Capstone 1

Bank Marketing Dataset

Submitted By,

Sankeerth Ankam

Submitted On,

October 10, 2018

CONTENTS

1. Abstract

1.1 Problem Statement:

To elevate the enrollment rate of a campaign (term deposit), understanding the clients and their behavior (from their data) plays a significant role. The goal of this project is – Given a client's attributes*, predict whether or not they end up subscribing for a term deposit*.

1.2 Client:

The data is related with direct marketing campaigns (phone calls), of a Portuguese banking institution (name of the firm has been anonymized, for confidentiality reasons).

1.3 Dataset:

This dataset is collected from University of California, Irvine – Machine Learning Repository.

Bank Client Data	
1	age (numeric)
2	job : type of job (categorical: 'admin.','blue collar','entrepreneur','housemaid','management','retired','self employed','services','student','technician','unemployed','unknown')
3	marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)
4	education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')
5	default: has credit in default? (categorical: 'no','yes','unknown')
6	housing: has housing loan? (categorical: 'no','yes','unknown')
7	loan: has personal loan? (categorical: 'no','yes','unknown')
Related with the last contact of the current campaign:	
8	contact: contact communication type (categorical: 'cellular','telephone')
9	month: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')
10	day_of_week: last contact day of the week (categorical: 'mon','tue','wed','thu','fri')

11	duration: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.
Other attributes	
12	campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
13	pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
14	previous: number of contacts performed before this campaign and for this client (numeric)
15	poutcome: outcome of the previous marketing campaign (categorical: 'failure','nonexistent','success')
Social and Economic context attributes	
16	emp.var.rate: employment variation rate
17	cons.price.idx: consumer price index
18	cons.conf.idx: consumer confidence index
19	euribor3m: euribor 3 month rate
20	nr.employed: number of employees
Output variable (desired target):	
21	y has the client subscribed a term deposit? (binary: 'yes','no')

2. Data Inspection

df.shape

```
df.shape
```

```
(41188, 21)
```

Interpretation: There are 41188 rows and 21 features.

df.info()

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 41188 entries, 0 to 32949
Data columns (total 21 columns):
age             41188 non-null int64
job             41188 non-null object
marital         41188 non-null object
education       41188 non-null object
default         41188 non-null object
housing         41188 non-null object
personal        41188 non-null object
contact_type    41188 non-null object
month           41188 non-null object
day             41188 non-null object
duration        41188 non-null int64
dcontacts       41188 non-null int64
pdays           41188 non-null int64
pcontacts       41188 non-null int64
poutcome        41188 non-null object
evr             41188 non-null float64
cpi             41188 non-null float64
cci             41188 non-null float64
euribor         41188 non-null float64
employees       41188 non-null float64
y               41188 non-null object
dtypes: float64(5), int64(5), object(11)
memory usage: 6.9+ MB
```

Interpretation: The dataset does not need any type casting operations since all the numeric, decimal and string attributes are have their respective data structures.

df.isnull():

```
df.isnull().any()
```

```
age             False
job             False
marital         False
education       False
default         False
housing         False
personal        False
contact_type    False
month           False
day             False
duration        False
dcontacts       False
pdays           False
pcontacts       False
poutcome        False
evr             False
cpi             False
cci             False
euribor         False
employees       False
y               False
dtype: bool
```

Interpretation: Upon inspecting the dataframe, it is apparent that there are no null values.

df.astype('object').describe().transpose():

```
df.astype('object').describe().transpose()
```

	count	unique	top	freq
age	41188	78	31	1947
job	41188	12	admin.	10422
marital	41188	4	married	24928
education	41188	8	university.degree	12168
default	41188	3	no	32588
housing	41188	3	yes	21576
personal	41188	3	no	33950
contact_type	41188	2	cellular	26144
month	41188	10	may	13769
day	41188	5	thu	8623
duration	41188	1544	90	170
dcontacts	41188	42	1	17642
pdays	41188	27	999	39673
pcontacts	41188	8	0	35563
poutcome	41188	3	nonexistent	35563
evr	41188	10	1.4	16234
cpi	41188	26	93.994	7763
cci	41188	26	-36.4	7763
euribor	41188	316	4.857	2868
employees	41188	11	5228.1	16234
y	41188	2	no	36548

Interpretation: Using the below code snippet, I can get an idea of most frequently occurring values in an attribute as well as their respective frequencies.

df.response_variable.value_counts():

The number of positive responses (yes) is largely fewer than the negative responses (no) implying that the dataset is significantly imbalanced.

```
df['y'].value_counts()
```

```
no      36548
yes      4640
Name: y, dtype: int64
```

Interpretation: Business problems in financial, banking and healthcare industries often have datasets that are massively imbalanced. Considering the reality surrounding these problems, addressing the class balance anomaly is not a major priority, for now. However, later in this report, I use 'upsampling' and 'downsampling' to address class imbalance.

3. Data Pre-Processing

df.columns

```
In [3]: # Columns in the data
        df.columns

Out[3]: Index(['age', 'job', 'marital', 'education', 'default', 'housing', 'loan',
               'contact', 'month', 'day_of_week', 'duration', 'campaign', 'pdays',
               'previous', 'poutcome', 'emp.var.rate', 'cons.price.idx',
               'cons.conf.idx', 'euribor3m', 'nr.employed', 'y'],
              dtype='object')
```

Following code-snippet demonstrates the updated column names.

```
In [6]: # Renaming columns
        df.columns = ['age', 'job', 'marital', 'education', 'default', 'housing', 'personal', 'contact', 'month
        ', 'day_of_week',
                      'duration', 'campaign', 'pdays', 'pcontacts', 'poutcome', 'evr', 'cpi', 'cci', 'euribor',
        'employees', 'y']
```

Interpretation: The attributes (column names) by default are self-explanatory. However, some of these are renamed to make it less confusing.

4 - EDA

It's always a good idea to get an idea about your data before creating an abstract of your model. Exploratory Data Analysis helps us with this. There are two goals of EDA:

1. Explain
2. Explore

df.describe()

Using describe() on the dataframe, Python returns the summary statistics of all the quantitative(numeric) variables.

```
df.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	max
age	41188.0	40.024060	10.421250	17.000	32.000	38.000	47.000	98.000
duration	41188.0	258.285010	259.279249	0.000	102.000	180.000	319.000	4918.000
dcontacts	41188.0	2.567593	2.770014	1.000	1.000	2.000	3.000	56.000
pdays	41188.0	962.475454	186.910907	0.000	999.000	999.000	999.000	999.000
pcontacts	41188.0	0.172963	0.494901	0.000	0.000	0.000	0.000	7.000
evr	41188.0	0.081886	1.570960	-3.400	-1.800	1.100	1.400	1.400
cpi	41188.0	93.575664	0.578840	92.201	93.075	93.749	93.994	94.767
cci	41188.0	-40.502600	4.628198	-50.800	-42.700	-41.800	-36.400	-26.900
euribor	41188.0	3.621291	1.734447	0.634	1.344	4.857	4.961	5.045
employees	41188.0	5167.035911	72.251528	4963.600	5099.100	5191.000	5228.100	5228.100

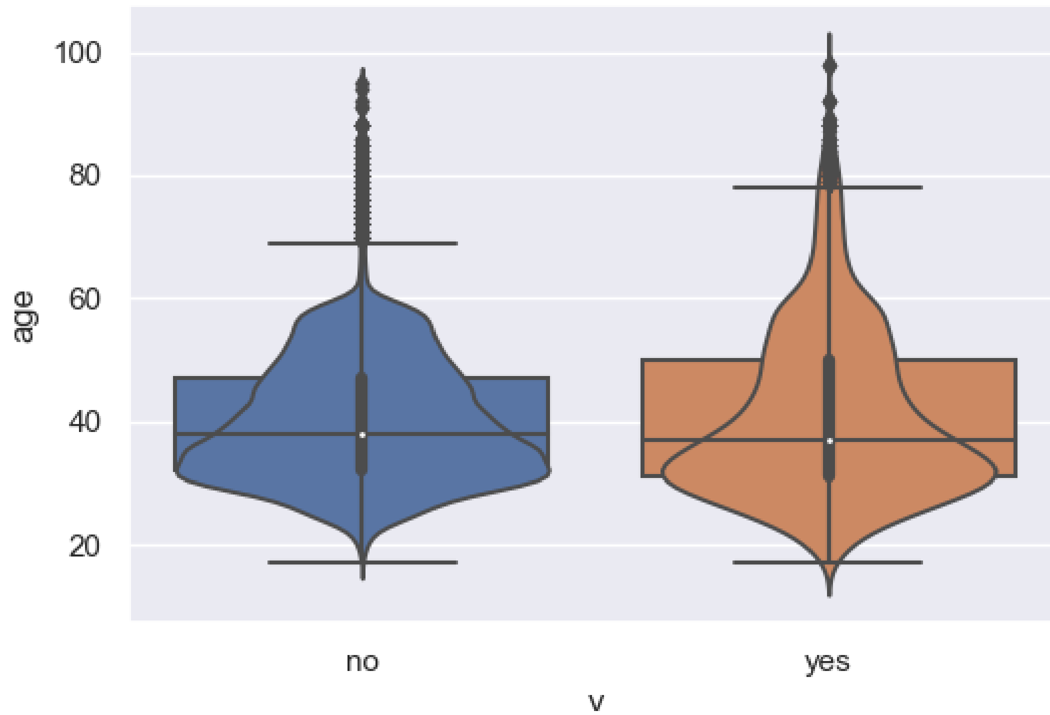
The standard deviations of 'duration', 'pdays' and 'employees' are tremendously large compared to that of other variables. These variables should be investigated to understand the underlying reason for this variability of the values.

As part of Graphical EDA, I plot two graphs for each numeric variable

- Histograms - to understand the distribution underlying the data
- Violin plot - to understand distribution of a variable with respect to the classes of the response variable)

4.1 EDA with Numeric Variables:

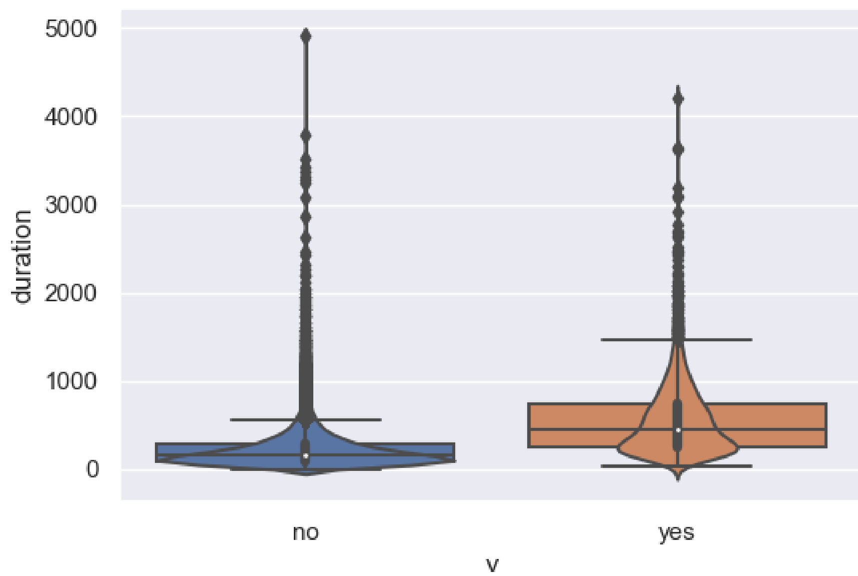
‘age’:



Interpretation: The variance of <u>age</u> of the customers who have rejected the offer is lower compared to that of the customers who have responded positively to the offer. Even though most observations are around early 30's, the mean has been recorded around late 30's for both the classes.

There are significant number of outliers for both classes. However, the outliers for 'no' are widespread. Binning the 'age' variable with respect to 'job' category might provide us better insights.

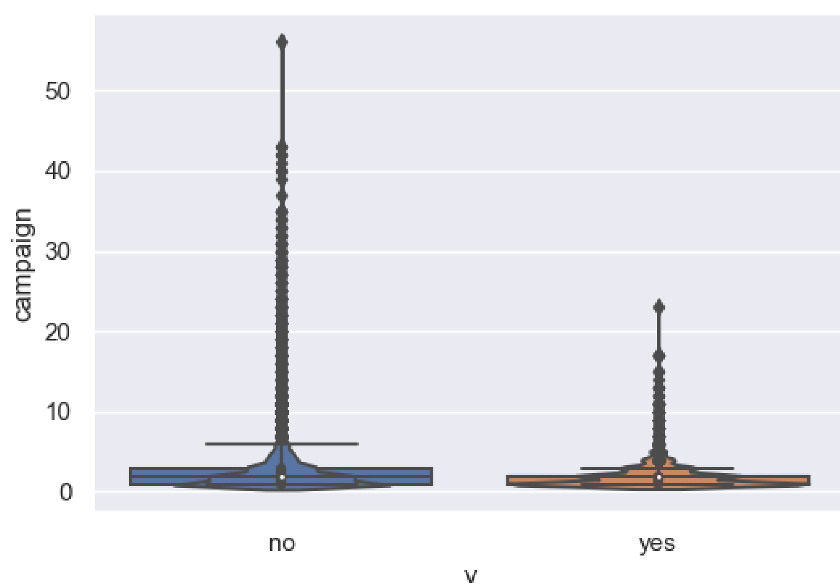
‘duration’:



Interpretation: The variance of class 'no' of the response variable is less compared to that of 'yes' class. Outliers for 'no' are widespread than the outliers of 'yes'.

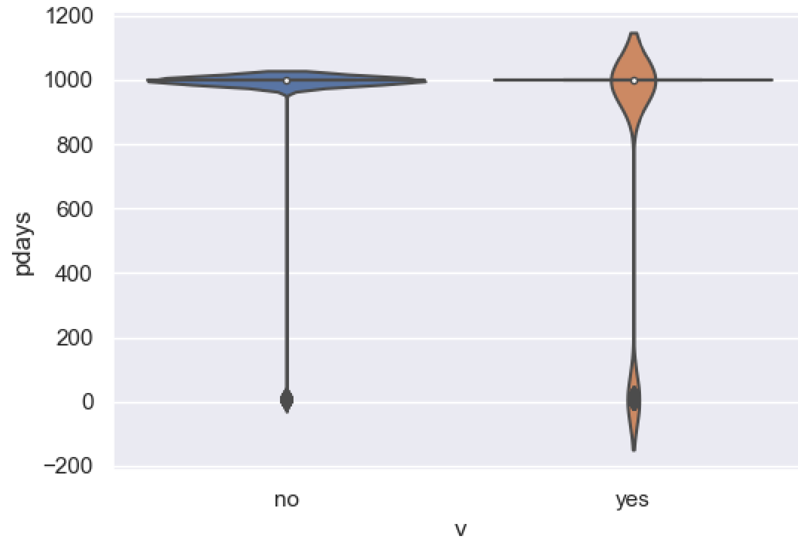
Since the data is widespread, it's a good idea to bin them and include upper bounds.

‘campaign’:



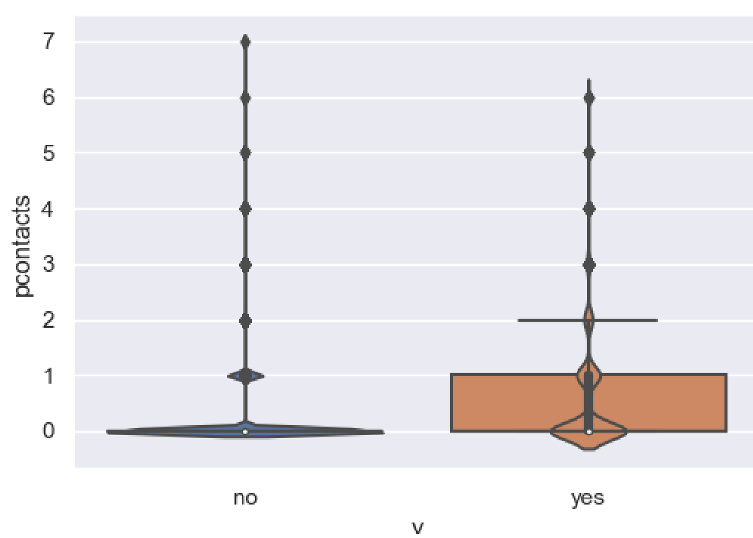
Interpretation: 95% of the data points are almost equally distributed for both the classes. However, the outliers are more loosely distributed for 'no' class than 'yes' class.

‘pdays’:



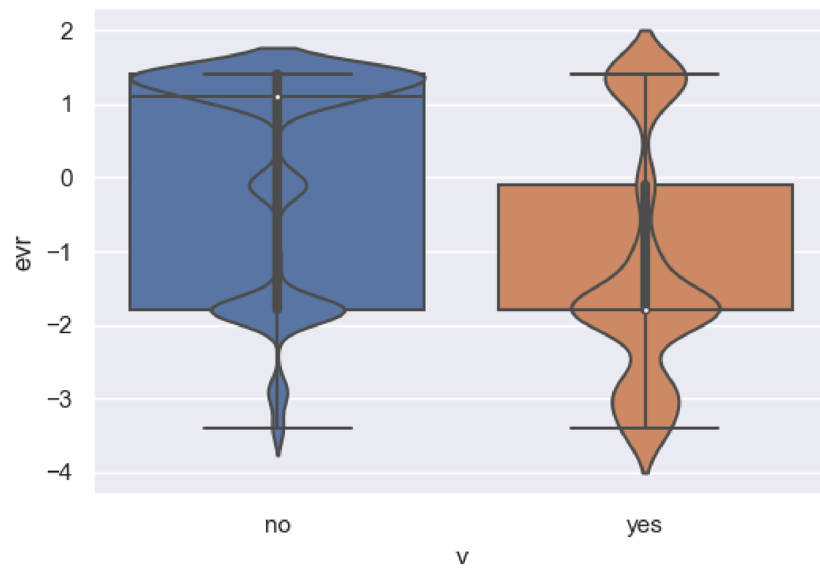
Interpretation: According to the data description (Section I), value ‘999’ denotes that the customer has not been contacted, it looks like majority of the customers from 'no' class have not been previously contacted. Same is the case for most of the customers in 'yes' class. However, a decent amount of customers have been contacted before this campaign which can be seen in the first peak of this bi-modal distribution.

‘pcontacts’:



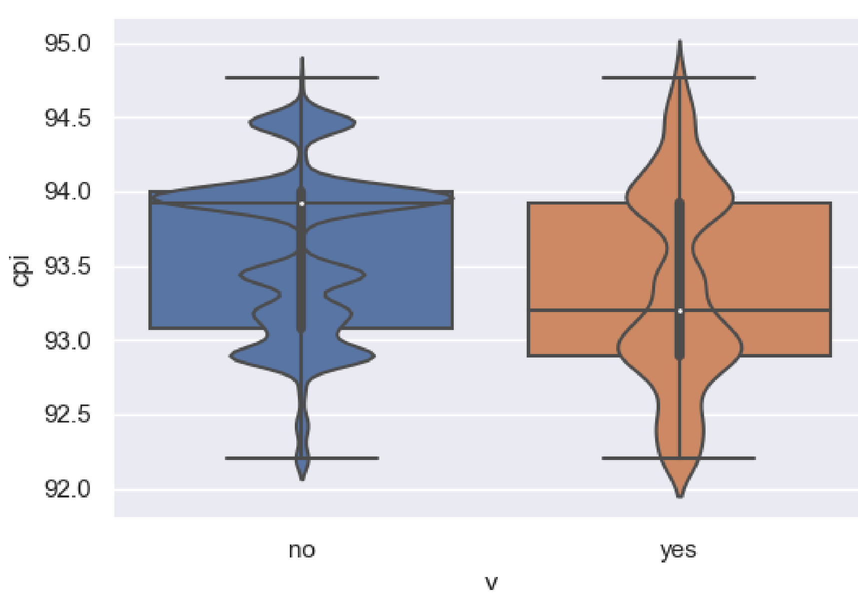
Interpretation: The 'no' class has a bi-modal while 'yes' class has a tri-modal distribution. There are few outliers for both classes which do not have significant effect on majority of the data.

‘evr’:



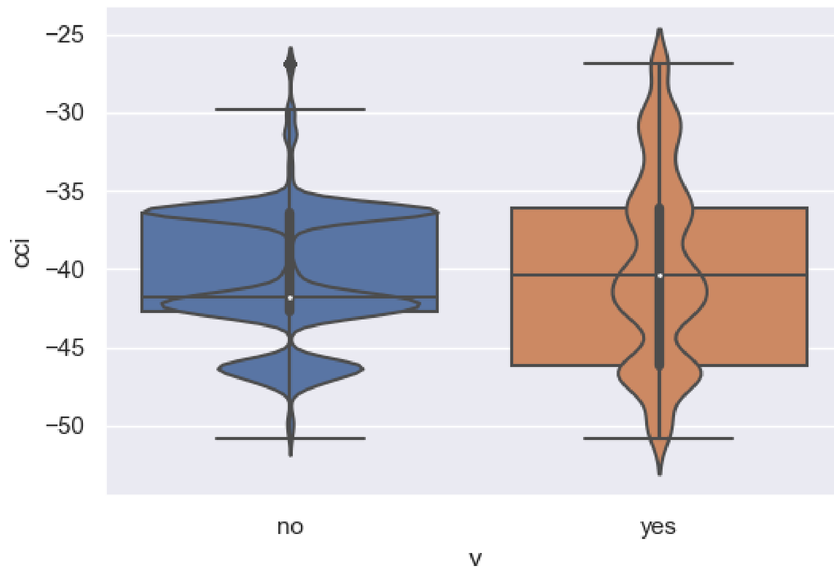
Interpretation: Most observations of EVR rate have been recorded between -2 and 2 for both classes. However, the observations of 'yes' class have recorded a significant amount of them with values between -4 and -2.

‘cpi’:



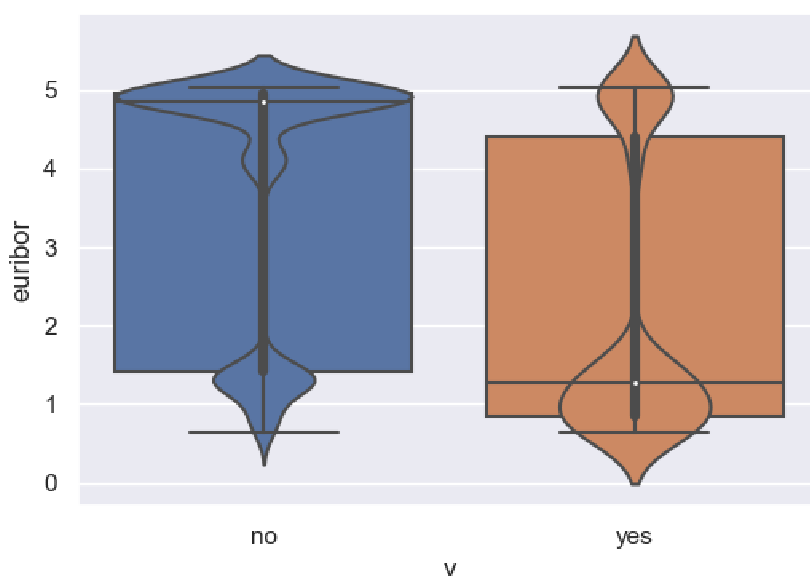
Interpretation: 80% of the observations have been recorded between 93 and 94. However, the emans for 'no' class is close to 94 while for 'yes' class is close to 93.

‘cci’:



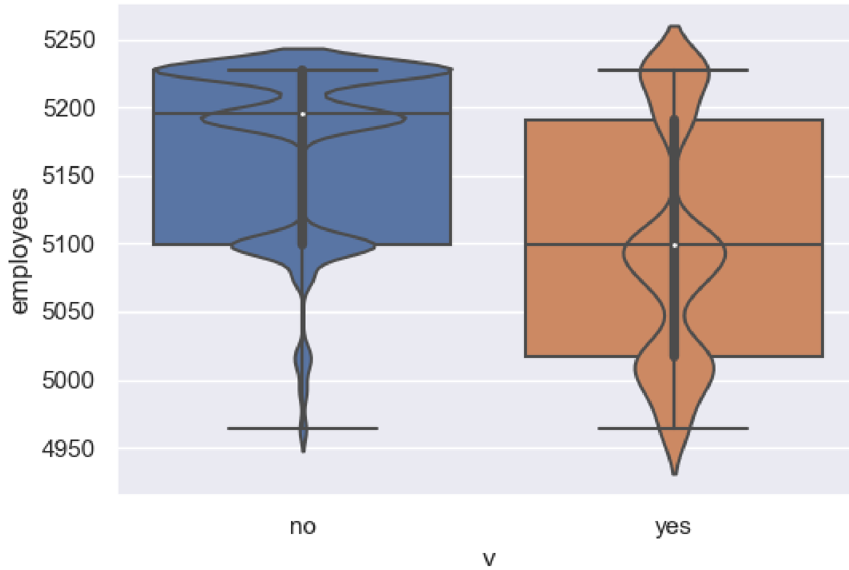
Interpretation: The means for both classes are almost equal. However, 'no' class has observations densely at -36 and -43 while 'yes' class has observations significantly spread from -52 to -30 in its multi-modal distribution.

‘euribor’:



Interpretation: Euribor mean for observations of 'yes' class is around 5 while that of 'no' class is close to 1. However, a decent number of observations have been recorded around 5 as well as 1 for both the classes.

‘employees’:



Interpretation: The mean number of employees for 'no' class is close to 5200 while the mean number of employees for 'yes' class is around 5100. Data is more evenly distributed for class 'yes' with most observations recorded less than 5150.

4.2 Socio-Economic Factors Definitions:

Evr - Employment Variation (EVR) is essentially the variation of how many people are being hired or fired due to the shifts in the conditions of the economy

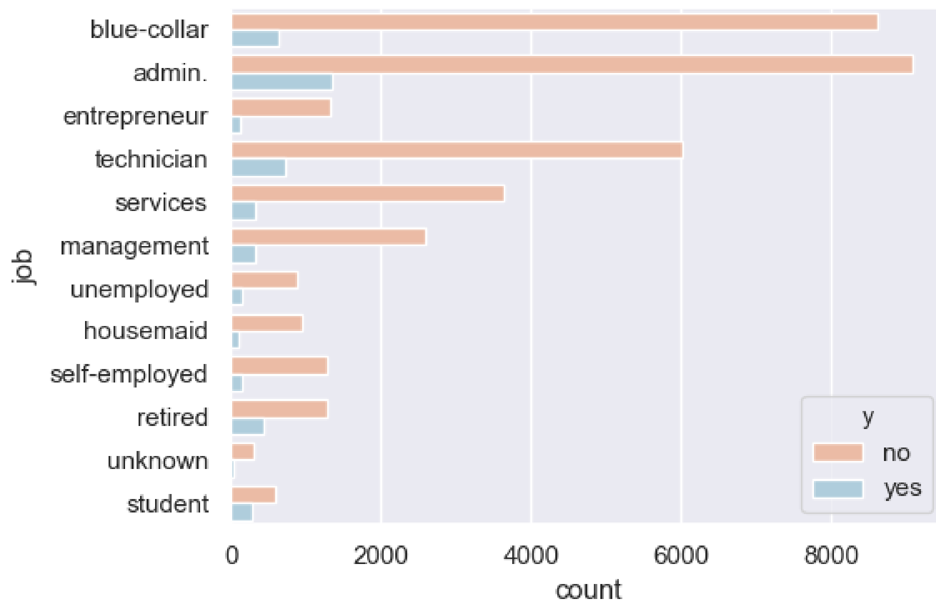
Cpi - The Consumer Price Index (CPI) is a measure of the average change over time in the prices paid by urban consumers for a market basket of consumer goods and services.

Cci - The Commodity Channel Index (CCI) measures the current price level relative to an average price level over a given period of time. CCI is relatively high when prices are far above their average.

Euribor - Euribor is short for Euro Interbank Offered Rate. The Euribor rates are based on the interest rates at which a panel of European banks borrow funds from one another.

4.2 EDA on Categorical Variables:

‘job’:



Interpretation: Admin category has the highest number of positive and negative responses while 'unknown' has the lowest for the both.

```
pd.crosstab(df.job, df.y, normalize='columns').transpose()
```

job y	admin.	blue-collar	entrepreneur	housemaid	management	retired	self-employed	services	student	technician	unemployed	unknown
no	0.248167	0.235745	0.036445	0.026103	0.07103	0.035187	0.034804	0.099759	0.016417	0.164523	0.023804	0.008017
yes	0.291379	0.137500	0.026724	0.022845	0.07069	0.093534	0.032112	0.069612	0.059267	0.157328	0.031034	0.007974

Interpretation: At category-level, Admin, Blue-Collar and Technicians contributed the highest percentage of positive response rate.

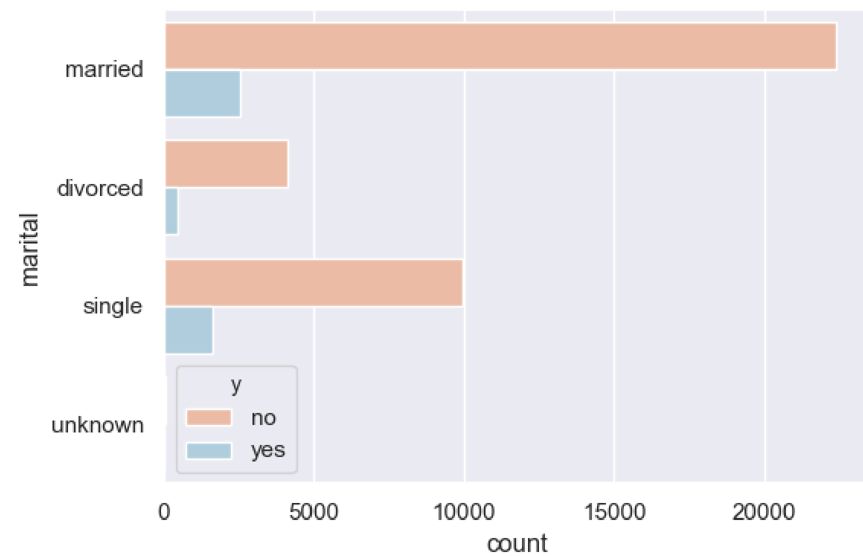
```
pd.crosstab(df.job, df.y, normalize='index').transpose()
```

job y	admin.	blue-collar	entrepreneur	housemaid	management	retired	self-employed	services	student	technician	unemployed	unknown
no	0.870274	0.931057	0.914835	0.9	0.887825	0.747674	0.895144	0.918619	0.685714	0.89174	0.857988	0.887879
yes	0.129726	0.068943	0.085165	0.1	0.112175	0.252326	0.104856	0.081381	0.314286	0.10826	0.142012	0.112121

Interpretation: At class-level, ‘blue-collar’ and ‘entrepreneur’ (6% and 8%) had the lowest positive response rate while retired and students had high positive response rate (25% and 31%).

This means that ‘admin’ and ‘blue-collar’ jobs were contacted frequently than any other job. However, the highest positive response rate, is among ‘retired’ and ‘student’, rather not ‘blue-collar’ and ‘entrepreneur’.

‘marital’:



Interpretation: Married category has the highest number of positive responses. Around 60% of the people considered for this survey belong to either 'married' category.

```
pd.crosstab(df.marital, df.y, normalize='index')
```

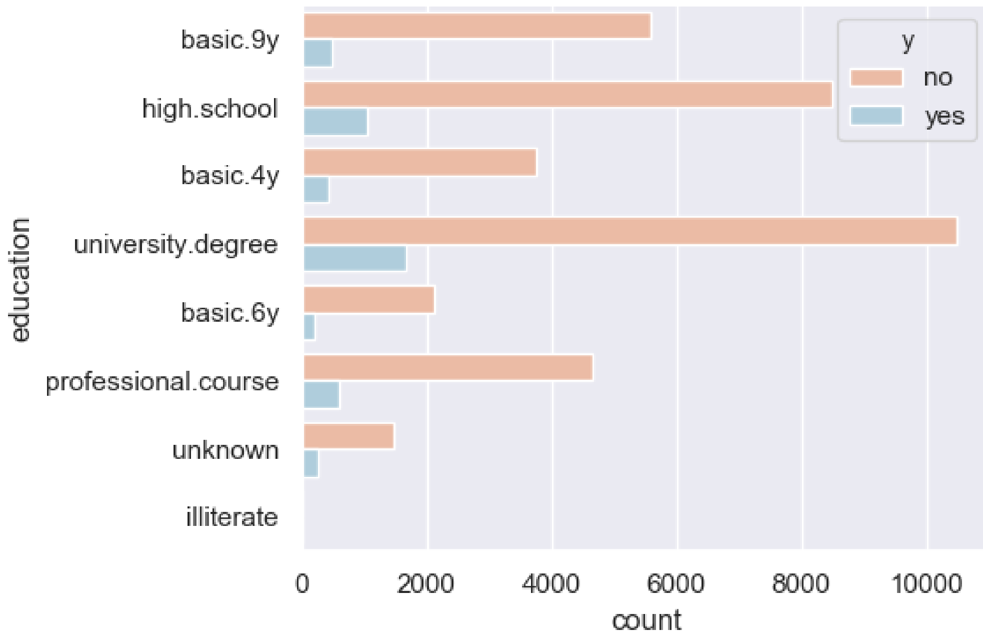
y	no	yes
marital		
divorced	0.893209	0.106791
married	0.897718	0.102282
single	0.859706	0.140294
unknown	0.838710	0.161290

```
pd.crosstab(df.marital, df.y, normalize='columns')
```

y	no	yes
marital		
divorced	0.111682	0.104222
married	0.614215	0.546232
single	0.272323	0.346873
unknown	0.001780	0.002672

Interpretation: At class-level, 'unknown' has the highest positive response rate. At category-level, with 55% 'married' contributed the highest percentage of positive response rate.

'education':



Interpretation: Around 5% of the datapoints has an education level of either 'illiterate' or 'unknown'. Though 'illiterate' and 'unknown' contribute to only 5% of the total datapoints, they have the highest positive response rate within categories.

```
pd.crosstab(df.education, df.y, normalize='index').transpose()
```

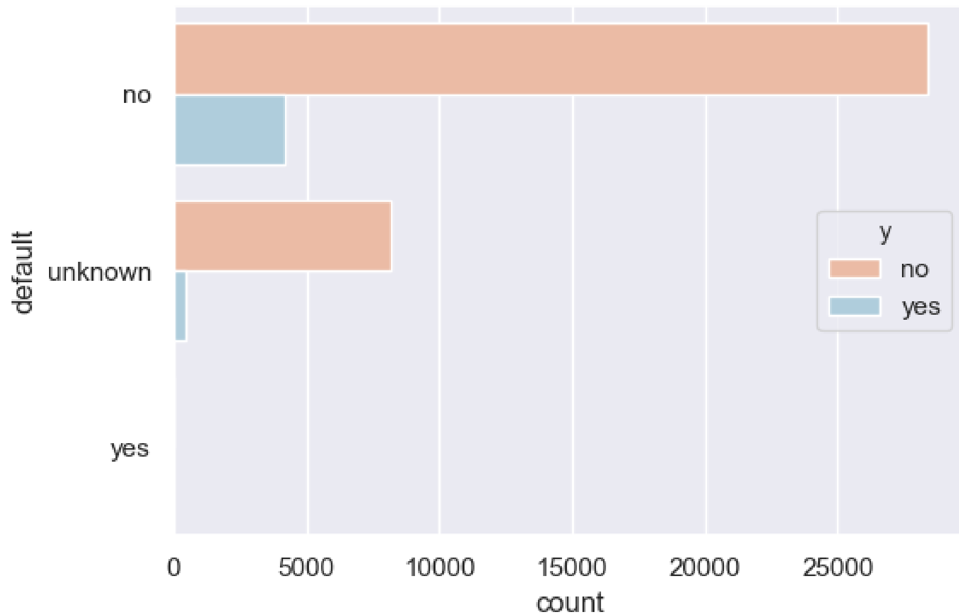
education y	basic.4y	basic.6y	basic.9y	high.school	illiterate	professional.course	university.degree	unknown
no	0.89751	0.917976	0.921754	0.891645	0.777778	0.886515	0.862755	0.854997
yes	0.10249	0.082024	0.078246	0.108355	0.222222	0.113485	0.137245	0.145003

```
pd.crosstab(df.education, df.y, normalize='columns').transpose()
```

education y	basic.4y	basic.6y	basic.9y	high.school	illiterate	professional.course	university.degree	unknown
no	0.102550	0.057568	0.152457	0.232133	0.000383	0.127175	0.287239	0.040495
yes	0.092241	0.040517	0.101940	0.222198	0.000862	0.128233	0.359914	0.054095

Interpretation: The distinction between the education descriptions is very minimal which makes it hard to combine similar classes in the category.

'default':



Interpretation: Most number of people who were approached do not have a default. Almost 20% of the datapoints are 'unknown'.

```
pd.crosstab(df.default, df.y, normalize='index').transpose()
```

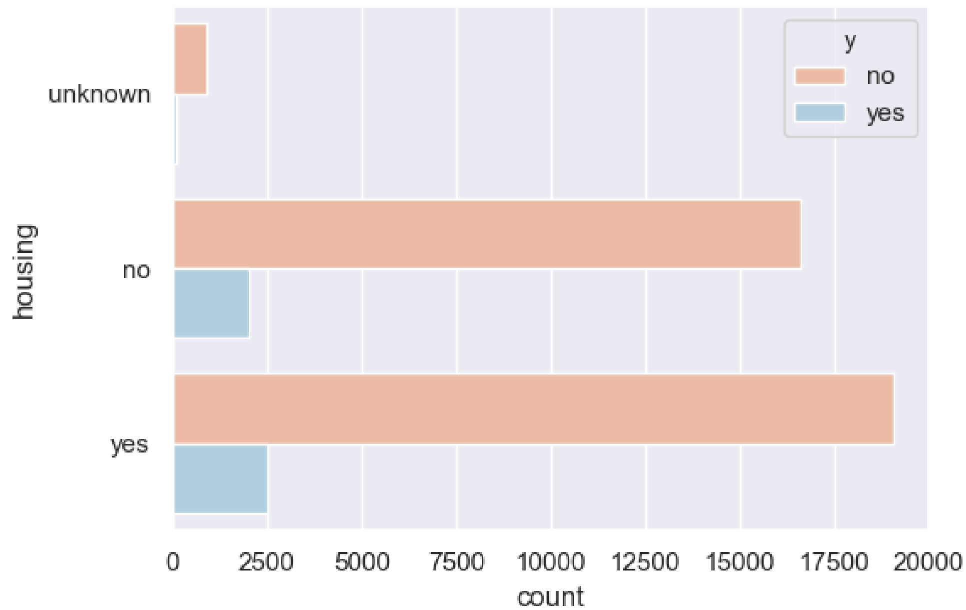
default	no	unknown	yes
y			
no	0.87121	0.94847	1.0
yes	0.12879	0.05153	0.0

```
pd.crosstab(df.default, df.y, normalize='columns').transpose()
```

default	no	unknown	yes
y			
no	0.776814	0.223104	0.000082
yes	0.904526	0.095474	0.000000

Interpretation: None of the customers with loan default have responded positively to the offer.

‘housing’:



Interpretation: Two major classes (people with/without housing loan) are almost equally distributed.

```
pd.crosstab(df.housing, df.y, normalize='index')
```

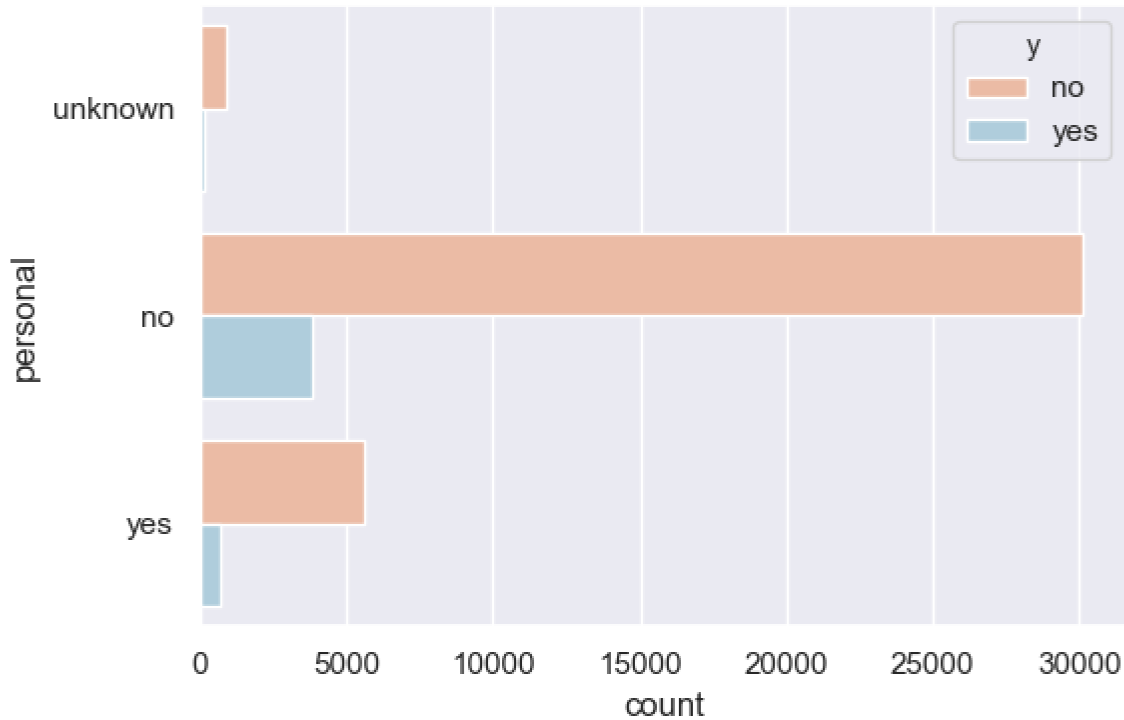
y	no	yes
housing		
no	0.891204	0.108796
unknown	0.891919	0.108081
yes	0.883806	0.116194

```
pd.crosstab(df.housing, df.y, normalize='columns')
```

y	no	yes
housing		
no	0.454088	0.436638
unknown	0.024160	0.023060
yes	0.521752	0.540302

Interpretation: Customers with no housing loan have less positive response rate compared to the ones with housing loan.

‘personal’:



Interpretation: Around 82% of datapoints do not have a personal loan on them.

```
pd.crosstab(df.personal, df.y, normalize='index')
```

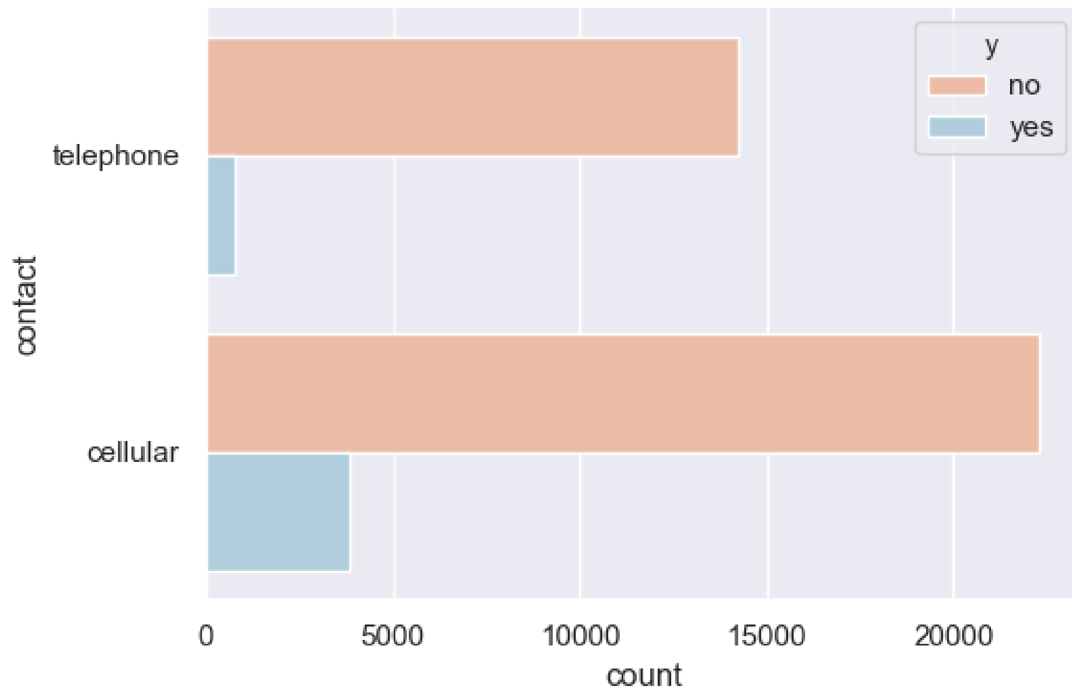
y	no	yes
personal		
no	0.886598	0.113402
unknown	0.891919	0.108081
yes	0.890685	0.109315

```
pd.crosstab(df.personal, df.y, normalize='columns')
```

y	no	yes
personal		
no	0.823574	0.829741
unknown	0.024160	0.023060
yes	0.152266	0.147198

Interpretation: Within each category, each of them contributed to the same percentage for a positive response.

‘contact’:



Interpretation: Customers with cellular phone are more likely to respond positively.

```
pd.crosstab(df.contact, df.y, normalize='index')
```

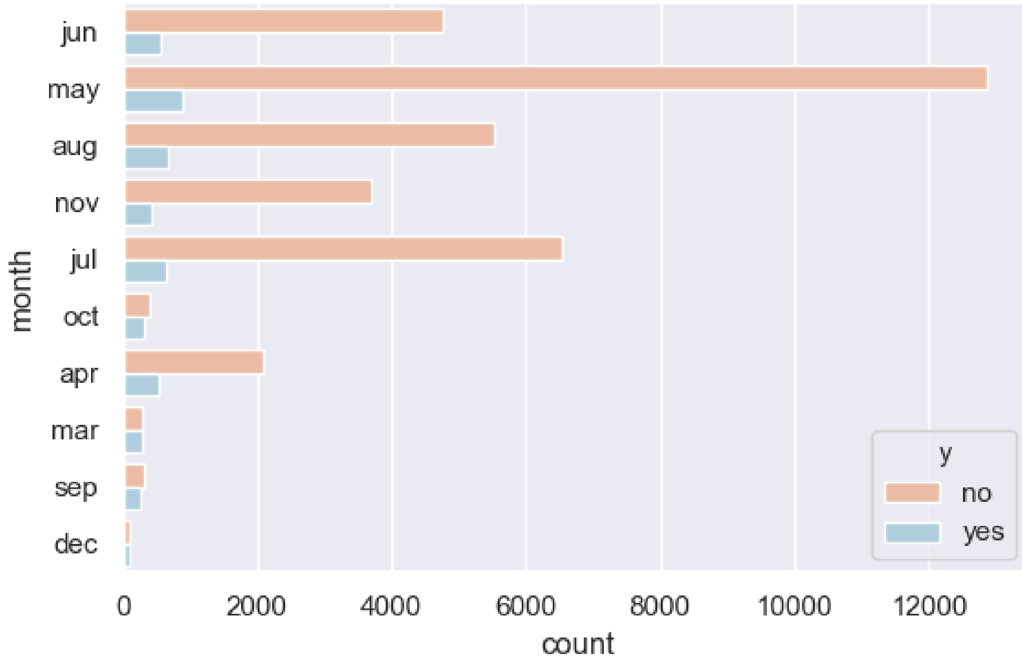
y contact	no	yes
cellular	0.852624	0.147376
telephone	0.947687	0.052313

```
pd.crosstab(df.contact, df.y, normalize='columns')
```

y contact	no	yes
cellular	0.60991	0.830388
telephone	0.39009	0.169612

Interpretation: At category-level, customers with telephone contributed least to the positive response rate.

'month':



Interpretation: Most customers were contacted during the second quarter of the calendar year.

```
pd.crosstab(df.month, df.y, normalize='index').transpose()
```

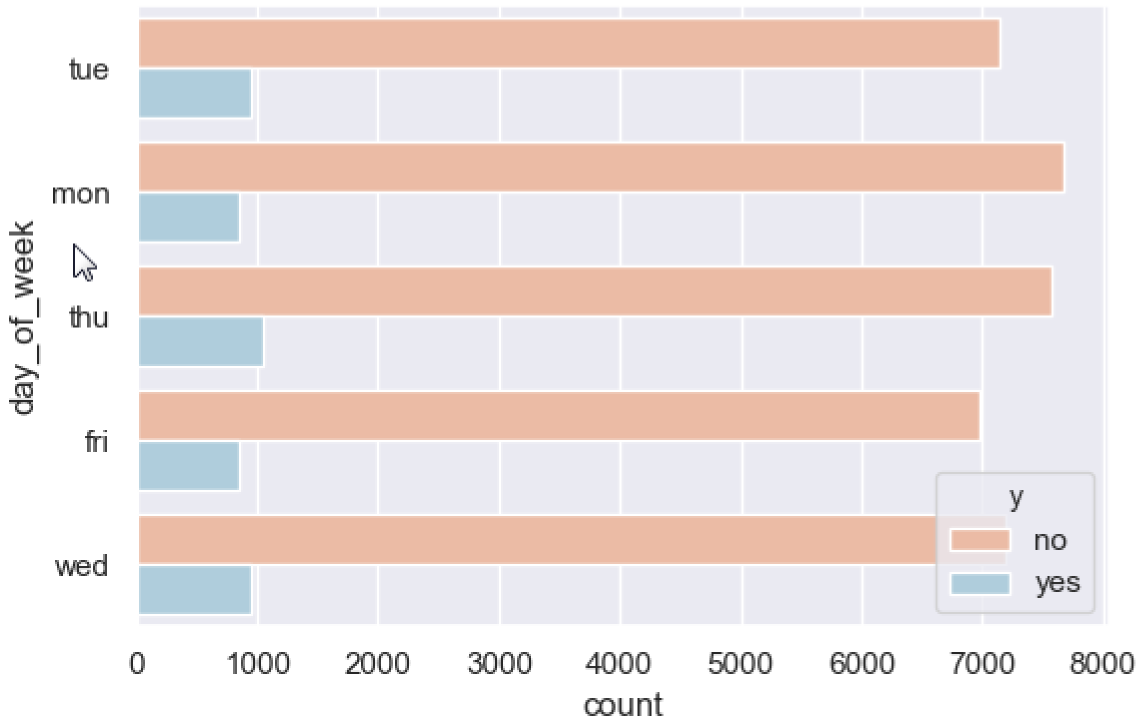
month y	apr	aug	dec	jul	jun	mar	may	nov	oct	sep
no	0.795213	0.893979	0.510989	0.909534	0.894885	0.494505	0.935653	0.898561	0.561281	0.550877
yes	0.204787	0.106021	0.489011	0.090466	0.105115	0.505495	0.064347	0.101439	0.438719	0.449123

```
pd.crosstab(df.month, df.y, normalize='columns').transpose()
```

month y	apr	aug	dec	jul	jun	mar	may	nov	oct	sep
no	0.057267	0.151116	0.002545	0.178532	0.130212	0.007388	0.352495	0.100826	0.011027	0.008591
yes	0.116164	0.141164	0.019181	0.139871	0.120474	0.059483	0.190948	0.089655	0.067888	0.055172

Interpretation: Highest positive response percentage was recorded in March, September and December. Highest percentage of positive response was recorded in the final quarter while the lowest percentage was recorded in 'November' which also fall in the final quarter of the year.

'day':



Interpretation: All classes are almost equally distributed both in terms of numbers.

```
pd.crosstab(df.day_of_week, df.y, normalize='index').transpose()
```

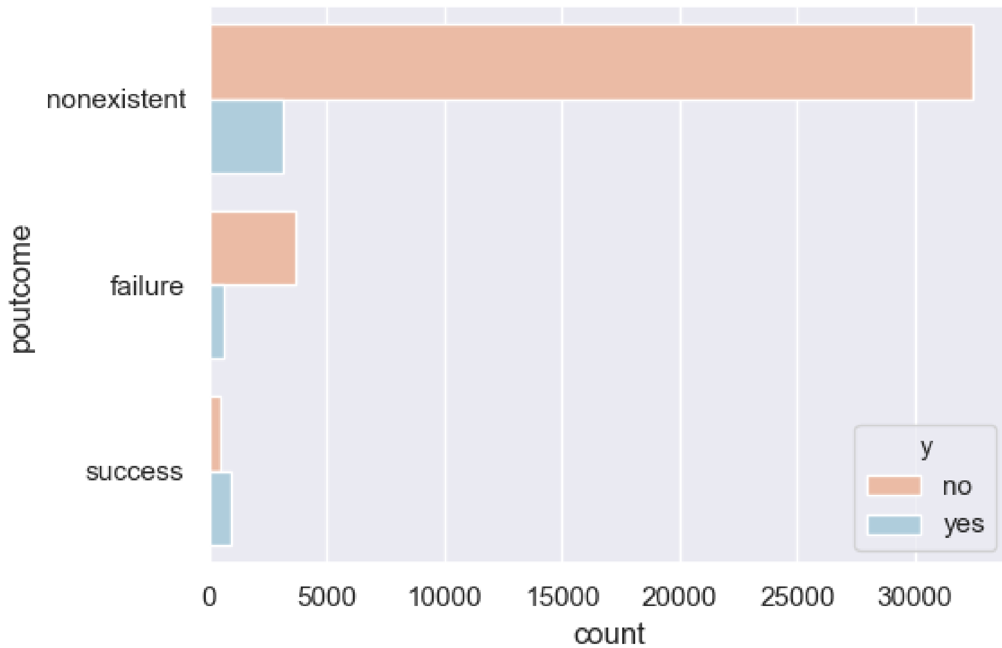
day_of_week	fri	mon	thu	tue	wed
y					
no	0.891913	0.900517	0.878812	0.8822	0.883329
yes	0.108087	0.099483	0.121188	0.1178	0.116671

```
pd.crosstab(df.day_of_week, df.y, normalize='columns').transpose()
```

day_of_week	fri	mon	thu	tue	wed
y					
no	0.191009	0.209779	0.207344	0.195277	0.196591
yes	0.182328	0.182543	0.225216	0.205388	0.204526

Interpretation: All classes are almost equally distributed both in terms of percentages as well.

‘poutcome’:



Interpretation: Most of the customers have not been contacted for the previous campaign.

```
pd.crosstab(df.poutcome, df.y, normalize='index')
```

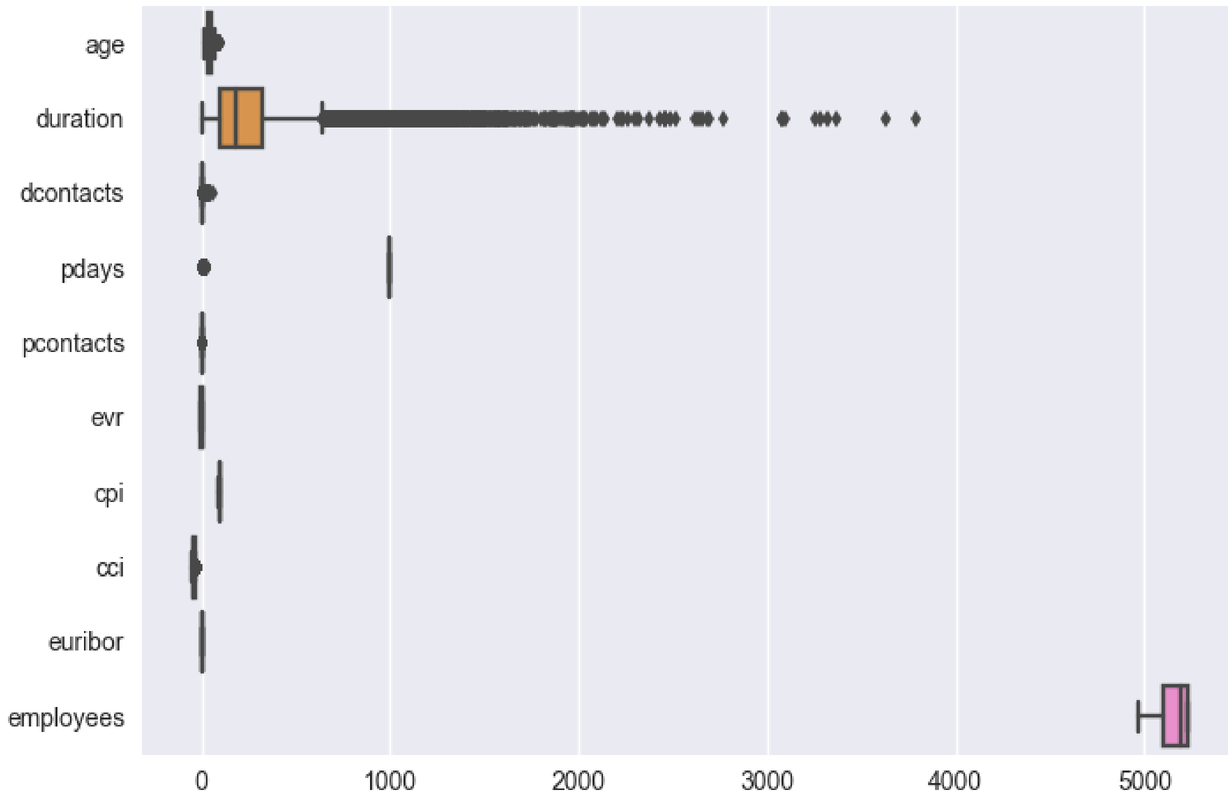
y	no	yes
poutcome		
failure	0.857312	0.142688
nonexistent	0.910610	0.089390
success	0.341644	0.658356

```
pd.crosstab(df.poutcome, df.y, normalize='columns')
```

y	no	yes
poutcome		
failure	0.099356	0.129075
nonexistent	0.887976	0.680385
success	0.012668	0.190540

Interpretation: Of the customers who have not accepted the previous offer, only 14% of them have accepted the current offer. About 65% of customers who have accepted the offer previously, have also accepted current offer.

4.3 Box plot of all the numeric variables:



Interpretation: We notice that there is a huge disparity among the scale which requires us to bring all the variables to a common scale.

4.4 Effect of classes of response variable with respect to other variables:

From the problem statement:

Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.

I have created two dataframes, which hold records that have response variable as 'yes' and 'no' separately. Idea is to analyze how 'duration' variable

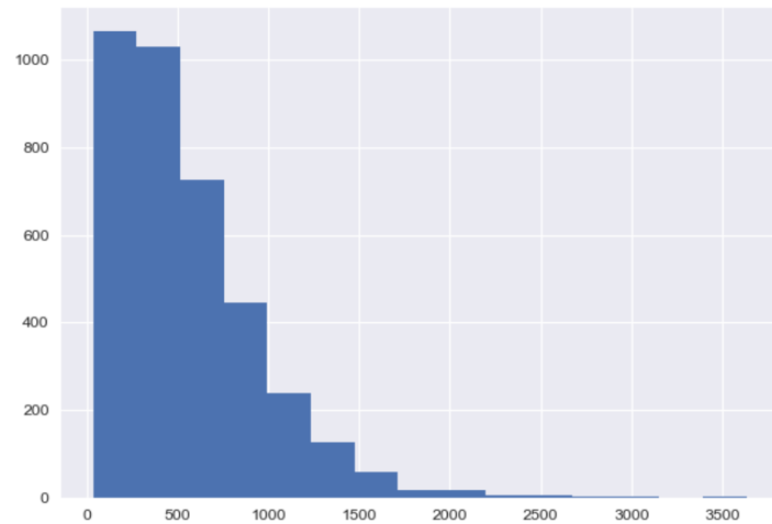
'duration' variable

```
duration_yes = z.loc[z['y']=='yes']
duration_no = z.loc[z['y']=='no']
```

Classes of response variable vs distribution of 'duration' variable:

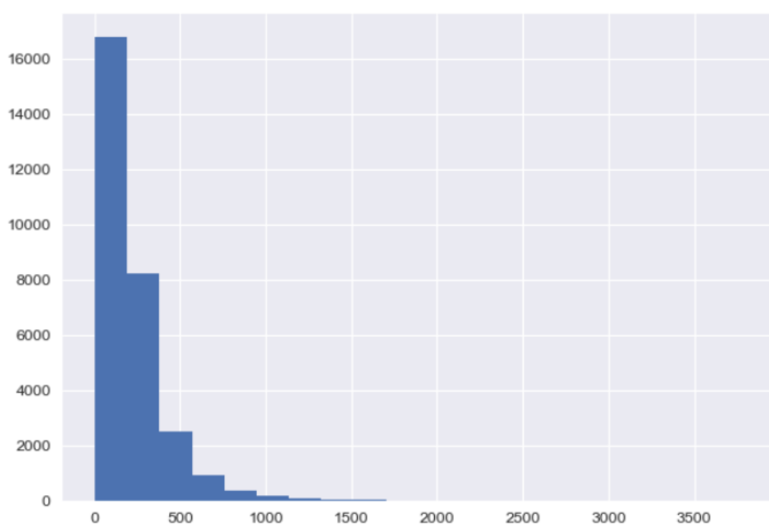
```
duration_yes['duration'].hist(bins=15)
```

```
<IPython.core.display.Javascript object>
```




```
duration_no['duration'].hist(bins=20)
```

```
<IPython.core.display.Javascript object>
```

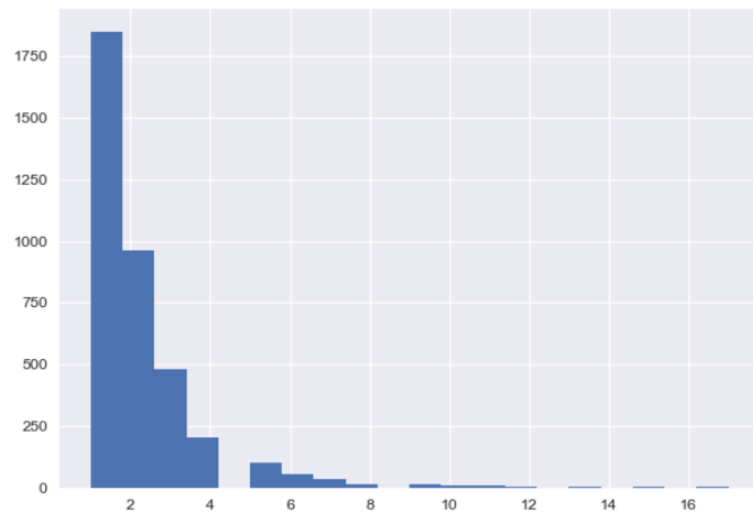


Interpretation: From the above two graphs, it is evident that if the call duration is less than 300 seconds, the customer is more likely to says 'NO' and if the call lasts for more than 300 seconds, the customer is more likely to say 'Yes'

Classes of response variable vs distribution of ‘dcontacts’ variable:

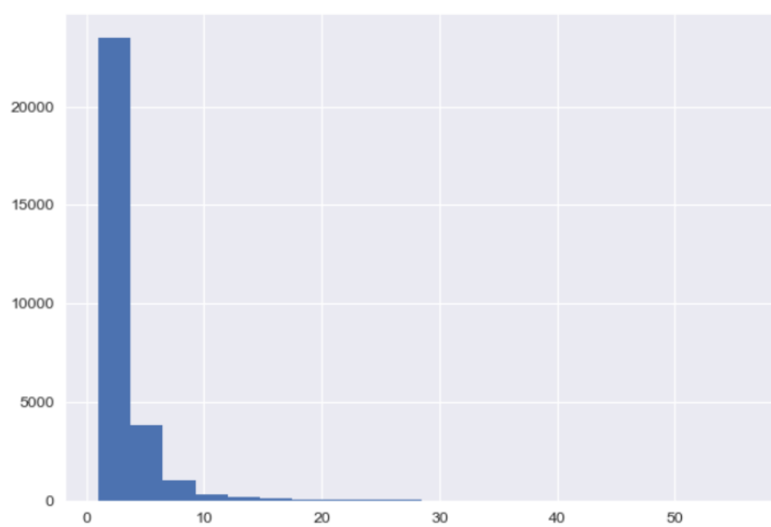
```
# z['dcontacts'].hist(bins=20)
duration_yes['dcontacts'].hist(bins=20)
```

```
<IPython.core.display.Javascript object>
```




```
duration_no['dcontacts'].hist(bins=20)
```

```
<IPython.core.display.Javascript object>
```

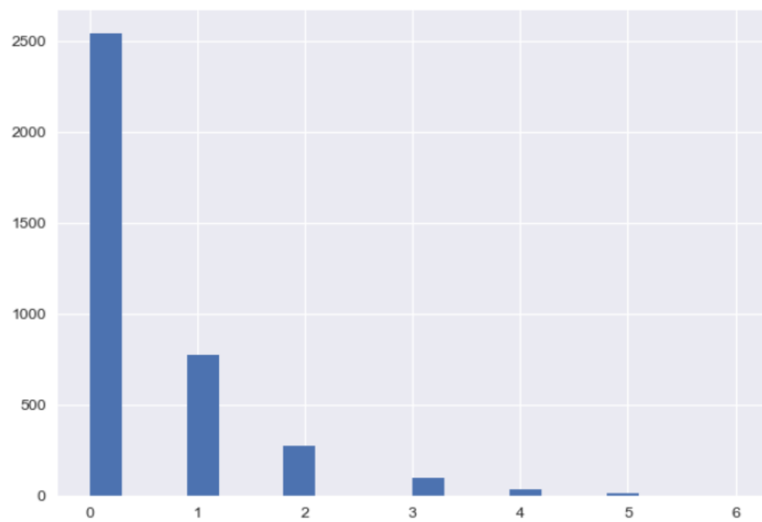


Interpretation: The above two histograms are drawn differently. Look at the size of X-Labels. It looks like both the graphs have the similar behavior. It's hard to differentiate. But we can infer that if a customer is made more than two contacts, they are highly likely to say 'yes' than 'no'

Classes of response variable vs distribution of 'pcontacts' variable:

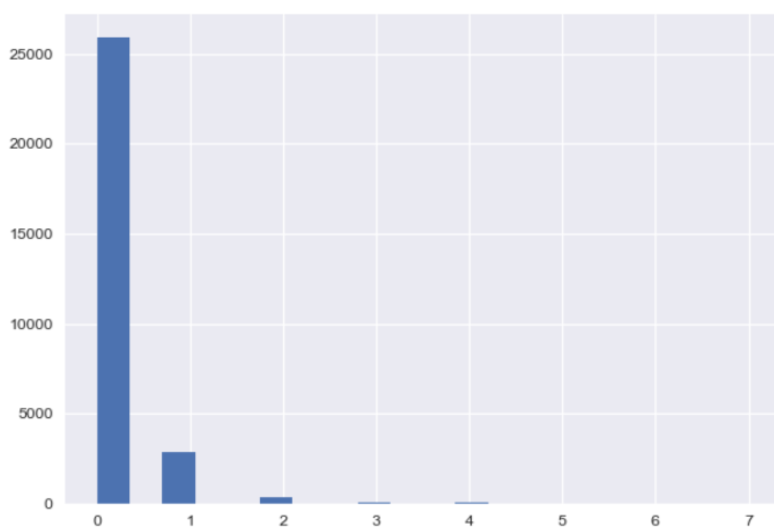
```
duration_yes['pcontacts'].hist(bins=20)
```

<IPython.core.display.Javascript object>




```
duration_no['pcontacts'].hist(bins=20)
```

<IPython.core.display.Javascript object>



Interpretation: The difference is very little between the two dataframes.

5. Feature Engineering

Considering the findings from EDA, the following feature engineering techniques are implemented.

5.1 Consolidate category classes:

I consolidate category classes into various levels based on the percentages of ‘yes’ class.

```
# Consolidate 'job', 'education' and 'month' variables based on percentage of positive and negative res
ponses.
z['job'].replace(['blue-collar', 'services', 'entrepreneur', 'housemaid', 'self-employed', 'technician'
,
                  'management', 'unknown', 'admin.', 'unemployed', 'retired', 'student'],
                   ['jl14', 'jl14', 'jl14', 'jl13', 'jl13', 'jl13', 'jl13', 'jl12', 'jl12', 'jl12', 'jl
l1', 'jl11'],
                   inplace=True)

z['education'].replace(['basic.9y','basic.6y','basic.4y','high.school','professional.course','universit
y.degree','unknown','illiterate'],
                      ['el14','el14','el13','el13','el13','el12','el12','el11'],
                      inplace=True)

z['month'].replace(['may','jul','nov','aug','jun','apr','oct','sep','dec','mar'],
                      ['ml13','ml13','ml13','ml13','ml13','ml12','ml11','ml11','ml11','ml11'],
                      inplace=True)
```

5.2 Binning the age:

```
age_groups = ['young_adult', 'adult', 'senior']
```

```
z['age_group'] = pd.qcut(df['age'], 3, labels = age_groups)
```

Given the data is highly imbalanced, 'age' is categorised into bins based using 'qcut' rather than 'cut'

5.3. Categorize 'day' with 'weekday_1', 'weekday_2' and 'weekend' classes:

```
# Replaced day with 'weekday_1', 'weekday_2' and 'weekend' categories.
for dataframe in (z, df):
    dataframe['day_cat'] = dataframe['day'].copy(deep=True)
    dataframe['day_cat'].replace(['sun', 'sat', 'mon', 'tue', 'wed', 'thu', 'fri'],
                        ['weekend', 'weekend', 'weekday_1', 'weekday_1', 'weekday_1', 'weekday_2', 'weekd
ay_2'],
                        inplace=True)
```

5.4 Merging 'marital' and 'age' variable:

```
z['age_marital'] = z.apply(lambda x: x['age_group'] + ' & ' + x['marital'], axis = 1)
```

```
z['age_marital'].value_counts()
```

```
senior & married          7805
adult & married           7099
young_adult & single      6088
young_adult & married     5080
adult & single            2407
senior & divorced         1796
adult & divorced          1243
senior & single            757
young_adult & divorced     613
young_adult & unknown       28
senior & unknown            22
adult & unknown             12
Name: age_marital, dtype: int64
```

5.5 Inclusion and Exclusion of 'duration' column:

Since this attribute highly affects the output target, I create two dataframes (one with 'duration' column another without).

5.6 Treating Outliers:

Idea:

(a). Replace valid outliers with logarithmic transformation
(b). Replace invalid outliers (human-error) with 90th percentile or upper bounds.

5.6.1. Applying Upper and Lower bounds to 'duration' and 'employees' variable

```
# Upper and Lower bounds for 'duration' column
z['duration'] = z['duration'].apply(lambda x: int(math.floor(x / 10.0)) * 10 if(x%10<5) else int(math.c
eil(x / 10.0)) * 10 )
z['employees'] = z['employees'].apply(lambda x: int(math.floor(x / 10.0)) * 10 if(x%10<5) else int(math
.ceil(x / 10.0)) * 10 )
```

5.6.2. Applying 90 percentiles and 5 percentiles for the lower and upper outliers

```
uq = 0.95
lq = 0.05
```

```
colz = ['duration', 'dcontacts', 'pdays', 'evr', 'cpi', 'cci', 'euribor', 'employees']
```

```
for col in colz:
    z[col] = z[col].clip_upper(int(z[col].quantile(uq)))
    z[col] = z[col].clip_lower(int(z[col].quantile(lq)))
```

5.6.3. Apply Logarithmic transformations to invalid outliers

Creating a new dataframe to apply logarithm transformations. From all the numerical columns, logarithmic transformations is applied to only a few

Excluded attributes

cci - (Negative values)
evr (negative)
pcontacts - (Zeros)
duration – (Zeros)
pdays – (Zeros)
pcontacts – (Zeros)
pdays - (value 999 - means client was not contacted previously)

```
# z.astype(bool).sum(axis=0)      # Count of zeros in a columns
# z[z<0].count()                  # Count of negative values in each column
```

```
# num = ['age','dcontacts','cpi','euribor','employees','duration_outliers','dcontacts_outliers','pdays_
outliers','euribor_outliers','employees_outliers']
num = ['age','dcontacts', 'cpi', 'euribor','employees']
```

```
z_log = z.copy(deep=True)
for n in num:
    z_log[n] = np.log(z_log[n])
```

Above code snippet applies logarithmic transformation to the numerical variables.

6. Ready for Machine Learning

6.1 Standardization and Normalization

Preprocessed data may contain attributes with a mixtures of scales for various quantities such as dollars, kilograms and sales volume. Many machine learning methods expect or are more effective if the data attributes have the same scale.

Two popular data scaling methods are normalization and standardization.

1. Data Normalization
2. Data Standardization

Normalization: It refers to rescaling real valued numeric attributes into the range 0 and 1. It is useful to scale the input attributes for a model that relies on the magnitude of values, such as distance measures used in k-nearest neighbors and in the preparation of coefficients in regression.

ML algorithms such as Linear Regression and SVM perform faster on normalized data.

Standardization: Standardization refers to shifting the distribution of each attribute to have a mean of zero and a standard deviation of one (unit variance). It is useful to standardize attributes for a model that relies on the distribution of attributes such as Gaussian processes.

Which Method to Use: It is hard to know whether rescaling your data will improve the performance of your algorithms before you apply them. If often can, but not always.

A good tip is to create rescaled copies of your dataset and race them against each other using your test harness and a handful of algorithms you want to spot check. This can quickly highlight the benefits (or lack thereof) of rescaling your data with given models, and which rescaling method may be worthy of further investigation.

```
numerical = ['age','duration','dcontacts','pdays','pcontacts','evr','cpi','cci','euribor','employees']
```

```
for dataframe in (z_normalized, df_duration_yes_normalized, df_duration_no_normalized, z_log_normalized
):
    for n in numerical:
        col = dataframe[[n]].values.astype(float)
        col_transformed = (preprocessing.MinMaxScaler()).fit_transform(col)
        dataframe[n+'_normalized'] = pd.DataFrame(col_transformed)

for dataframe in (z_standardized, df_duration_yes_standardized, df_duration_no_standardized, z_log_stan
dardized):
    for n in numerical:
        col = dataframe[[n]].values.astype(float)
        col_transformed = (preprocessing.StandardScaler()).fit_transform(col)
        dataframe[n+'_standardized'] = pd.DataFrame(col_transformed)
```

6.2 Upsampling and Downsampling:

```
# Upsampling Data - z_upsample
major_class = z[z.y == 'no']
minor_class = z[z.y == 'yes']

z_minor_upsample = resample(minor_class, replace = True, n_samples = len(major_class), random_state = 4
2)
z_upsample = pd.concat([major_class, z_minor_upsample])

print(z_upsample.y.value_counts())
```

```
no      29208
yes     29208
Name: y, dtype: int64
```

```
# Downsampling Data - z_downsample
major_class = z[z.y == 'no']
minor_class = z[z.y == 'yes']

z_major_downsample = resample(major_class, replace = False, n_samples = len(minor_class), random_state
= 42)
z_downsample = pd.concat([z_major_downsample, minor_class])

print(z_downsample.y.value_counts())
```

```
yes    3742
no     3742
Name: y, dtype: int64
```

6.3 Dummy Variables:

Since each dataframe has different categorical columns, all dataframes are divided into two lists.

```
# Untransformed dataframes have 'age' column
all_dataframes_1 = [df, df_outliers, df_log, df_outliers_log, df_normalization,  df_log_normalization,
df_standardization, df_log_standardization, df_upsample, df_downsample]
```

```
# Transformed dataframes have 'age_cat' column
all_dataframes_2 = [df_transformations, df_transformations_outliers, df_transformations_outliers_log, d
f_transformations_outliers_normalization, df_transformations_outliers_log_normalization, df_transformat
ions_outliers_standardization, df_transformations_outliers_log_standardization, df_transformations_outl
iers_upsample, df_transformations_outliers_log_upsample, df_transformations_outliers_log_normalization_
upsample, df_transformations_outliers_log_standardization_upsample, df_transformations_outliers_downsam
ple, df_transformations_outliers_log_downsample, df_transformations_outliers_log_normalization_downsamp
le, df_transformations_outliers_log_standardization_downsample]
```

Categorical columns for each dataframe are divided as two different lists so that they can be efficiently looped and dummied.

```
# Categorical fields for respective dataframes
categorical_fields_1 = ['job', 'marital', 'education', 'default', 'housing', 'personal',
       'contact_type', 'month', 'day', 'poutcome']

categorical_fields_2 = ['job', 'marital', 'education', 'default', 'housing', 'personal',
       'contact_type', 'month', 'day', 'poutcome', 'age_cat']
```

The following code snippet does not replace the categorical columns with their respective dummies in place.

```
all_dataframes_1 = [pd.get_dummies(df, columns = categorical_fields_1) for df in all_dataframes_1]

all_dataframes_2 = [pd.get_dummies(df, columns = categorical_fields_2) for df in all_dataframes_2]
```

Hence, I had to dummy each dataframe individually.

```
# Get dummies for all_dataframes_1 with categorical_fields_1
df, df_outliers, df_log, df_outliers_log, df_normalization,  df_log_normalization, \
df_standardization, df_log_standardization, df_upsample, df_downsample \
      = [pd.get_dummies(df, columns=categorical_fields_1) \
         for df in [df, df_outliers, df_log, df_outliers_log, df_normalization,  df_log_normalization,
\
                    df_standardization, df_log_standardization, df_upsample, df_downsample]]
```

```
# Get dummies for all_dataframes_2 with categorical_fields_2
df_transformations, df_transformations_outliers, df_transformations_outliers_log, \
df_transformations_outliers_normalization, df_transformations_outliers_log_normalization, \
df_transformations_outliers_standardization, df_transformations_outliers_log_standardization, \
df_transformations_outliers_upsample, df_transformations_outliers_log_upsample, \
df_transformations_outliers_log_normalization_upsample, df_transformations_outliers_log_standardization
_upsample, \
df_transformations_outliers_downsample, df_transformations_outliers_log_downsample, \
df_transformations_outliers_log_normalization_downsample, df_transformations_outliers_log_standardizati
on_downsample \
      = [pd.get_dummies(df, columns=categorical_fields_2) \
         for df in [df_transformations, df_transformations_outliers, df_transformations_outliers_log, \
                    df_transformations_outliers_normalization, df_transformations_outliers_log_normaliz
ation, \
                    df_transformations_outliers_standardization, df_transformations_outliers_log_standa
rdization, \
                    df_transformations_outliers_upsample, df_transformations_outliers_log_upsample, \
                    df_transformations_outliers_log_normalization_upsample, \
                    df_transformations_outliers_log_standardization_upsample, \
                    df_transformations_outliers_downsample, df_transformations_outliers_log_downsample,
\
                    df_transformations_outliers_log_normalization_downsample, \
                    df_transformations_outliers_log_standardization_downsample]]
```

7. Machine Learning

7.1 Logistic Regression:

All dataframes consolidated into one list.

```
all_dataframes = [df, df_outliers, df_log, df_outliers_log, df_normalization,  df_log_normalization, df
_standardization, df_log_standardization, df_upsample, df_downsample, df_transformations, df_transforma
tions_outliers, df_transformations_outliers_log, df_transformations_outliers_normalization, df_transfor
mations_outliers_log_normalization, df_transformations_outliers_standardization, df_transformations_out
liers_log_standardization, df_transformations_outliers_upsample, df_transformations_outliers_log_upsamp
le, df_transformations_outliers_log_normalization_upsample, df_transformations_outliers_log_standardiza
tion_upsample, df_transformations_outliers_downsample, df_transformations_outliers_log_downsample, df_t
ransformations_outliers_log_normalization_downsample, df_transformations_outliers_log_standardization_d
ownsample]
```

Training all models to find the best model that is more accurate.

```
scores = []
for dframe in all_dataframes:
    X = dframe.drop('y', 1)
    y = pd.DataFrame(dframe[['y']])
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

    reg = LogisticRegression()
    reg.fit(X_train, y_train)
    pred = reg.predict(X_test)
    print(dframe.name, '--', (accuracy_score(y_test, pred)))
```

```
Best df: df_outliers_log -- 0.907259043457
```

Hyperparamter tuning for model with best accuracy.

```
Cs = [0.001, 0.1, 1, 10, 100]
results = []
max_score = 0
```

```
for dframe in new_all_dataframes:

    X = dframe.drop('y', 1)
    y = pd.DataFrame(dframe[['y']])
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

    for c in Cs:

        reg = LogisticRegression(C=c)
        reg.fit(X_train, y_train)
        score = accuracy_score(reg.predict(X_test),y_test)
        print(dframe.name + " -- %f score: %f" % (c, score))

        if (score > max_score):
            max_score = score
            best_c = c
            best_df = dframe.name

print("Best df:"+ str(best_df))
print("Best c:" + str(best_c))
```

Training models to find the best hyper parameters and metrics

```
Cs = [0.001, 0.1, 1, 10, 100]
results = []
max_score = 0
max_recall = 0
min_precision = math.inf

for dframe in new_all_dataframes:
    for c in Cs:

        X = dframe.drop('y', 1)
        y = pd.get_dummies(dframe[['y']], drop_first = True)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

        reg = LogisticRegression(C=c)
        reg.fit(X_train, y_train)
        pred = reg.predict(X_test)

        score = accuracy_score(reg.predict(X_test),y_test)
        recall = recall_score(y_test, pred)
        precision = precision_score(y_test, pred)

        if (score > max_score):
            max_score = score
            best_c = c
            best_c_df = dframe.name

        if (recall > max_recall):
            max_recall = recall
            best_recall = recall
            best_recall_df = dframe.name

        if (precision < min_precision):
            min_precision = precision
            best_precision = precision
            best_precision_df = dframe.name
```

```
Best df: df_outliers
Best c: 0.1

Best df: df_upsample
Best recall: 0.889485273878

Best df: df_log_standardization
Best precision: 0.586419753086
```

7.2 KNN

Trying multiple neighbors to build a k-nearest neighbor model.

```
neighbors = []
for i in range(1, 50, 2):
    neighbors.append(i)
print(neighbors)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]
```

Training a knn model to find the best parameters.

```
for i in neighbors:
    knn = KNeighborsClassifier(n_neighbors = i)
    knn.fit(X_train, y_train)
    pred = knn.predict(X_test)
    score = (accuracy_score(y_test, pred))
    accuracy.append(score)
```

```
1 ---- 0.887351298859
3 ---- 0.896941005098
5 ---- 0.902403495994
7 ---- 0.902767662054
9 ---- 0.903131828114
11 ---- 0.904952658412
13 ---- 0.904588492353
15 ---- 0.905438213159
17 ---- 0.906894877397
19 ---- 0.907623209517
21 ---- 0.906409322651
23 ---- 0.905559601845
25 ---- 0.906287933965
27 ---- 0.907987375577
29 ---- 0.907259043457
31 ---- 0.907623209517
33 ---- 0.907380432144
35 ---- 0.907380432144
37 ---- 0.907016266084
39 ---- 0.906894877397
41 ---- 0.906409322651
43 ---- 0.907137654771
45 ---- 0.906652100024
47 ---- 0.907016266084
49 ---- 0.906773488711
```

All neighbors yield almost a similar accuracy score. Upon further analysis, it is apparent the best model is df_upsample.

7.3 SVM

An SVM model with base parameters

```
model = svm.svc(kernel='linear', c=1, gamma=1)
model.fit(X, y)
model.score(X, y)
predicted= model.predict(x_test)
```

Various hyper parameters

```
models = ['linear', 'rbf', 'poly']
C = [1, 10, 100, 1000]
gamma = [1, 10, 100, 1000]
```

Trying various hyper parameters

```
for m in models:
    for c in C:
        for g in gamma:
            model = SVC(kernel = m, C = c, gamma = g)
            model.fit(X_train, y_train.values.ravel())
            pred = model.predict(X_test)
            score = accuracy_score(y_test, pred)

            if best_accuracy < score:
                best_accuracy = score
                best_model = m
                best_C = c
                best_gamma = g
```

The best model is df_outliers.

7.4 Random Forests with untransformed data

Initially the data is trained on the base model with no transformations

```
# Train and Test data
X = df_full.drop(['y'], axis = 1)
y = pd.get_dummies(df_full[['y']], drop_first = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

# Converting dataframe to numpy array
data = [X_train, X_test, y_train, y_test]
for d in data:
    d = np.array(d)

# Instantiate model with 1000 decision trees
rf_df_full = RandomForestClassifier(n_estimators = 1000, random_state = 42)

# Train the model on training data
rf_df_full.fit(X_train, y_train.values.ravel());

# Predict test data
pred = rf_df_full.predict(X_test)

# Accuray Score
print("Accuracy -- ", metrics.accuracy_score(pred, y_test))
print("Precision -- ", metrics.precision_score(pred, y_test))
print("Recall -- ", metrics.recall_score(pred, y_test))
print("F1 Score -- ", metrics.f1_score(pred, y_test))
print("AUC -- ", metrics.auc(pred, y_test))
```

```
Accuracy --  0.912357368293
Precision --  0.445974576271
Recall --  0.679032258065
F1 Score --  0.538363171355
AUC --  51838.5
```

7.4.1 Randomized Search CV:

Random Forests has the following hyper parameters:

n_estimators = number of trees in the foreset
max_features = max number of features considered for splitting a node
max_depth = max number of levels in each decision tree
min_samples_split = min number of data points placed in a node before the node is split
min_samples_leaf = min number of data points allowed in a leaf node
bootstrap = method for sampling data points (with or without replacement)

Default parameters used in the Random Search:

```
{'bootstrap': True,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 10,
}
```

Using Randomized Search CV to pick the best parameters. Best parameters can take any values. Efficient approach is to narrow our search to evaluate a wide range of values for each hyperparameter.

```
# Number of trees in random forest
n_estimators = [100, 200, 400, 600, 800, 1000]

# Number of features to consider at every split
max_features = ['auto', 'sqrt', 0.2]

# Maximum number of levels in tree
max_depth = [1, 2, 3, 4, 5, 10, 25, 50, 75, 100, 110, None]

# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]

# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4, 5, 10, 50, 100, 200, 500]

# Method of selecting samples for training each tree
bootstrap = [True, False]
```

Training the base model with different sets of parameters to find the best set.

```
# First create the base model to tune
rf = RandomForestRegressor()

# search across 100 different combinations, and use all available cores
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid, n_iter = 100, cv = 5,
verbose=2, random_state=42, n_jobs = -1)

# Fit the random search model
rf_random.fit(X_train, y_train.values.ravel())
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed: 13.3min
```

```
rf_random.best_params_
```

```
{'bootstrap': False,
 'max_depth': 110,
 'max_features': 0.2,
 'min_samples_leaf': 5,
 'min_samples_split': 5,
 'n_estimators': 800}
```

7.4.2 Grid Search CV:

Using Grid Search CV to pick the best parameters. This gives us an idea where to concentrate our search.

```
# Create the parameter grid based on the results of random search
param_grid = {
    'bootstrap': [True],
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4],
    'min_samples_split': [8, 10],
    'n_estimators': [100, 200, 300]
}

# Create a based model
rf = RandomForestRegressor()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                          cv = 5, n_jobs = -1, verbose = 2)
```

Training the base model with different sets of parameters to find the best set.

```
X = df_full.drop(['y'], axis = 1)
y = pd.get_dummies(df_full[['y']], drop_first = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

data = [X_train, X_test, y_train, y_test]
for d in data:
    d = np.array(d)

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

grid_search.best_params_
```

```
Fitting 5 folds for each of 96 candidates, totalling 480 fits
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 154 tasks      | elapsed:  6.1min
[Parallel(n_jobs=-1)]: Done 357 tasks      | elapsed: 13.2min
[Parallel(n_jobs=-1)]: Done 480 out of 480 | elapsed: 18.0min finished
C:\Users\Nishu\Anaconda3\Lib\site-packages\sklearn\model_selection\_search.py:645: DataConversionWarning
: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,
), for example using ravel().
  best_estimator.fit(X, y, **self.fit_params)
```

```
{'bootstrap': True,
 'max_depth': 110,
 'max_features': 3,
 'min_samples_leaf': 3,
 'min_samples_split': 8,
 'n_estimators': 100}
```

Note: Since the goal of this project is to minimize False Negatives (How many did we miss), we focus on getting a recall value close to 100% with a less bad precision value

7.4.3 Training all the models with grid search CV best parameters:

Training all models with best parameters of Grid Search CV

```
# Grid Search - Best Params
for df in all_dfs:
    # Train and Test data
    X = df.drop(['y'], axis = 1)
    y = pd.get_dummies(df[['y']], drop_first = True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

    # Converting dataframe to numpy array
    data = [X_train, X_test, y_train, y_test]
    for d in data:
        d = np.array(d)

    # Instantiate model with 1000 decision trees
    rf = RandomForestClassifier(bootstrap = False, max_depth = 110 ,
     max_features = 3,
     min_samples_leaf = 5,
     min_samples_split = 8,
     n_estimators = 100, random_state = 42)

    # Train the model on training data
    rf.fit(X_train, y_train.values.ravel());

    # Predict test data
    pred = rf.predict(X_test)
```

Training all models with best parameters of Random Search CV

```
# Random Search - Best Params
for df in all_dfs:
    # Train and Test data
    X = df.drop(['y'], axis = 1)
    y = pd.get_dummies(df[['y']], drop_first = True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

    # Converting dataframe to numpy array
    data = [X_train, X_test, y_train, y_test]
    for d in data:
        d = np.array(d)

    # Instantiate model with 1000 decision trees
    rf = RandomForestClassifier(bootstrap = False, max_depth = 110,
     max_features = 0.2,
     min_samples_leaf = 5,
     min_samples_split = 5,
     n_estimators = 800, random_state = 42)

    # Train the model on training data
    rf.fit(X_train, y_train.values.ravel());

    # Predict test data
    pred = rf.predict(X_test)
```

Best parameters of Grid Search CV are chosen over the best parameters Random Search CV considering the computational resources I have.

Below is the table with models are their respective metrics

	Unnamed: 0	accuracy	auc	dataframe	f1	precision	recall	tuning
0	0	0.908352	46974.0	df_full	0.502962	0.404661	0.664348	random_base
1	1	0.907259	47360.0	df_age_1	0.501956	0.407839	0.652542	random_base
2	2	0.908109	48131.5	df_age_2	0.513809	0.423729	0.652529	random_base
3	3	0.972709	439811.0	df_upsample_1	0.973227	0.998073	0.949588	random_base
4	4	0.971546	436492.5	df_upsample_5	0.972133	0.998624	0.947011	random_base
5	5	0.971614	436747.5	df_upsample_6	0.972198	0.998624	0.947135	random_base
6	6	0.853448	52485.0	df_downsample_2	0.856389	0.860934	0.851891	random_base
7	7	0.852371	52605.5	df_downsample_5	0.856394	0.867304	0.845756	random_base
8	8	0.852371	52605.5	df_downsample_6	0.856394	0.867304	0.845756	random_base
9	0	0.916485	56956.5	df_full	0.579976	0.503178	0.684438	random_best
10	1	0.917213	58362.5	df_age_1	0.586667	0.512712	0.685552	random_best
11	2	0.917092	57850.0	df_age_2	0.583790	0.507415	0.687231	random_best
12	3	0.953283	423795.0	df_upsample_1	0.954915	0.995458	0.917544	random_best
13	4	0.950410	429567.5	df_upsample_5	0.951417	0.977016	0.927126	random_best
14	5	0.950479	429695.0	df_upsample_6	0.951481	0.977016	0.927247	random_best
15	6	0.889547	52224.0	df_downsample_2	0.896621	0.943737	0.853987	random_best
16	7	0.885776	52738.0	df_downsample_5	0.892495	0.934183	0.854369	random_best
17	8	0.885776	52738.0	df_downsample_6	0.892495	0.934183	0.854369	random_best
18	0	0.906652	32250.5	df_full	0.401556	0.273305	0.756598	grid_best
19	1	0.909080	37883.0	df_age_1	0.448859	0.323093	0.734940	grid_best
20	2	0.908837	35069.0	df_age_2	0.431491	0.301907	0.755968	grid_best
21	3	0.939466	418177.5	df_upsample_1	0.941488	0.979906	0.905968	grid_best
22	4	0.919357	421887.5	df_upsample_5	0.920149	0.934902	0.905854	grid_best
23	5	0.919289	421760.0	df_upsample_6	0.920087	0.934902	0.905733	grid_best
24	6	0.880388	50692.5	df_downsample_2	0.888218	0.936306	0.844828	grid_best
25	7	0.875539	51842.5	df_downsample_5	0.882203	0.918259	0.848871	grid_best
26	8	0.875539	51842.5	df_downsample_6	0.882203	0.918259	0.848871	grid_best

Here is the list of dataframes tested on the best parameters of Grid Search CV and their respective metrics.

	A	B	C	D	E	F	G
1	dataframe	accuracy	recall(1)	precision(1)	f1 (1)	transformations/feature engineering techniques applied	
2	df_full	91.23	0.68	0.45	0.57	age variable	
3	df_age_1	91.55	0.68	0.49	0.57	age replaced with 'young', 'young_adult', 'senior'	
4	df_age_2	91.42	0.67	0.49	0.56	age replaced with custome categories	
5	df_age_marital	91.35	0.67	0.48	0.56	age and marital features combined	
6	df_outliers_1	91.28	0.68	0.45	0.56	outliers replaced with upper and lower bounds	
7	df_outliers_2	91.41	0.67	0.49	0.56	outliers replaced with upper and lower quantiles	
8	df_outliers_3	91.39	0.67	0.48	0.56	logarithm transformations	
9	df_standardized_1	89.82	0.68	0.21	0.31	Standardized	
10	df_standardized_2	89.68	0.66	0.2	0.31	Transformed and Standardized	
11	df_standardized_3	89.53	0.64	0.2	0.31	Transformed, Outlier treatment and Standardized	
12	df_standardized_4	89.57	0.64	0.2	0.31	Transformed, Outlier treatment, Logged and Standardized	
13	df_normalized_1	89.82	0.68	0.21	0.31	Normalized	
14	df_normalized_2	89.68	0.66	0.2	0.31	Transformed and Normalized	
15	df_normalized_3	89.53	0.64	0.2	0.31	Transformed, Outlier treatment and Normalized	
16	df_normalized_4	89.57	0.64	0.2	0.92	Transformed, Outlier treatment, Logged and Normalized	
17	df_upsample_1	97.24	0.95	1	0.97	Upsample	
18	df_upsample_2	96.81	0.94	1	0.97	Upsample and Transformation	
19	df_upsample_3	96.73	0.94	1	0.97	Upsample, Transformations and Outlier Treatment	
20	df_upsample_4	96.75	0.94	1	0.97	Upsample, Transformations, Outlier Treatment and Logged	
21	df_upsample_5	97.25	0.95	1	0.97	Upsample, Transformations, Outlier Treatment, Logged and Standardized	
22	df_upsample_6	97.24	0.95	1	0.97	Upsample, Transformations, Outlier Treatment, Logged and Normalized	
23	df_downsample_1	88.48	0.94	0.85	0.97	Downsample	
24	df_downsample_2	96.81	0.94	1	0.97	Downsample and Transformation	
25	df_downsample_3	87.98	0.85	0.92	0.97	Downsample, Transformations and Outlier Treatment	
26	df_downsample_4	88.03	0.85	0.92	0.97	Downsample, Transformations, Outlier Treatment and Logged	
27	df_downsample_5	97.25	0.95	1	0.97	Downsample, Transformations, Outlier Treatment, Logged and Standardized	
28	df_downsample_6	97.24	0.95	1	0.97	Downsample, Transformations, Outlier Treatment, Logged and Normalized	

Highlighted are the models with optimal metrics.

8. Choosing the best model

From the models highlighted in the above screenshot, df_age_2 is the model that yields a better results on the test data.

Other models are not chosen (upsample and downsample) considering the weights each classes are given when the data is either upsampled/downsampled.

9. Other potential data sets I could use

The data provided could actually be considered very rich in terms of predicting the client's behavior for a given campaign. However, given additional data pertaining to client's financial spending such as income disposal, large credit purchases, demographic of the client.