column is converted into integers (e.g., *'Adelie'* = 0, *'Chinstrap'* = 1, *'Gentoo'* = 2). This transformation allows the model to process the data and learn from the relationships between features (such as bill length, flipper length, and body mass) and the target species.

Furthermore, encoding improves the interpretability of the model by allowing it to quantify how each feature contributes to predicting the species. In classification tasks, encoding is not only critical for model training but also aids in data visualization, making it easier to analyze relationships between the penguins' physical characteristics and their species classifications.

1.3 Purpose of dropping columns

`X = df.drop(['species', 'island', 'sex'], axis=1)` line of code removes the columns `'species'`, `'island'`, and `'sex'` from the DataFrame `df`, creating a new DataFrame `X` that contains only the features relevant for model training.

Key Purposes

- **Preparing the Features**: The `'species'` column is the target variable that the model will predict, so it is excluded from the feature set to avoid data leakage. The `'island'` and `'sex'` columns may not be directly useful for prediction or may be categorical features that aren't needed in their current form, so they are dropped to focus on the numerical or relevant features for modeling.

- **Avoiding Data Leakage**: Including the target variable (`'species'`) in the features would lead to data leakage, where the model has access to the outcome during training, which would skew the performance metrics.

- **Simplifying the Dataset**: This step also simplifies the dataset by retaining only numerical measurements (such as bill length, flipper length, etc.), which are more directly usable by machine learning algorithms.

1.4 Why can not use "island" and "sex" features?

The features "island" and "sex" may not be crucial for distinguishing between species in this dataset. Analyzing the physical characteristics like bill length, bill depth, flipper length, and body mass often provides more direct insights for classification. Visualizing the distribution of features by species and grouping them by "island" and "sex" shows that these categorical features do not contribute significant distinctions between species. As a result, "island" and "sex" can introduce noise rather than useful information and might be omitted to simplify the model without losing predictive power.

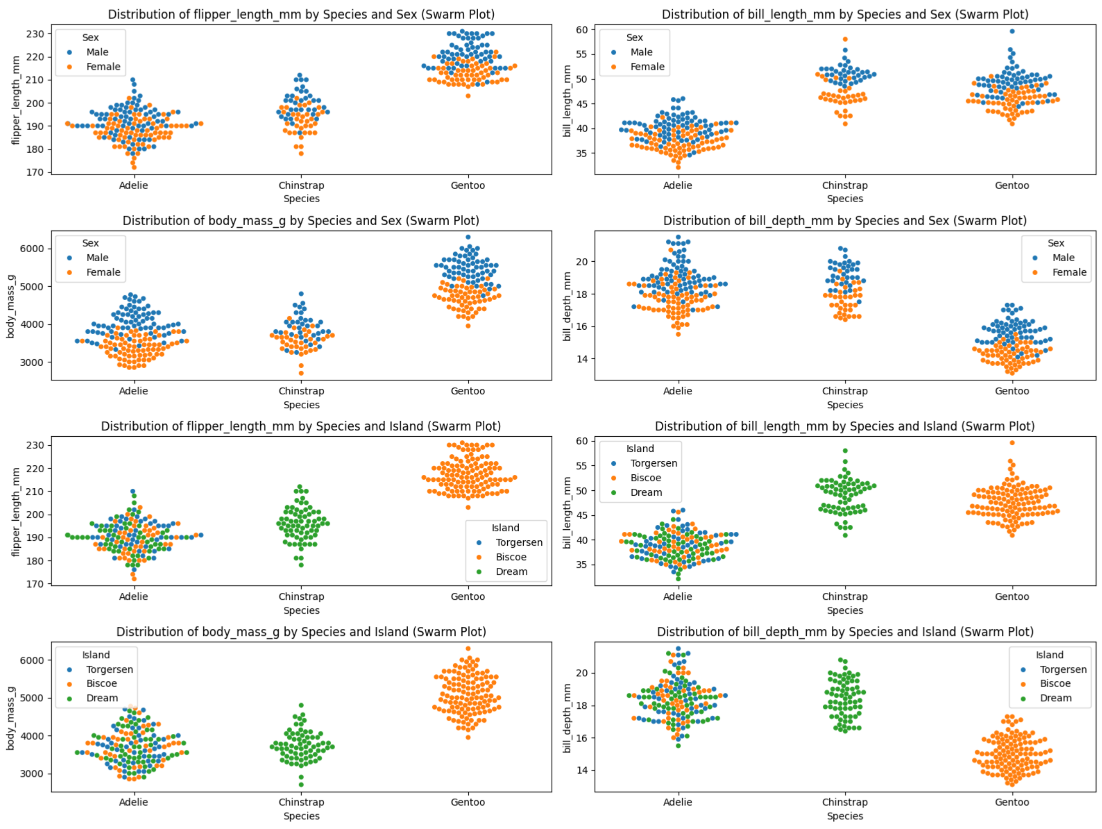
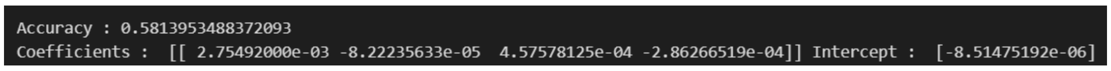


Figure 2: Distribution of feature by Species w. r. t. Island or Sex

1.5 Train a logistic regression model



```
Accuracy : 0.5813953488372093
Coefficients :  [[ 2.75492000e-03 -8.22235633e-05  4.57578125e-04 -2.86266519e-04]] Intercept :  [-8.51475192e-06]
```

Figure 3: Model accuracy, coefficients and intercept

1.6 Usage of `random_state = 42`

The `random_state = 42` ensures that the split between training and testing sets is consistent and reproducible across multiple runs of the code. This is important because, without setting a fixed random state, the dataset would be split differently each time, potentially causing variations in model performance. Setting a specific random state, like 42 (a commonly used arbitrary number), ensures the results remain consistent, making it easier to compare and evaluate the model's performance. The number itself holds no special meaning, and any integer could be used to achieve reproducibility.

1.7 Accuracy and saga solver performance

1.7.1 Why accuracy low?

The low accuracy could be due to the following reasons:

1. **Feature Selection:** The chosen features may not have a strong correlation with the target variable, leading to poor predictive power. Irrelevant features reduce the model's ability to distinguish between classes.

2. **Class Imbalance:** If one species is significantly more common, the model may overly predict the majority class, resulting in low accuracy for the minority class. Accuracy may not be the best metric in this scenario; metrics like precision, recall, or F1 score could provide a better evaluation.

3. **Model Limitation:** Logistic regression captures only linear relationships. If the data exhibits non-linear patterns, a more complex model (e.g., decision trees or random forests) might perform better.

4. **Data Quality:** Missing, noisy, or inconsistent data (though you've already dropped missing values) can reduce model performance. Data quality issues may persist even after cleaning.

1.7.2 Why does the saga solver perform poorly?

The **saga** solver can perform poorly due to two main reasons:

1. **Inappropriate Dataset:** The *saga* solver is optimized for large or sparse datasets. If the dataset is neither large nor sparse, other solvers like *lbfgs* or *liblinear* might provide better performance.

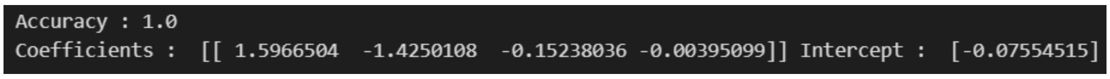
2. **Convergence Issues:** The *saga* solver may struggle with convergence if the features are not properly scaled. Logistic regression assumes features are on a similar scale, so large differences in feature magnitudes can cause the solver to have difficulties. Applying *StandardScaler* can help improve convergence and overall performance.

1.8 Using `liblinear` as solver

Classification accuracy = 1.0

```
# Split the data into training and testing sets
X_train , X_test , y_train , y_test = train_test_split (X , y ,
   test_size =0.2 , random_state =42)
#Train the logistic regression model . Here we are using saga
   solver to learn weights .
logreg = LogisticRegression(solver='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 ( "Coefficients : ", logreg.coef_ , "Intercept : ", logreg.
   intercept_ )
```

Listing 1: Logistic regression model with `liblinear` solver



```
Accuracy : 1.0
Coefficients :  [[ 1.5966504  -1.4250108  -0.15238036 -0.00395099]] Intercept :  [-0.07554515]
```

Figure 4: Model accuracy, coefficients and intercept

1.9 Better performance of `liblinear` solver over `saga` solver

The `liblinear` solver performs better than the `saga` solver in certain cases due to the following reasons:

1. **Dataset Size:** The `liblinear` solver is optimized for small to medium-sized datasets and uses a coordinate descent algorithm, which converges faster and more reliably on these datasets. In contrast, the `saga` solver is designed for large or sparse datasets and may struggle with slower convergence on smaller datasets.

2. **Binary vs. Multiclass:** `liblinear` is highly efficient for binary classification or one-vs-rest (OvR) multiclass problems. `saga`, while better for multinomial classification, performs less effectively on smaller datasets.

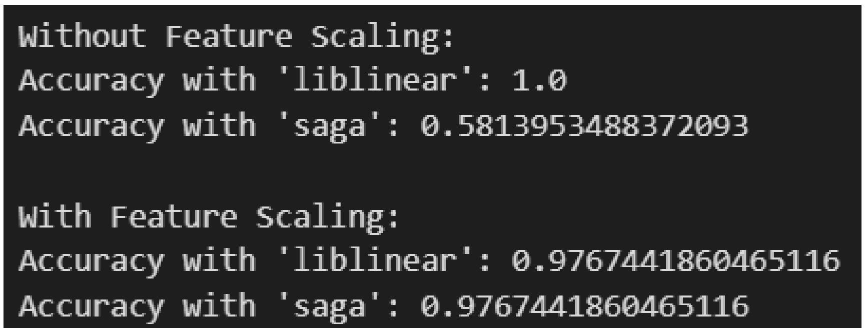
3. **Regularization Handling:** `liblinear` supports both L1 (lasso) and L2 (ridge) regularization and balances model complexity well, preventing over fitting on smaller datasets. `saga`, though it supports regularization, may not handle it as effectively on smaller data.

4. **Convergence Speed:** `liblinear` typically converges faster due to its deterministic approach, while `saga` can have slower convergence on smaller datasets because of its stochastic nature.

1.10 Comparison of the performance of `liblinear` and `saga` solvers with feature scaling

```
    # Scale the features using StandardScaler
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)
```

Listing 2: feature scaling using *Standard Scaler*



```
Without Feature Scaling:
Accuracy with 'liblinear': 1.0
Accuracy with 'saga': 0.5813953488372093

With Feature Scaling:
Accuracy with 'liblinear': 0.9767441860465116
Accuracy with 'saga': 0.9767441860465116
```

Figure 5: Model accuracy in each scenario

Explanation

Feature scaling plays a crucial role in the performance of optimization algorithms. The observed results can be summarized as follows:

1. **Impact of Feature Scaling:** Feature scaling standardizes the range of independent variables, ensuring equal contribution to distance calculations in optimization. This is particularly important for solvers like *saga*, which are sensitive to the scale of input features.

2. **Performance Without Feature Scaling:** The *liblinear* solver achieved perfect accuracy (1.0) without feature scaling, indicating its effectiveness in this scenario. In contrast, *saga* struggled, resulting in a significantly lower accuracy (0.58) due to difficulties in convergence caused by the varying scales of features.

3. **Performance With Feature Scaling:** After applying feature scaling, both solvers improved their performance. The *liblinear* accuracy slightly decreased to 0.98, suggesting that it may have been overfitting without scaling. Meanwhile, the *saga* solver's accuracy rose to 0.98, demonstrating that scaling allowed it to converge effectively and make accurate predictions.

1.11 Problem of the given code and the solution

Issue Explanation

The `ValueError` encountered in the logistic regression model, specifically the message "could not convert string to float: Dream", arises due to the presence of categorical variables in the feature set X. In the dataset, the columns `island` and `sex` contain string values, including the value `'Dream'`. Logistic regression requires all input features to be numeric, and the presence of these non-numeric entries leads to a failure during the fitting process.

Solution

To resolve this issue, the categorical features can be dropped entirely if they are deemed unnecessary for the analysis. The following line of code effectively removes these columns from the feature set:

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

By eliminating the categorical variables, the logistic regression model can be successfully fitted without encountering conversion errors. This approach simplifies the dataset and allows the model to focus on the numeric features relevant to the classification task.

1.12 Comment on the given approach

Applying label encoding to a categorical feature followed by feature scaling, such as Standard Scaling or Min-Max Scaling, is not a correct approach. Label encoding assigns integer values to each category (e.g., 'red' = 0, 'blue' = 1, 'green' = 2). This can create an ordinal relationship between the categories that does not exist in the original data. For example, the model might interpret 'green' (2) as being greater than 'blue' (1), which can lead to misleading results.

To properly handle categorical features, it is advisable to use one-hot encoding instead of label encoding. One-hot encoding converts each category into a new binary column (e.g., 'red' = [1, 0, 0], 'blue' = [0, 1, 0], 'green' = [0, 0, 1]). This way, the model does not assume any ordinal relationship between the categories. After one-hot encoding, you can then apply feature scaling to numerical features if necessary, but it is not required for the binary columns generated from one-hot encoding.

In summary, I think the best approach to avoid using label encoding followed by feature scaling for categorical features. Instead, use one-hot encoding to accurately represent categorical data without introducing false ordinal relationships.

Question 02

Probability Calculation

Estimating the probability that a student who has studied for 50 hours and has an undergraduate GPA of 3.6 will receive an A+ in the class.
The logistic regression model is given by:

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

Where:

- $w_0 = -5.9$ is the intercept,
- $w_1 = 0.06$ is the coefficient for the number of hours studied (x_1),
- $w_2 = 1.5$ is the coefficient for the undergraduate GPA (x_2).

The values for the student are:

- $x_1 = 50$ (hours studied),
- $x_2 = 3.6$ (undergraduate GPA).

Step 1: Compute the Linear Combination

First, calculate the value of the linear combination z:

$$z = w_0 + w_1x_1 + w_2x_2 = -5.9 + (0.06 \times 50) + (1.5 \times 3.6)$$

$$z = -5.9 + 3 + 5.4 = 2.5$$

Step 2: Compute the Probability

Now, apply the logistic function to obtain the probability:

$$P(y = 1|x_1, x_2) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-2.5}}$$

Using the value $e^{-2.5} \approx 0.0821$, calculate:

$$P(y = 1|x_1, x_2) = \frac{1}{1 + 0.0821} = \frac{1}{1.0821} \approx 0.923$$

The estimated probability that the student will receive an A+ in the class is approximately 0.923 or 92.3%.

Study hours calculation

To determine how many hours of study a student needs to achieve a 60% chance of receiving an A+ in the class, let's use the logistic regression model.
The logistic regression equation is given by:

$$P(y = 1|x_1, x_2) = \frac{1}{1 + e^{-(w_0 + w_1x_1 + w_2x_2)}}$$

Where:

- $w_0 = -5.9$ is the intercept,
- $w_1 = 0.06$ is the coefficient for the number of hours studied (x_1),
- $w_2 = 1.5$ is the coefficient for undergraduate GPA (x_2),
- $x_2 = 3.6$ (undergraduate GPA).

The goal is to achieve a 60% chance of receiving an A+, so we set $P(y = 1|x_1, x_2) = 0.60$ and solve for x_1 (hours studied).

Step 1: Set up the equation

We know that:

$$0.60 = \frac{1}{1 + e^{-(w_0 + w_1x_1 + w_2x_2)}}$$

Multiplying both sides by $1 + e^{-(w_0 + w_1x_1 + w_2x_2)}$ and solving for the exponential term:

$$0.60(1 + e^{-(w_0 + w_1x_1 + w_2x_2)}) = 1$$

$$0.60e^{-(w_0 + w_1x_1 + w_2x_2)} = 0.40$$

$$e^{-(w_0 + w_1x_1 + w_2x_2)} = \frac{0.40}{0.60} = \frac{2}{3}$$

Step 2: Take the natural logarithm

Taking the natural logarithm of both sides:

$$-(w_0 + w_1x_1 + w_2x_2) = \ln\left(\frac{2}{3}\right)$$

Using $\ln\left(\frac{2}{3}\right) \approx -0.405$, we get:

$$w_0 + w_1x_1 + w_2x_2 = 0.405$$

Step 3: Substitute the known values

Now, we substitute the known values for w_0, w_1, w_2, and x_2:

$$-5.9 + 0.06x_1 + 1.5(3.6) = 0.405$$
$$-5.9 + 0.06x_1 + 5.4 = 0.405$$
$$-0.5 + 0.06x_1 = 0.405$$

$$0.06x_1 = 0.405 + 0.5 = 0.905$$
$$x_1 = \frac{0.905}{0.06} \approx 15.08$$

The student needs to study approximately 15.08 hours to achieve a 60% chance of receiving an A+ in the class.

2 Logistic regression on real world data

2.1 Choosing a data set from UCI Machine Learning Repository

I have selected the **Bank Marketing** dataset.

```
from ucimlrepo import fetch_ucirepo

# fetch dataset
bank_marketing = fetch_ucirepo(id=222)

# data (as pandas dataframes)
X = bank_marketing.data.features
y = bank_marketing.data.targets

# metadata
print(bank_marketing.metadata)

# variable information
print(bank_marketing.variables)
```

Listing 3: Loading dataset

2.2 Correlation matrix and Pair plots

```
data['target'] = y  # Add the target variable to the DataFrame
pairplot = data[selected_features + ['target']]  # Select the
    features and target variable

# Create a pairplot
sns.pairplot(pairplot, hue='target', diag_kind='kde')

plt.show()
```

Listing 4: Pair plots

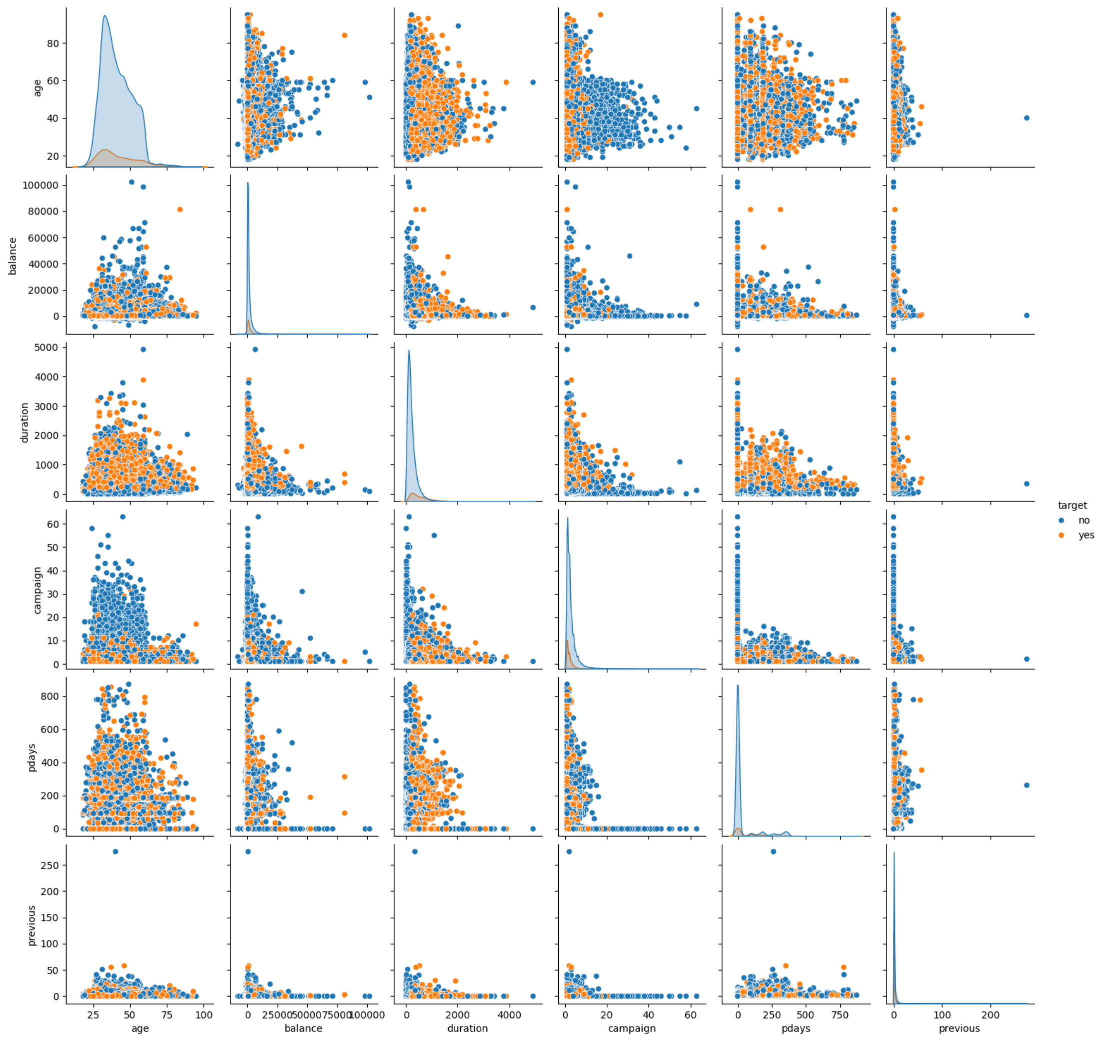
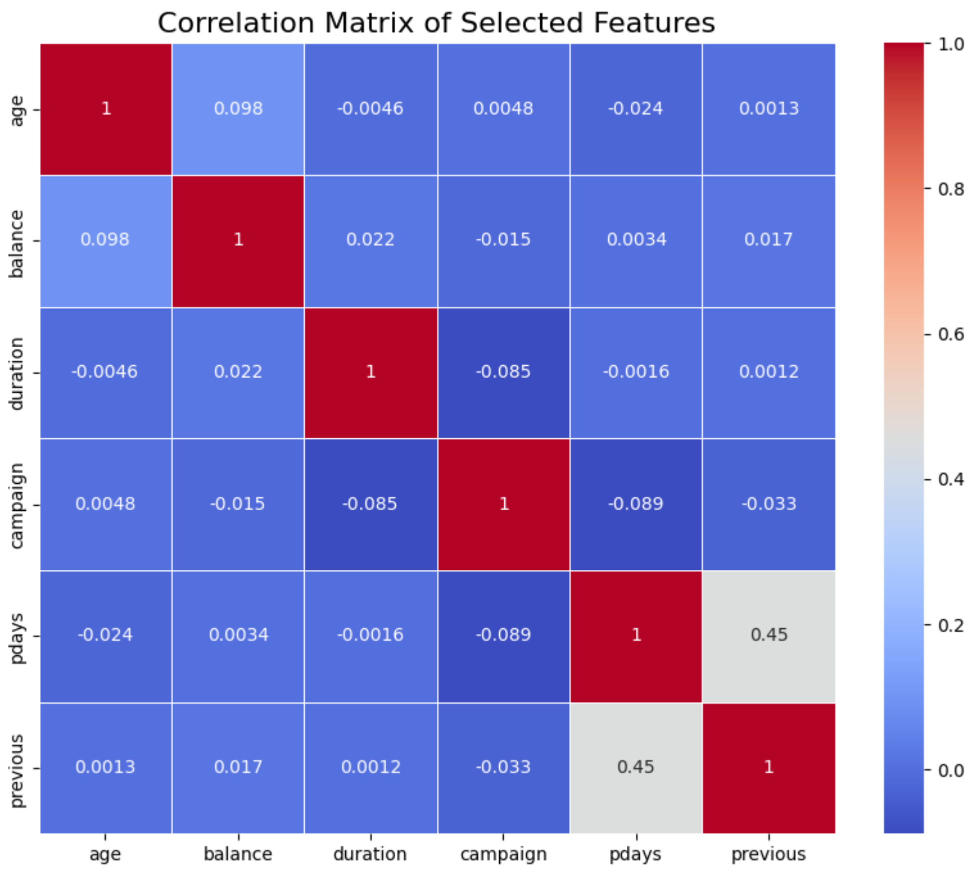


Figure 6: Pair plot

```
import matplotlib.pyplot as plt

# Combine the features and targets into a single DataFrame for
    analysis
data = pd.concat([X, y], axis=1)
```

```

selected_features = ['age','balance','duration','campaign','pdays
   ','previous']

# Compute the correlation matrix
correlation_matrix = data[selected_features].corr()

plt.figure( figsize =(10 , 8)) # Set the figure size
sns.heatmap (correlation_matrix , annot = True , cmap ='coolwarm'
   ,linewidths =0.5) # Create a heatmap

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

# Show the heatmap
plt.show ()

# Display the correlation matrix
print("Correlation Matrix:")
print(correlation_matrix)
```

Listing 5: Correlation Matrix



	age	balance	duration	campaign	pdays	previous
age	1	0.098	-0.0046	0.0048	-0.024	0.0013
balance	0.098	1	0.022	-0.015	0.0034	0.017
duration	-0.0046	0.022	1	-0.085	-0.0016	0.0012
campaign	0.0048	-0.015	-0.085	1	-0.089	-0.033
pdays	-0.024	0.0034	-0.0016	-0.089	1	0.45
previous	0.0013	0.017	0.0012	-0.033	0.45	1

Figure 7: Correlation Matrix

2.2.1 Observations on Results

Distribution Differences Between Classes (Target: Yes vs No)

- **Age:** Overlap exists between classes, with 'no' dominating across age ranges; no distinct separation is observed.
- **Balance:** A slight tendency for 'yes' (orange) to have higher balance values, suggesting potential influence on the target class.
- **Duration:** Clear separation is evident, indicating its importance in distinguishing between classes.
- **Campaign:** No strong separation; both classes are evenly distributed, implying limited predictive power.
- **Pdays and Previous:** Minimal separation, with most data clustered around lower values, suggesting limited contribution to class distinction.

Relationships Between Features

- **Age and Balance:** Weak positive correlation (0.098), indicating slight interdependence, but not significant.
- **Pdays and Previous:** Moderate correlation (0.45) suggests redundancy; consider potential removal or combination.
- **Campaign:** Weak correlations with other features (all ¡ —0.1—), indicating limited relevance for predictive modeling.
- **Overall Correlations:** Most features exhibit low correlations, minimizing concerns of multicollinearity and ensuring independent contributions.

2.3 Fit a logistic regression model to predict the dependent variable

```
from sklearn.metrics import accuracy_score, classification_report
    ,confusion_matrix

X = data[selected_features] # Features

X_train , X_test , y_train , y_test = train_test_split(X , y ,
    test_size =0.2 , random_state =50)

#Fit the Logistic Regression model
model = LogisticRegression(max_iter=200)
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)
```

Listing 6: Logistic Regression Model

Accuracy: 0.89

Confusion Matrix:

$$\begin{bmatrix} 7876 & 129 \\ 859 & 179 \end{bmatrix}$$

Classification Report:

```
              precision    recall  f1-score   support

          no       0.90      0.98      0.94      8005
         yes       0.58      0.17      0.27      1038

    accuracy                           0.89      9043
   macro avg       0.74      0.58      0.60      9043
weighted avg       0.86      0.89      0.86      9043
```

The model achieved an accuracy of **0.89**, correctly predicting the target class 89% of the time.
Confusion Matrix:

$$\begin{bmatrix} 7876 & 129 \\ 859 & 179 \end{bmatrix}$$

7876 true negatives and 179 true positives. 129 false positives and 859 false negatives.
Classification Report:

- For class "no":
  - Precision: 0.90, meaning 90% of predicted "no" instances were correct.
  - Recall: 0.98, indicating the model correctly identified 98% of actual "no" cases.
- For class "yes":
  - Precision: 0.58, indicating 58% of predicted "yes" instances were correct.
  - Recall: 0.17, showing the model struggles to identify actual "yes" cases.

While the model performs well in predicting the "no" class, it struggles with the "yes" class, reflected in the lower precision and recall for that class.

2.4 Interpretation of P values

```
    # Check if y is a DataFrame with more than one column - I wrote
      this as I experienced some errors while implementing the code

    print(y.head())  # Display the first few rows of y

    if isinstance(y, pd.DataFrame):
```

```
    y = y.iloc[:, 0]  # Select the first column

# Ensure y is numeric (binary 0/1)
y_numeric = y.apply(lambda x: 1 if x == 'yes' else 0)

# Add a constant to the model (intercept term)
X_const = sm.add_constant(X_encoded)

# Fit the Logistic Regression model using statsmodels
logit_model = sm.Logit(y_numeric, X_const)
result = logit_model.fit()

# Summary of the model, including p-values
print(result.summary())

# Extract p-values
p_values = result.pvalues
print("\nP-values for the predictors:")
print(p_values)

# Check if any features have high p-values (suggesting they can
   be discarded)
alpha = 0.05  # threshold for significance level
insignificant_features = p_values[p_values > alpha].index
print(f"\nFeatures with p-values > {alpha} (can potentially be
   discarded):")
print(insignificant_features)
```

Listing 7: Obtaining P values

Optimization terminated successfully.
Current function value: 0.293213
Iterations: 7

2.4.1 Logit Regression Results

Parameter	Value
Dependent Variable	y
No. Observations	45211
Method	MLE (Maximum Likelihood Estimation)
Converged	True
Iterations	7
Log-Likelihood	-13256
Pseudo R-squared	0.1875
LL-Null	-16315
LLR p-value	0.000

Table 1: Model Summary

Predictor	Coefficient	Std. Error	z	P	95% Conf. Interval
const	-3.4952	0.071	-49.538	0.000	[-3.633, -3.357]
age	0.0080	0.001	5.425	0.000	[0.005, 0.011]
balance	3.715e-05	4.29e-06	8.663	0.000	[2.87e-05, 4.56e-05]
duration	0.0036	5.64e-05	64.480	0.000	[0.004, 0.004]
campaign	-0.1288	0.010	-13.503	0.000	[-0.148, -0.110]
pdays	0.0021	0.000	13.789	0.000	[0.002, 0.002]
previous	0.0860	0.007	11.669	0.000	[0.072, 0.100]

Table 2: Logistic Regression Coefficients

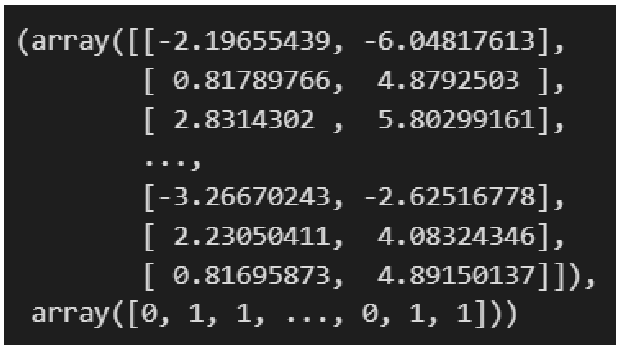
P-values for the predictors:

const	:	$0.000000e+00$
age	:	$5.783589e-08$
balance	:	$4.615247e-18$
duration	:	$0.000000e+00$
campaign	:	$1.511568e-41$
pdays	:	$2.955997e-43$
previous	:	$1.830548e-31$

The p-values of all predictors are below the standard threshold of 0.05, indicating that all features are statistically significant. Therefore, none of the features are candidates for removal based on their contribution to the model. Each predictor plays an important role in predicting the target variable.

3 Logistic regression First/Second-Order Methods

3.1 Data generation



```
(array([[-2.19655439, -6.04817613],
        [ 0.81789766,  4.8792503 ],
        [ 2.8314302 ,  5.80299161],
        ...,
        [-3.26670243, -2.62516778],
        [ 2.23050411,  4.08324346],
        [ 0.81695873,  4.89150137]]),
 array([0, 1, 1, ..., 0, 1, 1]))
```

Figure 8: Data generation using Listing 4

3.2 Batch Gradient Descent

```
    import numpy as np
    from sklearn.datasets import make_blobs

    # Generate synthetic data
    np.random.seed(0)
```

```python
    centers = [[-5, 0], [5, 1.5]]
    X, y = make_blobs(n_samples=2000, centers=centers, random_state
       =5)

    # Transformation to add more complexity to the data
    transformation = [[0.5, 0.5], [-0.5, 1.5]]
    X = np.dot(X, transformation)

    # Add a bias term (column of ones)
    X = np.c_[np.ones(X.shape[0]), X]  # Adding bias to the input

    # Initialize weights randomly with a small value
    np.random.seed(1)
    weights = np.random.normal(0, 0.01, X.shape[1])

    # Set hyperparameters
    learning_rate = 0.01
    n_iterations = 20

    # To store the loss over iterations
    losses = []

    # Batch Gradient Descent
    for iteration in range(n_iterations):
        # Compute the predictions (linear combination of weights and
           input features)
        predictions = X.dot(weights)

        # Compute Mean Squared Error (MSE) Loss
        loss = np.mean((predictions - y) ** 2)
        losses.append(loss)

        # Calculate gradients
        gradients = (2 / X.shape[0]) * X.T.dot(predictions - y)

        # Update the weights using the gradient descent rule
        weights -= learning_rate * gradients

        # Print loss at each iteration for tracking progress
        print(f"Iteration {iteration + 1}/{n_iterations}, Loss: {loss
           :.4f}")
```

Listing 8: Batch gradient descent code

3.2.1 Weight Initialization Method

The weights are initialized by sampling small random values from a normal distribution with a mean of 0 and a standard deviation of 0.01.

3.2.2 Reason for the Selection

- **Breaking Symmetry:** Initializing weights with small random values helps prevent symmetry issues. If all weights were initialized to the same value (e.g., zero), neurons would learn the same features during training, making the model ineffective. Random initialization ensures that different neurons learn distinct features.
- **Stability with Small Values:** Using small values ensures that the gradients remain stable during the early stages of training. Large initial weights could lead to large gradients, which may cause unstable optimization or slow convergence.

This method strikes a balance between introducing randomness and ensuring that values are small enough to prevent instability during training.

3.3 Loss Function

The loss function used in this model is the **Mean Squared Error (MSE)**:

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

Where:

- y_i is the true value for the i^{th} sample.
- $\hat{y}_i$ is the predicted value for the i^{th} sample.
- n is the total number of samples.

Reason for Selection

- **Continuous Target Values:** MSE is commonly used in regression tasks where the target variable is continuous. It measures the average squared difference between the true values and the predicted values, making it suitable for this context.
- **Sensitive to Large Errors:** MSE penalizes larger errors more heavily because it squares the differences, which encourages the model to reduce larger errors during training.
- **Smooth Gradient:** The MSE loss function has a continuous and smooth gradient, making it easy to optimize using gradient-based algorithms like gradient descent.

3.4 Loss plot

```
    # Plot the loss with respect to the number of iterations
    plt.plot(range(1, n_iterations + 1), losses, marker='o',
      linestyle='-', color='b')
    plt.title('Loss vs Number of Iterations')
    plt.xlabel('Number of Iterations')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.show()
```

Listing 9: Plotting the loss

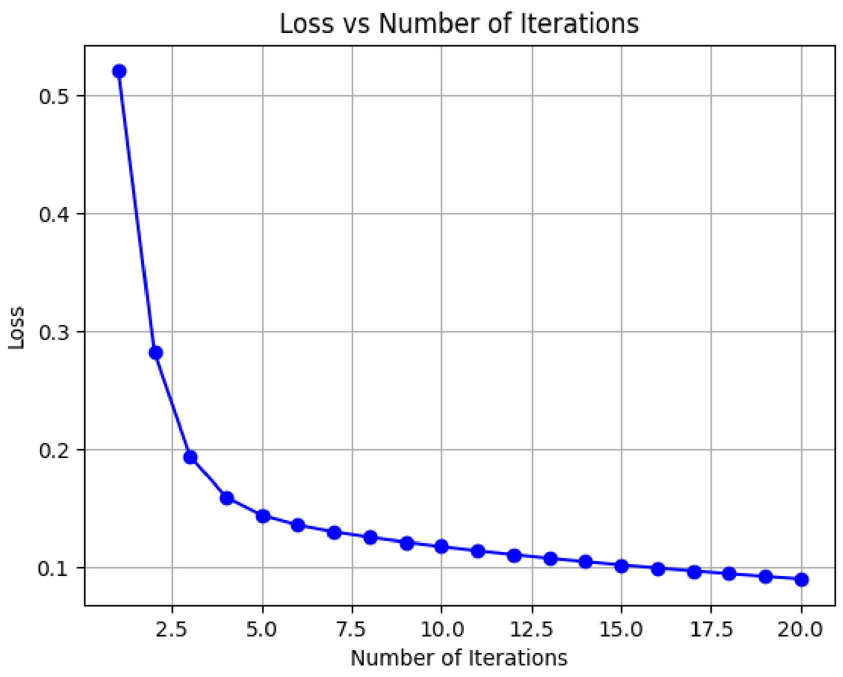


Figure 9: Plotting the loss

3.5 Stochastic Gradient Descent

```python
# Add a bias term to X (column of ones)
X = np.c_[np.ones(X.shape[0]), X]

# Initialize weights randomly
np.random.seed(1)
weights = np.random.normal(0, 0.01, X.shape[1])

# Hyperparameters
learning_rate = 0.01
n_iterations = 20

# To store the loss over iterations
losses = []

# Stochastic Gradient Descent
for iteration in range(n_iterations):
    total_loss = 0  # Track loss for the current iteration

    # Loop through each data point
    for i in range(X.shape[0]):
        # Get a single data point
        x_i = X[i, :]
        y_i = y[i]

        # Prediction for a single data point
        prediction = np.dot(x_i, weights)
```

```python

            # Calculate the loss (Mean Squared Error for this point)
            loss = (prediction - y_i) ** 2
            total_loss += loss

            # Calculate the gradient for this data point
            gradient = 2 * x_i * (prediction - y_i)

            # Update weights using the gradient
            weights -= learning_rate * gradient

        # Average loss over all data points for this iteration
        avg_loss = total_loss / X.shape[0]
        losses.append(avg_loss)

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

    # Plot the loss with respect to the number of iterations
    plt.plot(range(1, n_iterations + 1), losses, marker='o', linestyle='-', color='b')
    plt.title('SGD: Loss vs Number of Iterations')
    plt.xlabel('Number of Iterations')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.show()
```
Listing 10: Stochastic gradient descent

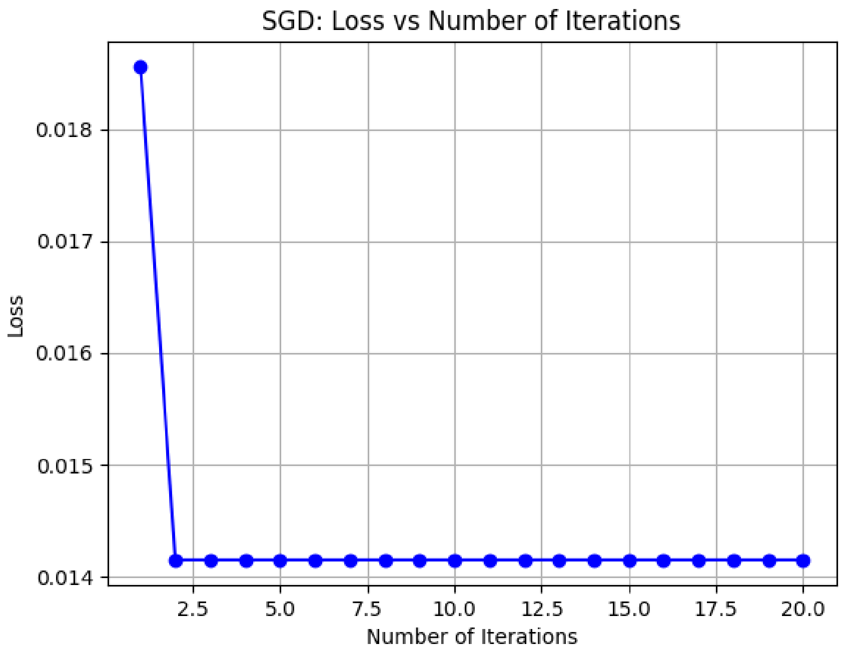


Figure 10: Plotting the loss

3.6 Implementation of Newton's method to update the weights for the given dataset over 20 iterations

```
# Initialize weights randomly
np.random.seed(1)
weights = np.random.normal(0, 0.01, X.shape[1])

# Hyperparameters
learning_rate = 0.01
n_iterations = 20

# To store the loss over iterations
losses = []

# Stochastic Gradient Descent
for iteration in range(n_iterations):
    total_loss = 0  # Track loss for the current iteration

    # Loop through each data point
    for i in range(X.shape[0]):
        # Get a single data point
        x_i = X[i, :]
        y_i = y[i]

        # Prediction for a single data point
        prediction = np.dot(x_i, weights)

        # Calculate the loss (Mean Squared Error for this point)
        loss = (prediction - y_i) ** 2
        total_loss += loss

        # Calculate the gradient for this data point
        gradient = 2 * x_i * (prediction - y_i)

        # Update weights using the gradient
        weights -= learning_rate * gradient

    # Average loss over all data points for this iteration
    avg_loss = total_loss / X.shape[0]
    losses.append(avg_loss)

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

Listing 11: Implementation of Newton's method

3.7 Loss plot

```
    plt.plot(range(1, iterations + 1), loss_history_newton, marker='o
    ', linestyle='-', color='b')
    plt.title("Newton's Method: Loss vs Number of Iterations")
    plt.xlabel("Number of Iterations")
    plt.ylabel("Loss")
    plt.grid(True)
    plt.show()
```

Listing 12: Plotting the loss

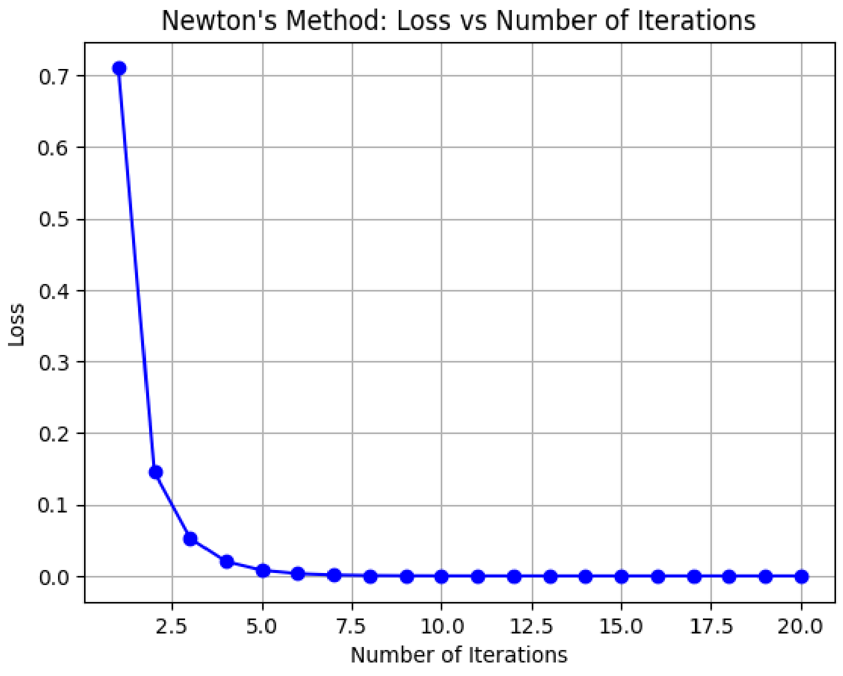


Figure 11: Plotting the loss

3.8 Plot in the same graph

```
    # Plotting the loss for each method
    plt.figure(figsize=(10, 6))
    plt.plot(range(1,iterations + 1), bgd_losses, marker='o',
    linestyle='-', color='b', label='Gradient Descent')
    plt.plot(range(1,iterations + 1), sgd_losses, marker='x',
    linestyle='-', color='r', label='Stochastic Gradient Descent')
    plt.plot(range(1,iterations + 1), loss_history_newton, marker='s'
    , linestyle='-', color='g', label='Newton\'s Method')

    plt.title("Loss Reduction over Iterations for Different Methods")
    plt.xlabel("Iterations")
    plt.ylabel("Loss (MSE)")
    plt.ylim(0, 0.3)  # Adjust y-axis limit to see more details
    plt.grid(True)
    plt.legend()
    plt.show()
```

Listing 13: Plotting in single graph

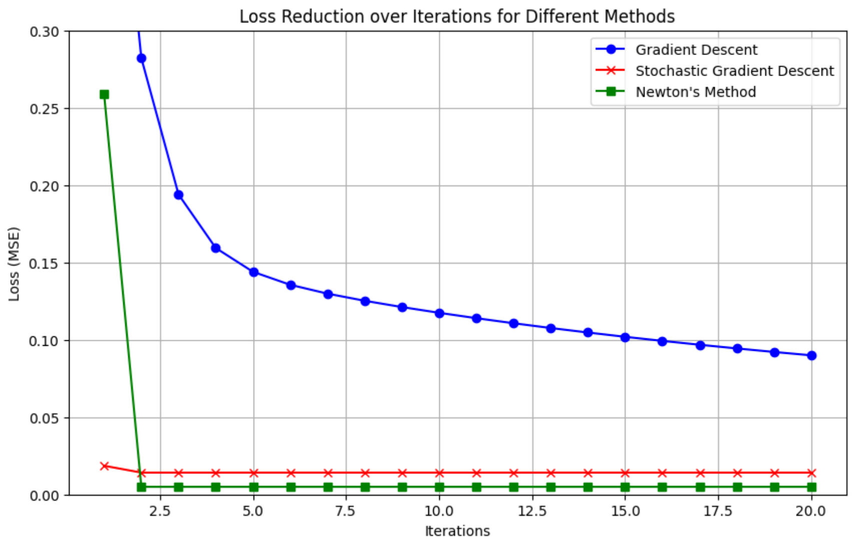


Figure 12: Plot

3.8.1 Comment on results

- **Gradient Descent:** Loss decreases steadily, showing effective convergence over iterations. The method performs well, but convergence is relatively slower compared to others.
- **Stochastic Gradient Descent (SGD):** Exhibits fluctuations in loss reduction due to its sample-by-sample update approach. While this adds variability, it can also help escape local minima.
- **Newton's Method:** Demonstrates rapid loss reduction, stabilizing at a lower value with fewer iterations. This method converges quickly but is computationally intensive due to the Hessian matrix calculation.
- **Overall Comparison:** Newton's Method is the most efficient in terms of convergence speed and final loss value. Gradient Descent offers steady improvement, while SGD can handle large datasets effectively but may require more tuning.

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

3.9.1 Convergence Criteria

- **Gradient Descent:** Set a convergence threshold based on the change in loss or gradient. Stop iterations when the change in loss is less than a predefined threshold (e.g., $\epsilon = 0.001$) or the gradient's magnitude is below a certain value.
- **Newton's Method:** Use similar convergence criteria, checking for small changes in loss or a small gradient norm. Additionally, set a maximum iteration limit to control computational resources.

3.9.2 Cross-Validation

- **Gradient Descent:** Implement k-fold cross-validation, training the model over a range of iterations (e.g., 10 to 1000) and selecting the number that minimizes the validation error.
- **Newton's Method:** Perform cross-validation with a narrower range (e.g., 5 to 50 iterations), focusing on validation performance to determine the optimal count.

3.10 Convergence behavior of the algorithm with updated data

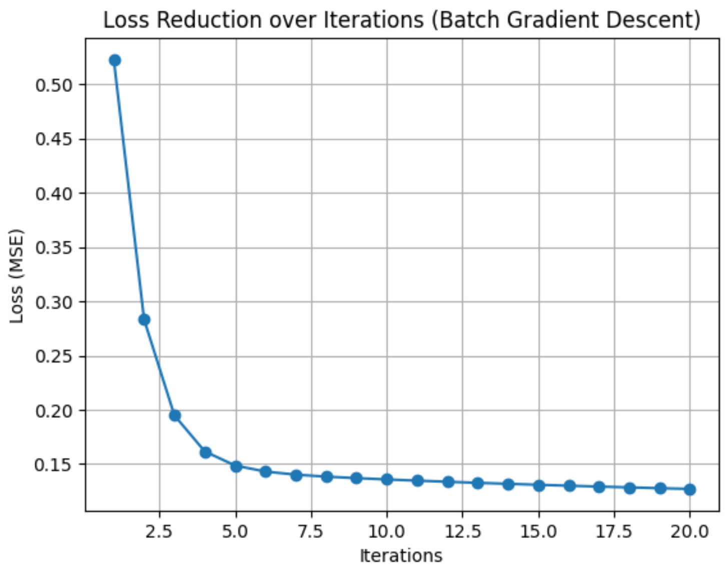


Figure 13: New plot

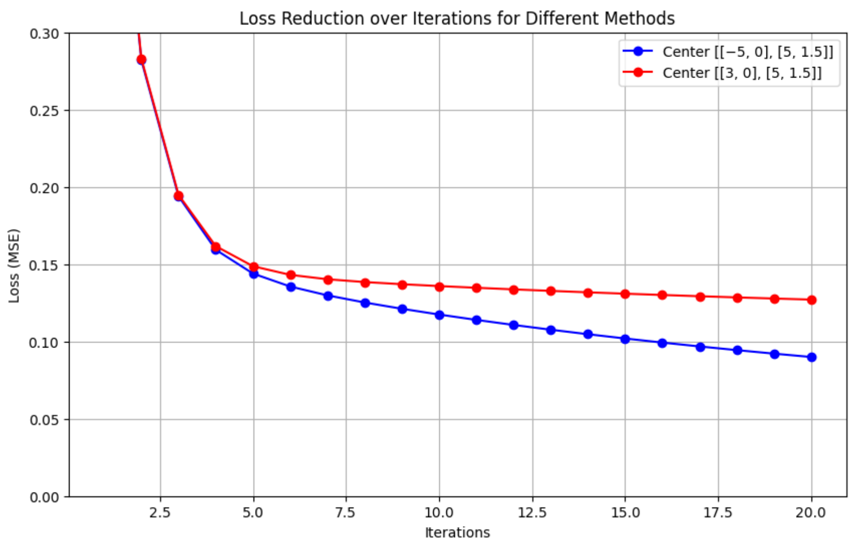


Figure 14: Comparison with previous center

The final loss values after 20 iterations for the two center configurations are as follows:

- Final Loss for Centers $[-5, 0]$ and $[5, 1.5]$: 0.0899

- Final Loss for Centers $[3, 0]$ and $[5, 1.5]$: 0.1271

The plots illustrate the Mean Squared Error (MSE) loss over the iterations. The configuration with centers $[-5, 0]$ and $[5, 1.5]$ achieves a final loss of 0.0899, indicating a better fit for the gradient descent algorithm. In contrast, the configuration with centers $[3, 0]$ and $[5, 1.5]$ results in a higher final loss of 0.1271, suggesting that the model finds it more challenging to minimize the error for this dataset.

3.10.1 Explanation for Convergence Behavior

- The lower final loss for the center configuration $[-5, 0]$ and $[5, 1.5]$ indicates that its data distribution is more conducive to effective modeling and error minimization by the gradient descent algorithm.
- The higher final loss for the configuration $[3, 0]$ and $[5, 1.5]$ implies a more complex data structure that presents greater difficulties in optimization, possibly due to increased variability or feature overlap.

3.10.2 Weight Changes Over Iterations

The weight updates reveal distinct behaviors:

- The weights for the configuration $[-5, 0]$ and $[5, 1.5]$ stabilize relatively quickly, reflecting a lower final loss.
- In contrast, the weights for the configuration $[3, 0]$ and $[5, 1.5]$ exhibit slower stabilization, aligning with the higher final loss observed.

Explanation for Weight Behavior

- The rapid convergence of weights for the configuration $[-5, 0]$ and $[5, 1.5]$ correlates with the model's ability to find optimal weights efficiently, contributing to a lower final loss.
- The slower adjustment of weights for the configuration $[3, 0]$ and $[5, 1.5]$ indicates that the model struggles to adapt to this dataset, resulting in a higher final loss and longer convergence time.

3.10.3 Comparing Convergence Behavior with Final Loss Values

- **Center Configuration** $[-5, 0]$ **and** $[5, 1.5]$:
  - **Final Loss**: 0.0899
  - The loss reduction is rapid, demonstrating effective convergence.
  - The data distribution is favorable for gradient descent, allowing for optimal weight adjustments.
- **Center Configuration** $[3, 0]$ **and** $[5, 1.5]$:
  - **Final Loss**: 0.1271
  - The loss reduction is slower, indicating challenges in convergence.
  - The more complex data distribution likely introduces greater variability or overlap, making it harder for the model to fit effectively.

In summary, the configuration with centers $[-5, 0]$ and $[5, 1.5]$ converges faster and achieves a lower final loss due to its more suitable data distribution for the gradient descent algorithm. Conversely, the configuration $[3, 0]$ and $[5, 1.5]$ experiences slower convergence and a higher final loss, reflecting the complexities inherent in its data distribution.