BSc (Hons) Artificial Intelligence and Data Science

Module: CM2604 Machine Learning

Report

Module Leader: Mr. Sahan Priyanayana

GitHub Repository Link:

https://github.com/ChanulT/Bank-Marketing-ML-Analysis.git

RGU Student ID : 2330948

IIT Student ID : 20231591

Student Name : Chanul Vitharana

Table of Contents

1 GitHub Instructions

The project files and code are hosted on a GitHub repository. Follow the instructions below to access and interact with the repository:

- To access the repository : Open the GitHub repository using the following link:GitHub Repository Link.
- Structure of the repository:
  1. bank-additional/bank-additional: Contains the dataset used for analysis.
  2. bank-additional/bank-additional: Contains the dataset used for analysis.
  3. Bank-Marketing-Prediction.ipynb: Jupyter Notebook with the implementation of the machine learning models.
  4. README.md: Documentation providing an overview of the project.
  5. ROC AUC curve.png: Visualization of the model's ROC-AUC curve.

2 Introduction

This report provides step-by-step guidance for creating a simple classification model to predict whether a client will subscribe to a term deposit based on a bank marketing dataset. The machine learning models referenced in the coursework specification include a neural network and random forest classifiers for making predictions. The report will detail the process, from downloading and preprocessing the dataset to training and testing it with machine learning algorithms.

3 Corpus Preparation

3.1 Downloading the data-set

The link given in the course description led to an official repository to download the required data set for the coursework. The data set was given in a zipped folder as CSV files. There were two main folders named 'bank' and 'bank-additional'. Depending on the preference, a data set from one of the above folders can be used to solve the problem. In this case, the 'bank-additional' folder was chosen and the following files were found inside.

1. bank-additional-full.csv with all examples, ordered by date (from May 2008 to November 2010).
2. bank-additional.csv with 10% of the examples (4119), randomly selected from bank-additional-full.csv.

The smallest dataset is provided to test more computationally demanding machine learning algorithms

3.2 Creating the Data Frame

```
[ ]
    test = pd.read_csv("/content/drive/MyDrive/ML/bank-additional.csv",delimiter=';')
    train = pd.read_csv("/content/drive/MyDrive/ML/bank-additional-full.csv", delimiter=';')

    test = test[train.columns]

    # view the data
    print("Train Data:")
    data_set = train
    data_set

    #view the data
    print("Test Data:")
    data_set_test = test
    data_set_test
```

Initially, there were two CSV data files in the folder. The file containing less data (bank-additional.csv) was used as the test dataset, while the larger file was designated for training purposes.
The 'read_csv' function in Pandas, a Python library, is used to read the CSV files and create a DataFrame. Finally, the datasets are displayed for clarity.

```
Train Data:
```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	...	1	999	0	nonexistent
2	37	services	married	high.school	no	yes	no	telephone	may	mon	...	1	999	0	nonexistent
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent
4	56	services	married	high.school	no	no	yes	telephone	may	mon	...	1	999	0	nonexistent
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
41183	73	retired	married	professional.course	no	yes	no	cellular	nov	fri	...	1	999	0	nonexistent
41184	46	blue-collar	married	professional.course	no	no	no	cellular	nov	fri	...	1	999	0	nonexistent
41185	56	retired	married	university.degree	no	yes	no	cellular	nov	fri	...	2	999	0	nonexistent
41186	44	technician	married	professional.course	no	no	no	cellular	nov	fri	...	1	999	0	nonexistent
41187	74	retired	married	professional.course	no	yes	no	cellular	nov	fri	...	3	999	1	failure

41188 rows × 21 columns

Test Data:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcom
0	30	blue-collar	married	basic.9y	no	yes	no	cellular	may	fri	...	2	999	0	nonexister
1	39	services	single	high.school	no	no	no	telephone	may	fri	...	4	999	0	nonexister
2	25	services	married	high.school	no	yes	no	telephone	jun	wed	...	1	999	0	nonexister
3	38	services	married	basic.9y	no	unknown	unknown	telephone	jun	fri	...	3	999	0	nonexister
4	47	admin.	married	university.degree	no	yes	no	cellular	nov	mon	...	1	999	0	nonexister
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	.
4114	30	admin.	married	basic.6y	no	yes	yes	cellular	jul	thu	...	1	999	0	nonexister
4115	39	admin.	married	high.school	no	yes	no	telephone	jul	fri	...	1	999	0	nonexister
4116	27	student	single	high.school	no	no	no	cellular	may	mon	...	2	999	1	failur
4117	58	admin.	married	high.school	no	no	no	cellular	aug	fri	...	1	999	0	nonexister
4118	34	management	single	high.school	no	yes	no	cellular	nov	wed	...	1	999	0	nonexister

4119 rows × 21 columns

3.3 Exploratory Data Analysis

This is the process of analyzing the trends in the dataset and identifying its characteristics so that it can be effectively utilized for training and testing purposes of the machine learning model. This process involves plotting graphs to determine the relationships, obtaining statistical measures, and detecting outliers.

3.3.1 Handeling Missing Values

Before moving into EDA, it is a habit to remove the missing values in the dataset or else missing values will lead to inaccurate analysis.

In our dataset, there are no missing values(i.e Rows with Nan values) but there are cells in the dataset with the term "unknown". Hence all the "unknown" values should be handled before moving to EDA.

Let's execute a simple code line to find which columns contain these "Unknown" values.

```
# Select categorical columns (object dtype)
categorical_columns = data_set.select_dtypes(include=['object']).columns

# Check for "unknown" in each categorical column
unknown_counts = {}
for column in categorical_columns:
    count = data_set[column].str.contains("unknown", na=False).sum()  # Count occurrences of "unknown"
    unknown_counts[column] = count

# Print the results
for col, count in unknown_counts.items():
    print(f"Column '{col}' has {count} occurrences of 'unknown'.")
```

```
Column 'job' has 330 occurrences of 'unknown'.
Column 'marital' has 80 occurrences of 'unknown'.
Column 'education' has 1731 occurrences of 'unknown'.
Column 'default' has 8597 occurrences of 'unknown'.
Column 'housing' has 990 occurrences of 'unknown'.
Column 'loan' has 990 occurrences of 'unknown'.
Column 'contact' has 0 occurrences of 'unknown'.
Column 'month' has 0 occurrences of 'unknown'.
Column 'day_of_week' has 0 occurrences of 'unknown'.
Column 'poutcome' has 0 occurrences of 'unknown'.
Column 'y' has 0 occurrences of 'unknown'.
```

In the above output, it is visible that multiple columns have these "Unknows" values with different occurrences. To decide whether to drop or impute the columns or rows with "unknow" term we need to find if there is a hidden connection between the columns and the output(y).

First let's calculate the 'yes' porportion of the target varaible

```
[ ] # Calculate the overall 'yes' rate in the target variable (assume target column is 'y')
    overall_yes_rate = (data_set['y'].value_counts(normalize=True)['yes']) * 100

    print(f"Overall 'Yes' Rate: {overall_yes_rate:.2f}%")
```

```
Overall 'Yes' Rate: 11.27%
```

Having identified the columns with unknown values, we can determine the proportion of 'yes' responses in the rows where these columns are unknown. This will help us assess whether these rows differ in behavior from the overall dataset.

```
# Analyze the 'yes' rate for rows where each column has 'unknown' values
unknown_analysis = {}
#Calculating and diplaying the proportion for each column
for col in unknown_columns:
    unknown_yes_rate = (data_set[data_set[col] == 'unknown']['y'].value_counts(normalize=True).get('yes', 0)) * 100
    unknown_analysis[col] = unknown_yes_rate
    print(f"{col}: 'Yes' Rate for 'unknown' rows = {unknown_yes_rate:.2f}%")
```

```
job: 'Yes' Rate for 'unknown' rows = 11.21%
marital: 'Yes' Rate for 'unknown' rows = 15.00%
education: 'Yes' Rate for 'unknown' rows = 14.50%
default: 'Yes' Rate for 'unknown' rows = 5.15%
housing: 'Yes' Rate for 'unknown' rows = 10.81%
loan: 'Yes' Rate for 'unknown' rows = 10.81%
```

Columns with rates near the "yes" rate of the target variable reveal a strong hidden relationship, whereas those with values far from that rate show a weak or no relationship at all.

```
# Decide which columns to impute or delete
columns_to_impute = []
columns_to_delete = []

for col, unknown_yes_rate in unknown_analysis.items():
    if abs(unknown_yes_rate - overall_yes_rate) <= 2:
        columns_to_impute.append(col)
    else:
        columns_to_delete.append(col)

print(f"Columns to Impute: {columns_to_impute}")
print(f"Columns to Delete: {columns_to_delete}")
```

```
Columns to Impute: ['job', 'housing', 'loan']
Columns to Delete: ['marital', 'education', 'default']
```

Next chosen columns will be imputed while the other columns with no significant variance will be dropped off from the dataset.

```
# Impute 'unknown' values for selected columns with the most frequent value
for col in columns_to_impute:
    most_frequent = data_set[col].mode()[0]
    data_set[col] = data_set[col].replace('unknown', most_frequent)

# Drop columns decided for deletion
data_set = data_set.drop(columns=columns_to_delete)

print("Data processing complete. Updated dataset:")
data_set
```

Finally all the unknowns values of the dataset are handled and the dataset is displayed again.

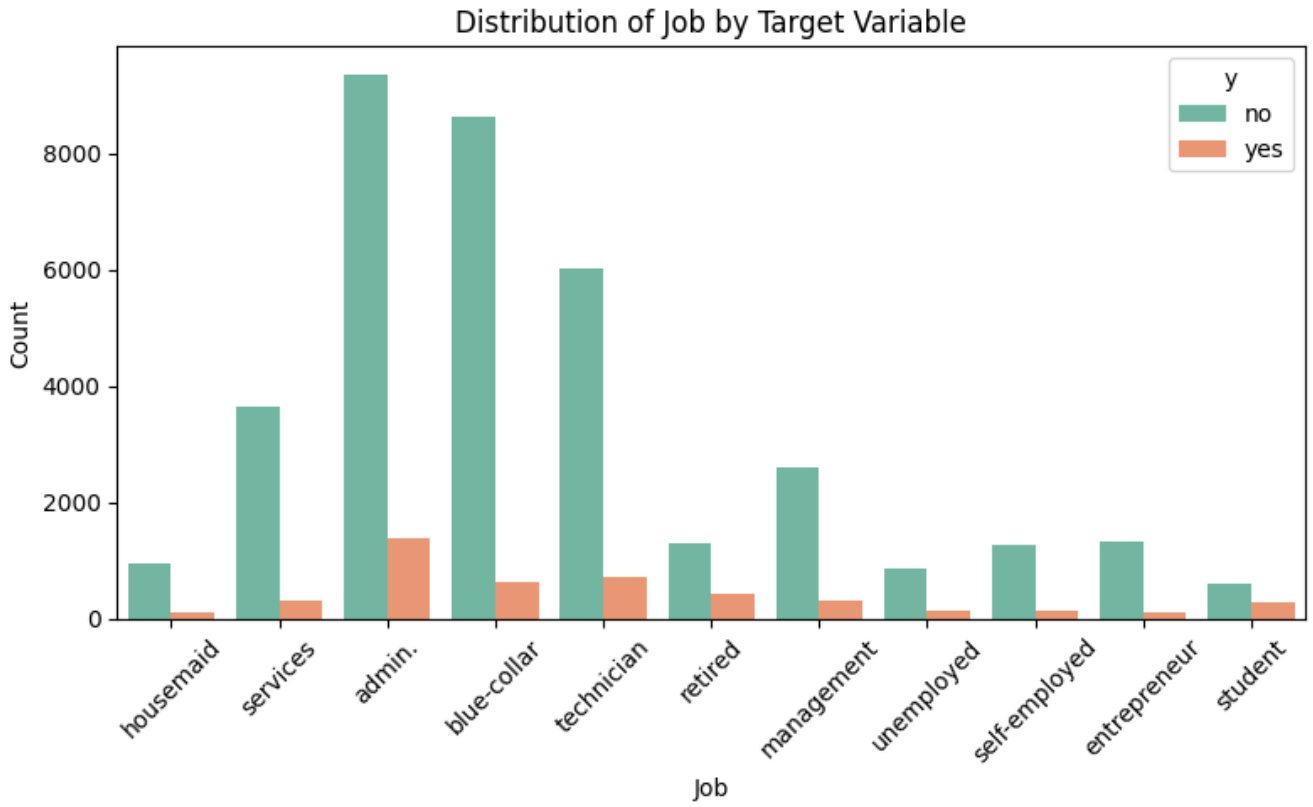
Data processing complete. Updated dataset:

	age	job	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	
0	56	housemaid	no	no	telephone	may	mon	261	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	
1	57	services	no	no	telephone	may	mon	149	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	
2	37	services	yes	no	telephone	may	mon	226	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	
3	40	admin.	no	no	telephone	may	mon	151	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	
4	56	services	no	yes	telephone	may	mon	307	1	999	0	nonexistent	1.1	93.994	-36.4	4.857	5191.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
41183	73	retired	yes	no	cellular	nov	fri	334	1	999	0	nonexistent	-1.1	94.767	-50.8	1.028	4963.6	y
41184	46	blue-collar	no	no	cellular	nov	fri	383	1	999	0	nonexistent	-1.1	94.767	-50.8	1.028	4963.6	
41185	56	retired	yes	no	cellular	nov	fri	189	2	999	0	nonexistent	-1.1	94.767	-50.8	1.028	4963.6	
41186	44	technician	no	no	cellular	nov	fri	442	1	999	0	nonexistent	-1.1	94.767	-50.8	1.028	4963.6	y
41187	74	retired	yes	no	cellular	nov	fri	239	3	999	1	failure	-1.1	94.767	-50.8	1.028	4963.6	

41188 rows × 18 columns

The dataset now has 18 columns, while the number of rows remains unchanged. With the missing value handling complete, we can proceed to EDA.

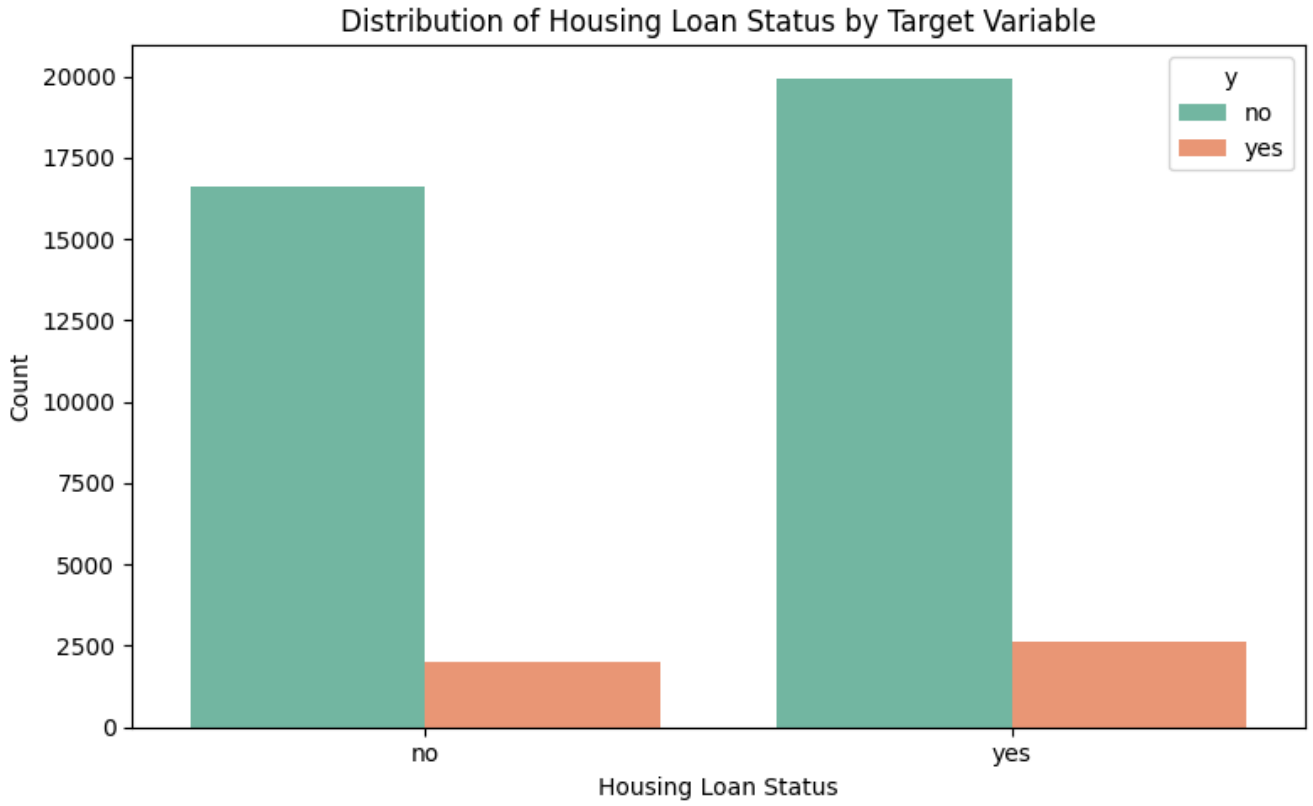
Let's start by plotting some graphs to analyze how different features affect our target variable.



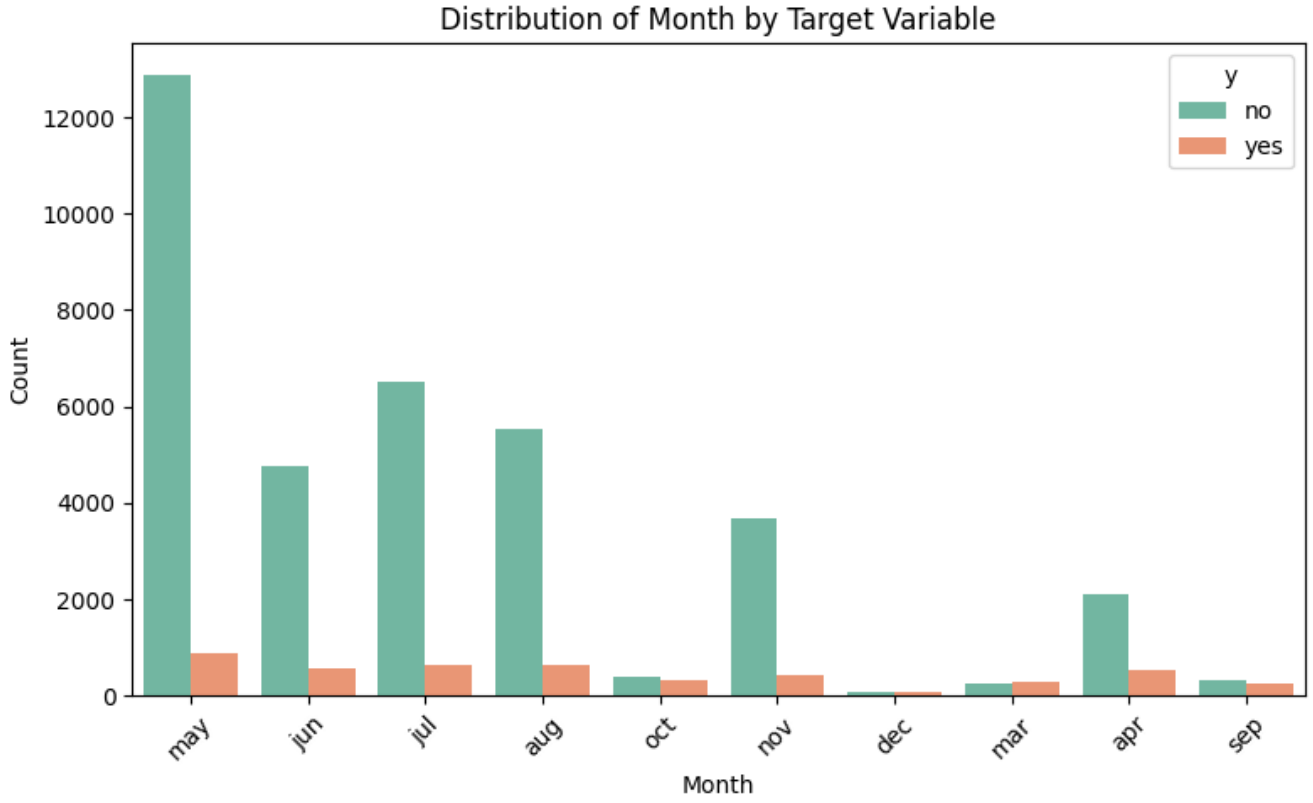
The plot above depicts how the job done by the clients affects the target variable being yes or no. It is clear that some occupations such as admin, blue-collar, and technicians have a higher percentage of subscribing for a

term deposit in the bank, while jobs like housemaid, entrepreneur, and self-employed clients have the least interest in subscribing for the term deposit.

Next,let's plot and analyze how housing loans are connected with the target variable being successful or not.



The graph above illustrates the impact of housing loans on the likelihood of clients subscribing to term deposits. It indicates that clients with housing loans at the bank are more inclined to invest in a term deposit, leading to a successful outcome for the target variable. In contrast, clients without housing loans are less likely to subscribe to a term deposit.



The plotted graph illustrates how seasonality impacts the target variables. It indicates that subscription rates are higher in May, July, and August, while lower rates are observed in December, March, and September. This pattern suggests that clients behave differently during various seasons, potentially influenced by economic factors and marketing campaigns.

3.4 Formatting Data Frame

The current data frame must be formatted before the machine learning model can function properly. In the target variable under the column 'y,' there are only two classifications: 'yes' and 'no.' These values are in string format, and we need to convert them into numerical values to predict the subscription rate using the machine learning model. Therefore, it is necessary to convert them to numerical values.

```
print(data_set['y'].head())

# Encode target variable 'y' to binary
data_set['y'] = data_set['y'].map({'yes': 1, 'no': 0})
data_set_test['y'] = data_set_test['y'].map({'yes': 1, 'no': 0})

print(data_set['y'].head())
```

```
0    no
1    no
2    no
3    no
4    no
Name: y, dtype: object
0    0
1    0
2    0
3    0
4    0
Name: y, dtype: int64
```

In the image above, if the value is 'yes', it will be encoded as the binary value 1, and if the value is 'no', it will be encoded as the binary value 0. The 'head()' function displays the target variable column of the training dataset, showing how the values appear before and after encoding.looks like before encoding and after encoding.

3.5 Feature Selection Engineering

To run the machine learning model accurately, it is essential to select features from the dataset based on their importance. This process is known as feature engineering, which involves removing unnecessary features and normalizing the data frame.

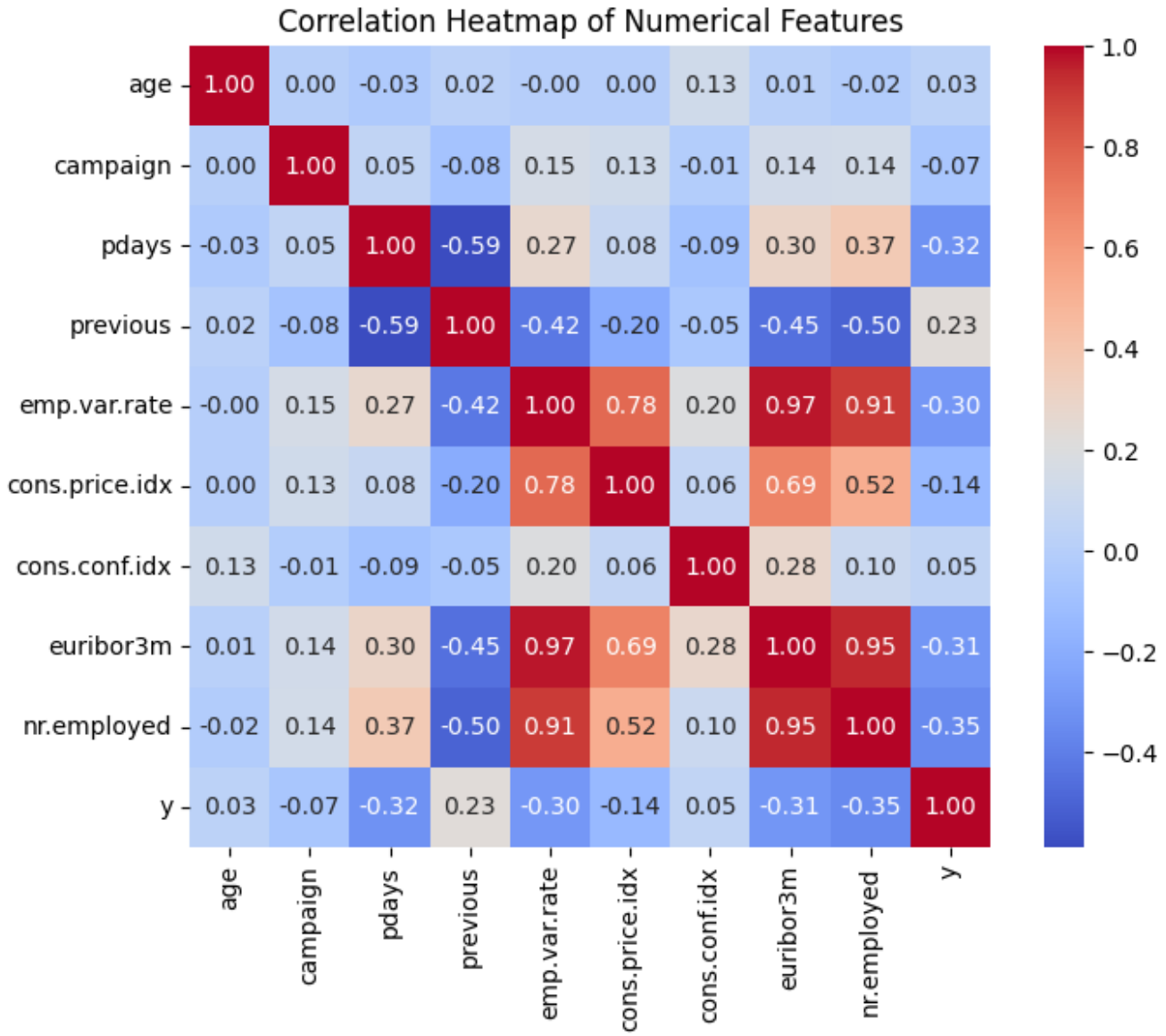
For instance, the column labeled "duration" can be eliminated because the duration of a call is not known until after the call has been completed. Additionally, the output variable 'y' is only known at the end of the call. Therefore, while this input may be useful for benchmarking purposes, it should be discarded when our goal is to create a realistic predictive model.

	age	job	housing	loan	contact	month	day_of_week	campaign	pdays	previous	poutcome	emp.var.rate	cons.price.idx	cons.conf.
0	56	housemaid	no	no	telephone	may	mon	1	999	0	nonexistent	1.1	93.994	-3
1	57	services	no	no	telephone	may	mon	1	999	0	nonexistent	1.1	93.994	-3
2	37	services	yes	no	telephone	may	mon	1	999	0	nonexistent	1.1	93.994	-3
3	40	admin.	no	no	telephone	may	mon	1	999	0	nonexistent	1.1	93.994	-3
4	56	services	no	yes	telephone	may	mon	1	999	0	nonexistent	1.1	93.994	-3
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
41183	73	retired	yes	no	cellular	nov	fri	1	999	0	nonexistent	-1.1	94.767	-5
41184	46	blue-collar	no	no	cellular	nov	fri	1	999	0	nonexistent	-1.1	94.767	-5
41185	56	retired	yes	no	cellular	nov	fri	2	999	0	nonexistent	-1.1	94.767	-5
41186	44	technician	no	no	cellular	nov	fri	1	999	0	nonexistent	-1.1	94.767	-5
41187	74	retired	yes	no	cellular	nov	fri	3	999	1	failure	-1.1	94.767	-5

41188 rows × 17 columns

The dataset now contains 17 columns while the number of rows remains unchanged.

The next step is to create a heatmap for all numerical columns to analyze the data. A very high or negative value for columns suggests a strong correlation and values close to zero can be identified as weak correlations. this can be used to determine which features should be dropped off or not from the dataset.



Now Covarience values can be calculated to find the relationship between the numerical variable and the binary variable('y'). In our data set, the target variable is the binary variable which says if a client will subscribe to a term deposit.

```
[24] # Extract numerical columns
     numerical_columns = data_set.select_dtypes(include=['int64', 'float64'])

     # Compute covariance of continuous variables with the target variable `y`
     covariance_with_y = numerical_columns.cov()['y']

     # Display the covariance values
     print("Covariance of Continuous Features with Target Variable 'y':")
     print(covariance_with_y)
```

```
Covariance of Continuous Features with Target Variable 'y':
age               0.100162
campaign         -0.058116
pdays           -19.201231
previous          0.036017
emp.var.rate     -0.148181
cons.price.idx   -0.024929
cons.conf.idx     0.080304
euribor3m        -0.168778
nr.employed      -8.102276
y                 0.099966
Name: y, dtype: float64
```

The covariance value of each variable with respect to the target variable is calculated with the function 'cov()' of the numpy in python, Then the output matrix is printed.

If the covariance value is greater than zero it identifies a direct relationship between the specific variable and the target variable and if the value is less than zero (i.e. negative) that means there is an inverse relationship between the variables.

The results show a strong negative covariance in the 'pdays' column, which can be further analyzed using logistic regression from the stats model library. Columns with near-zero negative covariance are excluded due to their lack of a significant inverse relationship between the variables.

```
Optimization terminated successfully.
         Current function value: 0.287265
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                41188
Model:                          Logit   Df Residuals:                    41178
Method:                           MLE   Df Model:                            9
Date:                Sat, 28 Dec 2024   Pseudo R-squ.:                  0.1840
Time:                        08:06:45   Log-Likelihood:                -11832.
converged:                       True   LL-Null:                       -14499.
Covariance Type:            nonrobust   LLR p-value:                     0.000
==================================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------
const            -20.2671     13.324     -1.521      0.128     -46.383       5.848
age                0.0021      0.001      1.455      0.146      -0.001       0.005
campaign          -0.0471      0.009     -5.055      0.000      -0.065      -0.029
pdays             -0.0018   7.78e-05    -23.764      0.000      -0.002      -0.002
previous          -0.2677      0.035     -7.755      0.000      -0.335      -0.200
emp.var.rate      -0.4806      0.059     -8.148      0.000      -0.596      -0.365
cons.price.idx     0.5477      0.085      6.475      0.000       0.382       0.713
cons.conf.idx      0.0230      0.005      4.453      0.000       0.013       0.033
euribor3m          0.0572      0.074      0.772      0.440      -0.088       0.202
nr.employed       -0.0060      0.001     -4.754      0.000      -0.008      -0.004
==================================================================================
```

The logistic regression chart indicates that all continuous variables impact the target variable, as none have coefficients equal to zero.

While the 'pdays' column has the lowest coefficient, this does not imply that it has no effect on the target; rather, it appears insignificant in this context. Therefore, the 'pdays' column can be removed from the dataset.

	age	job	housing	loan	contact	month	day_of_week	campaign	previous	poutcome	emp.var.rate	cons.price.idx	cons.conf.idx	eu
0	56	housemaid	no	no	telephone	may	mon	1	0	nonexistent	1.1	93.994	-36.4	
1	57	services	no	no	telephone	may	mon	1	0	nonexistent	1.1	93.994	-36.4	
2	37	services	yes	no	telephone	may	mon	1	0	nonexistent	1.1	93.994	-36.4	
3	40	admin.	no	no	telephone	may	mon	1	0	nonexistent	1.1	93.994	-36.4	
4	56	services	no	yes	telephone	may	mon	1	0	nonexistent	1.1	93.994	-36.4	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
41183	73	retired	yes	no	cellular	nov	fri	1	0	nonexistent	-1.1	94.767	-50.8	
41184	46	blue-collar	no	no	cellular	nov	fri	1	0	nonexistent	-1.1	94.767	-50.8	
41185	56	retired	yes	no	cellular	nov	fri	2	0	nonexistent	-1.1	94.767	-50.8	
41186	44	technician	no	no	cellular	nov	fri	1	0	nonexistent	-1.1	94.767	-50.8	
41187	74	retired	yes	no	cellular	nov	fri	3	1	failure	-1.1	94.767	-50.8	

41188 rows × 16 columns

The dataset now has 16 columns while maintaining the same number of rows.

3.6 Data Preprocessing

Data is preprocessed before the final step of applying the machine learning algorithm.

The first step in preprocessing is to use one-hot encoding, which creates binary columns for each unique categorical value. For example, if a categorical column has five unique values, it will generate five new binary columns, each representing a value as either true or false.let's say one categorical column had five unique values in it for each categorical value a binary columns is assigned with only the values true or false

```
# Identify categorical columns
categorical_columns = data_set.select_dtypes(include=['object']).columns

# One-Hot Encoding for Categorical Variables
data_set = pd.get_dummies(data_set, columns=categorical_columns, drop_first=True)
data_set_test = pd.get_dummies(data_set_test, columns=categorical_columns, drop_first=True)

# Align train and test datasets (to ensure same features after encoding)
data_set, data_set_test = data_set.align(data_set_test, join='inner', axis=1)

data_set
```

Here first categorical columns are identified and then a column is created for each of the columns with 'get_dummies' in pandas library.

	age	campaign	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y	job_blue-collar	...	month_may	month_nov	mont
0	56	1	0	1.1	93.994	-36.4	4.857	5191.0	0	False	...	True	False	
1	57	1	0	1.1	93.994	-36.4	4.857	5191.0	0	False	...	True	False	
2	37	1	0	1.1	93.994	-36.4	4.857	5191.0	0	False	...	True	False	
3	40	1	0	1.1	93.994	-36.4	4.857	5191.0	0	False	...	True	False	
4	56	1	0	1.1	93.994	-36.4	4.857	5191.0	0	False	...	True	False	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
41183	73	1	0	-1.1	94.767	-50.8	1.028	4963.6	1	False	...	False	True	
41184	46	1	0	-1.1	94.767	-50.8	1.028	4963.6	0	True	...	False	True	
41185	56	2	0	-1.1	94.767	-50.8	1.028	4963.6	0	False	...	False	True	
41186	44	1	0	-1.1	94.767	-50.8	1.028	4963.6	1	False	...	False	True	
41187	74	3	1	-1.1	94.767	-50.8	1.028	4963.6	0	False	...	False	True	

41188 rows × 37 columns

The dataset after performing one-hot encoding is given above, a separate binary column is created for each value in the categorical columns, Because of this, the overall number of columns increased to 37 but the number of rows is kept the same.

Up next preprocessing for numerical columns is performed, For this a method called min-max scaler is used. This technique is used especially for numerical columns in datasets, it assigns a value between 0 and 1 for each row in the columns based on the actual magnitude while preserving the relationships between the values.

```python
from sklearn.preprocessing import MinMaxScaler

# Select numerical columns from the dataset
numerical_columns = data_set.select_dtypes(include=['int64', 'float64']).columns

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Fit the scaler on the training data and transform both train and test datasets
data_set[numerical_columns] = scaler.fit_transform(data_set[numerical_columns])
data_set_test[numerical_columns] = scaler.transform(data_set_test[numerical_columns])

# Check the scaled datasets
print("Scaled Training Data:")
data_set
```

Min-Max is initially set up, after which the training data is fitted to determine the minimum and maximum values for each numerical column. These parameters are then applied to both the test and training data using the 'fit_transform' and 'transform' functions to maintain consistency and prevent data leakage.

	age	campaign	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y	job_blue-collar	...	month_may	month_nov
0	0.481481	0.000000	0.000000	0.937500	0.698753	0.60251	0.957379	0.859735	0.0	False	...	True	False
1	0.493827	0.000000	0.000000	0.937500	0.698753	0.60251	0.957379	0.859735	0.0	False	...	True	False
2	0.246914	0.000000	0.000000	0.937500	0.698753	0.60251	0.957379	0.859735	0.0	False	...	True	False
3	0.283951	0.000000	0.000000	0.937500	0.698753	0.60251	0.957379	0.859735	0.0	False	...	True	False
4	0.481481	0.000000	0.000000	0.937500	0.698753	0.60251	0.957379	0.859735	0.0	False	...	True	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...
41183	0.691358	0.000000	0.000000	0.479167	1.000000	0.00000	0.089322	0.000000	1.0	False	...	False	True
41184	0.358025	0.000000	0.000000	0.479167	1.000000	0.00000	0.089322	0.000000	0.0	True	...	False	True
41185	0.481481	0.018182	0.000000	0.479167	1.000000	0.00000	0.089322	0.000000	0.0	False	...	False	True
41186	0.333333	0.000000	0.000000	0.479167	1.000000	0.00000	0.089322	0.000000	1.0	False	...	False	True
41187	0.703704	0.036364	0.142857	0.479167	1.000000	0.00000	0.089322	0.000000	0.0	False	...	False	True

41188 rows × 37 columns

The scaled dataset shows that all numerical columns underwent min-max scaling, ensuring equal contribution from each feature to the machine learning models. Importantly, the dataset's dimension remains unchanged after min-max scaling, as it does not affect dimensions, unlike one-hot encoding.

Our dataset is now fully pre-processed and ready for machine learning model implementation.

4 Solution Methodology

4.1 Neural Network Model

This section implements a Neural Network model to classify data and predict the target variable using training and testing datasets. Inspired by the human brain, neural networks are popular machine learning models consisting of layers of interconnected nodes, (or 'neurons,') linked by weights and biases that process information and identify patterns. Real-life applications of neural networks include image recognition, natural language processing, and classification tasks.

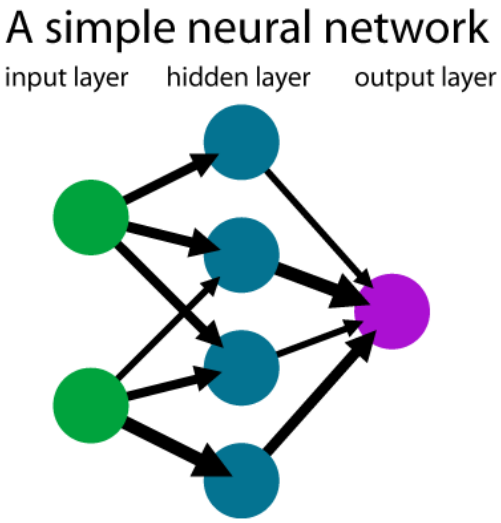


Figure 1: Layers of a neural network model

4.1.1 Key Components of Neural Netowrk

1. Input layer - This is the layer where the data enters to the neural network, each node in this layer represents a feature in the dataset(eg- age, campaign, pervious).

2. Hidden layer - This layer is in between the input layer and the output layer (As show in the Figure1).This layer conducts computations to detect patterns and relationships in data. Depending on the model, it may include one or more hidden layers, with activation functions aiding the network in uncovering complex patterns.

3. Output layer - This layers produces the final output or predictions that are gathered from the dataset with the help of the hidden layer.

A neural network model was implemented to classify the data and predict the target variable using the MLP-Classifier from the sklearn library.

```
[27] from sklearn.neural_network import MLPClassifier
     from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
     # Separate features and target variable
     X_train = data_set.drop(columns=["y"])  # Replace "target" with your target column name
     y_train = data_set["y"]
     X_test = data_set_test.drop(columns=["y"])  # Replace "target" with your target column name
     y_test = data_set_test["y"]
     # Define the neural network model
     mlp_model = MLPClassifier(
         hidden_layer_sizes=(128, 64, 32),  # Hidden layers with decreasing neurons
         activation='relu',                 # ReLU activation function
         solver='adam',                     # Adam optimizer
         max_iter=500,                      # Maximum iterations
         random_state=42,                   # Ensures reproducibility
         verbose=False                       # Print training progress
     )
     # Train the model
     mlp_model.fit(X_train, y_train)

     # Predict on the test set
     y_pred = mlp_model.predict(X_test)

     # Evaluate the model
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy: {accuracy:.4f}")

     # Detailed classification report
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred))
```

Three hidden layers with 128, 64, and 32 neurons each were used to train the model. Because of its effectiveness in managing non-linearity, the Rectified Linear Unit (ReLU) activation function was used. Because of its versatility and computing efficiency, the Adam optimizer was employed for gradient-based optimization. The model performed well after training, as evidenced by its 95% accuracy rate.

4.1.2 Analysis of Results

The 95% accuracy rate of the model is a clear sign that the neural network is capable of correctly identifying the provided dataset. But accuracy by itself doesn't paint a full picture. Metrics like accuracy, recall, and F1-score were included in the classification report that was created in order to obtain more in-depth understanding. These metrics demonstrate how well the model manages both positive and negative classes.

```
Accuracy: 0.9352

Classification Report:
              precision    recall  f1-score   support

         0.0       0.94      0.99      0.96      3668
         1.0       0.83      0.51      0.63       451

    accuracy                           0.94      4119
   macro avg       0.89      0.75      0.80      4119
weighted avg       0.93      0.94      0.93      4119


Confusion Matrix:
[[3622   46]
 [ 221  230]]
```

In summary, the neural network was a good fit for the classification challenge since it showed outstanding predicted performance on the dataset. Its performance could be enhanced by more hyperparameter exploration and tuning.

4.1.3 Hyperparameter tuning

Here the function 'GridSearchCV' from the library 'sklearn' will allow us to perform hyperparameter tuning on the nerual network model. This function will allow us to test various combinations of hyperparameters and find the optimal configuration for the model, Let us implement this.

```
[ ]  from sklearn.neural_network import MLPClassifier
     from sklearn.model_selection import GridSearchCV
     from sklearn.metrics import classification_report, accuracy_score

     # Define the MLPClassifier with a fixed random state for reproducibility
     mlp_model = MLPClassifier(max_iter=300, random_state=42)

     # Define a smaller hyperparameter grid for optimization
     param_grid = {
         'hidden_layer_sizes': [(64, 32), (128, 64), (64, 32, 16)],  # Fewer combinations
         'activation': ['relu', 'tanh'],  # Common activation functions
         'solver': ['adam'],  # Focusing on one efficient optimizer
         'learning_rate_init': [0.001, 0.01],  # Limited learning rate values
     }

     # Use GridSearchCV with 2-fold cross-validation for faster evaluation
     grid_search = GridSearchCV(
         estimator=mlp_model,
         param_grid=param_grid,
         cv=2,  # Reduced folds
         n_jobs=-1,  # Use all CPU cores
         verbose=2  # Display progress
     )
```

```
     # Train the model using GridSearchCV
     grid_search.fit(X_train, y_train)

     # Output the best parameters and score
     print(f"Best Parameters: {grid_search.best_params_}")
     print(f"Best Cross-Validation Score: {grid_search.best_score_:.4f}")

     # Predict on the test data using the best model
     best_model = grid_search.best_estimator_
     y_pred = best_model.predict(X_test)

     # Evaluate the model
     print("\nTest Accuracy:", accuracy_score(y_test, y_pred))
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred))
```

Above code performs hyperparameter tuning using 'GridSearchCV' for the Neural network model to identify the parameters for optimal performance. 'MLPClassifier' model from 'sklearn' library is used for this which is a multilayer perception. Parameters that were tested are,

1. hidden_layer_sizes - This is the structure of the nerual network which specifies the number of neurons in each hidden layer, Three configurations were tested under this which are (64, 32), (128, 64), and (64, 32, 16).
2. activation - 'relu'(Rectified liner unit) and 'tanh'(Hyperbolic tangent) are the two comment activation functions used for this model.
3. solver - For this a popular and a common efficient optimizer was used, 'adam'.
4. learning_rate_init - this is the initial learning rate of the model, tested values under this is 0.001 and 0.01

Above Hyperparametrs were used in the process of this and the function 'best_params' were used to identify the best hyperparameter from the given values.

In addition to this, 2 cross-validation(cv=2) and parallel processing(n_jobs=1) is used to speed up the execution process . The most optimal settings and cross-validation precision were chosen, and the optimized model was tested, by utilizing accuracy and a classification report, on the sets of the evaluation metrics of the prediction. This method guarantees a good balance between performance and execution time.

```
Fitting 2 folds for each of 12 candidates, totalling 24 fits
Best Parameters: {'activation': 'tanh', 'hidden_layer_sizes': (64, 32, 16), 'learning_rate_init': 0.01, 'solver': 'adam'}
Best Cross-Validation Score: 0.5393

Test Accuracy: 0.9249817916970139

Classification Report:
              precision    recall  f1-score   support

         0.0       0.93      0.99      0.96      3668
         1.0       0.83      0.39      0.54       451

    accuracy                           0.92      4119
   macro avg       0.88      0.69      0.75      4119
weighted avg       0.92      0.92      0.91      4119
```

After the process of hyperparameter tuning, accuracy of the nerual network model was reduced to 0.92%. This is totally normal since the goal of hyperparameter tuning is not to increase the accuracy but to find a model configuration that generalizes better to unseen data.

4.2 Random Forest Classification

Several decision trees are combined in the Random Forest ensemble learning technique to increase prediction accuracy and decrease overfitting. A random subset of the data is used to train each tree in the forest, and predictions are generated by summing up each tree's output (majority vote for classification tasks). This method improves the generalization and resilience of the model. Because Random Forest can efficiently handle big datasets and intricate feature connections, it is often employed.

To forecast the target variable (y) in our dataset, we use a Random Forest Classifier in the code snippet that follows. Important hyperparameters like 'random_state', which guarantees reproducibility, and 'n_estimators', which specifies the number of decision trees in the forest, are set up in the model. In order to regulate the trees' growth and prevent overfitting, we further set 'min_samples_split' and 'min_samples_leaf'. We use the test dataset to assess the model's performance after it has been trained on the training dataset.

```
[29] # Import necessary libraries
     from sklearn.ensemble import RandomForestClassifier
     from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

     # Separate features and target variable for training and testing datasets
     X_train = data_set.drop('y', axis=1)  # Replace 'target_column' with your actual target column name
     y_train = data_set['y']
     X_test = data_set_test.drop('y', axis=1)    # Same as above for test data
     y_test = data_set_test['y']

     # Initialize Random Forest Classifier
     rf_model = RandomForestClassifier(
         n_estimators=100,        # Number of trees in the forest
         max_depth=None,          # Maximum depth of the tree (default: None, grow until all leaves are pure)
         random_state=42,         # Ensures reproducibility
         min_samples_split=2,     # Minimum number of samples required to split an internal node
         min_samples_leaf=1       # Minimum number of samples required to be at a leaf node
     )

     # Train the model
     rf_model.fit(X_train, y_train)

     # Predict on test data
     y_pred = rf_model.predict(X_test)

     # Evaluate the model
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy: {accuracy:.4f}")

     # Print detailed classification report
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred))
```

We then calculate the accuracy, output the classification report, and visualize the confusion matrix to evaluate the performance of our Random Forest model, after running the above code. While the accuracy score provides a high level understanding of the performance of model,the classification report gives a more detailed metrics for each class (precision recall and f1-score). Model evaluation allows for verification if the respective testing set is able to generalize, and a confusion matrix is beneficial to this as it allows one to visualize the model's predictions with actual labels, to observe the misclassified labels and accuracy of labels.

4.2.1 Analysis of Results

```
Accuracy: 0.9852

Classification Report:
              precision    recall  f1-score   support

         0.0       0.99      1.00      0.99      3668
         1.0       0.98      0.88      0.93       451

    accuracy                           0.99      4119
   macro avg       0.98      0.94      0.96      4119
weighted avg       0.99      0.99      0.98      4119


Confusion Matrix:
[[3661    7]
 [  54  397]]
```

We note the following based on the evaluation results:

- The model's ability to accurately forecast the target variable is demonstrated by its 99% accuracy rate.
- The model exhibits consistent performance across all classes, with good precision and recall for the majority of classes, according to the classification report.
- According to the confusion matrix, there are very few misclassification mistakes and most predictions are properly categorized.

All things considered, the Random Forest Classifier works well as a model for this dataset, providing a balance between interpretability and accuracy while skillfully managing feature interactions and dataset complexity.

4.2.2 Hyperparameter Tuning for Random Forest Classification Model

A critical step in creating a successful machine learning model is hyperparameter tweaking. We employ automated tuning techniques to determine the ideal set of parameters that optimize the model's performance rather than choosing hyperparameters by hand. We used RandomizedSearchCV, a quicker substitute for GridSearchCV, for our Random Forest Classifier.

From a specified parameter distribution, RandomizedSearchCV chooses a certain number of random hyperparameter combinations, then uses cross-validation to assess each one's performance. Because it does not test every conceivable combination of parameters, this method is quicker than GridSearchCV. The following are the main parameters we adjusted during this process:

- n_estimators: The number of trees in the forest (e.g., 50, 100, 200).
- max_depth: Maximum depth of each tree (10, 20, None, etc.)
- min_samples_split: Minimum number of samples to split an internal node (2, 5, 10)
- min_samples_leaf: The minimal number of samples required to be at a leaf node (1, 2, 4, etc.).

To assess the effect of all combinations, we ran the RandomizedSearchCV algorithm to evaluate 20 random combinations of these hyperparameters and estimated how well the system would perform using 3-fold cross-validation. We then used this best parameter combination to train the Random Forest model and test it on the test dataset.

The code is below for how we can implement RandomizedSearchCV with Random Forest Classifier:

```
[35] from sklearn.ensemble import RandomForestClassifier
     from sklearn.model_selection import RandomizedSearchCV
     from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
     import numpy as np

     # Define the parameter distribution
     param_dist = {
         'n_estimators': [50, 100, 200],      # Number of trees
         'max_depth': [10, 20, None],         # Maximum depth of trees
         'min_samples_split': [2, 5, 10],     # Minimum samples required to split a node
         'min_samples_leaf': [1, 2, 4],       # Minimum samples at a leaf node
     }

     # Initialize Random Forest Classifier
     rf_model = RandomForestClassifier(random_state=42)

     # Initialize RandomizedSearchCV
     random_search = RandomizedSearchCV(
         estimator=rf_model,
         param_distributions=param_dist,
         n_iter=20,             # Number of parameter settings sampled
         scoring='accuracy',    # Using accuracy as the metric
         cv=3,                  # Cross-validation folds
         random_state=42,       # Ensures reproducibility
         n_jobs=-1,             # Use all CPU cores
         verbose=1              # Monitor progress
     )

     # Fit RandomizedSearchCV to training data
     random_search.fit(X_train, y_train)
```

```
     # Get the best parameters and the corresponding model
     best_params = random_search.best_params_
     best_model = random_search.best_estimator_

     print("Best Parameters:", best_params)

     # Evaluate the best model on the test set
     y_pred = best_model.predict(X_test)
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy: {accuracy:.4f}")

     # Print detailed classification report and confusion matrix
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred))
     print("\nConfusion Matrix:")
     print(confusion_matrix(y_test, y_pred))
```

After executing the above code, optimal hyperparameters were identified and an accuracy of 91% was achieved, which is shown in the figure below.

```
Fitting 3 folds for each of 20 candidates, totalling 60 fits
Best Parameters: {'n_estimators': 100, 'min_samples_split': 2, 'min_samples_leaf': 2, 'max_depth': 10}
Accuracy: 0.9148

Classification Report:
              precision    recall  f1-score   support

         0.0       0.92      0.99      0.95      3668
         1.0       0.80      0.30      0.43       451

    accuracy                           0.91      4119
   macro avg       0.86      0.64      0.69      4119
weighted avg       0.91      0.91      0.90      4119


Confusion Matrix:
[[3634   34]
 [ 317  134]]
```

By determining the ideal configuration, the hyperparameter tuning procedure not only enhanced the model's performance but also maximized its efficiency. An efficient and successful technique for fine-tuning the Random Forest Classifier was RandomizedSearchCV.

5 Model Analysis and Evaluation

Under this section of this report, each model implemented will be evaluated with their accuracy values. This helps us identify the best model that suits us for our purposes.

The table drawn below will show how the data is analyzed.

5.1 Model Analysis for 'term deposit' being no (class 0) Results

MODEL	ACCURACY	PRECISION	RECALL	F1-SCORE	HYPERPARAMETERS
Neural Network	0.9352	0.94	0.99	0.96	hidden_layer_sizes=(64, 32, 16), activation='relu', solver='adam', max_iter=500
Random Forest Classification	0.9852	0.90	1.00	0.99	n_estimators=100, max_depth=None, min_samples_split=2, min_samples_leaf=1
Neural Network (Hyperparameter Tuned)	0.9249	0.93	0.99	0.96	hidden_layer_sizes=(64, 32, 16), activation='tanh', solver='adam', learning_rate='0.01'
Random Forest (Hyperparameter Tuned)	0.9148	0.92	0.99	0.95	n_estimators=100, max_depth=10, min_samples_split=2, min_samples_leaf=2, bootstrap=True

Table 1: Model Evaluation Metrics and Hyperparameters

This table analyses the result for the term deposit being no, in other words class 0 in the classification report which is printed after the execution of the model.

5.2 Model Analysis for 'term deposit' being yes (class 1)

MODEL	ACCURACY	PRECISION	RECALL	F1-SCORE	HYPERPARAMETERS
Neural Network	0.9352	0.83	0.51	0.63	hidden_layer_sizes=(64, 32, 16), activation='relu', solver='adam', max_iter=500
Random Forest Classification	0.9852	0.98	0.88	0.93	n_estimators=100, max_depth=None, min_samples_split=2, min_samples_leaf=1
Neural Network (Hyperparameter Tuned)	0.9249	0.83	0.39	0.54	hidden_layer_sizes=(64, 32, 16), activation='tanh', solver='adam', learning_rate='0.01'
Random Forest (Hyperparameter Tuned)	0.9148	0.80	0.30	0.43	n_estimators=100, max_depth=10, min_samples_split=2, min_samples_leaf=2, bootstrap=True

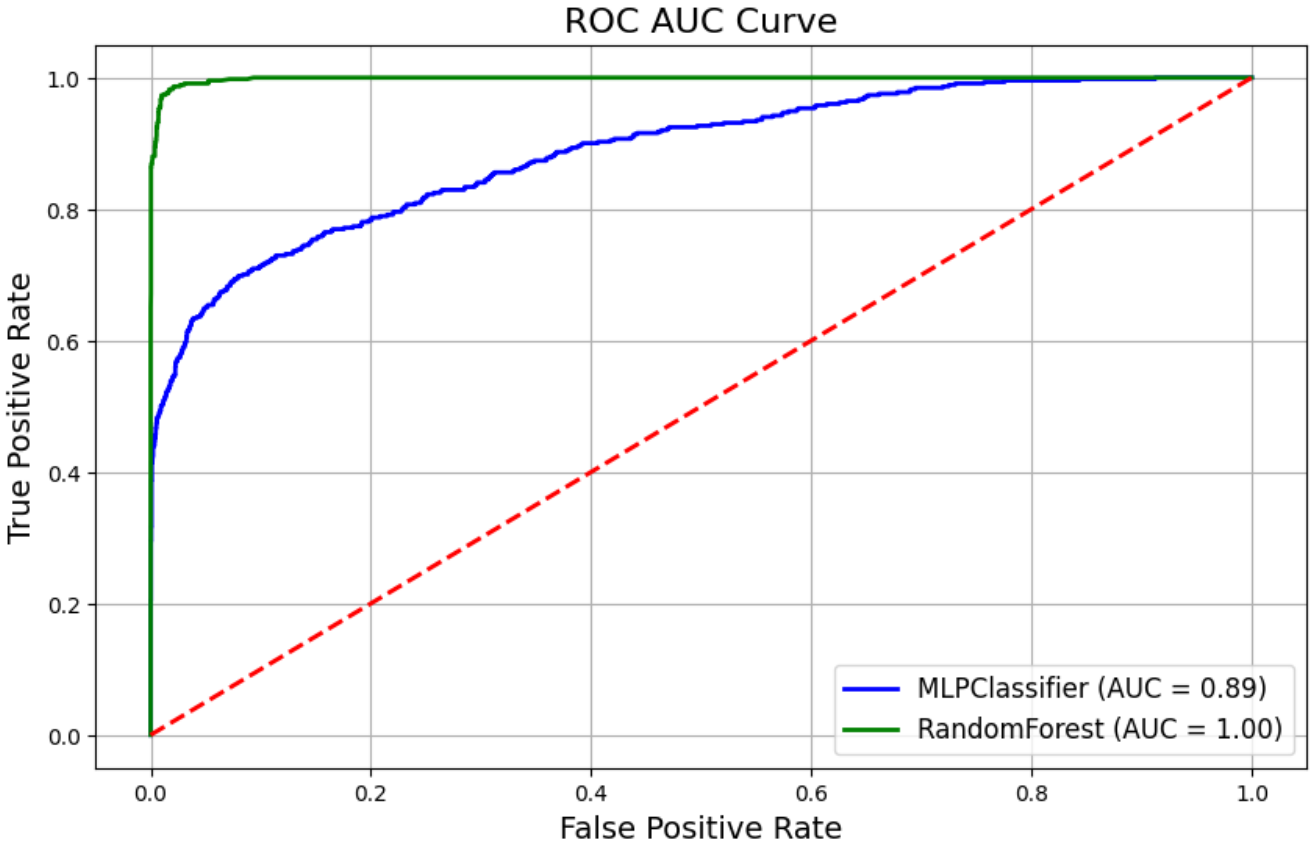
Table 2: Model Evaluation Metrics and Hyperparameters

This table analyses the result for the term deposit being yes, in other words class 1 in the classification report which is printed after the execution of the model.

Metrics like as the ROC AUC score, which is the area under the ROC curve, can be used to evaluate the effectiveness and predictability of our model. With an AUC value of 1.0, a perfect classifier would have a 100% true positive rate and a 0% false negative rate. On the other hand, an AUC of 0.5 from a random classifier would indicate no discriminating strength between positive and negative classifications. The ROC curve itself offers a thorough assessment of the model's performance over a range of classification thresholds by illuminating the trade-off between the true positive rate (sensitivity) and the false positive rate (1-specificity). Let's plot the ROC curve for the two model and analyze the area under the curve.

5.3 ROC - AUC Curve

A graphical tool for assessing a binary classification model's performance is the ROC-AUC curve. At different threshold levels, the True Positive Rate (Sensitivity) is shown versus the False Positive Rate (1-Specificity) on the ROC curve. The model's overall capacity to differentiate between the two classes is measured by the AUC (Area Under the Curve), where values nearer 1 denote superior performance. Because it considers the model's ability to rank predictions rather than just accuracy, it offers a useful tool for comparing models, particularly when working with unbalanced datasets.



The MLPClassifier and RandomForestClassifier's performance in class distinction is contrasted in the ROC-AUC curve above. With an AUC of 1.00, the RandomForestClassifier performs flawlessly in classification. The MLPClassifier, on the other hand, has an AUC of 0.89, which is somewhat less successful in class separation but still represents a great performance.

This outcome shows how reliable the Random Forest model is for this dataset, most likely as a result of its ensemble learning methodology and capacity to manage intricate patterns. But the MLPClassifier also does well, indicating that neural networks can successfully identify connections in the data. The comparison emphasizes how crucial it is to assess several models in order to determine which one performs best for a particular issue.

6 Limitations and Future Enhancements

6.1 Limitations

1. Dataset Characteristics-
   - Scope of the dataset used in this project is limited since it only includes data from the specific bank.Hence it can results in models that may not generalize well for other industries or geographical contents.
   - Missing values with the term "Unknown" were present in the dataset, these columns are imputed and dropped off based on logical assumptions but this may not represent real-world scenarios.
2. Imbalanced Classes -
   - Dataset has an imbalance between the positive and negative classes in the target variable, Even though ROC-AUC curve were plotted to find the optimal solution it may still affect the model's predictive accuracy, especially for minority class.
   - Missing values with the term "Unknown" were present in the dataset, these columns are imputed and dropped off based on logical assumptions but this may not represent real-world scenarios.
3. Hyperparameter Optimization
   - For each an every model hyperparameter tuning were done but it was limited to small subsets of configurations due to computational constraints. this might have prevented finding the most optimal model as the solution
4. Ethical Considerations -
   - There was not a thorough examination of the moral ramifications of exploiting customer data for predictive modeling. This includes worries about potential biases in forecasts, data privacy, and model usage openness.

6.2 Future Enhancements

1. Improving Data Quality -
   - Collect data from different bank and banks in different regions as well to improve the models generalizability.
   - Exploring advanced imputation techniques, such as using machine learning models for imputation, to better handle missing values.
2. Addressing Class Imbalance -
   - To get a more balanced dataset, apply techniques such as SMOTE oversampling or undersampling the dominant class.
   - utilizing cost-sensitive learning strategies to punish minority class misrepresentation.
3. Extensive Hyperparameter Tuning -
   - use automated technologies for more thorough hyperparameter searches, such as genetic algorithms or Bayesian optimization.

4. Ethical and Legal Compliance -
    - include a thorough ethical study that addresses any biases in the dataset, data protection, and compliance with laws like the GDPR.
    - establishing standards for the models' appropriate implementation in actual banking applications.

7 Appendix

Let us first import the necessary libraries.

```
# Import necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

let's make the dataframes for training data and test data, we have already splitted the files beforehand for convenience

```
#this is the main file with all the data for training
train = pd.read_csv("/content/drive/MyDrive/ML/bank-additional-full.csv", delimiter=';')
# this is the testing data, this file consists 10% of the data in the bank-additional-full.csv
    file
test = pd.read_csv("/content/drive/MyDrive/ML/bank-additional.csv",delimiter=';')

test = test[train.columns]

# view the data
print("Train Data:")
data_set = train
data_set
```

```
#view the data
print("Test Data:")
data_set_test = test
data_set_test
```

check the data types for all the columns

```
data_set.dtypes
```

Let's search if data contains null values

```
print("\nMissing Values:")
print(data_set.isnull().sum())
```

Upon observations, it could be seen that there are no null values instead null values there are columns are filled with the term "unknow"

Let us check for term "Unknow" in the columns, this can be checked by checking the possible values in each categorical column.

```
# Select categorical columns (object dtype)
categorical_columns = data_set.select_dtypes(include=['object']).columns

# Check for "unknown" in each categorical column
```

```
unknown_counts = {}
for column in categorical_columns:
    count = data_set[column].str.contains("unknown", na=False).sum()  # Count occurrences of "
    unknown"
    unknown_counts[column] = count

# Print the results
for col, count in unknown_counts.items():
    print(f"Column '{col}' has {count} occurrences of 'unknown'.")
```

Next, let's analyze our taget variable

```
print("\nClass Distribution:")
print(data_set['y'].value_counts())
```

To decide whether to drop or impute the columns or rows with "unknow" term we need to find if there is a hidden connection between the columns and the output(y).

First let's calculate the 'yes' porportion of the target varaible

```
# Calculate the overall 'yes' rate in the target variable (assume target column is 'y')
overall_yes_rate = (data_set['y'].value_counts(normalize=True)['yes']) * 100

print(f"Overall 'Yes' Rate: {overall_yes_rate:.2f}%")
```

Now let's identify the columns with the rows containing the term "unknown"

```
# Identify columns containing 'unknown' values
unknown_columns = [col for col in data_set.columns if 'unknown' in data_set[col].unique()]

print(f"Columns with 'unknown' values: {unknown_columns}")
```

For each column with 'unknown' values, calculate the proportion of 'yes' in rows where the column is 'unknown'. This helps us understand if these rows behave differently from the overall dataset.

```
# Analyze the 'yes' rate for rows where each column has 'unknown' values
unknown_analysis = {}
#Calculating and diplaying the proportion for each column
for col in unknown_columns:
    unknown_yes_rate = (data_set[data_set[col] == 'unknown']['y'].value_counts(normalize=True).get
    ('yes', 0)) * 100
    unknown_analysis[col] = unknown_yes_rate
    print(f"{col}: 'Yes' Rate for 'unknown' rows = {unknown_yes_rate:.2f}%")
```

Based on the calculated rates, decide whether to impute 'unknown' values with the most frequent value or delete the column entirely.

```
# Decide which columns to impute or delete
columns_to_impute = []
columns_to_delete = []

for col, unknown_yes_rate in unknown_analysis.items():
    if abs(unknown_yes_rate - overall_yes_rate) <= 2:
        columns_to_impute.append(col)
    else:
        columns_to_delete.append(col)

print(f"Columns to Impute: {columns_to_impute}")
print(f"Columns to Delete: {columns_to_delete}")
```

Now, we impute the 'unknown' values for selected columns with the most frequent value. For columns with no significant variance, we drop them.

```
# Impute 'unknown' values for selected columns with the most frequent value
for col in columns_to_impute:
    most_frequent = data_set[col].mode()[0]
    data_set[col] = data_set[col].replace('unknown', most_frequent)

# Drop columns decided for deletion
data_set = data_set.drop(columns=columns_to_delete)

print("Data processing complete. Updated dataset:")
data_set
```

Check for duplicate rows and and columns to drop

```
# Check for duplicate rows
print(f"Duplicate rows: {data_set.duplicated().sum()}")

# Check for duplicate columns
print(f"Duplicate columns: {data_set.T.duplicated().sum()}")
```

Even though the data shows there are duplicate rows, there are not. this is becuase some rows contain same value for each columns.

Let us plot some graphs for EDA to indentify the relationship with target variable and the columns

```
plt.figure(figsize=(8, 5))
sns.countplot(data=data_set, x='job', hue='y', palette='Set2')
plt.title('Distribution of Job by Target Variable')
plt.xlabel('Job')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
plt.figure(figsize=(8, 5))
sns.countplot(data=data_set, x='housing', hue='y', palette='Set2')
plt.title('Distribution of Housing Loan Status by Target Variable')
plt.xlabel('Housing Loan Status')
plt.ylabel('Count')
plt.tight_layout()
plt.show()
```

```
plt.figure(figsize=(8, 5))
sns.countplot(data=data_set, x='month', hue='y', palette='Set2')
plt.title('Distribution of Month by Target Variable')
plt.xlabel('Month')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Target variable('y') was encoded into binary values, yes - 1 and no - 0.

```
print(data_set['y'].head())

# Encode target variable 'y' to binary
```

```
data_set['y'] = data_set['y'].map({'yes': 1, 'no': 0})
data_set_test['y'] = data_set_test['y'].map({'yes': 1, 'no': 0})

print(data_set['y'].head())
```

Duration column can be dropped since it has no effect of the target varible, duration value is not knows until the empolyee finishes the call with client.

```
# Drop the 'duration' column from the dataset
data_set = data_set.drop(columns=['duration'], errors='ignore')
data_set
```

Let us plot a correlation heat map to identify the relationship between the numerical columns of the dataset and the target variable.

```
# Extract numerical columns
numerical_columns = data_set.select_dtypes(include=['int64', 'float64'])
numerical_columns['y'] = data_set['y']

# Calculate the correlation matrix
correlation_matrix = numerical_columns.corr()

# Plot the heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(
    correlation_matrix,
    annot=True,
    fmt=".2f",
    cmap="coolwarm",
    cbar=True,
    square=True
)
plt.title("Correlation Heatmap of Numerical Features")
plt.show()
```

We can also calculate the covariance values of the continuous functions with the target varaible.

```
# Extract numerical columns
numerical_columns = data_set.select_dtypes(include=['int64', 'float64'])

# Compute covariance of continuous variables with the target variable `y`
covariance_with_y = numerical_columns.cov()['y']

# Display the covariance values
print("Covariance of Continuous Features with Target Variable 'y':")
print(covariance_with_y)
```

If the covarince value is greater than zero it shows a strog direct relationship between the variables and the values lesser than zero shows inverse relationship between the variables. we can confirm these negative values furture with logistics regression

```
import statsmodels.api as sm
# Select only numerical columns and the target variable
numerical_columns = data_set.select_dtypes(include=['int64', 'float64']).drop(columns=['y'])

# Add a constant (for the intercept term in the logistic regression)
X = sm.add_constant(numerical_columns)  # Independent variables
```

```
y = data_set['y']  # Target variable

# Fit the logistic regression model
logistic_model = sm.Logit(y, X)
result = logistic_model.fit()

# Print the summary
print(result.summary())
```

'pdays' column has the lowest coefficient, this does not imply that it has no effect on the target; rather, it appears insignificant in this context. Therefore, the 'pdays' column can be removed from the dataset.

```
# Drop the 'duration' column from the dataset
data_set = data_set.drop(columns=['pdays'], errors='ignore')
data_set
```

ne Hot Encoding was used to generate binary columns for every categorical value

```
# Identify categorical columns
categorical_columns = data_set.select_dtypes(include=['object']).columns

# One-Hot Encoding for Categorical Variables
data_set = pd.get_dummies(data_set, columns=categorical_columns, drop_first=True)
data_set_test = pd.get_dummies(data_set_test, columns=categorical_columns, drop_first=True)

# Align train and test datasets (to ensure same features after encoding)
data_set, data_set_test = data_set.align(data_set_test, join='inner', axis=1)

data_set
```

Min-Max scaling is done for the numerical columns to ensure all the featurers contribute equally for the final machine learning models.This also helps preserve relationships for the values.

```
from sklearn.preprocessing import MinMaxScaler

# Select numerical columns from the dataset
numerical_columns = data_set.select_dtypes(include=['int64', 'float64']).columns

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Fit the scaler on the training data and transform both train and test datasets
data_set[numerical_columns] = scaler.fit_transform(data_set[numerical_columns])
data_set_test[numerical_columns] = scaler.transform(data_set_test[numerical_columns])

# Check the scaled datasets
print("Scaled Training Data:")
data_set
```

Now let's implement the neural netowrk model using the dataset we preprocessed

```
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
# Separate features and target variable
X_train = data_set.drop(columns=["y"])
y_train = data_set["y"]
X_test = data_set_test.drop(columns=["y"])
y_test = data_set_test["y"]
```

```
# Define the neural network model
mlp_model = MLPClassifier(
    hidden_layer_sizes=( 64, 32, 16),  # Hidden layers with decreasing neurons
    activation='relu',                 # ReLU activation function
    solver='adam',                     # Adam optimizer
    max_iter=500,                      # Maximum iterations
    random_state=42,                   # Ensures reproducibility
    verbose=False                       # Print training progress
)
# Train the model
mlp_model.fit(X_train, y_train)

# Predict on the test set
y_pred = mlp_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")

# 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))
```

For furture improvments, let us perform hyperparameter tuning with the help of 'GridSearchCV' from the library 'sklearn'

```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score

# Define the MLPClassifier with a fixed random state for reproducibility
mlp_model = MLPClassifier(max_iter=300, random_state=42)

# Define a smaller hyperparameter grid for optimization
param_grid = {
    'hidden_layer_sizes': [(64, 32), (128, 64), (64, 32, 16)],  # Fewer combinations
    'activation': ['relu', 'tanh'],  # Common activation functions
    'solver': ['adam'],  # Focusing on one efficient optimizer
    'learning_rate_init': [0.001, 0.01],  # Limited learning rate values
}

# Use GridSearchCV with 2-fold cross-validation for faster evaluation
grid_search = GridSearchCV(
    estimator=mlp_model,
    param_grid=param_grid,
    cv=2,  # Reduced folds
    n_jobs=-1,  # Use all CPU cores
    verbose=2  # Display progress
)

# Train the model using GridSearchCV
grid_search.fit(X_train, y_train)

# Output the best parameters and score
```

```
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Best Cross-Validation Score: {grid_search.best_score_:.4f}")

# Predict on the test data using the best model
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

# Evaluate the model
print("\nTest Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Next, let's implement random forest classification model

```
# Import necessary libraries
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Separate features and target variable for training and testing datasets
X_train = data_set.drop('y', axis=1)
y_train = data_set['y']
X_test = data_set_test.drop('y', axis=1)    # Same as above for test data
y_test = data_set_test['y']

# Initialize Random Forest Classifier
rf_model = RandomForestClassifier(
    n_estimators=100,        # Number of trees in the forest
    max_depth=None,          # Maximum depth of the tree (default: None, grow until all leaves are
     pure)
    random_state=42,         # Ensures reproducibility
    min_samples_split=2,     # Minimum number of samples required to split an internal node
    min_samples_leaf=1       # Minimum number of samples required to be at a leaf node
)

# Train the model
rf_model.fit(X_train, y_train)

# Predict on test data
y_pred = rf_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")

# Print detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Display confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

For furture improvments, let us perform hyperparameter tuning with the help of 'GridSearchCV' from the library 'sklearn'

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

```python
import numpy as np

# Define the parameter distribution
param_dist = {
    'n_estimators': [50, 100, 200],      # Number of trees
    'max_depth': [10, 20, None],         # Maximum depth of trees
    'min_samples_split': [2, 5, 10],     # Minimum samples required to split a node
    'min_samples_leaf': [1, 2, 4],       # Minimum samples at a leaf node
}

# Initialize Random Forest Classifier
rf_model = RandomForestClassifier(random_state=42)

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rf_model,
    param_distributions=param_dist,
    n_iter=20,              # Number of parameter settings sampled
    scoring='accuracy',     # Using accuracy as the metric
    cv=3,                   # Cross-validation folds
    random_state=42,        # Ensures reproducibility
    n_jobs=-1,              # Use all CPU cores
    verbose=1               # Monitor progress
)

# Fit RandomizedSearchCV to training data
random_search.fit(X_train, y_train)

# Get the best parameters and the corresponding model
best_params = random_search.best_params_
best_model = random_search.best_estimator_

print("Best Parameters:", best_params)

# Evaluate the best model on the test set
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")

# Print detailed classification report and confusion matrix
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

AOC- ROC curve

Let's use the roc_curve and auc functions to calculate the False Positive Rate (FPR), True Positive Rate (TPR), and Area Under the Curve (AUC).

```python
# Import necessary libraries
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Define Neural Network (MLPClassifier)
mlp_model = MLPClassifier(
    hidden_layer_sizes=(64, 32, 16),  # Example architecture
```

```
    activation='relu',
    solver='adam',
    max_iter=500,
    random_state=42
)

# Define Random Forest Classifier
rf_model = RandomForestClassifier(
    n_estimators=100,
    random_state=42,
    n_jobs=-1
)

# Train both models on training data
mlp_model.fit(X_train, y_train)
rf_model.fit(X_train, y_train)

# Generate predicted probabilities for the positive class
mlp_y_pred_proba = mlp_model.predict_proba(X_test)[:, 1]
rf_y_pred_proba = rf_model.predict_proba(X_test)[:, 1]

# Compute ROC Curve and AUC for both models
fpr_mlp, tpr_mlp, _ = roc_curve(y_test, mlp_y_pred_proba)
roc_auc_mlp = auc(fpr_mlp, tpr_mlp)

fpr_rf, tpr_rf, _ = roc_curve(y_test, rf_y_pred_proba)
roc_auc_rf = auc(fpr_rf, tpr_rf)

# Plotting the ROC Curve
plt.figure(figsize=(10, 6))
plt.plot(fpr_mlp, tpr_mlp, label=f"MLPClassifier (AUC = {roc_auc_mlp:.2f})", lw=2, color='blue')
plt.plot(fpr_rf, tpr_rf, label=f"RandomForest (AUC = {roc_auc_rf:.2f})", lw=2, color='green')
plt.plot([0, 1], [0, 1], color='red', linestyle='--', lw=2)

plt.title('ROC AUC Curve', fontsize=16)
plt.xlabel('False Positive Rate', fontsize=14)
plt.ylabel('True Positive Rate', fontsize=14)
plt.legend(loc='lower right', fontsize=12)
plt.grid(True)
plt.show()
```

It is visible that the random forest classification has the highest rank compared to neural network model.

8 References

- Scikit-learn Developers (n.d.), *MLPClassifier — Scikit-learn Documentation*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.neural$_n$*etwork.MLPClassifier.html* (Accessed: 1 January 2025).
- Scikit-learn Developers (n.d.), *MinMaxScaler — Scikit-learn Documentation*. Available at: https://scikit-learn.org/1.5/modules/generated/sklearn.preprocessing.MinMaxScaler.html (Accessed: 1 January 2025).
- Scikit-learn Developers (n.d.), *RandomForestClassifier — Scikit-learn Documentation*. Available at: https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.RandomForestClassifier.html (Accessed: 1 January 2025).
- KDnuggets (n.d.), *Hyperparameter Tuning with GridSearchCV and RandomizedSearchCV Explained*. Available at: https://www.kdnuggets.com/hyperparameter-tuning-gridsearchcv-and-randomizedsearchcv-explained (Accessed: 1 January 2025).
- GeeksforGeeks (November 2024), *ML — One Hot Encoding*. Available at: https://www.geeksforgeeks.org/ml-one-hot-encoding/ (Accessed: 1 January 2025).