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

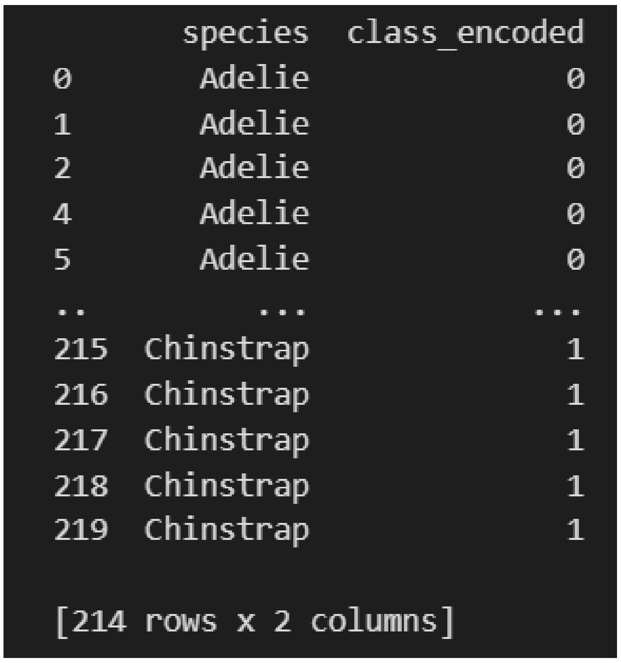
Kodikara U. S. S.
210293K

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.

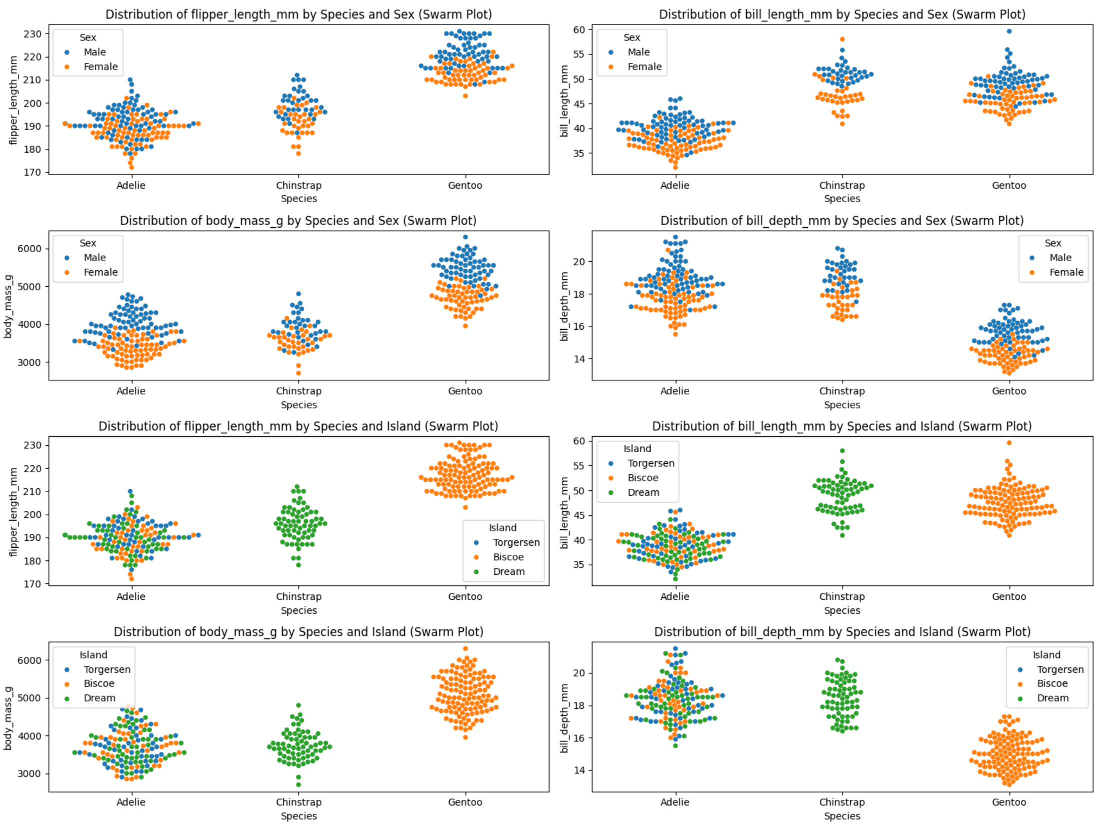
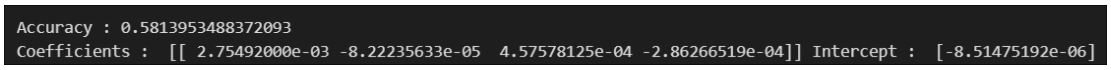


Figure 2: Distribution of feature by Species w. r. t. Island or Sex

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.

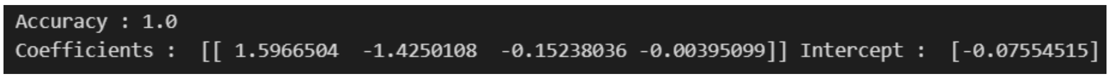
2. **Convergence Issues:** The *saga* solver may struggle with convergence if the features are not properly scaled. Logistic regression assumes features are on a similar scale, so large differences in feature magnitudes can cause the solver to have difficulties. Applying *StandardScaler* can help improve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

1.8 Using `liblinear` as solver

Classification accuracy = 1.0

```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 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.

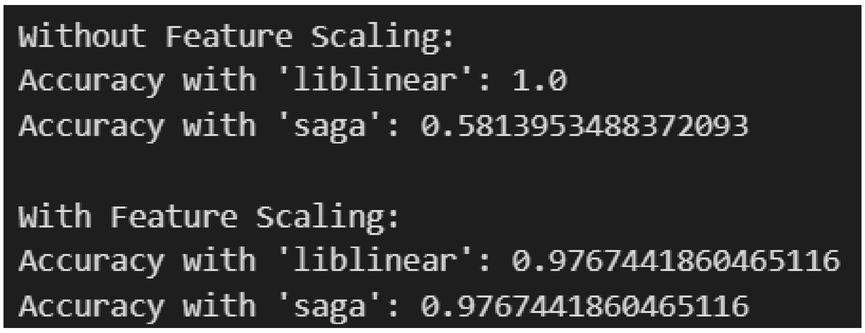
3. **Regularization Handling:** `liblinear` supports both L1 (lasso) and L2 (ridge) regularization and balances model complexity well, preventing over fitting on smaller datasets. `saga`, though it supports regularization, may not handle it as effectively on smaller data.

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.

```python
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

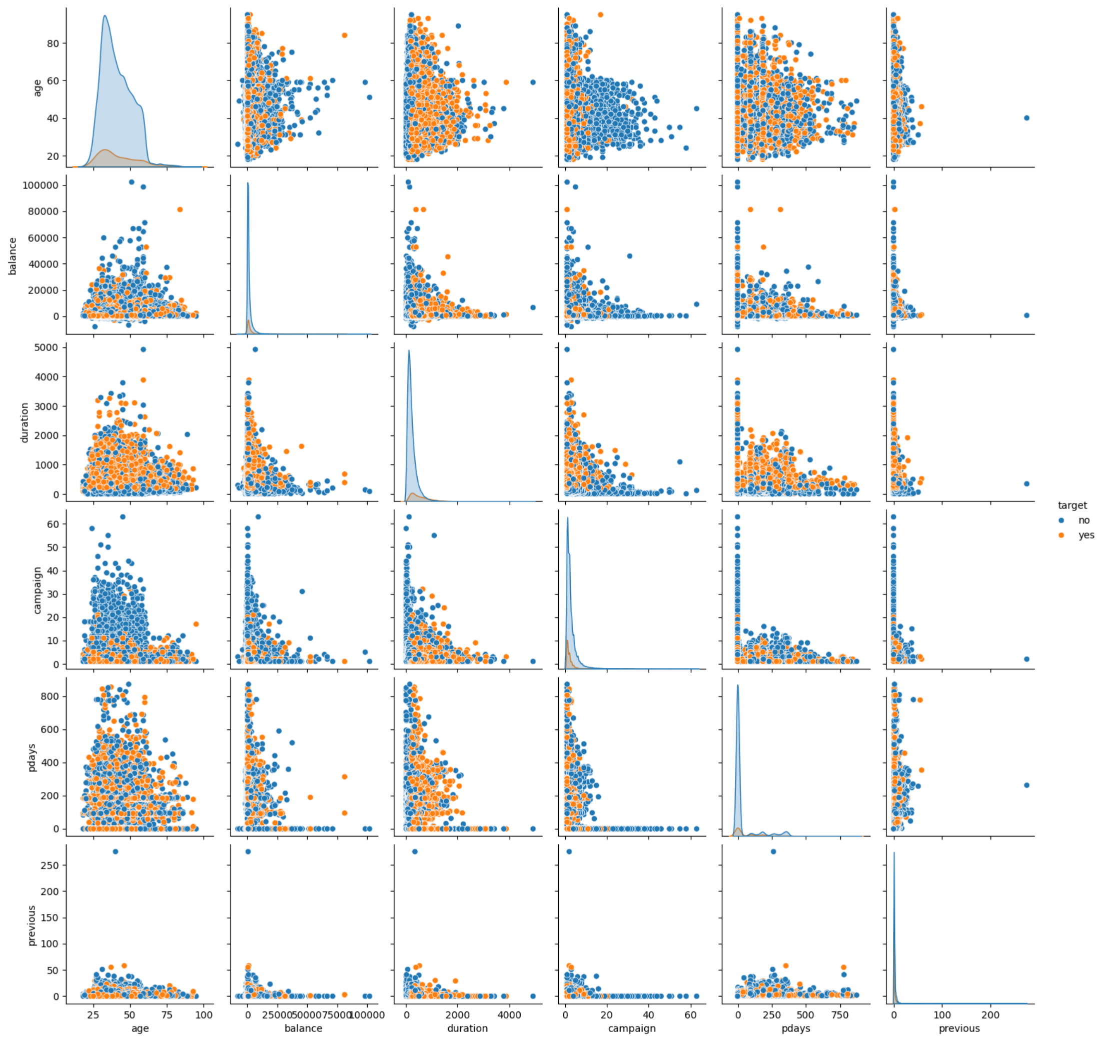


Figure 6: Pair plot

```
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

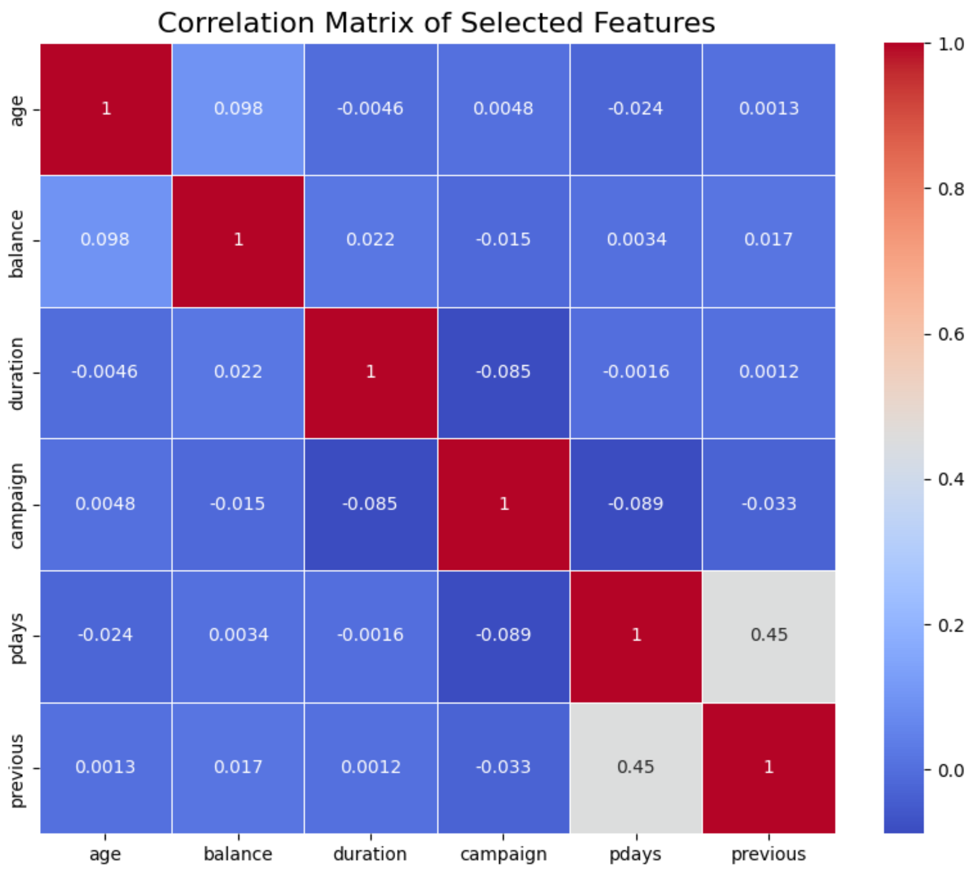


Figure 7: Correlation Matrix

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 )
```

```python

    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

```python
    # 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

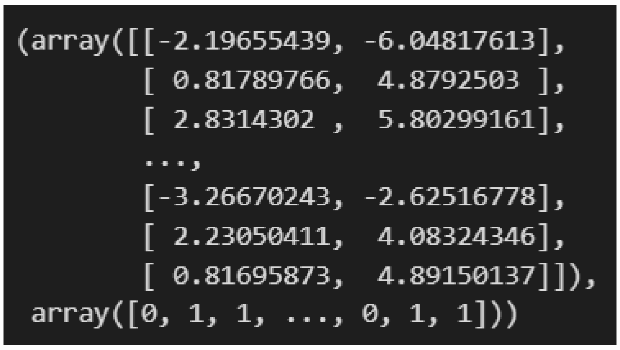
P-values for the predictors:

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
import matplotlib.pyplot as plt

# Sigmoid activation function for logistic regression
def sigmoid_function(z):
    return 1 / (1 + np.exp(-z))

# Function to calculate the logistic regression cost (cross-
   entropy)
def calculate_cost(X, y, weights):
    m = len(y)
    predictions = sigmoid_function(np.dot(X, weights))
    cost = (-1 / m) * (np.dot(y, np.log(predictions)) + np.dot((1
        - y), np.log(1 - predictions)))
    return cost

# Batch Gradient Descent algorithm implementation
def perform_batch_gradient_descent(X, y, weights, learning_rate,
   num_iterations):
    m = len(y)
    cost_history = []
    for iteration in range(num_iterations):
        predictions = sigmoid_function(np.dot(X, weights))
        gradients = (1 / m) * np.dot(X.T, (predictions - y))
        weights -= learning_rate * gradients  # Update weights
        cost = calculate_cost(X, y, weights)
        cost_history.append(cost)
        if iteration % 1 == 0:  # Print cost at every iteration
            print(f"Iteration {iteration}: Cost = {cost:.4f}")
    return weights, cost_history

# Adding intercept (bias term) to the feature matrix
num_samples, num_features = X.shape
X_with_intercept = np.c_[np.ones(num_samples), X]  # Add a column
    of ones for intercept

# Initialize weight coefficients (theta) to zeros
weights = np.zeros(num_features + 1)  # +1 for the intercept term

# Hyperparameters
learning_rate = 0.01
num_iterations = 20

# Execute batch gradient descent
weights, cost_history = perform_batch_gradient_descent(
   X_with_intercept, y, weights, learning_rate, num_iterations)

# Plotting the cost function over iterations
plt.plot(range(num_iterations), cost_history, color='blue')
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.title("Cost Reduction in Batch Gradient Descent")
plt.grid()
plt.show()

# Display final weight values
```

```
    print("Final weights (theta):", weights)
```

Listing 8: Batch gradient descent code

3.2.1 Weight Initialization Method

In this implementation, the weights are initialized to zeros, expressed as:

$$W = \text{np.zeros}(X.\text{shape}[1])$$

This approach is simple and ensures that the model starts with neutral values. While initializing to zeros is effective here, it's important to note that this is particularly suitable for logistic regression, where the learning process is more robust to different weight initialization choices.

3.2.2 Reason for the Selection

- **Simplicity and Efficiency:** Initializing with zeros avoids adding unnecessary complexity to the model. Since logistic regression optimizes based on gradient descent, the algorithm will still adjust the weights appropriately over iterations, even when starting from zeros.
- **Convergence Stability:** With a small dataset and basic gradient descent, initializing with zeros provides stable convergence without encountering large gradient issues that can happen with larger, randomized values.
- **Avoiding Bias in Logistic Regression:** In logistic regression, symmetry isn't as critical as in neural networks. Thus, starting with zeros does not result in all features learning the same information, making it a safe choice for this task.

3.3 Loss Function

The loss function employed in this implementation is the Logistic Loss, also known as Binary Cross-Entropy, defined mathematically as:

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

Where:

- $\hat{y}_i$ represents the predicted probability of the positive class (ranging from 0 to 1),
- y_i denotes the actual binary target (either 0 or 1),
- n is the total number of data points.

Reason for Selection

- **Suitability for Classification Tasks:** This model addresses a binary classification problem where the goal is to predict the probabilities for two distinct classes. Logistic Loss is well-suited for such scenarios as it effectively penalizes incorrect classifications.
- **Output as Probabilities:** Logistic Loss guarantees that the model outputs a probability distribution, enhancing the decision-making process regarding class thresholds.
- **Emphasis on Confident Predictions:** This loss function imposes a heavier penalty on predictions that are both confident and incorrect. This characteristic encourages the model to be cautious in making high-probability predictions for the wrong classes.

- **Differentiability:** Logistic Loss is smooth and differentiable, making it appropriate for gradient-based optimization methods, which facilitates efficient model training.

3.4 Loss plot

```python
# Plotting the cost function over iterations
plt.plot(range(num_iterations), cost_history, color='blue')
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.title("Cost Reduction in Batch Gradient Descent")
plt.grid()
plt.show()

# Display final weight values
print("Final weights (theta):", weights)
```

Listing 9: Plotting the loss

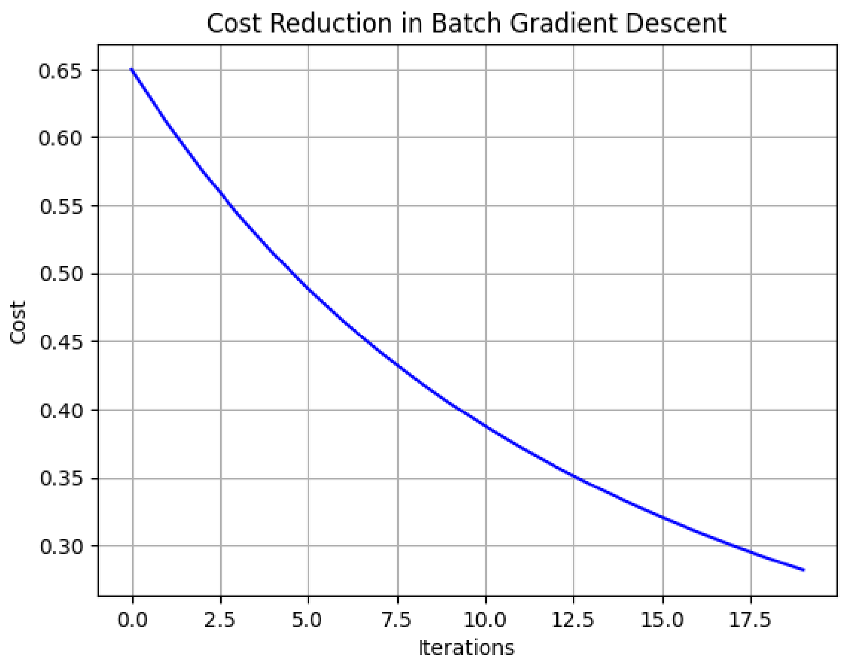


Figure 9: Plotting the loss

3.5 Stochastic Gradient Descent

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

    # Cost function for logistic regression (cross-entropy)
    def calculate_cost(X, y, weights):
        m = len(y)
        predictions = sigmoid_function(np.dot(X, weights))
        cost = (-1 / m) * (np.dot(y, np.log(predictions)) + np.dot((1 - y), np.log(1 - predictions)))
        return cost

    # Stochastic Gradient Descent implementation
    def stochastic_gradient_descent(X, y, weights, learning_rate, num_iterations):
        m = len(y)
        cost_history = []

        for iteration in range(num_iterations):
            total_cost = 0
            for _ in range(m):
                # Randomly select an index
                random_index = np.random.randint(m)
                X_sample = X[random_index].reshape(1, -1)  # Reshape to (1, n)
                y_sample = y[random_index].reshape(1)

                # Make a prediction
                prediction = sigmoid_function(np.dot(X_sample, weights))

                # Calculate the gradient for this sample
                gradient = np.dot(X_sample.T, (prediction - y_sample))
                weights -= learning_rate * gradient

                # Update total cost with the cost for the selected sample
                total_cost += calculate_cost(X_sample, y_sample, weights)

            # Compute and record the average cost for the current iteration
            average_cost = total_cost / m
            cost_history.append(average_cost)
            print(f"Iteration {iteration + 1}: Cost = {average_cost}")

        return weights, cost_history

    # Hyperparameters for SGD
    weights_sgd = np.zeros(n + 1)  # Initialize weights for SGD
    learning_rate_sgd = 0.01
    num_iterations_sgd = 20

    # Execute Stochastic Gradient Descent
    weights_sgd, sgd_cost_history = stochastic_gradient_descent(X, y,
        weights_sgd, learning_rate_sgd, num_iterations_sgd)

    # Plot the cost function over iterations for SGD
    plt.plot(range(num_iterations_sgd), sgd_cost_history, 'r-')
    plt.xlabel("Iterations")
    plt.ylabel("Cost")
    plt.title("SGD: Cost vs Iterations")
    plt.show()

    # Display the final weights after SGD
    print("Final weights (theta) after Stochastic Gradient Descent:",
        weights_sgd)
```

Listing 10: Stochastic gradient descent

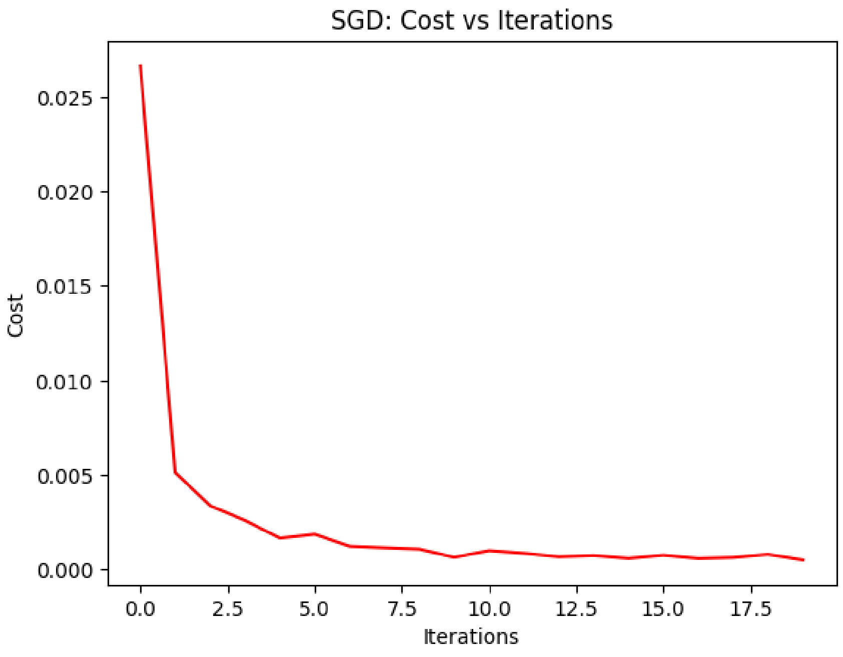


Figure 10: Plotting the loss

The Stochastic Gradient Descent (SGD) approach effectively updates the model weights using individual data points, leading to a more dynamic learning process. As observed in the plotted loss values, the model demonstrates a decreasing trend in loss over iterations, indicating improved accuracy and convergence towards optimal weights.

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

Newton's method is an iterative optimization technique that utilizes both the first and second derivatives of the loss function to update the model weights. It aims to find the optimal parameters more efficiently than gradient descent by incorporating the Hessian matrix, which represents the curvature of the loss function. The update rule is given by:

$$W_{new} = W_{old} - H^{-1} \cdot \nabla L$$

where H is the Hessian matrix, and ∇L is the gradient of the loss function. This method often converges faster than first-order methods, particularly near the optimum, but it requires the computation of the Hessian, which can be computationally expensive for large datasets.

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

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

# Logistic regression cost function with epsilon correction
def compute_cost(X, y, theta, epsilon=1e-15):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    h = np.clip(h, epsilon, 1 - epsilon)  # Clipping to prevent
        log(0)
    cost = (-1 / m) * (np.dot(y, np.log(h)) + np.dot((1 - y), np.
        log(1 - h)))
    return cost

# Gradient computation of the cost function
def compute_gradient(X, y, theta):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    gradient = (1 / m) * np.dot(X.T, (h - y))
    return gradient

# Hessian matrix computation (second derivative of the cost
   function)
def compute_hessian(X, theta):
    m = len(y)
    h = sigmoid(np.dot(X, theta))
    diag = np.diag(h * (1 - h))  # Diagonal matrix with h * (1 -
        h)
    H = (1 / m) * np.dot(np.dot(X.T, diag), X)
    return H

# Newton's Method implementation
def newtons_method(X, y, theta, iterations):
    costs = []
    for i in range(iterations):
        gradient = compute_gradient(X, y, theta)
        hessian = compute_hessian(X, theta)

        # Update weights using Newton's method formula
        theta -= np.linalg.inv(hessian).dot(gradient)

        # Calculate the cost for the current iteration
        cost = compute_cost(X, y, theta)
        costs.append(cost)
        print(f"Iteration {i + 1}: Cost = {cost:.4f}")

    return theta, costs

# Set a random seed for reproducibility
np.random.seed(0)

# Define centers for two classes
centers = [[-5, 0], [5, 1.5]]

# Generate synthetic dataset with two clusters
X, y = make_blobs(n_samples=2000, centers=centers, random_state
    =5)

# Transform data to introduce linear inseparability
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)

# Add intercept (bias term) to the feature matrix
m, n = X.shape
X = np.c_[np.ones(m), X]  # Add a column of ones for the
    intercept term

# Initialize weights (theta) to zero
theta = np.zeros(n + 1)  # n + 1 for the intercept

# Hyperparameters for Newton's method
iterations_newton = 20

# Execute Newton's method
theta_newton, newton_costs = newtons_method(X, y, theta,
    iterations_newton)

# Plot cost function against iterations for Newton's method
plt.plot(range(iterations_newton), newton_costs, 'g-')
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.title("Newton's Method: Cost vs Iterations")
plt.grid(True)
plt.show()
```

```
# Output final weights (theta) from Newton's method
print("Final weights (theta) using Newton's method:",
   theta_newton)
```

Listing 11: Implementation of Newton's method

3.7 Loss plot

```
# Plot cost function against iterations for Newton's method
plt.plot(range(iterations_newton), newton_costs, 'g-')
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.title("Newton's Method: Cost vs Iterations")
plt.grid(True)
plt.show()

# Output final weights (theta) from Newton's method
print("Final weights (theta) using Newton's method:",
   theta_newton)
```

Listing 12: Plotting the loss

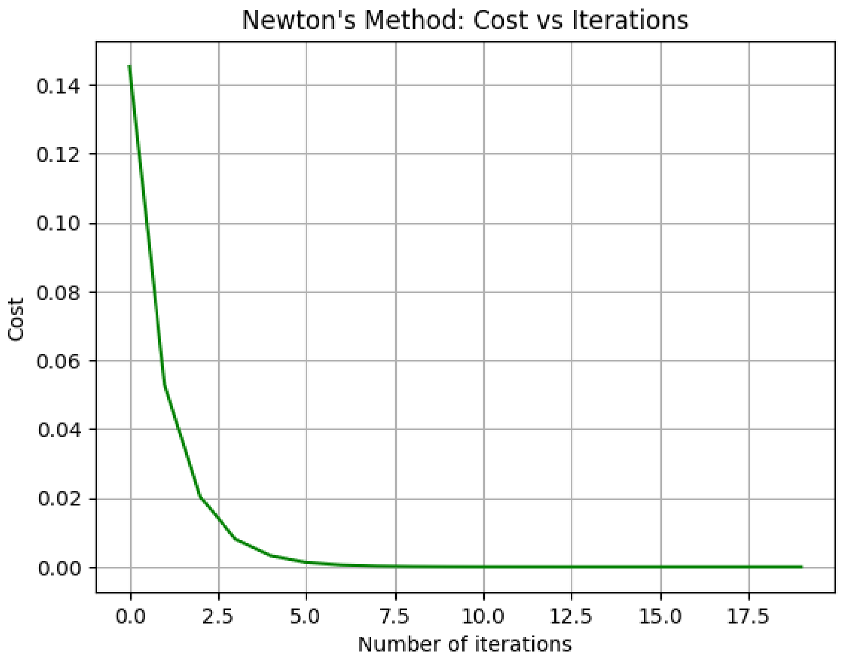


Figure 11: Plotting the loss

3.8 Plot in the same graph

```
# Plot the cost functions for BGD, SGD, and Newton's Method in a
   single plot
plt.plot(range(num_iterations), cost_history, 'b-', label='Batch
   Gradient Descent')
```

```
    plt.plot(range(num_iterations_sgd), sgd_cost_history, 'r-', label
        ='Stochastic Gradient Descent')
    plt.plot(range(iterations_newton), newton_costs, 'g-', label="
        Newton's Method")

    plt.ylim([0, max(max(cost_history), max(sgd_cost_history), max(
        newton_costs)) * 1.1])  # Slightly larger than max value to
        prevent cropping
    plt.xlim([0, iterations])

    plt.xlabel("Number of iterations")
    plt.ylabel("Cost")
    plt.title("Cost vs Iterations for BGD, SGD, and Newton's Method")

    # Display legend
    plt.legend()

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

Listing 13: Plotting in single graph

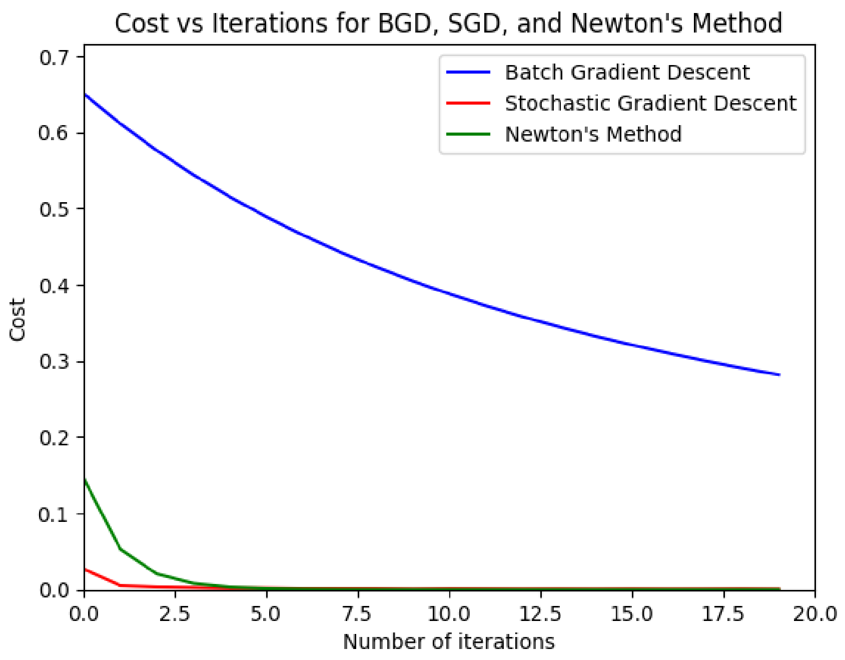


Figure 12: Plot

3.8.1 Results and Discussion

The performance of three optimization methods—Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Newton's Method—was evaluated by tracking the loss during training. The following observations were made based on the loss reduction over iterations.

- **Batch Gradient Descent (BGD):** The blue curve shows that BGD converges steadily but at a slower rate compared to the other methods. This behavior is expected, as BGD updates the weights using all samples, making it computationally intensive and slower to converge. While effective, it requires more iterations to achieve a sufficiently low loss.

- **Stochastic Gradient Descent (SGD):** The red curve demonstrates the rapid convergence of SGD within the first few iterations. However, as anticipated, the cost function fluctuates slightly due to SGD updating the weights for each sample individually. This introduces more variance during the updates, leading to a less stable convergence path but generally faster overall progress toward an optimal solution.
- **Newton's Method:** The green curve indicates that Newton's Method converges the fastest among the three methods. It reaches a low cost within just a few iterations, highlighting its efficiency in solving optimization problems with fewer iterations. The use of second-order information allows for more informed weight updates, which accelerates the convergence process significantly.

3.8.2 Conclusion

- **Efficiency:** Newton's method is the most efficient regarding the number of iterations needed to minimize the cost function. It converges significantly faster than both BGD and SGD, making it a powerful choice for smaller datasets.
- **Stability:** Batch Gradient Descent is the most stable method due to its consistent updates across the entire dataset. While SGD converges quickly, it shows more fluctuations in loss values. Newton's method balances efficiency and stability effectively.

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

3.9.1 Gradient Descent – Convergence Criteria

In Gradient Descent, determining the number of iterations can rely on specific convergence criteria. Two commonly employed approaches include:

1. **Cost Function Convergence:** The iterations can be halted when the absolute difference between the cost function values of two consecutive iterations falls below a predetermined threshold ϵ:
$$|J(\theta^{(k+1)}) - J(\theta^{(k)})| < \epsilon \tag{1}$$
where:
   - $J(\theta)$ represents the cost function.
   - ϵ is a small value, typically set to 10^{-5} or smaller, indicating convergence.
   - k denotes the iteration number.

   This approach ensures the algorithm stops when the cost function stabilizes, thereby avoiding unnecessary iterations after reaching a minimum.
2. **Gradient Norm:** Another method involves monitoring the norm of the gradient $\|\nabla J(\theta)\|$. When the gradient approaches zero, it signals that the cost function is likely at a local minimum:
$$\|\nabla J(\theta)\| < \epsilon \tag{2}$$
This condition is particularly useful when the cost function flattens out, indicating that additional iterations may not significantly enhance the solution.

3.9.2 Newton's Method – Hessian Condition

In Newton's Method, the number of iterations can be determined by assessing the condition of the Hessian matrix. The Hessian matrix, which contains the second-order partial derivatives of the cost function, is critical for updating the weights. Two effective strategies include:

1. **Condition Number:** A key approach is to monitor the condition number of the Hessian matrix H, which indicates how sensitive the solution is to minor changes in the input. The condition number is defined as:
$$\kappa(H) = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}} \tag{3}$$
where:
    - λ_{max} is the largest eigenvalue of the Hessian matrix.
    - λ_{min} is the smallest eigenvalue of the Hessian matrix.

    When $\kappa(H)$ becomes substantially large (e.g., $\kappa(H) > 10^6$), it suggests that the Hessian is nearly singular (ill-conditioned), at which point further iterations may not yield meaningful updates. Thus, iterations should be terminated.

2. **Gradient Norm:** Similar to Gradient Descent, Newton's Method can also utilize the norm of the gradient to determine when to cease iterations. If the gradient norm $\|\nabla J(\theta)\|$ becomes very small, it indicates that the cost function is close to a minimum, allowing for the optimization process to be concluded:
$$\|\nabla J(\theta)\| < \epsilon \tag{4}$$
This ensures the method stops when it is sufficiently close to convergence, thereby avoiding unnecessary computations.

3.10 Convergence behavior of the algorithm with updated data

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

# Set random seed for reproducibility
np.random.seed(0)

# Define the updated centers of the two classes
centers = [[3, 0], [5, 1.5]]

# Generate synthetic data with two clusters (classes)
X, y = make_blobs(n_samples=2000, centers=centers, random_state
    =5)

# Apply a transformation to introduce some linear inseparability
    in the data
transformation = [[0.5, 0.5], [-0.5, 1.5]]
X = np.dot(X, transformation)

# Visualize the data to understand the distribution
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', edgecolor='k')
plt.title('Generated Data (New Centers)')
```

```python
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()

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

    # Logistic regression cost function (cross-entropy)
    def compute_cost(X, y, theta):
        m = len(y)
        h = sigmoid(np.dot(X, theta))
        cost = (-1 / m) * (np.dot(y, np.log(h)) + np.dot((1 - y), np.log(1 - h)))
        return cost

    # Batch Gradient Descent function
    def batch_gradient_descent(X, y, theta, learning_rate, iterations):
        m = len(y)
        costs = []
        for i in range(iterations):
            h = sigmoid(np.dot(X, theta))
            gradient = (1 / m) * np.dot(X.T, (h - y))
            theta = theta - learning_rate * gradient
            cost = compute_cost(X, y, theta)
            costs.append(cost)
            if i % 1 == 0:  # Print every iteration
                print(f"Iteration {i}: Cost = {cost}")
        return theta, costs

    # Add intercept (bias term) to the features
    m, n = X.shape
    X = np.c_[np.ones(m), X]  # Add a column of ones for the intercept term

    # Initialize weights (theta) to zero
    theta = np.zeros(n + 1)  # n + 1 because of the intercept term

    # Hyperparameters
    learning_rate = 0.01
    iterations = 20

    # Perform batch gradient descent
    theta, costs = batch_gradient_descent(X, y, theta, learning_rate, iterations)

    # Plot the cost function with respect to iterations
    plt.plot(range(iterations), costs, 'b-')
    plt.xlabel("Number of iterations")
    plt.ylabel("Cost")
    plt.title("Batch Gradient Descent (New Centers): Cost vs
       Iterations")
    plt.show()

    # Print final weights (theta)
    print("Final weights (theta):", theta)
```

Listing 14: Generate the new dataset and perform BGD to update the weights

3.10.1 Generated data

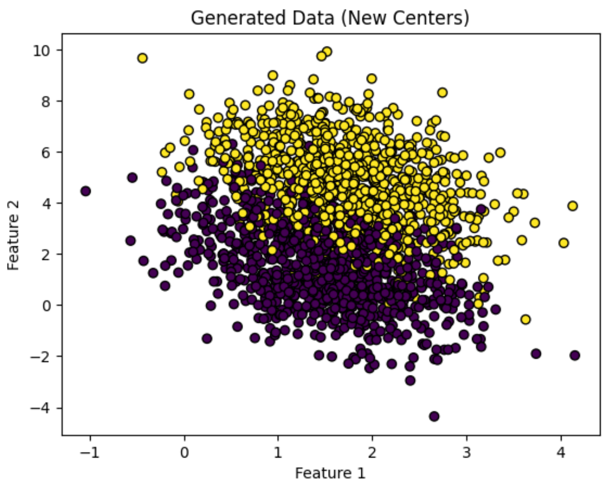


Figure 13: Generated data

3.10.2 Cost over iterations

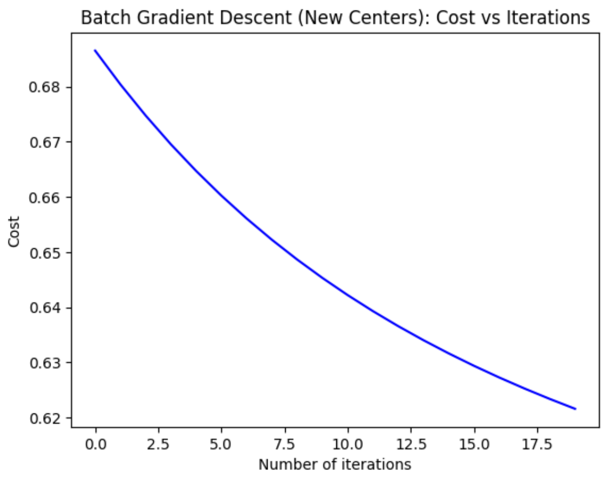
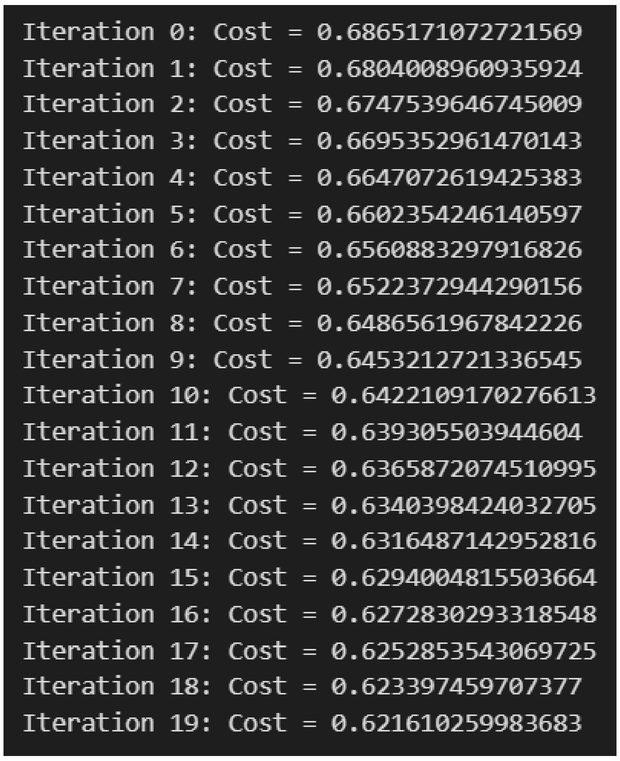


Figure 14: Batch Gradient Descent (New Centers): Cost vs Iterations

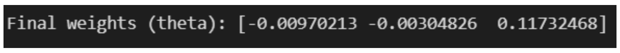
3.10.3 Behavior of convergence



```
Iteration 0: Cost = 0.6865171072721569
Iteration 1: Cost = 0.6804008960935924
Iteration 2: Cost = 0.6747539646745009
Iteration 3: Cost = 0.6695352961470143
Iteration 4: Cost = 0.6647072619425383
Iteration 5: Cost = 0.6602354246140597
Iteration 6: Cost = 0.6560883297916826
Iteration 7: Cost = 0.6522372944290156
Iteration 8: Cost = 0.6486561967842226
Iteration 9: Cost = 0.6453212721336545
Iteration 10: Cost = 0.6422109170276613
Iteration 11: Cost = 0.639305503944604
Iteration 12: Cost = 0.6365872074510995
Iteration 13: Cost = 0.6340398424032705
Iteration 14: Cost = 0.6316487142952816
Iteration 15: Cost = 0.6294004815503664
Iteration 16: Cost = 0.6272830293318548
Iteration 17: Cost = 0.6252853543069725
Iteration 18: Cost = 0.623397459707377
Iteration 19: Cost = 0.621610259983683
```

Figure 15: Loss over iterations

3.10.4 Final Weights



```
Final weights (theta): [-0.00970213 -0.00304826  0.11732468]
```

Figure 16: Final weights after 20 iterations

3.10.5 Analysis of Convergence Behavior

The loss values for the centers are as follows:

- **Loss for the initial center** $[[-5, 0], [5, 1.5]] = 0.2816$
- **Loss for the updated center** $[[3, 0], [5, 1.5]] = 0.6216$

Loss Reduction Over Iterations The updated center exhibits a higher final loss of 0.6216 compared to the initial center's lower final loss of 0.2816. This indicates that the dataset associated with the updated center is more challenging for the logistic regression model to fit effectively.

Explanation for Convergence Behavior

- The higher final loss for the updated center implies that the data distribution corresponding to the coordinates $[[3, 0], [5, 1.5]]$ contains complexities that hinder optimization. Factors such as overlapping data points or non-linear relationships in the dataset may contribute to this difficulty.

- In contrast, the initial center's lower loss suggests that its data distribution is more manageable for the gradient descent algorithm, allowing it to minimize error effectively and converge more rapidly.
- **Data Overlap:** The updated centers have caused the data clusters to overlap more, making the classification task harder. The increased overlap between classes leads to smaller gradient updates as the algorithm struggles to find the decision boundary that separates the classes well.
- **Transformation Impact:** The linear transformation applied to introduce some inseparability might have further complicated the optimization process.

Weight Changes Over Iterations

- The weights for the updated center stabilize more slowly than those for the initial center, reflecting the higher final loss and suggesting that the model is struggling to converge on optimal weights.
- This slower adjustment indicates that the updated center may require more iterations to achieve a satisfactory fit, possibly due to the complexities in its underlying data structure.

Comparing Convergence Behavior with Final Loss Values

- **Initial Center** $[[-5, 0], [5, 1.5]]$:
  - Final Loss: 0.2816
  - The model experiences faster convergence and a lower final loss, indicating a simpler data distribution.
- **Updated Center** $[[3, 0], [5, 1.5]]$:
  - Final Loss: 0.6216
  - The model converges more slowly, reflecting challenges in optimizing a more complex data distribution.

In summary, the updated center's higher final loss underscores the difficulties in fitting the logistic regression model to its dataset, while the initial center demonstrates a more favorable convergence behavior, making it easier for the model to optimize effectively.