Business Definition

Provide a background of your client's business and key business decisions they need to make. Provide specific business question(s) you are helping them answer.

- Background of OnPoint: It was established in 1932 by 16 teachers from Portland Public Schools as the Portland Teachers Credit Union, OnPoint is a community-oriented financial institution providing banking, loans, and credit services. In 2005, it received a community charter in Oregon and Southwest Washington, enabling it to broaden its membership and rebrand as OnPoint Community Credit Union. With a membership exceeding 563,000, OnPoint also offers financial education and community outreach programs.
- Current Situation: OnPoint has recently observed that fewer new savings accounts are being opened compared to previous years. Also, they have been seeing fewer young adults or students opening saving accounts lately. So in response, OnPoint's leadership is curious to understand the underlying socio-economic factors influencing these trends. They want to know whether changes in the broader economic environment, shifts in customer financial behavior, or demographic changes are affecting their ability to attract new savings account holders. Understanding these factors could help them adjust their marketing strategies, develop new products, or tailor their services to better meet the needs of potential customers.
- Questions of Interest

What are the socio-economic factors that determines a person's interest in opening a savings account?

For the business questions above, the outcome variable is stored in the target variable 'y', which is stored as a binary y string variable with 'yes' means the client has interest in opening a saving account, and 'no' means the client has no interests in opening a saving account.

Data Dictionary

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

```
!pip install ucimlrepo
from ucimlrepo import fetch_ucirepo

from ucimlrepo import fetch_ucirepo

# fetch dataset
bank_marketing = fetch_ucirepo(id=222)

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

# join X (independent variables) and y (target variable) as a single dataframe for exploratory data analysis
bank = X.join(y['y'])
```

```
Requirement already satisfied: ucimlrepo in /root/venv/lib/python3.9/site-packages (0.0.7)
Requirement already satisfied: pandas>=1.0.0 in /shared-libs/python3.9/py/lib/python3.9/site-packages (from ucimlrep
Requirement already satisfied: certifi>=2020.12.5 in /shared-libs/python3.9/py/lib/python3.9/site-packages (from uci
Requirement already satisfied: python-dateutil>=2.8.2 in /shared-libs/python3.9/py-core/lib/python3.9/site-packages
Requirement already satisfied: numpy<2,>=1.22.4 in /shared-libs/python3.9/py/lib/python3.9/site-packages (from panda
Requirement already satisfied: pytz>=2020.1 in /shared-libs/python3.9/py/lib/python3.9/site-packages (from pandas>=1
Requirement already satisfied: tzdata>=2022.1 in /shared-libs/python3.9/py/lib/python3.9/site-packages (from pandas>
Requirement already satisfied: six>=1.5 in /shared-libs/python3.9/py-core/lib/python3.9/site-packages (from python-d

[notice] A new release of pip is available: 23.0.1 -> 24.2
[notice] To update, run: pip install --upgrade pip
```

This dataset has 45211 rows (indexed from 0 to 45210) and 16 columns (indexed from 0 to 15). The columns 0-14 are independent variables while 15, 'y', is a candidate for a dependent variable.

```
bank.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
Data columns (total 17 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   age          45211 non-null  int64 
 1   job          44923 non-null  object
 2   marital      45211 non-null  object
 3   education    43354 non-null  object
 4   default      45211 non-null  object
 5   balance      45211 non-null  int64 
 6   housing      45211 non-null  object
 7   loan         45211 non-null  object
 8   contact      32191 non-null  object
 9   day_of_week  45211 non-null  int64 
 10  month        45211 non-null  object
 11  duration     45211 non-null  int64 
 12  campaign     45211 non-null  int64 
 13  pdays        45211 non-null  int64 
 14  previous     45211 non-null  int64 
 15  poutcome     8252 non-null   object
 16  y            45211 non-null  object
dtypes: int64(7), object(10)
memory usage: 5.9+ MB
```

Variable #1/ Column index 0 - age

Business (non-technical) information about the variable: Provides the age of the client.

Technical information about the variable: Integer. No missing values or data input errors

```
bank['age'].mean()
```

```
40.93621021432837
```

```
bank['age'].min()
```

```
18
```

```
bank['age'].max()
```

```
95
```

```
from statistics import stdev
stdev(bank['age'])
```

```
10.61876204097542
```

Statistical information about the variable: mean 40.94y, standard deviation: 10.62, minimum 18y, maximum: 95y.

Variable #2/ Column index 1 - job

Business (non-technical) information about the variable: This specifies the job type of the client.

```
# What are the different unique categories within job?
bank['job'].unique()
```

```
array(['management', 'technician', 'entrepreneur', 'blue-collar', nan,
       'retired', 'admin.', 'services', 'self-employed', 'unemployed',
       'housemaid', 'student'], dtype=object)
```

Technical information about the variable: this data type is object/string (from bank.info()) and this is broken down into 12 categories: 'management', 'technician', 'entrepreneur', 'blue-collar', 'retired', 'admin.', 'services', 'self-employed', 'unemployed', 'housemaid', 'student' and unknown (represented by 'nan'). There are 288 (0.6%) missing values. There are no error except for the unknown.

```
# Count the frequency of the unique categories within the variable Job
df = bank['job'].value_counts()
print(df)
```

```
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
Name: count, dtype: int64
```

Statistical information about the variable: The commonest job is blue-collar (9732) and the least common is student (938).

Variable #3/ Column index 2 - marital

Business (non-technical) information: This specifies the marital status of the client.

```
# What are the different unique categories within marital?
bank['marital'].unique()
```

```
array(['married', 'single', 'divorced'], dtype=object)
```

Technical information about the variable: The data type is Object (from bank.info()). and this is broken down into 3 categories: 'married', 'single', ' divorced'. There are no missing values or data input errors.

```
# Count the frequency of the unique catergories in the 'marital' column
mr = bank['marital'].value_counts()
print(mr)
```

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

Statistical information about the variable: The commonest marital is married (27214), and the least common is divorced (5207).

Variable #4/ Column index 3 - education

Business (non-technical) information about the variable: This specifies the client's educational level attained

```
# What are the different unique categories within education?
bank['education'].unique()
```

```
array(['tertiary', 'secondary', nan, 'primary'], dtype=object)
```

```
# Count the frequency of the unique categories in the 'education' column
ed = bank['education'].value_counts()
print(ed)
```

```
education
secondary    23202
tertiary     13301
primary       6851
Name: count, dtype: int64
```

Technical information about the variable: The data type is Object (from bank.info()). and this is broken down into 3 categories: 'tertiary', 'secondary', and 'primary'. There are 1,857 missing (4% of data set total). There is no data input errors except for the unknown.

Statistical information about the variable: The commonest education is secondary (23202), and the least common is primary (6851).

Variable #5/ Column index 4 - default

Business (non-technical) information about the variable: This specifies whether the client has defaulted on a previous payment "yes" or "no"(has credit in default?)

```
# What are the different unique categories within default?
bank['default'].unique()
```

```
array(['no', 'yes'], dtype=object)
```

```
# Count the frequency of the unique categories in the 'default' column
de = bank['default'].value_counts()
print(de)
```

```
default
no     44396
yes      815
Name: count, dtype: int64
```

Technical information about the variable: The data type is object (from bank.info()) and this is a binary variable with the following possible values: 'no' and 'yes'. There are no missing values or data input errors.

Statistical information about the variable: The commonest answer is no (44396), and the least common is yes (815).

Variable #6/ Column index 5 - balance

Business (non-technical) information about the variable: This shows the amount a client has in their checking's account (average yearly balance)

```
bank['balance'].mean()
```

```
1362.2720576850766
```

```
bank['balance'].min()
```

```
-8019
```

```
bank['balance'].max()
```

```
102127
```

```
stdev(bank['balance'])
```

```
3044.765829168518
```

Technical information about the variable: Integer. No missing values or data input errors

Statistical information about the variable: mean 1362.27, standard deviation 3044.77, minimum -8,019, and maximum 102127

Variable #7/ Column index 6 - housing

Business (non-technical) information about the variable: This shows whether the client has a house loan

```
# What are the different unique categories within housing?
bank['housing'].unique()
```

```
array(['yes', 'no'], dtype=object)
```

```
# Count the frequency of the unique categories in the 'housing' column
ho = bank['housing'].value_counts()
print(ho)
```

```
housing
yes    25130
no     20081
Name: count, dtype: int64
```

Technical information about the variable: The data type is object (from bank.info()) and this is a binary variable with the following possible values: 'no' and 'yes'. There are no missing values or data input errors.

Statistical information about the variable: The commonest answer is yes (25130), and the least common is no (20081).

Variable #8/ Column index 7 - loan

Business (non-technical) information about the variable: Provides information if client has personal loan

```
# What are the different unique categories within loan?
bank['loan'].unique()
```

```
array(['no', 'yes'], dtype=object)
```

Technical information about the variable: The data type is Object (from bank.info()) and this is binary variable with the following possible values: 'no' and 'yes'. There are no missing values or data input errors.

```
# Count the frequency of the unique categories in the 'loan' column
lo = bank['loan'].value_counts()
print(lo)
```

```
loan
no     37967
yes     7244
Name: count, dtype: int64
```

Statistical information about the variable: The commonest personal loan status is No (37967), and the least common is Yes (7244).

Variable #9/ Column index 8 - contact

Business (non-technical) information about the variable: Provides the contact information type of client.

```
# What are the different unique categories within contact?
bank['contact'].unique()
```

```
array([nan, 'cellular', 'telephone'], dtype=object)
```

Technical information about the variable: The data type is Object (from bank.info()) and this is categorial variable with broken down into 3 categories: 'cellular', ' telephone', unknown (represented by nan). There are 13,020 (28.8%) missing values. There are no errors except for the unknown values.

```
# Count the frequency of the unique categories in the 'contact' column
co = bank['contact'].value_counts()
print(co)
```

```
contact
cellular     29285
telephone     2906
Name: count, dtype: int64
```

Statistical information about the variable: The commonest contact is Cellular (29285), and the least common is Telephone (2906).

Variable #10/ Column index 9 - day_of_week

Business (non-technical) information about the variable: Provides the information of the last contact day to client.

```
bank['day_of_week'].unique()
```

```
array([ 5,  6,  7,  8,  9, 12, 13, 14, 15, 16, 19, 20, 21, 23, 26, 27, 28,
       29, 30,  2,  3,  4, 11, 17, 18, 24, 25,  1, 10, 22, 31])
```

Technical information about the variable: The data type is Integer (from bank.info()), but it makes more sense to see it as the 'categorical' variable instead. There is error where variable names (day_of_week) does not agree with the variable values (day_of_month).

```
# Count the frequency of the unique categories in the 'day_of_week' column
mo = bank['day_of_week'].value_counts()
print(mo)
```

```
day_of_week
20    2752
18    2308
21    2026
17    1939
6     1932
5     1910
14    1848
8     1842
28    1830
7     1817
19    1757
29    1745
15    1703
12    1603
13    1585
30    1566
9     1561
11    1479
4     1445
16    1415
2     1293
27    1121
3     1079
26    1035
23     939
22     905
25     840
31     643
10     524
```

Statistical information about the variable: The commonest Day is 20 (2752), and the least common Day is 1 (322).

Variable #11/ Column index 10 - month

Business (non-technical) information about the variable: Provides the information of the last contact month of year to client.

```
# What are the different unique categories within month?
bank['month'].unique()
```

```
array(['may', 'jun', 'jul', 'aug', 'oct', 'nov', 'dec', 'jan', 'feb',
       'mar', 'apr', 'sep'], dtype=object)
```

Technical information about the variable: The data type is Date (from bank.info()) . There are no errors or missing values

```
# Count the frequency of the unique categories in the 'month' column
mo = bank['month'].value_counts()
print(mo)
```

```
month
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
```

Statistical information about the variable: The commonest month is May (13766), and the least common month is December (214).

Variable #12/ Column index 11 - duration

Business (non-technical) information about the variable: indicate the last contact duration to client. This attribute highly affects the output target. The duration is not known before call is performed '0'.

Technical information about the variable: Integer. No missing values or data input errors

```
bank['duration'].mean()
```

```
258.1630797814691
```

```
bank['duration'].min()
```

```
0
```

```
bank['duration'].max()
```

```
4918
```

```
from statistics import stdev
stdev(bank['duration'])
```

```
257.5278122651719
```

Statistical information about the variable: mean 258.16 duration, standard deviation 257.53, minimum 0, and maximum 4918.

Variable #13/ Column index 12 - campaign

Business (non-technical) information about the variable: This indicates the number of phone call attempts made to the client in this current marketing campaign , including the most recent contact.

Technical information about the variable: Integer. (from the bank.info code above). No missing values or data input errors

```
bank['campaign'].mean()
```

```
2.763840658246887
```

```
bank['campaign'].min()
```

```
1
```

```
bank['campaign'].max()
```

```
63
```

```
stdev(bank['campaign'])
```

```
3.098020883279169
```

Statistical information about the variable: mean 2.76 number of phone calls, standard deviation 3.10, minimum 1 phone call, and maximum 63 phone calls were made.

Variable #14/ Column index 13 - pdays

Business (non-technical) information about the variable: this indicates the number of days that have passed since the client was last contacted in any previous marketing campaign. A value of -1 indicates that the client was not previously contacted in any prior campaign. This is a numeric variable measured in days.

Technical information about the variable: Integer. (from the bank.info code above). No missing values or data input errors

```
bank['pdays'].mean()
```

```
40.19782796222158
```

```
bank['pdays'].min()
```

```
-1
```

```
bank['pdays'].max()
```

```
871
```

```
stdev(bank['pdays'])
```

```
100.12874599059835
```

Statistical information about the variable: mean 40.19 days, standard deviation 100.13, minimum -1 days, and maximum 871 days

Variable #15/ Column index 14 - previous

Business (non-technical) information about the variable: this indicates the number of phone call attempts made to the client in the previous marketing campaign.

Technical information about the variable: Integer. (from the bank.info code above). No missing values or data input errors

```
bank['previous'].mean()
```

```
0.5803233726305546
```

```
bank['previous'].min()
```

```
0
```

```
bank['previous'].max()
```

```
275
```

```
stdev(bank['previous'])
```

```
2.3034410449312213
```

Statistical information about the variable: mean 0.58 number of phone calls, standard deviation 2.30, minimum 0 phone call, and maximum 275 phone calls were made.

Variable #16/ Column index 15 - poutcome

Business (non-technical) information about the variable: This specifies the outcome of the previous marketing campaign for the client.

```
# What are the different unique categories within poutcome?
bank['poutcome'].unique()
```

```
array([nan, 'failure', 'other', 'success'], dtype=object)
```

Technical information about the variable: The data type is object/string (from bank.info()) and this is broken down into 4 categories: 'failure', 'other', 'success', and unknown (represented by nan). There are 36,959 (81.75%) missing values. There are no errors except for the missing values.

```
# Count the frequency of the unique categories within the variable Poutcome
df = bank['poutcome'].value_counts()
print(df)
```

```
poutcome
failure    4901
other      1840
success    1511
Name: count, dtype: int64
```

Statistical information about the variable: The commonest outcome is failure (4901) and the least common is success (1511).

Variable #17/ Column index 16 - y

Business (non-technical) information about the variable: This indicates whether the client has subscribed to a term deposit as a result of this current marketing campaign. The variable "Y" is the target variable, aka the response or explained variable, which is what you are trying to predict or explain.

```
# What are the different unique categories within 'y'?
bank['y'].unique()
```

```
array(['no', 'yes'], dtype=object)
```

Technical information about the variable: The data type is object/string (from bank.info()) and this is binary variable with the following possible values: 'no' and 'yes'. There are no errors.

```
# Count the frequency of the unique categories within the variable 'y'
df = bank['y'].value_counts()
print(df)
```

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

Statistical information about the variable: The most common result is 'no' (39,922), and the least common is 'yes' (5,289)

Data Cleaning

1. Dropping unnecessary variables

Given the business question, "What are the socio-economic factors that determine a person's interest in opening a savings account?", we drop certain variables to focus on those most relevant to socio-economic status. Variables that are drop includes:

- Contact, day_of_week, and month are related to the logistics of the marketing campaign rather than the socio-economic characteristics of the clients.
- Duration is excluded because it is known only after the call and doesn't reflect socio-economic factors, making it unsuitable for realistic predictions.
- Campaign, pdays, previous, and poutcome are more about campaign history and persistence rather than the socio-economic attributes of individuals.

By removing these variables, we ensure that our analysis focuses on the socio-economic factors directly relevant to predicting a person's interest in opening a savings account.

```
#Drop the variables that are not relevant to answer the question stated in the business definition
bank2 = bank.drop(['contact','day_of_week','month','duration','campaign','pdays','previous','poutcome'], axis
bank2.head()
```

	age int64	**job** object	**marital** object	**education** object	**default** object	**balance** int64	**housin**
0	58	management	married	tertiary	no	2143	yes
1	44	technician	single	secondary	no	29	yes
2	33	entrepreneur	married	secondary	no	2	yes
3	47	blue-collar	married	nan	no	1506	yes
4	33	nan	single	nan	no	1	no

5 rows, 9 cols, showing 10 rows/page Page 1 of 1

2. Cleaning the missing values

First, we check for the number of missing values:

```
a=bank2.isna().sum()
print(a)
```

```
age             0
job           288
marital         0
education    1857
default         0
balance         0
housing         0
loan            0
y               0
dtype: int64
```

From the above table, we can see that 2 variables have missing values, job (288), education (1857).

Given the low number of missing values compared to the entire dataset (less than 5%) so we decided to delete those rows that have the missing values

```
print('Total missing values before cleaning: ',a.sum())
bank2= bank2.dropna()
b=bank2.isna().sum()
print('Total missing values after cleaning: ',b.sum())
```

```
Total missing values before cleaning:  2145
Total missing values after cleaning:  0
```

Data Exploration

```
bank2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 43193 entries, 0 to 45210
Data columns (total 9 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   age        43193 non-null  int64 
 1   job        43193 non-null  object
 2   marital    43193 non-null  object
 3   education  43193 non-null  object
 4   default    43193 non-null  object
 5   balance    43193 non-null  int64 
 6   housing    43193 non-null  object
 7   loan       43193 non-null  object
 8   y          43193 non-null  object
dtypes: int64(2), object(7)
memory usage: 3.3+ MB
```

1. Relationship between target variable 'y' and feature variable 'age'

First, we are creating a new column called "age_group" in our dataset. This column groups the ages into four categories: 18-30, 31-50, 51-70, and 71-95. We use the pd.cut() function to assign each person to one of these age groups based on their age.

```
#the pandas 'Cut' method is used, and the values of x are assigned to the coare assigned to the column age, a
bank2['age_group'] = pd.cut(bank2['age'], bins = [17, 30, 50, 70, 95], labels = ['18-30', '31-50', '51-70', '
bank2.head()
```

	age int64	job object	marital object	education object	default object	balance int64	housin
0	58	management	married	tertiary	no	2143	yes
1	44	technician	single	secondary	no	29	yes
2	33	entrepreneur	married	secondary	no	2	yes
5	35	management	married	tertiary	no	231	yes
6	28	management	single	tertiary	no	447	yes

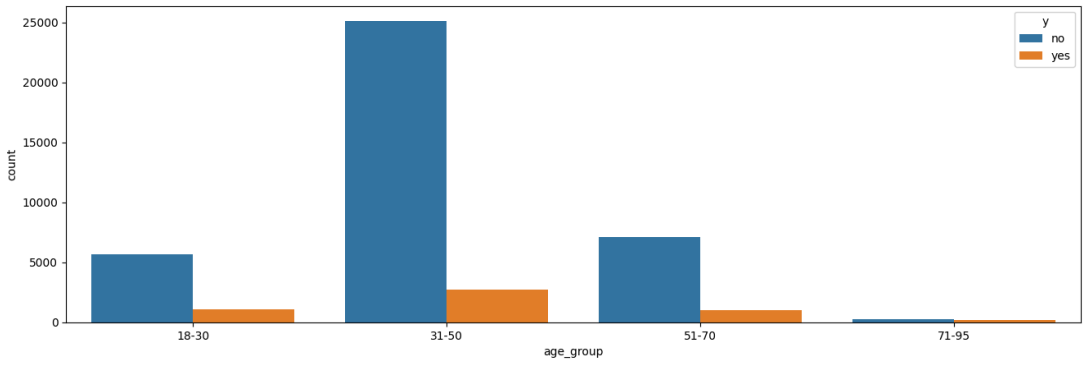
5 rows, 10 cols, showing 50 rows/page Page 1 of 1

Next, we calculate how many people in each age group said "yes" or "no" to subscribing to a term deposit. We do this by grouping the data by age_group and y. The .size() function counts the number of people in each group, and .unstack() helps us organize the results so that "yes" and "no" are shown in separate columns. Then, we create a bar plot to show the counts of people in each age group who either subscribed (yes) or did not subscribe (no) to the term deposit. The hue parameter helps us to use different colors for "yes" and "no" responses within the same plot.

```
#Frequency Distribution
age_y_counts = bank2.groupby(['age_group', 'y']).size().unstack(fill_value=0)
print(age_y_counts)

#Plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "age_group", hue ="y")
plt.show()
```

```
y             no   yes
age_group
18-30       5688  1091
31-50      25111  2713
51-70       7135  1021
71-95        238   196
/tmp/ipykernel_389/3555017892.py:2: FutureWarning: The default of observed=False is deprecated and will be changed t
  age_y_counts = bank2.groupby(['age_group', 'y']).size().unstack(fill_value=0)
```



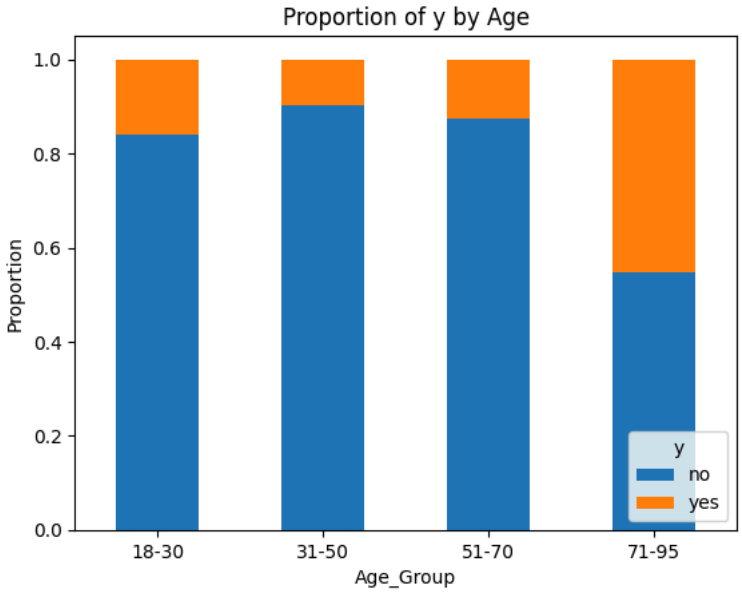
The plot above visualizes the distribution of clients across different age groups (18-30, 31-50, 51-70, 71-95) and whether they subscribed to the term deposit (y). It is evident that the 31-50 age group is the largest, with a significantly higher number of clients who did not subscribe (no) compared to those who did (yes). This trend is consistent in the 18-30 and 51-70 age groups, where the number of non-subscribers far exceeds that of subscribers. However, the 71-95 age group presents a different pattern, with a more balanced distribution between subscribers and non-subscribers, indicating that older clients are more inclined to subscribe to the term deposit.

Next, we calculate the proportion of "yes" and "no" responses within each age group. We do this by dividing the count of "yes" and "no" by the total number of people in that age group. This way, we can see what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each age group. Stacked bars make it easy to compare the percentages across different age groups.

```
#Calculate the proportion of 'yes' and 'no' responses for each age.
age_y_proportion = age_y_counts.div(age_y_counts.sum(axis=1), axis=0)
print(age_y_proportion)

# Plot the stacked bar chart
age_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Age_Group')
plt.ylabel('Proportion')
plt.title('Proportion of y by Age')
plt.xticks(rotation=360)
plt.legend(title='y', loc='lower right')
plt.show()
```

```
y                no       yes
age_group
18–30       0.839062  0.160938
31–50       0.902494  0.097506
51–70       0.874816  0.125184
71–95       0.548387  0.451613
```



In the second plot, we examine the proportion of yes and no responses within each age group. The stacked bar chart shows that the younger age groups (18-30, 31-50 and 51-70) have a higher proportion of clients who did not subscribe. In contrast, the 71-95 age group shows a significant shift, with nearly half of the clients subscribing to the term deposit. This suggests that as clients age, their likelihood of subscribing increases, particularly in the older age group of 71-95. These findings highlight the importance of considering age when predicting subscription behavior, as older clients may represent a more receptive audience for marketing campaigns focused on term deposits.

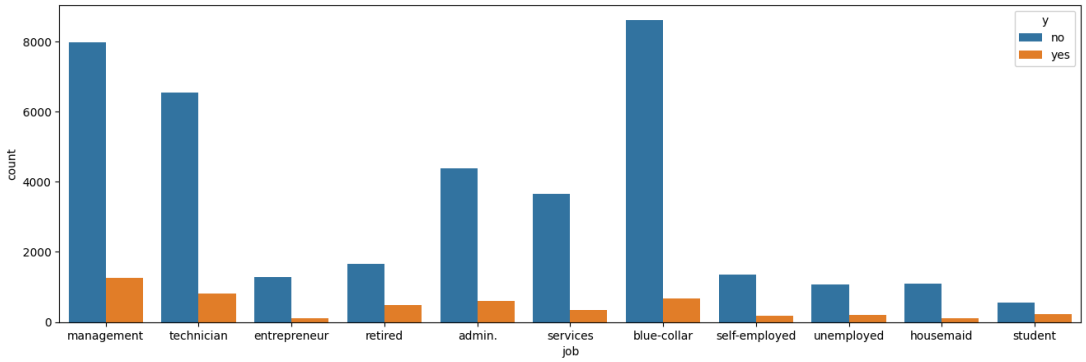
2. Relationship between target variable 'y' and feature variable 'job'

First, we count the number of people in each job tended to say "Yes" or "No" to subscribing opening accounts. We do this by grouping the data by job and y. The .size() function counts the number of people in each job group, and .unstack() helps us organize the results so that "yes" and "no" are shown in separate columns. To visualize, we create a bar plot to show the counts of people in each job group who either subscribed (yes) or did not subscribe (no) to the term deposit. The hue parameter helps us to use different colors for "yes" and "no" responses within the same plot.

```
#Frequency Distribution
job_y_counts = bank2.groupby(['job', 'y']).size().unstack(fill_value=0)
print(job_y_counts)

# plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "job", hue ="y")
plt.show()
```

```
y               no   yes
job
admin.        4387   613
blue-collar   8603   675
entrepreneur  1295   116
housemaid     1090   105
management    7963  1253
retired       1659   486
self-employed 1358   182
services      3654   350
student        549   226
technician    6538   817
unemployed    1076   198
```



The plot above visualizes the distribution of clients across different job groups (management, technician, entrepreneur, retired, admin, services job, blue-collar, self-employed, unemployed, housemaid, student) and whether they subscribed to the term deposit (y). It is evident that the blue-collar group is the largest, with a significantly higher number of clients (8603) who did not subscribe (no) compared to those who did (yes). This trend is consistent in the others job groups, where the number of non-subscribers far exceeds that of subscribers. However, the student group presents a bit different pattern, with a more balanced distribution between subscribers and non-subscribers, indicating that student clients are more inclined to subscribe to the term deposit.

Next, we calculate the proportion of "yes" and "no" responses within each job group. We do this by dividing the count of "yes" and "no" by the total number of people in that job group. This way, we can see what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each job group. Stacked bars make it easy to compare the percentages across different groups.

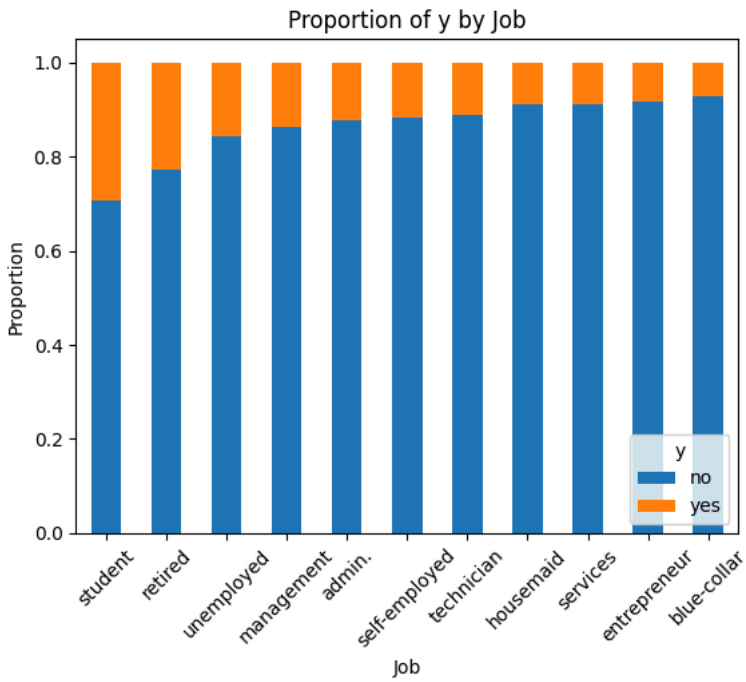
```
#Calculate the proportion of 'yes' and 'no' responses for each job category.
job_y_proportion = job_y_counts.div(job_y_counts.sum(axis=1), axis=0)

# Sort by 'yes' proportion in descending order
job_y_proportion = job_y_proportion.sort_values(by='yes', ascending=False)

print(job_y_proportion)

# Plotting
job_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Job')
plt.ylabel('Proportion')
plt.title('Proportion of y by Job')
plt.legend(title='y', loc='lower right')
plt.xticks(rotation=45)
plt.show()
```

```
y                   no       yes
job
student         0.708387  0.291613
retired         0.773427  0.226573
unemployed      0.844584  0.155416
management      0.864041  0.135959
admin.          0.877400  0.122600
self-employed   0.881818  0.118182
technician      0.888919  0.111081
housemaid       0.912134  0.087866
services        0.912587  0.087413
entrepreneur    0.917789  0.082211
blue-collar     0.927247  0.072753
```



To get the visualization, we examine the proportion of yes and no responses within each group in the second chart. The stacked bar chart reveals that the most of job groups have a higher proportion of clients who did not subscribe. However, student and retired groups are more likelihood of subscribing increases to the term deposit compared to other groups. These findings highlight the importance of considering job background when predicting subscription behavior, as students and retired (older clients) may represent a more receptive audience for marketing campaigns focused on term deposits.

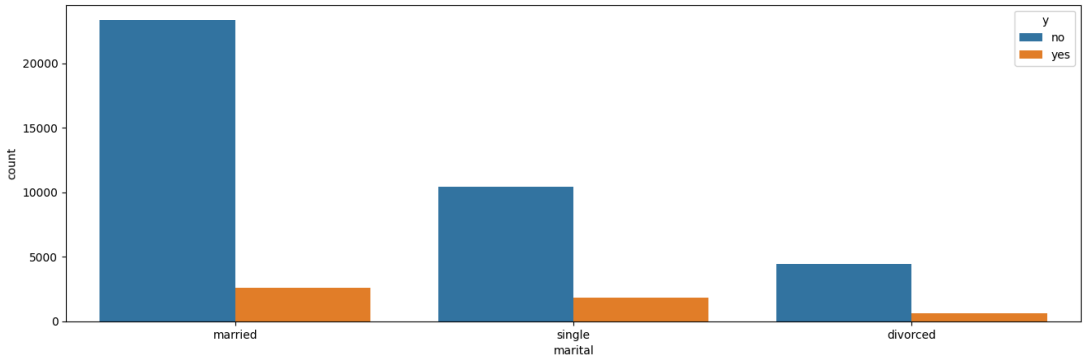
3. Relationship between target variable 'y' and feature variable 'marital'

First, we count the number of people in each marital status tended to say "Yes" or "No" to subscribing opening accounts. We do this by grouping the data by marital and y. The .size() function counts the number of people in each marital status group, and .unstack() helps us organize the results so that "yes" and "no" are shown in separate columns. To visualize, we create a bar plot to show the counts of people in each group who either subscribed (yes) or did not subscribe (no) to the term deposit. The hue parameter helps us to use different colors for "yes" and "no" responses within the same plot.

```
#Frequency Distribution
marital_y_counts = bank2.groupby(['marital', 'y']).size().unstack(fill_value=0)
print(marital_y_counts)

# Plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "marital", hue ="y")
plt.show()
```

```
y            no   yes
marital
divorced   4430   598
married   23343  2603
single    10399  1820
```



The plot above visualizes the distribution of a client's martial status, and whether they subscribed to the term deposit (y). It is evident that the married group is the largest, with a significantly higher number of clients (23343) who did not subscribe (no) compared to those who did (yes). This trend is consistent in the single, and divorced categories as well where the number of non-subscribers far exceeds that of subscribers. However, the single group seems to be the one with a higher number of individuals per group (18%) saying yes to opening a savings account vs the married, and divorced group which are showing 11% and 13% respectively.

Next, we calculate the proportion of "yes" and "no" responses within each marital group. We do this by dividing the count of "yes" and "no" by the total number of people in that marital group. This way, we can see what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each marital group. Stacked bars make it easy to compare the percentages across different groups.

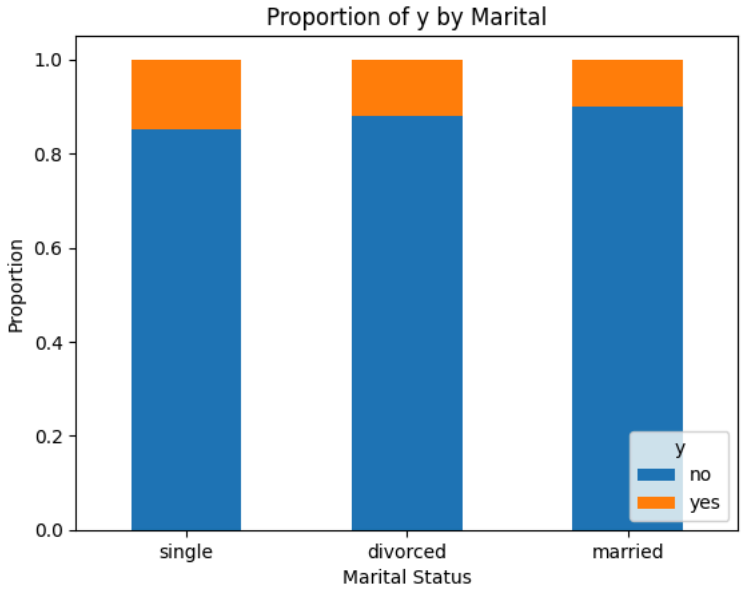
```
#Calculate the proportion of 'yes' and 'no' responses for each marital category.
marital_y_proportion = marital_y_counts.div(marital_y_counts.sum(axis=1), axis=0)

# Sort by 'yes' proportion in descending order
marital_y_proportion =marital_y_proportion.sort_values(by='yes', ascending=False)

print(marital_y_proportion)

# Plotting
marital_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Marital Status')
plt.ylabel('Proportion')
plt.title('Proportion of y by Marital')
plt.legend(title='y', loc='lower right')
plt.xticks(rotation=360)
plt.show()
```

```
y              no      yes
marital
single    0.851052  0.148948
divorced  0.881066  0.118934
married   0.899676  0.100324
```



To get the visualization, we examine the proportion of yes and no responses within each group in the second chart. The single martial group has more likelihood of subscribing increases to the term deposit compared to other groups. These findings highlight the importance of considering marital background when predicting subscription behavior, as single clients may represent a more receptive audience for marketing campaigns focused on term deposits.

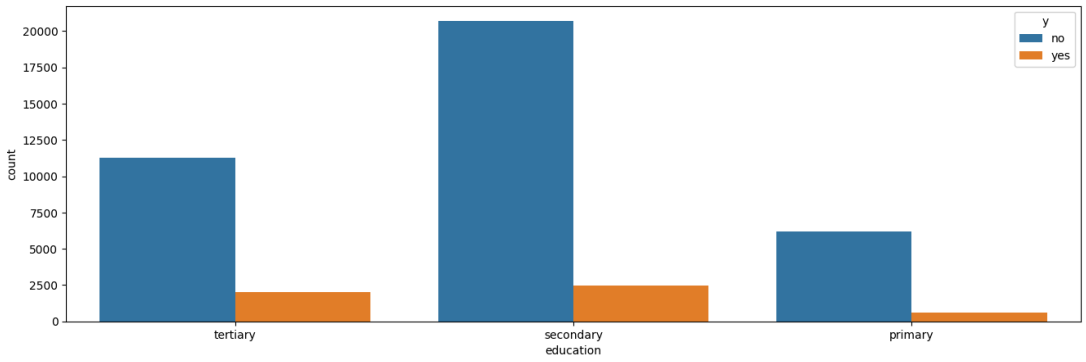
4. Relationship between target variable 'y' and feature variable 'education'

At first, we count the number of people in each education status tended to say "Yes" or "No" to subscribing opening accounts. We do this by grouping the data by education and y. The .size() function counts the number of people in each education group, and .unstack() helps us organize the results so that "yes" and "no" are shown in separate columns. To visualize, we create a bar plot to show the counts of people in each education group who either subscribed (yes) or did not subscribe (no) to the term deposit. The hue parameter helps us to use different colors for "yes" and "no" responses within the same plot.

```
#Frequency Distribution
education_y_counts = bank2.groupby(['education', 'y']).size().unstack(fill_value=0)
print(education_y_counts)

# Plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "education", hue ="y")
plt.show()
```

```
y             no   yes
education
primary     6212   588
secondary  20690  2441
tertiary   11270  1992
```



The plot above visualizes the distribution of clients across different education groups (primary, secondary, and tertiary) and whether they subscribed to the term deposit (y). It is evident that secondary group is the largest, with the highest number of clients (20690) who did not subscribe (no) compared to those who did (yes). This trend is consistent in the others education groups, where the number of non-subscribers far exceeds that of subscribers. This does not have the big change between education groups.

Next, we calculate the proportion of "yes" and "no" responses within each education group. We do this by dividing the count of "yes" and "no" by the total number of people in that education group. By this way, we can observe what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each education group. Stacked bars make it easy to compare the percentages across different groups.

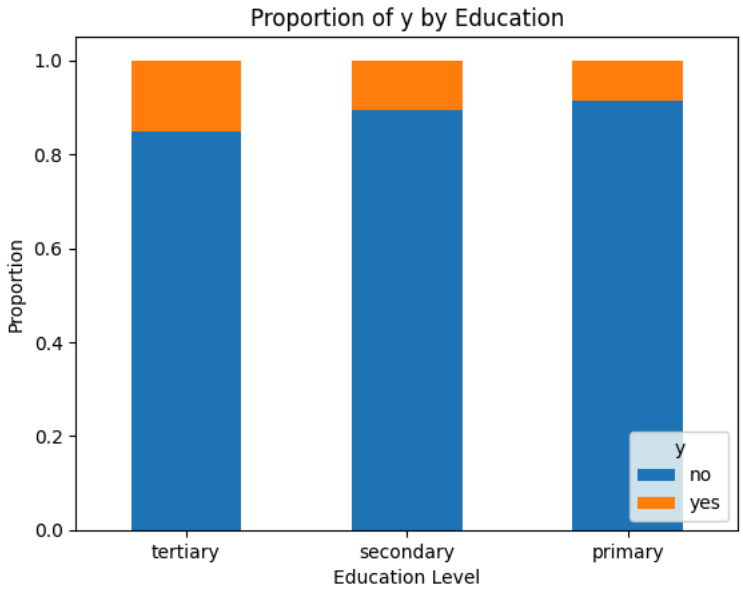
```
#Calculate the proportion of 'yes' and 'no' responses for each education category.
education_y_proportion = education_y_counts.div(education_y_counts.sum(axis=1), axis=0)

# Sort by 'yes' proportion in descending order
education_y_proportion =education_y_proportion.sort_values(by='yes', ascending=False)

print(education_y_proportion)

# Plotting
education_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Education Level')
plt.ylabel('Proportion')
plt.title('Proportion of y by Education')
plt.legend(title='y', loc='lower right')
plt.xticks(rotation=360)
plt.show()
```

```
y                no       yes
education
tertiary   0.849796  0.150204
secondary  0.894471  0.105529
primary    0.913529  0.086471
```



To get the visualization, we examine the proportion of yes and no responses within each group in the second chart. The stacked bar chart reveals that the most of education level groups have a higher proportion of clients who did not subscribe. Tertiary groups are more likelihood of subscribing increases to the term deposit compared to other groups.These findings highlight the importance of considering education level background when predicting subscription behavior, as higher level education may represent a more receptive audience for marketing campaigns focused on term deposit.

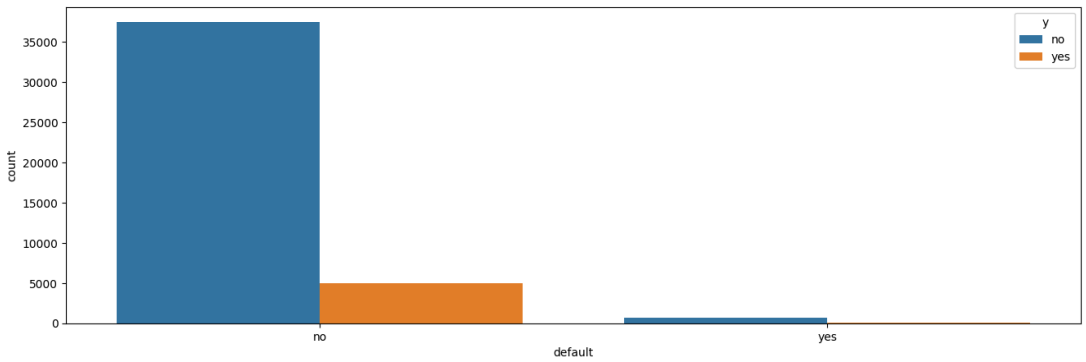
5. Relationship between target variable 'y' and feature variable 'default'

First, we count the number of people in each defaulted payment status tended to say "Yes" or "No" to subscribing opening accounts. We do this by grouping the data by default and y. The .size() function counts the number of people in each default status group, and .unstack() helps us organize the results so that "yes" and "no" are shown in separate columns. To visualize, we create a bar plot to show the counts of people in each group who either subscribed (yes) or did not subscribe (no) to the term deposit. The hue parameter helps us to use different colors for "yes" and "no" responses within the same plot.

```
#Frequency Distribution
default_y_counts = bank2.groupby(['default', 'y']).size().unstack(fill_value=0)
print(default_y_counts)

# Plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "default", hue ="y")
plt.show()
```

```
y           no   yes
default
no       37438  4973
yes        734    48
```



The plot above visualizes the distribution of clients across previous payment default history (yes, or no) and whether they subscribed to the term deposit (y). It is evident that the group that did not default on their payment (no) is the largest, with the highest number of clients (37438) who did not subscribe (no) compared to those who did (yes).

Next, we calculate the proportion of "yes" and "no" responses within each default group. We do this by dividing the count of "yes" and "no" by the total number of people in that default group. By this way, we can observe what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each default group. Stacked bars make it easy to compare the percentages across different groups.

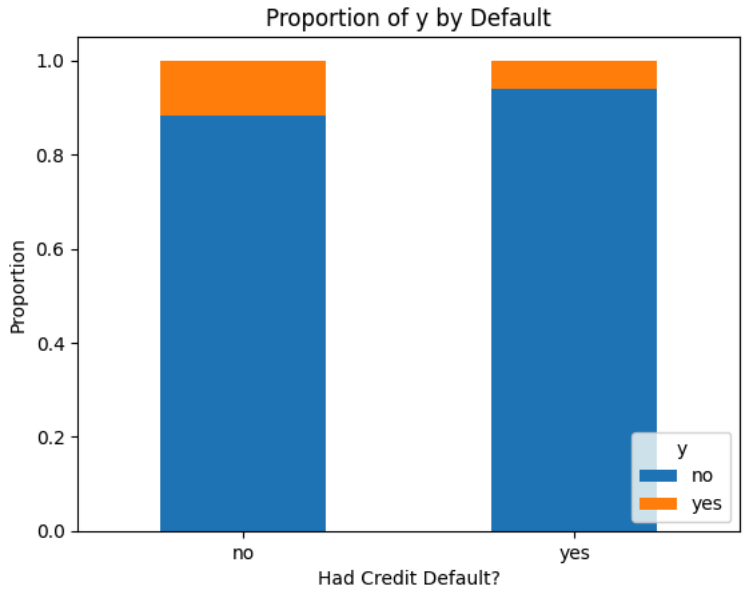
```
#Calculate the proportion of 'yes' and 'no' responses for each education category.
default_y_proportion = default_y_counts.div(default_y_counts.sum(axis=1), axis=0)

# Sort by 'yes' proportion in descending order
default_y_proportion =default_y_proportion.sort_values(by='yes', ascending=False)

print(default_y_proportion)

# Plotting
default_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Had Credit Default?')
plt.ylabel('Proportion')
plt.title('Proportion of y by Default')
plt.legend(title='y', loc='lower right')
plt.xticks(rotation=0)
plt.show()
```

```
y              no       yes
default
no       0.882743  0.117257
yes      0.938619  0.061381
```



To get the visualization, we examine the proportion of yes and no responses within each group in the second chart. The individuals who have not defaulted on a payment have a high likelihood of subscribing to the term deposit compared to other groups. These findings highlight the importance of considering whether a client has defaulted on a previous payment "yes" or "no"(has credit in default?)when predicting subscription behavior as clients who have not defaulted on payments are a more receptive audience for marketing campaigns focused on term deposits.

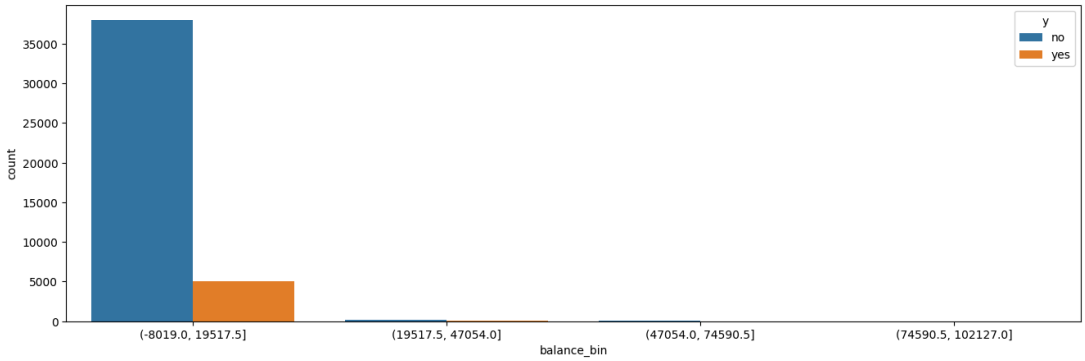
6. Relationship between target variable 'y' and feature variable 'balance'

```
# Binning the balance
bins = np.linspace(bank2['balance'].min(), bank2['balance'].max(),5 )
bank2['balance_bin'] = pd.cut(bank2['balance'], bins)
```

```
#Frequency Distribution
balance_y_counts = bank2.groupby(['balance_bin', 'y']).size().unstack(fill_value=0)
print(balance_y_counts)

# Plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "balance_bin", hue ="y")
plt.show()
```

```
/tmp/ipykernel_389/4224440018.py:2: FutureWarning: The default of observed=False is deprecated and will be changed t
  balance_y_counts = bank2.groupby(['balance_bin', 'y']).size().unstack(fill_value=0)
y                      no   yes
balance_bin
(-8019.0, 19517.5]   38006  4993
(19517.5, 47054.0]     152    24
(47054.0, 74590.5]      11     2
(74590.5, 102127.0]      2     2
```

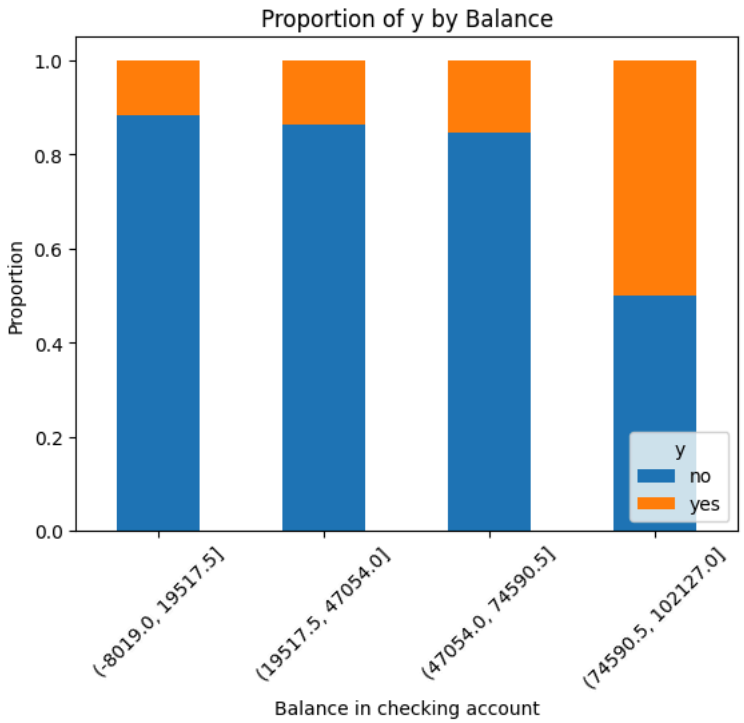


The plot above visualizes the distribution of clients based on the average yearly balance in their checking's account and whether they subscribed to the term deposit (y). It is evident that the group with a low average yearly balance is the highest number of clients who did not subscribe (no) . This variable has limited amount of data, thus the reason why the graph above is right-skewed.

Next, we calculate the proportion of "yes" and "no" responses within each balance category. We do this by dividing the count of "yes" and "no" by the total number of people in that balance category. By this way, we can observe what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each balance category. Stacked bars make it easy to compare the percentages across different categories.

```
#Calculate the proportion of 'yes' and 'no' responses for each balance category.
balance_y_proportion = balance_y_counts.div(balance_y_counts.sum(axis=1), axis=0)

# Plotting
balance_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Balance in checking account')
plt.ylabel('Proportion')
plt.title('Proportion of y by Balance')
plt.legend(title='y', loc='lower right')
plt.xticks(rotation=45)
plt.show()
```



To get the visualization, we examine the proportion of yes and no responses within each category in the second chart. The individuals who have a balance of $74591 -$102127 have a high likelihood of subscribing yes to the term deposit compared to other categories. These findings highlight the importance of considering a client's account balance when predicting subscription behavior as clients who have higher balances are a more receptive audience for marketing campaigns focused on term deposits.

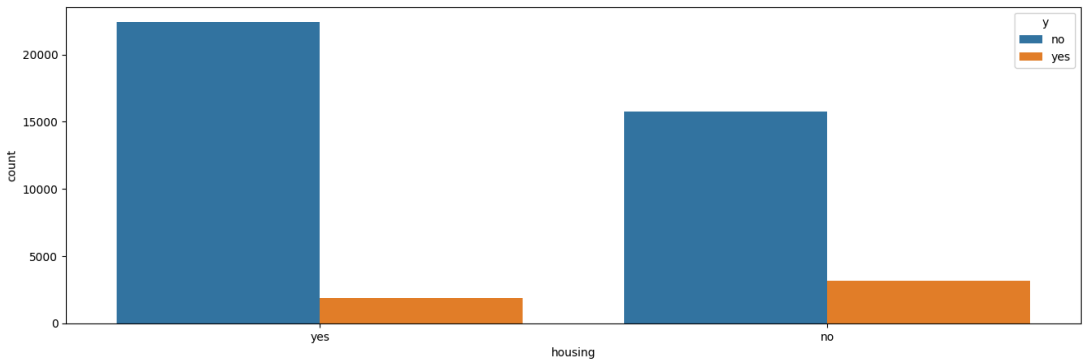
7. Relationship between target variable 'y' and feature variable 'housing'

At first, we count the number of people in each housing loan status tended to say "Yes" or "No" to subscribing opening accounts. We do this by grouping the data by housing and y. The .size() function counts the number of people in each housing group, and .unstack() helps us organize the results so that "yes" and "no" are shown in separate columns. To visualize, we create a bar plot to show the counts of people in each housing group who either subscribed (yes) or did not subscribe (no) to the term deposit. The hue parameter helps us to use different colors for "yes" and "no" responses within the same plot.

```
#Frequency Distribution
housing_y_counts = bank2.groupby(['housing', 'y']).size().unstack(fill_value=0)
print(housing_y_counts)

#Plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "housing", hue ="y")
plt.show()
```

```
y           no   yes
housing
no       15754  3147
yes      22418  1874
```



The plot above visualizes the distribution of clients across have housing loan group, no housing loan group (x) and whether they subscribed to the term deposit (y). It is evident that having housing loan group is the largest, with the highest number of clients (22418) who did not subscribe (no) compared to those who did (yes). This trend is consistent in the no having housing loan group as well, where the number of non-subscribers far exceeds that of subscribers.

Next, we calculate the proportion of "yes" and "no" responses within each housing loan status group. We do this by dividing the count of "yes" and "no" by the total number of people in these groups. By this way, we can observe what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each housing loan group. Stacked bars make it easy to compare the percentages across different groups.

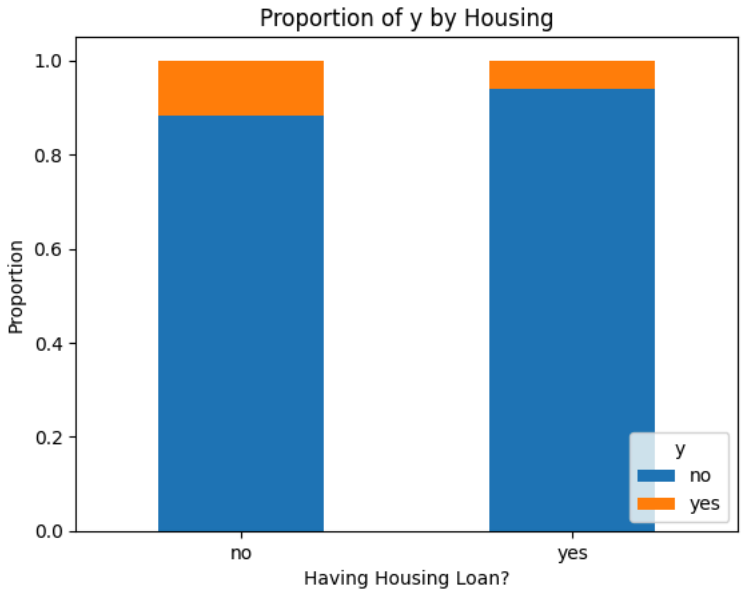
```
#Calculate the proportion of 'yes' and 'no' responses for each housing category.
housing_y_proportion = housing_y_counts.div(housing_y_counts.sum(axis=1), axis=0)

# Sort by 'yes' proportion in descending order
housing_y_proportion =housing_y_proportion.sort_values(by='yes', ascending=False)

print(housing_y_proportion)

# Plotting
default_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Having Housing Loan?')
plt.ylabel('Proportion')
plt.title('Proportion of y by Housing')
plt.legend(title='y', loc='lower right')
plt.xticks(rotation=0)
plt.show()
```

```
y             no       yes
housing
no       0.833501  0.166499
yes      0.922855  0.077145
```



To get the visualization, we examine the proportion of yes and no responses within each group in the second chart. The stacked bar chart reveals that the having housing loan group has a higher proportion of clients who did not subscribe. No having housing loan group are more likelihood of subscribing increases to the term deposit.These findings highlight the importance of considering the status of housing loan groups (yes or no) when predicting subscription behavior, as no having housing loan may represent a more receptive audience for marketing campaigns focused on term deposit.

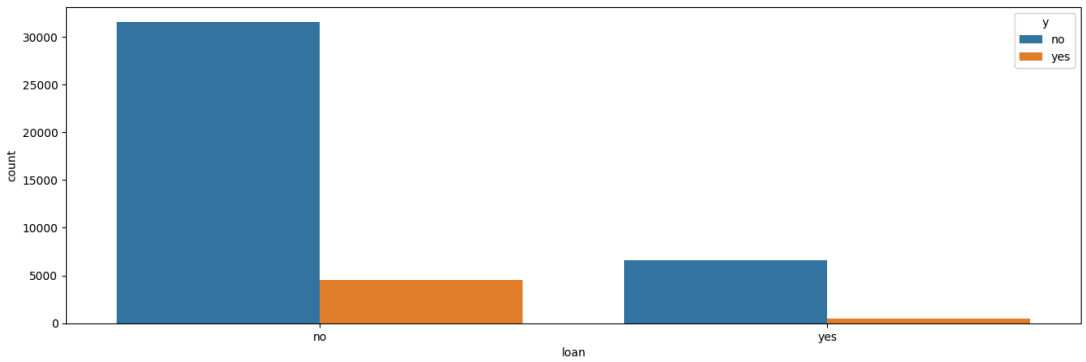
8. Relationship between target variable 'y' and feature variable 'loan'

At first, we count the number of people in each having personal loan status tended to say "Yes" or "No" to subscribing opening accounts. We do this by grouping the data by loan and y. The .size() function counts the number of people in each having personal loan group, and .unstack() helps us organize the results so that "yes" and "no" are shown in separate columns. To visualize, we create a bar plot to show the counts of people in each loan group who either subscribed (yes) or did not subscribe (no) to the term deposit. The hue parameter helps us to use different colors for "yes" and "no" responses within the same plot.

```
#Frequency Distribution
loan_y_counts = bank2.groupby(['loan', 'y']).size().unstack(fill_value=0)
print(loan_y_counts)

#Plotting
fig, ax = plt.subplots(figsize = (16,5))
g = sns.countplot(data = bank2, x = "loan", hue ="y")
plt.show()
```

```
y        no   yes
loan
no    31538  4548
yes    6634   473
```



The plot above visualizes the distribution of clients across having personal loan group, no personal loan group (x) and whether they subscribed to the term deposit (y). It is evident that no having personal loan group is the largest, with the highest number of clients (31538) who did not subscribe (no) compared to those who did (yes). This trend is consistent in the having personal loan group as well, where the number of non-subscribers far exceeds that of subscribers.

Next, we calculate the proportion of "yes" and "no" responses within each having personal loan status group. We do this by dividing the count of "yes" and "no" by the total number of people in these groups. By this way, we can observe what percentage of each group said "yes" or "no". And then, we create a stacked bar chart to visualize the proportion of "yes" and "no" responses within each personal loan status group. Stacked bars make it easy to compare the percentages across different groups.

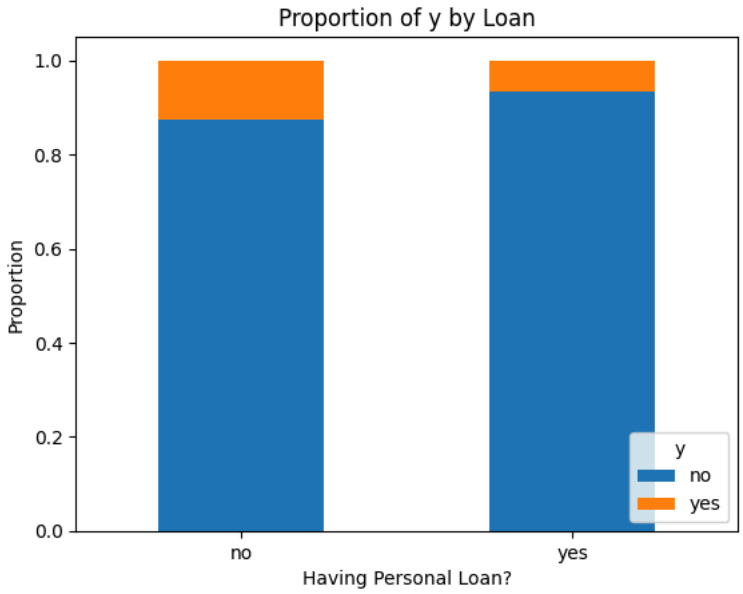
```
#Calculate the proportion of 'yes' and 'no' responses for each housing category.
loan_y_proportion = loan_y_counts.div(loan_y_counts.sum(axis=1), axis=0)

# Sort by 'yes' proportion in descending order
loan_y_proportion =loan_y_proportion.sort_values(by='yes', ascending=False)

print(loan_y_proportion)

# Plotting
loan_y_proportion.plot(kind='bar', stacked=True)
plt.xlabel('Having Personal Loan?')
plt.ylabel('Proportion')
plt.title('Proportion of y by Loan')
plt.legend(title='y', loc='lower right')
plt.xticks(rotation=0)
plt.show()
```

```
y           no       yes
loan
no    0.873968  0.126032
yes   0.933446  0.066554
```



To get the visualization, we examine the proportion of yes and no responses within each group in the second chart. The stacked bar chart reveals that the having personal loan group has a higher proportion of clients who did not subscribe. Clients in the group that is having no personal loan are more likelihood of subscribing increases to the term deposit. These findings highlight the importance of considering the status of having personal loan groups (yes or no) when predicting subscription behavior, as no having personal loan may represent a more receptive audience for marketing campaigns focused on term deposit.

9. Relationship between feature variable 'age_group', feature variable 'loan', and target 'y'

We wanted to explore how a client's age group, whether they have a personal loan, and their decision to subscribe to a term deposit are related. To do this, we divided clients into different age groups (18-30, 31-50, 51-70, 71-95), please refer to the relationship between target 'y' and features 'age'. Then, we grouped them based on their age group, whether they have a personal loan, and whether they subscribed to a term deposit (y). We counted how many clients fell into each group and created graphs to see the patterns.

```
# Data Frequencies
loan_age_y_counts = bank2.groupby(['age_group', 'loan', 'y']).size().unstack(fill_value=0)
print(loan_age_y_counts)

#Plotting
g = sns.FacetGrid(bank2, col="loan", height=5, aspect=1.5)
g.map(sns.countplot, "age_group", hue="y", data=bank2)
g.add_legend(title="y")  # Adds a title to the legend
g.set_axis_labels("Age_Group", "Count")  # Label the axes
plt.show()
```

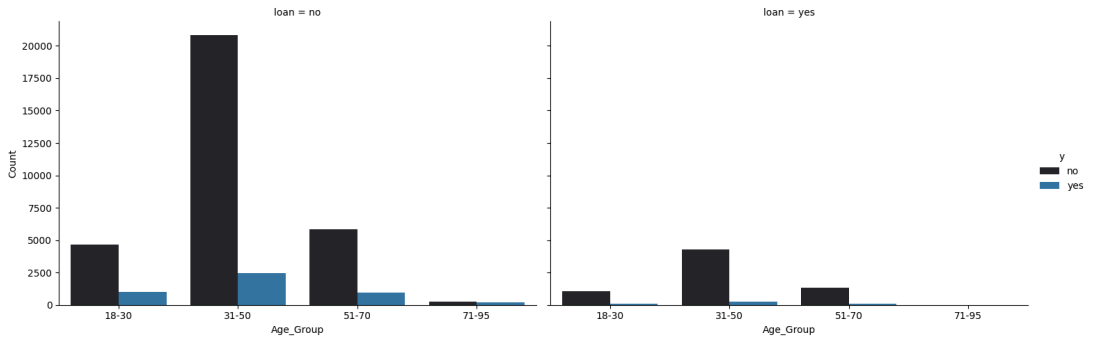
```
/tmp/ipykernel_389/1181786240.py:2: FutureWarning: The default of observed=False is deprecated and will be changed t
  loan_age_y_counts = bank2.groupby(['age_group', 'loan', 'y']).size().unstack(fill_value=0)
y                 no   yes
age_group loan
18-30     no     4635   984
          yes    1053   107
31-50     no    20843  2437
          yes    4268   276
51-70     no     5823   932
          yes    1312    89
71-95     no      237   195
          yes       1     1
/shared-libs/python3.9/py/lib/python3.9/site-packages/seaborn/axisgrid.py:718: UserWarning: Using the countplot func
  warnings.warn(warning)
/shared-libs/python3.9/py/lib/python3.9/site-packages/seaborn/axisgrid.py:854: FutureWarning:

Setting a gradient palette using color= is deprecated and will be removed in v0.14.0. Set `palette='dark:#1f77b4'` t

  func(*plot_args, **plot_kwargs)
/shared-libs/python3.9/py/lib/python3.9/site-packages/seaborn/axisgrid.py:854: FutureWarning:

Setting a gradient palette using color= is deprecated and will be removed in v0.14.0. Set `palette='dark:#1f77b4'` t

  func(*plot_args, **plot_kwargs)
```



The graphs are split into two sections: one for clients without a personal loan (on the left) and one for clients with a personal loan (on the right).

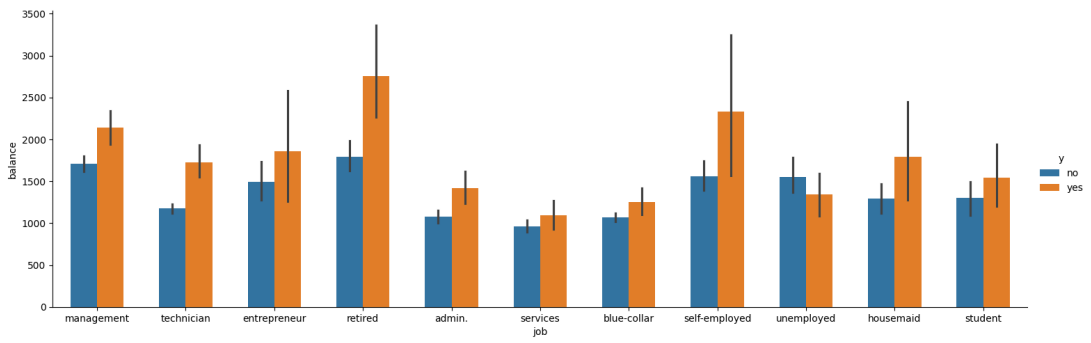
- Clients Without a Loan (loan = 0) : The largest group is clients aged 31-50, but most of them did not subscribe to the term deposit. This pattern is similar across all age groups, where more clients chose not to subscribe than to subscribe.
- Clients With a Loan (loan = 1) : The trend is similar here. The 31-50 age group is still the largest, and most of them also did not subscribe to the term deposit. However, compared to those without a loan, slightly more clients with a loan decided to subscribe.

From these graphs, we can see that most clients, especially those in the 31-50 age group, are not subscribing to a term deposit, whether they have a personal loan or not. However, having a loan seems to make a small difference, as slightly more people with loans are subscribing. This tells us that age and whether someone has a loan could be factors to consider when trying to understand who is likely to open a savings account.

10. Relationship between feature variable 'job', feature variable 'balance', and target 'y'

```
plt.figure(figsize=(16,5))
sns.catplot(x="job", y="balance", hue="y", kind="bar", data=bank2,  height=5, aspect=3, width=0.6, dodge=True
plt.show()
```

```
<Figure size 1600x500 with 0 Axes>
```



In general, all the job category except 'unemployed', people that say 'yes' has higher balance than people say 'no' (orange bar is higher than blue bar). 'Unemployed' is the exception because in this category, the blue bar (people who said "no") is higher than the orange bar, meaning that unemployed people who did not subscribe tend to have higher balances than those who did.

Also, Retired, and Self-Employed has a significant difference in balances, with people say 'yes' generally holding much higher balances than people say 'no'

Data Manipulation

```
bank3= bank2.copy()
```

1. Grouping/Combining

For marital Status, we apply np.where() to group ‘single’ and ‘divorce’ as 0, and ‘married’ as 1. Also, we convert all the binary value of 'yes' and 'no' from variable 'default', 'housing' , 'loan', and our target variable 'y' into 0, 1. These steps will support us later for our data analysis to calculate correlations. Please refer to coding below

```
bank3['marital_married'] = np.where(bank3['marital'] == 'married', 1, 0)
bank3['default'] = np.where(bank3['default'] == 'yes', 1, 0)
bank3['housing'] = np.where(bank3['housing'] == 'yes', 1, 0)
bank3['loan'] = np.where(bank3['loan'] == 'yes', 1, 0)
bank3['y'] = np.where(bank3['y'] == 'yes', 1, 0)
bank3.head()
```

	age int64	**job** object	**marital** object	**education** object	**default** int64	**balance** int64	**housin**
0	58	management	married	tertiary	0	2143	
1	44	technician	single	secondary	0	29	
2	33	entrepreneur	married	secondary	0	2	
5	35	management	married	tertiary	0	231	
6	28	management	single	tertiary	0	447	

5 rows, 12 cols, showing 10 rows/page Page 1 of 1

2. Dropping unnecessary variables

In the below code, we are dropping the columns 'age', 'balance_bin', and 'marital' from the bank3 DataFrame.

* The 'age' column is removed because it has been grouped into a new variable called 'age_group', which categorizes ages into distinct groups.
* The 'balance_bin' column was created solely for visualization purposes and is no longer needed for further analysis.
* As for the 'marital' column, the statuses of 'single' and 'divorced' have been combined into one category. To reflect this change, a new variable called 'marital_married' has been introduced, where the value is 1 if the individual is married and 0 if they are not.

After dropping these columns, we use the info() method to display the updated structure of the DataFrame, confirming the changes made.

```
bank3.drop(['age','balance_bin','marital'], axis='columns', inplace=True)
bank3.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 43193 entries, 0 to 45210
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype   
---  ------           --------------  -----   
 0   job              43193 non-null  object  
 1   education        43193 non-null  object  
 2   default          43193 non-null  int64   
 3   balance          43193 non-null  int64   
 4   housing          43193 non-null  int64   
 5   loan             43193 non-null  int64   
 6   y                43193 non-null  int64   
 7   age_group        43193 non-null  category
 8   marital_married  43193 non-null  int64   
dtypes: category(1), int64(6), object(2)
memory usage: 3.0+ MB
```

3. Creating Dummy Variables

Next, we proceed by creating dummy variables using the Pandas get_dummies method. This step involves converting categorical variables into a format that can be provided to ML algorithms to do a better job in prediction. To avoid redundancy and multicollinearity, we drop the first dummy variable for each categorical feature by setting drop_first=True. We apply the pd.get_dummies() function to the variables “job,” “education,” and “age_group” to generate their respective dummy variables. As shown below, after converting these categorical variables into dummy variables, the data type for these new columns is int64, where the values are either 0 or 1. A value of 1 indicates that the category is true for that observation, while 0 indicates that it is not true.

```
bank3=pd.get_dummies(bank3, columns=["job","education","age_group"], drop_first=True, dtype=np.int64)
bank3.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 43193 entries, 0 to 45210
Data columns (total 21 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   default              43193 non-null  int64
 1   balance              43193 non-null  int64
 2   housing              43193 non-null  int64
 3   loan                 43193 non-null  int64
 4   y                    43193 non-null  int64
 5   marital_married      43193 non-null  int64
 6   job_blue-collar      43193 non-null  int64
 7   job_entrepreneur     43193 non-null  int64
 8   job_housemaid        43193 non-null  int64
 9   job_management       43193 non-null  int64
 10  job_retired          43193 non-null  int64
 11  job_self-employed    43193 non-null  int64
 12  job_services         43193 non-null  int64
 13  job_student          43193 non-null  int64
 14  job_technician       43193 non-null  int64
 15  job_unemployed       43193 non-null  int64
 16  education_secondary  43193 non-null  int64
 17  education_tertiary   43193 non-null  int64
 18  age_group_31-50      43193 non-null  int64
 19  age_group_51-70      43193 non-null  int64
 20  age_group_71-95      43193 non-null  int64
dtypes: int64(21)
memory usage: 7.2 MB
```

```
bank3.head().transpose()
```

	0 int64 0 - 2143	**1** int64 0 - 29	**2** int64 0 - 2	**5** int64 0 - 231	**6** int64 0 - 447
default	0	0	0	0	0
balance	2143	29	2	231	447
housing	1	1	1	1	1
loan	0	0	1	0	1
y	0	0	0	0	0
marital_married	1	0	1	1	0
job_blue-collar	0	0	0	0	0
job_entrepreneur	0	0	1	0	0
job_housemaid	0	0	0	0	0
job_management	1	0	0	1	1
job_retired	0	0	0	0	0
job_self-employed	0	0	0	0	0
job_services	0	0	0	0	0
job_student	0	0	0	0	0
job_technician	0	1	0	0	0
job_unemployed	0	0	0	0	0
education_secondary	0	1	1	0	0
education_tertiary	1	0	0	1	1
age_group_31-50	0	1	1	1	0
age_group_51-70	1	0	0	0	0
age_group_71-95	0	0	0	0	0

21 rows, 5 cols, showing 100 rows/page Page 1 of 1

Data Analysis

1. Structure the Data

First, we're getting the data ready for our model. We create a list of all the features from the bank3 DataFrame, excluding the target variable y, which tells us whether a client subscribed to a term deposit or saying 'yes'. This way, we have all the important information in X that we'll use to make predictions. As shown below, The X DataFrame now has 20 columns, each containing numerical data, like the client's job, marital status, education, and more. Meanwhile, the y variable is our target, showing whether each client said yes (1) or no (0) to a term deposit. This setup is essential for training and testing our machine learning model.

```
features = [col for col in bank3.columns if col != 'y']
X = bank3[features]
y = bank3['y']
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 43193 entries, 0 to 45210
Data columns (total 20 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   default              43193 non-null  int64
 1   balance              43193 non-null  int64
 2   housing              43193 non-null  int64
 3   loan                 43193 non-null  int64
 4   marital_married      43193 non-null  int64
 5   job_blue-collar      43193 non-null  int64
 6   job_entrepreneur     43193 non-null  int64
 7   job_housemaid        43193 non-null  int64
 8   job_management       43193 non-null  int64
 9   job_retired          43193 non-null  int64
 10  job_self-employed    43193 non-null  int64
 11  job_services         43193 non-null  int64
 12  job_student          43193 non-null  int64
 13  job_technician       43193 non-null  int64
 14  job_unemployed       43193 non-null  int64
 15  education_secondary  43193 non-null  int64
 16  education_tertiary   43193 non-null  int64
 17  age_group_31-50      43193 non-null  int64
 18  age_group_51-70      43193 non-null  int64
 19  age_group_71-95      43193 non-null  int64
dtypes: int64(20)
memory usage: 6.9 MB
```

Next step, we applied a Min-Max scaling transformation to normalize the data, particularly focusing on the balance variable. This transformation scales all the values in the X DataFrame to a range between 0 and 1, which is important for ensuring that all variables are on the same scale, especially when using machine learning algorithms that are sensitive to the magnitude of the data.

We used the MinMaxScaler from sklearn.preprocessing to perform this transformation. After fitting the scaler to the data and transforming it, we reassigned the scaled values back to the X DataFrame. Now, all features, including balance, are scaled between 0 and 1.

The descriptive statistics for the balance column after scaling show that:

- The minimum value is 0
- The maximum value is 1.
- The mean value is approximately 0.085, indicating the average balance is near the lower end of the scale.
- The data is tightly clustered, as seen from the small standard deviation of about 0.027.

This normalization helps in preventing any single feature from dominating the model due to its scale and ensures that the machine learning model can process the data more effectively.

```
#MinMax transformation to get all data into a 0 to 1 range for variable "balance"
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
mm_scaler = preprocessing.MinMaxScaler()
X = mm_scaler.fit_transform(X)
X = pd.DataFrame(X, columns=features)
X.head()
```

	default float64	balance float64	housing float64	loan float64	marital_married f...	job_blue-collar fl...	job_en
0	0	0.09225936484	1	0	1	0	
1	0	0.07306665698	1	0	0	0	
2	0	0.07282152779	1	1	1	0	
3	0	0.07490058649	1	0	1	0	
4	0	0.07686162003	1	1	0	0	

5 rows, 20 cols, showing 10 rows/page Page 1 of 1

```
X['balance'].describe()
```

```
count    43193.000000
mean         0.085096
std          0.027619
min          0.000000
25%          0.073448
50%          0.076816
75%          0.085623
max          1.000000
Name: balance, dtype: float64
```

2. Explore the correlations with HeatMap

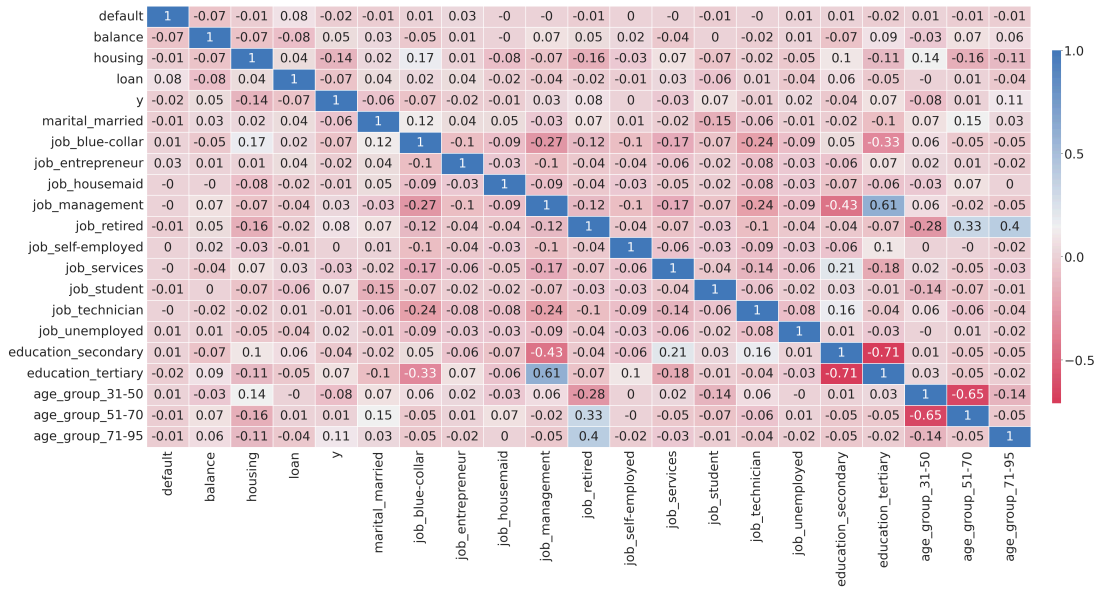
Next, we generated a HeatMap to examine the correlations between the variables. In the HeatMap, darker red shades indicate a stronger positive correlation between variables, while darker blue shades represent a stronger negative (or inverse) correlation.

```
sns.set(rc={"figure.figsize": (60, 25)})
heatmap = sns.heatmap(
    bank3.corr().round(2),
    linewidths=2.0,
    annot=True,
    annot_kws={"size": 40},  # Increase the size of the numbers in the heatmap
    cmap=sns.diverging_palette(5, 250, as_cmap=True),
    cbar_kws={"shrink": 0.8, "aspect": 40, "pad": 0.02, "ticks": [-1, -0.5, 0, 0.5, 1]}) # Adjust color bar s


# Increase the size of the x and y tick labels
plt.xticks(fontsize=40, rotation=90)
plt.yticks(fontsize=40)

# Modify the color bar tick labels size
colorbar = heatmap.collections[0].colorbar
colorbar.ax.tick_params(labelsize=40)  # Increase the font size of the color bar ticks

plt.show()
```



This heatmap shows how different features in the data relate to each other and to whether someone will subscribe to a term deposit (y). Most features don't have a strong connection with the target, meaning no single feature is a good predictor on its own. Some features, like different education levels and age groups, are strongly negatively related to each other (correl of -0.71 and -0.65), which makes sense since people typically fall into one specific category. Education_tetiary has high correlation of 0.61 with job_management which makes senses since people at manager level mostly have tertiary level. Also, people with age group of 51-70, and 71-95 has high correlation with job retired, which makes senses. However, features like balance and housing are mostly independent and provide unique information. Overall, this heatmap suggests that predicting whether someone will subscribe might require looking at how multiple features interact, rather than relying on any one feature alone.

3. Decision Tree Analysis

Build the decision tree with the sklearn class DecisionTreeClassifier. Instantiate the procedure,here as the object dt_model. Limit the depth of the obtained decision trees by setting theparameter max_depth to 14

```
from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(max_depth=14)
dt_model
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(max_depth=14)
```

Step 1 - Hyperparameter Tuning with Cross-Validation to find the optimal decision tree

In this step, we define a range of potential values for the max_depth parameter of our Decision Tree model. We create a dictionary called kparams where the key is 'max_depth', and the values are a list of depths: [2, 4, 6, 8, 10, 12, 14]. These values represent the different depths we will test to find the optimal tree depth that balances model accuracy and complexity. By evaluating the model's performance at each of these depths, we can identify the best max_depth that avoids overfitting while still capturing important patterns in the data.

```
kparams = {'max_depth':[2,4,6,8,10,12,14]}
```

For each of the 7 models (1 depth level= 1 model), do cross-validations on 3 different folds, defined by the n_splits parameterwith the KFold procedure.

kf3 is an instance of the KFold cross-validation method from sklearn.model_selection, set up to split the dataset into 3 folds (n_splits=3). In this setup, the data is divided into three parts, where in each iteration, two parts are used for training the model, and the remaining part is used for testing. This process is repeated three times, ensuring that every part of the data is used as a test set once. The shuffle=True parameter shuffles the data before splitting, ensuring a random distribution of data across the folds. Additionally, the random_state=1 ensures that the shuffling and splitting are reproducible, meaning the folds will be the same every time the code is executed. This approach provides a reliable assessment of the model's performance by training and testing it on different subsets of the data, helping to reduce the risk of overfitting and providing a better estimate of how the model will perform on unseen data

```
from sklearn.model_selection import KFold
kf3 = KFold(n_splits=3, shuffle=True, random_state=1)
```

Next, we perform Grid Search, which involves running 21 different analyses (7 models x 3splits) automatically using the GridSearchCV class. This process is called grid-search cross-validation, and it's set up as an object named grid_search. The param_grid parameter allows us to specify the different settings (hyper-parameters) we want to test. We use the kparams we defined earlier to systematically try all possible combinations of these settings. GridSearchCV sets up a grid of models based on these combinations and runs the specified cross-validations (three in this case) for each model using the kf3 setup we created before. This process helps us determine which combination of settings works best by evaluating multiple models and scoring them on accuracy, recall, and precision without refitting the models after each test.

```
from sklearn.model_selection import GridSearchCV
grid_search = GridSearchCV(dt_model, param_grid=kparams, cv=kf3,
    scoring=('accuracy', 'recall', 'precision'),
    refit=False, return_train_score=True)
grid_search.fit(X,y)
```

```
/shared-libs/python3.9/py/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1334: UndefinedMetricWarnir
  _warn_prf(average, modifier, msg_start, len(result))
/shared-libs/python3.9/py/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1334: UndefinedMetricWarnir
  _warn_prf(average, modifier, msg_start, len(result))
/shared-libs/python3.9/py/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1334: UndefinedMetricWarnir
  _warn_prf(average, modifier, msg_start, len(result))
/shared-libs/python3.9/py/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1334: UndefinedMetricWarnir
  _warn_prf(average, modifier, msg_start, len(result))
/shared-libs/python3.9/py/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1334: UndefinedMetricWarnir
  _warn_prf(average, modifier, msg_start, len(result))
/shared-libs/python3.9/py/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1334: UndefinedMetricWarnir
  _warn_prf(average, modifier, msg_start, len(result))
```

```
▸ GridSearchCV
▸ estimator: DecisionTreeClassifier
    ▸ DecisionTreeClassifier
```

Next, we are organizing and preparing the results from the Grid Search for further analysis.

First, we create a DataFrame called d_results from the cv_results_ attribute of the grid_search object. This DataFrame contains all the cross-validation results from the Grid Search, rounded to three decimal places for easier readability.

Next, we drop the 'params' column from the DataFrame because we're focusing on the performance metrics rather than the specific parameters used in each model.

Finally, we transpose the DataFrame with d_results.transpose() to make it easier to visualize and compare the different models and their corresponding metrics. This setup allows us to review the results of the Grid Search more effectively, helping us to identify the best-performing model configurations.

```
d_results = pd.DataFrame(grid_search.cv_results_).round(3)
d_results = d_results.drop(['params'], axis='columns')
d_results.transpose()
```

	0 object 0.0 39.5% 0.884 13.2% 14 others 47.4%	1 object 0.0 23.7% 0.001 15.8% 15 others 60.5%	2 object 0.007 10.5% 0.001 7.9% 27 others 81.6%	3 object 0.024 5.3% 0.882 5.3% 30 others 89.5%	4 object 0.89 7.9% 0.022 5.3% 31 others 86.8%	5 object 0.05 0.875 34 others
mean_fit_time	0.064	0.074	0.102	0.115	0.162	0.201
std_fit_time	0.031	0.025	0.003	0.004	0.029	0.06
mean_score_time	0.035	0.033	0.016	0.049	0.033	0.05
std_score_time	0.021	0.024	0.001	0.024	0.022	0.024
param_max_depth	2	4	6	8	10	12

38 rows, 7 cols, showing 5 rows/page Page 1 of 8

Based on the Grid Search results above, we are now summarizing the key results to better understand how the model performed at different depths.

First, we create a new DataFrame called d_summary by selecting specific columns from the d_results DataFrame. These columns include the maximum depth (param_max_depth), and the mean values of accuracy, recall, and precision for both the training and test sets. This selection helps us focus on the most important metrics that indicate how well the model is performing.

Next, we rename the columns to make them easier to understand, such as renaming param_max_depth to depth, and renaming the various mean_ columns to reflect whether they pertain to the training or test set (e.g., mean_test_accuracy becomes test_accuracy).

Finally, we calculate the difference between the training and test metrics to measure overfitting. We create three new columns in d_summary:

- accuracy_overfit: Measures the difference between training accuracy and test accuracy.
- precision_overfit: Measures the difference between training precision and test precision.
- recall_overfit: Measures the difference between training recall and test recall.

These overfitting metrics help us understand how much the model's performance on the training data differs from its performance on the test data. High differences might indicate that the model is overfitting to the training data and not generalizing well to new, unseen data. The d_summary DataFrame now provides a clear overview of the model's performance across different depths and helps in identifying the optimal model configuration.

```
d_summary = d_results[['param_max_depth', 'mean_test_accuracy',
                       'mean_test_recall', 'mean_test_precision', 'mean_train_accuracy',
                       'mean_train_recall', 'mean_train_precision']]

d_summary = d_summary.rename(columns={
    'param_max_depth': 'depth',
    'mean_test_accuracy': 'test_accuracy',
    'mean_test_recall': 'test_recall',
    'mean_test_precision': 'test_precision',
    'mean_train_accuracy': 'train_accuracy',
    'mean_train_recall': 'train_recall',
    'mean_train_precision': 'train_precision'})

d_summary['accuracy_overfit']= d_summary['train_accuracy'] - d_summary['test_accuracy']
d_summary['precision_overfit']= d_summary['train_precision'] - d_summary['test_precision']
d_summary['recall_overfit']= d_summary['train_recall'] - d_summary['test_recall']
d_summary
```

	depth	test_ac...	test_re...	test_pr...	train_ac...	train_re...	train_pr...	accuracy_overfit	precision_ove...	recall_overfit
0	2	0.884	0	0	0.884	0.001	0.286	0	0.286	0.001
1	4	0.884	0	0.167	0.884	0.001	0.667	0	0.5	0.001
2	6	0.883	0.007	0.354	0.885	0.015	0.787	0.002	0.433	0.008
3	8	0.882	0.017	0.349	0.887	0.035	0.758	0.005	0.409	0.018
4	10	0.88	0.026	0.297	0.89	0.072	0.826	0.01	0.529	0.046
5	12	0.876	0.045	0.295	0.896	0.128	0.865	0.02	0.57	0.083
6	14	0.87	0.077	0.286	0.905	0.218	0.86	0.035	0.574	0.141

7 rows, 10 cols, showing 50 rows/page Page 1 of 1

We chose a depth of 4 for our Decision Tree model because it provides a good balance between performance and overfitting. At depth 4, the test accuracy remains high at 0.884, which is the same as at depth 2, but with better precision (0.167 compared to 0.0). This means the model makes fewer false positive predictions at this depth.

Moreover, the overfitting indicators (accuracy, precision, and recall overfit) are still very low at depth 4, meaning the model is not overly tailored to the training data and should generalize well to new data. If we were to go deeper than 4, the test accuracy starts to slightly decrease, and the model begins to overfit.

Step 2 - Construct the tree

Do one estimation of the decision tree for the model with a depth of 4 and with the chosen parameters on all the data with the fit()function. Here, save the results into the object named dt_fit to use later for visualizing the tree.

```
dt_model = DecisionTreeClassifier(max_depth=4,class_weight='balanced')
dt_fit = dt_model.fit(X,y)
```

Step 3 - Feature Importance

In this step, we are analyzing the importance of each feature in the Decision Tree model.

First, we extract the feature importance scores from the trained model using the feature_importances_ attribute and store them in a DataFrame called dImp. Each feature's importance score indicates how much it contributes to the model's decisions.

Next, we set the index of dImp to match the columns in our X DataFrame (which contains the features), ensuring that each feature's importance score is clearly associated with its corresponding feature.

We then rename the column in dImp to 'Importance' for clarity. Finally, we round the importance scores to three decimal places to make them easier to read.

This process allows us to identify which features are most influential in making predictions, helping us understand the model's behavior and potentially refine it further.

```
dImp = pd.DataFrame(dt_model.feature_importances_)
dImp = dImp.set_index(X.columns, drop=False)
dImp.columns = ['Importance']
dImp.round(3)
```

	Importance float... 0.0 - 0.504
default	0
balance	0.229
housing	0.504
loan	0.097
marital_married	0.057
job_blue-collar	0
job_entrepreneur	0
job_housemaid	0
job_management	0
job_retired	0.009
job_self-employed	0
job_services	0
job_student	0.019
job_technician	0
job_unemployed	0
education_secondary	0
education_tertiary	0.029
age_group_31-50	0.056
age_group_51-70	0
age_group_71-95	0

20 rows, 1 col, showing 25 rows/page Page 1 of 1

The feature importance results show which factors are most influential in the Decision Tree model's predictions:

- Housing: The most important feature with a score of 0.504. This indicates that whether or not a client has a housing loan plays a significant role in predicting the outcome.
- Balance: The second most important feature with a score of 0.229. The amount of balance a client has also heavily influences the model's decisions.
- Loan: With a score of 0.097, this feature is also somewhat important, suggesting that having a personal loan affects the prediction.
- Other Features: Features like marital_married, age_group_31-50, education_tertiary, and job_student have very low importance scores, indicating they have minimal influence on the model's predictions.
- Zero Importance: Several features, such as job_blue-collar, job_management, and education_secondary, have an importance score of 0, meaning they don't contribute to the model's decision-making process.

Step 4 - Model Fit

In this step, we used the trained Decision Tree model to make predictions on the dataset (X). The model's predictions are stored in the variable y_fit. The code then prints the first 20 predictions using y_fit[0:19].

The output shows an array of predicted values, where 0 and 1 represent the two possible classes (0 for no and 1 for yes). In this case, most predictions are 0, with only a few 1s scattered throughout. This suggests that the model is predicting that most clients will not subscribe to a term deposit, but there are some instances where it predicts a positive outcome.

```
y_fit = dt_model.predict(X)
y_fit[0:19]
```

```
array([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Step 5 - Visualize the tree

One way we can visualize the tree is with text, using the sklearn procedure export_text.

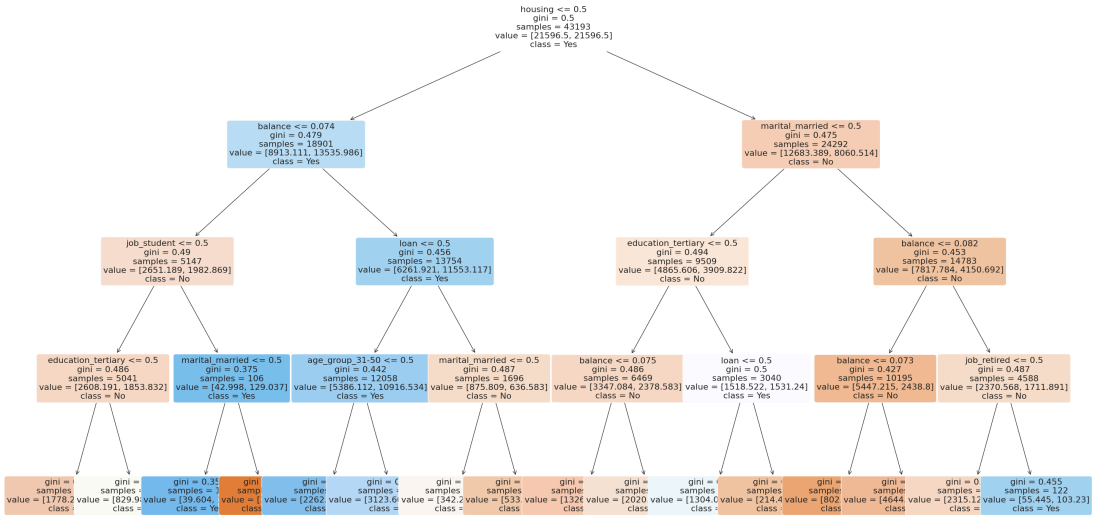
```
from sklearn.tree import export_text
text = export_text(dt_fit, feature_names=features)
print(text)
```

```
|--- housing <= 0.50
|   |--- balance <= 0.07
|   |   |--- job_student <= 0.50
|   |   |   |--- education_tertiary <= 0.50
|   |   |   |   |--- class: 0
|   |   |   |--- education_tertiary >  0.50
|   |   |   |   |--- class: 0
|   |   |--- job_student >  0.50
|   |   |   |--- marital_married <= 0.50
|   |   |   |   |--- class: 1
|   |   |   |--- marital_married >  0.50
|   |   |   |   |--- class: 0
|   |--- balance >  0.07
|   |   |--- loan <= 0.50
|   |   |   |--- age_group_31-50 <= 0.50
|   |   |   |   |--- class: 1
|   |   |   |--- age_group_31-50 >  0.50
|   |   |   |   |--- class: 1
|   |   |--- loan >  0.50
|   |   |   |--- marital_married <= 0.50
|   |   |   |   |--- class: 0
|   |   |   |--- marital_married >  0.50
|   |   |   |   |--- class: 0
|--- housing >  0.50
|   |--- marital_married <= 0.50
|   |   |--- education_tertiary <= 0.50
|   |   |   |--- balance <= 0.07
|   |   |   |   |--- class: 0
|   |   |   |--- balance >  0.07
|   |   |   |   |--- class: 0
```

Plot is another way that we are visualizing the Decision Tree model that we trained. The tree.plot_tree function from sklearn is used to create a visual representation of the decision tree.

```
from sklearn import tree
plt.figure(figsize=(35,20))
tree.plot_tree(dt_fit,
               feature_names=features,
               class_names=['No', 'Yes'],
               rounded=True,
               filled=True,
               fontsize=16)
```

```
[Text(0.5, 0.9, 'housing <= 0.5\ngini = 0.5\nsamples = 43193\nvalue = [21596.5, 21596.5]\nclass = Yes'),
 Text(0.25, 0.7, 'balance <= 0.074\ngini = 0.479\nsamples = 18901\nvalue = [8913.111, 13535.986]\nclass = Yes'),
 Text(0.125, 0.5, 'job_student <= 0.5\ngini = 0.49\nsamples = 5147\nvalue = [2651.189, 1982.869]\nclass = No'),
 Text(0.0625, 0.3, 'education_tertiary <= 0.5\ngini = 0.486\nsamples = 5041\nvalue = [2608.191, 1853.832]\nclass = N
 Text(0.03125, 0.1, 'gini = 0.468\nsamples = 3389\nvalue = [1778.209, 1058.104]\nclass = No'),
 Text(0.09375, 0.1, 'gini = 0.5\nsamples = 1652\nvalue = [829.982, 795.728]\nclass = No'),
 Text(0.1875, 0.3, 'marital_married <= 0.5\ngini = 0.375\nsamples = 106\nvalue = [42.998, 129.037]\nclass = Yes'),
 Text(0.15625, 0.1, 'gini = 0.359\nsamples = 100\nvalue = [39.604, 129.037]\nclass = Yes'),
 Text(0.21875, 0.1, 'gini = 0.0\nsamples = 6\nvalue = [3.395, 0.0]\nclass = No'),
 Text(0.375, 0.5, 'loan <= 0.5\ngini = 0.456\nsamples = 13754\nvalue = [6261.921, 11553.117]\nclass = Yes'),
 Text(0.3125, 0.3, 'age_group_31-50 <= 0.5\ngini = 0.442\nsamples = 12058\nvalue = [5386.112, 10916.534]\nclass = Ye
 Text(0.28125, 0.1, 'gini = 0.397\nsamples = 5398\nvalue = [2262.507, 6017.428]\nclass = Yes'),
 Text(0.34375, 0.1, 'gini = 0.476\nsamples = 6660\nvalue = [3123.606, 4899.106]\nclass = Yes'),
 Text(0.4375, 0.3, 'marital_married <= 0.5\ngini = 0.487\nsamples = 1696\nvalue = [875.809, 636.583]\nclass = No'),
 Text(0.40625, 0.1, 'gini = 0.5\nsamples = 680\nvalue = [342.29, 322.593]\nclass = No'),
 Text(0.46875, 0.1, 'gini = 0.466\nsamples = 1016\nvalue = [533.519, 313.99]\nclass = No'),
 Text(0.75, 0.7, 'marital_married <= 0.5\ngini = 0.475\nsamples = 24292\nvalue = [12683.389, 8060.514]\nclass = No')
 Text(0.625, 0.5, 'education_tertiary <= 0.5\ngini = 0.494\nsamples = 9509\nvalue = [4865.606, 3909.822]\nclass = Nc
 Text(0.5625, 0.3, 'balance <= 0.075\ngini = 0.486\nsamples = 6469\nvalue = [3347.084, 2378.583]\nclass = No'),
 Text(0.53125, 0.1, 'gini = 0.468\nsamples = 2529\nvalue = [1326.726, 791.427]\nclass = No'),
 Text(0.59375, 0.1, 'gini = 0.493\nsamples = 3940\nvalue = [2020.358, 1587.156]\nclass = No'),
 Text(0.6875, 0.3, 'loan <= 0.5\ngini = 0.5\nsamples = 3040\nvalue = [1518.522, 1531.24]\nclass = Yes'),
 Text(0.65625, 0.1, 'gini = 0.499\nsamples = 2635\nvalue = [1304.095, 1419.407]\nclass = Yes'),
 Text(0.71875, 0.1, 'gini = 0.451\nsamples = 405\nvalue = [214.426, 111.832]\nclass = No'),
 Text(0.875, 0.5, 'balance <= 0.082\ngini = 0.453\nsamples = 14783\nvalue = [7817.784, 4150.692]\nclass = No'),
 Text(0.8125, 0.3, 'balance <= 0.073\ngini = 0.427\nsamples = 10195\nvalue = [5447.215, 2438.8]\nclass = No'),
 Text(0.78125, 0.1, 'gini = 0.355\nsamples = 1474\nvalue = [802.259, 240.869]\nclass = No'),
 Text(0.84375, 0.1, 'gini = 0.436\nsamples = 8721\nvalue = [4644.956, 2197.931]\nclass = No'),
 Text(0.9375, 0.3, 'job_retired <= 0.5\ngini = 0.487\nsamples = 4588\nvalue = [2370.568, 1711.891]\nclass = No'),
 Text(0.90625, 0.1, 'gini = 0.484\nsamples = 4466\nvalue = [2315.123, 1608.662]\nclass = No'),
 Text(0.96875, 0.1, 'gini = 0.455\nsamples = 122\nvalue = [55.445, 103.23]\nclass = Yes')]
```



Best Decision Rules for Predicting "Yes":

Rule path
* housing <= 0.5: The client does not have a housing loan.
* balance > 0.074: The client's balance is greater than 0.074.
* loan <= 0.5: The client does not have another loan.
* age_group_31-50 <= 0.5: The client's age group is not within 31-50 years.

Interpretation:
This rule indicates that clients who do not have a housing loan, have a balance >0.07, do not have another loan, and are outside the 31-50 age group are also likely to subscribe to a term deposit. This rule highlights the importance of financial independence (no loans) and financial capability (higher balance) in predicting term deposit subscription. Also, this path predicted as "yes" at all of the nodes.

Another strong rule path
* housing <= 0.5: The client does not have a housing loan.
* balance <= 0.074: The client's balance is less than 0.074.
* job_student > 0.5: The client is student
* marital married <= 0.5: The client is not married

Interpretation: This rule suggests that a student who is not married, does not have a housing loan, and has a relatively low balance is predicted to subscribe to a term deposit. Despite their limited financial resources, the model likely sees these clients as being in a life stage where they are beginning to build their financial future, and a term deposit is an attractive, low-risk option for them to start saving.

Step 6 - Evaluating the Model

```
from sklearn.metrics import confusion_matrix
pd.DataFrame(confusion_matrix(y, y_fit))
```

	0 int64	1 int64
0	26179	11993
1	2099	2922

2 rows, 2 cols, showing 10 rows/page Page 1 of 1

True Negatives (26179): The model correctly predicted 26,179 clients who did not subscribe to a term deposit ("No").
This indicates that the model is effective at identifying clients who are unlikely to subscribe.

False Positives (11993): The model incorrectly predicted that 11,993 clients would subscribe to a term deposit ("Yes"), but they did not. This high number of false positives suggests the model may be over-predicting "Yes" cases, identifying many clients as likely subscribers when they are not.

False Negatives (2099): The model incorrectly predicted that 2,099 clients would not subscribe ("No"), but they actually did.
This relatively lower number of false negatives indicates that the model is somewhat conservative, missing some true "Yes" predictions, but the number isn't as large as the false positives.

True Positives (2922): The model correctly predicted 2,922 clients who subscribed to a term deposit ("Yes"). This shows the model has some capability to correctly identify subscribers

```
from sklearn.metrics import accuracy_score, recall_score, precision_score,f1_score
print ('Accuracy: %.3f' % accuracy_score(y, y_fit))
print ('Recall: %.3f' % recall_score(y, y_fit))
print ('Precision: %.3f' % precision_score(y, y_fit))
print ('F1: %.3f' % f1_score(y, y_fit))
```

```
Accuracy: 0.674
Recall: 0.582
Precision: 0.196
F1: 0.293
```

1. Accuracy: 0.674 - The model gets the prediction right 67.4% of the time, but this number can be misleading because it treats all correct predictions equally, even if one class (like "No") is much more common.

2. Recall: 0.582 - Out of all the people who actually said "Yes" to subscribing, the model correctly identified 58.2% of them. This shows how good the model is at finding those who will subscribe.

3. Precision: 0.196- When the model predicts someone will subscribe, it's only right 19.6% of the time. This low number means the model often thinks people will subscribe when they actually won't.

4. F1 Score: 0.293-The F1 score is a combination of precision and recall. A low score of 0.293 indicates the model struggles to find a good balance between correctly identifying subscribers and not falsely predicting subscriptions.

In Summary, based on the current results, our model isn't reliable enough to use for making decisions. It often predicts that people will subscribe to a term deposit when they actually won't, and it's missing a significant number of actual subscribers. Because of this, the model might lead us to target the wrong people, wasting time and resources. Even though the model needs further improvement to be trustworthy, we won't go further to improve it in this assignment. Instead, we'll focus on the insights gained from the decision tree to draw some conclusions about the business question.

4. Analysis Results

Based on the entire analysis, including the correlations HeatMap and decision tree, the key takeaway is that a person's interest in opening a savings account is influenced by a combination of socio-economic factors rather than any single factor.

The decision tree suggests that factors such as having no loans, maintaining a higher balance, being a student, and certain age groups (like retirees) play a role in predicting whether someone will open a savings account. The heatmap further supports this by showing that while no individual factor strongly correlates with account opening on its own, certain factors like education level, job type, and age group have complex relationships with each other

Based on the entire analysis, the most significant insight is that the best predictors for someone likely to say "Yes" to opening a savings account involve a combination of the following factors:

- No Housing Loan/Persnal Loan: People without a housing loan and personal loan are more likely to be interested in opening a savings account, possibly because they have fewer financial obligations.
- Higher Balance: A higher balance indicates financial stability, making these individuals more likely to open a savings account.
- Specific Job:
  - Students: Students are more likely to open a savings account as they start planning for their financial future.
  - Retirees: Retired individuals, who often focus on preserving and managing their savings, are also more likely to say "Yes."
- Marital Status: People who are not married might be more likely to open a savings account, perhaps because they have fewer family-related financial commitments and are focused on their personal financial goals.

So, the best combination of factors for predicting who will open a savings account includes having no loans, maintaining a higher balance, being in a specific life stage (like a student or retiree), and not being married. This mix of financial and personal factors provides a strong indication that a person is likely to say "Yes" to opening a savings account.

Conclusion and Recommendation on actionable strategies for OnPoint

Based on the analysis, Onpoint should focus on targeting individuals who are financially stable and have fewer financial obligations. Specifically, the following strategies are recommended:

1. Target Individuals Without Loans

- Action/Strategy: Market savings accounts to those without housing loans, emphasizing financial security and stability

2. Focus on Clients with Higher Balances

- Action/Strategy: Offer special savings incentives to clients with higher balances, such as better interest rates.

3. Engage Specific Life Stages

- Students: Partner with colleges to offer student-focused savings accounts with low minimums.
- Retirees: Promote savings accounts to retirees highlighting stability and easy access to funds.

4. Consider Marital Status

- Action/Strategy: Tailor messaging to single individuals, focusing on personal financial growth and independence.

5. Leverage Digital Channels

- Action/Strategy: Utilize social media, email, and apps to reach targeted customers with personalized offers.

In conclusion, the analysis reveals that a person's likelihood of opening a savings account is influenced by several key factors, including their loans status, financial stability, job type, and marital status. Onpoint should focus on these insights to guide its marketing and product development strategies.

Targeting individuals without housing/personal loans, those with higher balances, students, retirees, and non-married individuals will likely yield the most success. By offering tailored products and personalized messaging that resonates with these groups, Onpoint can effectively attract new clients and encourage them to open savings accounts. This strategic focus will help Onpoint align its services with the needs of its customers, ultimately driving growth and strengthening client relationships.