BSc (Hons) Artificial Intelligence and Data Science

Module: CM2601 Machine Learning

Machine Learning Report

Module Leader: Mr. Sahan Priyanaya

RGU Student ID : 2330923

IIT Student ID : 20230976

Student Name : Abdullah Nazly

Acknowledgment .

I want to extend my sincere appreciation to all those who contributed and provided steadfast support throughout this coursework. A special acknowledgment goes to Mr. Sahan Priyanaya, the module leader, for his guidance that led to the successful completion of this coursework. Additionally, I express extra gratitude to Mr. Pushpika Prasad Liyanaarachchi assistant lecturer, for assisting throughout the tutorials and addressing doubts, which facilitated the smooth execution of the coursework. I genuinely appreciate these individuals for their assistance in the successful completion of the coursework.

Executive Summary

This course work aims to build two machine learning models using python. Implementation is focused on understanding the key concepts of data preprocessing and training a model. A bank dataset is used to train the model here. Two models were used. One is a neural network model, and the other is a Random Forest model. For both models, the same way of data preprocessing is done. Though both got good predicting accuracy.

Contents

Table of figures

1. Introduction

The Machine Learning course work is processed on a simple approach. Starting with data preprocessing and going up to training the model. First the data is processed accordingly to train the neural network model. First the null values checked, the duplicates checked, and problems were solved. Then unique values were checked to make decisions on encoding the categorical variables. Outliers and extreme values were checked for numerical values and no problems were found. Did EDA (Exploitary Data Analysis) for feature selection. For some features we must consider are influencing the target variable or not. No standardizations were made to test whether this processed data is enough to train the model. As the test the model showed a great performance to both models. The conclusion was made to keep the data as it is without further standardization and improvements. Finally, the models went through a hyper parameter tuning and evaluated.

2. Neural Network Model

2.1. Data Preprocessing

Bank detail dataset is used here to train the model. As always the dataset has to be preprocessed. The process starts with importing the library to manipulate dataset which is pandas. This dataset is a little different in csv format. Usually, it will be comma separated file. But in this dataset values are separated by semicolons.

```
"age";"job";"marital";"education";"default";"balance";"housing";"loan";"contact";"day
";"month";"duration";"campaign";"pdays";"previous";"poutcome";"y"
58;"management";"married";"tertiary";"no";2143;"yes";"no";"unknown";5;"may";261;1;-
1;0;"unknown";"no"
44;"technician";"single";"secondary";"no";29;"yes";"no";"unknown";5;"may";151;1;-
1;0;"unknown";"no"
33;"entrepreneur";"married";"secondary";"no";2;"yes";"yes";"unknown";5;"may";76;1;-
1;0;"unknown";"no"
47;"blue-collar";"married";"unknown";"no";1506;"yes";"no";"unknown";5;"may";92;1;-
1;0;"unknown";"no"
```

So as a result. When we try to read it with normal csv format, it won't be able to read properly. Therefor another parameter is passed to divide the semicolons and read the csv file properly

```
df = pd.read_csv('./Dataset/bank-full.csv', delimiter=';')
```

```
df.head(20)
```

Then we get the head output as usual.

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	no
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	no
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	no
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	no
5	35	management	married	tertiary	no	231	yes	no	unknown	5	may	139	1	-1	0	unknown	no

Figure 1. head

If we didn't use the delimiter parameter, the output would be like this.

	age;"job";"marital";"education";"default";"balance";"housing";"loan";"contact";"day";"month";"duration";"campaign";"pdays";"previous";"poutcome";"y"
0	58;"management";"married";"tertiary";"no";2143...
1	44;"technician";"single";"secondary";"no";29;"...
2	33;"entrepreneur";"married";"secondary";"no";2...
3	47;"blue-collar";"married";"unknown";"no";1506...
4	33;"unknown";"single";"unknown";"no";1;"no";"n...
5	35;"management";"married";"tertiary";"no";231;...

Figure 2. head incorrect format

2.2. Checking NULL values and duplicates.

The next step is to check whether there's any null values or duplicate values in the dataset. This is an important step to be considered before training the model. Checking the null values is performed using the below code which it gets the null values for every column in the dataset.

```
print(df.isnull().sum())
```

The output I got gave me the result that it doesn't have any null value.

```
age          0
job          0
marital      0
education    0
default      0
balance      0
housing      0
loan         0
contact      0
day          0
month        0
duration     0
campaign     0
pdays        0
previous     0
poutcome     0
y            0
dtype: int64
```

Figure 3. null value

Next identifying the duplicate value is performed in the code. In this code also. The duplication is checked for all the columns at once by using this code.

```
print(df.duplicated().sum())
```

The result was zero, which means there's no duplicate value either. As a result, the part of data preprocessing becomes easier. We don't have to handle missing values or duplicate values. We can directly go for the other areas of data preprocessing.

2.3. Data format analysis.

The usual format of checking for format is performed here, with .info() code the Dtypes including null counts are displayed.

```
df.info()
```

```
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 17 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        45211 non-null  int64
 1   job        45211 non-null  object
 2   marital    45211 non-null  object
 3   education  45211 non-null  object
 4   default    45211 non-null  object
 5   balance    45211 non-null  int64
 6   housing    45211 non-null  object
 7   loan       45211 non-null  object
 8   contact    45211 non-null  object
 9   day        45211 non-null  int64
 10  month      45211 non-null  object
 11  duration   45211 non-null  int64
 12  campaign   45211 non-null  int64
 13  pdays      45211 non-null  int64
 14  previous   45211 non-null  int64
 15  poutcome   45211 non-null  object
 16  y          45211 non-null  object
dtypes: int64(7), object(10)
```

Figure 4. data type

Then we get a statistic of the features to analyze the mean values, min-max values and other values as well.

```
df.describe()
```

	age	balance	day	duration	campaign	pdays	previous
count	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000
mean	40.936210	1362.272058	15.806419	258.163080	2.763841	40.197828	0.580323
std	10.618762	3044.765829	8.322476	257.527812	3.098021	100.128746	2.303441
min	18.000000	-8019.000000	1.000000	0.000000	1.000000	-1.000000	0.000000
25%	33.000000	72.000000	8.000000	103.000000	1.000000	-1.000000	0.000000
50%	39.000000	448.000000	16.000000	180.000000	2.000000	-1.000000	0.000000
75%	48.000000	1428.000000	21.000000	319.000000	3.000000	-1.000000	0.000000
max	95.000000	102127.000000	31.000000	4918.000000	63.000000	871.000000	275.000000

Figure 5. statistics

By this we can understand that They are acceptable values. Theres no extreme values in these numerical features.

2.4. Getting the values of the features

First the code below is run to get all the numerical features which don't need data transformation currently.

```
numeric_columns = df.select_dtypes(include=['int64', 'float64']).columns
print("Numeric Columns:", numeric_columns)
```

```
Numeric Columns: Index(['age', 'balance', 'day', 'duration', 'campaign', 'pdays', 'previous'], dtype='object')
```

Figure 6. numerics

and the output given out put had all the numerical values columns. Next to get all the categorical valued features, the code below is run.

```
categorical_columns = df.select_dtypes(include=['object']).columns
print("Categorical Columns:", categorical_columns)
```

The output gave all the columns which has categorical values.

```
Categorical Columns: Index(['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact',
       'month', 'poutcome', 'y'],
      dtype='object')
```

Figure 7. categoricals

Finally, checking whether if any Boolean valued features are available using the below code

```
boolean_columns = df.select_dtypes(include=['bool']).columns
print("Boolean Columns:", boolean_columns)
```

but there weren't any Boolean valued features available.

```
Boolean Columns: Index([], dtype='object')
```

Figure 8. Boolean

2.5. Checking the Unique values for each column

The unique values of each column are printed to make decision on encoding. Only categorical values should be considered under encoding, so categorical valued features are chosen here.

```
df['job'].unique()

df['marital'].unique()

df['education'].unique()

df['default'].unique()

df['housing'].unique()

df['contact'].unique()

df['month'].unique()

df['poutcome'].unique()

df['y'].unique()

df['campaign'].unique()
```

Codes were ran multiple times to identify the unique values of each columns to take the decision for encoding.

2.6. Analyzing feature contact

When we consider the feature contact it seems like y variable doesn't actually influenced by it. To make sure some EDA is done to the feature. We used chi-square test to evaluate whether there is a statistical relationship between two categorical variables. It will help to identify whether the feature is potentially important for the predictive model. This is done by the codes below.

```
from scipy.stats import chi2_contingency

# If p value is < 0.05, the feature is influencing the targeted variable

crosstab = pd.crosstab(df['contact'], df['y'])
chi2, p, dof, expected = chi2_contingency(crosstab)
print("Chi-square statistic:", chi2)
print("p-value:", p)
```

scipy library is imported to do statistical tests for the categorical variables. This is the output that was given to the above code.

```
Chi-square statistic: 1035.714225356292
p-value: 1.2517383253406 38e-225
```

Figure 9. p value contact

As we can see, the p-value is extremely low, and chi-square value is a bit higher. Therefore, the contact feature makes a huge influence in the prediction model.

2.7. One hot encoding contact feature.

As we can see it is a categorical variable, it must be encoded. Since it has no relation between it's values, one hot encoding is the best option. The code below is used to encode the contact feature.

```
# Apply One-Hot Encoding to the 'contact' column
df_encoded = pd.get_dummies(df['contact'], prefix='contact')

# Convert True/False to 1/0
df_encoded = df_encoded.astype(int)

# concatenate the encoded columns with the original dataframe
df = pd.concat([df, df_encoded], axis=1)

# Drop the original 'contact' column
df.drop('contact', axis=1, inplace=True)

df.head()
```

the feature is encoded nicely.

2.8. Analyzing poutcome feature.

The same way of chi-square test is used to evaluate the relationship statistics between target variable and poutcome. This resulted in this feature also deeply influences the target variable as the p-value is exactly equal to zero and chi-square value has a higher value.

```
Chi-square statistic: 4391.5065887686615
p-value: 0.0
```

Figure 10. p value poutcome

2.9. Label Encoding poutcome.

This feature has unique values like success, failure, and other values. So, we can identify a relationship between each other. For example, success > failure. So the best option is to label encoding here. The code below is used to label code the poutcome feature.

```
from sklearn.preprocessing import LabelEncoder
# Initialize the LabelEncoder
label_encoder = LabelEncoder()

# Apply Label Encoding to the 'poutcome' column
df['poutcome_encoded'] = label_encoder.fit_transform(df['poutcome'])

df.drop('poutcome', axis=1, inplace=True)

# Display the resulting DataFrame
print(df)
```

2.10. Encoding Y variable and analyzing numerical columns.

In this step we evaluate the relationship between numerical features and the binary target variable y by using Point-Biserial Correlation. By this we can determine the significance of association between these features. First Label encoding is done for the y variable. Then numerical columns are selected. Then the point-biserial correlation is performed for each feature. Then the result is printed. These are the codes which are used to calculate the correlation.

```
from scipy.stats import pointbiserialr

# Step 1: Convert 'y' to numeric (binary)
df['y'] = df['y'].map({'no': 0, 'yes': 1})

# Step 2: Define numerical features
numerical_features = ['age', 'balance', 'day', 'duration', 'campaign', 'pdays', 'previous']

# Step 3: Calculate Point-Biserial Correlation for each feature
correlation_results = []
for col in numerical_features:
    corr, p_value = pointbiserialr(df[col], df['y'])
    correlation_results.append((col, corr, p_value))

# Step 4: Print results
print("Feature-wise Point-Biserial Correlation and p-values:")
for feature, corr, p_value in correlation_results:
    print(f"Feature: {feature}, Correlation: {corr:.3f}, p-value: {p_value:.3f}")
```

The output we got is below.

```
Feature-wise Point-Biserial Correlation and p-values:
Feature: age, Correlation: 0.025, p-value: 0.000
Feature: balance, Correlation: 0.053, p-value: 0.000
Feature: day, Correlation: -0.028, p-value: 0.000
Feature: duration, Correlation: 0.395, p-value: 0.000
Feature: campaign, Correlation: -0.073, p-value: 0.000
Feature: pdays, Correlation: 0.104, p-value: 0.000
Feature: previous, Correlation: 0.093, p-value: 0.000
```

Figure 11. correlation for numerics

Then for a better visualization the output is plotted in a heatmap. In the heatmap the correlation of each feature with y is plotted.

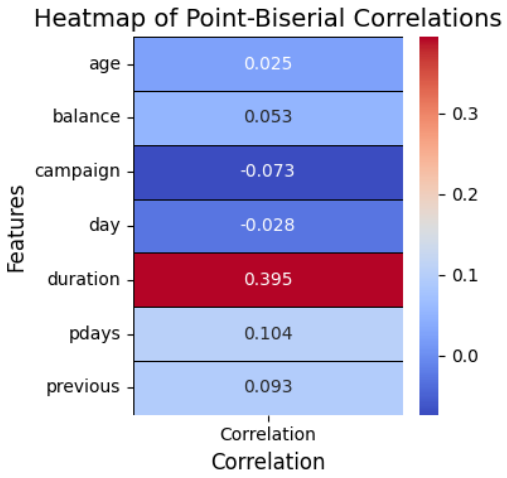

Figure 12. heatmap numerics

As we can see, the day column has a weak negative correlation. So, it might be less meaningful to the model. Age column also has a less correlation value, even though when considering real life scenarios, it may influence prediction. Below output is a Countplot to analyze the relationship between campaign feature and y variable. So, considering the Countplot we assume that there might be a tiny influence on the prediction. Therefore, the campaign feature is kept as it is.

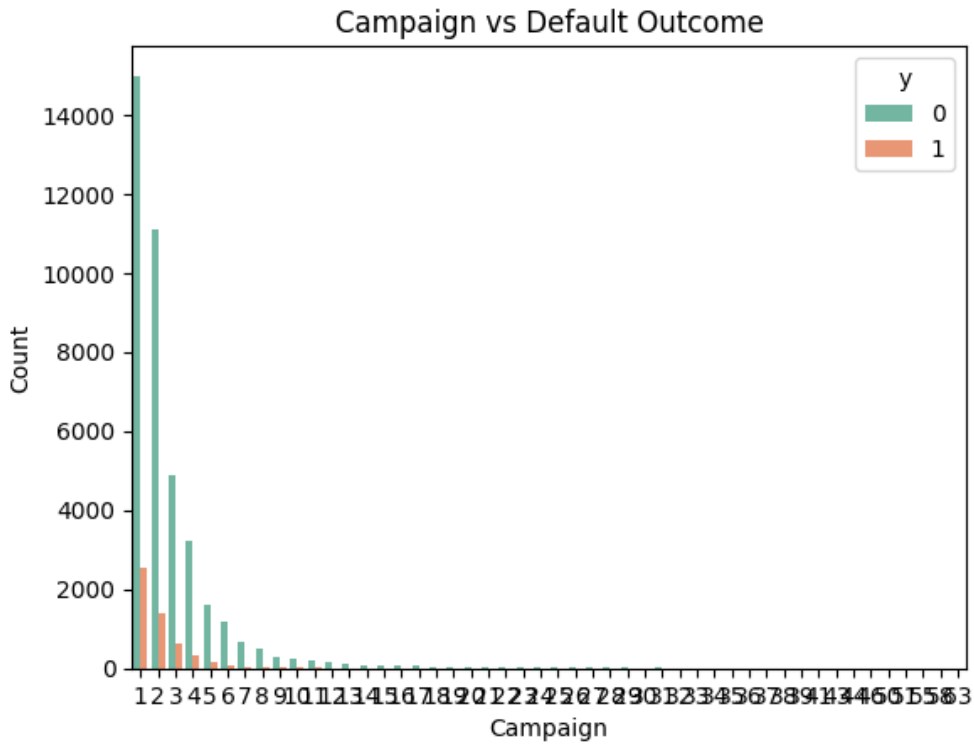

Figure 13. bar plot camapign

2.11. Removing unnecessary columns.

So according to analysis, the month column also can be removed. Because both day and month have similar characteristics. As the day has less influence, the month is also considered to be removed. Columns were dropped using this code

```
df = df.drop(columns=['day', 'month'])
```

Box plotting is performed between numerical values and target values to analyse the extreme values or incorrect values. For every column similar format of this code is performed for plotting.

```
sns.boxplot(x='y', y='duration', data=df)
```

2.12. Plotting on boxplot to identify extreme values and outliers

2.10.1. Plotting duration column.

The result we got looks like this.

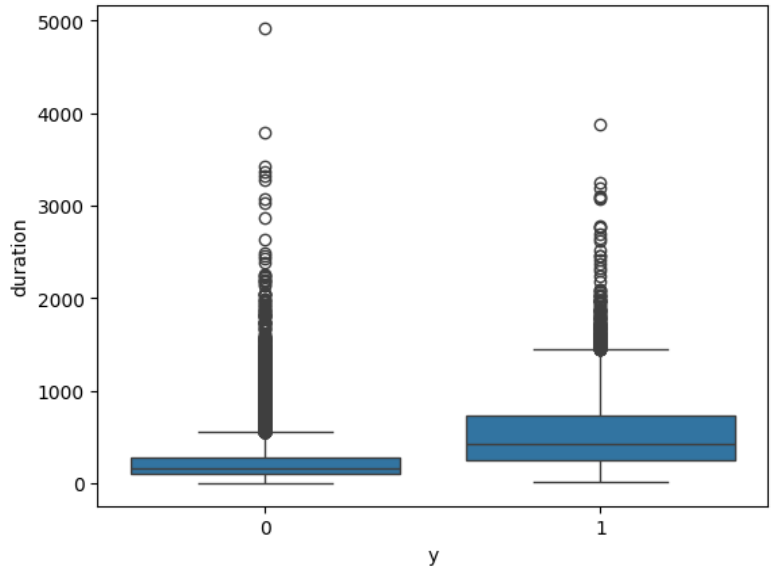

Figure 14. box plot duration

As we can see for the duration there's no negative values which can be incorrect format. So, this feature is good to go.

2.10.2. Plotting age column.

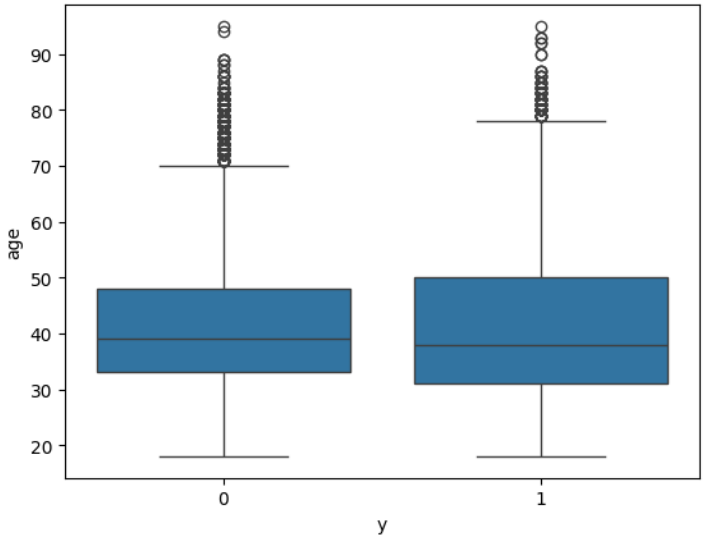

Figure 15. box plot age

As we can see there are some extreme values in the above plotting. Though these values are acceptable. These can happen in real life scenario as well. So, this column is also good to go.

2.10.3. Plotting balance column.

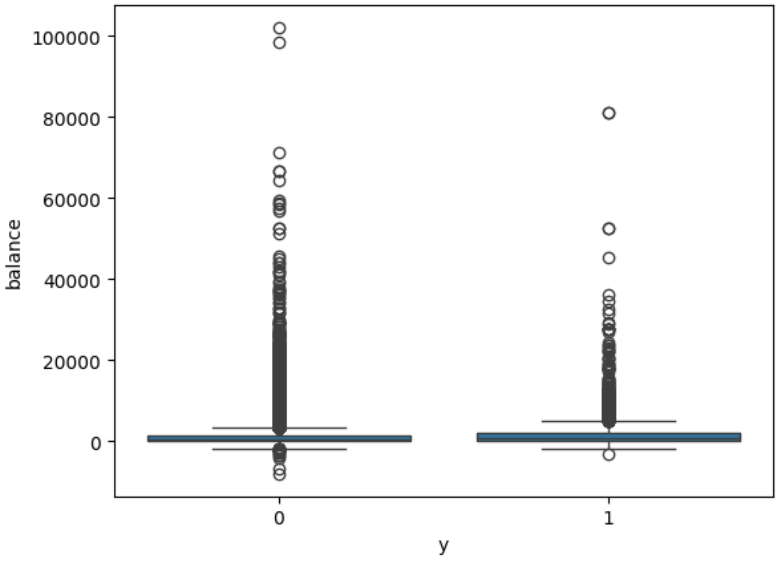

Figure 16. box plot balance

As we can see from the plotting that there are some extreme values. But those can rarely happen. There are negative values as well. These could give a false assumption that there cannot be

negative values for balance. But we can assume that the client could be under credit rather than having balance. Therefore, there is no need to do cleanup or transformation here.

2.12.4. Plotting previous column.

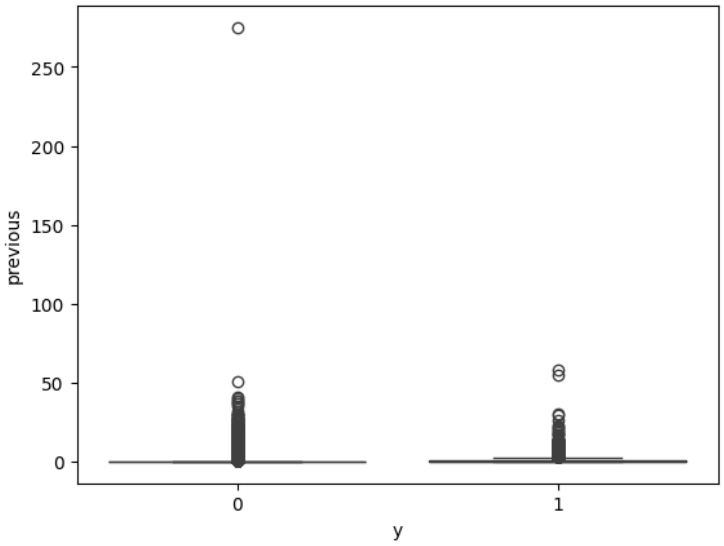

Figure 17. box plot previous

As we can see from the plotting there is one particular data point which is unusually extreme. It's better to remove that record. It is performed by the code below.

```
# Identify the record with the extreme value in 'previous'
outlier_row = df[df['previous'] > 250]

# Display the details of the record
print("Outlier row details:")
print(outlier_row)

# Drop the specific row
df = df.drop(outlier_row.index)

# Verify the row is removed
print(f"Updated dataset shape: {df.shape}")
```

2.12.5. Analysing pdays.

First the box plotting is done to identify extreme values.

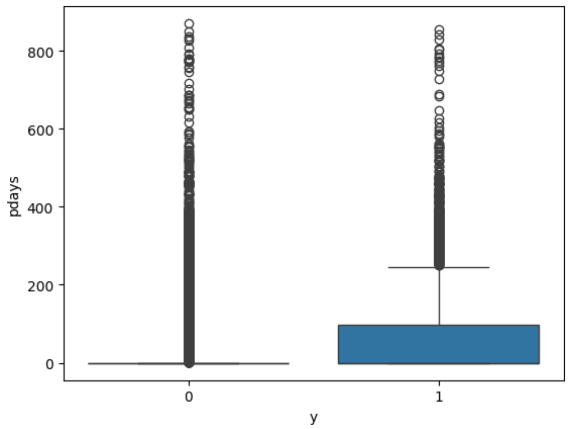

Figure 18. boxplot pdays

Theres negative value -1 in this feature which represents the client has not contacted previously. This must be handled. If not, the model may get false information. first buckets are created to put a range for the amount of calls. By this we can take the -1 as not contacted. It is done by the code below.

```
# Create buckets or categories for 'pdays'
df['pdays_category'] = pd.cut(
    df['pdays'],
    bins=[-2, 0, 100, 300, 900],
    labels=['Not Contacted', 'Recently Contacted', 'Contacted Long Ago', 'Very Long Ago']
)
```

Then one hot encoding is done by the code below for each label as it is the most suitable for this feature.

```
# One-hot encode the categories
df = pd.get_dummies(df, columns=['pdays_category'], prefix='pdays_cat')

# Ensure all boolean-like columns are integers
df = df.astype({col: 'int' for col in df.select_dtypes(include='bool').columns})

# Display the transformed dataset
print(df)
```

2.13. Analyzing other features.

Categorical features like default, housing and loan have only yes and no as their values which is binary datatype. So doing label encoding will reduce the dimension of the dataset. To analyse the importance of these features a simple cross-tabulation heatmap is plotted for each feature with y variable. The result looked like this.

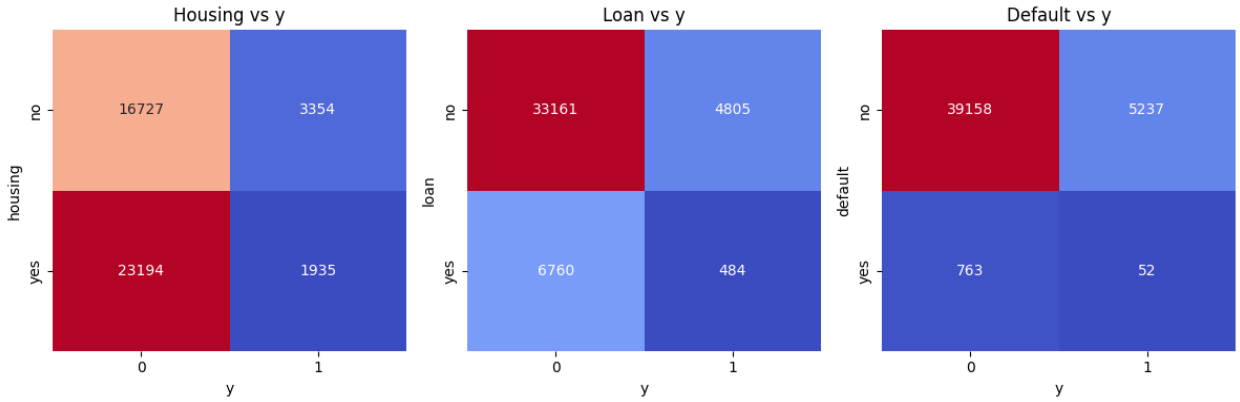

Figure 19.heat map for binary

By this diagram we can understand that each feature has an influence on y variable. When a client has a housing loan there's more chance that client can subscribe to a term deposit. Likewise, the other two features also have that influence.

Here we didn't use any libraries. This can be achieved easily as they only have yes and no. The code below is used to encode the columns.

```
# List of columns to apply Label Encoding to (yes/no columns)
yes_no_columns = ['default', 'housing', 'loan']

# Apply Label Encoding to each of the columns in the list
df[yes_no_columns] = df[yes_no_columns].replace({'yes': 1, 'no': 0})

df.head()
```

2.14. Performing Label Encoding for education feature.

Education feature's values have a relationship with each other. For example, primary < secondary < tertiary < unknown. for this kind of relationship, it is better to use Label encoding. The label encoding is performed by the code below.

```
# Label Encoding
education_mapping = {'primary': 0, 'secondary': 1, 'tertiary': 2, 'unknown': 3}

df['education_encoded'] = df['education'].map(education_mapping)
df = df.drop(columns=['education'])
df.head()
```

2.15. Encoding Job feature.

Job is the main determining feature for the prediction. It is the source of income. The most suitable encoding type for this feature is one hot encoding as it does not have any relationship between its categorical values. The below plotting shows how the y is distributed for the job feature.

Figure 20. bar plot job

The one hot encoding is performed for the job column by the code below.

```
# Apply One-Hot Encoding to the 'job' column
df_encoded = pd.get_dummies(df['job'], prefix='job')

# Convert True/False to 1/0
df_encoded = df_encoded.astype(int)

# Optionally, concatenate the encoded columns with the original dataframe
```

```
df = pd.concat([df, df_encoded], axis=1)

# Drop the original 'job' column
df.drop('job', axis=1, inplace=True)
df.head()
```

2.16. Encoding marital feature.

Next the same chi-square test is done for marital feature. It gave an output like this.

```
Chi-square statistic: 196.45890449554065
p-value: 2.1851986745328 52e-43
```

Figure 21. p value marital

We can see that there's a strong connection between marital and y variable as the p-value is extremely low. As per my opinion, the one hot encoding is the better opinion. Because value might have connection. But we cannot decide which can be greater or lower.

```
# Apply One-Hot Encoding to the 'marital' column
df_encoded = pd.get_dummies(df['marital'], prefix='marital')


# Convert boolean columns to integers (1 for True, 0 for False)
df_encoded = df_encoded.astype(int)


# Concatenate the encoded columns with the original dataframe
df = pd.concat([df, df_encoded], axis=1)


# Drop the original 'marital' column
df.drop('marital', axis=1, inplace=True)


# Display the resulting DataFrame
print(df)
```

2.17. Training the model.

A simple fully connected neural network is used for training the model. This is a simple feedforward neural network for binary classification from tensorflow. The model is trained two times. One for raw data and one for scaled data. Neural networks work efficiently with scaled data. Both ways of training are evaluated to check which one performs best. First, important libraries were imported. After that the y values is separated for the prediction. Then the dataset is split into a training set and a testing set. The code below is used to perform these actions.

```
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Features (all columns except 'y')
X = df.drop('y', axis=1).values
y = df['y'].values

# Split the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

After that, the training data of x and test data of x is standardized for one model. Standardscaler is used from the scikit learn. Then the model is built with keras. For now, it only has one hidden layer with Relu function and 10 neurons. The reason for using Relu function is because it is computationally efficient and avoids saturation problems. Then the output layer with sigmoid function as it is for binary classification.

```
# Standardize the data (standardization)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Model architecture (both models will be the same)
def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(10, activation='relu', input_shape=(X_train.shape[1],)),  #
Hidden layer with 10 neurons
        tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer for binary
classification
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Then two more models were trained. One for standardized dataset and other for raw dataset and the model was evaluated and plotted.

```
# Train the model on raw data (without standardization)

model_raw = build_model()

history_raw = model_raw.fit(X_train, y_train, epochs=70, batch_size=32, validation_split=0.2,
verbose=1)


# Train the model on standardized data (with standardization)

model_scaled = build_model()

history_scaled = model_scaled.fit(X_train_scaled, y_train, epochs=70, batch_size=32,
validation_split=0.2, verbose=1)
```

Then the plotting codes were coded. The result for training the model with raw data is displayed below.

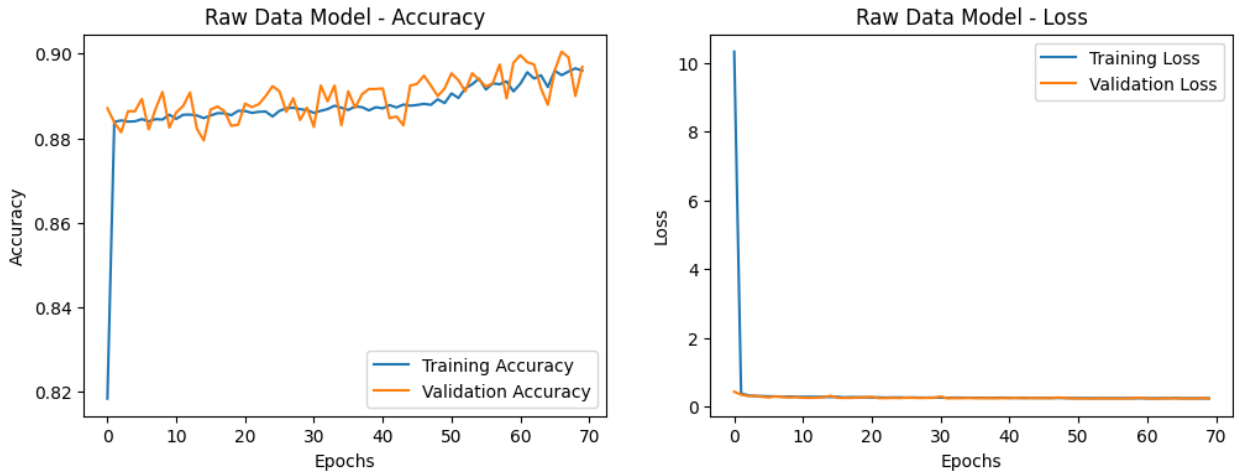

Figure 22. neural network raw

This shows that raw data works pretty well for the model. The model is learning effectively. Though there might be overfitting as it the validation accuracy is rising more than the training accuracy.in the loss, there's a high initial loss, therefore it has to be considered. Secondly the model which trained with the standardized data is displayed below.

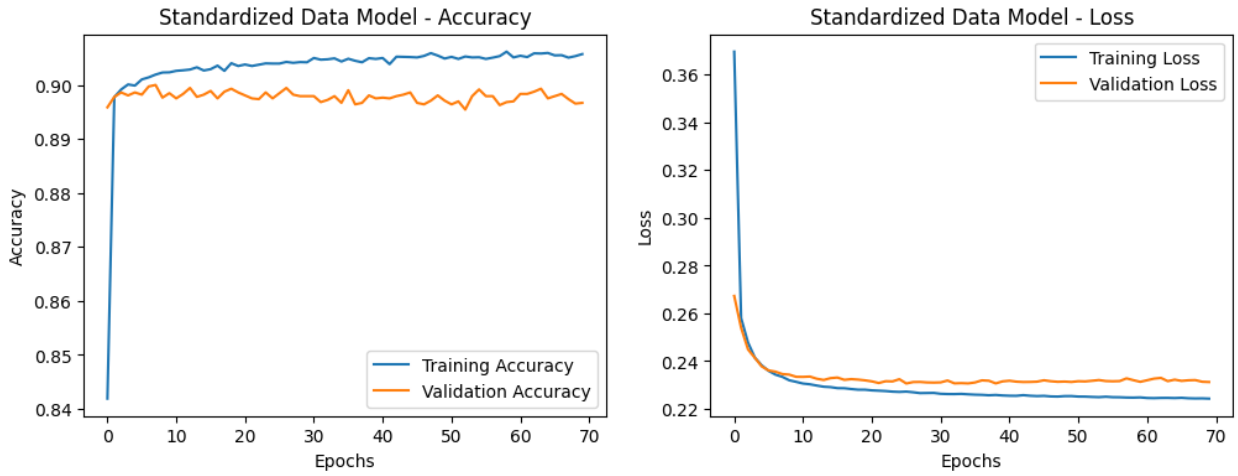

Figure 23. neural network standardize

The standardized data model is also performing better. Both validation and accuracy are stabilizing faster compared to raw data. This indicates better learning. Validation loss is also gradually decreasing which is a good sign. But it's not possible to decide as it has only one hidden layer. The model might not learn properly.

2.18. Model Evaluation.

Finally, the above model is evaluated with the test accuracy for raw data and test accuracy for standardized data. Scikit learn is used for the evaluation metrices.

```
from sklearn.metrics import  classification_report, confusion_matrix

# Evaluate the model on raw data
test_loss_raw, test_accuracy_raw = model_raw.evaluate(X_test, y_test)
print(f"Test Accuracy (Raw Data): {test_accuracy_raw * 100:.2f}%")

# Evaluate the model on standardized data
test_loss_scaled, test_accuracy_scaled = model_scaled.evaluate(X_test_scaled, y_test)
print(f"Test Accuracy (Standardized Data): {test_accuracy_scaled * 100:.2f}%")
```

Then the predictions were made and evaluated by the code below.

```
# Make predictions and evaluate using sklearn (both raw and standardized data)
y_pred_raw = (model_raw.predict(X_test) > 0.5).astype("int32")
y_pred_scaled = (model_scaled.predict(X_test_scaled) > 0.5).astype("int32")

# Evaluation metrics for raw data
print("\nClassification Report (Raw Data):")
print(classification_report(y_test, y_pred_raw))
print("\nConfusion Matrix (Raw Data):")
print(confusion_matrix(y_test, y_pred_raw))

# Evaluation metrics for standardized data
print("\nClassification Report (Standardized Data):")
```

```
print(classification_report(y_test, y_pred_scaled))
print("\nConfusion Matrix (Standardized Data):")
print(confusion_matrix(y_test, y_pred_scaled))
```
The result we got is displayed below. The first one is for raw data.

```
Classification Report (Raw Data):
              precision    recall  f1-score   support

           0       0.91      0.98      0.94      7949
           1       0.62      0.28      0.39      1093

    accuracy                           0.89      9042
   macro avg       0.76      0.63      0.66      9042
weighted avg       0.87      0.89      0.87      9042


Confusion Matrix (Raw Data):
[[7760  189]
 [ 786  307]]
```

Figure 24. report raw

As you can see, there's class imbalance for the raw data model. The recall is very low. And the f1-score as well.

```
Classification Report (Standardized Data):
              precision    recall  f1-score   support

           0       0.92      0.97      0.94      7949
           1       0.63      0.38      0.47      1093

    accuracy                           0.90      9042
   macro avg       0.77      0.67      0.71      9042
weighted avg       0.88      0.90      0.89      9042


Confusion Matrix (Standardized Data):
[[7702  247]
 [ 679  414]]
```

Figure 25. report standardized

As you can see. The class imbalance is slightly reduced for the standardized data. Even though it must be handled. It might not predict class 1 properly with this metrics

```
283/283 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8887 - loss: 0.2489
Test Accuracy (Raw Data): 89.22%
283/283 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8931 - loss: 0.2386
Test Accuracy (Standardized Data): 89.76%
283/283 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step
283/283 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
```

Figure 26. accuracy first model

We can see that it has got a descent accuracy. Both models have validations close to training metrics. So we can see that there's less chances for overfitting. But we cannot finalize it with the class imbalance. So further improvements must be made.

2.19. Adding class weight to balance the class.

As we can see there is a class imbalance, class weights were added for the model to handle the class imbalance. So we can get better scores for other metrics as well. First class weights were assigned by these codes.

```
# Compute class weights
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

class_weights = compute_class_weight(
    class_weight="balanced",
    classes=np.unique(y_train),
    y=y_train
)
class_weights = dict(enumerate(class_weights))
```

It is applied for y variable. Then the model is built again with scaled data. The same parameters were used to build the model again.

```
# Train the model with class weights
model_scaled_class_weights = build_model()
history_class_weights = model_scaled_class_weights.fit(
    X_train_scaled, y_train,
    epochs=70,
    batch_size=32,
    validation_split=0.2,
    class_weight=class_weights,  # Add class weights here
    verbose=1
)

# Plot for the model trained with class weights
plot_history(history_class_weights, 'Standardized Data Model with Class Weights')
```

The result is displayed below.

Figure 27. class weight model

We can see that the accuracy is pretty low as it is 80 percent. Also, we can see a significant increase in the validation loss that suggests there might be an overfit. So again, those things have to be considered. A classification report is also being made to check the changes in the other metrics.

```
Classification Report:
              precision    recall  f1-score   support

     Class 0       0.97      0.80      0.88      7949
     Class 1       0.37      0.84      0.51      1093

    accuracy                           0.81      9042
   macro avg       0.67      0.82      0.69      9042
weighted avg       0.90      0.81      0.83      9042

Accuracy: 0.81
```

Figure 28. report for weigted model

We can see that the recall has come to a better-balanced position. Though precision is pretty low. So going for further improvements.

2.20. Hyperparameter tuning for the Model.

To enhance the model, Hyperparameter tuning is performed. First libraries were imported.

```
import tensorflow as tf
import keras_tuner as kt
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils.class_weight import compute_class_weight
```

again, the standardized data is used here as neural network works better on standardized data. Here the number of layers, neurons, and learning rates were optimized using keras tuner. Each tunable part are coded according to the tuner. The code below is the code used to build the model.

```
# Define the model-building function for the tuner
def build_model(hp):
    model = tf.keras.Sequential()

    # Tuning the number of layers and neurons
    model.add(tf.keras.layers.Dense(
        hp.Int('units1', min_value=8, max_value=64, step=8),
        activation='relu',
        input_shape=(X_train.shape[1],)
    ))

    # Optionally add more layers
    for i in range(hp.Int('num_layers', 1, 3)):  # 1 to 3 hidden layers
        model.add(tf.keras.layers.Dense(
            hp.Int(f'units_{i+2}', min_value=8, max_value=64, step=8),
            activation='relu'
        ))

    model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

```
    # Compile the model with a tunable learning rate
    model.compile(
        optimizer=tf.keras.optimizers.Adam(
                    learning_rate=hp.Float('learning_rate',  min_value=1e-5,  max_value=1e-2,
sampling='LOG')
        ),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model
```

and the below is the code for the tuner.

```
tuner = kt.Hyperband(
    build_model,  # The model-building function
    objective='val_accuracy',  # The metric to optimize
    max_epochs=60,  # Max epochs to train each model
    factor=3,  # Factor by which the number of trials is reduced
    directory='tuner_results',  # Directory to save results
    project_name='hyperparameter_tuning',  # Name for the project
)
```

Hyperband optimizes validation accuracy. It trains the model upto 60 epochs. Then tuner.search executes it with different training models with different training data and retrieves the best parameters.

```
tuner.search(X_train_scaled, y_train, epochs=60, validation_split=0.2, verbose=1)

# Get the best hyperparameters
best_hyperparameters = tuner.get_best_hyperparameters(num_trials=1)[0]
print("Best hyperparameters: ", best_hyperparameters.values)
```

class weights also considered for the imbalance data as we faced earlier. It is tuned with different code set

```
# Compute class weights
class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train),
    y=y_train
)
class_weights = dict(enumerate(class_weights))
print("Class Weights:", class_weights)
```

The best model which is retrieved is saved and trained separately to get the results. The trained history with each trail with the parameters are saved for efficiency. It does not require more time to train again as it retrieves it from the information saved. And finally, the graph of the training is plotted.

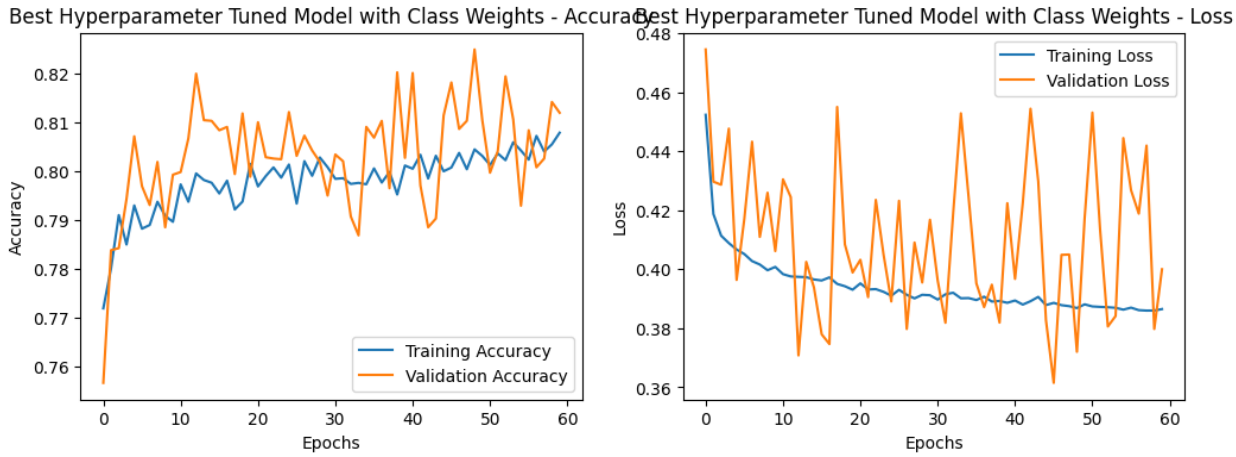

Figure 29. parameter tuned model

The best parameter was this.

```
Best hyperparameters:  {'units1': 8, 'num_layers': 1, 'units_2': 24, 'learning_rate':
0.008583846101691946, 'units_3': 48, 'units_4': 40, 'tuner/epochs': 60,
'tuner/initial_epoch': 0, 'tuner/bracket': 0, 'tuner/round': 0}
```

We can see that it has 3 hidden layers. With 8, 24, 48 neurons respectively. Learning rate also pretty much low. Even though it has got 3 hidden layers it considers only first 2 layers as the given num_layers is 1. So, the 3rd layer is ignored. From the graph we can see that that accuracy is optimized to 80 percent. It is fairly a low accuracy. Validation loss also has fairly a larger value. When we have a look at the classification report.

```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.80      0.88      7949
           1       0.37      0.85      0.51      1093

    accuracy                           0.81      9042
   macro avg       0.67      0.82      0.70      9042
weighted avg       0.90      0.81      0.84      9042

Accuracy: 0.8068
Confusion Matrix:
[[6369 1580]
 [ 167  926]]
```

Figure 30. report best param

We can see that the class imbalance is handled a bit. Still, there's an imbalance in precision. The accuracy is also pretty low compared to the other model. There for further improvement are considered.

2.21. Manually adjusting parameters.

The next step that was taken is adjusting the parameters and repeat training. By adjusting class weights for each and every time and the results were monitored. The class weights were handled by SMOTE and ADASYN. Both of them used different times to check whether they performed best. Many attempts of changing weights, thresholds, increasing and decreasing hidden layers, adjusting neurons, adjusting learning rate, adding class weights, changing epoch value and so much other stuff. Kernel regularization is made in each layer and batch normalization is also made between the layers. After hours of training, I managed to get a graph in this pattern.

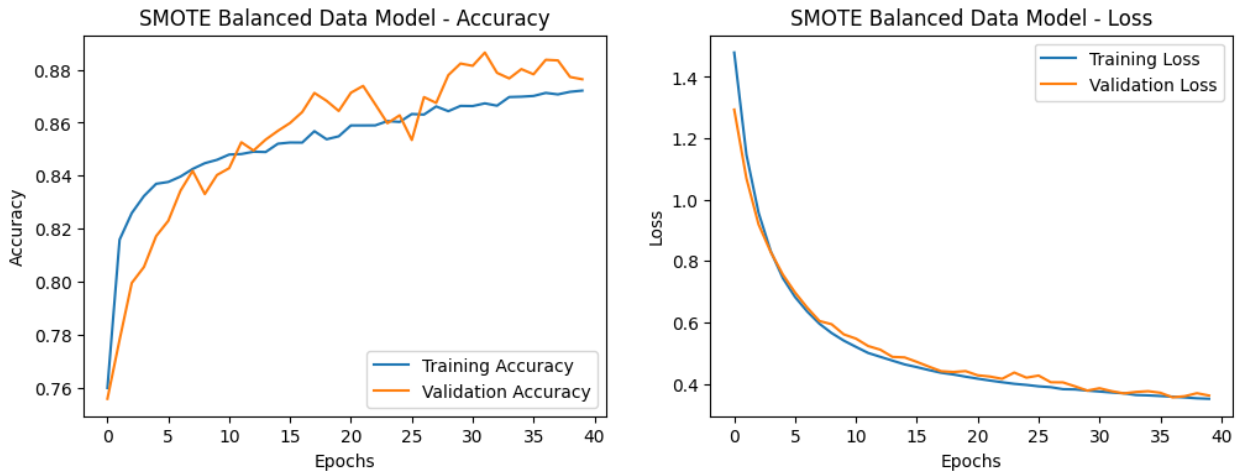

Figure 31. manual adjustedd result

Some how managed to align the training and validation together. Overall, this model performed well. The parameter is mentioned below.

The model had these parameters.

```
def build_model():
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],),
kernel_regularizer=regularizers.l2(0.01)),
        tf.keras.layers.BatchNormalization(),  # Add BatchNormalization layer
        tf.keras.layers.Dense(64, activation='relu',kernel_regularizer=regularizers.l2(0.01)),
        tf.keras.layers.BatchNormalization(),  # Add BatchNormalization layer
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

class weights had this parameter to handle class imbalance.

```
class_weights = {0: 1, 1: 1.15}
```

and other model parameters.

```
history_smote = model_smote.fit(
    X_train_smote, y_train_smote,
    epochs=40,
    batch_size=64,
    validation_split=0.2,
    verbose=1,
    class_weight=class_weights
)
```

2.22. Handling class imbalance.

To get a good imbalance. Model is predicted with different thresholds. It was performed by the code below.

```
thresholds = [0.71, 0.73, 0.75, 0.78]

for t in thresholds:
    y_test_pred = (model_smote.predict(X_test_scaled) > t).astype(int)
    print(f"Threshold: {t}")
    print(classification_report(y_test, y_test_pred, target_names=['Class 0', 'Class 1']))
```

among these threshold, threshold = 0.71 was the best one. The classification report for it is displayed below.

```
Threshold: 0.71
              precision    recall  f1-score   support

     Class 0       0.94      0.93      0.93      7949
     Class 1       0.52      0.57      0.54      1093

    accuracy                           0.88      9042
   macro avg       0.73      0.75      0.74      9042
weighted avg       0.89      0.88      0.89      9042
```

Figure 32. manual adjusted report

It has a class imbalance. But this is the best value that could be retrieved by this model. Further it can be achieved by more experiments.

2.23. ROC curve.

Finally, the ROC curve is plotted for the above model. It was plotted by the code below.

```
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# Calculate ROC curve and AUC
y_pred_probs = model_smote.predict(X_test_scaled)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_probs)
roc_auc = roc_auc_score(y_test, y_pred_probs)
```

```
# Plot ROC Curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

The threshold is applied, and this was the result.

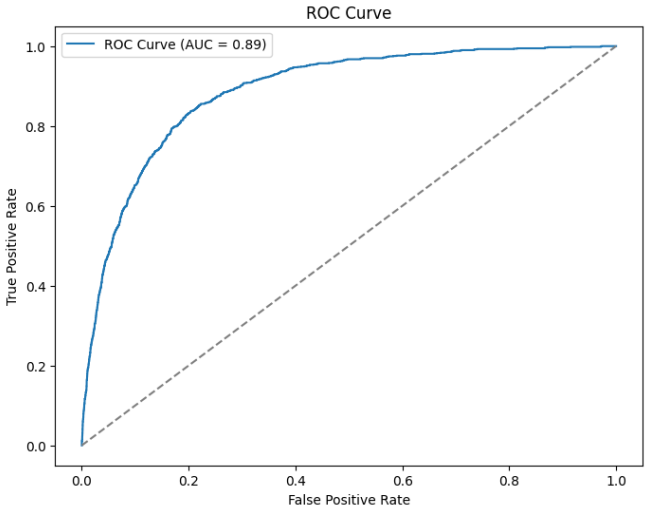

Figure 33. ROC for neural network

We can see that the model is performing well. The curve is closer to the left top corner. This indicates better performance.

2.24. Conclusion for Neural Network.

As we can see from the results, after many attempts of manually adjusted parameter model worked the best. Both training and validation aligned almost the same. Training accuracy increased from 0.70 to 0.87 and validation loss decreased from 1.6 to 0.35 indicating that it can predict well. We can see that in the validation accuracy it went from 0.75 to 0.87 which indicates that it can perform well on unseen data. Validation loss from 1.29 to 0.36 shows the good performance of the model. So, we can come to the conclusion that this model is performing well for unseen data.

3. Random Forest.

3.1. Data Preprocessing.

The same way of data preprocessing which was used in the neural network is applied here. Therefore, there's no change in the way of data cleaning data transformation and other analysis. Directly training section will be explained for the random forest classifier.

3.2. Model training.

First the libraires were imported. Randomforest classifier is imported from scikit learn. The dataset is split for training set and testing set then rf model is Initialized and trained. 100 decision trees were used for training. Below code performs these tasks.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Features (all columns except 'y')
X = df.drop('y', axis=1)

# Target variable
y = df['y']

# Split the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the Random Forest model on the training data
rf_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf_model.predict(X_test)
```

3.3. Model Evaluation.

To evaluate the model, simply usual metrices were used. Accuracy and the classification report with f1score, support and precision etc. more than that cross validation are performed here. Commonly used k-fold-cross-validation is used with k equal to 5. We can see if it overfits or not. Also, it can utilize the full dataset to be trained. The code below is the way it is implemented.

```
# Evaluate the model

accuracy = accuracy_score(y_test, y_pred)

print(f"Random Forest Model Accuracy: {accuracy * 100:.2f}%")

# Detailed Classification Report

print("\nClassification Report:")

print(classification_report(y_test, y_pred))

# Confusion Matrix

print("\nConfusion Matrix:")

print(confusion_matrix(y_test, y_pred))

# Optional: Perform cross-validation

cv_scores = cross_val_score(rf_model, X, y, cv=5, scoring='accuracy')

print(f"\nCross-Validation Scores: {cv_scores}")

print(f"Mean Cross-Validation Accuracy: {cv_scores.mean() * 100:.2f}%")
```

The result we got for the above model is displayed below.

```
Random Forest Model Accuracy: 89.62%

Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.97      0.94      7949
           1       0.62      0.37      0.46      1093

    accuracy                           0.90      9042
   macro avg       0.77      0.67      0.70      9042
weighted avg       0.88      0.90      0.88      9042


Confusion Matrix:
[[7697  252]
 [ 687  406]]

Cross-Validation Scores: [0.89062154 0.88321168 0.87115682 0.86540588 0.72063703]
Mean Cross-Validation Accuracy: 84.62%
```

Figure 34. report random forest

We can see that the above model is not performing well, as for different subsets it gives different accuracy even though the accuracy it gave is 89%. When we look at the cross-validation accuracy it ranges from 73 to 89, which can be a worse case. So further improvements must be made. There's class imbalance as well. That thing also must be considered.

3.4. Hyperparameter tuning for RF.

Now the model is trained with hyperparameter. RandomizedSearchCV is used to the randomforest for the hyperparameter tuning. Define arrays of parameters for the random combination. The code below shows the parameter distribution. SMOTE is applied to handle class imbalances.

```
# Apply SMOTE to balance the dataset
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

param_distributions = { 'n_estimators': [100, 200, 300],
                        'max_depth': [10, 20, 30, None],
                        'min_samples_split': [2, 5, 10],
                        'min_samples_leaf': [1, 2, 4] }
```

After that a model is initialized and then the randomsearch with random combination is performed by using the codes below. Then randomized search is trained.

```
# Initialize the Random Forest model

rf_model = RandomForestClassifier(random_state=42)


# Perform RandomizedSearchCV

random_search = RandomizedSearchCV(

    estimator=rf_model,

    param_distributions=param_distributions,

    n_iter=30,  # Number of random combinations to try

    cv=5,       # 5-fold cross-validation

    scoring='accuracy',  # Scoring metric

    random_state=42,

    n_jobs=-1    # Use all available CPU cores

)


# Fit RandomizedSearchCV on the training data

random_search.fit(X_train, y_train)
```

selected parameters are defined as below.

```
# Define the parameter distribution for random sampling
param_distributions = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'class_weight': ['balanced', 'balanced_subsample', None]
}
```

After all the randomized search the best model is selected which has the best accuracy. Then the classification report is displayed for that.

```
# Get the best model
best_rf_model = random_search.best_estimator_

# Print the best hyperparameters
print("Best Hyperparameters:", random_search.best_params_)

# Evaluate the best model on the test set
y_pred = best_rf_model.predict(X_test)

# Metrics and evaluation
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

The best rf model that detected was displayed with it’s classification report with it.

```
Class Distribution after SMOTE:
{0: 31972, 1: 31972}
Best Hyperparameters: {'n_estimators': 300, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_depth': 30, 'class_weight': 'balanced_subsample'}
Model Accuracy: 89.44%

Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.95      0.94      7949
           1       0.58      0.48      0.52      1093

    accuracy                           0.89      9042
   macro avg       0.75      0.71      0.73      9042
weighted avg       0.89      0.89      0.89      9042


Confusion Matrix:
[[7565  384]
 [ 571  522]]
```

Figure 35. best rf model report

By the report we can see that it has a decent accuracy with 89 percent accuracy. But recall has comparatively less value. So still the class imbalance is not handled properly by the best model from the hyperparameter also. Then cross validation is performed for the best model by the code below.

```
from sklearn.model_selection import cross_val_score


# Perform cross-validation on the best model
cv_scores = cross_val_score(
    best_rf_model,    # Best model selected by RandomizedSearchCV
    X_train_smote,    # The SMOTE-augmented training data
    y_train_smote,    # The target labels
    cv=5,             # 5-fold cross-validation
    scoring='accuracy',  # Use accuracy as the scoring metric
    n_jobs=-1         # Use all available CPU cores
)


# Print the cross-validation scores
print("Cross-validation scores for the best model:")
print(cv_scores)


# Print the mean and standard deviation of the cross-validation scores
print(f"\nMean Accuracy: {cv_scores.mean() * 100:.2f}%")
print(f"Standard Deviation: {cv_scores.std() * 100:.2f}%")
```

the output it gave for the cross validation is displayed below.

```
Cross-validation scores for the best model:
[0.73961999 0.9659082  0.96285871 0.96872312 0.96731311]

Mean Accuracy: 92.09%
Standard Deviation: 9.07%
```

Figure 36. best rf cross validation

Here we can see that it trained with 5 different subsets of dataset. Then the mean accuracy is calculated with the standard deviation. We can see that it is clearly overfitting. For one data fold it goes down to 0.73 but for others it is more than 0.90. This model cannot be used for prediction. Further adjustments and improvements must be made.

3.5. Manually adjusting the parameters.

The next task was to adjust the parameters of the random forest model multiple times to get a good accuracy with class balance. The parameters were defined in this manner. Changes will be made within this code.

```
# Adjust parameters
manual_params = {
    "n_estimators": 200,
    "max_depth": 25,
    "min_samples_split": 5,
    "min_samples_leaf": 2,
    "class_weight": "balanced"
}
```

The model is trained with a def function so that the parameters can be passed easily when it changes each time. Classification report with accuracy is displayed for each run. The function is defined as below.

```
# Function to manually adjust hyperparameters and evaluate the model
def evaluate_rf_model(X_train, y_train, X_test, y_test, params):
    # Create the Random Forest model with the given parameters
    rf_model = RandomForestClassifier(**params, random_state=42)

    # Fit the model to the training data
    rf_model.fit(X_train, y_train)

    # Make predictions on the test data
    y_pred = rf_model.predict(X_test)

    # Compute accuracy, classification report, and confusion matrix
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Random Forest Model Accuracy: {accuracy * 100:.2f}%\n")

    print("Classification Report:")
    print(classification_report(y_test, y_pred))
```

```
    print("Confusion Matrix:")
    print(confusion_matrix(y_test, y_pred))
```

Then the function is called, and the model is trained.

```
# Call the function with the parameters to evaluate
evaluate_rf_model(X_train, y_train, X_test, y_test, manual_params)
```

After many attempts at adjusting parameters and rerunning the codes the best accuracy with a better class balance that could be retrieved is displayed below.

```
Random Forest Model Accuracy: 88.95%

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.92      0.94      7949
           1       0.54      0.65      0.59      1093

    accuracy                           0.89      9042
   macro avg       0.74      0.78      0.76      9042
weighted avg       0.90      0.89      0.89      9042

Confusion Matrix:
[[7336  613]
 [ 386  707]]
```

Figure 37. manually adjusted report

Comparatively this is the best model with good accuracy and better class balance than other models. So, this can be used for prediction. The cross validation is performed for this model to see the performance for different subsets of dataset. This was the result we got for this model is displayed below.

```
Performing 5-Fold Cross-Validation...
Cross-Validation Scores (Per Fold):
Fold 1: Accuracy = 88.62%
Fold 2: Accuracy = 88.78%
Fold 3: Accuracy = 87.88%
Fold 4: Accuracy = 88.75%
Fold 5: Accuracy = 88.73%

Mean Cross-Validation Accuracy: 88.55%
Standard Deviation of Accuracy: 0.34%
```

Figure 38. manually adjusted cross validation

So we can see that this model performs well even on the cross validation. The ROC is plotted to identify how well it can be performed. The ROC is plotted for the above model to see the performance using the code below.

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, roc_auc_score
```

```
# Fit the model and get predictions with probabilities
rf_model = RandomForestClassifier(**manual_params, random_state=42)
rf_model.fit(X_train, y_train)

# Get predicted probabilities for the positive class (class 1)
y_prob = rf_model.predict_proba(X_test)[:, 1]

# Compute the ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = roc_auc_score(y_test, y_prob)

# Plot the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--', label='Random Guess')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve for Random Forest Classifier')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```

the output of the code is displayed.

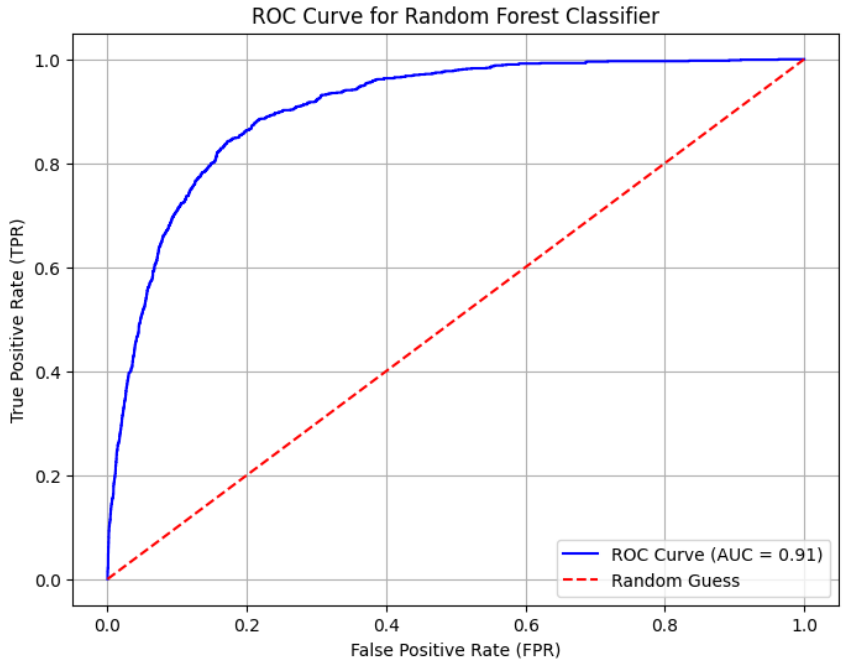

Figure 39. ROC random forest

We can see that there's a great performance in the model. It has AUC value of 0.91. We can see that the class is well balanced. So, this model can be trusted for predictions. Further evaluations can be made of the model by adjusting a few more parameters or adding some other improvements. But for now, overall, this model is perfect for predictions.

4. Git hub.

Git hub URL for the codes - https://github.com/Abdullah-Nazly/ML_CW.git

5. Appendix.

5.1. Source codes for Neural Network

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

df = pd.read_csv('./Dataset/bank-full.csv', delimiter=';')
df.head(20)
print(df.isnull().sum())
print(df.duplicated().sum())  # Count the number of duplicate rows
df.info()
df.describe()
numeric_columns = df.select_dtypes(include=['int64', 'float64']).columns
print("Numeric Columns:", numeric_columns)
categorical_columns = df.select_dtypes(include=['object']).columns
print("Categorical Columns:", categorical_columns)
boolean_columns = df.select_dtypes(include=['bool']).columns
print("Boolean Columns:", boolean_columns)
df['job'].unique()
df['marital'].unique()
df['education'].unique()
df['default'].unique()
df['housing'].unique()
df['contact'].unique()
df['month'].unique()
df['poutcome'].unique()
df['y'].unique()
df['campaign'].unique()

from scipy.stats import chi2_contingency

# If p value is < 0.05, the feature is influencing the targeted variable

crosstab = pd.crosstab(df['contact'], df['y'])
chi2, p, dof, expected = chi2_contingency(crosstab)
```

```
print("Chi-square statistic:", chi2)
print("p-value:", p)

# Apply One-Hot Encoding to the 'contact' column
df_encoded = pd.get_dummies(df['contact'], prefix='contact')

# Convert True/False to 1/0
df_encoded = df_encoded.astype(int)

# concatenate the encoded columns with the original dataframe
df = pd.concat([df, df_encoded], axis=1)

# Drop the original 'contact' column
df.drop('contact', axis=1, inplace=True)

df.head()

crosstab = pd.crosstab(df['poutcome'], df['y'])
chi2, p, dof, expected = chi2_contingency(crosstab)
print("Chi-square statistic:", chi2)
print("p-value:", p)
from scipy.stats import pointbiserialr
from sklearn.preprocessing import LabelEncoder
# Initialize the LabelEncoder
label_encoder = LabelEncoder()

# Apply Label Encoding to the 'poutcome' column
df['poutcome_encoded'] = label_encoder.fit_transform(df['poutcome'])

df.drop('poutcome', axis=1, inplace=True)

# Display the resulting DataFrame
print(df)

from scipy.stats import pointbiserialr

# Step 1: Convert 'y' to numeric (binary)
df['y'] = df['y'].map({'no': 0, 'yes': 1})

# Step 2: Define numerical features
numerical_features = ['age', 'balance', 'day', 'duration', 'campaign',
   'pdays', 'previous']

# Step 3: Calculate Point-Biserial Correlation for each feature
correlation_results = []
```

```
for col in numerical_features:
    corr, p_value = pointbiserialr(df[col], df['y'])
    correlation_results.append((col, corr, p_value))

# Step 4: Print results
print("Feature-wise Point-Biserial Correlation and p-values:")
for feature, corr, p_value in correlation_results:
    print(f"Feature: {feature}, Correlation: {corr:.3f}, p-value:
  {p_value:.3f}")

import seaborn as sns

# Convert correlation results to a DataFrame
correlation_df = pd.DataFrame(correlation_results, columns=['Feature',
  'Correlation', 'p_value'])

# Create a pivot for heatmap visualization (correlation coefficients)
heatmap_data = correlation_df.pivot_table(values='Correlation',
  index=['Feature'])

    # Plot heatmap
    plt.figure(figsize=(4, 4))
    sns.heatmap(heatmap_data, annot=True, cmap='coolwarm', cbar=True,
  fmt='.3f', linewidths=0.6, linecolor='black')
    plt.title('Heatmap of Point-Biserial Correlations', fontsize=14)
    plt.xlabel('Correlation', fontsize=12)
    plt.ylabel('Features', fontsize=12)
    plt.tight_layout()

    # Display the heatmap
    plt.show()

    import seaborn as sns
    import matplotlib.pyplot as plt

    # Count plot for the 'campaign' variable (discrete) and binary
  target 'y'
    sns.countplot(x='campaign', hue='y', data=df, palette='Set2')

    plt.title('Campaign vs Default Outcome')
    plt.xlabel('Campaign')
    plt.ylabel('Count')
    plt.show()

    # Assuming df is your DataFrame with 'campaign' and 'y' columns
```

```
    campaign_counts = df.groupby('campaign')['y'].value_counts().unstack(fill_value=0)

    # Display the result
    print(campaign_counts)

    df = df.drop(columns=['day', 'month'])

    sns.boxplot(x='y', y='duration', data=df)
    sns.boxplot(x='y', y='age', data=df)
    sns.boxplot(x='y', y='balance', data=df)
    sns.boxplot(x='y', y='previous', data=df)
    # Identify the record with the extreme value in 'previous'
    outlier_row = df[df['previous'] > 250]

    # Display the details of the record
    print("Outlier row details:")
    print(outlier_row)

    # Drop the specific row
    df = df.drop(outlier_row.index)

    # Verify the row is removed
    print(f"Updated dataset shape: {df.shape}")
    df.head()

    sns.boxplot(x='y', y='pdays', data=df)

    from sklearn.preprocessing import StandardScaler

    # Create buckets or categories for 'pdays'
    df['pdays_category'] = pd.cut(
        df['pdays'],
        bins=[-2, 0, 100, 300, 900],
        labels=['Not Contacted', 'Recently Contacted', 'Contacted Long Ago', 'Very Long Ago']
    )

    # One-hot encode the categories
    df = pd.get_dummies(df, columns=['pdays_category'], prefix='pdays_cat')

    # Ensure all boolean-like columns are integers
    df = df.astype({col: 'int' for col in df.select_dtypes(include='bool').columns})
```

```
164.
165.      # Display the transformed dataset
166.      print(df)
167.
168.
169.      # Create cross-tabulation for each feature with 'y'
170.      cross_tab_housing = pd.crosstab(df['housing'], df['y'])
171.      cross_tab_loan = pd.crosstab(df['loan'], df['y'])
172.      cross_tab_default = pd.crosstab(df['default'], df['y'])
173.
174.      # Plot heatmaps
175.      plt.figure(figsize=(12, 4))
176.
177.      # Heatmap for housing feature
178.      plt.subplot(1, 3, 1)
179.      sns.heatmap(cross_tab_housing, annot=True, fmt='d', cmap='coolwarm',
    cbar=False)
180.      plt.title('Housing vs y')
181.
182.      # Heatmap for loan feature
183.      plt.subplot(1, 3, 2)
184.      sns.heatmap(cross_tab_loan, annot=True, fmt='d', cmap='coolwarm',
    cbar=False)
185.      plt.title('Loan vs y')
186.
187.      # Heatmap for default feature
188.      plt.subplot(1, 3, 3)
189.      sns.heatmap(cross_tab_default, annot=True, fmt='d', cmap='coolwarm',
    cbar=False)
190.      plt.title('Default vs y')
191.
192.      plt.tight_layout()
193.      plt.show()
194.
195.      # List of columns to apply Label Encoding to (yes/no columns)
196.      yes_no_columns = ['default', 'housing', 'loan']
197.
198.      # Apply Label Encoding to each of the columns in the list
199.      df[yes_no_columns] = df[yes_no_columns].replace({'yes': 1, 'no': 0})
200.
201.      df.head()
202.      # Label Encoding
203.      education_mapping = {'primary': 0, 'secondary': 1, 'tertiary': 2,
    'unknown': 3}
204.
```

```
205.      df['education_encoded'] = df['education'].map(education_mapping)
206.      df = df.drop(columns=['education'])
207.      df.head()
208.
209.      df['job'].unique()
210.      # Grouped bar plot for job and y
211.      plt.figure(figsize=(10, 6))
212.
213.      # Create a crosstab of job vs y
214.      job_y_crosstab = pd.crosstab(df['job'], df['y'])
215.
216.      # Plot the grouped bar chart
217.      job_y_crosstab.plot(kind='bar', figsize=(12, 6), color=['skyblue',
    'salmon'])
218.
219.      plt.title("Distribution of Target Variable (y) by Job", fontsize=16)
220.      plt.xlabel("Job", fontsize=14)
221.      plt.ylabel("Count", fontsize=14)
222.      plt.xticks(rotation=45, fontsize=12)
223.      plt.legend(["No", "Yes"], title="Target (y)", fontsize=12)
224.      plt.tight_layout()
225.      plt.grid(axis='y', alpha=0.3)
226.      plt.show()
227.
228.      # Apply One-Hot Encoding to the 'job' column
229.      df_encoded = pd.get_dummies(df['job'], prefix='job')
230.
231.      # Convert True/False to 1/0
232.      df_encoded = df_encoded.astype(int)
233.
234.      # Optionally, concatenate the encoded columns with the original
    dataframe
235.      df = pd.concat([df, df_encoded], axis=1)
236.
237.      # Drop the original 'job' column
238.      df.drop('job', axis=1, inplace=True)
239.
240.      df.head()
241.
242.
243.      from scipy.stats import chi2_contingency
244.
245.      # If p value is < 0.05, the feature is influencing the targeted
    variable
246.
```

```
    crosstab = pd.crosstab(df['marital'], df['y'])
    chi2, p, dof, expected = chi2_contingency(crosstab)
    print("Chi-square statistic:", chi2)
    print("p-value:", p)

    # Apply One-Hot Encoding to the 'marital' column
    df_encoded = pd.get_dummies(df['marital'], prefix='marital')

    # Convert boolean columns to integers (1 for True, 0 for False)
    df_encoded = df_encoded.astype(int)

    # Concatenate the encoded columns with the original dataframe
    df = pd.concat([df, df_encoded], axis=1)

    # Drop the original 'marital' column
    df.drop('marital', axis=1, inplace=True)

    # Display the resulting DataFrame
    print(df)
    df.info()
    import tensorflow as tf
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler
    import matplotlib.pyplot as plt

    # Features (all columns except 'y')
    X = df.drop('y', axis=1).values
    y = df['y'].values

    # Split the dataset into training and testing sets (80% train, 20%
test)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

    # Standardize the data (standardization)
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Model architecture (both models will be the same)
    def build_model():
        model = tf.keras.Sequential([
            tf.keras.layers.Dense(10, activation='relu',
input_shape=(X_train.shape[1],)),  # Hidden layer with 10 neurons
```

```
            tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer for binary classification
        ])
        model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
        return model

    # Train the model on raw data (without standardization)
    model_raw = build_model()
    history_raw = model_raw.fit(X_train, y_train, epochs=70, batch_size=32, validation_split=0.2, verbose=1)

    # Train the model on standardized data (with standardization)
    model_scaled = build_model()
    history_scaled = model_scaled.fit(X_train_scaled, y_train, epochs=70, batch_size=32, validation_split=0.2, verbose=1)

    # Plotting the graphs
    def plot_history(history, title):
        # Plot Accuracy
        plt.figure(figsize=(12, 4))
        plt.subplot(1, 2, 1)
        plt.plot(history.history['accuracy'], label='Training Accuracy')
        plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
        plt.title(f'{title} - Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.legend()

        # Plot Loss
        plt.subplot(1, 2, 2)
        plt.plot(history.history['loss'], label='Training Loss')
        plt.plot(history.history['val_loss'], label='Validation Loss')
        plt.title(f'{title} - Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()

        plt.show()

    # Plot for the raw data model
    plot_history(history_raw, 'Raw Data Model')

    # Plot for the standardized data model
```

```
328.     plot_history(history_scaled, 'Standardized Data Model')
329.
330.     from sklearn.metrics import classification_report, confusion_matrix
331.
332.     # Evaluate the model on raw data
333.     test_loss_raw, test_accuracy_raw = model_raw.evaluate(X_test,
    y_test)
334.     print(f"Test Accuracy (Raw Data): {test_accuracy_raw * 100:.2f}%")
335.
336.     # Evaluate the model on standardized data
337.     test_loss_scaled, test_accuracy_scaled =
    model_scaled.evaluate(X_test_scaled, y_test)
338.     print(f"Test Accuracy (Standardized Data): {test_accuracy_scaled *
    100:.2f}%")
339.
340.     # Make predictions and evaluate using sklearn (both raw and
    standardized data)
341.     y_pred_raw = (model_raw.predict(X_test) > 0.5).astype("int32")
342.     y_pred_scaled = (model_scaled.predict(X_test_scaled) >
    0.5).astype("int32")
343.
344.     # Evaluation metrics for raw data
345.     print("\nClassification Report (Raw Data):")
346.     print(classification_report(y_test, y_pred_raw))
347.     print("\nConfusion Matrix (Raw Data):")
348.     print(confusion_matrix(y_test, y_pred_raw))
349.
350.     # Evaluation metrics for standardized data
351.     print("\nClassification Report (Standardized Data):")
352.     print(classification_report(y_test, y_pred_scaled))
353.     print("\nConfusion Matrix (Standardized Data):")
354.     print(confusion_matrix(y_test, y_pred_scaled))
355.
356.     # Compute class weights
357.     from sklearn.utils.class_weight import compute_class_weight
358.     import numpy as np
359.
360.     class_weights = compute_class_weight(
361.         class_weight="balanced",
362.         classes=np.unique(y_train),
363.         y=y_train
364.     )
365.     class_weights = dict(enumerate(class_weights))
366.
367.     print("Class Weights:", class_weights)
```

```
368.
369.      # Train the model with class weights
370.      model_scaled_class_weights = build_model()
371.      history_class_weights = model_scaled_class_weights.fit(
372.          X_train_scaled, y_train,
373.          epochs=70,
374.          batch_size=32,
375.          validation_split=0.2,
376.          class_weight=class_weights,  # Add class weights here
377.          verbose=1
378.      )
379.
380.      # Plot for the model trained with class weights
381.      plot_history(history_class_weights, 'Standardized Data Model with
    Class Weights')
382.
383.      from sklearn.metrics import classification_report, accuracy_score
384.
385.      # Make predictions on the test set
386.      y_pred = model_scaled_class_weights.predict(X_test_scaled)
387.      y_pred_classes = (y_pred > 0.5).astype(int)  # Convert probabilities
    to class labels (0 or 1)
388.
389.      # Calculate accuracy
390.      accuracy = accuracy_score(y_test, y_pred_classes)
391.
392.      # Generate the classification report
393.      report = classification_report(y_test, y_pred_classes,
    target_names=['Class 0', 'Class 1'])
394.
395.      print("Classification Report:")
396.      print(report)
397.      print(f"Accuracy: {accuracy:.2f}")
398.
399.      import tensorflow as tf
400.      import keras_tuner as kt
401.      import matplotlib.pyplot as plt
402.      from sklearn.model_selection import train_test_split
403.      from sklearn.preprocessing import StandardScaler
404.      from sklearn.utils.class_weight import compute_class_weight
405.
406.      # Features (all columns except 'y')
407.      X = df.drop('y', axis=1).values
408.      y = df['y'].values
409.
```

```
    # Split the dataset into training and testing sets (80% train, 20% test)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Standardize the data (standardization)
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Define the model-building function for the tuner
    def build_model(hp):
        model = tf.keras.Sequential()

        # Tuning the number of layers and neurons
        model.add(tf.keras.layers.Dense(
            hp.Int('units1', min_value=8, max_value=64, step=8),
            activation='relu',
            input_shape=(X_train.shape[1],)
        ))

        # Optionally add more layers
        for i in range(hp.Int('num_layers', 1, 3)):  # 1 to 3 hidden layers
            model.add(tf.keras.layers.Dense(
                hp.Int(f'units_{i+2}', min_value=8, max_value=64, step=8),
                activation='relu'
            ))

        model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

        # Compile the model with a tunable learning rate
        model.compile(
            optimizer=tf.keras.optimizers.Adam(
                learning_rate=hp.Float('learning_rate', min_value=1e-5, max_value=1e-2, sampling='LOG')
            ),
            loss='binary_crossentropy',
            metrics=['accuracy']
        )

        return model

    # Define the tuner
```

```
    tuner = kt.Hyperband(
        build_model,  # The model-building function
        objective='val_accuracy',  # The metric to optimize
        max_epochs=60,  # Max epochs to train each model
        factor=3,  # Factor by which the number of trials is reduced
        directory='tuner_results',  # Directory to save results
        project_name='hyperparameter_tuning',  # Name for the project
    )

    # Run the tuner
    tuner.search(X_train_scaled, y_train, epochs=60,
validation_split=0.2, verbose=1)

    # Get the best hyperparameters
    best_hyperparameters =
tuner.get_best_hyperparameters(num_trials=1)[0]
    print("Best hyperparameters: ", best_hyperparameters.values)

    # Build the best model with the best hyperparameters
    best_model = tuner.hypermodel.build(best_hyperparameters)

    # Compute class weights
    class_weights = compute_class_weight(
        class_weight='balanced',
        classes=np.unique(y_train),
        y=y_train
    )
    class_weights = dict(enumerate(class_weights))
    print("Class Weights:", class_weights)

    # Modify the training step to use class weights
    history = best_model.fit(
        X_train_scaled, y_train,
        epochs=60,
        validation_split=0.2,
        batch_size=32,
        verbose=1,
        class_weight=class_weights  # Pass class weights here
    )

    # Evaluate on the test set
    test_loss, test_accuracy = best_model.evaluate(X_test_scaled,
y_test)
    print(f"Test Loss: {test_loss}")
    print(f"Test Accuracy: {test_accuracy}")
```

```
492.
493.      # Plot the history of the best model
494.      plot_history(history, 'Best Hyperparameter Tuned Model with Class
   Weights')
495.
496.      # Plotting the training and validation metrics
497.      def plot_history(history, title):
498.          # Plot Accuracy
499.          plt.figure(figsize=(12, 4))
500.          plt.subplot(1, 2, 1)
501.          plt.plot(history.history['accuracy'], label='Training Accuracy')
502.          plt.plot(history.history['val_accuracy'], label='Validation
   Accuracy')
503.          plt.title(f'{title} - Accuracy')
504.          plt.xlabel('Epochs')
505.          plt.ylabel('Accuracy')
506.          plt.legend()
507.
508.          # Plot Loss
509.          plt.subplot(1, 2, 2)
510.          plt.plot(history.history['loss'], label='Training Loss')
511.          plt.plot(history.history['val_loss'], label='Validation Loss')
512.          plt.title(f'{title} - Loss')
513.          plt.xlabel('Epochs')
514.          plt.ylabel('Loss')
515.          plt.legend()
516.
517.          plt.tight_layout()
518.          plt.show()
519.
520.      # Make predictions on the test set
521.      y_pred = best_model.predict(X_test_scaled)
522.      y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to
   binary (0 or 1)
523.
524.      # Print the classification report
525.      from sklearn.metrics import classification_report, accuracy_score,
   confusion_matrix
526.
527.      print("Classification Report:")
528.      print(classification_report(y_test, y_pred))
529.
530.      # Calculate the accuracy
531.      accuracy = accuracy_score(y_test, y_pred)
532.      print(f"Accuracy: {accuracy:.4f}")
```

```
533.
534.      # Confusion Matrix
535.      cm = confusion_matrix(y_test, y_pred)
536.      print("Confusion Matrix:")
537.      print(cm)
538.
539.      from imblearn.over_sampling import SMOTE
540.      import tensorflow as tf
541.      from sklearn.model_selection import train_test_split
542.      from sklearn.preprocessing import StandardScaler
543.      import matplotlib.pyplot as plt
544.      from tensorflow.keras import regularizers
545.
546.      # Features (all columns except 'y')
547.      X = df.drop('y', axis=1).values
548.      y = df['y'].values
549.
550.      # Split the dataset into training and testing sets (80% train, 20%
   test)
551.      X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)
552.
553.      # Standardize the data (standardization)
554.      scaler = StandardScaler()
555.      X_train_scaled = scaler.fit_transform(X_train)
556.      X_test_scaled = scaler.transform(X_test)
557.
558.      smote = SMOTE(random_state=42)
559.      X_train_smote, y_train_smote = smote.fit_resample(X_train_scaled,
   y_train)
560.
561.      def build_model():
562.          optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
563.          model = tf.keras.Sequential([
564.              tf.keras.layers.Dense(64, activation='relu',
   input_shape=(X_train.shape[1],),
   kernel_regularizer=regularizers.l2(0.01)),
565.              tf.keras.layers.BatchNormalization(),  # Add
   BatchNormalization layer
566.              tf.keras.layers.Dense(64,
   activation='relu',kernel_regularizer=regularizers.l2(0.01)),
567.              tf.keras.layers.BatchNormalization(),  # Add
   BatchNormalization layer
568.              tf.keras.layers.Dense(1, activation='sigmoid')
569.          ])
```

```
570.         model.compile(optimizer=optimizer, loss='binary_crossentropy',
   metrics=['accuracy'])
571.         return model
572.
573.     # Train the model on SMOTE-balanced data
574.     model_smote = build_model()
575.
576.     class_weights = {0: 1, 1: 1.15}
577.
578.     history_smote = model_smote.fit(
579.         X_train_smote, y_train_smote,
580.         epochs=40,
581.         batch_size=64,
582.         validation_split=0.2,
583.         verbose=1,
584.         class_weight=class_weights
585.     )
586.
587.     # Function to plot training and validation metrics
588.     def plot_history(history, title):
589.         # Plot Accuracy
590.         plt.figure(figsize=(12, 4))
591.         plt.subplot(1, 2, 1)
592.         plt.plot(history.history['accuracy'], label='Training Accuracy')
593.         plt.plot(history.history['val_accuracy'], label='Validation
   Accuracy')
594.         plt.title(f'{title} - Accuracy')
595.         plt.xlabel('Epochs')
596.         plt.ylabel('Accuracy')
597.         plt.legend()
598.
599.         # Plot Loss
600.         plt.subplot(1, 2, 2)
601.         plt.plot(history.history['loss'], label='Training Loss')
602.         plt.plot(history.history['val_loss'], label='Validation Loss')
603.         plt.title(f'{title} - Loss')
604.         plt.xlabel('Epochs')
605.         plt.ylabel('Loss')
606.         plt.legend()
607.
608.         plt.show()
609.
610.     # Plot for the SMOTE-balanced data model
611.     plot_history(history_smote, 'SMOTE Balanced Data Model')
612.
```

```
    thresholds = [0.71, 0.73, 0.75, 0.78]
    for t in thresholds:
        y_test_pred = (model_smote.predict(X_test_scaled) > t).astype(int)
        print(f"Threshold: {t}")
        print(classification_report(y_test, y_test_pred, target_names=['Class 0', 'Class 1']))

    from sklearn.metrics import roc_curve, roc_auc_score
    import matplotlib.pyplot as plt

    # Calculate ROC curve and AUC
    y_pred_probs = model_smote.predict(X_test_scaled)
    fpr, tpr, thresholds = roc_curve(y_test, y_pred_probs)
    roc_auc = roc_auc_score(y_test, y_pred_probs)

    # Plot ROC Curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, label=f"ROC Curve (AUC = {roc_auc:.2f})")
    plt.plot([0, 1], [0, 1], linestyle='--', color='gray')
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend()
    plt.show()


```

5.2. Source code for RandomForest.

```
1. import pandas as pd
2. import matplotlib.pyplot as plt
3. import seaborn as sns
4.
5. df = pd.read_csv('./Dataset/bank-full.csv', delimiter=';')
6. df.head(20)
7. print(df.isnull().sum())
8. print(df.duplicated().sum())  # Count the number of duplicate rows
9. df.info()
10.df.describe()
1. numeric_columns = df.select_dtypes(include=['int64', 'float64']).columns
2. print("Numeric Columns:", numeric_columns)
3. categorical_columns = df.select_dtypes(include=['object']).columns
4. print("Categorical Columns:", categorical_columns)
5. boolean_columns = df.select_dtypes(include=['bool']).columns
6. print("Boolean Columns:", boolean_columns)
7. df['job'].unique()
8. df['marital'].unique()
9. df['education'].unique()
10.df['default'].unique()
11.df['housing'].unique()
12.df['contact'].unique()
13.df['month'].unique()
14.df['poutcome'].unique()
15.df['y'].unique()
16.df['campaign'].unique()
17.
18.from scipy.stats import chi2_contingency
19.
20.# If p value is < 0.05, the feature is influencing the targeted variable
21.
22.crosstab = pd.crosstab(df['contact'], df['y'])
23.chi2, p, dof, expected = chi2_contingency(crosstab)
24.print("Chi-square statistic:", chi2)
25.print("p-value:", p)
26.
27.# Apply One-Hot Encoding to the 'contact' column
28.df_encoded = pd.get_dummies(df['contact'], prefix='contact')
29.
30.# Convert True/False to 1/0
31.df_encoded = df_encoded.astype(int)
32.
```

```
# concatenate the encoded columns with the original dataframe
df = pd.concat([df, df_encoded], axis=1)

# Drop the original 'contact' column
df.drop('contact', axis=1, inplace=True)

df.head()


crosstab = pd.crosstab(df['poutcome'], df['y'])
chi2, p, dof, expected = chi2_contingency(crosstab)
print("Chi-square statistic:", chi2)
print("p-value:", p)

from sklearn.preprocessing import LabelEncoder
# Initialize the LabelEncoder
label_encoder = LabelEncoder()

# Apply Label Encoding to the 'poutcome' column
df['poutcome_encoded'] = label_encoder.fit_transform(df['poutcome'])

df.drop('poutcome', axis=1, inplace=True)

# Display the resulting DataFrame
print(df)


from scipy.stats import pointbiserialr

# Step 1: Convert 'y' to numeric (binary)
df['y'] = df['y'].map({'no': 0, 'yes': 1})

# Step 2: Define numerical features
numerical_features = ['age', 'balance', 'day', 'duration', 'campaign',
   'pdays', 'previous']

# Step 3: Calculate Point-Biserial Correlation for each feature
correlation_results = []
for col in numerical_features:
    corr, p_value = pointbiserialr(df[col], df['y'])
    correlation_results.append((col, corr, p_value))

# Step 4: Print results
print("Feature-wise Point-Biserial Correlation and p-values:")
for feature, corr, p_value in correlation_results:
```

```
    print(f"Feature: {feature}, Correlation: {corr:.3f}, p-value: {p_value:.3f}")


import seaborn as sns

# Convert correlation results to a DataFrame
correlation_df = pd.DataFrame(correlation_results, columns=['Feature', 'Correlation', 'p_value'])

# Create a pivot for heatmap visualization (correlation coefficients)
heatmap_data = correlation_df.pivot_table(values='Correlation', index=['Feature'])

# Plot heatmap
plt.figure(figsize=(4, 4))
sns.heatmap(heatmap_data, annot=True, cmap='coolwarm', cbar=True, fmt='.3f', linewidths=0.6, linecolor='black')
plt.title('Heatmap of Point-Biserial Correlations', fontsize=14)
plt.xlabel('Correlation', fontsize=12)
plt.ylabel('Features', fontsize=12)
plt.tight_layout()

# Display the heatmap
plt.show()

import seaborn as sns
    import matplotlib.pyplot as plt

    # Count plot for the 'campaign' variable (discrete) and binary target 'y'
    sns.countplot(x='campaign', hue='y', data=df, palette='Set2')

    plt.title('Campaign vs Default Outcome')
    plt.xlabel('Campaign')
    plt.ylabel('Count')
    plt.show()

    import pandas as pd

    campaign_counts = df.groupby('campaign')['y'].value_counts().unstack(fill_value=0)

    # Display the result
    print(campaign_counts)
```

```
116.
117.
118.
119.      df = df.drop(columns=['day', 'month'])
120.      sns.boxplot(x='y', y='duration', data=df)
121.      sns.boxplot(x='y', y='age', data=df)
122.      sns.boxplot(x='y', y='balance', data=df)
123.      sns.boxplot(x='y', y='previous', data=df)
124.
125.      # Identify the record with the extreme value in 'previous'
126.      outlier_row = df[df['previous'] > 250]
127.
128.      # Display the details of the record
129.      print("Outlier row details:")
130.      print(outlier_row)
131.
132.      # Drop the specific row
133.      df = df.drop(outlier_row.index)
134.
135.      # Verify the row is removed
136.      print(f"Updated dataset shape: {df.shape}")
137.
138.      df.head()
139.
140.      sns.boxplot( x='y', y= 'pdays', data=df )
141.
142.      from sklearn.preprocessing import StandardScaler
143.
144.      # Create buckets or categories for 'pdays'
145.      df['pdays_category'] = pd.cut(
146.          df['pdays'],
147.          bins=[-2, 0, 100, 300, 900],
148.          labels=['Not Contacted', 'Recently Contacted', 'Contacted Long
   Ago', 'Very Long Ago']
149.      )
150.
151.      # One-hot encode the categories
152.      df = pd.get_dummies(df, columns=['pdays_category'],
   prefix='pdays_cat')
153.
154.      # Ensure all boolean-like columns are integers
155.      df = df.astype({col: 'int' for col in
   df.select_dtypes(include='bool').columns})
156.
157.      # Display the transformed dataset
```

```
158.      print(df)
159.
160.
161.
162.      # Create cross-tabulation for each feature with 'y'
163.      cross_tab_housing = pd.crosstab(df['housing'], df['y'])
164.      cross_tab_loan = pd.crosstab(df['loan'], df['y'])
165.      cross_tab_default = pd.crosstab(df['default'], df['y'])
166.
167.      # Plot heatmaps
168.      plt.figure(figsize=(12, 4))
169.
170.      # Heatmap for housing feature
171.      plt.subplot(1, 3, 1)
172.      sns.heatmap(cross_tab_housing, annot=True, fmt='d', cmap='coolwarm',
    cbar=False)
173.      plt.title('Housing vs y')
174.
175.      # Heatmap for loan feature
176.      plt.subplot(1, 3, 2)
177.      sns.heatmap(cross_tab_loan, annot=True, fmt='d', cmap='coolwarm',
    cbar=False)
178.      plt.title('Loan vs y')
179.
180.      # Heatmap for default feature
181.      plt.subplot(1, 3, 3)
182.      sns.heatmap(cross_tab_default, annot=True, fmt='d', cmap='coolwarm',
    cbar=False)
183.      plt.title('Default vs y')
184.
185.      plt.tight_layout()
186.      plt.show()
187.
188.      # List of columns to apply Label Encoding to (yes/no columns)
189.      yes_no_columns = ['default', 'housing', 'loan']
190.
191.      # Apply Label Encoding to each of the columns in the list
192.      df[yes_no_columns] = df[yes_no_columns].replace({'yes': 1, 'no': 0})
193.
194.      df.head()
195.
196.      # Label Encoding
197.      education_mapping = {'primary': 0, 'secondary': 1, 'tertiary': 2,
    'unknown': 3}
198.
```

```
199.      df['education_encoded'] = df['education'].map(education_mapping)
200.      df = df.drop(columns=['education'])
201.      df.head()
202.
203.      df['job'].unique()
204.
205.      # Grouped bar plot for job and y
206.      plt.figure(figsize=(10, 6))
207.
208.      # Create a crosstab of job vs y
209.      job_y_crosstab = pd.crosstab(df['job'], df['y'])
210.
211.      # Plot the grouped bar chart
212.      job_y_crosstab.plot(kind='bar', figsize=(12, 6), color=['skyblue',
    'salmon'])
213.
214.      plt.title("Distribution of Target Variable (y) by Job", fontsize=16)
215.      plt.xlabel("Job", fontsize=14)
216.      plt.ylabel("Count", fontsize=14)
217.      plt.xticks(rotation=45, fontsize=12)
218.      plt.legend(["No", "Yes"], title="Target (y)", fontsize=12)
219.      plt.tight_layout()
220.      plt.grid(axis='y', alpha=0.3)
221.      plt.show()
222.
223.      # Apply One-Hot Encoding to the 'job' column
224.      df_encoded = pd.get_dummies(df['job'], prefix='job')
225.
226.      # Convert True/False to 1/0
227.      df_encoded = df_encoded.astype(int)
228.
229.      # Optionally, concatenate the encoded columns with the original
    dataframe
230.      df = pd.concat([df, df_encoded], axis=1)
231.
232.      # Drop the original 'job' column (optional)
233.      df.drop('job', axis=1, inplace=True)
234.
235.      df.head()
236.
237.      # Apply One-Hot Encoding to the 'marital' column
238.      df_encoded = pd.get_dummies(df['marital'], prefix='marital')
239.
240.      # Convert boolean columns to integers (1 for True, 0 for False)
241.      df_encoded = df_encoded.astype(int)
```

```
242.
243.      # Concatenate the encoded columns with the original dataframe
244.      df = pd.concat([df, df_encoded], axis=1)
245.
246.      # Drop the original 'marital' column
247.      df.drop('marital', axis=1, inplace=True)
248.
249.      # Display the resulting DataFrame
250.      print(df)
251.
252.      df.info()
253.
254.      from sklearn.ensemble import RandomForestClassifier
255.      from sklearn.model_selection import train_test_split,
   cross_val_score
256.      from sklearn.metrics import accuracy_score, classification_report,
   confusion_matrix
257.
258.      # Features (all columns except 'y')
259.      X = df.drop('y', axis=1)
260.
261.      # Target variable
262.      y = df['y']
263.
264.      # Split the dataset into training and testing sets (80% train, 20%
   test)
265.      X_train, X_test, y_train, y_test = train_test_split(X, y,
   test_size=0.2, random_state=42)
266.
267.      # Initialize the Random Forest model
268.      rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
269.
270.      # Train the Random Forest model on the training data
271.      rf_model.fit(X_train, y_train)
272.
273.      # Make predictions on the test set
274.      y_pred = rf_model.predict(X_test)
275.      # Evaluate the model
276.      accuracy = accuracy_score(y_test, y_pred)
277.      print(f"Random Forest Model Accuracy: {accuracy * 100:.2f}%")
278.
279.      # Detailed Classification Report
280.      print("\nClassification Report:")
281.      print(classification_report(y_test, y_pred))
282.
```

```
    # Confusion Matrix
    print("\nConfusion Matrix:")
    print(confusion_matrix(y_test, y_pred))

    # Optional: Perform cross-validation
    cv_scores = cross_val_score(rf_model, X, y, cv=5,
scoring='accuracy')
    print(f"\nCross-Validation Scores: {cv_scores}")

    print(f"Mean Cross-Validation Accuracy: {cv_scores.mean() *
100:.2f}%")

    from imblearn.over_sampling import SMOTE
    from sklearn.model_selection import RandomizedSearchCV
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
    import numpy as np

    # Apply SMOTE to balance the dataset
    smote = SMOTE(random_state=42)
    X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

    # Print class distribution after SMOTE
    print("Class Distribution after SMOTE:")
    unique, counts = np.unique(y_train_smote, return_counts=True)
    print(dict(zip(unique, counts)))

    # Define the parameter distribution for random sampling
    param_distributions = {
        'n_estimators': [100, 200, 300],
        'max_depth': [10, 20, 30, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4],
        'class_weight': ['balanced', 'balanced_subsample', None]  # No
custom weights since data is now balanced
    }

    # Initialize the Random Forest model
    rf_model = RandomForestClassifier(random_state=42)

    # Perform RandomizedSearchCV
    random_search = RandomizedSearchCV(
        estimator=rf_model,
        param_distributions=param_distributions,
```

```
        n_iter=30,  # Number of random combinations to try
        cv=5,       # 5-fold cross-validation
        scoring='accuracy',  # Scoring metric
        random_state=42,
        n_jobs=-1    # Use all available CPU cores
    )

    # Fit RandomizedSearchCV on the SMOTE-augmented training data
    random_search.fit(X_train_smote, y_train_smote)

    # Get the best model
    best_rf_model = random_search.best_estimator_

    # Print the best hyperparameters
    print("Best Hyperparameters:", random_search.best_params_)

    # Evaluate the best model on the test set
    y_pred = best_rf_model.predict(X_test)

    # Metrics and evaluation
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Model Accuracy: {accuracy * 100:.2f}%")
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))
    print("\nConfusion Matrix:")
    print(confusion_matrix(y_test, y_pred))

    from sklearn.model_selection import cross_val_score

    # Perform cross-validation on the best model
    cv_scores = cross_val_score(
        best_rf_model,    # Best model selected by RandomizedSearchCV
        X_train_smote,    # The SMOTE-augmented training data
        y_train_smote,    # The target labels
        cv=5,             # 5-fold cross-validation
        scoring='accuracy',  # Use accuracy as the scoring metric
        n_jobs=-1         # Use all available CPU cores
    )

    # Print the cross-validation scores
    print("Cross-validation scores for the best model:")
    print(cv_scores)

    # Print the mean and standard deviation of the cross-validation
scores
```

```
    print(f"\nMean Accuracy: {cv_scores.mean() * 100:.2f}%")
    print(f"Standard Deviation: {cv_scores.std() * 100:.2f}%")

    from sklearn.ensemble import RandomForestClassifier
    from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score

    # Function to manually adjust hyperparameters and evaluate the model
    def evaluate_rf_model(X_train, y_train, X_test, y_test, params):
        # Create the Random Forest model with the given parameters
        rf_model = RandomForestClassifier(**params, random_state=42)

        # Fit the model to the training data
        rf_model.fit(X_train, y_train)

        # Make predictions on the test data
        y_pred = rf_model.predict(X_test)

        # Compute accuracy, classification report, and confusion matrix
        accuracy = accuracy_score(y_test, y_pred)
        print(f"Random Forest Model Accuracy: {accuracy * 100:.2f}%\n")

        print("Classification Report:")
        print(classification_report(y_test, y_pred))

        print("Confusion Matrix:")
        print(confusion_matrix(y_test, y_pred))

    # Adjust parameters
    manual_params = {
        "n_estimators": 200,
        "max_depth": 25,
        "min_samples_split": 5,
        "min_samples_leaf": 2,
        "class_weight": "balanced"
    }

    # Call the function with the parameters to evaluate
    evaluate_rf_model(X_train, y_train, X_test, y_test, manual_params)

    from sklearn.ensemble import RandomForestClassifier
    from sklearn.model_selection import cross_val_score
    import numpy as np

    # Perform cross-validation
```

```
    print("Performing 5-Fold Cross-Validation...")
    cv_scores = cross_val_score(rf_model, X_train, y_train, cv=5,
scoring='accuracy')

    # Display results
    print("Cross-Validation Scores (Per Fold):")
    for i, score in enumerate(cv_scores):
        print(f"Fold {i+1}: Accuracy = {score * 100:.2f}%")

    # Calculate and display the mean and standard deviation of cross-
validation accuracy
    mean_accuracy = np.mean(cv_scores) * 100
    std_accuracy = np.std(cv_scores) * 100
    print(f"\nMean Cross-Validation Accuracy: {mean_accuracy:.2f}%")
    print(f"Standard Deviation of Accuracy: {std_accuracy:.2f}%")

    import matplotlib.pyplot as plt
    from sklearn.metrics import roc_curve, roc_auc_score

    # Fit the model and get predictions with probabilities
    rf_model = RandomForestClassifier(**manual_params, random_state=42)
    rf_model.fit(X_train, y_train)

    # Get predicted probabilities for the positive class (class 1)
    y_prob = rf_model.predict_proba(X_test)[:, 1]

    # Compute the ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(y_test, y_prob)
    roc_auc = roc_auc_score(y_test, y_prob)

    # Plot the ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='blue', label=f'ROC Curve (AUC =
{roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='red', linestyle='--', label='Random
Guess')
    plt.xlabel('False Positive Rate (FPR)')
    plt.ylabel('True Positive Rate (TPR)')
    plt.title('ROC Curve for Random Forest Classifier')
    plt.legend(loc='lower right')
    plt.grid()
    plt.show()
```

6. References.

GeeksforGeeks, 2024. *Neural Networks – A Beginner's Guide*. [online] Available at: https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/ [Accessed 17 December 2024].

YouTube, 2024. *Neural Networks – A Beginner's Guide*. [video] Available at: https://youtu.be/v6VJ2RO66Ag?si=OKzPFFoR8huOMl1T [Accessed 18 December 2024].

YouTube, 2024. *Deep Learning Tutorial - Full Course for Beginners*. [video] Available at: https://youtu.be/jmmW0F0biz0?si=PXdfjI-Dyjfi7APu [Accessed 17 December 2024].