NAVRACHANA UNIVERSITY

a UGC recognized University

BANK MARKETING ANALYSIS

PROJECT

OF

MACHINE LEARNING

(SPRING 2022)

PREPARED BY: **SOHIL VHORA**
20167007
BSc. Data Science

MENTOR: **DR.SALMA PIRZADA**

Bank Marketing Dataset

This dataset refers to the problem of telemarketing for a bank. The dataset is collected from a Portuguese bank and the bank wants to have an effective telemarketing strategy to sell long-term deposit accounts (e.g., bonds, saving accounts, etc.). These marketing campaigns were based on phone calls and multiple contacts were often needed to determine whether a customer would subscribe to a long-term deposit account. Your team of data scientists will help this bank in determining such customers and devising an effective telemarketing strategy by applying data analytics method on the given dataset.

1. Age: Age of the customer (numeric).
2. Job: Type of job (qualitative).
3. Marital: Marital status (qualitative).
4. Education: Education of the customer (qualitative).
5. Default: Shows whether the customer has credit in default or not (qualitative).
6. Balance: Average yearly balance in Euros (numeric).
7. Housing: Shows whether the customer has housing loan or not (qualitative).
8. Loan: Shows whether the customer has personal loan or not (qualitative/categorical).
9. Contact: Shows how the last contact for marketing campaign has been made (qualitative)
10. Day: Shows on which day of the month last time customer was contacted (numeric).
11. Month: Shows on which month of the year last time customer was contacted (qualitative).
12. Duration: Shows the last contact duration in seconds (numeric).
13. Campaign: Number of contacts performed during the marketing campaign and for this customer (numeric).
14. Pdays: Number of days that passed by after the client was last contacted from a previous campaign (numeric, -1 means client was not previously contacted).
15. Previous: Number of contacts performed before this campaign and for this client (numeric).
16. Poutcome: Outcome of the previous marketing campaign (qualitative).
17. Y – Class attribute showing whether the client has subscribed a term deposit or not (binary: "yes","no")

IMPORTING LIBRARIES

In [2]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import seaborn as sns
```

LOADING, DISPLAYING AND PLOTTING DATA

LOADING DATA

Below are the list of column we have in our data:

1. Age: Age of the customer (numeric).
2. Job: Type of job (qualitative).
3. Marital: Marital status (qualitative).
4. Education: Education of the customer (qualitative).
5. Default: Shows whether the customer has credit in default or not (qualitative).
6. Balance: Average yearly balance in Euros (numeric).
7. Housing: Shows whether the customer has housing loan or not (qualitative).
8. Loan: Shows whether the customer has personal loan or not (qualitative/categorical).
9. Contact: Shows how the last contact for marketing campaign has been made (qualitative)
10. Day: Shows on which day of the month last time customer was contacted (numeric).
11. Month: Shows on which month of the year last time customer was contacted (qualitative).
12. Duration: Shows the last contact duration in seconds (numeric).
13. Campaign: Number of contacts performed during the marketing campaign and for this customer (numeric).
14. Pdays: Number of days that passed by after the client was last contacted from a previous campaign (numeric, -1 means client was not previously contacted).
15. Previous: Number of contacts performed before this campaign and for this client (numeric).
16. Poutcome: Outcome of the previous marketing campaign (qualitative).
17. Y – Class attribute showing whether the client has subscribed a term deposit or not (binary: "yes","no")

In [3]:

```
# Load dataset
df_bank = pd.read_csv('bank.csv')

# Drop 'duration' column
df_bank = df_bank.drop('duration', axis=1)

# print(df_bank.info())
print('Shape of dataframe:', df_bank.shape)
df_bank.head()
```

```
Shape of dataframe: (4521, 16)
```

Out[3]:

	age	job	marital	education	default	balance	housing	loan	contact	day	mo
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	n
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	n

VIEW DATASET DETAILS

In [4]:

```
#SHAPE
df_bank.shape
```

Out[4]:

```
(4521, 16)
```

We have 4521 rows and 16 columns in our banking dataset.

DATA TYPES OF COLUMNS

In [5]:

```
df_bank.dtypes
```

Out[5]:

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

We can see, some columns are object types, that we have to convert them into numerical data. Before that let us visualize our dataset.

VISUALIZE DATASET

Let us see the count of each type of job.

In [6]:

```
job_count = df_bank['job'].value_counts()
job_count
```

Out[6]:

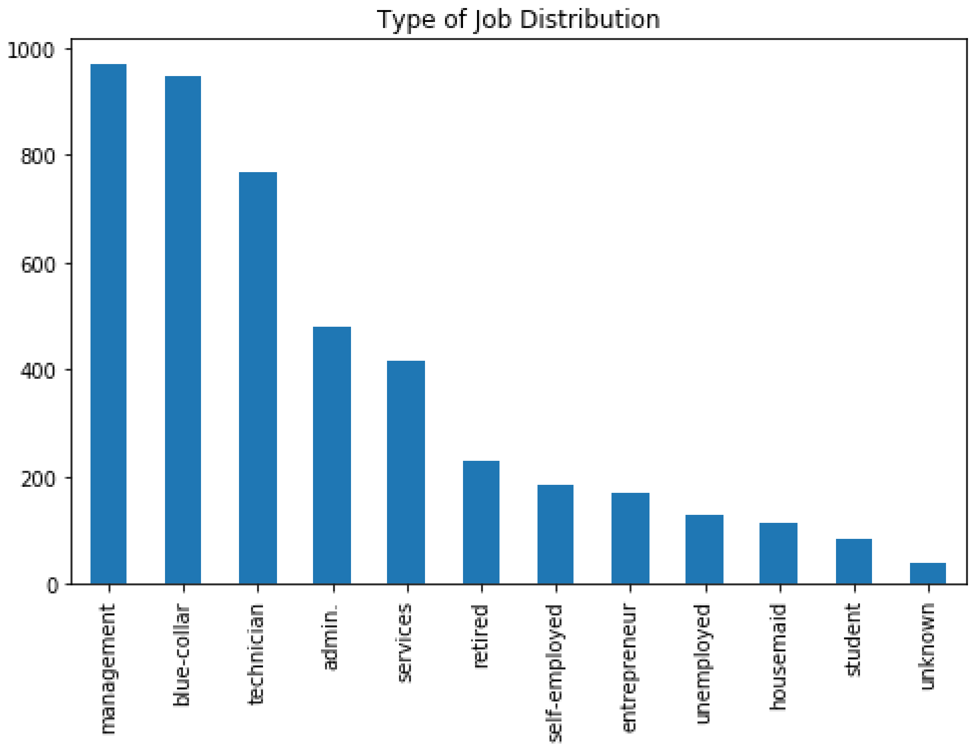
```
management       969
blue-collar      946
technician       768
admin.           478
services         417
retired          230
self-employed    183
entrepreneur     168
unemployed       128
housemaid        112
student           84
unknown           38
Name: job, dtype: int64
```

In [7]:

```
plt.figure(figsize = (8, 5))
job_count.plot(kind = "bar")
plt.title("Type of Job Distribution")
```

Out[7]:

```
Text(0.5, 1.0, 'Type of Job Distribution')
```



PLOT DEFAULT COLUMN

Column default says that client has credit in default or not. It has categorical value: ‘no’,’yes’,’unknown’.

In [8]:

```
default_count = df_bank['default'].value_counts()
default_count
```

Out[8]:

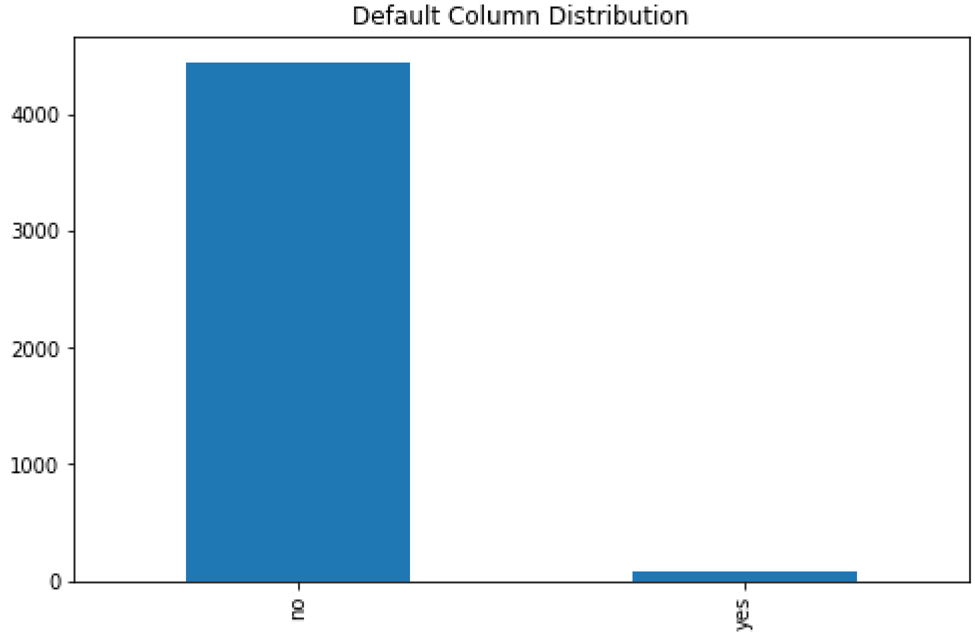
```
no     4445
yes      76
Name: default, dtype: int64
```

In [9]:

```
plt.figure(figsize = (8, 5))
default_count.plot(kind='bar').set(title='Default Column Distribution')
```

Out[9]:

```
[Text(0.5, 1.0, 'Default Column Distribution')]
```



PLOT MARITAL STATUS

In [10]:

```
marital_count = df_bank['marital'].value_counts()
marital_count
```

Out[10]:

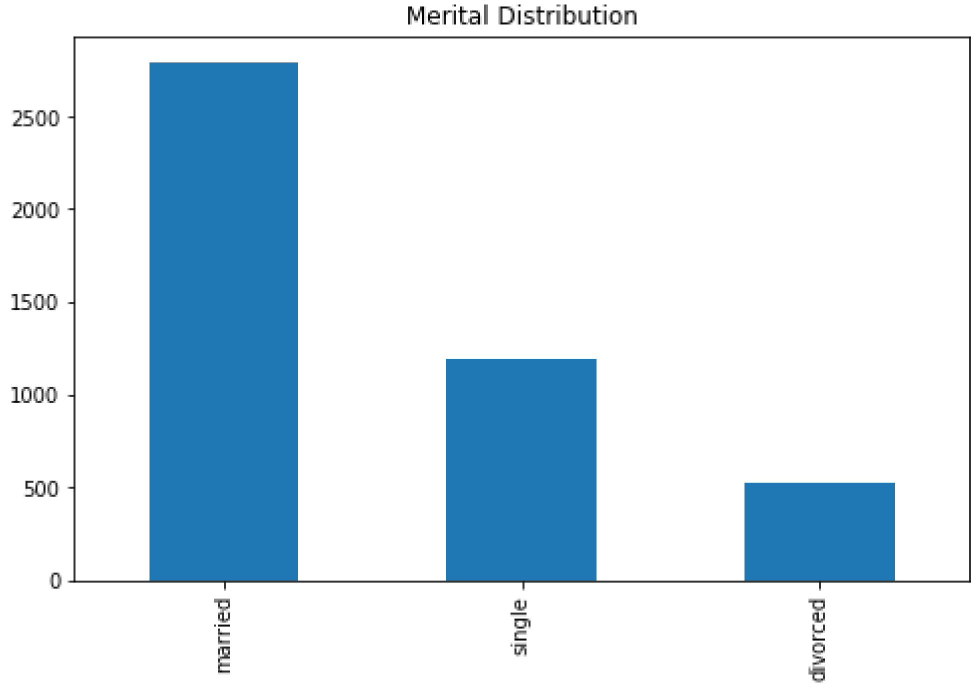
```
married     2797
single      1196
divorced     528
Name: marital, dtype: int64
```

In [11]:

```
plt.figure(figsize = (8, 5))
marital_count.plot(kind = "bar").set(title = "Merital Distribution")
```

Out[11]:

```
[Text(0.5, 1.0, 'Merital Distribution')]
```



PLOT THAT CUSTOMER HAS PERSONAL LOAN OR NOT

In [12]:

```
loan_count = df_bank['loan'].value_counts()
loan_count
```

Out[12]:

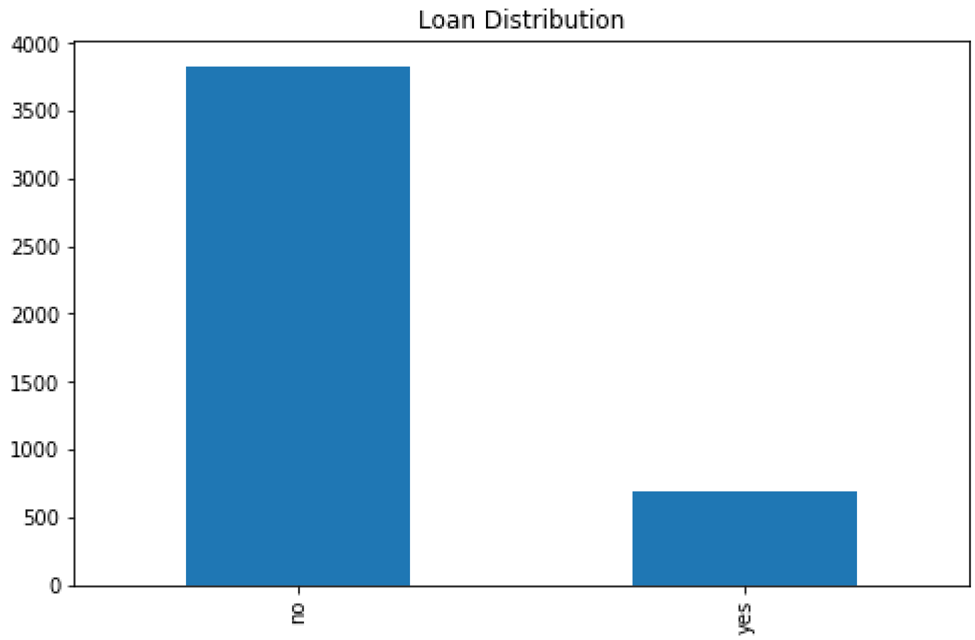
```
no     3830
yes     691
Name: loan, dtype: int64
```

In [13]:

```
plt.figure(figsize = (8, 5))
loan_count.plot(kind = "bar").set(title = "Loan Distribution")
```

Out[13]:

```
[Text(0.5, 1.0, 'Loan Distribution')]
```



As per data, some client has taken the personal loan.

PLOT THAT CLIENT HAS HOUSING LOAN OR NOT

In [14]:

```
housing_count = df_bank['housing'].value_counts()
housing_count
```

Out[14]:

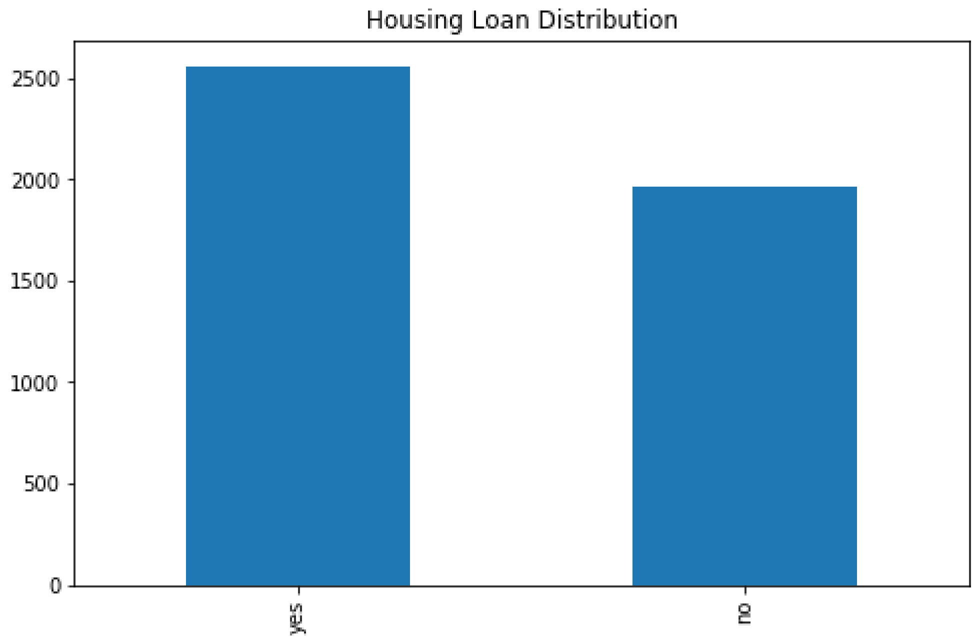
```
yes    2559
no     1962
Name: housing, dtype: int64
```

In [15]:

```
plt.figure(figsize = (8, 5))
housing_count.plot(kind = "bar").set(title = "Housing Loan Distribution")
```

Out[15]:

```
[Text(0.5, 1.0, 'Housing Loan Distribution')]
```



Most of the client has taken the housing loan.

PLOT EDUCATION COLUMN

In [16]:

```
education_count = df_bank['education'].value_counts()
education_count
```

Out[16]:

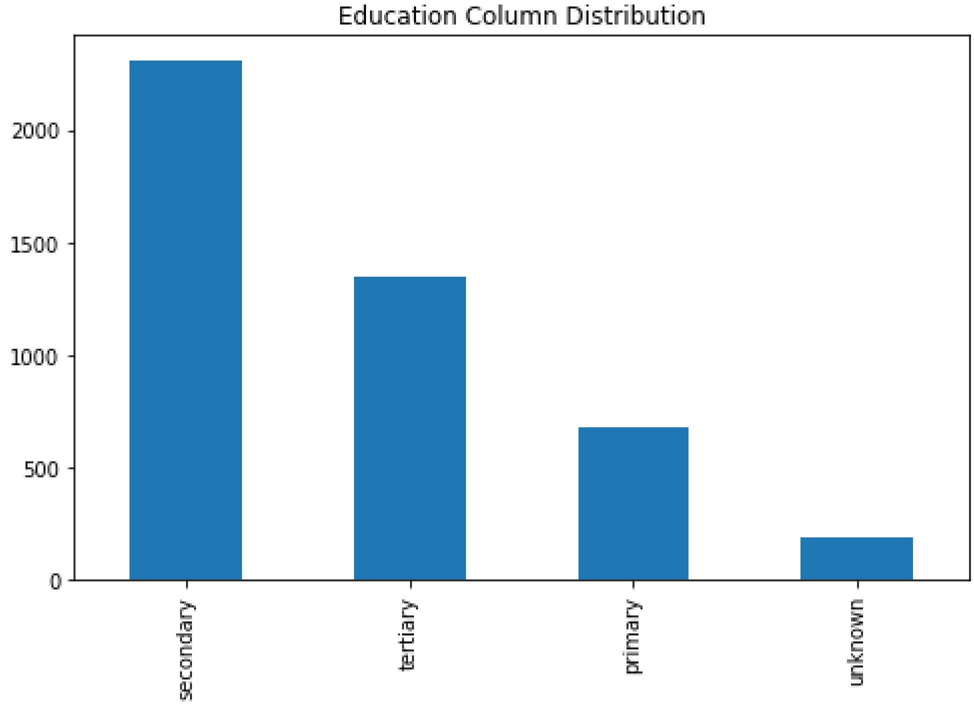
```
secondary    2306
tertiary     1350
primary       678
unknown       187
Name: education, dtype: int64
```

In [17]:

```
plt.figure(figsize = (8, 5))
education_count.plot(kind = "bar").set(title = "Education Column Distribution")
```

Out[17]:

```
[Text(0.5, 1.0, 'Education Column Distribution')]
```



PLOT CONTACT COLUMN

Contact column says client were contacted by cellular or telephone.

In [18]:

```
contact_count = df_bank['contact'].value_counts()
contact_count
```

Out[18]:

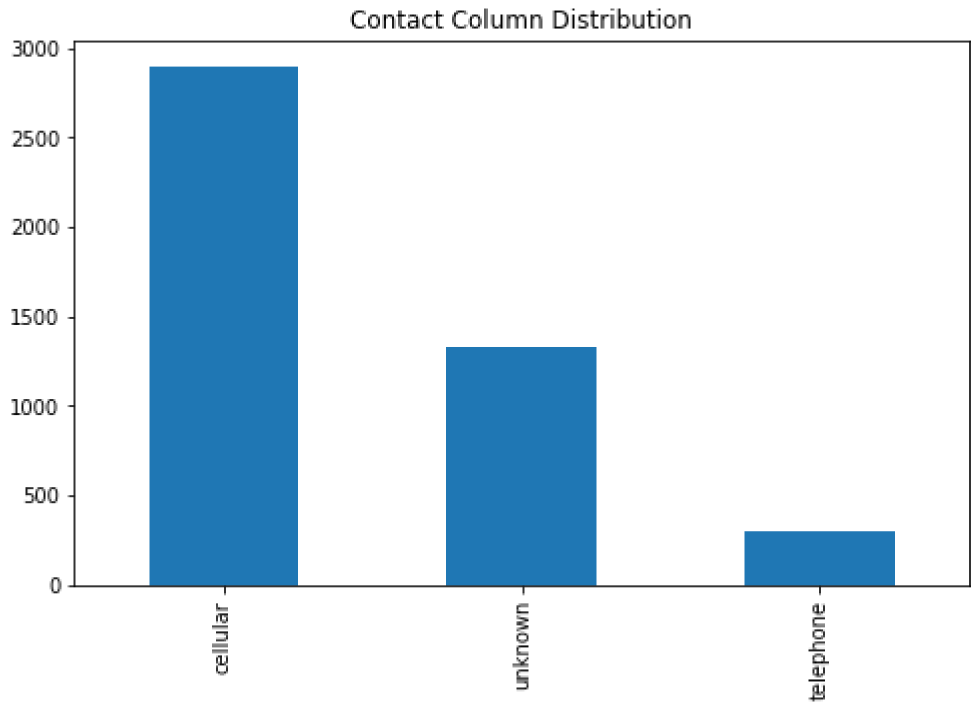
```
cellular     2896
unknown      1324
telephone     301
Name: contact, dtype: int64
```

In [19]:

```
plt.figure(figsize = (8, 5))
contact_count.plot(kind = "bar").set(title = "Contact Column Distribution")
```

Out[19]:

```
[Text(0.5, 1.0, 'Contact Column Distribution')]
```



PLOT MONTH COLUMN

In [20]:

```
month_count = df_bank['month'].value_counts()
month_count
```

Out[20]:

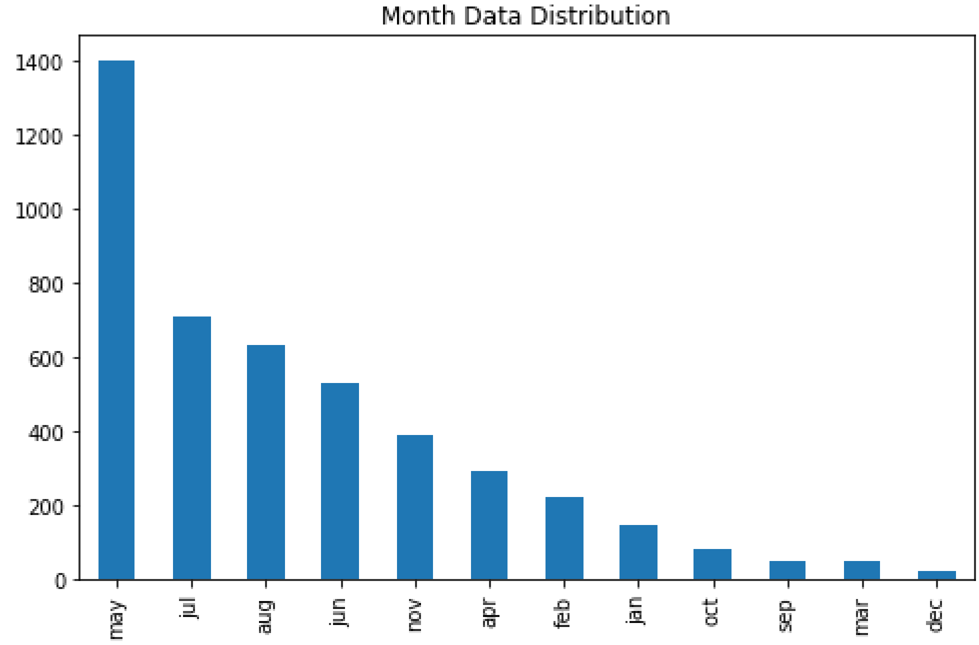
```
may    1398
jul     706
aug     633
jun     531
nov     389
apr     293
feb     222
jan     148
oct      80
sep      52
mar      49
dec      20
Name: month, dtype: int64
```

In [21]:

```
plt.figure(figsize = (8, 5))
month_count.plot(kind = "bar").set(title = "Month Data Distribution")
```

Out[21]:

```
[Text(0.5, 1.0, 'Month Data Distribution')]
```



PLOT pdays COLUMN

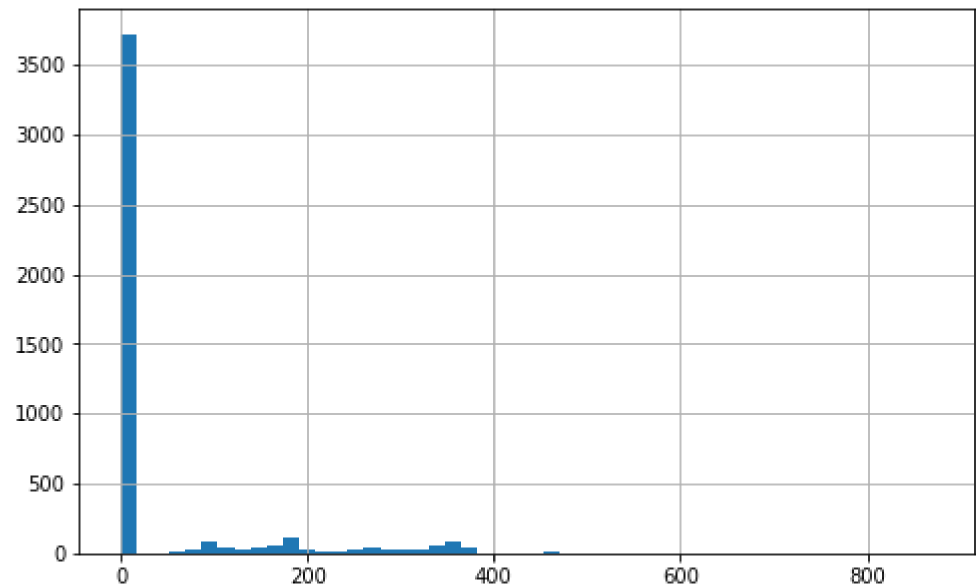
‘pdays’ column into a binary variable indicating whether they were contacted or not.

In [22]:

```
plt.figure(figsize = (8, 5))
df_bank['pdays'].hist(bins = 50)
```

Out[22]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1d0afda5e10>
```



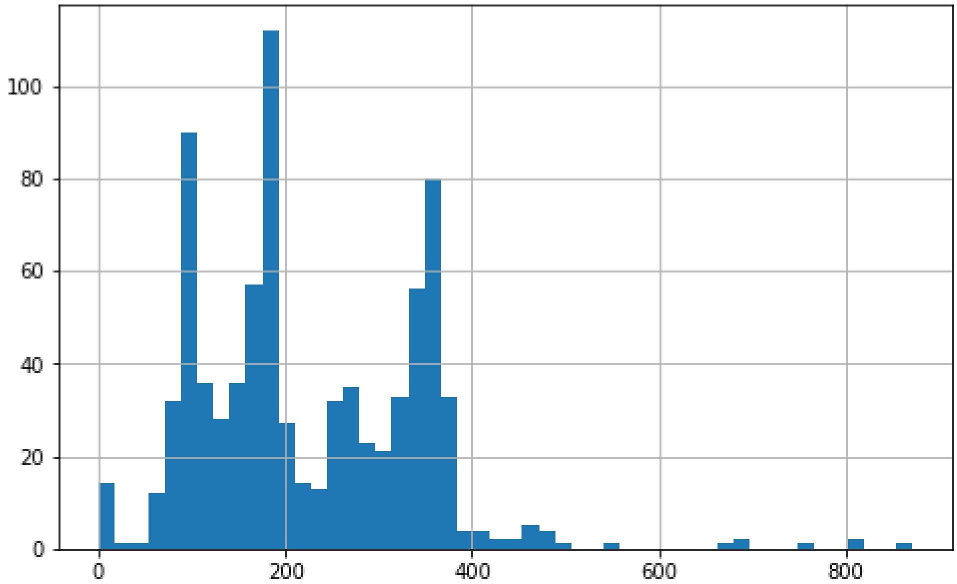
PLOT pdays WHOSE VALUES IS GREATER THAN 0

In [23]:

```
plt.figure(figsize = (8, 5))
df_bank[df_bank['pdays'] > 0]['pdays'].hist(bins=50)
```

Out[23]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1d0aff23518>
```



PLOT TARGET COLUMN

In [24]:

```
target_count = df_bank['y'].value_counts()
target_count
```

Out[24]:

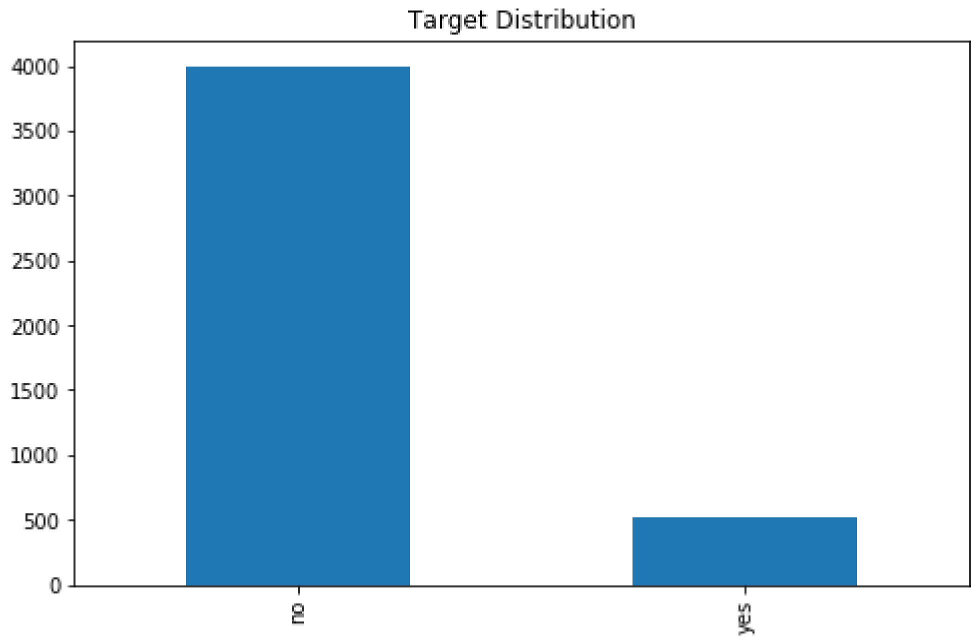
```
no     4000
yes     521
Name: y, dtype: int64
```

In [25]:

```
plt.figure(figsize = (8, 5))
target_count.plot(kind = "bar").set(title = "Target Distribution")
```

Out[25]:

```
[Text(0.5, 1.0, 'Target Distribution')]
```



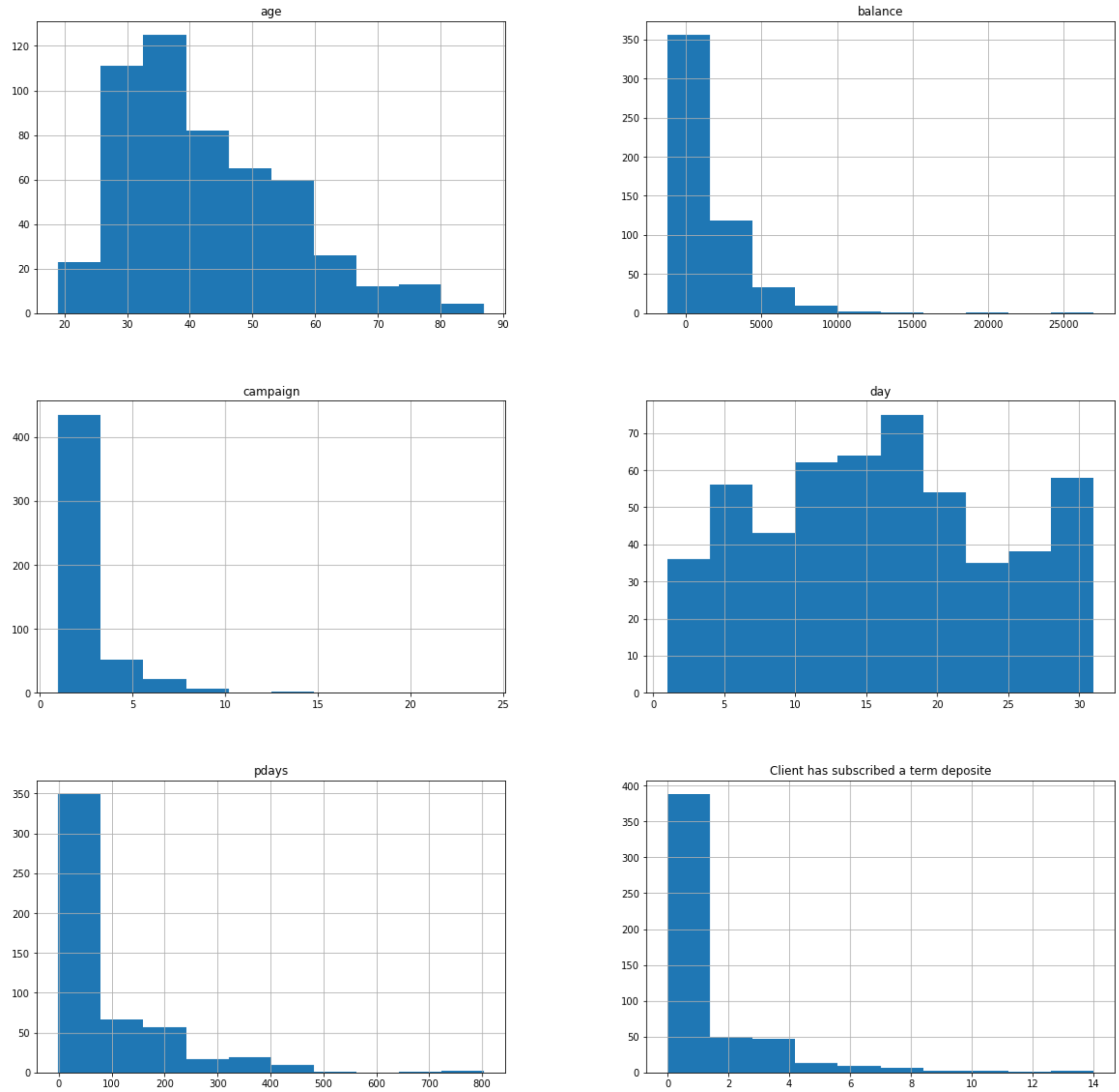
PLOT THAT CLIENT HAS SUBSCRIBED A TERM DEPOSIT

In [26]:

```
df_bank[df_bank['y'] == 'yes'].hist(figsize = (20,20))
plt.title('Client has subscribed a term deposite')
```

Out[26]:

```
Text(0.5, 1.0, 'Client has subscribed a term deposite')
```



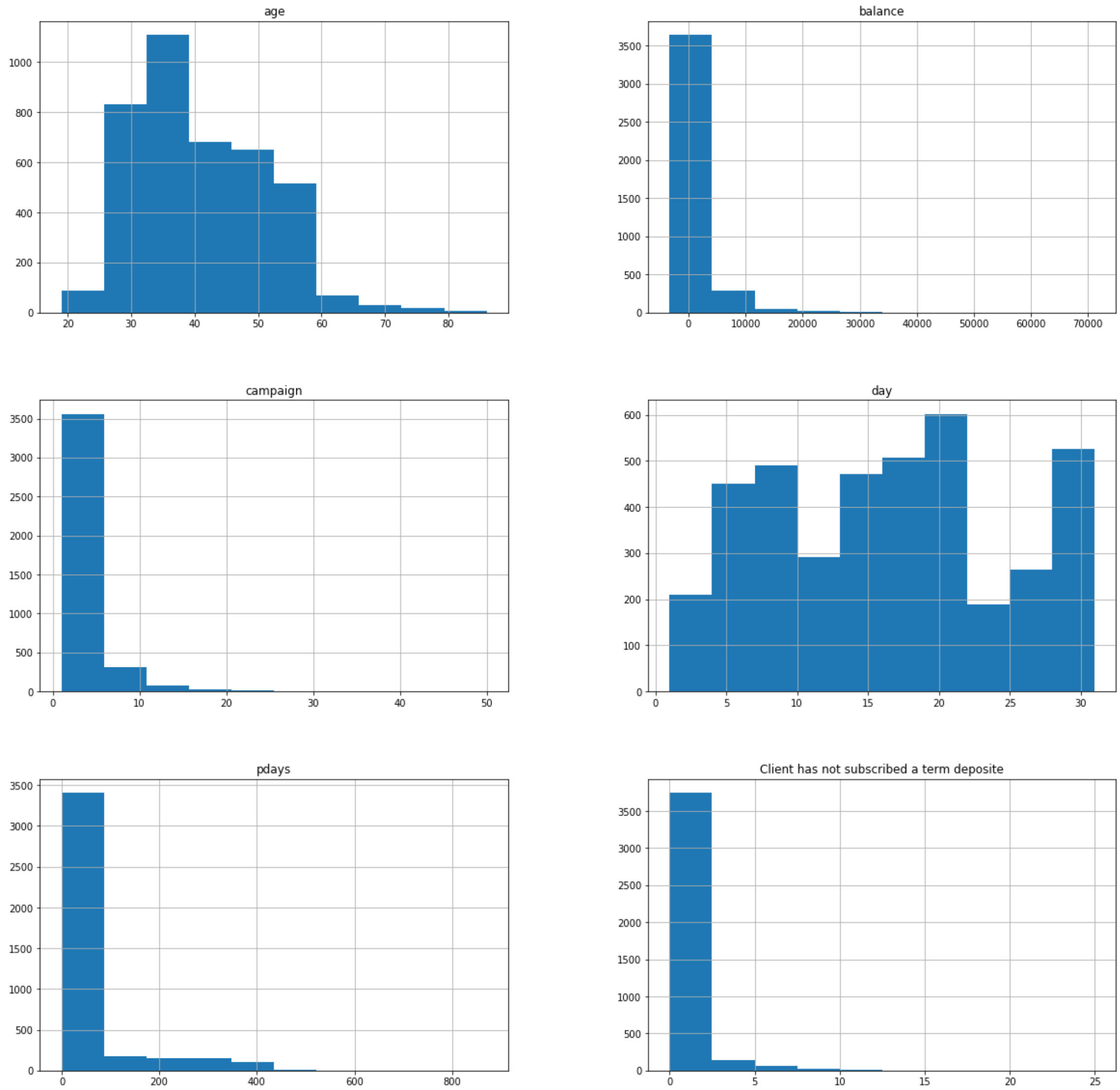
PLOT THAT CLIENT HAS NOT SUBSCRIBED A TERM DEPOSIT

In [27]:

```
df_bank[df_bank['y'] == 'no'].hist(figsize = (20,20))
plt.title('Client has not subscribed a term deposite')
```

Out[27]:

```
Text(0.5, 1.0, 'Client has not subscribed a term deposite')
```



DATA PREPROCESSING

In [28]:

```
df_bank.head(10)
```

Out[28]:

	age	job	marital	education	default	balance	housing	loan	contact	day	mo
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	n
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	n
5	35	management	single	tertiary	no	747	no	no	cellular	23	
6	36	self-employed	married	tertiary	no	307	yes	no	cellular	14	n
7	39	technician	married	secondary	no	147	yes	no	cellular	6	n
8	41	entrepreneur	married	tertiary	no	221	yes	no	unknown	14	n
9	43	services	married	primary	no	-88	yes	yes	cellular	17	

We can see there are some binary columns(default, housing, loan) which are object type, we need to convert into numeric value.

There are categorical columns also, but there are a limited number of choices. They are job, marital, education, contact, month, and poutcome. That also need to converted into numerical format.

All feature columns we need to convert into numeric values then only we can feed into the model.

CONVERT DEFAULT COLUMN INTO NUMERIC VALUE

We can convert the yes values to 1, and the no values to 0 for default column.

In [29]:

```
df_bank['is_default'] = df_bank['default'].apply(lambda row: 1 if row == 'yes' else 0)
```

In [30]:

```
df_bank[['default','is_default']].tail(10)  #view
```

Out[30]:

	default	is_default
4511	no	0
4512	no	0
4513	no	0
4514	no	0
4515	no	0
4516	no	0
4517	yes	1
4518	no	0
4519	no	0
4520	no	0

CONVERT HOUSING COLUMN INTO NUMERIC VALUE

For housing column also we will do the same.

In [31]:

```
df_bank['is_housing'] = df_bank['housing'].apply(lambda row: 1 if row == 'yes' else 0)
df_bank[['housing','is_housing']].tail(10)
```

Out[31]:

	housing	is_housing
4511	yes	1
4512	yes	1
4513	no	0
4514	yes	1
4515	yes	1
4516	yes	1
4517	yes	1
4518	no	0
4519	no	0
4520	yes	1

CONVERT LOAN COLUMN INTO NUMERIC VALUE

In [32]:

```
df_bank['is_loan'] = df_bank['loan'].apply(lambda row: 1 if row == 'yes' else 0)

df_bank[['loan', 'is_loan']].tail(10)
```

Out[32]:

	loan	is_loan
4511	no	0
4512	no	0
4513	no	0
4514	no	0
4515	no	0
4516	no	0
4517	yes	1
4518	no	0
4519	no	0
4520	yes	1

CONVERT TARGET COLUMN ‘y’ INTO NUMERIC VALUE

In [33]:

```
df_bank['target'] = df_bank['y'].apply(lambda row: 1 if row == 'yes' else 0)

df_bank[['y', 'target']].tail(10)
```

Out[33]:

	y	target
4511	yes	1
4512	no	0
4513	no	0
4514	no	0
4515	no	0
4516	no	0
4517	no	0
4518	no	0
4519	no	0
4520	no	0

CREATING ONE-HOT ENCODING FOR NON-NUMERIC MARITAL COLUMN

For marital column, we have three values married, single and divorced. We will use pandas' get_dummies function to convert categorical variable into dummy/indicator variables.

In [34]:

```
marital_dummies = pd.get_dummies(df_bank['marital'], prefix = 'marital')
marital_dummies.tail()
```

Out[34]:

	marital_divorced	marital_married	marital_single
4516	0	1	0
4517	0	1	0
4518	0	1	0
4519	0	1	0
4520	0	0	1

In [35]:

```
# Merge marital_dummies with marital column

pd.concat([df_bank['marital'], marital_dummies], axis=1).head(n=10)
```

Out[35]:

	marital	marital_divorced	marital_married	marital_single
0	married	0	1	0
1	married	0	1	0
2	single	0	0	1
3	married	0	1	0
4	married	0	1	0
5	single	0	0	1
6	married	0	1	0
7	married	0	1	0
8	married	0	1	0
9	married	0	1	0

We can see in each of the rows there is one value of 1, which is in the column corresponding the value in the marital column.

There are three values, if two of the values in the dummy columns are 0 for a particular row, then the remaining column must be equal to 1. It is important to eliminate any redundancy and correlations in features as it becomes difficult to determine which feature is most important in minimizing the total error.

So let us remove one column divorced.

In [36]:

```
# Remove marital_divorced column

marital_dummies.drop('marital_divorced', axis=1, inplace=True)
marital_dummies.head()
```

Out[36]:

	marital_married	marital_single
0	1	0
1	1	0
2	0	1
3	1	0
4	1	0

In [37]:

```
# Merge marital_dummies into main dataframe

df_bank = pd.concat([df_bank, marital_dummies], axis=1)
df_bank.head()
```

Out[37]:

	age	job	marital	education	default	balance	housing	loan	contact	day	...
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	...
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	...
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	...
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	...
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	...

5 rows × 22 columns

CREATING ONE HOT ENCODING FOR JOB COLUMN

In [38]:

```
job_dummies = pd.get_dummies(df_bank['job'], prefix = 'job')
job_dummies.tail()
```

Out[38]:

	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	job_management	job_retired
4516	0	0	0	0	0	0
4517	0	0	0	0	0	0
4518	0	0	0	0	0	0
4519	0	1	0	0	0	0
4520	0	0	1	0	0	0

In [39]:

```
job_dummies.drop('job_unknown', axis=1, inplace=True)
```

In [40]:

```
# Merge job_dummies into main dataframe

df_bank = pd.concat([df_bank, job_dummies], axis=1)
df_bank.head()
```

Out[40]:

	age	job	marital	education	default	balance	housing	loan	contact	day	...
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	...
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	...
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	...
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	...
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	...

5 rows × 33 columns

CREATING ONE HOT ENCODING FOR EDUCATION COLUMN

In [41]:

```
education_dummies = pd.get_dummies(df_bank['education'], prefix = 'education')
education_dummies.tail()
```

Out[41]:

	education_primary	education_secondary	education_tertiary	education_unknown
4516	0	1	0	0
4517	0	0	1	0
4518	0	1	0	0
4519	0	1	0	0
4520	0	0	1	0

In [42]:

```
education_dummies.drop('education_unknown', axis=1, inplace=True)
education_dummies.tail()
```

Out[42]:

	education_primary	education_secondary	education_tertiary
4516	0	1	0
4517	0	0	1
4518	0	1	0
4519	0	1	0
4520	0	0	1

In [43]:

```
df_bank = pd.concat([df_bank, education_dummies], axis=1)
df_bank.head()
```

Out[43]:

	age	job	marital	education	default	balance	housing	loan	contact	day	...
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	...
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	...
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	...
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	...
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	...

5 rows × 36 columns

CREATING ONE HOT ENCODING FOR CONTACT COLUMN

In [44]:

```
contact_dummies = pd.get_dummies(df_bank['contact'], prefix = 'contact')
contact_dummies.tail()
```

Out[44]:

	contact_cellular	contact_telephone	contact_unknown
4516	1	0	0
4517	0	0	1
4518	1	0	0
4519	1	0	0
4520	1	0	0

In [45]:

```
contact_dummies.drop('contact_unknown', axis=1, inplace=True)
contact_dummies.tail()
```

Out[45]:

	contact_cellular	contact_telephone
4516	1	0
4517	0	0
4518	1	0
4519	1	0
4520	1	0

In [46]:

```
df_bank = pd.concat([df_bank, contact_dummies], axis=1)
df_bank.head()
```

Out[46]:

	age	job	marital	education	default	balance	housing	loan	contact	day	...
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	...
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	...
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	...
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	...
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	...

5 rows × 38 columns

CREATING ONE HOT ENCODING FOR POUTCOME COLUMN

In [47]:

```
poutcome_dummies = pd.get_dummies(df_bank['poutcome'], prefix = 'poutcome')
poutcome_dummies.tail()
```

Out[47]:

	poutcome_failure	poutcome_other	poutcome_success	poutcome_unknown
4516	0	0	0	1
4517	0	0	0	1
4518	0	0	0	1
4519	0	1	0	0
4520	0	1	0	0

In [48]:

```
poutcome_dummies.drop('poutcome_unknown', axis=1, inplace=True)
poutcome_dummies.tail()
```

Out[48]:

	poutcome_failure	poutcome_other	poutcome_success
4516	0	0	0
4517	0	0	0
4518	0	0	0
4519	0	1	0
4520	0	1	0

In [49]:

```
df_bank = pd.concat([df_bank, poutcome_dummies], axis=1)
df_bank.head()
```

Out[49]:

	age	job	marital	education	default	balance	housing	loan	contact	day	...
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	...
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	...
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	...
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	...
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	...

5 rows × 41 columns

CONVERT MONTH COLUMN INTO NUMERIC VALUE

In [50]:

```
months = {'jan':1, 'feb':2, 'mar':3, 'apr':4, 'may':5, 'jun':6, 'jul':7, 'aug':8, 'sep'
:9, 'oct':10, 'nov':11, 'dec': 12}
df_bank['month'] = df_bank['month'].map(months)
df_bank['month'].head()
```

Out[50]:

```
0    10
1     5
2     4
3     6
4     5
Name: month, dtype: int64
```

pdays COLUMN

‘pdays’ column indicates the number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted). If the value of ‘pdays’ is ‘-1’, if so we will associate that with a value of 0,

In [51]:

```
df_bank[df_bank['pdays'] == -1]['pdays'].count()
```

Out[51]:

```
3705
```

In [52]:

```
df_bank['was_contacted'] = df_bank['pdays'].apply(lambda row: 0 if row == -1 else 1)
df_bank[['pdays','was_contacted']].head()
```

Out[52]:

	pdays	was_contacted
0	-1	0
1	339	1
2	330	1
3	-1	0
4	-1	0

In [53]:

```
df_bank.drop(['job', 'education', 'marital', 'default', 'housing', 'loan', 'contact',
'pdays', 'poutcome', 'y'], axis=1, inplace=True)
```

View After converting all columns into numeric column

In [59]:

```
df_bank.dtypes
```

Out[59]:

```
age                    int64
balance                int64
day                    int64
month                  int64
campaign               int64
previous               int64
is_default             int64
is_housing             int64
is_loan                int64
target                 int64
marital_married        uint8
marital_single         uint8
job_admin.             uint8
job_blue-collar        uint8
job_entrepreneur       uint8
job_housemaid          uint8
job_management         uint8
job_retired            uint8
job_self-employed      uint8
job_services           uint8
job_student            uint8
job_technician         uint8
job_unemployed         uint8
education_primary      uint8
education_secondary    uint8
education_tertiary     uint8
contact_cellular       uint8
contact_telephone      uint8
poutcome_failure       uint8
poutcome_other         uint8
poutcome_success       uint8
was_contacted          int64
dtype: object
```

In [60]:

```
df_bank.head(10)
```

Out[60]:

	age	balance	day	month	campaign	previous	is_default	is_housing	is_loan	target	..
0	30	1787	19	10	1	0	0	0	0	0	..
1	33	4789	11	5	1	4	0	1	1	0	..
2	35	1350	16	4	1	1	0	1	0	0	..
3	30	1476	3	6	4	0	0	1	1	0	..
4	59	0	5	5	1	0	0	1	0	0	..
5	35	747	23	2	2	3	0	0	0	0	..
6	36	307	14	5	1	2	0	1	0	0	..
7	39	147	6	5	2	0	0	1	0	0	..
8	41	221	14	5	2	0	0	1	0	0	..
9	43	-88	17	4	1	2	0	1	1	0	..

10 rows × 32 columns

CONVERT INTO X(features) and y(target)

In [61]:

```
#The axis=1 argument drop columns
X = df_bank.drop('target', axis=1)
y = df_bank['target']
```

SHAPE of X and y

In [62]:

```
X.shape
```

Out[62]:

```
(4521, 31)
```

In [63]:

```
y.shape
```

Out[63]:

```
(4521,)
```

DIVIDE FEATURES AND TARGET INTO TRAIN AND TEST DATA

In [64]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state
= 32)
```

VIEW THE SHAPE OF X_train, X_test, y_train, y_test

In [65]:

```
X_train.shape
```

Out[65]:

```
(3616, 31)
```

In [66]:

```
y_train.shape
```

Out[66]:

```
(3616,)
```

In [67]:

```
X_test.shape
```

Out[67]:

```
(905, 31)
```

In [68]:

```
y_test.shape
```

Out[68]:

```
(905,)
```

MODELLING

In [69]:

```
def evaluate_model(model, X_test, y_test):
    from sklearn import metrics

    # Predict Test Data
    y_pred = model.predict(X_test)

    # Calculate accuracy, precision, recall, f1-score, and kappa score
    acc = metrics.accuracy_score(y_test, y_pred)
    prec = metrics.precision_score(y_test, y_pred)
    rec = metrics.recall_score(y_test, y_pred)
    f1 = metrics.f1_score(y_test, y_pred)
    kappa = metrics.cohen_kappa_score(y_test, y_pred)

    # Calculate area under curve (AUC)
    y_pred_proba = model.predict_proba(X_test)[::,1]
    fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
    auc = metrics.roc_auc_score(y_test, y_pred_proba)

    # Display confussion matrix
    cm = metrics.confusion_matrix(y_test, y_pred)

    return {'acc': acc, 'prec': prec, 'rec': rec, 'f1': f1, 'kappa': kappa,
            'fpr': fpr, 'tpr': tpr, 'auc': auc, 'cm': cm}
```

After making sure our data is good and ready we can continue to building our model. In this notebook we will try to build 4 different models with different algorithm. In this step we will create a baseline model for each algorithm using the default paramaeters set by sklearn and after building all 4 of our models we will compare them to see which works best for our case.

DECISION TREE

Decision tree is a tree shaped diagram used to determine a course of action. Each branch of the tree represents a possible decision, occurrence or reaction.

BUILDING MODEL

In [70]:

```
from sklearn import tree

# Building Decision Tree model
dtc = tree.DecisionTreeClassifier(random_state=0)
dtc.fit(X_train, y_train)
```

Out[70]:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=Non
e,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=0, splitter='best')
```

MODEL EVALUATION

In [71]:

```
# Evaluate Model
dtc_eval = evaluate_model(dtc, X_test, y_test)

# Print result
print('Accuracy:', dtc_eval['acc'])
print('Precision:', dtc_eval['prec'])
print('Recall:', dtc_eval['rec'])
print('F1 Score:', dtc_eval['f1'])
print('Cohens Kappa Score:', dtc_eval['kappa'])
print('Area Under Curve:', dtc_eval['auc'])
print('Confusion Matrix:\n', dtc_eval['cm'])
```

```
Accuracy: 0.8121546961325967
Precision: 0.1984126984126984
Recall: 0.26595744680851063
F1 Score: 0.2272727272727273
Cohens Kappa Score: 0.12292203498050303
Area Under Curve: 0.5707099194585094
Confusion Matrix:
 [[710 101]
 [ 69  25]]
```

RANDOM FOREST

Random forest or Random Decision Forest is a method that operates by constructing multiple decision trees during training phases. The decision of the majority of the trees is chosen as final decision.

BUILDING MODEL

In [72]:

```
from sklearn.ensemble import RandomForestClassifier

# Building Random Forest model
rf = RandomForestClassifier(random_state=0)
rf.fit(X_train, y_train)
```

```
C:\Users\sohil\Anaconda3\lib\site-packages\sklearn\ensemble\forest.py:245:
FutureWarning: The default value of n_estimators will change from 10 in ve
rsion 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

Out[72]:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gin
i',
                       max_depth=None, max_features='auto', max_leaf_nodes
=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=0, verbo
se=0,
                       warm_start=False)
```

MODEL EVALUATION

In [73]:

```
# Evaluate Model
rf_eval = evaluate_model(rf, X_test, y_test)

# Print result
print('Accuracy:', rf_eval['acc'])
print('Precision:', rf_eval['prec'])
print('Recall:', rf_eval['rec'])
print('F1 Score:', rf_eval['f1'])
print('Cohens Kappa Score:', rf_eval['kappa'])
print('Area Under Curve:', rf_eval['auc'])
print('Confusion Matrix:\n', rf_eval['cm'])
```

```
Accuracy: 0.8994475138121547
Precision: 0.5555555555555556
Recall: 0.1595744680851064
F1 Score: 0.24793388429752067
Cohens Kappa Score: 0.21137806547989535
Area Under Curve: 0.7278379725581762
Confusion Matrix:
 [[799  12]
 [ 79  15]]
```

NAIVE BAYES

Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. There is not a single algorithm for training such classifiers, but a family of algorithms based on a common principle: all naive Bayes classifiers assume that the value of a particular feature is independent of the value of any other feature, given the class variable.

BUILDING MODEL

In [74]:

```
from sklearn.naive_bayes import GaussianNB

# Building Naive Bayes model
nb = GaussianNB()
nb.fit(X_train, y_train)
```

Out[74]:

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

MODEL EVALUATION

In [75]:

```
# Evaluate Model
nb_eval = evaluate_model(nb, X_test, y_test)

# Print result
print('Accuracy:', nb_eval['acc'])
print('Precision:', nb_eval['prec'])
print('Recall:', nb_eval['rec'])
print('F1 Score:', nb_eval['f1'])
print('Cohens Kappa Score:', nb_eval['kappa'])
print('Area Under Curve:', nb_eval['auc'])
print('Confusion Matrix:\n', nb_eval['cm'])
```

```
Accuracy: 0.8320441988950277
Precision: 0.2835820895522388
Recall: 0.40425531914893614
F1 Score: 0.3333333333333333
Cohens Kappa Score: 0.24062092874334795
Area Under Curve: 0.6987826953852613
Confusion Matrix:
 [[715  96]
 [ 56  38]]
```

K-NEAREST NEIGHBORS

K-Nearest Neighbors (KNN) classify new data by finding k-number of closest neighbor from the training data and then decide the class based on the majority of it's neighbors.

BUILDING MODEL

In [76]:

```
from sklearn.neighbors import KNeighborsClassifier

# Building KNN model
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
```

Out[76]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

In [77]:

```
# Evaluate Model
knn_eval = evaluate_model(knn, X_test, y_test)

# Print result
print('Accuracy:', knn_eval['acc'])
print('Precision:', knn_eval['prec'])
print('Recall:', knn_eval['rec'])
print('F1 Score:', knn_eval['f1'])
print('Cohens Kappa Score:', knn_eval['kappa'])
print('Area Under Curve:', knn_eval['auc'])
print('Confusion Matrix:\n', knn_eval['cm'])
```

```
Accuracy: 0.8939226519337017
Precision: 0.25
Recall: 0.010638297872340425
F1 Score: 0.020408163265306122
Cohens Kappa Score: 0.012031203802679
Area Under Curve: 0.5475706377731722
Confusion Matrix:
 [[808   3]
 [ 93   1]]
```

LOGISTIC REGRESSION

In [78]:

```
from sklearn.linear_model import LogisticRegression
logr = LogisticRegression()
logr.fit(X_train, y_train)
```

```
C:\Users\sohil\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.p
y:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. S
pecify a solver to silence this warning.
  FutureWarning)
```

Out[78]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=
0,
                   warm_start=False)
```

MODEL EVALUATION

In [79]:

```
# Evaluate Model
logr_eval = evaluate_model(logr, X_test, y_test)

# Print result
print('Accuracy:', logr_eval['acc'])
print('Precision:', logr_eval['prec'])
print('Recall:', logr_eval['rec'])
print('F1 Score:', logr_eval['f1'])
print('Cohens Kappa Score:', logr_eval['kappa'])
print('Area Under Curve:', logr_eval['auc'])
print('Confusion Matrix:\n', logr_eval['cm'])
```

```
Accuracy: 0.9049723756906077
Precision: 0.6666666666666666
Recall: 0.1702127659574468
F1 Score: 0.2711864406779661
Cohens Kappa Score: 0.2390347875398423
Area Under Curve: 0.7008027914054097
Confusion Matrix:
 [[803   8]
 [ 78  16]]
```

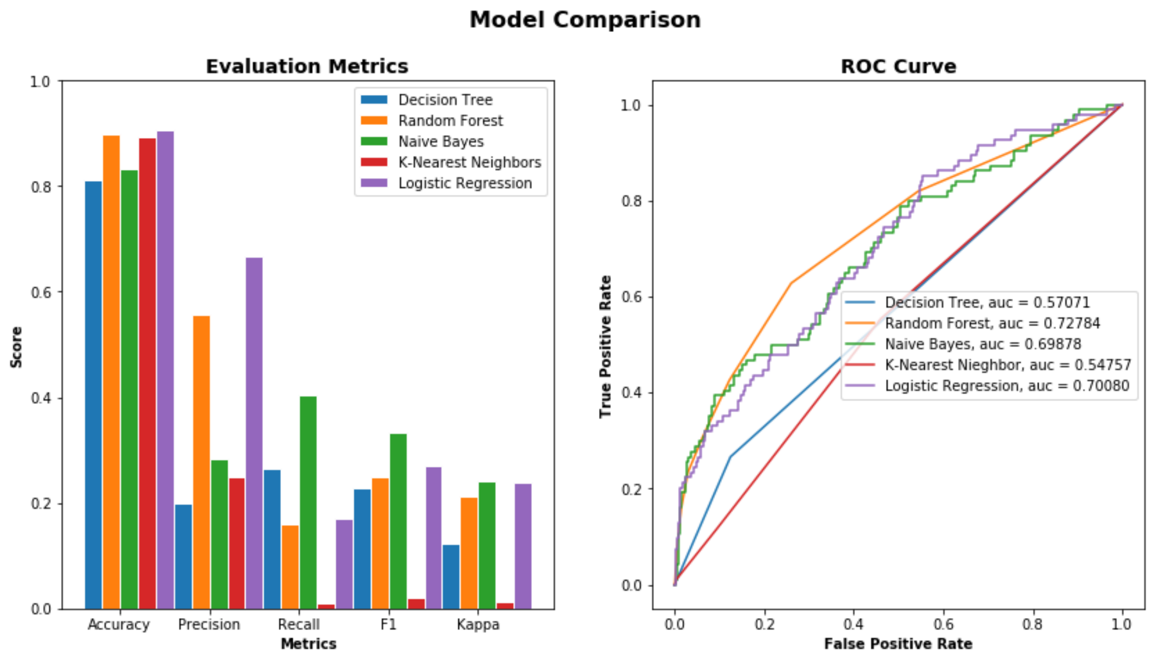
MODEL COMPARISON

After building all of our model, we can now compare how well each model perform. To do this we will create two chart, first is a grouped bar chart to display the value of accuracy, precision, recall, f1, and kappa score of our model, and second a line chart to show the AUC of all our models.

In [80]:

```
# Intitialize figure with two plots
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle('Model Comparison', fontsize=16, fontweight='bold')
fig.set_figheight(7)
fig.set_figwidth(14)
fig.set_facecolor('white')

# First plot
## set bar size
barWidth = 0.2
dtc_score = [dtc_eval['acc'], dtc_eval['prec'], dtc_eval['rec'], dtc_eval['f1'], dtc_ev
al['kappa']]
rf_score = [rf_eval['acc'], rf_eval['prec'], rf_eval['rec'], rf_eval['f1'], rf_eval['ka
ppa']]
nb_score = [nb_eval['acc'], nb_eval['prec'], nb_eval['rec'], nb_eval['f1'], nb_eval['ka
ppa']]
knn_score = [knn_eval['acc'], knn_eval['prec'], knn_eval['rec'], knn_eval['f1'], knn_ev
al['kappa']]
logr_score = [logr_eval['acc'], logr_eval['prec'], logr_eval['rec'], logr_eval['f1'], l
ogr_eval['kappa']]

## Set position of bar on X axis
r1 = np.arange(len(dtc_score))
r2 = [x + barWidth for x in r1]
r3 = [x + barWidth for x in r2]
r4 = [x + barWidth for x in r3]
r5 = [x + barWidth for x in r4]

## Make the plot
ax1.bar(r1, dtc_score, width=barWidth, edgecolor='white', label='Decision Tree')
ax1.bar(r2, rf_score, width=barWidth, edgecolor='white', label='Random Forest')
ax1.bar(r3, nb_score, width=barWidth, edgecolor='white', label='Naive Bayes')
ax1.bar(r4, knn_score, width=barWidth, edgecolor='white', label='K-Nearest Neighbors')
ax1.bar(r5, logr_score, width=barWidth, edgecolor='white', label='Logistic Regression')

## Configure x and y axis
ax1.set_xlabel('Metrics', fontweight='bold')
labels = ['Accuracy', 'Precision', 'Recall', 'F1', 'Kappa']
ax1.set_xticks([r + (barWidth * 1.5) for r in range(len(dtc_score))], )
ax1.set_xticklabels(labels)
ax1.set_ylabel('Score', fontweight='bold')
ax1.set_ylim(0, 1)

## Create legend & title
ax1.set_title('Evaluation Metrics', fontsize=14, fontweight='bold')
ax1.legend()

# Second plot
## Comparing ROC Curve
ax2.plot(dtc_eval['fpr'], dtc_eval['tpr'], label='Decision Tree, auc = {:0.5f}'.format(
dtc_eval['auc']))
ax2.plot(rf_eval['fpr'], rf_eval['tpr'], label='Random Forest, auc = {:0.5f}'.format(rf
_eval['auc']))
ax2.plot(nb_eval['fpr'], nb_eval['tpr'], label='Naive Bayes, auc = {:0.5f}'.format(nb_e
val['auc']))
ax2.plot(knn_eval['fpr'], knn_eval['tpr'], label='K-Nearest Nieghbor, auc = {:0.5f}'.fo
rmat(knn_eval['auc']))
ax2.plot(logr_eval['fpr'], logr_eval['tpr'], label='Logistic Regression, auc = {:0.5f}'
.format(logr_eval['auc']))
```

```
## Configure x and y axis
ax2.set_xlabel('False Positive Rate', fontweight='bold')
ax2.set_ylabel('True Positive Rate', fontweight='bold')

## Create legend & title
ax2.set_title('ROC Curve', fontsize=14, fontweight='bold')
ax2.legend(loc=5)

plt.show()
```



Overall, Logistic Regression is the top contributor in four out of six categories, except recall and F1 score. So we can assume that Logistic Regression is the right choice to solve our problem.

MODEL OPTIMIZATION

Now, we will try to optimise our RandomForest model by tuning the hyper parameters available from the scikit-learn library. After finding the optimal parameters we will then evaluate our new model by comparing it against our base line model before.

Tuning Hyperparameter with GridSearchCV

We will use GridSearchCV functionality from sklearn to find the optimal parameter for our model. We will provide our baseline model (named rf_grids), scoring method (in our case we will use recall as explained before), and also various parameters value we want to try with our model. The GridSearchCV function will then iterate through each parameters combination to find the best scoring parameters.

This function also allow us to use cross validation to train our model, where on each iteration our data will be divided into 5 (the number are adjustable from the parameter) fold. The models then will be trained on 4/5 fold of the data leaving the final fold as validation data, this process will be repeated for 5 times until all of our folds are used as validation data.

In [89]:

```
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search
param_grid = {
    'max_depth': [50, 80, 100],
    'max_features': [2, 3, 4],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 10, 12],
    'n_estimators': [100, 300, 500]
}

# Create a base model
rf_grids = RandomForestClassifier(random_state=0)

# Initiate the grid search model
grid_search = GridSearchCV(estimator=rf_grids, param_grid=param_grid, scoring='recall',
                           cv=5, n_jobs=-1, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

grid_search.best_params_
```

```
Fitting 5 folds for each of 243 candidates, totalling 1215 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    5.6s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:   16.2s
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed:   35.9s
[Parallel(n_jobs=-1)]: Done 624 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 989 tasks      | elapsed:  1.7min
[Parallel(n_jobs=-1)]: Done 1215 out of 1215 | elapsed:  2.1min finished
```

Out[89]:

```
{'max_depth': 50,
 'max_features': 4,
 'min_samples_leaf': 3,
 'min_samples_split': 12,
 'n_estimators': 100}
```

EVALUATING OPTIMIZED MODEL

After finding the best parameter for the model we can access the best*estimator* attribute of the GridSearchCV object to save our optimised model into variable called best_grid. We will calculate the 6 evaluation metrics using our helper function to compare it with our base model on the next step.

In [90]:

```
# Select best model with best fit
best_grid = grid_search.best_estimator_

# Evaluate Model
best_grid_eval = evaluate_model(best_grid, X_test, y_test)

# Print result
print('Accuracy:', best_grid_eval['acc'])
print('Precision:', best_grid_eval['prec'])
print('Recall:', best_grid_eval['rec'])
print('F1 Score:', best_grid_eval['f1'])
print('Cohens Kappa Score:', best_grid_eval['kappa'])
print('Area Under Curve:', best_grid_eval['auc'])
print('Confusion Matrix:\n', best_grid_eval['cm'])
```

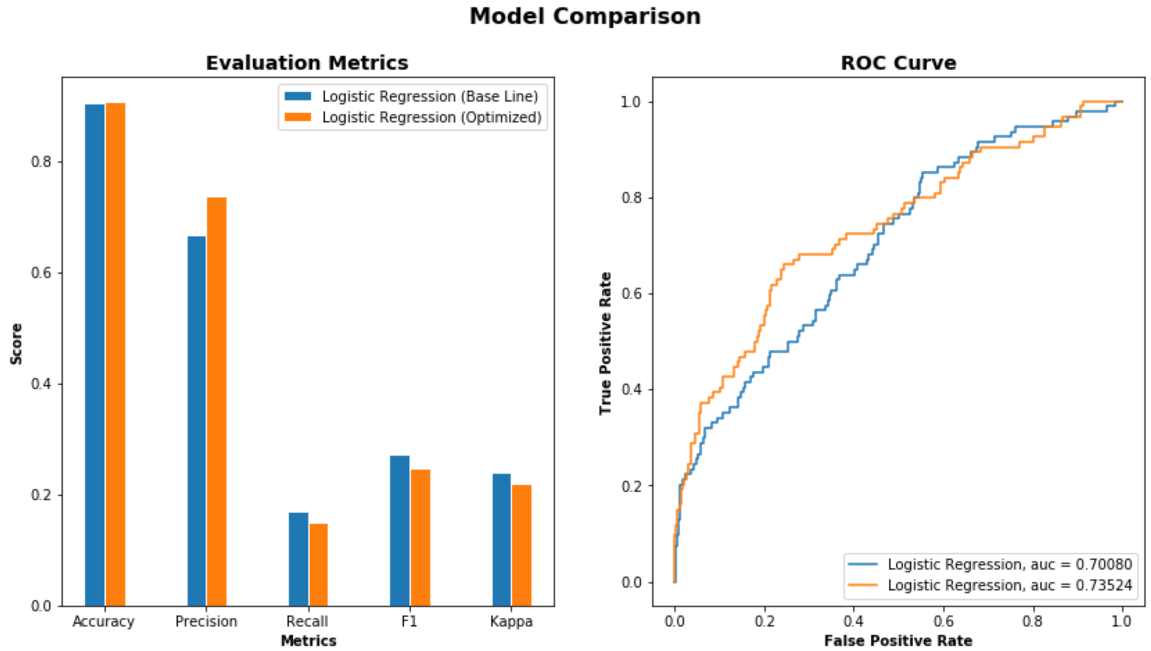
```
Accuracy: 0.9060773480662984
Precision: 0.7368421052631579
Recall: 0.14893617021276595
F1 Score: 0.24778761061946902
Cohens Kappa Score: 0.22056275521060265
Area Under Curve: 0.7352362462943043
Confusion Matrix:
 [[806   5]
 [ 80  14]]
```

MODEL COMPARISON

In [91]:

```
# Intitialize figure with two plots
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle('Model Comparison', fontsize=16, fontweight='bold')
fig.set_figheight(7)
fig.set_figwidth(14)
fig.set_facecolor('white')

# First plot
## set bar size
barWidth = 0.2
logr_score = [logr_eval['acc'], logr_eval['prec'], logr_eval['rec'], logr_eval['f1'], logr_eval['kappa']]
best_grid_score = [best_grid_eval['acc'], best_grid_eval['prec'], best_grid_eval['rec'], best_grid_eval['f1'], best_grid_eval['kappa']]

## Set position of bar on X axis
r1 = np.arange(len(logr_score))
r2 = [x + barWidth for x in r1]

## Make the plot
ax1.bar(r1, logr_score, width=barWidth, edgecolor='white', label='Logistic Regression (Base Line)')
ax1.bar(r2, best_grid_score, width=barWidth, edgecolor='white', label='Logistic Regression (Optimized)')

## Add xticks on the middle of the group bars
ax1.set_xlabel('Metrics', fontweight='bold')
labels = ['Accuracy', 'Precision', 'Recall', 'F1', 'Kappa']
ax1.set_xticks([r + (barWidth * 0.5) for r in range(len(dtc_score))], )
ax1.set_xticklabels(labels)
ax1.set_ylabel('Score', fontweight='bold')
# ax1.set_ylim(0, 1)

## Create Legend & Show graphic
ax1.set_title('Evaluation Metrics', fontsize=14, fontweight='bold')
ax1.legend()

# Second plot
## Comparing ROC Curve
ax2.plot(logr_eval['fpr'], logr_eval['tpr'], label='Logistic Regression, auc = {:0.5f}'.format(logr_eval['auc']))
ax2.plot(best_grid_eval['fpr'], best_grid_eval['tpr'], label='Logistic Regression, auc = {:0.5f}'.format(best_grid_eval['auc']))

ax2.set_title('ROC Curve', fontsize=14, fontweight='bold')
ax2.set_xlabel('False Positive Rate', fontweight='bold')
ax2.set_ylabel('True Positive Rate', fontweight='bold')
ax2.legend(loc=4)

plt.show()

print('Change of {:0.2f}% on accuracy.'.format(100 * ((best_grid_eval['acc'] - logr_eval['acc']) / logr_eval['acc'])))
print('Change of {:0.2f}% on precision.'.format(100 * ((best_grid_eval['prec'] - logr_eval['prec']) / logr_eval['prec'])))
print('Change of {:0.2f}% on recall.'.format(100 * ((best_grid_eval['rec'] - logr_eval['rec']) / logr_eval['rec'])))
print('Change of {:0.2f}% on F1 score.'.format(100 * ((best_grid_eval['f1'] - logr_eval['f1']) / logr_eval['f1'])))
```

```
print('Change of {:0.2f}% on Kappa score.'.format(100 * ((best_grid_eval['kappa'] - log
r_eval['kappa']) / logr_eval['kappa'])))
print('Change of {:0.2f}% on AUC.'.format(100 * ((best_grid_eval['auc'] - logr_eval['au
c']) / logr_eval['auc'])))
```




```
Change of 0.12% on accuracy.
Change of 10.53% on precision.
Change of -12.50% on recall.
Change of -8.63% on F1 score.
Change of -7.73% on Kappa score.
Change of 4.91% on AUC.
```

The result show that our optimised performed little bit better than the original model, as well as slightly decrease in case of recall, F1 and Kappa. The optimised models show an increase in 3 out of the 6 metrics.

OUTPUT

As data scientist it's important to be able to develop a model with good re-usability. In this final part I will explain on how to create a prediction based on new data and also how to save (and load) your model using joblib so you can use it in production or just save it for later use without having to repeat the whole process.

PREDICTION

In this step we will predict the expected outcome of all the row from our dataset then save it into a csv file for easier access in the future.

In [92]:

```
df_bank['deposit_prediction'] = logr.predict(X)
df_bank['deposit_prediction'] = df_bank['deposit_prediction'].apply(lambda x: 'yes' if
x==0 else 'no')

# Save new dataframe into csv file
df_bank.to_csv('deposit_prediction.csv', index=False)

df_bank.head(10)
```

Out[92]:

	age	balance	day	month	campaign	previous	is_default	is_housing	is_loan	target	..
0	30	1787	19	10	1	0	0	0	0	0	..
1	33	4789	11	5	1	4	0	1	1	0	..
2	35	1350	16	4	1	1	0	1	0	0	..
3	30	1476	3	6	4	0	0	1	1	0	..
4	59	0	5	5	1	0	0	1	0	0	..
5	35	747	23	2	2	3	0	0	0	0	..
6	36	307	14	5	1	2	0	1	0	0	..
7	39	147	6	5	2	0	0	1	0	0	..
8	41	221	14	5	2	0	0	1	0	0	..
9	43	-88	17	4	1	2	0	1	1	0	..

10 rows × 33 columns

SAVING MODEL

We can also save our model for further model reusability. This model can then be loaded on another machine to make new prediction without doing the whole training process again.

In [93]:

```
from joblib import dump, load

# Saving model
dump(logr, 'bank_deposit_classification.joblib')
# Loading model
# clf = load('bank_deposit_classification.joblib')
```

Out[93]:

```
['bank_deposit_classification.joblib']
```

CONCLUSION

For a simple model we can see that our model did decently on classifying the data. But there are still some weakness on our model, especially shown on the recall metric where we only get about 17%. This means that our model are only able to detect 17% of potential customer and miss the other 83%. But, referring to the positive part our model gives us 90% accuracy and 66% precision. Our AUC is around 70%, the higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes. The result is not that much different after optimising the model using GridSearchCV which can means that we hit our limit with this model. To improve our performance we can try to look into another algorithm such as GradientBoostingClassifier.

In [ ]:

```

```