University of Tehran

Electrical and Computer Engineering Department

Artificial Intelligence

Computer Assignment 0

Name	Danial Saeedi
Student No	810198571
Date	Thursday - 2022 24 February

Table of contents

Importing Dependencies

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import scipy.stats as stats
        np.warnings.filterwarnings('ignore')
```

Part 1: Loading the dataset

A) Info

The info() method prints information about the DataFrame. The information contains the **number of columns**, **column labels**, **column data types**, **memory usage**, **range index**, and the **number of cells** in each column (non-null values).

This function is useful for finding which column has missing values. For example age column has 4521 - 3984 = 537 missing values.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 12 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        3984 non-null   float64
 1   job        4521 non-null   object
 2   marital    4401 non-null   object
 3   education  4521 non-null   object
 4   balance    4164 non-null   float64
 5   housing    4302 non-null   object
 6   loan       4521 non-null   object
 7   duration   4388 non-null   float64
 8   campaign   4521 non-null   int64
 9   pdays      4521 non-null   int64
 10  poutcome   4521 non-null   object
 11  y          4087 non-null   object
dtypes: float64(3), int64(2), object(7)
memory usage: 424.0+ KB
```

Figure 1 info() method output

B) Head

The head() method prints the top n-rows of a Pandas DataFrame (which by default n = 5).

	age	job	marital	education	balance	housing	loan	duration	campaign	pdays	poutcome	y
0	30.0	unemployed	married	primary	1787.0	no	no	79.0	1	-1	unknown	no
1	33.0	services	married	secondary	4789.0	yes	yes	NaN	1	339	failure	no
2	NaN	management	single	tertiary	135.0	yes	no	185.0	1	330	failure	no
3	30.0	management	married	tertiary	1476.0	yes	yes	199.0	4	-1	unknown	no
4	59.0	blue-collar	married	secondary	NaN	yes	no	226.0	1	-1	unknown	no

Figure 2 head() method output

C) Tail

The tail() method prints the bottom n-rows of a Pandas DataFrame (which by default n = 5).

	age	job	marital	education	balance	housing	loan	duration	campaign	pdays	poutcome	y
4516	33.0	services	married	secondary	-333.0	yes	no	329.0	5	-1	unknown	no
4517	57.0	self-employed	married	tertiary	-3313.0	yes	yes	153.0	1	-1	unknown	no
4518	57.0	technician	married	secondary	295.0	no	no	151.0	11	-1	unknown	no
4519	28.0	blue-collar	married	secondary	1137.0	no	no	129.0	4	211	other	no
4520	44.0	entrepreneur	single	tertiary	1136.0	yes	yes	345.0	2	249	other	no

Figure 3 tail() method output

D) Describe

The describe() method returns description of the data in the DataFrame.

If the DataFrame contains numerical data, the description contains these information for each column:

1. **count**: The number of not-empty values.
2. **mean**: The average (mean) value.
3. **std**: The standard deviation.
4. **min**: The minimum value.
5. **max**: The maximum value.
6. **25%**: The 25% percentile*.
7. **50%**: The 50% percentile*.
8. **75%**: The 75% percentile*.

	age	balance	duration	campaign	pdays
count	3984.000000	4164.000000	4388.000000	4521.000000	4521.000000
mean	41.617470	1136.750240	264.724020	2.793630	39.766645
std	10.696378	2726.204918	261.057119	3.109807	100.121124
min	19.000000	-3313.000000	4.000000	1.000000	-1.000000
25%	32.000000	58.000000	104.000000	1.000000	-1.000000
50%	40.000000	316.000000	185.500000	2.000000	-1.000000
75%	49.000000	997.000000	331.000000	3.000000	-1.000000
max	87.000000	71188.000000	3025.000000	50.000000	871.000000

Figure 4 describe() method output

Part 2: Dealing with non-numerical values

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 12 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   age        3984 non-null   float64
 1   job        4521 non-null   object 
 2   marital    4401 non-null   object 
 3   education  4521 non-null   object 
 4   balance    4164 non-null   float64
 5   housing    4302 non-null   object 
 6   loan       4521 non-null   object 
 7   duration   4388 non-null   float64
 8   campaign   4521 non-null   int64  
 9   pdays      4521 non-null   int64  
 10  poutcome   4521 non-null   object 
 11  y          4087 non-null   object 
dtypes: float64(3), int64(2), object(7)
memory usage: 424.0+ KB
```

Figure 5 info() method

Here, we're going to set the categorical columns data type:

```
In [10]: df['job'] = df['job'].astype('category')
         df['marital'] = df['marital'].astype('category')
         df['education'] = df['education'].astype('category')
         df['housing'] = df['housing'].astype('category')
         df['loan'] = df['loan'].astype('category')
         df['poutcome'] = df['poutcome'].astype('category')
         df['y'] = df['y'].astype('category')
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 12 columns):
 #   Column     Non-Null Count  Dtype   
---  ------     --------------  -----   
 0   age        3984 non-null   float64 
 1   job        4521 non-null   category
 2   marital    4401 non-null   category
 3   education  4521 non-null   category
 4   balance    4164 non-null   float64 
 5   housing    4302 non-null   category
 6   loan       4521 non-null   category
 7   duration   4388 non-null   float64 
 8   campaign   4521 non-null   int64   
 9   pdays      4521 non-null   int64   
 10  poutcome   4521 non-null   category
 11  y          4087 non-null   category
dtypes: category(7), float64(3), int64(2)
memory usage: 208.9 KB
```

Figure 6 info() method

Here, we're going to replace categorical string with their corresponding codes:

```
In [13]: new_df['job'] = df['job'].cat.codes
         new_df['marital'] = df['marital'].cat.codes
         new_df['education'] = df['education'].cat.codes
         new_df['poutcome'] = df['poutcome'].cat.codes
```

Part 3: Dealing with missing values

Here's a pros and cons of replacing missing values with the mean of the column:

Pros:

1. This is a better approach when the data size is small
2. It can prevent data loss which results in removal of the rows and columns

Cons:

3. Imputing the approximations add variance and bias
4. Works poorly compared to other multiple-imputations method

We can count missing values per column using these methods:

```
In [16]: new_df.isnull().sum()
```

```
age          537
job            0
marital        0
education      0
balance      357
housing      219
loan           0
duration     133
campaign       0
pdays          0
poutcome       0
y            434
dtype: int64
```

Figure 7 Missing values per column

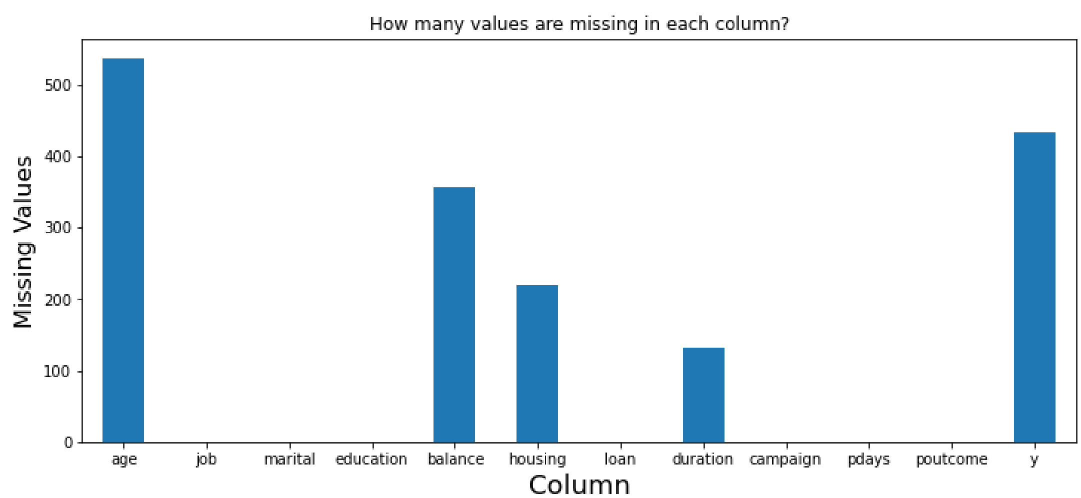


Figure 8 Missing values per column

Here, we're going to replace NaN values of other columns with its mean:

```
In [18]: new_df['age'].fillna(value=new_df['age'].mean(),inplace=True)
         new_df['balance'].fillna(value=new_df['balance'].mean(),inplace=True)
         new_df['duration'].fillna(value=new_df['duration'].mean(),inplace=True)
```

Since housing column is a binary feature, we can't just replace it with the mean value. Here, we're going to replace NaN values of housing column with 1:

```
In [62]: new_df['housing'].fillna(value=new_df['housing'].mode()[0], inplace=True)
```

Here, we're going to remove rows that their target is NaN. We're going to use them as test set.

```
In [41]: new_df = new_df.dropna(subset=['y'])
```

```
In [38]: test = new_df[new_df['y'].isnull()].copy()
```

Part 4

We can count how many housing loans have been given to people by using value_counts() method. As you can see, 2389 loans have been given to people.

```
In [25]: df2['housing'].value_counts()
Out[25]: 1.0    2389
         0.0    1698
         Name: housing, dtype: int64
```

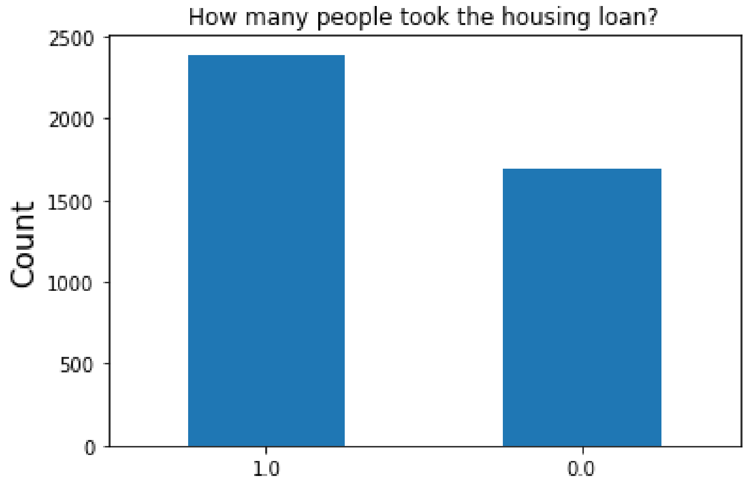


Figure 9 How many loans have been given?

464 long deposit term have been given to people. As you can see, the dataset is not balanced.

```
In [27]: df2['y'].value_counts()
Out[27]: 0.0    3623
         1.0     464
         Name: y, dtype: int64
```

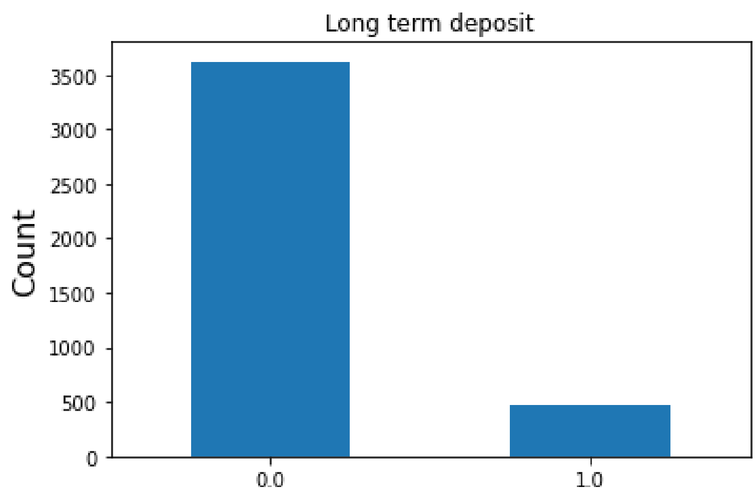


Figure 10 Long term deposit

Part 5

Note that the categorical code for divorced, married, and single status are 0,1, and 2 respectively. There are **14 records** that statisfies the mentioned conditions.

```
In [29]: df['marital'].cat.categories
Out[29]: Index(['divorced', 'married', 'single'], dtype='object')

In [30]: df['poutcome'].cat.categories
Out[30]: Index(['failure', 'other', 'success', 'unknown'], dtype='object')

In [31]: len(df2[(df2['age'] > 35) & (df2['marital'] == 2) & (df2['poutcome'] == 2)])
Out[31]: 14
```

Part 6

The mean balance for secondary education status is 950.7017237980879.

```
In [32]: df['education'].cat.categories
Out[32]: Index(['primary', 'secondary', 'tertiary', 'unknown'], dtype='object')

In [33]: %%time
         df2[df2['education'] == 1]['balance'].mean()

         CPU times: user 2.07 ms, sys: 1.59 ms, total: 3.66 ms
         Wall time: 2.25 ms

Out[33]: 950.7017237980879
```

Part 7: Without vectorization

It takes 2.49ms to run the code with vectorization. But without any vectorization, it took longer to run(550 ms!!).

```
In [34]: %%time
         balance_sum = 0
         count = 0
         for i in range(0, len(df2)):
             if df2.iloc[i]['education'] == 1:
                 balance_sum += df2.iloc[i]['balance']
                 count += 1
         mean = balance_sum/count
         print(f'The mean value is {mean}')

         The mean value is 950.7017237980879
         CPU times: user 547 ms, sys: 3.33 ms, total: 550 ms
         Wall time: 549 ms
```

Figure 11 Comparing runtime

Vectorization	3.66 ms
Without Vectorization	550 ms

Part 8: Distribution of columns

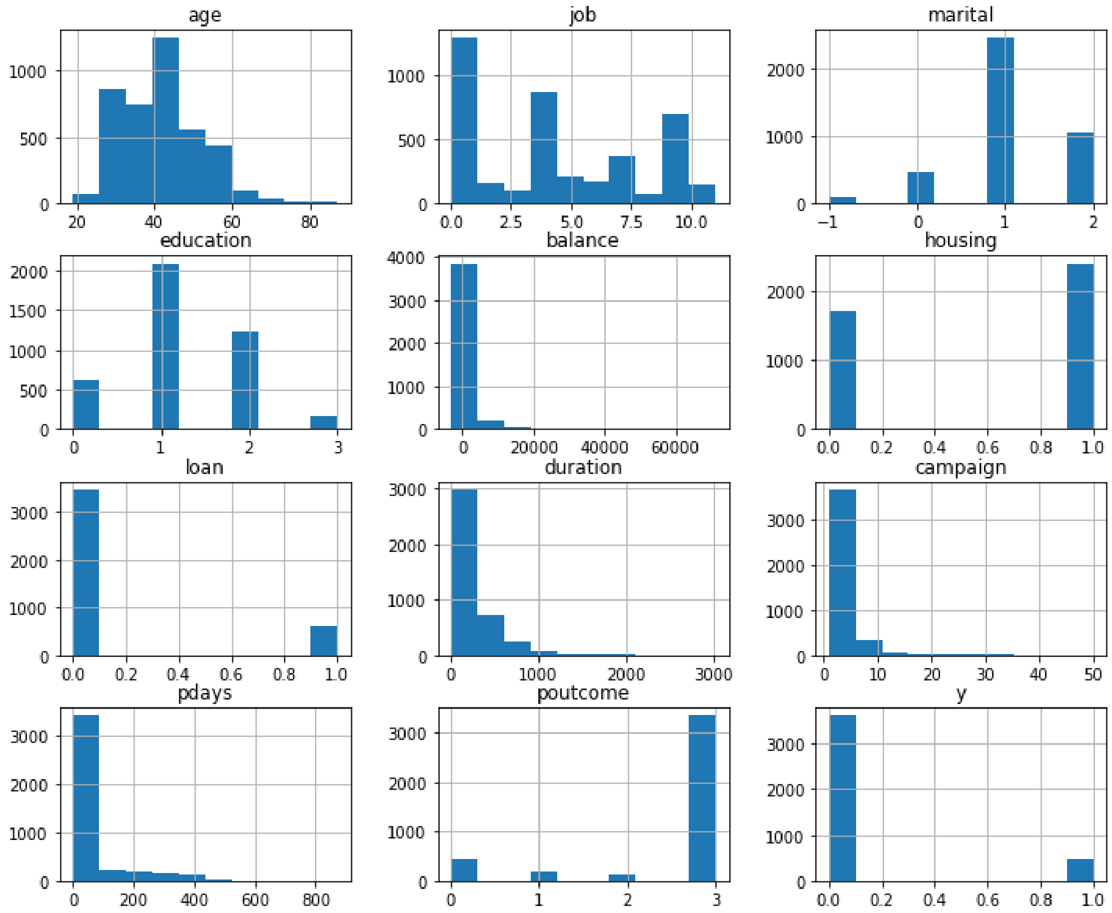


Figure 12 Distribution of columns

Part 9: Normalize

Normalizing a vector most often means dividing by a norm of the vector. It also often refers to rescaling by the minimum and range of the vector, to make all the elements lie between 0 and 1 thus bringing all the values of numeric columns in the dataset to a common scale.

```
In [36]: def normalize(data):
             mean = data.mean()
             std = data.std()
             return (data-mean)/std
```

```
In [37]: new_df['age'] = normalize(new_df['age'])
         new_df['balance'] = normalize(new_df['balance'])
         new_df['duration'] = normalize(new_df['duration'])
         new_df['campaign'] = normalize(new_df['campaign'])
         new_df['pdays'] = normalize(new_df['pdays'])
```

	age	job	marital	education	balance	housing	loan	duration	campaign	pdays	poutcome	y
0	-1.157015e+00	10	1	0	2.485351e-01	0.0	0	-7.221342e-01	-0.576766	-0.407173	3	0.0
1	-8.582367e-01	7	1	1	1.395944e+00	1.0	1	2.210192e-16	-0.576766	2.988713	0	0.0
2	1.981415e-14	4	2	2	-3.828838e-01	1.0	0	-3.099838e-01	-0.576766	2.898822	0	0.0
3	-1.157015e+00	4	1	2	1.296663e-01	1.0	1	-2.555489e-01	0.387925	-0.407173	3	0.0
4	1.731172e+00	1	1	1	1.477395e-15	1.0	0	-1.505672e-01	-0.576766	-0.407173	3	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...
4516	-8.582367e-01	7	1	1	-5.617603e-01	1.0	0	2.499186e-01	0.709488	-0.407173	3	0.0
4517	1.531987e+00	6	1	2	-1.700760e+00	1.0	1	-4.344066e-01	-0.576766	-0.407173	3	0.0
4518	1.531987e+00	9	1	1	-3.217294e-01	0.0	0	-4.421830e-01	2.638868	-0.407173	3	0.0
4519	-1.356200e+00	1	1	1	9.546191e-05	0.0	0	-5.277237e-01	0.387925	1.710262	1	0.0
4520	2.372825e-01	2	2	2	-2.867529e-04	1.0	1	3.121300e-01	-0.255202	2.089802	1	0.0

Figure 13 Data frame after normalization

Part 10

Here, we're going to divide the dataframe into 2 dataframes, one contains positive target and one contains negative target:

In [42]:
```
positive_df = new_df[new_df['y'] == 1]
negative_df = new_df[new_df['y'] == 0]
positive_df
```

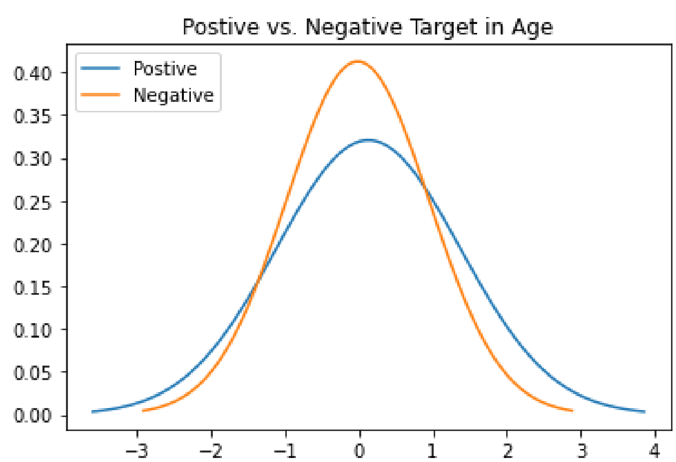
Out[42]:

	age	job	marital	education	balance	housing	loan	duration	campaign	pdays	poutcome	y
13	-2.152941	8	2	1	-0.414608	0.0	0	-0.014480	-0.576766	-0.407173	3	1.0
30	2.627506	5	0	1	1.166615	0.0	0	2.458423	-0.255202	-0.407173	3	1.0
33	-0.957829	4	2	2	0.534814	1.0	0	2.695604	1.031051	-0.407173	3	1.0
36	3.623433	5	0	0	-0.346956	0.0	0	-0.652146	-0.576766	-0.407173	3	1.0
37	-0.957829	1	1	1	-0.324023	1.0	0	-0.516059	-0.576766	-0.407173	3	1.0
...	...	...	...	...	...	...	...	...	...	...	...	...
4494	-1.555385	9	2	1	-0.179163	1.0	0	1.210307	0.066361	-0.407173	3	1.0
4503	1.830765	6	1	0	-0.296121	0.0	1	2.143478	1.031051	-0.407173	3	1.0
4504	0.038097	1	2	1	-0.427603	1.0	1	2.668386	0.066361	3.298338	0	1.0
4505	-0.957829	0	2	1	-0.410785	1.0	0	3.768750	0.066361	-0.407173	3	1.0
4511	0.436468	1	1	1	-0.179163	1.0	0	3.881508	-0.255202	-0.407173	3	1.0

464 rows × 12 columns

A) Age

This column has a low difference of mean values in 0 and 1 classes and high variance in both classes as well. So this column is not a good choice.

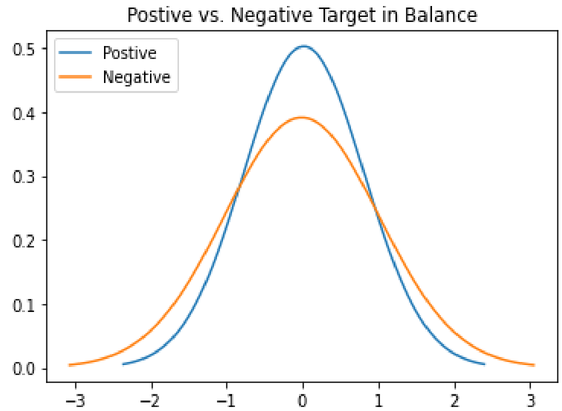


In [45]:
```
print('Column Name: Age')
print('Mean difference: ',abs(positive_mu_age - negative_mu_age))
print('std positive: ', positive_std_age,'std negative: ', negative_std_age)
```

```
Column Name: Age
Mean difference:  0.14378185308679126
std positive:  1.243621879574519 std negative:  0.9666343765199411
```

B) Balance

This column has the lowest difference of mean values in 0 and 1 classes and high variance in both classes as well. So this column is not a good choice.

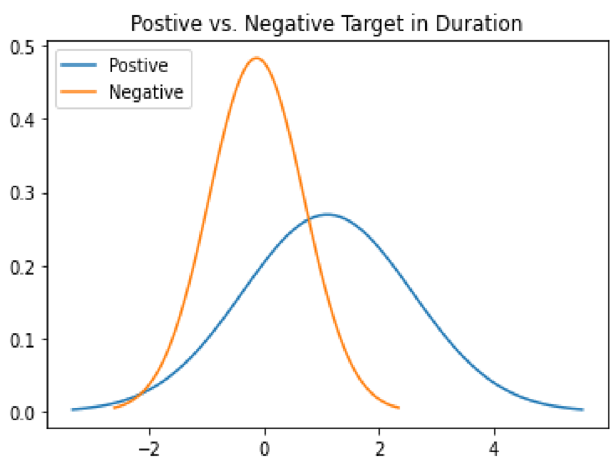



```
In [45]: print('Column Name: Age')
         print('Mean difference: ',abs(positive_mu_age - negative_mu_age))
         print('std positive: ', positive_std_age,'std negative: ', negative_std_age)

Column Name: Age
Mean difference:  0.14378185308679126
std positive:  1.243621879574519 std negative:  0.9666343765199411
```

C) Duration

This column has the highest difference of mean values in 0 and 1 classes. So this column is the best one to choose.

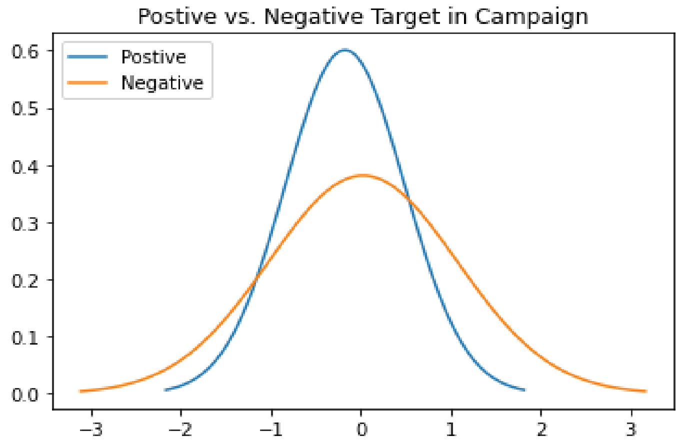



```
In [47]: print('Column Name: Balance')
         print('Mean difference: ',abs(positive_mu_balance - negative_mu_balance))
         print('std positive: ', positive_std_balance,'std negative: ', negative_std_balance)

Column Name: Balance
Mean difference:  0.028517030433160166
std positive:  0.7941521531675242 std negative:  1.0199960940761061
```

D) Campaign

This column has a low difference of mean values in 0 and 1 classes and high variance in both classes as well. So this column is not a good choice.

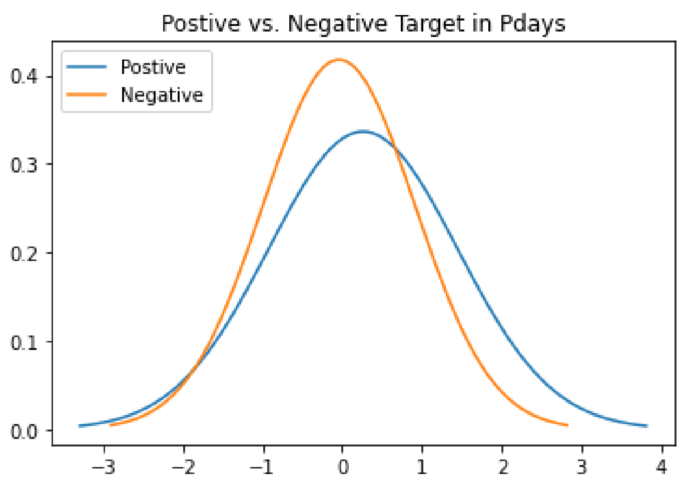



```
In [51]: print('Column Name: Campaign')
         print('Mean difference: ',abs(positive_mu_campaign - negative_mu_campaign))
         print('std positive: ', positive_std_campaign,'std negative: ', negative_std_campaign)

Column Name: Campaign
Mean difference:  0.20043349320360143
std positive:  0.6638036831361747 std negative:  1.0456746706446718
```

E) Pdays

This column has a low difference of mean values in 0 and 1 classes and high variance in both classes as well. So this column is not a good choice.




```
In [53]: print('Column Name: PDays')
         print('Mean difference: ',abs(positive_mu_pdays - negative_mu_pdays))
         print('std positive: ', positive_std_pdays,'std negative: ', negative_std_pdays)

Column Name: PDays
Mean difference:  0.3028186502220774
std positive:  1.186072743572173 std negative:  0.9549927730758819
```

Which feature can separate these 2 classes better?

The criteria for selecting the best column for seperation of these two classes are:

1. Biggest difference of mean value
2. Low variance

Duration column satisfies the above condition.

Part 11: Classification

```
In [54]: def predict(duration):
             positive_prob = stats.norm.pdf(duration, positive_mu_duration, positive_std_duration)
             negative_prob = stats.norm.pdf(duration, negative_mu_duration, negative_std_duration)

             if positive_prob > negative_prob:
                 return 'yes'
             else:
                 return 'no'
```

The result is saved at “predictions.csv”.

```
In [56]: test['y'] = test['duration'].apply(predict)
```

```
In [57]: test['y'].value_counts()
```

```
Out[57]: no      379
         yes      55
         Name: y, dtype: int64
```

```
In [58]: test['y'].to_csv('prediction.csv', encoding='utf-8')
```