General approach to all binary classification problems

Applied to an Example Dataset From Cleaning to Model Evaluation

Saman Teymouri

September 2024

CONTENTS

ABSTRACT

This paper addresses a full study of a binary classification problem. The problem is about the prediction for customer conversion for term life insurance in the HashSysTech Insurance company. The study starts with data description and cleaning, continues with model selection and end with model interpretation and evaluation. It uses four classification model: decision tree, random forest, logistic regression, and k-nearest neighbors. The results show that the random forest algorithm performs better than the others and the “dur” feature has the most impact on the model behavior. In this study, all codes are written in a way to be usable for any binary classification dataset in the future, and they avoid redundancy as much as possible. Every block of code has comments to describe what it does.

INTRODUCTION

Binary classification problems need different phases to solve. These phases include data exploration and cleaning, model selection and training, and finally model interpretation and evaluation.

The first phase is divided into three parts. The first part is about providing descriptive information about the dataset. The next step is cleaning the dataset, then data visualizations make the relationships between variables of the dataset more understandable. Descriptive analysis gives information about columns of the dataset including names, data types, etc. It also specifies measures of central tendency such as mean and median (Australian Bureau of Statistics, 2023), and dispersion such as standard deviation and interquartile ranges (Ruhil, 2021) for numeric variables. For categorical variables, the measures are only limited to mode. Cleaning data includes different steps such as handling missing values, removing duplicate values, dealing with outliers, standardizing and normalizing, encoding categorical variables, etc (Mellowacademy, 2023). Last but not least part is visualization which is done using Matplotlib (Hunter, 2007) and Seaborn (Seaborn, 2024) utilities. It shows different types of plots such as scatter plots, histograms, box plots, and bar plots to clarify the trends and patterns of data.

The second phase describes the process of model selection and training. Choosing a suitable model is vital for every machine-learning problem. For binary classification problems, there are many choices such as decision tree, random forest, logistic regression, k-nearest neighbors, support vector machine, etc (Johnston and Mathur, 2019). This phase includes selecting models, splitting the dataset to train and test sets, fitting the models with the train set, and getting back the prediction results of the test set.

While it is essential to focus on model performance, it's also important to understand how the features in the model contribute to the prediction. We need to be able to explain the model and how different variables affect the prediction to the relevant stakeholders who might demand insight into why the model is successful (Johnston and Mathur, 2019). The model interpretation is the first step in the third phase. The next step is model evaluation which gives the comparison between models based on their evaluation metrics such as accuracy, precision, recall, f1-score, etc.

TASK 1: DATA EXPLORATION AND PREPARATION

1.1 DATA EXPLORATION

The program written for this study opens the dataset and extracts the first five rows via the head method. load_data function is responsible for this (Figure 1).

```python
def load_data(file_path : str, logger : Logger = None) -> pd.DataFrame:
    # opening the file and show 5 first rows
    logger.log("################### First 5 rows of original data ####################")
    # open csv file and load it into a dataframe
    data = pd.read_csv(file_path)
    # return the first 5 rows of the dataset
    logger.log(data.head())
    return data
```

Figure 1: Loading the dataset

This program is written completely functional and all the functions are called from the main function (Figure 2).

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
import graphviz
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report
from general import Logger
import io
from sklearn.preprocessing import minmax_scale, normalize, LabelEncoder
import warnings
from math import ceil
import time

# Target variable name is assigned here and used in the code globally
target_col = "y"

def main() -> int:
    # Folder path to save outputs
    output_folder = "output"
    logger = Logger()

    current_time = time.time()

    # Load dataset
    # SAMAN TEYMOURI FDA WRITTEN EXERCISE
    df = load_data("input/dataset.csv", logger=logger)

    # describe dataset characteristics
    describe_data(df, logger=logger)

    # Show data exploration
    plot_pair(df, "before_data_cleaning", output_folder=output_folder,logger=logger)

    # Plot before data cleaning
    plot_hist(df, "before_data_cleaning", output_folder, logger=logger)

    elapsedtime = int(time.time() - current_time)
    current_time = time.time()
```

Figure 2: part of the main function and used libraries

There are the first five rows of the dataset below (Figure 3).

```
################### First 5 rows of original data ####################
   age           job  marital education_qual call_type  day  mon  dur  num_calls prev_outcome   y
0   58    management  married       tertiary   unknown    5  may  261          1      unknown  no
1   44    technician   single      secondary   unknown    5  may  151          1      unknown  no
2   33  entrepreneur  married      secondary   unknown    5  may   76          1      unknown  no
3   47   blue-collar  married        unknown   unknown    5  may   92          1      unknown  no
4   33       unknown   single        unknown   unknown    5  may  198          1      unknown  no
```

Figure 3: The first five rows of the dataset before data cleaning

The info method of dataframe shows that the dataset has eleven columns (four numeric and seven categorical) (Figure 4). Since the assignment gives the definition of columns, this paper avoids repeating them.

```
################### Describe original data specifications ##############
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 11 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_qual  45211 non-null  object
 4   call_type       45211 non-null  object
 5   day             45211 non-null  int64
 6   mon             45211 non-null  object
 7   dur             45211 non-null  int64
 8   num_calls       45211 non-null  int64
 9   prev_outcome    45211 non-null  object
 10  y               45211 non-null  object
dtypes: int64(4), object(7)
memory usage: 3.8+ MB
```

Figure 4: The info method output

Describe method extracts measures of central tendency and dispersion for numeric variables (Figure 5).

```
def describe_data(data : pd.DataFrame, logger : Logger = None):
    # extract some descriptive analysis
    logger.log("\n################### Describe original data specifications ##############")

    # show the rows and columns count plus columns data types
    info = io.StringIO()
    data.info(buf=info)
    logger.log(info.getvalue())

    # show statistic of the columns one by one
    describe = data.describe()

    # since median and mode are not included in describe method outputs, they are added manually to the output for numeric columns
    median = data.median(axis="index", numeric_only=True)
    numeric_mode = data.mode(numeric_only=True)
    for col in describe:
        describe.at["median",col] = median[col]
        describe.at["mode",col] = numeric_mode.loc[0,col]

    logger.log(describe)

    # for categorical columns only mode measure is available
    categorical_mode = data.mode().drop(columns=describe.columns)
    logger.log(f"\nModes of categorical variables are:\n{categorical_mode}")
    for col in categorical_mode:
        logger.log(f"\n{data.value_counts([col])}")

    logger.log("#########################################################################\n")
```

Figure 5: The describe_data function

The method calculates the count, mean, standard deviation, and Interquartile ranges of each column. But the program also calculates the median and mode of the columns and adds them to the end of the output (Figure 6).

	age	day	dur	num_calls
count	45211.000000	45211.000000	45211.000000	45211.000000
mean	40.936210	15.806419	258.163080	2.763841
std	10.618762	8.322476	257.527812	3.098021
min	18.000000	1.000000	0.000000	1.000000
25%	33.000000	8.000000	103.000000	1.000000
50%	39.000000	16.000000	180.000000	2.000000
75%	48.000000	21.000000	319.000000	3.000000
max	95.000000	31.000000	4918.000000	63.000000
median	39.000000	16.000000	180.000000	2.000000
mode	32.000000	20.000000	124.000000	1.000000

Figure 6: Measures of central tendency and dispersion

It identifies not only the mode for the categorical variables (Figure 7) but also more elaborate information about the frequency of use of them (Figure 8).

```
Modes of categorical variables are:
          job  marital education_qual call_type  mon prev_outcome   y
0  blue-collar  married     secondary  cellular  may      unknown  no
################################################################################
```

Figure 7: The mode of the categorical variables

```
job
blue-collar      9732
management       9458
technician       7597
admin.           5171
services         4154
retired          2264
self-employed    1579
entrepreneur     1487
unemployed       1303
housemaid        1240
student           938
unknown           288
Name: count, dtype: int64

marital
married     27214
single      12790
divorced     5207
Name: count, dtype: int64

education_qual
secondary         23202
tertiary          13301
primary            6851
unknown            1857
Name: count, dtype: int64

call_type
cellular     29285
unknown      13020
telephone     2906
Name: count, dtype: int64

Name: count, dtype: int64

mon
may    13766
jul     6895
aug     6247
jun     5341
nov     3970
apr     2932
feb     2649
jan     1403
oct      738
sep      579
mar      477
dec      214
Name: count, dtype: int64

prev_outcome
unknown         36959
failure          4901
other            1840
success          1511
Name: count, dtype: int64

y
no     39922
yes     5289
Name: count, dtype: int64
```

Figure 8: The frequency of use of categorical variables

This study, based on the above information, shows that clients have an average age of 40 with a norm 2-3 calls that last 258 seconds on average. It is interesting to know that most of the time calls are taken in the middle of the month. Based on categorical variables, we discovered that the most of contacts are with hard manual labor workers. Most of the clients are married with unknown previous status. They are not very well-educated and are being contacted via their mobile phones. It is good to know that 30 percent of calls are taken in May!

1.2 HANDLING MISSING VALUES AND OUTLIERS

At first, the program examines the dataset to find missing values (Figure 9). The results show that the dataset has no missing values (Figure 10) so it does not need to do any further operations to address them.

```python
def handle_missing_values(data : pd.DataFrame, logger : Logger = None) -> pd.DataFrame:
    # examining data to discover missing values
    logger.log(f"Dataset has these missing values:\n{data.isna().sum()}\n")
    # Replace missing values via bfill method
    data = data.bfill()
    logger.log(f"Missing values are replaced by next valid value of that column using bfill method.\n")
    return data
```

Figure 9: Handling missing values

If there were any missing values, they could be handled by a method such as bfill.

```
################### Data cleaning and dataset changes ###################
Dataset has these missing values:
age               0
job               0
marital           0
education_qual    0
call_type         0
day               0
mon               0
dur               0
num_calls         0
prev_outcome      0
y                 0
dtype: int64

Missing values are replaced by next valid value of that column using bfill method.
```

Figure 10: Missing values of the dataset

After managing missing values, this study investigates the dataset to find duplicate rows (Figure 11).

```python
def handle_duplicate_values(data : pd.DataFrame, logger : Logger = None) -> pd.DataFrame:
    # exploriing data to find duplicated rows
    logger.log(f"Duplicate rows are:\n{data.loc[data.duplicated()]}")
    logger.log(f"Duplicate rows count: {data.duplicated().sum()}")
    # eliminate duplicate rows from data
    data = data.drop_duplicates()
    # summary about data after removing duplicates
    logger.log(f"After removing duplicate rows the dataset has {data.shape[0]} rows.\n")
    return data
```

Figure 11: Handling duplicate values

The results show that there are 6 duplicate rows in the dataset (Figure 12). they get removed, and after that, the rows of the dataset are reduced to 45205.

```
Duplicate rows are:
       age          job  marital education_qual call_type  day  mon  dur  num_calls prev_outcome   y
6893    34     services  married      secondary   unknown   28  may  124          1      unknown  no
8138    29       admin.   single      secondary   unknown    2  jun  121          4      unknown  no
11630   39  blue-collar  married        primary   unknown   19  jun  112          4      unknown  no
13400   36  blue-collar  married      secondary  cellular    9  jul  183          1      unknown  no
19826   36   management  married       tertiary  cellular    8  aug   75          2      unknown  no
19854   32   technician   single       tertiary  cellular    8  aug   31          2      unknown  no
Duplicate rows count: 6
After removing duplicate rows the dataset has 45205 rows.
```

Figure 12: Duplicate values of the dataset

At this step, the program goes for outliers detection. There are different ways to find the presence of outliers in the dataset (Magaga, 2021). This paper uses interquartile range and skewness to identify outliers (Figure 13). Visualizations by box plots are also used which is covered in section 1.3. the skewness value should be within the range of -1 to 1 for a normal distribution, and much higher or lower values increase the chance of outliers presence (Magaga, 2021). IQR

method indicates that the “age”, “dur”, and “num_calls” columns contain outliers, and skewness admits this fact about the “dur”, “num_calls” columns.

At first, this study decides to remove outliers from the dataset (Figure 14). After removing them, it is discovered that it loses about 15% of the data so it decides to replace them with their column’s mode value.

```python
def handle_outlier_values(data : pd.DataFrame, output_folder : str = "output", logger : Logger = None) -> pd.DataFrame:
    # Distinguish and remove the outliers based on IQR analysis
    describe = data.describe()
    try:
        describe = describe.drop(columns=target_col)
    except:
        pass
    # Box plot to show outliers
    plot_box(data=data, output_folder=output_folder, logger=logger)

    # Detecting outliers with IQR method
    logger.log("Outliers based on IQR method and skewness before handling them:\n")

    outliers_index = {}
    for col in describe.columns:
        # acquiring q1 and q3 to establish allowed area
        q1 = describe.loc["25%", col]
        q3 = describe.loc["75%", col]
        iqr = q3 - q1
        # every value outside of allowed area is distiguished as an outlier
        outliers = data[col].loc[(data[col] > q3 + 1.5*iqr) | (data[col] < q1 - 1.5*iqr)]
        logger.log(f"Outliers of Column {col} count: {outliers.count()}  ---  skewness: {data[col].skew()}")
        # outlier index of each column
        if not outliers.empty:
            outliers_index[col] = outliers.index

    # data = data.drop(outliers_index)
    # # summary about data after removing outliers
    # logger.log(f"After removing outliers rows the dataset has {data.shape[0]} rows.\n")

    # Replace outliers with mode of that columns
    for col in outliers_index.keys():
        data.loc[outliers_index[col],col] = data.mode(numeric_only=True).loc[0,col]

    # Detecting outliers with IQR method after handling them
    logger.log("\nOutliers based on IQR method and skewness after handling them:\n")

    outliers_index = {}
    for col in describe.columns:
        # acquiring q1 and q3 to establish allowed area
        q1 = describe.loc["25%", col]
        q3 = describe.loc["75%", col]
        iqr = q3 - q1
        # every value outside of allowed area is distiguished as an outlier
        outliers = data[col].loc[(data[col] > q3 + 1.5*iqr) | (data[col] < q1 - 1.5*iqr)]
        logger.log(f"Outliers of Column {col} count: {outliers.count()}  ---  skewness: {data[col].skew()}")
        # outlier index of each column
        if not outliers.empty:
            outliers_index[col] = outliers.index

    return data
```

Figure 13: Handling outliers

```
##################### input_outliers.png file saved #####################

Outliers of Column age count: 487  ---  skewness: 0.6846453089841554
Outliers of Column day count: 0  ---  skewness: 0.09300853841061418
Outliers of Column dur count: 3235  ---  skewness: 3.1441453453578547
Outliers of Column num_calls count: 3064  ---  skewness: 4.898452689018327
After removing outliers rows the dataset has 38661 rows.
```

Figure 14: Outlier values of the dataset

After replacing outliers, the results show that skewness gets back to its acceptable range (Figure 15).

```
Outliers based on IQR method and skewness after handling them:

Outliers of Column age count: 0  ---  skewness: 0.4353528088779051
Outliers of Column day count: 0  ---  skewness: 0.09300853841061418
Outliers of Column dur count: 0  ---  skewness: 1.1294656133921699
Outliers of Column num_calls count: 0  ---  skewness: 1.3141750431998713
```

Figure 15: Skewness after handling outliers

After passing steps, the program encodes categorical variables (Figure 16). For this, it uses the LabelEncoder class. It assigns numbers starting with zero to categorical variables' different classes.

```python
def encode_data(data : pd.DataFrame, logger : Logger) -> pd.DataFrame:
    # Encode categorical variables to numeric
    # Replace encoded variable using Label Encoding

    logger.log(f"\nBelow columns need encoding:")

    le = LabelEncoder()
    # for each columns in the dataset
    for col in data.columns:
        # if the column is categorical
        if data[col].dtype == "object":
            # encode the data into numeric
            data[col] = le.fit_transform(data[col])
            #unique values of encoded column
            logger.log(f"Values for {col} are {le.classes_}")

    return data
```

Figure 16: Encoding categorical variables

The encoded variables and their different classes are shown below (Figure 17).

```
Below columns need encoding:
Values for job are ['admin.' 'blue-collar' 'entrepreneur' 'housemaid' 'management' 'retired'
 'self-employed' 'services' 'student' 'technician' 'unemployed' 'unknown']
Values for marital are ['divorced' 'married' 'single']
Values for education_qual are ['primary' 'secondary' 'tertiary' 'unknown']
Values for call_type are ['cellular' 'telephone' 'unknown']
Values for mon are ['apr' 'aug' 'dec' 'feb' 'jan' 'jul' 'jun' 'mar' 'may' 'nov' 'oct' 'sep']
Values for prev_outcome are ['failure' 'other' 'success' 'unknown']
Values for y are ['no' 'yes']
```

Figure 17: Categorical variables with their different classes

The last step is scaling. This paper decides to scale all features with minmax_scale to have a uniform view of features (Figure 18). It helps to understand features better and does not change data shape.

```python
def scale_data(data : pd.DataFrame, logger : Logger = None) -> pd.DataFrame:
    # Scale features values

    # Scale all columns except target column
    scaling_cols = list(data.drop(target_col,axis="columns").columns)

    # Extract columns which need Scaling
    need_scaling = data[scaling_cols]

    # Scale data
    ndarr = minmax_scale(need_scaling, axis=0) #axis=0 means column-wise

    scaled_data = pd.DataFrame()
    # Build dataframe based on the ndarray and original data
    scaled_data.index = data.index
    for col in data.columns:
        # all features values come from ndarr and the target values come from original data
        if col in scaling_cols:
            scaled_data[col] = ndarr[:,scaling_cols.index(col)]
        else:
            scaled_data[col] = data.loc[:,col]

    # summary about data after encoding and scaling
    logger.log(f"\nAfter encoding and Scaling dataset is:")
    logger.log(scaled_data)
    logger.log(f"'No' Count: {scaled_data[target_col].count() - scaled_data[target_col].sum()} --- 'Yes' Count: {scaled_data[target_col].sum()}\n")

    return scaled_data
```

Figure 18: Scaling the data

There is the dataset structure after data cleaning below (Figure 19).

```
After encoding and Scaling dataset is:
            age       job  marital  education_qual  call_type       day       mon       dur  num_calls  prev_outcome  y
0      0.769231  0.363636      0.5        0.666667        1.0  0.133333  0.727273  0.405910        0.0      1.000000  0
1      0.500000  0.818182      1.0        0.333333        1.0  0.133333  0.727273  0.234837        0.0      1.000000  0
2      0.288462  0.181818      0.5        0.333333        1.0  0.133333  0.727273  0.118196        0.0      1.000000  0
3      0.557692  0.090909      0.5        1.000000        1.0  0.133333  0.727273  0.143079        0.0      1.000000  0
4      0.288462  1.000000      1.0        1.000000        1.0  0.133333  0.727273  0.307932        0.0      1.000000  0
...         ...       ...      ...             ...        ...       ...       ...       ...        ...           ... ..
45206  0.634615  0.818182      0.5        0.666667        0.0  0.533333  0.818182  0.192846        0.4      1.000000  1
45207  0.269231  0.454545      0.0        0.000000        0.0  0.533333  0.818182  0.709176        0.2      1.000000  1
45208  0.269231  0.454545      0.5        0.333333        0.0  0.533333  0.818182  0.192846        0.8      0.666667  1
45209  0.750000  0.090909      0.5        0.333333        0.5  0.533333  0.818182  0.790047        0.6      1.000000  0
45210  0.365385  0.181818      0.5        0.333333        0.0  0.533333  0.818182  0.561431        0.2      0.333333  0

[45205 rows x 11 columns]
'No' Count: 39916 --- 'Yes' Count: 5289
```

Figure 19: The dataset values after data cleaning finished

The result of describe function after cleaning (except the scaling step) shows no categorical variable in the dataset (Figure 20).

	age	job	marital	education_qual	call_type	day	mon	dur	num_calls	prev_outcome	y
count	45205.000000	45205.000000	45205.000000	45205.000000	45205.000000	45205.00000	45205.000000	45205.000000	45205.000000	45205.000000	45205.000000
mean	40.454286	4.339852	1.167703	1.224820	0.640195	15.80688	5.523150	197.815065	2.053335	2.559916	0.117000
std	9.963815	3.272637	0.608243	0.748005	0.897927	8.32234	3.006935	137.216384	1.301834	0.989112	0.321424
min	18.000000	0.000000	0.000000	0.000000	0.000000	1.00000	0.000000	0.000000	1.000000	0.000000	0.000000
25%	32.000000	1.000000	1.000000	1.000000	0.000000	8.00000	3.000000	103.000000	1.000000	3.000000	0.000000
50%	39.000000	4.000000	1.000000	1.000000	0.000000	16.00000	6.000000	156.000000	2.000000	3.000000	0.000000
75%	48.000000	7.000000	2.000000	2.000000	2.000000	21.00000	8.000000	265.000000	3.000000	3.000000	0.000000
max	70.000000	11.000000	2.000000	3.000000	2.000000	31.00000	11.000000	643.000000	6.000000	3.000000	1.000000
median	39.000000	4.000000	1.000000	1.000000	0.000000	16.00000	6.000000	156.000000	2.000000	3.000000	0.000000
mode	32.000000	1.000000	1.000000	1.000000	0.000000	20.00000	8.000000	124.000000	1.000000	3.000000	0.000000

Figure 20: The dataset description after cleaning (before scaling)

After encoding categorical variables, this study calculates correlation between columns of dataset (Figure 21)

```python
def plot_correlation(data : pd.DataFrame, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    # Extract correlatoin between features themselves and also with label plus plot their heatmap
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    correlations = data.corr()
    logger.log(f"correlations of features:\n{correlations}")
    sns.heatmap(correlations,annot=True)
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/correlations{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"################### {output_folder}/correlations{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 21: Calculating correlation between the dataset columns

The correlation report (Figure 22) shows that there is no high correlation between columns but to some extent, there is a correlation between “call_type” with both “mon” and “prev_outcome” columns. It also exists between the “marital” and “age” columns. The columns that have the highest correlation with the target variable “y” are “dur” and “call_type”. The opposite signs indicate that they affect different classes of the target variable. There is also a heatmap for correlations that is shown in section 1.3.

```
correlations of features:
                    age       job   marital  education_qual  call_type       day       mon       dur  num_calls  prev_outcome         y
age             1.000000 -0.032499 -0.399883       -0.094783   0.037129 -0.006388 -0.042507 -0.034945   0.033439      0.020859 -0.023055
job            -0.032499  1.000000  0.062042        0.166651  -0.082021  0.022802 -0.092848 -0.003807   0.008903      0.011023  0.040431
marital        -0.399883  0.062042  1.000000        0.108558  -0.039221 -0.005183 -0.006955  0.016199  -0.025963     -0.016868  0.045603
education_qual -0.094783  0.166651  0.108558        1.000000  -0.110841  0.022724 -0.057230 -0.001247  -0.010769     -0.019358  0.066241
call_type       0.037129 -0.082021 -0.039221       -0.110841   1.000000 -0.027990  0.361111 -0.028934  -0.013068      0.272219 -0.148391
day            -0.006388  0.022802 -0.005183        0.022724  -0.027990  1.000000 -0.006116 -0.045750   0.064215      0.083492 -0.028371
mon            -0.042507 -0.092848 -0.006955       -0.057230   0.361111 -0.006116  1.000000  0.009550  -0.091659     -0.033020 -0.024490
dur            -0.034945 -0.003807  0.016199       -0.001247  -0.028934 -0.045750  0.009550  1.000000  -0.042398     -0.014447  0.171948
num_calls       0.033439  0.008903 -0.025963       -0.010769  -0.013068  0.064215 -0.091659 -0.042398   1.000000      0.060474 -0.051128
prev_outcome    0.020859  0.011023 -0.016868       -0.019358   0.272219  0.083492 -0.033020 -0.014447   0.060474      1.000000 -0.077821
y              -0.023055  0.040431  0.045603        0.066241  -0.148391 -0.028371 -0.024490  0.171948  -0.051128     -0.077821  1.000000
```

Figure 22: Correlation between the dataset columns

This cleaned dataset is fully prepared to apply some classification models that performed in task 2.

1.3 DATA VISUALIZATION

This section depicts different types of plots such as scatter plots, histograms, box plots, and bar plots. These plots help us understand relationships between features and the target variable much better.

This paper starts with pair plots that show not only the relationship of every two features but also their relationship with the target variable (Figure 23).

```
def plot_pair(data : pd.DataFrame, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    # show data exploration by plotting relationship between every two columns
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)
    # data Exploration
    sns.pairplot(data)
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/data_exploration{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/data_exploration{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 23: Creating pair plots

Pair plots before data cleaning (Figure 24) and after that (Figure 25) are shown below.

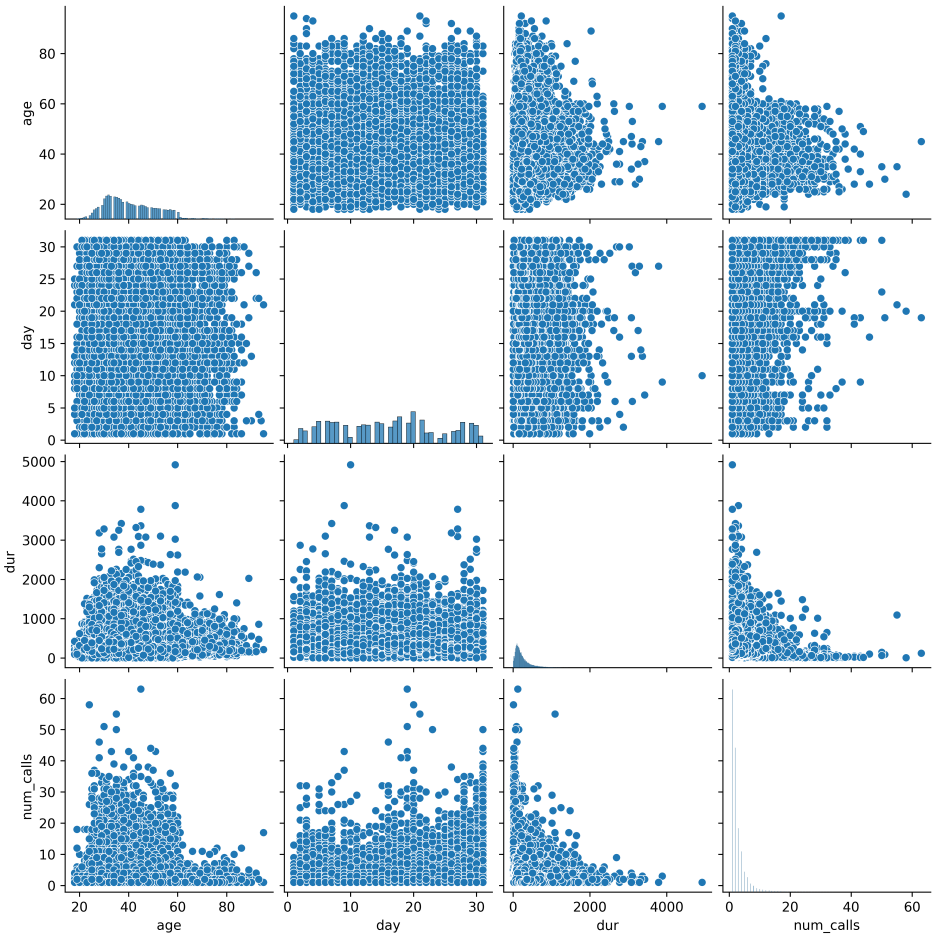


Figure 24: Pair plots before data cleaning

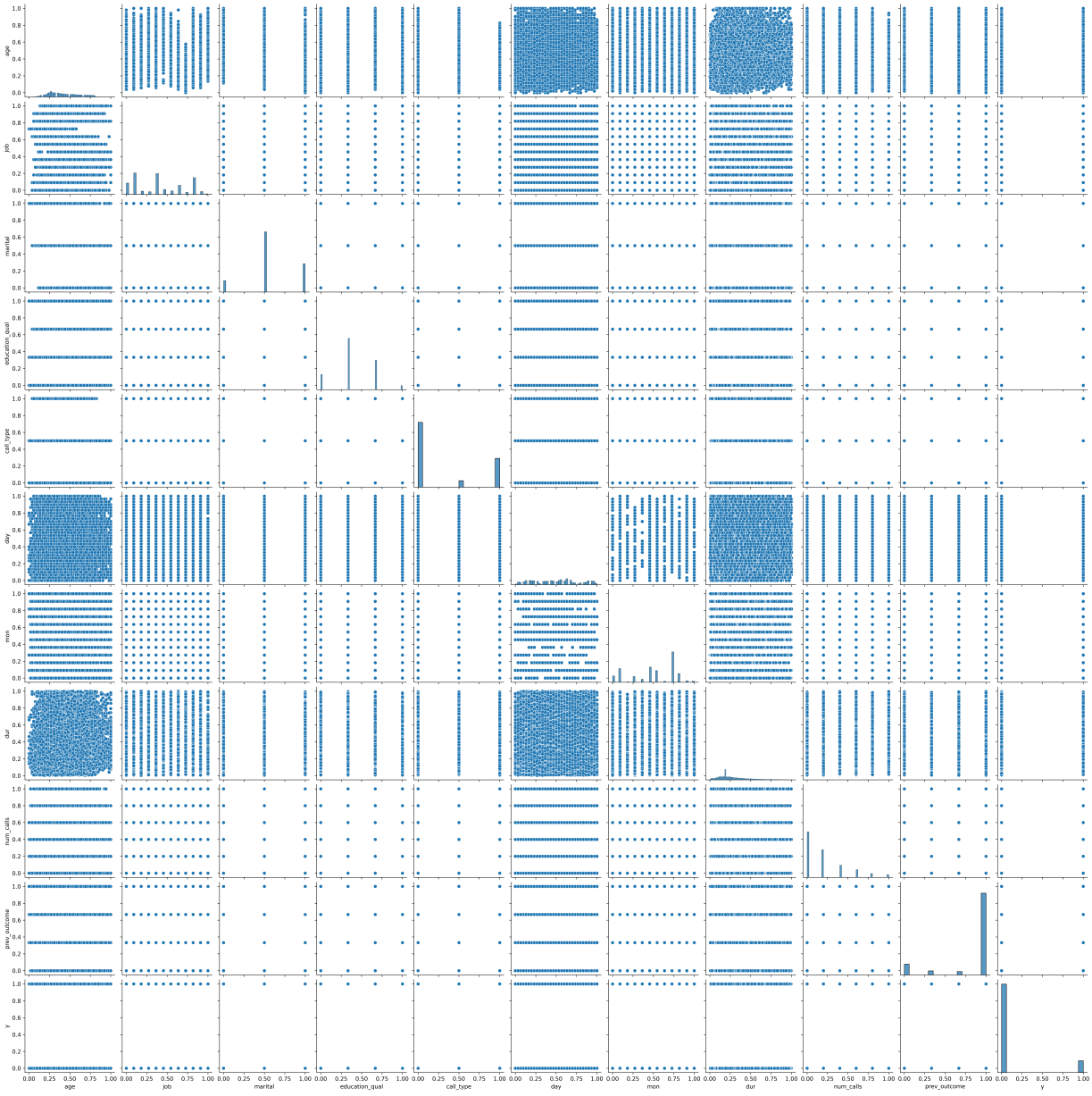


Figure 25: Pair plots after data cleaning

There are bunch of useful information underlying in these plots. For example, when the number of calls increases the call duration decreases. Another example is that clients older than 60 get less than 10 calls with shorter duration. These examples are based on the uncleaned dataset and after cleaning they are cleaned as outliers (Figure 25).

The program also provides the histograms of the dataset before (Figure 26) and after data cleaning (Figure 27).

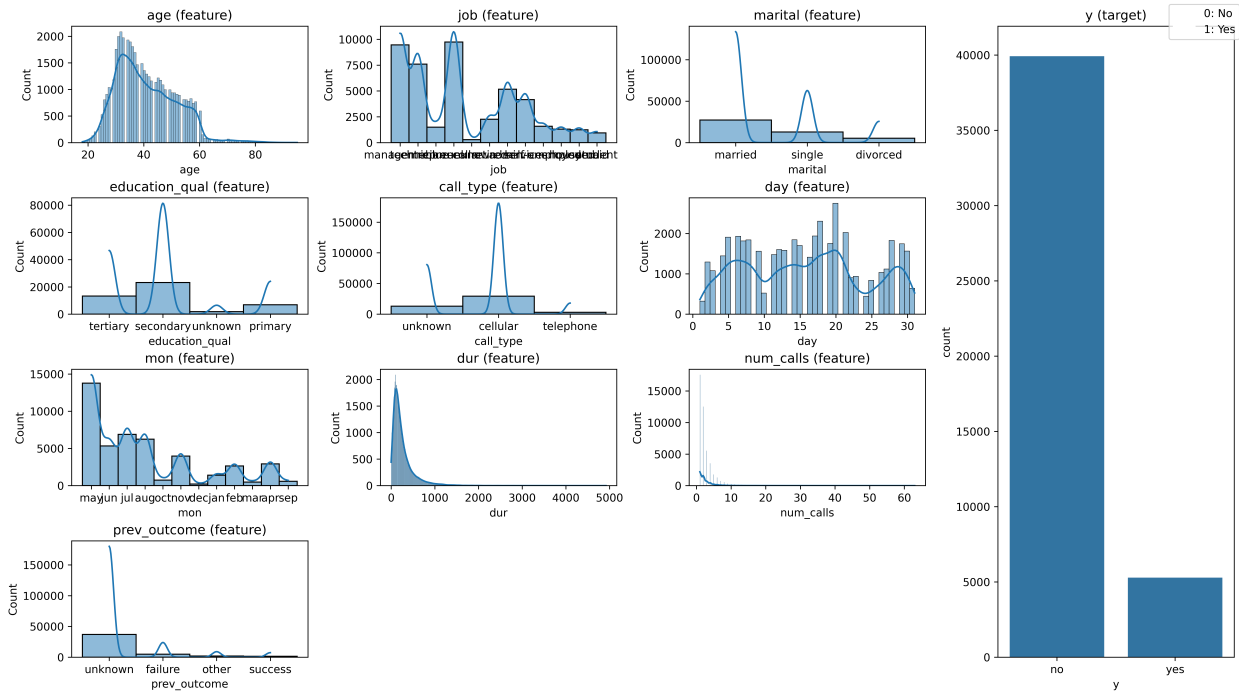


Figure 26: Histograms before data cleaning

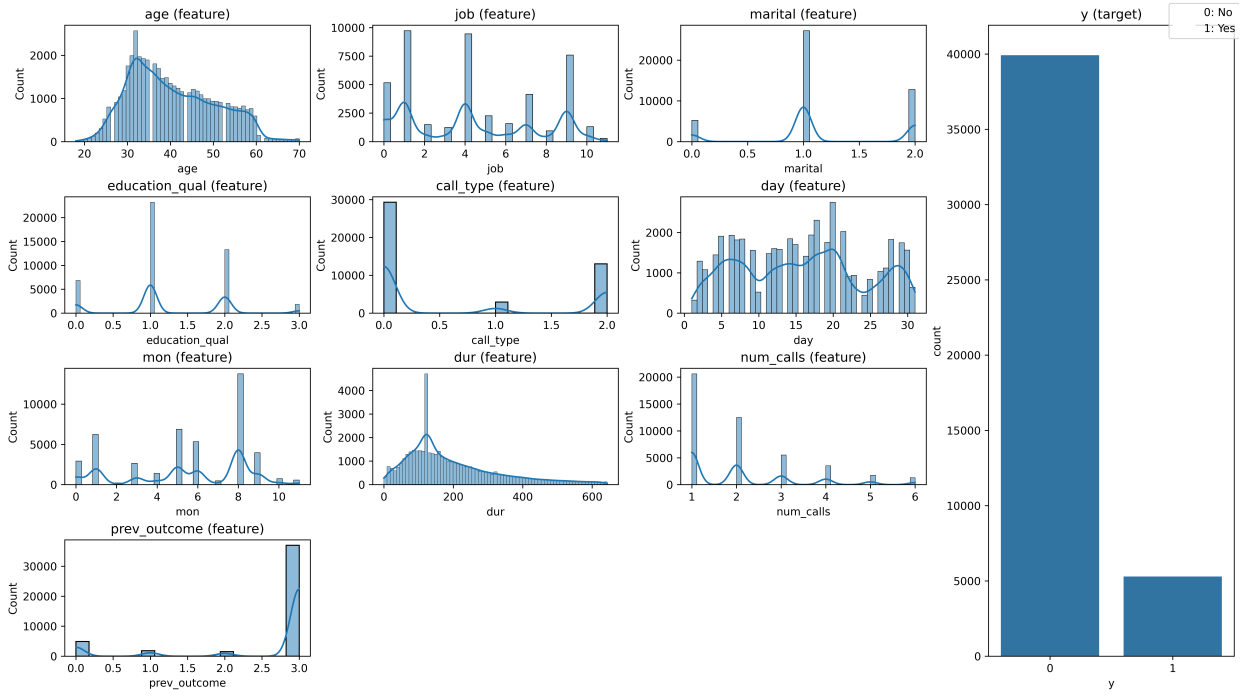


Figure 27: Histograms after data cleaning

Histograms and count plots demonstrate some useful information. For example, it is discovered from pair plots that clients older than 60 get less than 10 calls with shorter duration but now these plots reveal that there are a few clients older than 60. Therefore, the previous fact may not

be very dependable. Another fact that is extracted from previous analysis and is refuted now is that most calls are made in the middle of the month. On the other hand, histograms approve some older understandings such as most calls duration are about 120 minutes, most call numbers are between 1 and 3, unsubscribed clients are significantly more than subscribed ones, or ages of clients have semi-normal distribution, etc.

As this paper mentions in the previous section, there are box plots that show the outliers graphically (Figure 28).

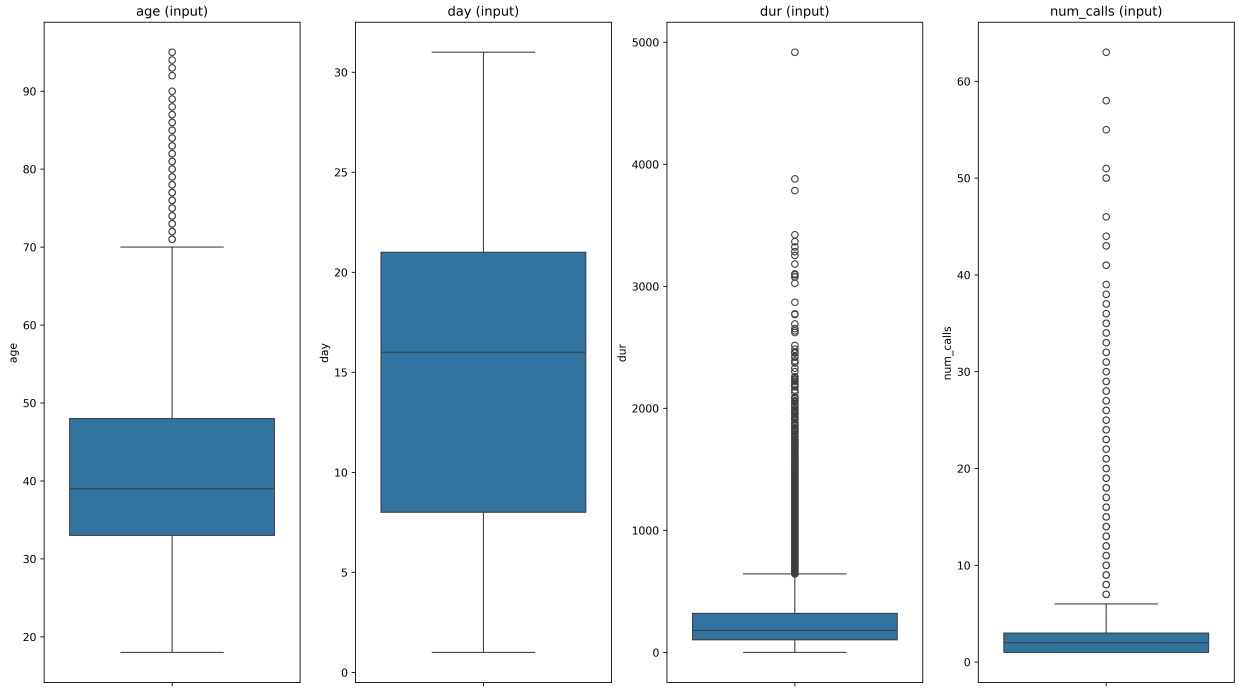


Figure 28: Box plots to investigate for outliers

These plots approve the previous understandings about the presence of outliers. There are lots of them in the “dur” and “num_calls” columns, some of them in the “age” column, and nothing in the “day” column.

Another visualization is the heat map of correlation between columns (Figure 29). out findings about the correlation between columns are visible in this heat map graphically and much easier to understand.

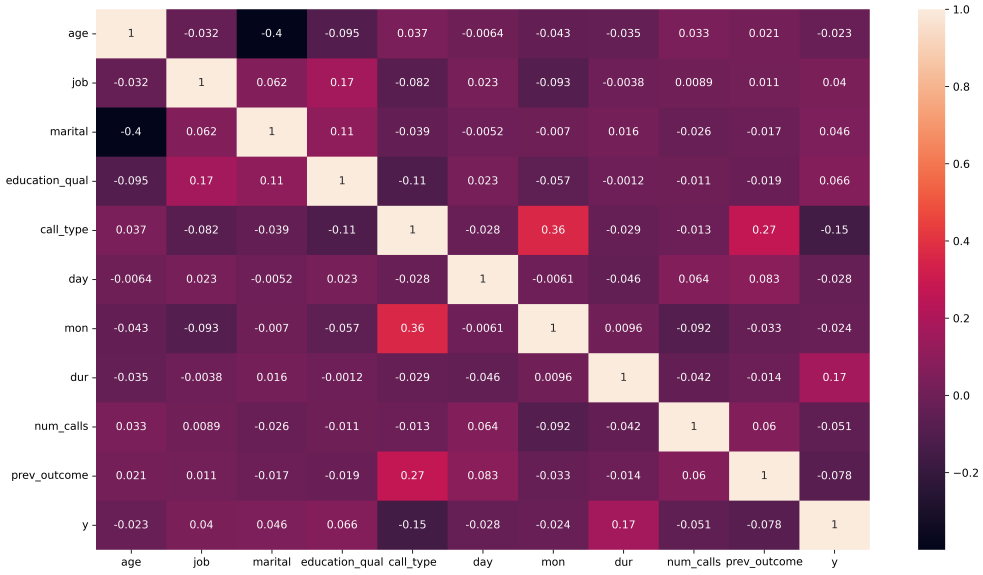


Figure 29: The heat map of correlation between columns

TASK 2: MODEL SELECTION AND TRAINING

2.1 MODEL SELECTION

The problem is a binary classification so this study selects four algorithms including decision tree, random forest, logistic regression, and k-nearest neighbors to solve it.

The first model is decision tree (Figure 30). As the name suggests, decision tree is a learning algorithm that applies a sequential series of decisions based on input information to make the final classification (Johnston and Mathur, 2019). Bonaccorso (2018) states that building a decision tree starts from the root and continues with creating children with minimum entropy until it gets zero.

```
def classification_decisiontree(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = N
    logger.log("################ classification by decision tree ################\n")
    # classification by decision tree

    # grid_best_params is not empty when we are classifying after the optimization
    if grid_best_params:
        clf = DecisionTreeClassifier(max_depth=grid_best_params["max_depth"], min_samples_leaf=grid_best_params["min_samples_leaf"], min_samples_split=grid_best_params["min_samp
    else:
        clf = DecisionTreeClassifier()

    # extract train and test sets from the train_test_sets list
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    clf.fit(X_train, y_train)

    # Predict test_dataset via model
    y_pred = clf.predict(X_test)

    # Exporting decesion tree to png file
    export_tree(clf, suffix, clf.feature_names_in_, output_folder, logger)

    # Evaluate the model
    evaluate_model_decisiontree(clf=clf, data=data, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Before Optimization", output_folder=outpu
```

Figure 30: Classification by decision tree

There are some codes for model evaluation and hypertuning parameters which are covered in task 3. The first step in classification is to split dataset to train and test sets. This is done in another function and the list of train and test sets are passed to the classification_decisiontree function. Then the model is trained. Afterward, it is prepared to predict the test set.

The decision tree classifier uses default parameters at first ("max_depth": None, "min_samples_leaf": 1, "min_samples_split": 2, "random_state": None, "max_features" : None). Task 3 of this paper optimizes these parameters.

The visualization of the created decision tree is provided below (Figure 31). Although regarding a large number of nodes, the picture is not very clear, it is still useful to see the decision tree graphically.

Figure 31: The whole decision tree

To have a better visual, part of the decision tree which contains the root node is provided zoomed in below (Figure 32).

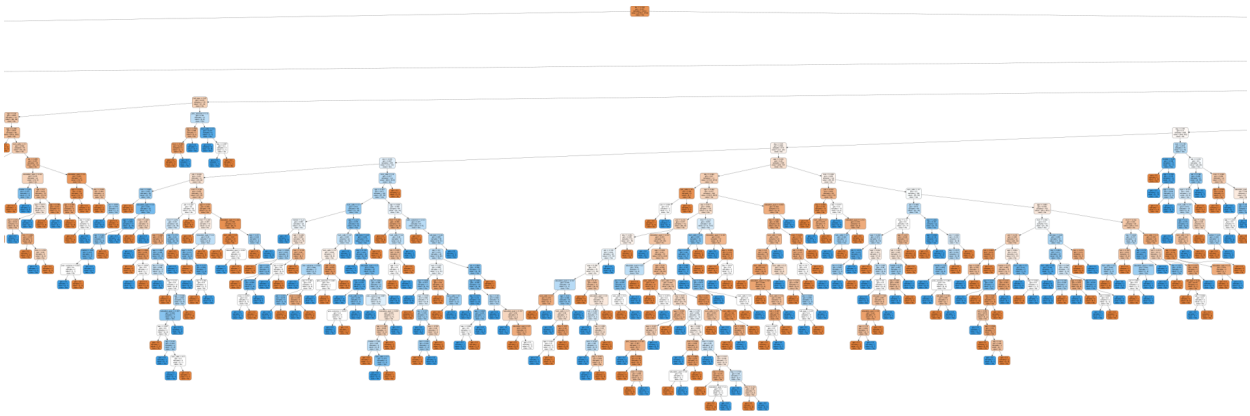


Figure 32: Part of the decision tree that contains root node

The second algorithm used by this study is random forest (Figure 33). Random forest algorithm creates decision trees based on selected features not all of them. This causes the creation of several decision trees with less probability of being correlated (Johnston and Mathur, 2019).

```
def classification_randomforest(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = N
    logger.log("################ classification by random forest ################\n")
    # classification by random forest

    # grid best params is not empty when we are classifying after the optimization
    if grid_best_params:
        clf = RandomForestClassifier(max_depth=grid_best_params["max_depth"], min_samples_leaf=grid_best_params["min_samples_leaf"], min_samples_split=grid_best_params["min_samp
    else:
        clf = RandomForestClassifier()

    # extract train and test sets from the train_test_sets list
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    clf.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = clf.predict(X_test)

    # Exporting decesion trees of the random forest to png file
    est_count = 1
    for estimator in clf.estimators_:
        export_tree(estimator, clf.feature_names_in_, suffix + "_" + str(est_count), output_folder, logger)
        est_count += 1

    # Evaluate the model
    evaluate_model_randomforest(clf=clf, data=data, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Before Optimization", output_folder=outpu
```

Figure 33: Classification by random forest

The steps of the algorithm are the same as the previous one. The default parameters of random forest classifier are the same as decision tree with one difference random forest does not accept “None” for the "max_features" parameter and the default value is “sqrt”.

The random forest algorithm creates 100 decision trees for classification (Figure 34).

Figure 34: Decision trees created by random forest

There are first five decision trees created by the random forest algorithm below (Figure 35).

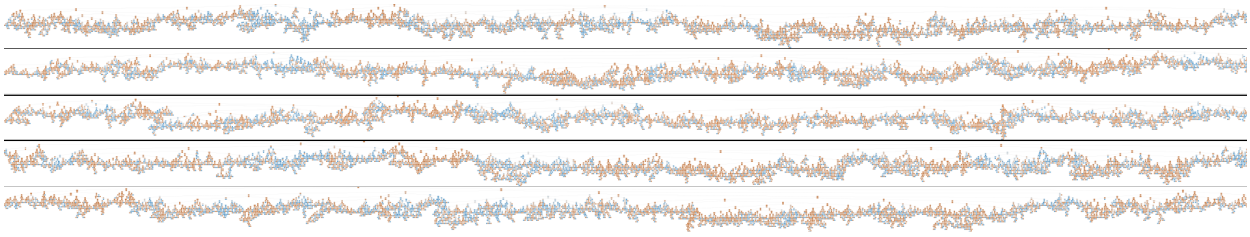


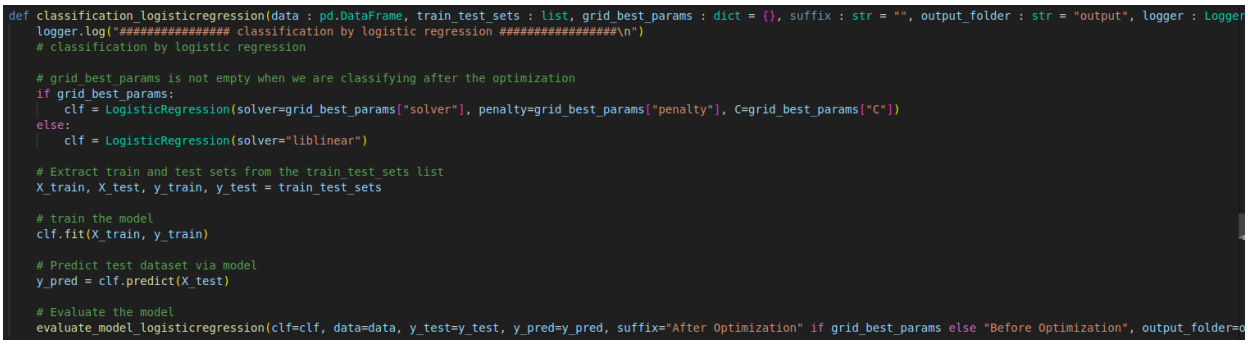
Figure 35: First five decision trees created by random forest algorithm

Exporting decision trees is done by export_tree function using Graphviz (Graphviz, 2021) utilities (Figure 36).

```
def export_tree(clf : DecisionTreeClassifier, feature_names_in_ : np.ndarray, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    # export decision tree into dot file notaion
    dot_data = export_graphviz(clf, out_file =None ,feature_names=feature_names_in_,class_names=["No","Yes"], impurity=True, filled=True, rounded=True)
    # create png file based on dot data
    graph = graphviz.Source(dot_data, format="png")
    # render and save png file
    graph.render(filename=f"{output_folder}/decision_tree{('_' if suffix else '') + suffix}", view=False)
    logger.log(f"#################### {output_folder}/decision_tree{('_' if suffix else '') + suffix}.png file saved ##############################\n")
```

Figure 36: Export decision tree to an image file

The third algorithm that is used in this paper is logistic regression (Figure 37). The logistic or logit model is one of the non-linear models that has been effectively used for classification tasks in different domains (Johnston and Mathur, 2019). Miller and Forte (2017) describe that in logistic regression, the logit function provides a nonlinear transformation on its input and ensures that the range of the output, which is interpreted as the probability of the input belonging to class 1, lies in the interval [0,1].



```
def classification_logisticregression(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger
    logger.log("################ classification by logistic regression #################\n")
    # classification by logistic regression

    # grid_best_params is not empty when we are classifying after the optimization
    if grid_best_params:
        clf = LogisticRegression(solver=grid_best_params["solver"], penalty=grid_best_params["penalty"], C=grid_best_params["C"])
    else:
        clf = LogisticRegression(solver="liblinear")

    # Extract train and test sets from the train_test_sets list
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    clf.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = clf.predict(X_test)

    # Evaluate the model
    evaluate_model_logisticregression(clf=clf, data=data, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Before Optimization", output_folder=o
```

Figure 37: Classification by logistic regression

The model is created by its default parameters at this stage ("solver": "lbfgs", "penalty": "l2", and "C": 1).

The last algorithm is k-nearest neighbors (Figure 38). The k-nearest neighbors (KNN) algorithm is a supervised learning classifier, which uses proximity to make classifications. IBM (2024) explains that the KNN algorithm can work based on different distance metrics such as Euclidean, Manhattan, Minkowski, or Hamming distance and it is a simple algorithm with a few hyperparameters.

```
def classification_knn(data : pd.DataFrame, train_test_sets : list, grid_best_params : dict = {}, suffix : str = "", output_folder : str = "output", logger : Logger = None):
    logger.log("################ classification by k-nearest neighbour #################\n")
    # classification by k-nearest neighbour

    # grid_best_params is not empty when we are classifying after the optimization
    if grid_best_params:
        clf = KNeighborsClassifier(n_neighbors=grid_best_params["n_neighbors"], weights=grid_best_params["weights"], metric=grid_best_params["metric"])
    else:
        clf = KNeighborsClassifier()

    # Extract train and test sets from the train_test_sets list
    X_train, X_test, y_train, y_test = train_test_sets

    # train the model
    clf.fit(X_train, y_train)

    # Predict test dataset via model
    y_pred = clf.predict(X_test)

    # Evaluate the model
    evaluate_model_knn(clf=clf, data=data, y_test=y_test, y_pred=y_pred, suffix="After Optimization" if grid_best_params else "Before Optimization", output_folder=output_folder, l
```

Figure 38: Classification by KNN

2.2 DATA SPLITTING

Splitting data into train and test sets is the first step in all classification algorithms. In this study, the extract_train_test function is responsible for splitting the dataset (Figure 39).

```python
def extract_train_test(data : pd.DataFrame, logger : Logger = None) -> list:
    # Seperate features and labels
    X = data.drop(columns=target_col)
    y = data.loc[:, target_col]

    # split train and test datasets
    train_test_sets = train_test_split(X, y, test_size=0.25, random_state=0)

    # Evaluating the proportion of target variable in train and test sets
    X_train, X_test, y_train, y_test = train_test_sets

    y_value_counts = data.value_counts(target_col)
    y_train_value_counts = pd.DataFrame(y_train).value_counts(target_col)
    y_test_value_counts = pd.DataFrame(y_test).value_counts(target_col)

    # Comparing proportion of target variable classes of train and test sets to the whole dataset
    logger.log(f"\nThe proportion of 'yes' in the whole dataset: {y_value_counts[1]/(y_value_counts[0]+y_value_counts[1])}")
    logger.log(f"The proportion of 'yes' in the y_train set: {y_train_value_counts[1]/(y_train_value_counts[0]+y_train_value_counts[1])}")
    logger.log(f"The proportion of 'yes' in the y_test set: {y_test_value_counts[1]/(y_test_value_counts[0]+y_test_value_counts[1])}")

    return train_test_sets
```

Figure 39: Splitting dataset to train and test sets

Buhl (2023) highlighted that there are various methods of splitting datasets including random sampling, stratified dataset splitting, and cross-validation sampling. It emphasizes that different factors play a role in this issue such as the use case, amount of data, quality of data, and the number of hyperparameters.

```
The proportion of 'yes' in the whole dataset: 0.11700033182170114
The proportion of 'yes' in the y_train set: 0.1162205508240239
The proportion of 'yes' in the y_test set: 0.12011945581241013
```

Figure 40: Comparing the proportion of target variable classes of train and test sets to the whole dataset (test_size = 0.2)

```
The proportion of 'yes' in the whole dataset: 0.11700033182170114
The proportion of 'yes' in the y_train set: 0.11650886352240215
The proportion of 'yes' in the y_test set: 0.11847460626437799
```

Figure 41: Comparing the proportion of target variable classes of train and test sets to the whole dataset (test_size = 0.25)

```
The proportion of 'yes' in the whole dataset: 0.11700033182170114
The proportion of 'yes' in the y_train set: 0.1162026356540151
The proportion of 'yes' in the y_test set: 0.11886152484884235
```

Figure 42: Comparing the proportion of target variable classes of train and test sets to the whole dataset (test_size = 0.3)

This paper uses the random sampling method to split the dataset into train and test sets. In this process the proportion of target variable classes is examined for different test sizes. The results are reported for test_size = 0.2 (Figure 40), test_size = 0.25 (Figure 41), and test_size = 0.3 (Figure 42). They show that the proportions in train and test target sets (y_train, y_test) get closer to the whole dataset proportion from test_size 0.2 to 0.25 but not any significant changes from 0.25 to 0.3. As a result, 0.25 is the optimum value for the test_size.

TASK 3: MODEL INTERPRETATION AND EVALUATION

3.1 MODEL INTERPRETATION

This task involves discovering the impact of features on the model's behavior. The first model is decision tree and the feature importance is evaluated after training the model (Figure 43).

```
# Extract model feature_importances and plot their barh
# Creating a dataframe based on the feature importances
feature_importances = pd.DataFrame(clf.feature_importances_,index=clf.feature_names_in_, columns=["feature_importances"])
feature_importances = feature_importances.sort_values("feature_importances",axis="index")
logger.log(f"Importance of features:\n{feature_importances}")
plt.clf()
# Create a bar plot to compare feature importances visually
plt.barh(feature_importances.index, feature_importances.iloc[:,0])
plt.xlabel("feature_importances")
# Save the file with proper dpi
plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
logger.log(f"################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved #################\n")
```

Figure 43: Calculating the feature importance of decision tree model

The results of feature importance are shown below (Figure 44).

```
Importance of features:
               feature_importances
marital                   0.026059
call_type                 0.030077
education_qual            0.038837
num_calls                 0.055131
job                       0.070610
prev_outcome              0.089678
mon                       0.123910
age                       0.143284
day                       0.147634
dur                       0.274779
```

Figure 44: Feature importance of decision tree model before optimization

The results are sorted, and they show that the "dur" feature has the most importance and "day", "age", and "mon" features have more impact in comparison to others. The bar plot shows them visually to provide a better understanding (Figure 46).

```
Importance of features:
               feature_importances
marital                   0.010908
num_calls                 0.018762
education_qual            0.031819
day                       0.033429
job                       0.036555
prev_outcome              0.067944
call_type                 0.084957
mon                       0.087012
age                       0.094901
dur                       0.533714
```

Figure 45: Feature importance of decision tree model after optimization

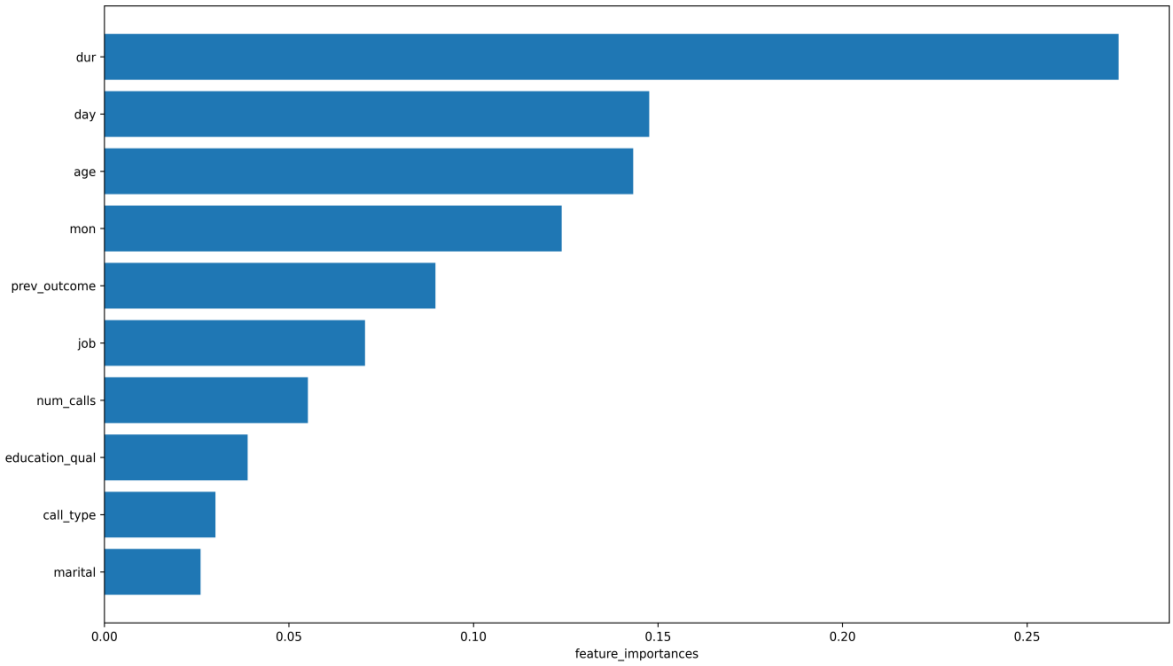


Figure 46: Bar plot of feature importance of decision tree model before optimization

This study performs optimization on models, so the results of feature importance after optimization are also available (Figure 45). The bar plot of these results is also provided (Figure 47). The comparison of results shows that the importance of the "dur" feature increases significantly and that can be a reason for the better accuracy of the model after optimization. The other ranks are also changed and they are "age", "mon", and "call_type" respectively.

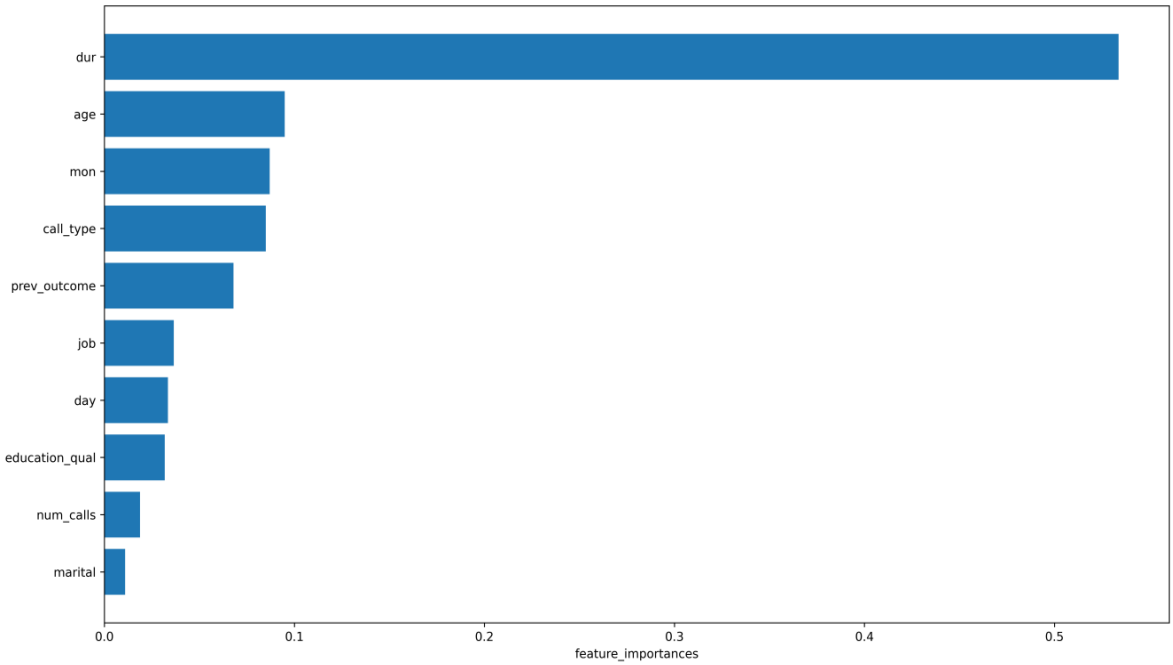


Figure 47: Bar plot of feature importance of decision tree model after optimization

The second algorithm is random forest which is very similar to the decision tree (Figure 48). As expected, the results of feature importance are close to the decision tree results (Figure 49). The "dur" has the first place, and after that "age", "day", and "mon" come respectively.

```
# Extract model feature_importances and plot their barh
# Creating a dataframe based on the feature importances
feature_importances = pd.DataFrame(clf.feature_importances_,index=clf.feature_names_in_, columns=["feature_importances"])
feature_importances = feature_importances.sort_values("feature_importances",axis="index")
logger.log(f"Importance of features:\n{feature_importances}")
plt.clf()
# Create a bar plot to compare feature importances visually
plt.barh(feature_importances.index, feature_importances.iloc[:,0])
plt.xlabel("feature_importances")
# Save the file with proper dpi
plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
logger.log(f"##################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 48: Calculating the feature importance of random forest model

```
Importance of features:
                feature_importances
marital                    0.030640
call_type                  0.031187
education_qual             0.037715
num_calls                  0.051563
job                        0.072780
prev_outcome               0.073072
mon                        0.121091
day                        0.142151
age                        0.153580
dur                        0.286220
```

Figure 49: Feature importance of random forest model before optimization

After optimization, the order of feature importance does not change but the effect of the "dur" feature increases slightly (Figure 51). This can be the reason for the unnoticeable change of accuracy after optimization. The bar plot of the feature importance of the random forest model before (Figure 50) and after (Figure 52) optimization are shown below.

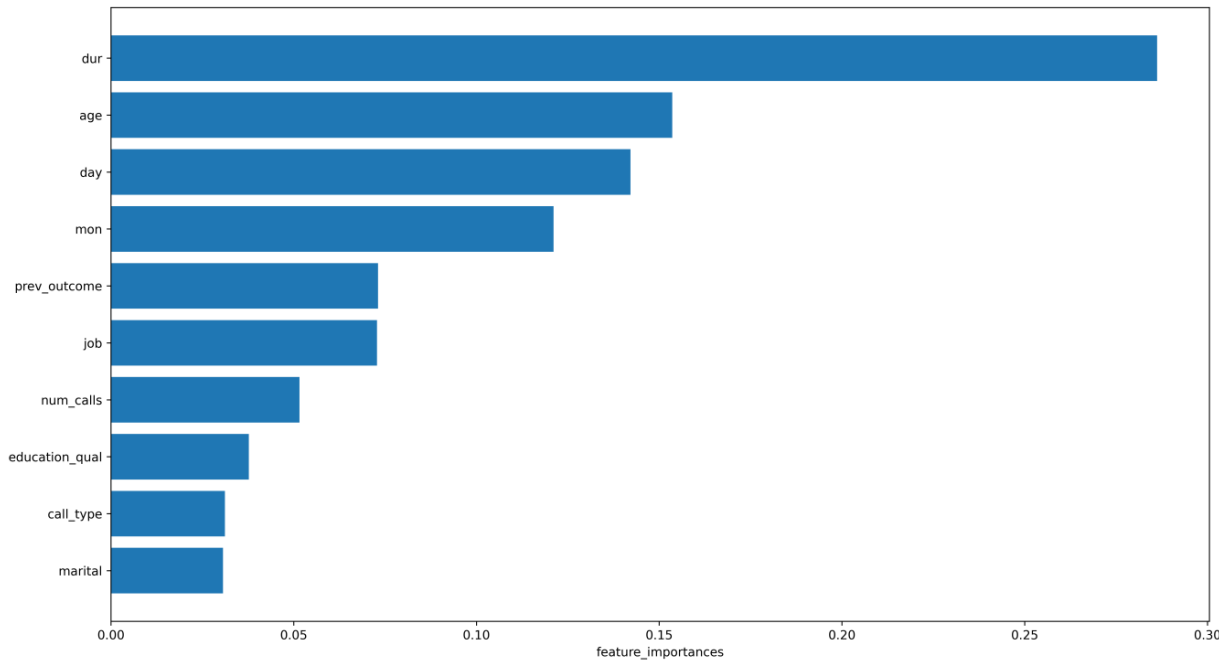


Figure 50: Bar plot of feature importance of random forest model before optimization

```
Importance of features:
               feature_importances
marital                   0.027723
call_type                 0.031352
education_qual            0.035361
num_calls                 0.046561
job                       0.066372
prev_outcome              0.087978
mon                       0.123640
day                       0.137100
age                       0.140764
dur                       0.303149
```

Figure 51: Feature importance of random forest model after optimization

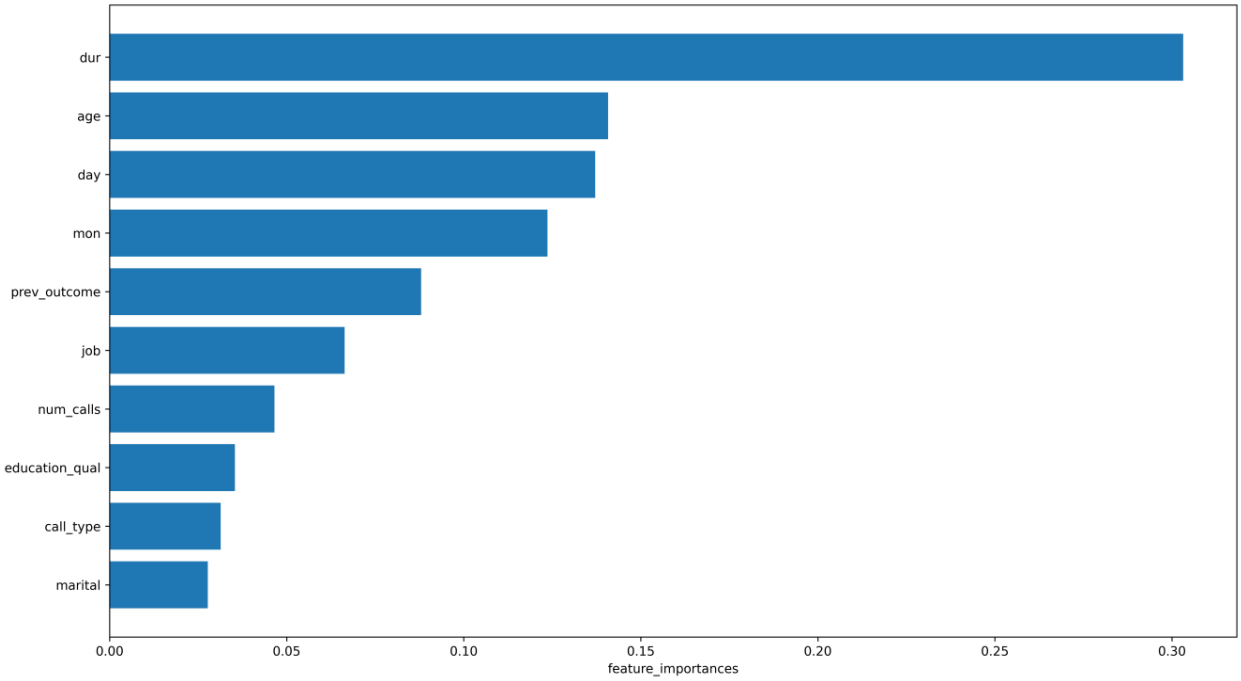


Figure 52: Bar plot of feature importance of random forest model after optimization

The third model in this study is logistic regression which has coefficients to interpret rather than explicit feature importance (Figure 53). In logistic regression, when a coefficient is positive, increasing the feature importance will effectively increase the probability of the output of one class. Similarly, increasing a feature with a negative coefficient shifts the balance toward predicting the other class (Miller and Forte, 2017). Therefore, the absolute value of coefficients shows their importance, and their signs show their efftect on the probability of each target class.

```
# Extract model coefficients and plot their barh
# Creating a dataframe based on the coefficients
coef = pd.DataFrame(clf.coef_[0],index=clf.feature_names_in_, columns=["Coefficient"])
coef = coef.sort_values("Coefficient",axis="index")
logger.log(f"Coefficient of features:\n{coef}")
plt.clf()
# Create a bar plot to compare coefficients visually
plt.barh(coef.index, coef.iloc[:,0])
plt.xlabel("Coefficient")
# Save the file with proper dpi
plt.savefig(fname=f"{output_folder}/Coefficients{("_" if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
logger.log(f"#################### {output_folder}/Coefficient{("_" if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 53: Calculating the coefficients of logistic regression model

The results show that the “dur” has still the most effect. “education_qual” and “marital” features on one class and “call_type” and “num_calls” on the other class have the most impacts (Figure 54). There is also the bar plot of coefficients before optimization below (Figure 55).

```
Coefficient of features:
                Coefficient
call_type         -1.268205
num_calls         -0.559318
day               -0.256647
prev_outcome      -0.220941
age                0.100085
mon                0.208713
job                0.296815
marital            0.341934
education_qual     0.584791
dur                2.131655
```

Figure 54: Coefficients of logistic regression model before optimization

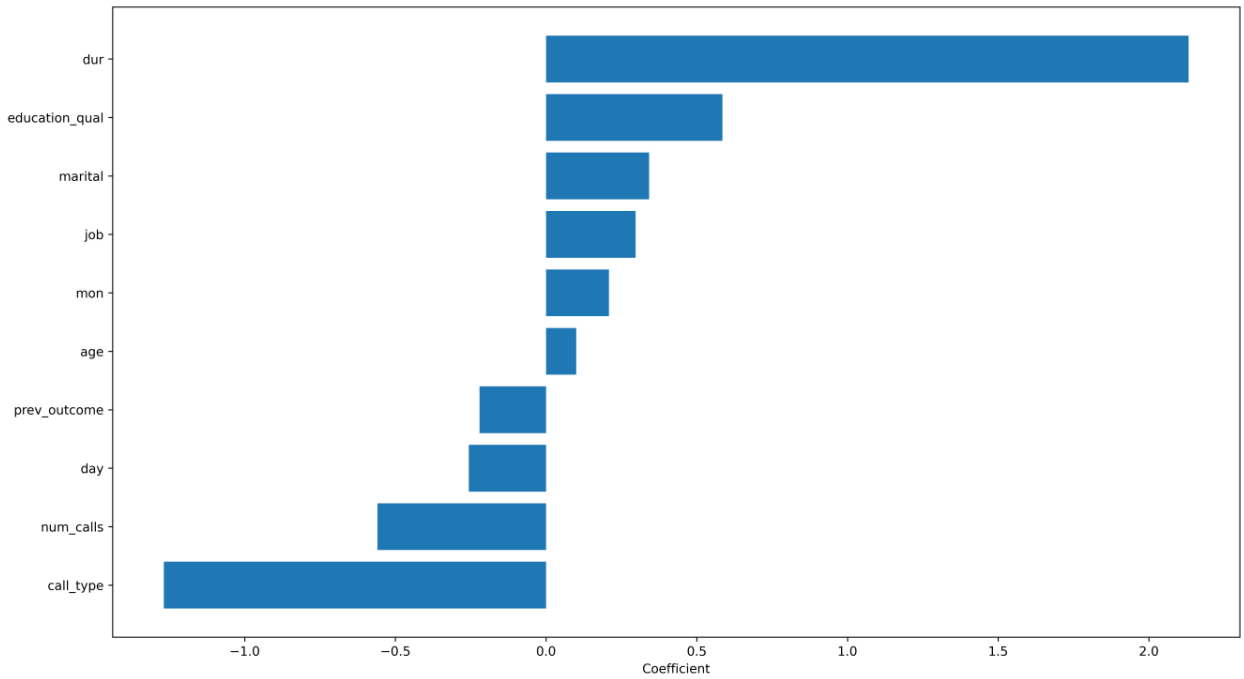


Figure 55: Bar plot of coefficients of logistic regression model before optimization

The coefficients after optimization show no significant changes and consequently no noticeable accuracy increase (Figure 56).

```
Coefficient of features:
               Coefficient
call_type        -1.263528
num_calls        -0.568803
day              -0.257131
prev_outcome     -0.228491
age               0.087071
mon               0.197139
job               0.295275
marital           0.333180
education_qual    0.575133
dur               2.127404
```

Figure 56: Coefficients of logistic regression model after optimization

The bar plot of coefficients of logistic regression after optimization is shown below (Figure 57).

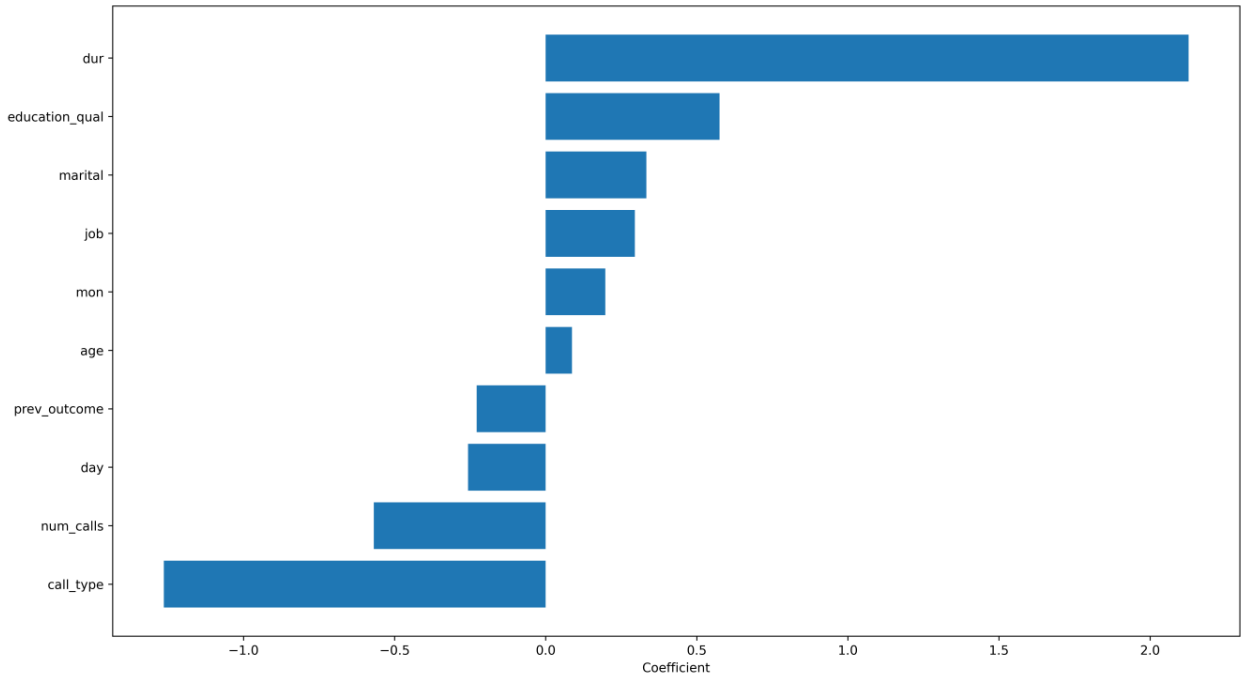


Figure 57: Bar plot of coefficients of logistic regression model after optimization

The last algorithm to interpret is k-nearest neighbors (KNN). Unfortunately, KNN does not provide feature importance or coefficients that indicate the influence of each feature on the prediction. Since KNN considers all features equally in the distance calculation, it does not offer insights into which features contribute more or less to the final decision (Sachinsoni, 2023).

3.2 MODEL EVALUATION

This paper covers the model evaluation and hyperparameter tuning in this section. For each algorithm, the results of evaluation metrics such as accuracy, precision, recall, and F1-score are stated in the first step. After hyperparameter tuning, the updated results are shown for comparison. All these metrics are calculated from the confusion matrix so it is necessary to have it to fully understand all metrics.

```python
def evaluate_model_decisiontree(clf : DecisionTreeClassifier, data : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFrame, suffix : str = "", output_folder : str = "output
    # Evaluating decision tree model
    logger.log(f"################ Evaluating the decision tree model{'(' + suffix + ')' if suffix else ''} ################")
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)

    # Extract confusion matrix and plot its heatmap
    conf_matrix = confusion_matrix(y_test, y_pred)
    logger.log(f"Confusion matrix:\n{conf_matrix}\n")
    sns.heatmap(conf_matrix.reshape(-1,2),annot=True)
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png file saved ##################\n")

    # log the classification report including class-specific precision, recall, f1-score and their macro avg and weighted avg plus model accuracy
    logger.log(f"Classification Report:\n{classification_report(y_test, y_pred,zero_division=0)}")

    # Extract model feature_importances and plot their barh
    # Creating a dataframe based on the feature importances
    feature_importances = pd.DataFrame(clf.feature_importances_,index=clf.feature_names_in_, columns=["feature_importances"])
    feature_importances = feature_importances.sort_values("feature_importances",axis="index")
    logger.log(f"Importance of features:\n{feature_importances}")
    plt.clf()
    # Create a bar plot to compare feature importances visually
    plt.barh(feature_importances.index, feature_importances.iloc[:,0])
    plt.xlabel("feature_importances")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved ##################\n")

    logger.log(f"Classes are: {clf.classes_}")
```

Figure 58: Decision tree evaluation

The same as before, the first algorithm is decision tree (Figure 58). The results (Figure 59) show 87% accuracy which is acceptable. There are 93% and 46% precision for 0 and 1 classes respectively. The values of recall and f1-score are in the same range which indicates the model is more successful in predicting the 0 class rather than the "1" class.

```
################ Evaluating the decision tree model(Before Optimization) #################
Confusion matrix:
[[9233  734]
 [ 704  631]]

#################### output/decisiontree_model/before_optimization/confusion_matrix_Before Optimization.png file saved ###################

Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.93      0.93      9967
           1       0.46      0.47      0.47      1335

    accuracy                           0.87     11302
   macro avg       0.70      0.70      0.70     11302
weighted avg       0.87      0.87      0.87     11302
```

Figure 59: Decision tree evaluation results before optimization

The heat map of the confusion matrix is shown below (Figure 60).

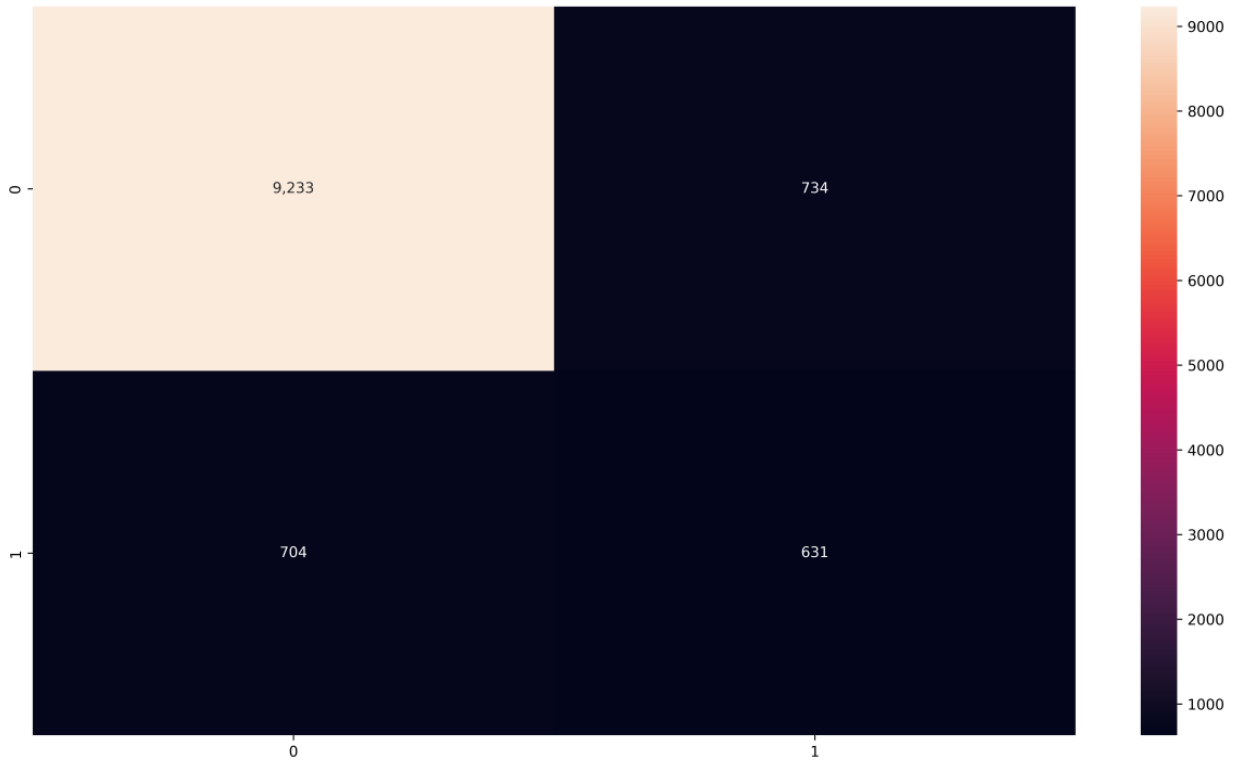


Figure 60: The heat map of confusion matrix of decision tree model before optimization

After optimization, the accuracy is increased to 89%. The precision of class "1" also gets better to 54% (Figure 61).

```
################ Evaluating the decision tree model(After Optimization) #################
Confusion matrix:
[[9601  366]
 [ 898  437]]

#################### output/decisiontree_model/after_optimization/confusion_matrix_After Optimization.png file saved ###################

Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.96      0.94      9967
           1       0.54      0.33      0.41      1335

    accuracy                           0.89     11302
   macro avg       0.73      0.65      0.67     11302
weighted avg       0.87      0.89      0.88     11302
```

Figure 61: Decision tree evaluation results after optimization

The heat map of the confusion matrix of the decision tree is shown below (Figure 62).

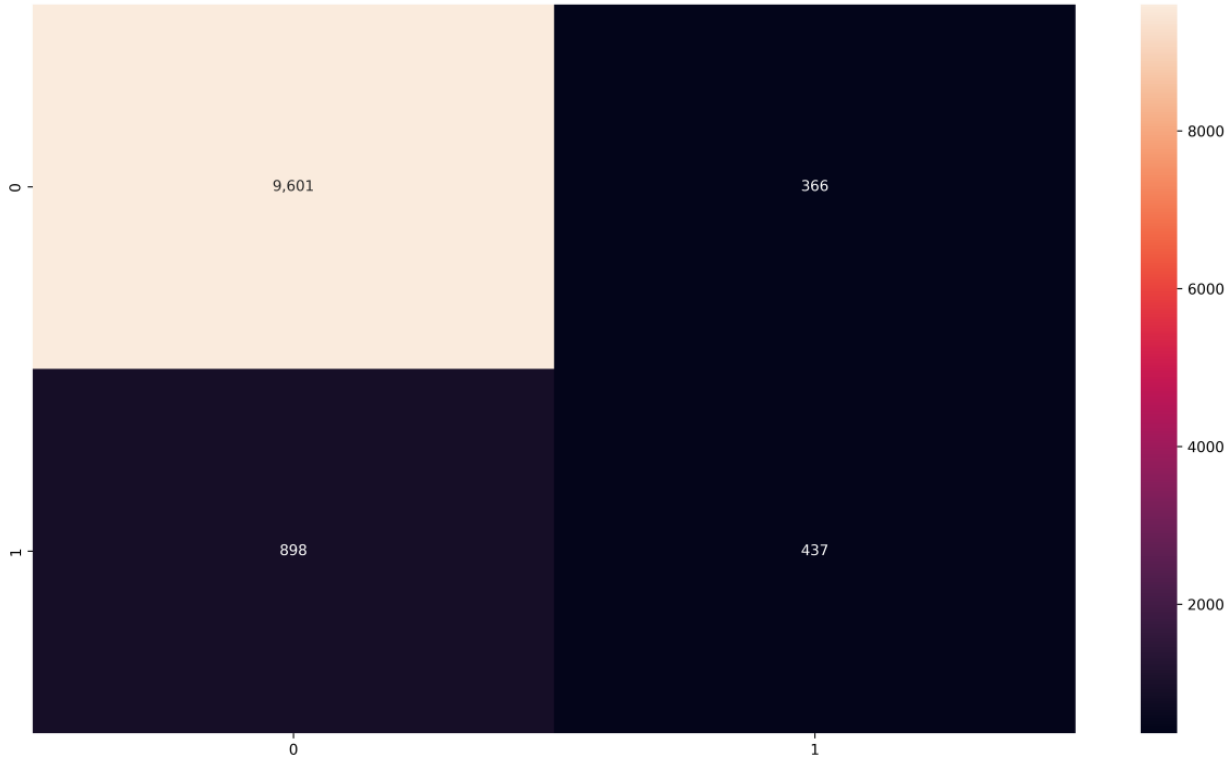


Figure 62: The heat map of confusion matrix of decision tree model after optimization

this study performs the optimization process via hyperparameter tuning using GridSearchCV. The hyperparameter_tuning_classification function is responsible for this task for all algorithms (Figure 63).

```python
def hyperparameter_tuning_classification(data : pd.DataFrame, train_test_sets : list, estimator, param : dict, logger : Logger = None) -> dict:
    logger.log("############ Hyperparameter tuning to optimized model parameters ############\n")
    # grid search to find best classifier parameters (Hyperparameter tuning)

    # Extract train and test sets
    X_train, X_test, y_train, y_test = train_test_sets
    # Some parameters do not match with each other. This code avoids unnecessary warnings
    warnings.filterwarnings("ignore")

   # Create and train gridsearch
    grid = GridSearchCV(estimator, param_grid=param, refit=True, cv=5, n_jobs=-1)
    grid.fit(X_train, y_train)

    # Find the best paramters
    grid_best_params = grid.best_params_
    grid_best_score = grid.best_score_

    logger.log(f"Best Parameters: {grid_best_params}")
    logger.log(f"Best Score: {grid_best_score}")

    return grid_best_params
```

Figure 63: General function for hyperparameter tuning of all algorithms

There are the input parameters for tuning the decision tree (Figure 64).

```python
    # Optimizing the parameters of the decision tree model using hyperparameter tuning
    grid_best_params = hyperparameter_tuning_classification(data=df, train_test_sets=train_test_sets, estimator=DecisionTreeClassifier(), param = {
        "max_depth": [10,50,100,200,None],
        "min_samples_leaf": [1, 2],
        "min_samples_split": [2, 3, 4],
        "random_state": [0, 1, 10, 20, 42, None],
        "max_features" : ["sqrt", "log2", None]
    }, logger=logger)

    # Print elapsed time for the below job
    logger.print_elapsed_time("hyperparameter tuning of decision tree")
```

Figure 64: Parameters for tuning decision tree

The results of hyperparameter tuning of the decision tree are shown below (Figure 65).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'max_depth': 10, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 3, 'random_state': 1}
Best Score: 0.8942570665302453
```

Figure 65: The results of hyperparameter tuning of decision tree

The second algorithm to evaluate is random forest (Figure 66).

```
def evaluate_model_randomforest(clf : DecisionTreeClassifier, data : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFrame, suffix : str = "", output_folder : str = "outp
    # Evaluating random forest model
    logger.log(f"############### Evaluating the random forest model{'(' + suffix + ')' if suffix else ''} ################")
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)

    # Extract confusion matrix and plot its heatmap
    conf_matrix = confusion_matrix(y_test, y_pred)
    logger.log(f"Confusion matrix:\n{conf_matrix}\n")
    sns.heatmap(conf_matrix.reshape(-1,2),annot=True, fmt=',d')
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png file saved ##################\n")

    # log the classification report including class-specific precision, recall, f1-score and their macro avg and weighted avg plus model accuracy
    logger.log(f"Classification Report:\n{classification_report(y_test, y_pred,zero_division=0)}")

    # Extract model feature_importances and plot their barh
    # Creating a dataframe based on the feature importances
    feature_importances = pd.DataFrame(clf.feature_importances_,index=clf.feature_names_in_, columns=["feature_importances"])
    feature_importances = feature_importances.sort_values("feature_importances",axis="index")
    logger.log(f"Importance of features:\n{feature_importances}")
    plt.clf()
    # Create a bar plot to compare feature importances visually
    plt.barh(feature_importances.index, feature_importances.iloc[:,0])
    plt.xlabel("feature_importances")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/feature_importances{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/feature_importances{('_' if suffix else '') + suffix}.png file saved ##################\n")

    logger.log(f"Classes are: {clf.classes_}")
```

Figure 66: Random forest evaluation

The results before optimization (Figure 67) show that the accuracy is 90% which seems great. The precision for 0 and 1 classes are 92% and 62% which indicates better precision for both classes in comparison to the decision tree model.

```
################ Evaluating the random forest model(Before Optimization) #################
Confusion matrix:
[[9651  316]
 [ 812  523]]

#################### output/randomforest_model/before_optimization/confusion_matrix_Before Optimization.png file saved ###################

Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.97      0.94      9967
           1       0.62      0.39      0.48      1335

    accuracy                           0.90     11302
   macro avg       0.77      0.68      0.71     11302
weighted avg       0.89      0.90      0.89     11302
```

Figure 67: Random forest evaluation results before optimization

The heat map of the confusion matrix of the random forest model is shown below (Figure 68).

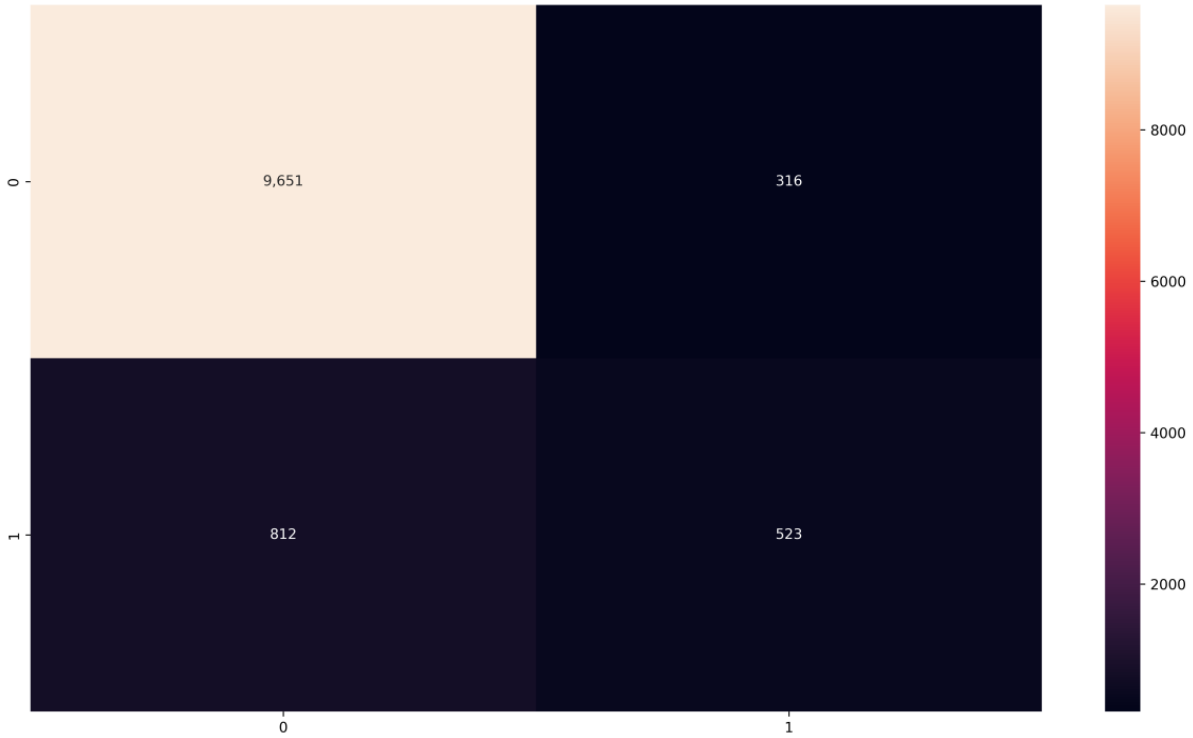


Figure 68: The heat map of the confusion matrix of the random forest before optimization

After optimization (Figure 69) the results (Figure 70) show no significant improvement in the evaluation metrics. Therefore, an accuracy of 90% is the highest accuracy that the random forest algorithm can reach for this problem.

```
# Optimizing the parameters of the random forest model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_classification(data=df, train_test_sets=train_test_sets, estimator=RandomForestClassifier(), param = {
    "max_depth": [10,50,100,200,None],
    "min_samples_leaf": [1, 2],
    "min_samples_split": [2, 3, 4],
    "random_state": [0, 1, 10, 20, 42, None],
    "max_features" : ["sqrt", "log2"]
}, logger=logger)
```

Figure 69: Parameters for tuning random forest

```
################ Evaluating the random forest model(After Optimization) #################
Confusion matrix:
[[9656  311]
 [ 826  509]]

#################### output/randomforest_model/after_optimization/confusion_matrix_After Optimization.png file saved ###################

Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.97      0.94      9967
           1       0.62      0.38      0.47      1335

    accuracy                           0.90     11302
   macro avg       0.77      0.68      0.71     11302
weighted avg       0.89      0.90      0.89     11302
```

Figure 70: Random forest evaluation results after optimization

The heat map of the confusion matrix of the random forest model after optimization is shown below (Figure 71).

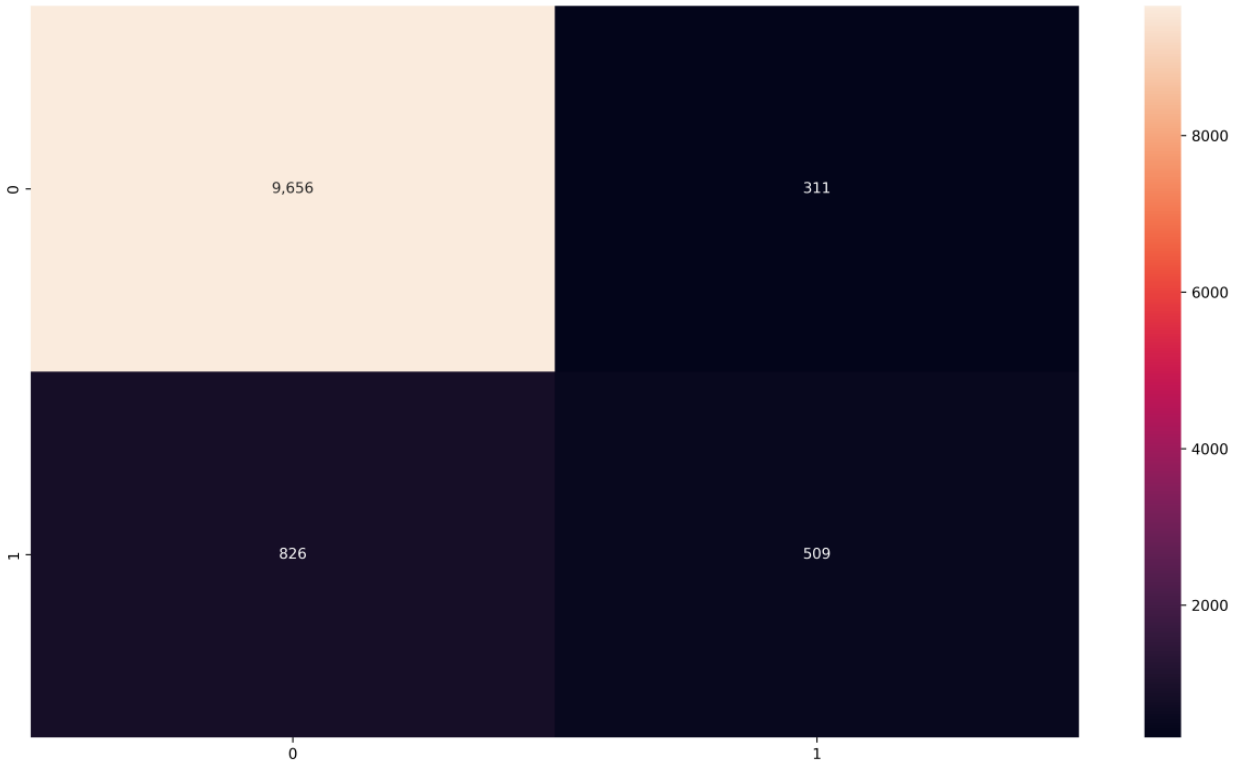


Figure 71: The heat map of confusion matrix of random forest After optimization

The results of hyperparameter tuning are shown below (Figure 72).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 2, 'random_state': None}
Best Score: 0.9008641010214642
```

Figure 72: The results of hyperparameter tuning of random forest

This study also evaluates the logistic regression algorithm (Figure 74) and applies these parameters for tuning (Figure 73).

```
# Optimizing the parameters of the logistic regression model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_classification(data=df, train_test_sets=train_test_sets, estimator=LogisticRegression(), param = {
    "solver": ["newton-cg", "newton-cholesky", "lbfgs", "liblinear", "sag", "saga"],
    "penalty": ["l1", "l2", "elasticnet", "None"],
    "C": [100, 10, 1, 0.1, 0.01],
    "max_iter"  : [100,1000,2500,5000],
    "random_state": [0, 1, 10, 20, 42, None],
}, logger=logger)
```

Figure 73: Parameters for tuning logistic regression

```python
def evaluate_model_logisticregression(clf : LogisticRegression, data : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFrame, suffix : str = "", output_folder : str = "
    # Evaluating logistic regression model
    logger.log(f"################ Evaluating the logistic regression model{'(' + suffix + ')' if suffix else ''} ################")
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)

    # Extract confusion matrix and plot its heatmap
    conf_matrix = confusion_matrix(y_test, y_pred)
    logger.log(f"Confusion matrix:\n{conf_matrix}\n")
    sns.heatmap(conf_matrix.reshape(-1,2),annot=True, fmt=',d')
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png file saved ##################\n")

    # log the classification report including class-specific precision, recall, f1-score and their macro avg and weighted avg plus model accuracy
    logger.log(f"Classification Report:\n{classification_report(y_test, y_pred,zero_division=0)}")

    # Extract model coefficients and plot their barh
    # Creating a dataframe based on the coefficients
    coef = pd.DataFrame(clf.coef_[0],index=clf.feature_names_in_, columns=["Coefficient"])
    coef = coef.sort_values("Coefficient",axis="index")
    logger.log(f"Coefficient of features:\n{coef}")
    plt.clf()
    # Create a bar plot to compare coefficients visually
    plt.barh(coef.index, coef.iloc[:,0])
    plt.xlabel("Coefficient")
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/Coefficients{("_" if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/Coefficient{("_" if suffix else '') + suffix}.png file saved ##################\n")
```

Figure 74: Logistic regression evaluation

The results state accuracy of 88% and precision of 88% and 71% for 0 and 1 classes respectively. This indicates that logistic regression has better precision for the 1 class rather than the previous two algorithms but the recall and f1-score have near zero value for this class. This is caused by the very few results for 1 class in this model and based on this fact it would not be fair to say it has better performance in comparison with the other algorithms. On the other hand, the precision of 88% for the 0 class is more valuable in this model since the recall is 100% which shows that all the 0 class values in the test set are distinguished by this model (Figure 75).

```
################ Evaluating the logistic regression model(Before Optimization) #################
Confusion matrix:
[[9965    2]
 [1330    5]]

Classification Report:
              precision    recall  f1-score   support

           0       0.88      1.00      0.94      9967
           1       0.71      0.00      0.01      1335

    accuracy                           0.88     11302
   macro avg       0.80      0.50      0.47     11302
weighted avg       0.86      0.88      0.83     11302
```

Figure 75: Logistic regression evaluation results before optimization

The heat maps of the confusion matrix for logistic regression are shown below (Figures 77, 78).

```
################ Evaluating the logistic regression model(After Optimization) #################
Confusion matrix:
[[9965    2]
 [1329    6]]

#################### output/logisticregression_model/after_optimization/confusion_matrix_After Optimization.png file saved ##################

Classification Report:
              precision    recall  f1-score   support

           0       0.88      1.00      0.94      9967
           1       0.75      0.00      0.01      1335

    accuracy                           0.88     11302
   macro avg       0.82      0.50      0.47     11302
weighted avg       0.87      0.88      0.83     11302
```

Figure 76: Logistic regression evaluation results after optimization

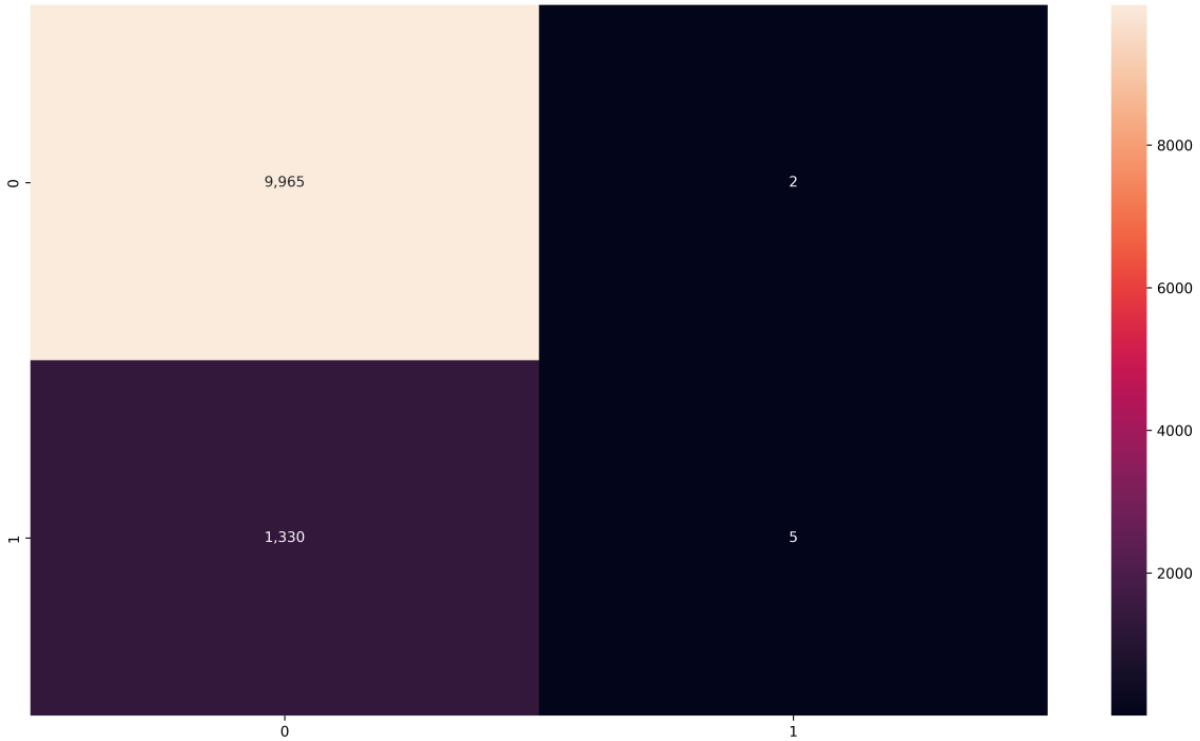


Figure 77: The heat map of confusion matrix of logistic regression before optimization

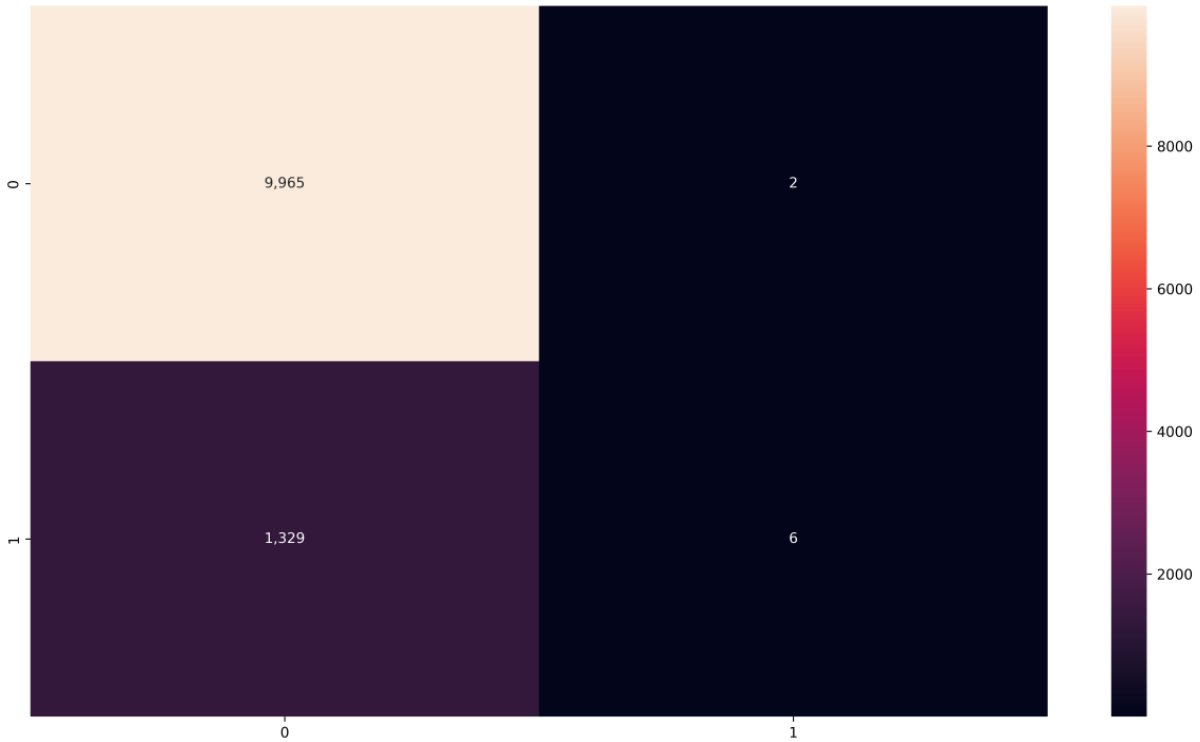


Figure 78: The heat map of confusion matrix of logistic regression after optimization

After optimization, the precision of the “1” class improves slightly but the other metrics ramain the same as before (Figure 76).

The results of hyperparameter tuning of logistic regression is shown below (Figure 79)

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'C': 1, 'max_iter': 100, 'penalty': 'l2', 'random_state': 0, 'solver': 'liblinear'}
Best Score: 0.8834616547450167
```

Figure 79: The results of hyperparameter tuning of logistic regression

The last model is the k-nearest neighbors (KNN) model (Figure 80).

```
def evaluate_model_knn(clf : LogisticRegression, data : pd.DataFrame, y_test : pd.DataFrame, y_pred : pd.DataFrame, suffix : str = "", output_folder : str = "output", logge
    # Evaluating k-nearest neighbour model
    logger.log(f"################ Evaluating the knn model{'(' + suffix + ')' if suffix else ''} ################")
    # set the figure resolution and dpi
    fig = plt.figure(figsize=(16, 9), dpi=600)

    # Extract confusion matrix and plot its heatmap
    conf_matrix = confusion_matrix(y_test, y_pred)
    logger.log(f"Confusion matrix:\n{conf_matrix}\n")
    sns.heatmap(conf_matrix.reshape(-1,2),annot=True, fmt=',d')
    # Save the file with proper dpi
    plt.savefig(fname=f"{output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png", format="png", dpi = fig.dpi)
    logger.log(f"#################### {output_folder}/confusion_matrix{('_' if suffix else '') + suffix}.png file saved ##################\n")

    # log the classification report including class-specific precision, recall, f1-score and their macro avg and weighted avg plus model accuracy
    logger.log(f"Classification Report:\n{classification_report(y_test, y_pred,zero_division=0)}")
```

Figure 80: KNN model evaluation

The results (Figure 81) show that this model provides 89% accuracy and 90% and 55% precision for the “0” and the “1” classes respectively. The values of recall and f1-score are also acceptable for this problem.

```
################ Evaluating the knn model(Before Optimization) #################
Confusion matrix:
[[9789  178]
 [1114  221]]

#################### output/knn_model/before_optimization/confusion_matrix_Before Optimization.png file saved ###################

Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.98      0.94      9967
           1       0.55      0.17      0.25      1335

    accuracy                           0.89     11302
   macro avg       0.73      0.57      0.60     11302
weighted avg       0.86      0.89      0.86     11302
```

Figure 81: KNN evaluation results before optimization

The heat maps of the confusion matrix for this model are available (Figures 82, 83). The results after optimization (Figure 84) show better outcomes for the “1” class precision.

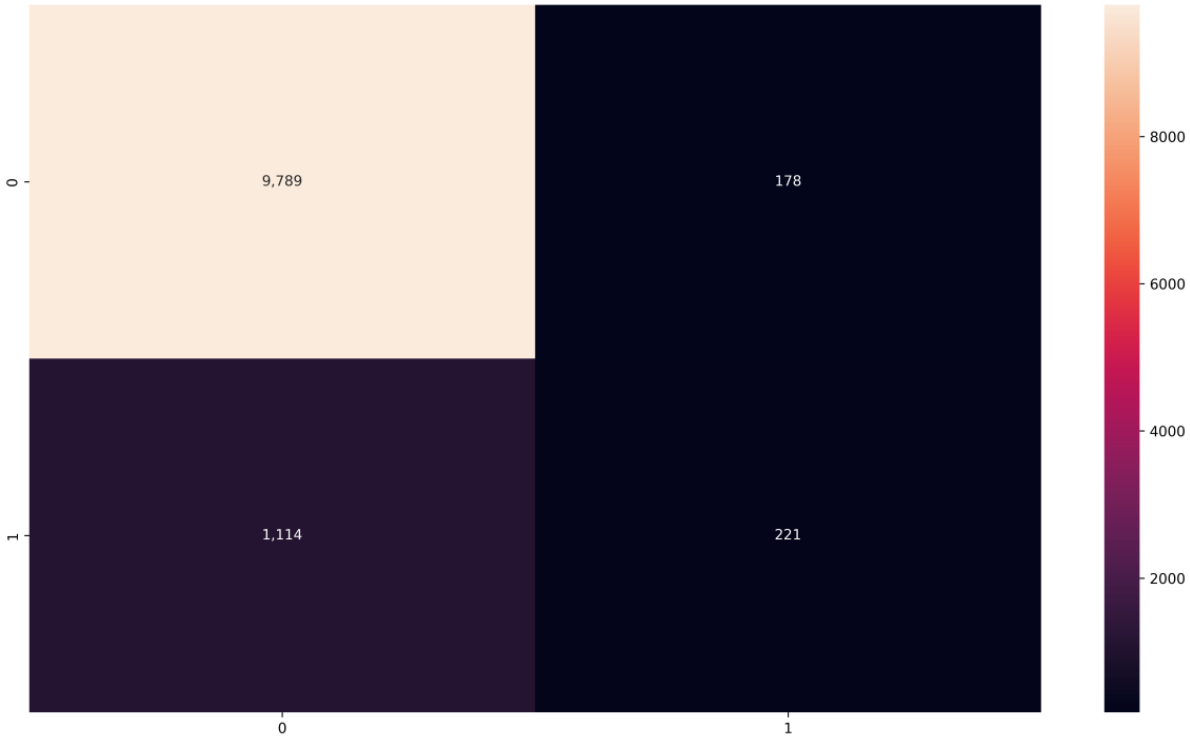


Figure 82: The heat map of confusion matrix for KNN before optimization

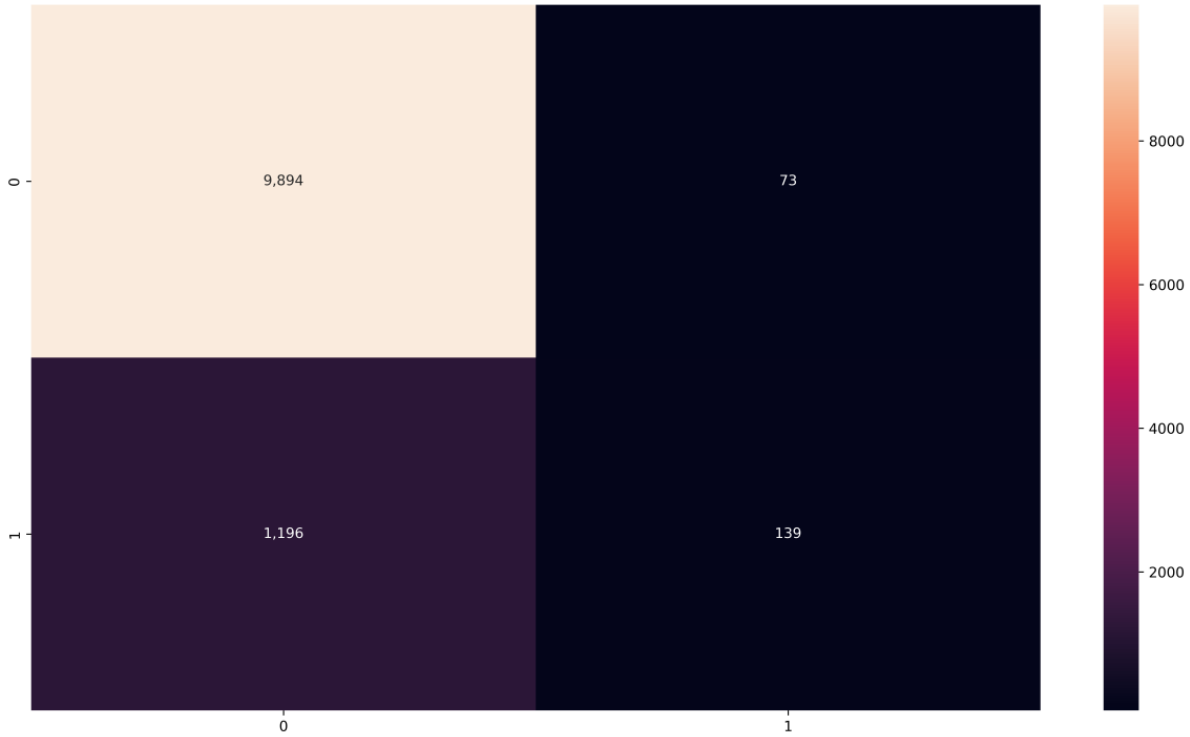


Figure 83: The heat map of confusion matrix for KNN after optimization

```
################ Evaluating the knn model(After Optimization) #################
Confusion matrix:
[[9894   73]
 [1196  139]]

#################### output/knn_model/after_optimization/confusion_matrix_After Optimization.png file saved ###################

Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.99      0.94      9967
           1       0.66      0.10      0.18      1335

    accuracy                           0.89     11302
   macro avg       0.77      0.55      0.56     11302
weighted avg       0.86      0.89      0.85     11302
```

Figure 84: KNN evaluation results after optimization

The parameters for tuning the KNN model are shown below (Figure 85).

```
# Optimizing the parameters of the k-nearest neighbour model using hyperparameter tuning
grid_best_params = hyperparameter_tuning_classification(data=df, train_test_sets=train_test_sets, estimator=KNeighborsClassifier(), param = {
    "n_neighbors": [3,5,7,9,11,13,15],
    "weights": ["uniform","distance"],
    "metric": ["minkowski","euclidean","manhattan"]
}, logger=logger)
```

Figure 85: Parameters for tuning KNN

The results of the hyperparameter tuning of this model is also provided (Figure 86).

```
############ Hyperparameter tuning to optimized model parameters ############

Best Parameters: {'metric': 'manhattan', 'n_neighbors': 11, 'weights': 'uniform'}
Best Score: 0.8902161644609115
```

Figure 86: The results of hyperparameter tuning of KNN

At the end, it could be inetersting to know how much each operation takes to complete (Figure 87).

```
Elapsed time for loading and describing the data: 18 seconds
Elapsed time for cleaning the data: 101 seconds
Elapsed time for classification via decision tree before optimization: 4 seconds
Elapsed time for hyperparameter tuning of decision tree: 41 seconds
Elapsed time for classification via decision tree after optimization: 3 seconds
Elapsed time for classification via random forest before optimization: 7 seconds
Elapsed time for hyperparameter tuning of random forest: 683 seconds
Elapsed time for classification via random forest after optimization: 7 seconds
Elapsed time for classification via logistic regression before optimization: 3 seconds
Elapsed time for hyperparameter tuning of logistic regression: 153 seconds
Elapsed time for classification via logistic regression after optimization: 3 seconds
Elapsed time for classification via K-nearest neighbour before optimization: 3 seconds
Elapsed time for hyperparameter tuning of K-nearest neighbour: 46 seconds
Elapsed time for classification via K-nearest neighbour after optimization: 4 seconds
```

Figure 87: Time elapsed for each operation

At last, this paper provides some charts to compare algorithms based on some of their evaluation metrics after optimization.

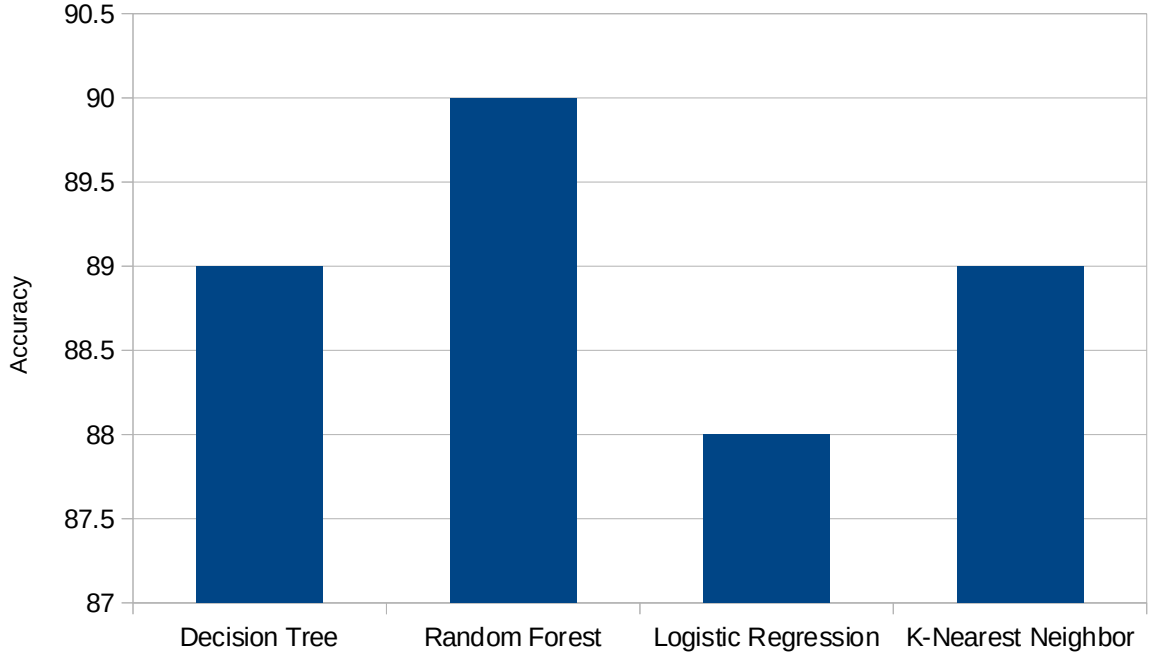


Figure 88: Models comparison based on accuracy

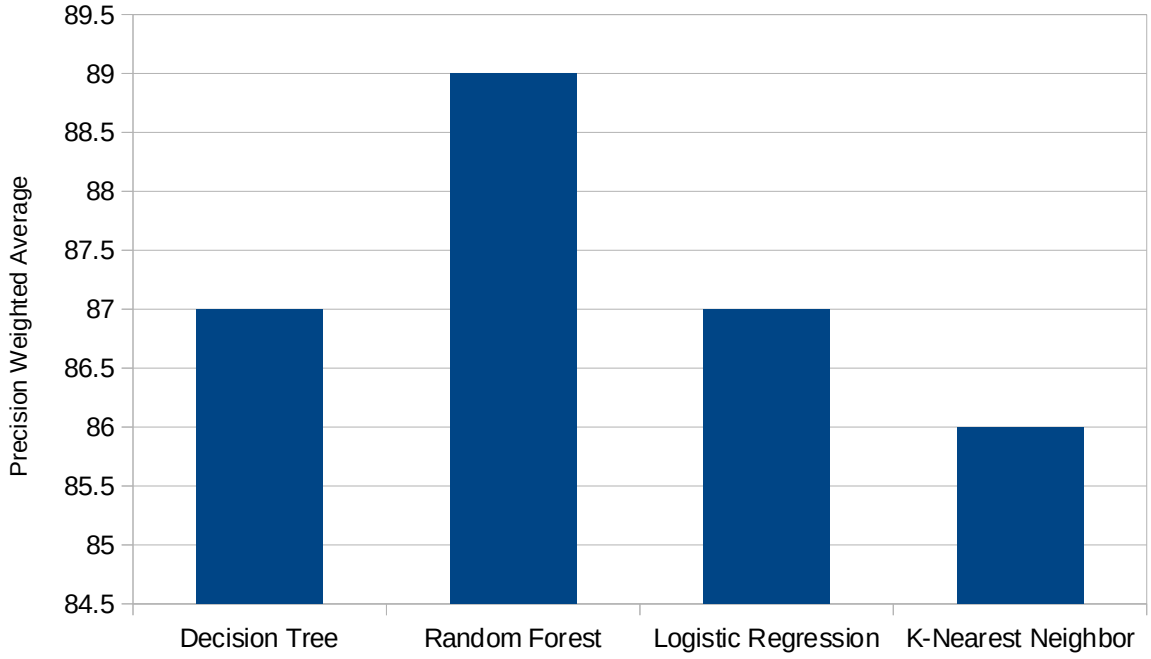


Figure 89: Models comparison based on weighted average precision

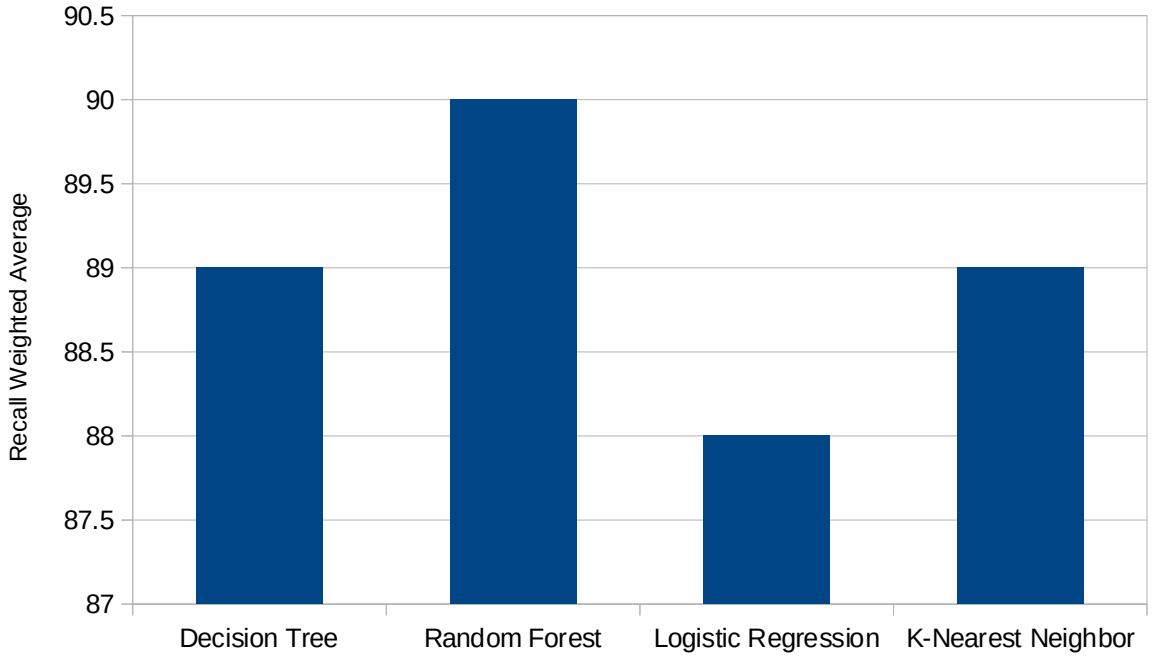


Figure 90: Models comparison based on weighted average recall

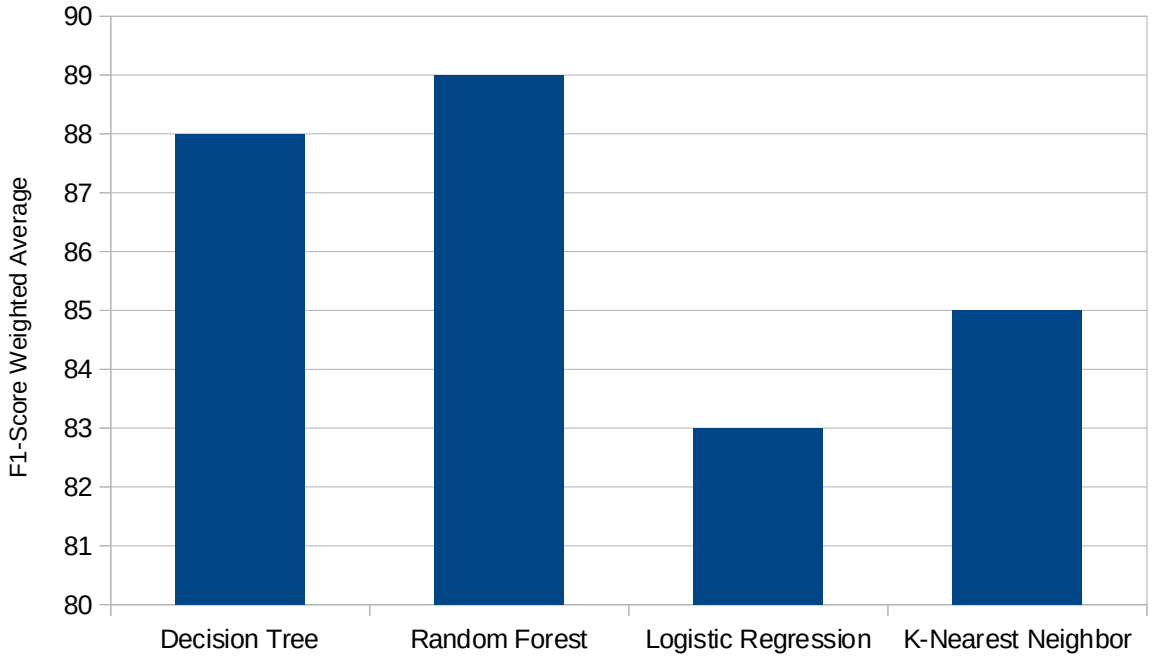


Figure 91: Models comparison based on weighted average f1-score

The above comparisons define that the random forest model has the best performance among all algorithms although it is not the most resource-friendly one.

CONCLUSION

There are different steps done in this study including data exploration and cleaning, model selection, and model interpretation and evaluation. In each step, some information is revealed, some of them approved in the next steps and some others are refuted then. One of the most important findings is about the “dur” feature which is discovered as the most important feature in the interpretation of all models. This finding is also approved by correlation analysis that shows the “dur” column has the highest correlation with the target variable “y”. This study also defines the optimal size of the test set at 25% which is obtained by calculating the proportion of the “1” class in the target variable. Moreover, this paper indicates that the random forest model has the best performance based on evaluation metrics although it may take more time to complete than the others. The highest evaluation metrics achieved by the random forest algorithm in this study are 90%, 89%, 90%, and 89% for accuracy, weighted average of precision, weighted average of recall, and weighted average of f1-score respectively.

BIBLIOGRAPHY

Magaga, A.M. (2021) Identifying, Cleaning and replacing outliers | Titanic Dataset. Available at: https://medium.com/analytics-vidhya/identifying-cleaning-and-replacing-outliers-titanic-dataset-20182a062893 (Accessed: 05 September 2024).

Ruhil, A.V.S. (2021) Data Analysis for Leadership & Public Affairs. Available at: https://people.ohio.edu/ruhil/statsbook/ (Accessed: 05 September 2024).

Australian Bureau of Statistics (2023) Measures of central tendency. Available at: https://www.abs.gov.au/statistics/understanding-statistics/statistical-terms-and-concepts/measures-central-tendency (Accessed: 07 September 2024).

Bonaccorso, G., 2018. Mastering machine learning algorithms: expert techniques to implement popular machine learning algorithms and fine-tune your models. Packt Publishing Ltd.

Graphviz (2021) Graph Visualization Software. Available at: https://graphviz.org/ (Accessed: 12 September 2024).

IBM (2024) What is the k-nearest neighbors (KNN) algorithm?. Available at: https://www.ibm.com/topics/knn#:~:text=What%20is%20the%20KNN%20algorithm,of%20an%20individual%20data%20point. (Accessed: 15 September 2024).

Hunter, J.D. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), 90-95.

Johnston, B. and Mathur, I., 2019. Applied supervised learning with Python: use scikit-learn to build predictive models from real-world datasets and prepare yourself for the future of machine learning. Packt Publishing Ltd.

Mellowacademy (2023) A Step-by-Step Guide to Data Cleaning and Preprocessing. Available at: https://medium.com/@mellowacademy0507/a-step-by-step-guide-to-data-cleaning-and-preprocessing-a095fef9674e (Accessed: 09 September 2024).

Miller, J.D. and Forte, R.M., 2017. Mastering Predictive Analytics with R. Packt Publishing Ltd.

Buhl N. (2023) Training, Validation, Test Split for Machine Learning Datasets. Available at: https://encord.com/blog/train-val-test-split/#:~:text=The%20train%2Dtest%20split%20is,model's%20performance%20and%20generalization%20capabilities.(Accessed: 11 September 2024).

Sachinsoni (2023) K Nearest Neighbours — Introduction to Machine Learning Algorithms Available at: https://medium.com/@sachinsoni600517/k-nearest-neighbours-introduction-to-machine-learning-algorithms-9dbc9d9fb3b2#:~:text=No%20Feature%20Importance%20or%20Coefficients,each%20feature%20on%20the%20prediction. (Accessed: 11 September 2024).

Seaborn (2024) statistical data visualization. Available at: https://seaborn.pydata.org/ (Accessed: 13 September 2024).

TABLE OF FIGURES