# Introduction to Data Science - Bank Marketing Project

Team Members:

Ophir Shurany ID: 304867716

Amit Shreiber ID: 200867174

## 1. Load Database

In [154]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
plt.close('all')
import time
start_time = time.time()
```

### 1.1.-1.3 Create and load dataframe

In [155]:

```
data = pd.read_csv("bank.csv",sep='|',encoding='utf8')
#drop duplicate data
df = data
df = df.drop_duplicates('Unnamed: 0',keep=False)
#drop #rows
df=df.drop('Unnamed: 0',axis=1)
df_copy_original=df #Keep original
```

## 2. Data Exploration

### 2.1. view first 5 rows in df

In [156]:

```
df.head()
```

Out[156]:

| | age | job | marital | education | default | housing | loan | contact | month | day |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 39.0 | admin. | married | university.degree | no | no | no | cellular | jul | |
| 2 | 51.0 | management | married | university.degree | NaN | no | no | telephone | jun | |
| 4 | 51.0 | blue-collar | married | basic.4y | NaN | no | yes | telephone | jun | |
| 5 | 53.0 | services | married | high.school | NaN | no | no | telephone | may | |
| 6 | 40.0 | blue-collar | married | basic.6y | no | no | no | cellular | may | |

5 rows × 21 columns

## 2.2. Presenting all columns, number of rows and type

In [157]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 35370 entries, 0 to 39895
Data columns (total 21 columns):
age               35239 non-null float64
job               35370 non-null object
marital           35303 non-null object
education         33880 non-null object
default           27970 non-null object
housing           34527 non-null object
loan              34527 non-null object
contact           35370 non-null object
month             35370 non-null object
day_of_week       35370 non-null object
duration          35370 non-null int64
campaign          34954 non-null float64
pdays             35370 non-null int64
previous          35370 non-null int64
poutcome          35370 non-null object
emp.var.rate      35370 non-null float64
cons.price.idx    35370 non-null float64
cons.conf.idx     35370 non-null float64
euribor3m         35370 non-null float64
nr.employed       35370 non-null float64
y                 35370 non-null object
dtypes: float64(7), int64(3), object(11)
memory usage: 5.9+ MB
```

## 2.3. Feature statistics for numerical categories

In [158]:

```
df.describe()
```

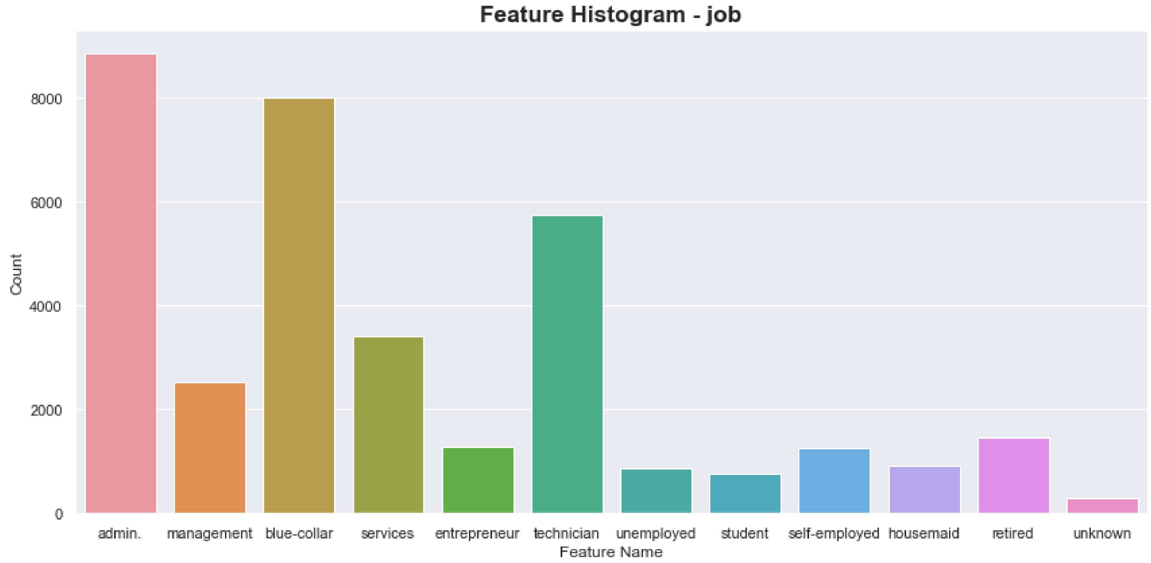
Out[158]:

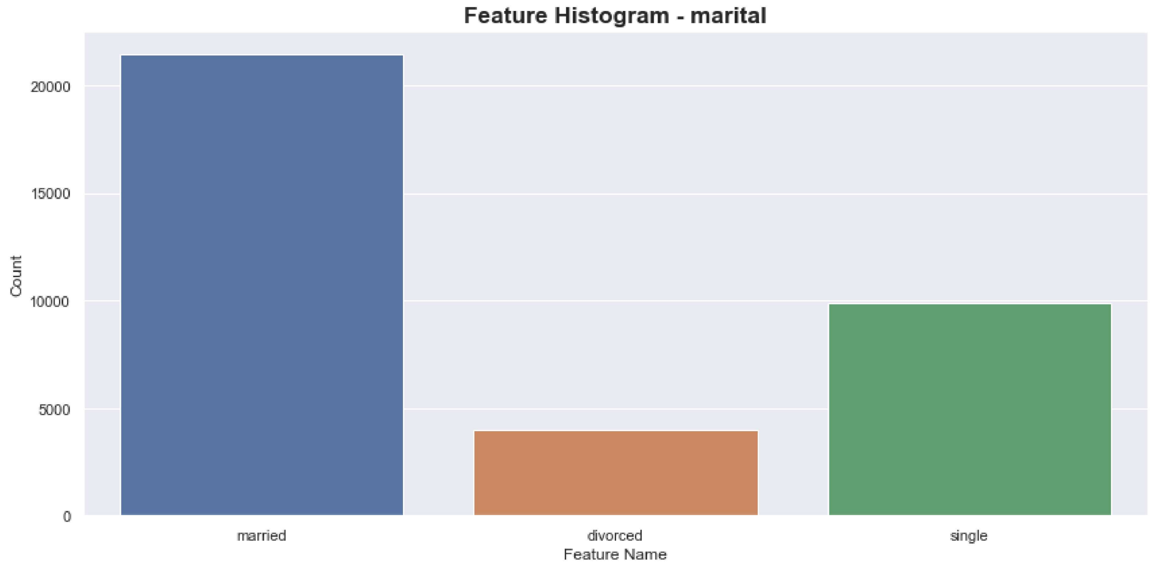
| | age | duration | campaign | pdays | previous | emp.var.rate |
|---|---|---|---|---|---|---|
| **count** | 35239.000000 | 35370.000000 | 34954.000000 | 35370.000000 | 35370.000000 | 35370.000000 |
| **mean** | 40.042879 | 257.985129 | 2.572209 | 963.598728 | 0.168335 | 0.097665 |
| **std** | 10.416313 | 258.531122 | 2.767796 | 184.121868 | 0.483834 | 1.564469 |
| **min** | 17.000000 | 0.000000 | 1.000000 | 0.000000 | 0.000000 | -3.400000 |
| **25%** | 32.000000 | 103.000000 | 1.000000 | 999.000000 | 0.000000 | -1.800000 |
| **50%** | 38.000000 | 180.000000 | 2.000000 | 999.000000 | 0.000000 | 1.100000 |
| **75%** | 47.000000 | 318.000000 | 3.000000 | 999.000000 | 0.000000 | 1.400000 |
| **max** | 98.000000 | 4918.000000 | 56.000000 | 999.000000 | 7.000000 | 1.400000 |

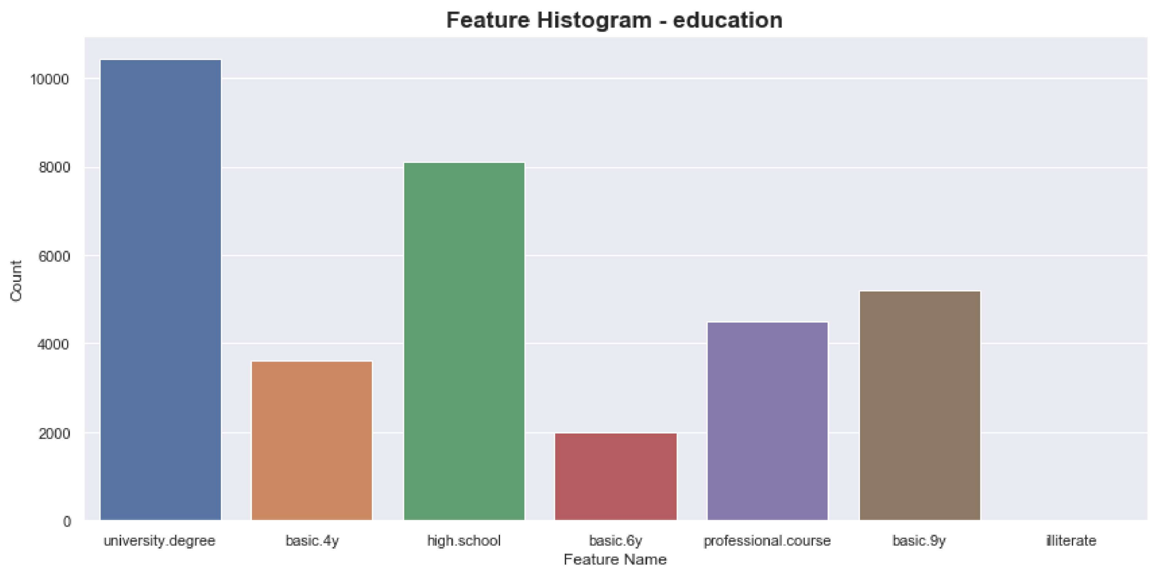
Histograms for categorial features

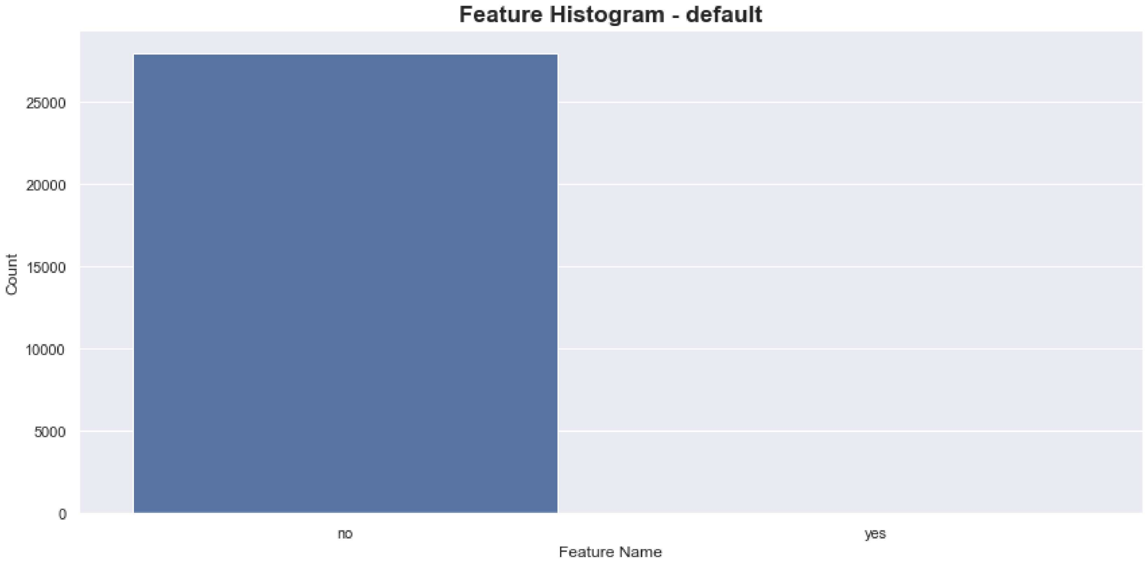
In [159]:

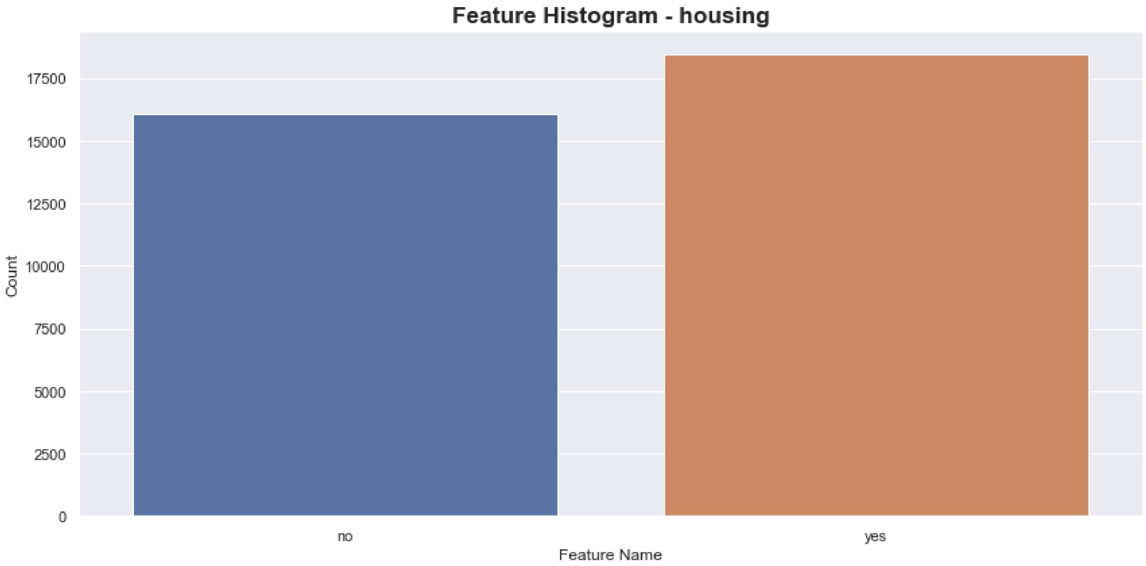
```
categorcial_variables =list(df.select_dtypes(include="object").columns)
for feature in categorcial_variables:
    plt.figure(figsize=(12, 6))
    sns.countplot(x=feature,data=df)
    plt.title("Feature Histogram - " + feature,fontsize='xx-large', fontweight='bold')
    plt.ylabel("Count")
    plt.xlabel("Feature Name")
    plt.tight_layout()
```

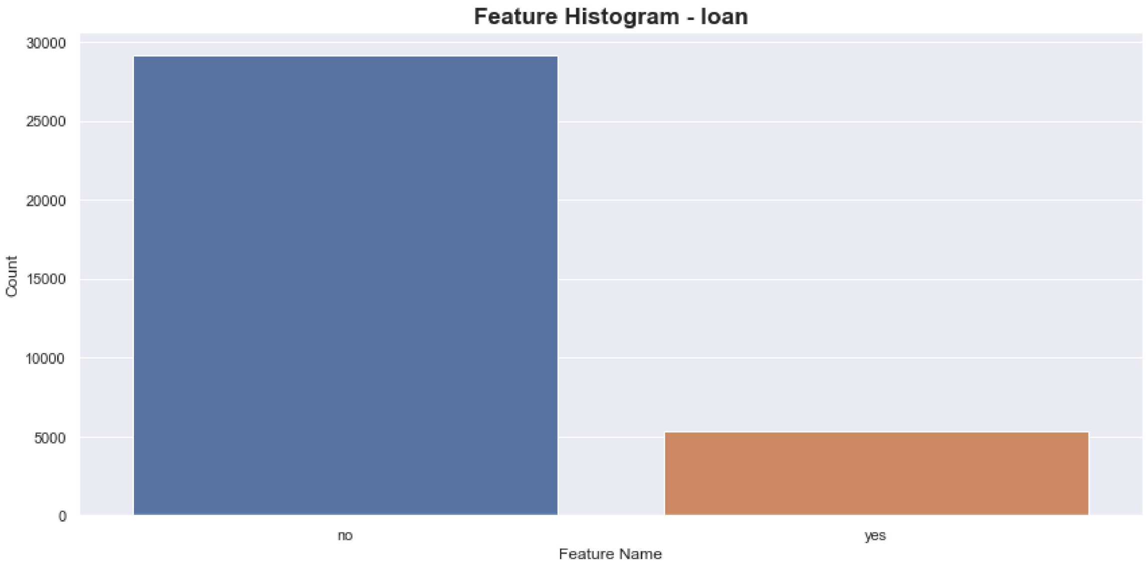

Feature Histogram - job
Count
0
2000
4000
6000
8000
admin.
management
blue-collar
services
entrepreneur
technician
unemployed
student
self-employed
housemaid
retired
unknown
Feature Name

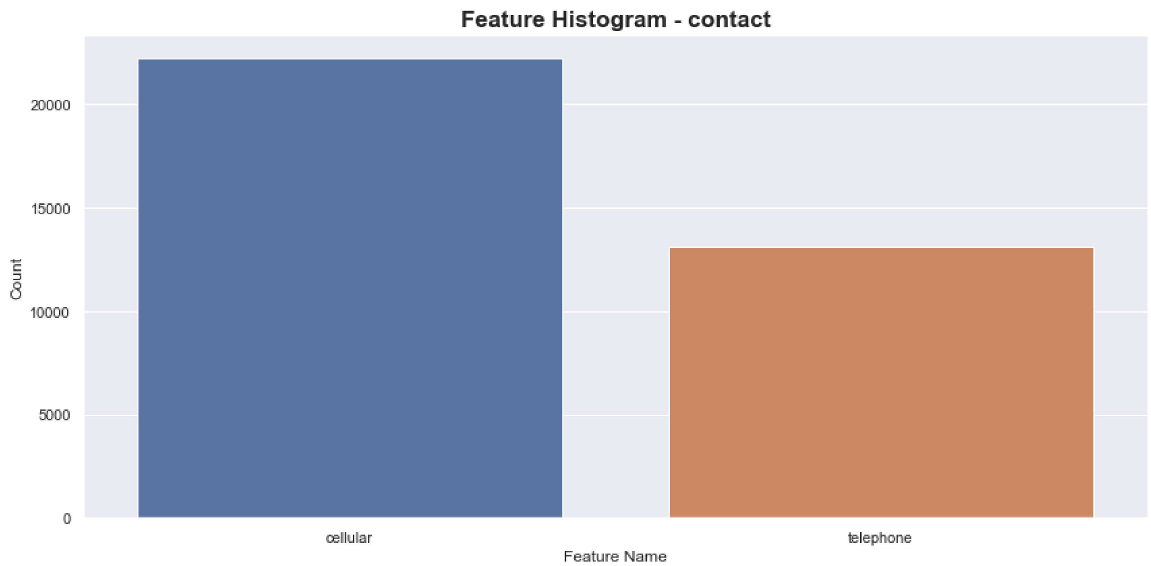

Feature Histogram - marital
Count
0
5000
10000
15000
20000
married
divorced
single
Feature Name

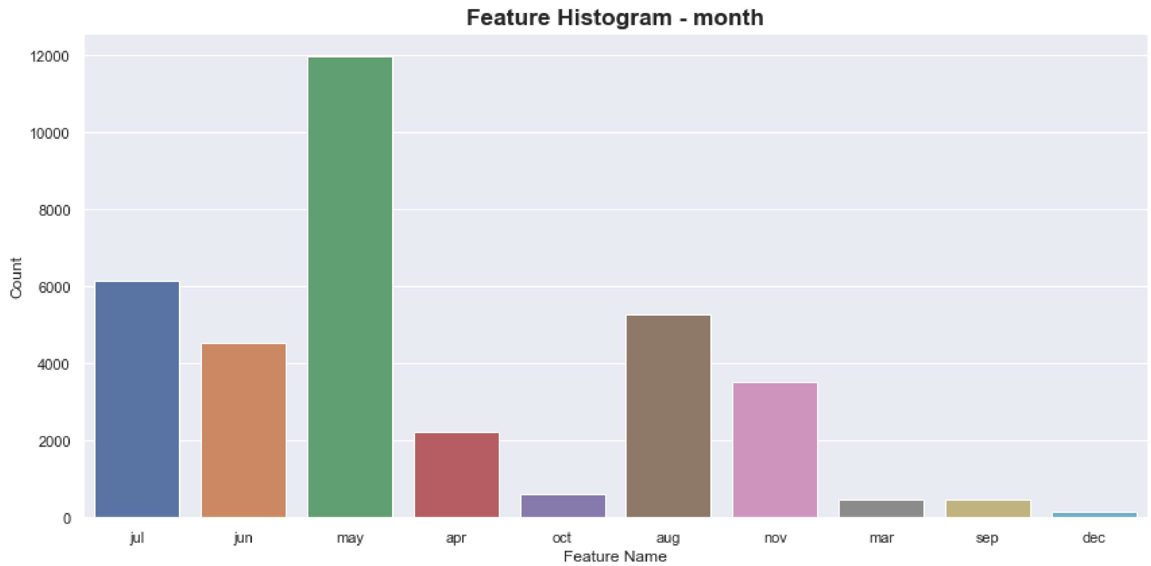

Feature Histogram - education
Count
0
2000
4000
6000
8000
10000
university.degree
basic.4y
high.school
basic.6y
professional.course
basic.9y
illiterate
Feature Name

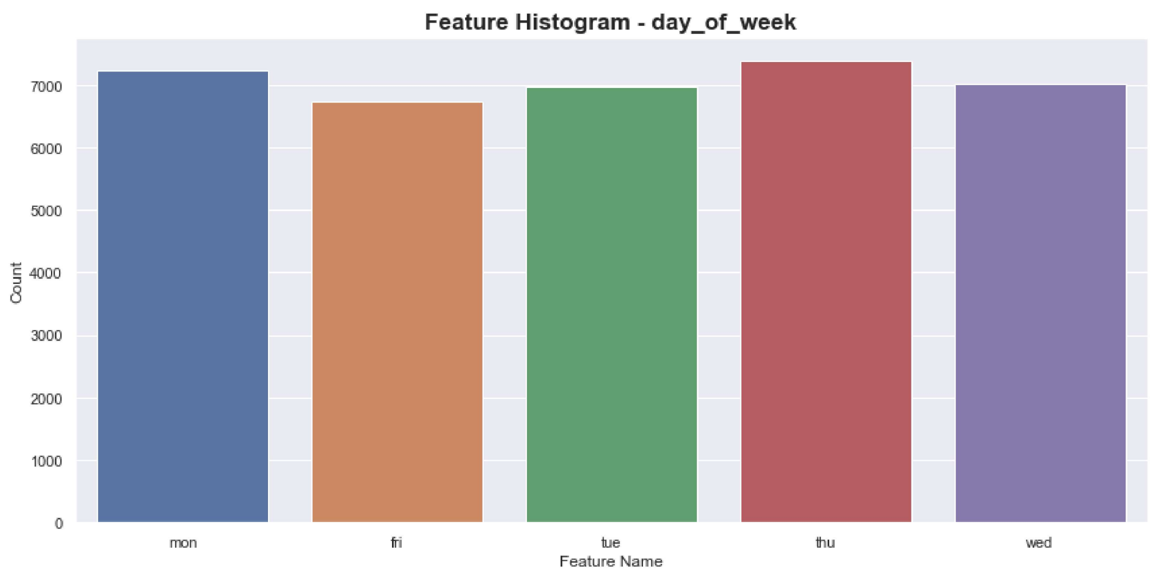

Feature Histogram - default
Count
25000
20000
15000
10000
5000
0
no
yes
Feature Name

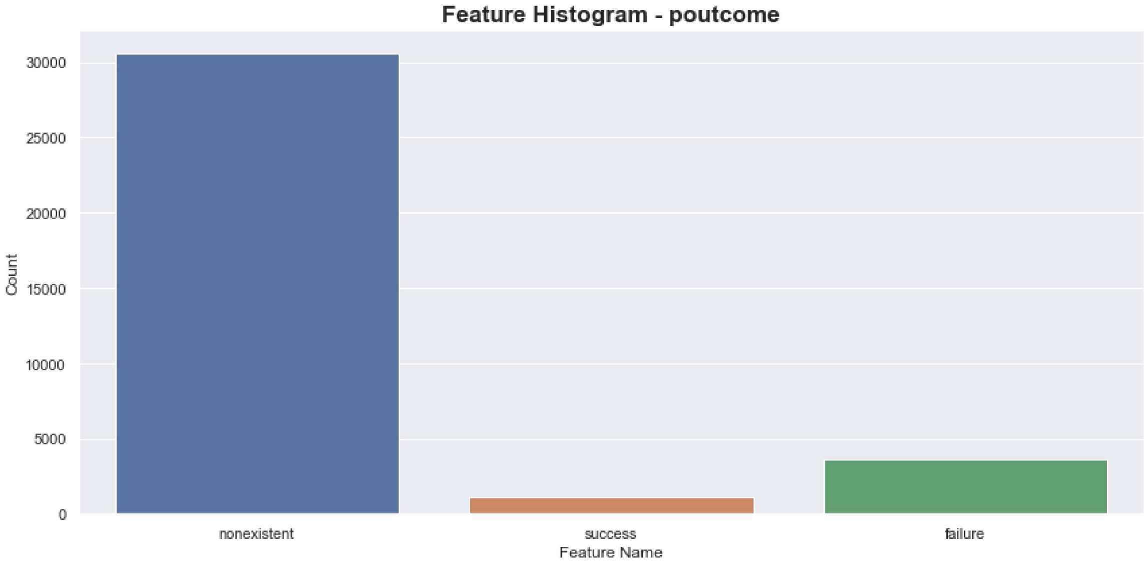

Feature Histogram - housing
Count
17500
15000
12500
10000
7500
5000
2500
0
no
yes
Feature Name

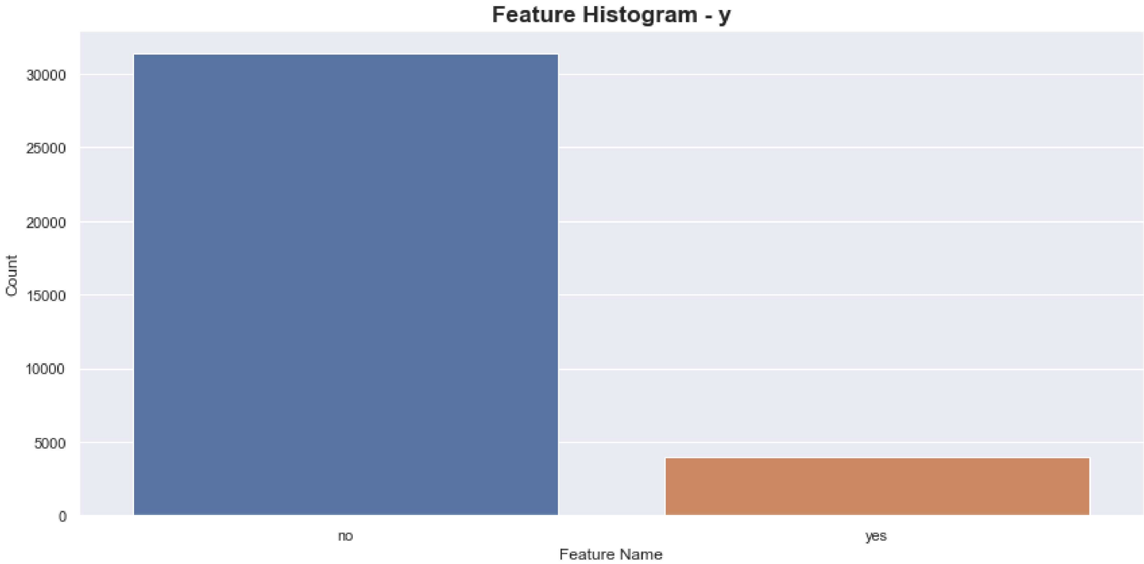

Feature Histogram - loan
Count
30000
25000
20000
15000
10000
5000
0
no
yes
Feature Name


Feature Histogram - contact
Count
0
5000
10000
15000
20000
cellular
telephone
Feature Name


Feature Histogram - month
Count
0
2000
4000
6000
8000
10000
12000
jul
jun
may
apr
oct
aug
nov
mar
sep
dec
Feature Name


Feature Histogram - day_of_week
Count
0
1000
2000
3000
4000
5000
6000
7000
mon
fri
tue
thu
wed
Feature Name

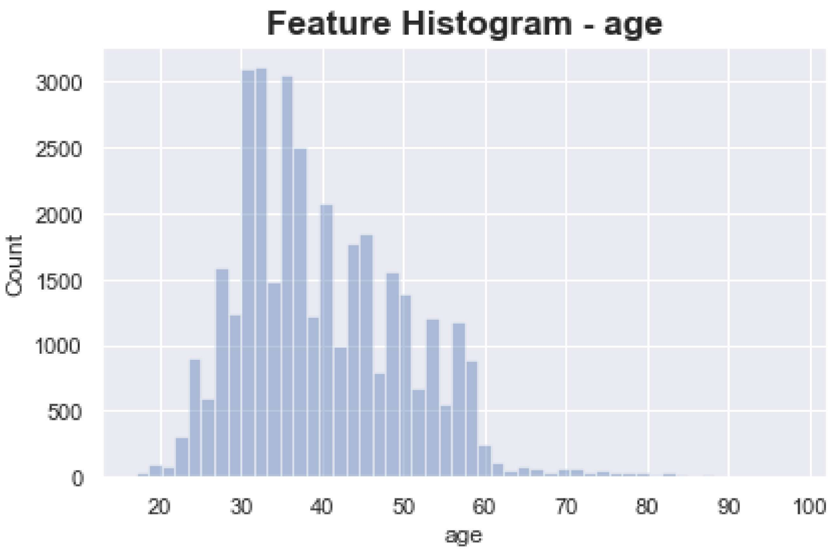


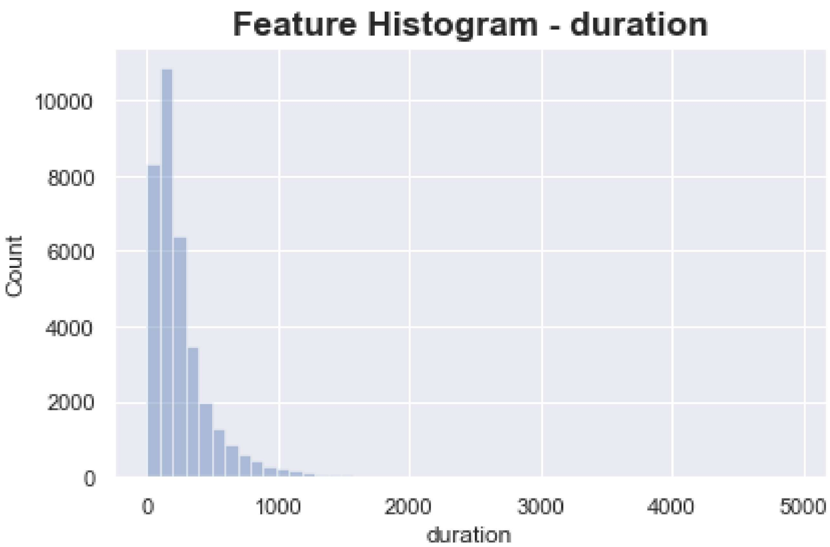



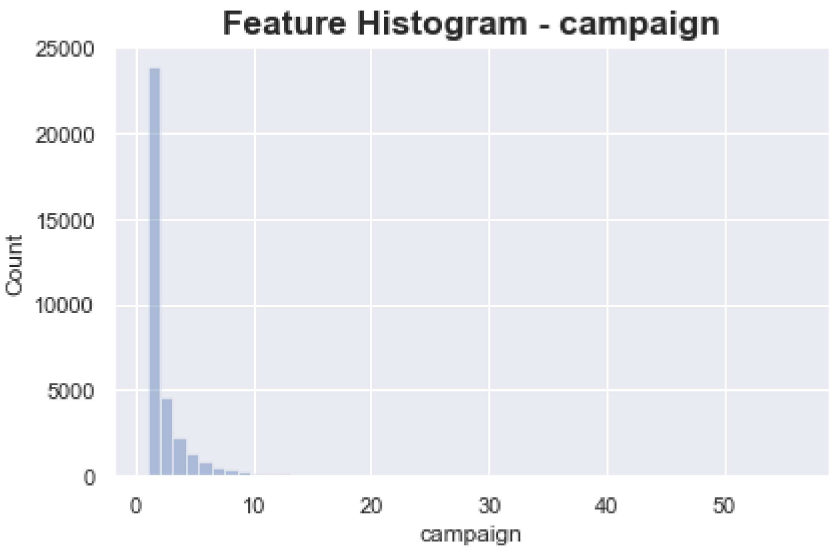

Histograms for Numeric features

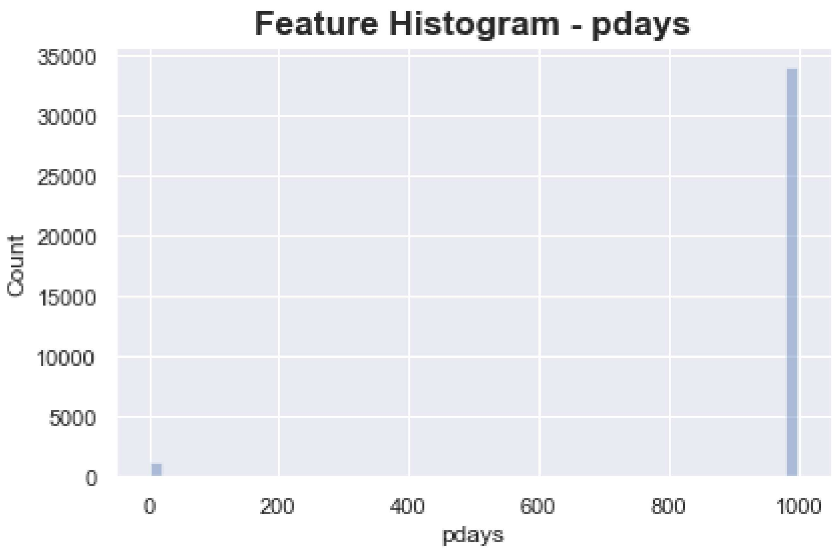
In [160]:

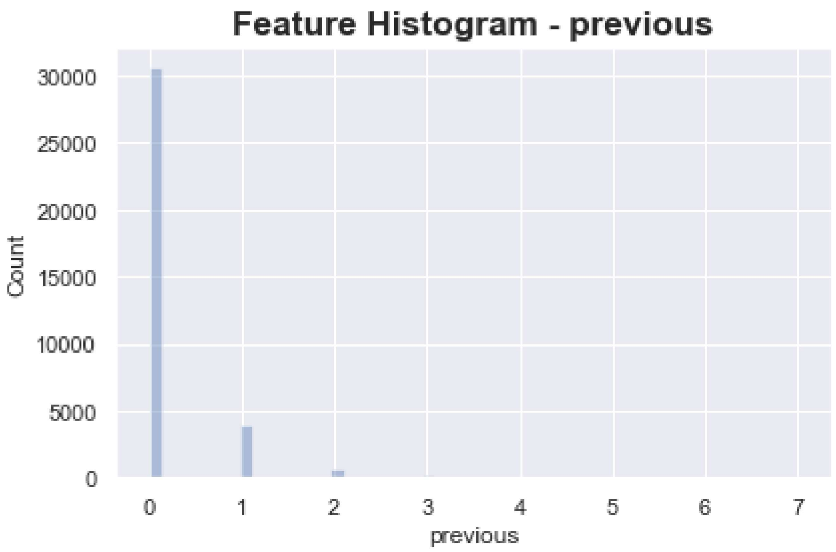
```
num_features = list(df.select_dtypes(exclude="object").columns)
for feature in num_features:
#devide for economic
    plt.figure()
    sns.distplot(df[feature].dropna(),kde=False)
    plt.title("Feature Histogram - " + feature,fontsize='xx-large', fontweight='bold')
    plt.ylabel("Count")
    plt.xlabel(feature)
    plt.tight_layout()
```

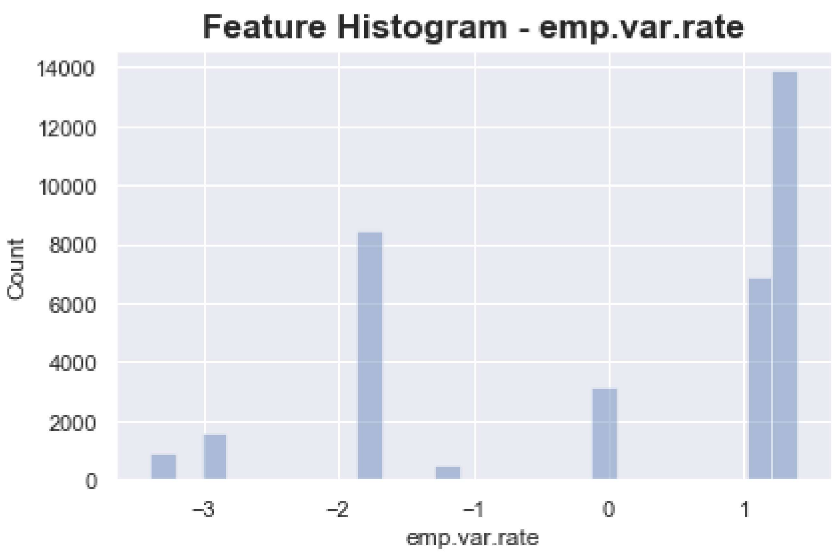

Feature Histogram - age
Count
3000
2500
2000
1500
1000
500
0
20
30
40
50
60
70
80
90
100
age

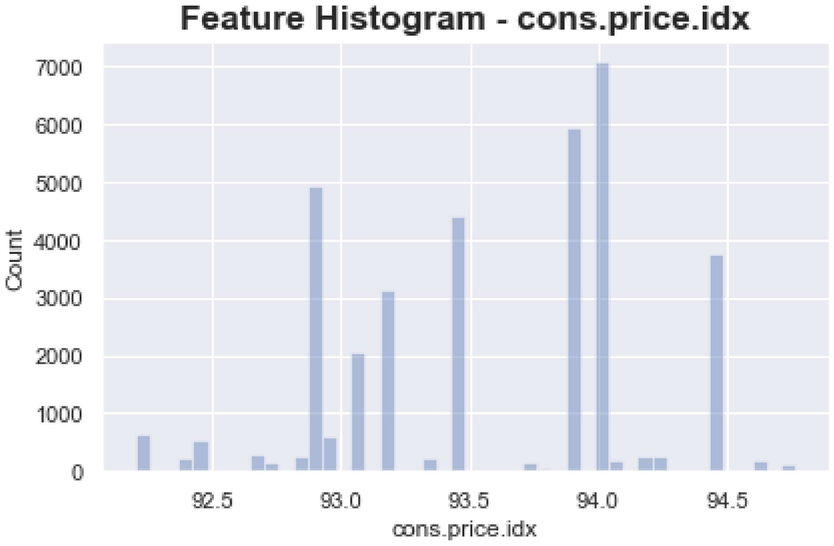

Feature Histogram - duration
Count
10000
8000
6000
4000
2000
0
0
1000
2000
3000
4000
5000
duration

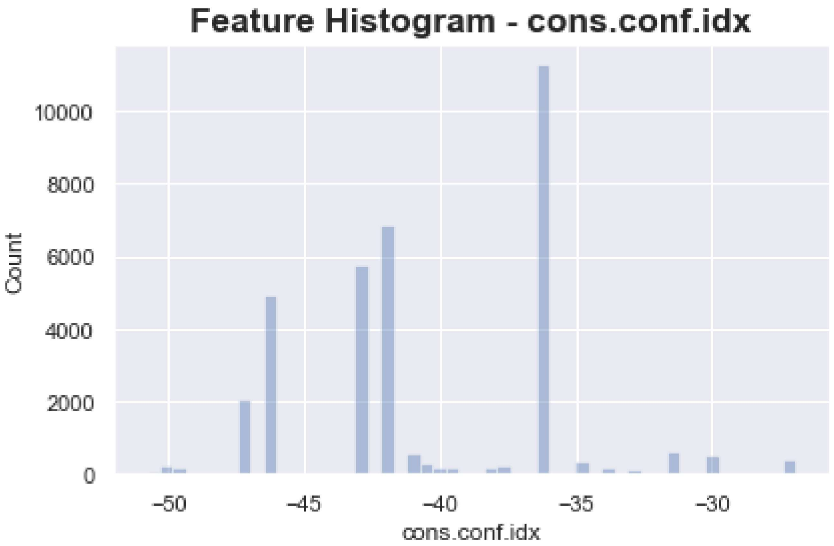

Feature Histogram - campaign
Count
25000
20000
15000
10000
5000
0
0
10
20
30
40
50
campaign

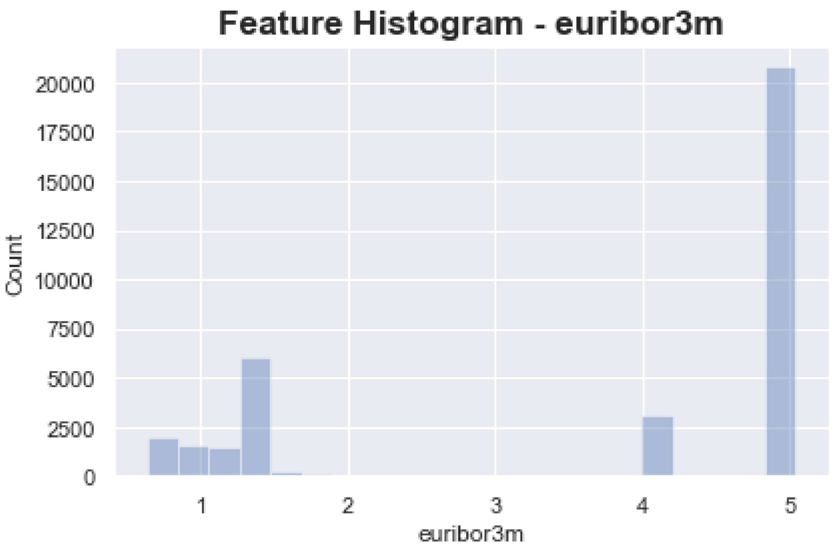

Feature Histogram - pdays
Count
pdays
0
5000
10000
15000
20000
25000
30000
35000
200
400
600
800
1000

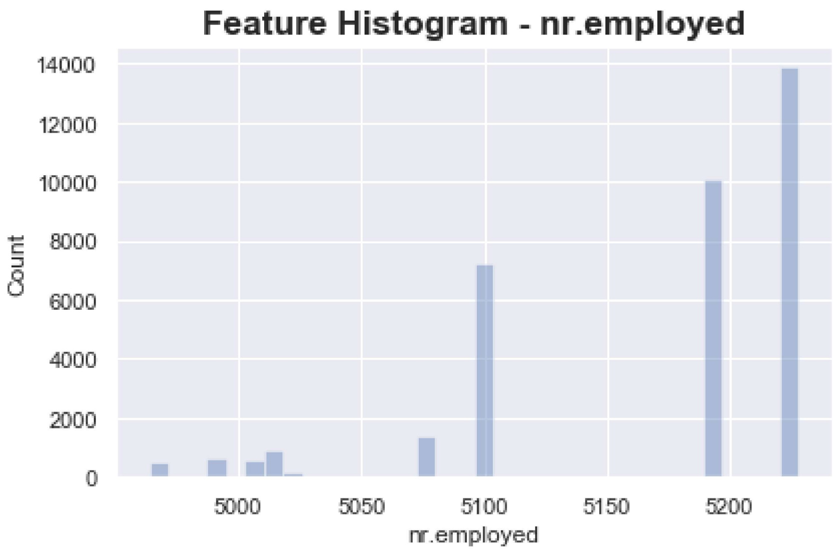

Feature Histogram - previous
Count
previous
0
5000
10000
15000
20000
25000
30000
1
2
3
4
5
6
7


Feature Histogram - emp.var.rate
Count
emp.var.rate
0
2000
4000
6000
8000
10000
12000
14000
−3
−2
−1
1


Feature Histogram - cons.price.idx
Count
cons.price.idx
7000
6000
5000
4000
3000
2000
1000
0
92.5
93.0
93.5
94.0
94.5


Feature Histogram - cons.conf.idx
Count
cons.conf.idx
10000
8000
6000
4000
2000
0
−50
−45
−40
−35
−30


Feature Histogram - euribor3m
Count
euribor3m
20000
17500
15000
12500
10000
7500
5000
2500
0
1
2
3
4
5




## 2.4. Categorial manipulations:

### 2.4.1 change "yes" or "no" to 1 or 0

In [161]:

```
df['y'] = df.y.map(dict(yes=1, no=0))
```

### 2.4.2 Convert the month list to 4 binary quarters column

In [162]:

```
months=['jan','feb','mar','apr','may','jun','jul','aug','sep','oct','nov','dec'];
Q = [1,1,1,1,2,2,2,3,3,3,4,4,4];month_dic=dict(zip(months,Q))
df['month']=df.month.replace(month_dic)
df=pd.get_dummies(df, columns=['month'],prefix='Q')
```

### 2.4.3. convert categorial features to numeric and drop the number of variables

In [163]:

```
#education
df['education']=df.education.replace(['basic.6y','basic.4y', 'basic.9y'], 'basic')
#job
df.job.replace(['admin.', 'management'], 'administration_management', inplace=True)
df.loc[(df['age'] > 60 ) & (df['job'] == 'administration_management' ) , 'job'] = 'reti
red'
df.job.replace(['retired', 'unemployed'], 'no_active_income', inplace=True)
df.job.replace('housemaid', 'services',inplace=True)
df['job']=df.job.replace('entrepreneur', 'self-employed')
# Convert other Series from yes or no to binary
df['housing'] = df.housing.map(dict(yes=1, no=0));
df['loan'] = df.loan.map(dict(yes=1, no=0));
df=df.rename(columns = {'contact':'contact_by_cellular'})
df['contact_by_cellular'] = df.contact_by_cellular.map(dict(cellular = 1, telephone = 0
))
```

# 3. Missing Values

In [164]:

```
print("3.1. Total NaN rows = " + str(sum(df.isna().sum())))
```

```
3.1. Total NaN rows = 11190
```

Present NaN % in each feature.

In [165]:

```
(100*df.isna().sum()/df.shape[0]).round(1)
```

Out[165]:

```
age                    0.4
job                    0.0
marital                0.2
education              4.2
default               20.9
housing                2.4
loan                   2.4
contact_by_cellular    0.0
day_of_week            0.0
duration               0.0
campaign               1.2
pdays                  0.0
previous               0.0
poutcome               0.0
emp.var.rate           0.0
cons.price.idx         0.0
cons.conf.idx          0.0
euribor3m              0.0
nr.employed            0.0
y                      0.0
Q_1                    0.0
Q_2                    0.0
Q_3                    0.0
Q_4                    0.0
dtype: float64
```

In [166]:

```
#we need to see how values are distributed:
#first, we convert unknown values from NaN so they will be countable as unknown:
df['default'] = df.default.replace(np.nan,'unknown',regex=True)
#default
pd.crosstab(df['y'],df['default'],dropna=True).apply(lambda r: r/r.sum(), axis=1).round
(4)
#most of No are at default, so we cant really learn from it. then, it will be deleted
df=df.drop("default",axis=1)
#loan
df['loan'] = df.loan.replace(np.nan,'unknown',regex=True)
pd.crosstab(df['y'],df['loan']).apply(lambda r: r/r.sum(), axis=1).round(2)
df['loan'] = df['loan'].replace('unknown',0)
#housing
df['housing'] = df.housing.replace(np.nan,'unknown',regex=True)
pd.crosstab(df['y'],df['housing']).apply(lambda r: r/r.sum(), axis=1).round(2)
#values distribute practicly evenly, therefore we can delete uknowns:
df = df[df.housing != "unknown"]
```

We think that ‘job’ is influenced by the ‘education’ of a person. because that, we can infer ‘job’ based on the education of the person. Moreover, since we are just filling the missing values, we are not much concerned about the causal inference. We can use the job to predict education.

to infer the missing values in 'job' and 'education', we make use of the cross-tabulation between 'job' and 'education'.

In [167]:

```
df['education'] = df.education.replace(np.nan,'unknown',regex=True)
df['job'] = df.job.replace(np.nan,'unknown',regex=True)
pd.crosstab(df['job'], df['education'], rownames=['job'], colnames=['education'],margin
s=True)
```

Out[167]:

| education<br>job | basic | high.school | illiterate | professional.course | university.degr |
|---|---|---|---|---|---|
| administration_management | 896 | 2974 | 1 | 379 | 65 |
| blue-collar | 6214 | 746 | 8 | 378 | [illegible] |
| no_active_income | 958 | 446 | 3 | 324 | 4 |
| self-employed | 646 | 298 | 3 | 254 | 11 |
| services | 1171 | 2387 | 1 | 242 | 2 |
| student | 119 | 306 | 0 | 36 | 1 |
| technician | 443 | 731 | 0 | 2764 | 15 |
| unknown | 89 | 30 | 0 | 10 | [illegible] |
| All | 10536 | 7918 | 16 | 4387 | 102 |

While imputing the values for job and education, we followed the fact that the correlations should make real world sense.

In [168]:

```
#education
#for education it makes sense to use ranking
education_dic={'illiterate': 0,'basic' : 1,'high.school' : 2,'professional.course' : 3,
'university.degree' : 4}
df['education']=df.education.replace(education_dic)
```

By the cross table above we can infer the following:

In [169]:

```
#Most customers with "basic" education work as "blue-collar"
df.loc[(df['job']=='unknown') & (df['education']==1), 'job'] = 'blue-collar'
df.loc[(df['education']=='unknown') & (df['job']=='blue-collar'), 'education'] = 1
#Most customers in "services" have a "high.school"  degree
df.loc[(df['education']=='unknown') & (df['job']=='services'), 'education'] = 2
df.loc[(df['job']=='unknown') & (df['education']==2), 'job'] = 'services'
#Most customers with "professional.course education work as "technician"
df.loc[(df['job']=='unknown') & (df['education']==3), 'job'] = 'technician'
df.loc[(df['education']=='unknown') & (df['job']=='technician'), 'education'] = 3
#Most customers in "administration_management" have a "university.degree"
df.loc[(df['education']=='unknown') & (df['job']=='administration_management'), 'educat
ion'] = 4
df.loc[(df['job']=='unknown') & (df['education']=='administration_management'), 'job']
= 'administration_management'
```

In [170]:

```
pd.crosstab(df['job'], df['education'], rownames=['job'], colnames=['education'],margin
s=True)
```

Out[170]:

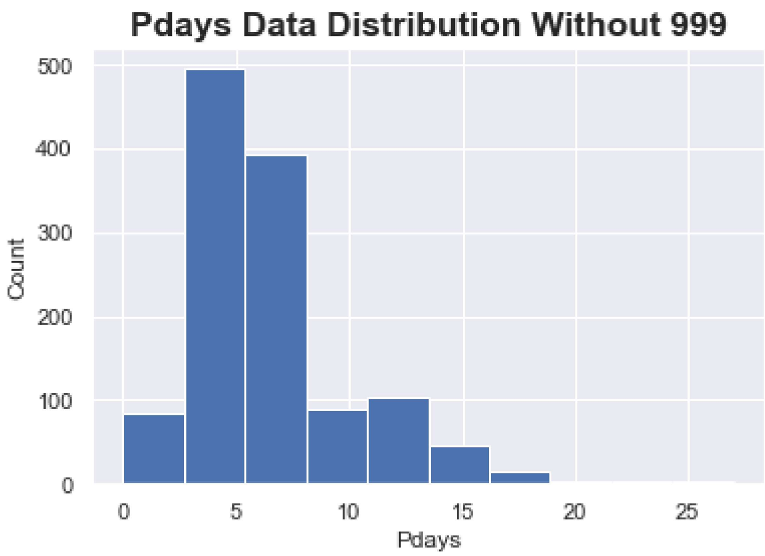
| education<br>job | 0 | 1 | 2 | 3 | 4 | unknown | All |
|---|---|---|---|---|---|---|---|
| **administration_management** | 1 | 896 | 2974 | 379 | 6828 | 0 | 11078 |
| **blue-collar** | 8 | 6674 | 746 | 378 | 84 | 0 | 7890 |
| **no_active_income** | 3 | 958 | 446 | 324 | 488 | 101 | 2320 |
| **self-employed** | 3 | 646 | 298 | 254 | 1185 | 80 | 2466 |
| **services** | 1 | 1171 | 2587 | 242 | 249 | 0 | 4250 |
| **student** | 0 | 119 | 306 | 36 | 143 | 139 | 743 |
| **technician** | 0 | 443 | 731 | 2950 | 1510 | 0 | 5634 |
| **unknown** | 0 | 0 | 0 | 0 | 37 | 109 | 146 |
| **All** | 16 | 10907 | 8088 | 4563 | 10524 | 429 | 34527 |

In [171]:

```
#Impute by mean value for age & campaign
#age
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan,strategy='mean')
df["age"] = imputer.fit_transform(df[['age']])
#campaign
df['campaign'] = imputer.fit_transform(df[['campaign']])
# Examine the missing values in 'pdays'
plt.figure()
plt.hist(df.loc[df.pdays != 999, 'pdays'])
plt.title("Pdays Data Distribution Without 999", fontsize='xx-large', fontweight='bold'
)
plt.xlabel("Pdays")
plt.ylabel("Count")
```

Out[171]:

```
Text(0, 0.5, 'Count')
```




two varibale are connected - pdays and poutcome. Lets check their connection and how many NaN we have-

In [172]:

```
pd.crosstab(df['pdays'],df['poutcome'],dropna=False,margins=True)
```

Out[172]:

| poutcome | failure | nonexistent | success | All |
|---|---|---|---|---|
| **pdays** | | | | |
| **0** | 0 | 0 | 14 | 14 |
| **1** | 0 | 0 | 20 | 20 |
| **2** | 0 | 0 | 51 | 51 |
| **3** | 3 | 0 | 356 | 359 |
| **4** | 2 | 0 | 96 | 98 |
| **5** | 4 | 0 | 34 | 38 |
| **6** | 22 | 0 | 309 | 331 |
| **7** | 13 | 0 | 34 | 47 |
| **8** | 4 | 0 | 10 | 14 |
| **9** | 20 | 0 | 27 | 47 |
| **10** | 5 | 0 | 36 | 41 |
| **11** | 3 | 0 | 20 | 23 |
| **12** | 10 | 0 | 38 | 48 |
| **13** | 7 | 0 | 25 | 32 |
| **14** | 4 | 0 | 12 | 16 |
| **15** | 9 | 0 | 11 | 20 |
| **16** | 2 | 0 | 7 | 9 |
| **17** | 5 | 0 | 3 | 8 |
| **18** | 5 | 0 | 2 | 7 |
| **19** | 0 | 0 | 2 | 2 |
| **21** | 1 | 0 | 0 | 1 |
| **22** | 0 | 0 | 3 | 3 |
| **25** | 1 | 0 | 0 | 1 |
| **26** | 0 | 0 | 1 | 1 |
| **27** | 0 | 0 | 1 | 1 |
| **999** | 3399 | 29896 | 0 | 33295 |
| **All** | 3519 | 29896 | 1112 | 34527 |

As we can see from the above table, the majority of the values for 'pdays' are missing. The majority of these missing values occur when the 'poutcome' is 'non-existent'. This means that the majority of the values in 'pdays' are missing because the customer was never contacted before. To deal with this variable, we removed the numerical variable 'pdays' and replaced it with categorical variables with following categories: p_days_missing, pdays_less_5, pdays_bet_5_15, and pdays_greater_15.

Add new categorical variables to our dataframe.

In [173]:

```
df['pdays_missing'] = 0
df['pdays_less_5'] = 0
df['pdays_greater_15'] = 0
df['pdays_bet_5_15'] = 0
df['pdays_missing'][df['pdays']==999] = 1
df['pdays_less_5'][df['pdays']<5] = 1
df['pdays_greater_15'][(df['pdays']>15) & (df['pdays']<999)] = 1
df['pdays_bet_5_15'][(df['pdays']>=5)&(df['pdays']<=15)]= 1
df= df.drop(['pdays','pdays_less_5'], axis=1)
```

Since we have many categorical variables, dummy variables needs to be created for those vaiables.

In [174]:

```
#convert categorical variables to dummy
df = pd.get_dummies(df , columns = ['job', 'marital' ,'day_of_week', 'poutcome'])
```

In [175]:

```
df = df[df != "unknown"]
print("we remove all other NaNs")
df = df.dropna()
```

```
we remove all other NaNs
```

In [176]:

```
print("Number of deleted rows = " + str(df_copy_original.shape[0]-df.shape[0]))
```

```
Number of deleted rows = 1272
```

In [177]:

```
print("only "+ str(round(100*(df_copy_original.shape[0]-df.shape[0])/df.shape[0],1))+"
 %")
```

```
only 3.7 %
```

In [178]:

```
print("Finally, the total NaN rows = " + str(sum(df.isna().sum())))
```

```
Finally, the total NaN rows = 0
```

#### Q4

Delete Q4 in order to avoid dummy variable trap.

#### duration

The variable “duration” will need to be dropped before we start building a predictive model because it highly affects the output target (e.g., if duration=0 then y=”no”).

Yet, the duration is not known before a call is performed.

In [179]:

```
df=df.drop(["duration","Q_4"],axis=1)
```

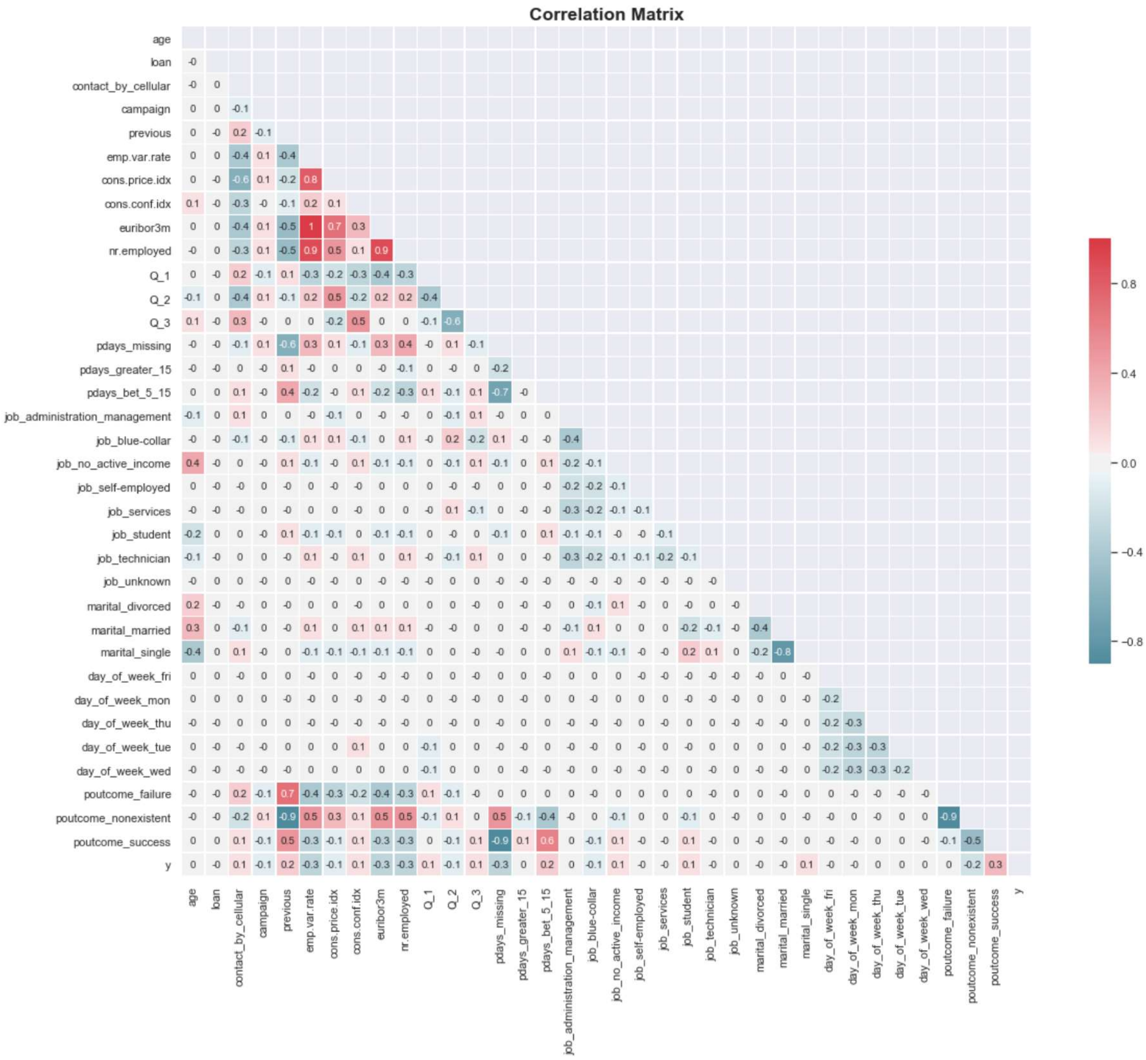
## 2.5. correlation heat map

In [180]:

```
#Put the target as last feature
y=df.y
df=df.drop("y",axis=1)
df["y"]=y
cor = df.corr().round(1)
mask = np.zeros_like(cor, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(220, 10, as_cmap=True)
plt.figure(figsize=(20, 15))
heatmap=sns.heatmap(cor,mask=mask,annot=True,annot_kws={"size": 10},
                    center=0,cmap=cmap,square=True, linewidths=.5,
                    cbar_kws={"shrink": .5},yticklabels=1,xticklabels=1)
plt.title("Correlation Matrix", fontsize='xx-large', fontweight='bold')
```

Out[180]:

```
Text(0.5, 1, 'Correlation Matrix')
```



we can see from heatmap that the highest correlate features (abs(0.9) and above) the features the economic features: **["nr.employes"-"emp.var.rate"],["cons.price.idx"-"emp.var.rate"] ["euribor3m"-"emp.var.rate"],["nr.employes"-"euribor3m"],["poutcome_nonexsist-previous"], ["poutcome_nonexsist-poutcome_failure"]** nr.employed and emp.var.rate are *highly corelated* and also nr.employed and euribor3m are highly corelated.
because that we will remove emp.var.rate and euribor3m

In [181]:

```
#Delete the features above at 1 command
df=df.drop(["euribor3m","emp.var.rate","poutcome_nonexistent","marital_single","pdays_missing"],axis=1)
```

In [182]:

```
df_copy_feature_filtered=df.copy()
```

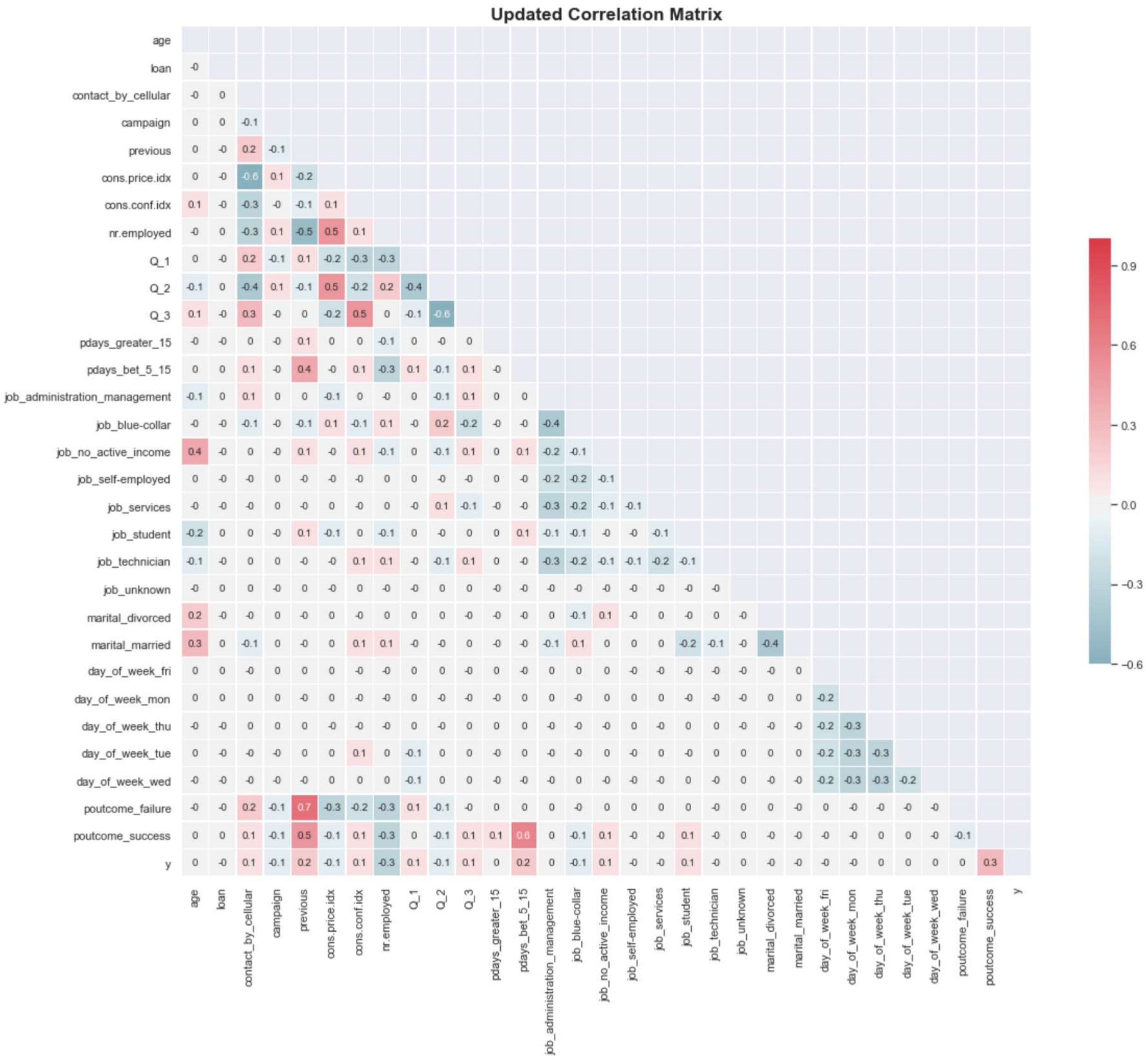
Now we want to see the updated correlation matrix

In [183]:

```
cor = df.corr().round(1)
mask = np.zeros_like(cor, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(220, 10, as_cmap=True)
plt.figure(figsize=(20, 15))
heatmap=sns.heatmap(cor,mask=mask,annot=True,annot_kws={"size": 10},
                    center=0,cmap=cmap,square=True, linewidths=.5,
                    cbar_kws={"shrink": .5},yticklabels=1,xticklabels=1)
plt.title("Updated Correlation Matrix", fontsize='xx-large', fontweight='bold')
```

Out[183]:

```
Text(0.5, 1, 'Updated Correlation Matrix')
```



# 4. Data Normalization

## 4.1. Box Plot

**Outliers: Outliers are defined as 1.5 x Q3 value (75th percentile).**

In [184]:

```
feature_lst=["cons.price.idx","nr.employed","cons.conf.idx","age","campaign","previous"
]
df_with_outliers=df.copy()
```

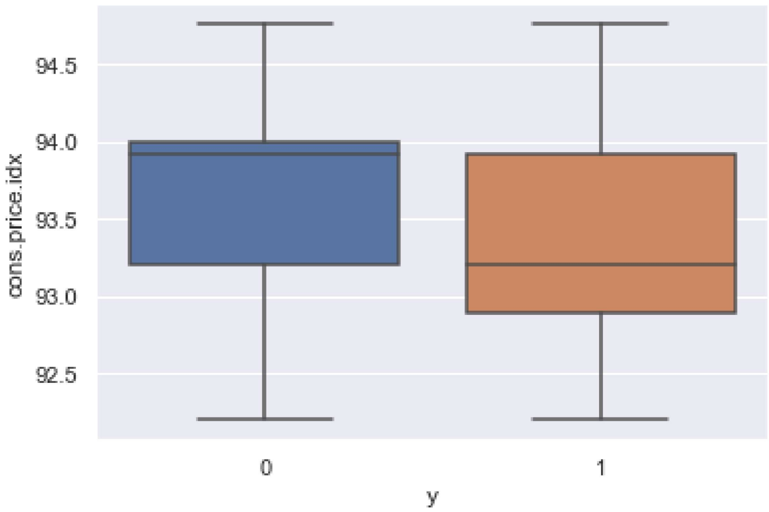
**cons.price.idx**

In [185]:

```
sns.boxplot(x='y', y="cons.price.idx", data=df_with_outliers)
```

Out[185]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fedcfd9f60>
```




There are no outliers for this feature.

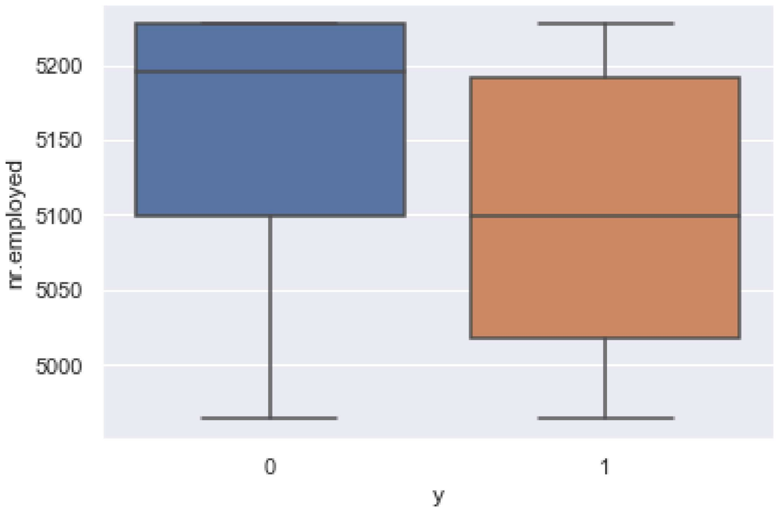
**nr.employed**

In [186]:

```
sns.boxplot(x='y', y="nr.employed", data=df_with_outliers)
```

Out[186]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fef060c0f0>
```




There are no outliers for this feature.

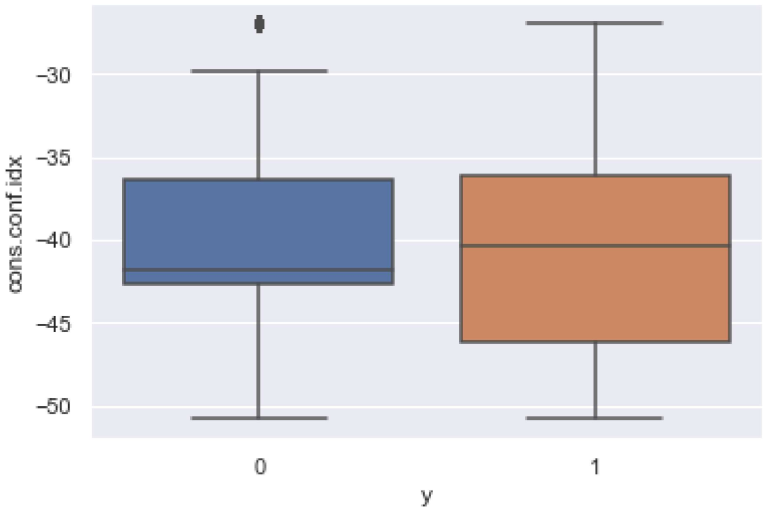
**cons.conf.idx**

In [187]:

```
sns.boxplot(x='y', y="cons.conf.idx", data=df_with_outliers)
```

Out[187]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fedaee86a0>
```



There are some unusual results in the target variable "no", but these do not significantly exceed the upper limit.

Then, they fit the upper bound of the target variable "yes".

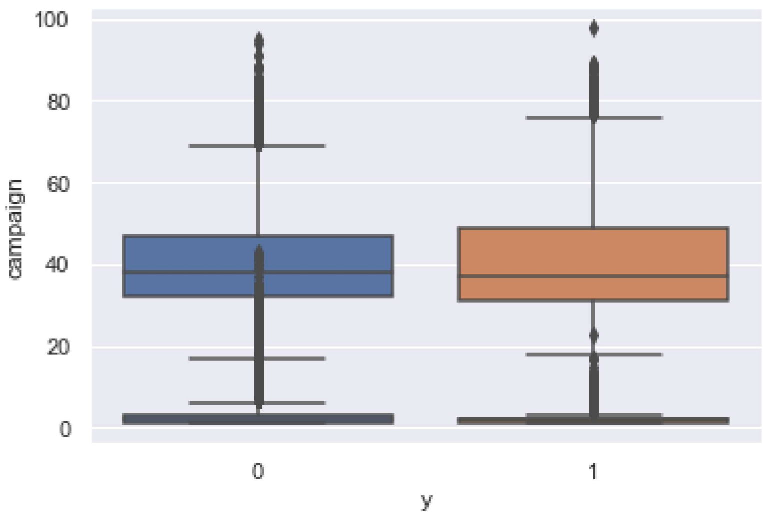
Therefore, we chose to leave it.

In [188]:

```
sns.boxplot(x='y', y="age", data=df_with_outliers)
sns.boxplot(x='y', y="campaign", data=df_with_outliers)
```

Out[188]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fed9892908>
```




We have outliers as max('age') and max('campaign') > 1.5Q3('age') and >1.5Q3('campaign') respectively.

But we also see that the value of these outliers are not so unrealistic (max('age')=98 and max('campaign')=56).

Hence, we need not remove them since the prediction model should represent the real world.

This improves the generalizability of the model and makes it robust for real world situations.

The outliers, therefore, are not removed.

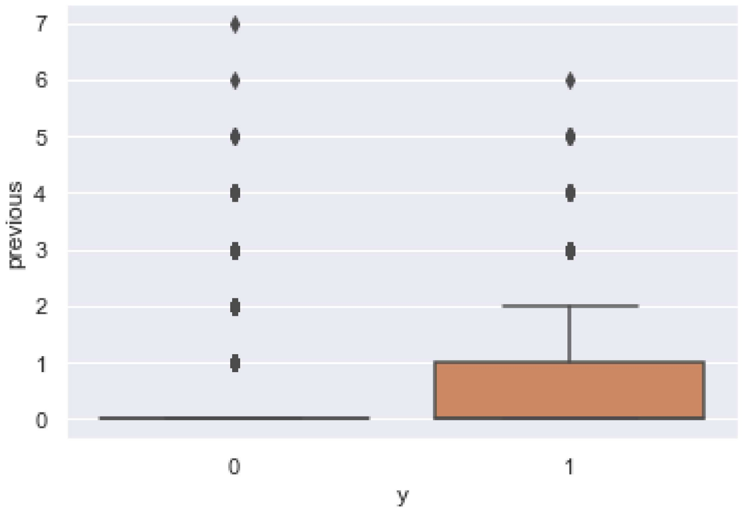
**"previous"**

In [189]:

```
sns.boxplot(x='y', y="previous", data=df_with_outliers)
```

Out[189]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1fed65feb38>
```




This variable has many unusual results, from the database, The unusual results belong to many calls made to a customer and therefore the outlierresults are much higher.

We decided to sift the top results that exceed 3 times the upper limit, leaving the other results less than the top limit. we will do it at 5.3.

In [190]:

```
df_outliers=df_with_outliers[["previous"]]
```

## 4.2. Normalize features

In [191]:

```
from sklearn.preprocessing import MinMaxScaler
numeric_df = df.select_dtypes(exclude="object")
scaler = MinMaxScaler(feature_range = (0,1))
normalized_df_data =scaler.fit_transform(numeric_df.values)
df_scaled=pd.DataFrame(normalized_df_data,columns=numeric_df.columns)
```

In [192]:

```
df_scaled
```

Out[192]:

| | age | loan | contact_by_cellular | campaign | previous | cons.price.idx | cons.conf.idx |
|---|---|---|---|---|---|---|---|
| **0** | 0.271605 | 0.0 | 1.0 | 0.047619 | 0.000000 | 0.669135 | 0.338912 |
| **1** | 0.419753 | 0.0 | 0.0 | 0.214286 | 0.000000 | 0.882307 | 0.376569 |
| **2** | 0.419753 | 1.0 | 0.0 | 0.000000 | 0.000000 | 0.882307 | 0.376569 |
| **3** | 0.444444 | 0.0 | 0.0 | 0.023810 | 0.000000 | 0.698753 | 0.602510 |
| **4** | 0.283951 | 0.0 | 1.0 | 0.000000 | 0.142857 | 0.269680 | 0.192469 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **34093** | 0.518519 | 0.0 | 1.0 | 0.023810 | 0.142857 | 0.000000 | 0.811715 |
| **34094** | 0.407407 | 0.0 | 0.0 | 0.023810 | 0.000000 | 0.882307 | 0.376569 |
| **34095** | 0.358025 | 0.0 | 0.0 | 0.023810 | 0.000000 | 0.389322 | 0.368201 |
| **34096** | 0.333333 | 0.0 | 0.0 | 0.000000 | 0.000000 | 0.882307 | 0.376569 |
| **34097** | 0.135802 | 0.0 | 0.0 | 0.023810 | 0.000000 | 0.698753 | 0.602510 |

34098 rows × 31 columns

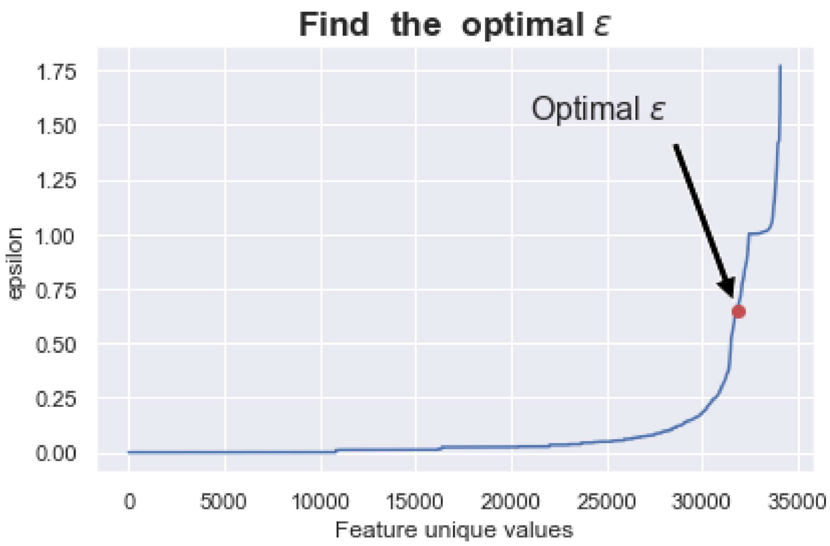
# 5. Outlier Detection

## 5.1. Clustrering by DBSCAN

*The optimal value for epsilon will be found at the point of maximum curvature.*

In [193]:

```
from sklearn.neighbors import NearestNeighbors
neigh = NearestNeighbors(n_neighbors=2)
nbrs = neigh.fit(df_scaled)
distances, indices = nbrs.kneighbors(df_scaled)
distances = np.sort(distances, axis=0)
distances = distances[:,1]
plt.plot(distances)
plt.title("Find  the  optimal "+r'$  \varepsilon$',fontsize='xx-large', fontweight='bol
d')
plt.ylabel("epsilon")
plt.xlabel("Feature unique values")
plt.plot([31825], [0.65], 'ro')
plt.annotate('Optimal '+r'$\varepsilon$', (31825,0.65),
            xytext=(0.8, 0.9), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=16,
            horizontalalignment='right', verticalalignment='top')
plt.tight_layout()
```




eps = the best epsilon is at the "elbow" of NearestNeighbors graph

In [194]:

```
from sklearn.cluster import DBSCAN
db = DBSCAN(eps=0.65,min_samples=5).fit(df_scaled)
labels=db.labels_
clusterNum=len(set(labels))
print("number of clusters is "+str(clusterNum))
noise=np.count_nonzero(labels == -1)
noise_percentage=round(100*noise/df_scaled.shape[0],0)
print("Number of outliers is "+str(noise)+ ", Noise accounts for "+str(noise_percentage
)+"%  of the total dataset" )
```

```
number of clusters is 956
Number of outliers is 5332, Noise accounts for 16.0%  of the total dataset
```

In [195]:

```
df=df_scaled
df_DBSCAN=df.copy()
df_DBSCAN["cluster_Db"]=labels
df_DBSCAN = df_DBSCAN[df_DBSCAN.cluster_Db != -1]
df_DBSCAN=df_DBSCAN.drop("cluster_Db",axis=1)
```

### 5.2. Multiple Clustrers by DBSCAN

We found out that the best epsilon is just above the "elbow" - not that much noisy (about 15%) but lots of clusters that can define us different groups with different features. **Lots of clusters means low number of noise, therefore low number of outliers.**

### 5.3. Another method to remove outliers - LOF

In [196]:

```
from scipy import stats
z = np.abs(stats.zscore(df_outliers))
#define a threshold to identify an outlier
threshold = 3
df_outliers=df_outliers[(z < threshold).all(axis=1)]
Num_outliers_2nd=df_with_outliers.shape[0]-df_outliers.shape[0]
print("outliers by each feature boxplot: "+str(Num_outliers_2nd))
from sklearn.neighbors import LocalOutlierFactor
# fit the model for outlier detection (default)
n_outliers = Num_outliers_2nd
ground_truth = np.ones(len(df_scaled), dtype=int)
ground_truth[-n_outliers:] = -1
LOF = LocalOutlierFactor(n_neighbors=20, contamination=0.1)
# use fit_predict to compute the predicted labels of the training samples
# (when LOF is used for outlier detection, the estimator has no predict,
# decision_function and score_samples methods).
y_pred = LOF.fit_predict(df_scaled)
n_errors = (y_pred != ground_truth).sum()
X_scores = LOF.negative_outlier_factor_
LOF_outliers=np.count_nonzero(y_pred == -1)
LOF_outliers_percentage=round(100*np.count_nonzero(y_pred == -1)/df_scaled.shape[0],0)
print("Number of outliers by LOF is "+str(LOF_outliers)+ ", Noise accounts for "+str(LO
F_outliers_percentage)+"%  of the total dataset" )
```

```
outliers by each feature boxplot: 814
Number of outliers by LOF is 3410, Noise accounts for 10.0%  of the total
dataset
```

we can learn that local outliers (from each feature) does not predict the total outliers of the whole dataset by the combination of features. That is why we have more outliers by LOF **and** DBSCAN than that shown in the boxplots above.

# 6. Predictive Models

In [197]:

```
#%% 6. Predictive Models
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import RandomizedSearchCV
from pprint import pprint
from sklearn.utils import resample
kfold = model_selection.KFold(n_splits=3)
y=df_DBSCAN.y
X=df_DBSCAN.drop("y",axis=1)
x_train, x_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.2
, random_state=0) #80/20 split
```

Data is imbalanced, therefore we used Upsampling. Due to overfitting results due to equal size targets values, **we decided to optimize this option** - we don't equal the two targets [0,1], we only take a portion of samples add add to our minority target

In [198]:

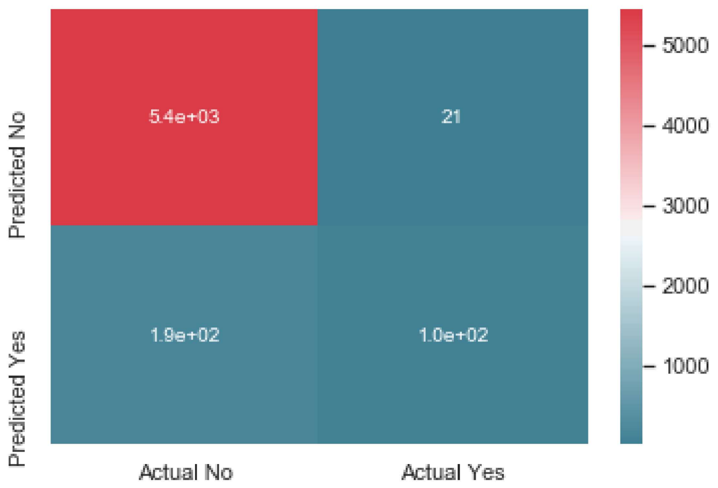
```
y_train_no_yes=y_train.value_counts()
df_majority = df_DBSCAN[df_DBSCAN.y==0]
df_minority = df_DBSCAN[df_DBSCAN.y==1]
# Upsample minority class
df_minority_upsampled = resample(df_minority,
                                 replace=True,     # sample with replacement
                                 n_samples=int(y_train_no_yes[0]/16),    # take a bunch
from majority class
                                 random_state=123) # reproducible results
df_upsampled = pd.concat([df_majority, df_minority_upsampled])
df_upsampled.y.value_counts()
y_upsample=df_upsampled.y
X_upsample=df_upsampled.drop("y",axis=1)
x_train_up, x_test_up, y_train_up, y_test_up = model_selection.train_test_split(X_upsam
ple, y_upsample, test_size=0.2, random_state=0) #80/20 split
x_train=x_train_up
y_train= y_train_up
```

## 6.1. Random Forest

In [199]:

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
#explore the hyperparameters of this classifier
pprint(rfc.get_params())
# Number of trees in random forest
n_estimators =[100,200,300]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [3,6,8]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 8]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
pprint(random_grid)
# search across 100 different combinations, and use all available cores
rf_random = RandomizedSearchCV(estimator = rfc,
                               param_distributions = random_grid,
                               n_iter = 100, cv = kfold, verbose=2,
                               random_state=42, n_jobs = -1)
rf_random.fit(x_train, y_train)
print("best parameters are:")
print(rf_random.best_params_)
rf_best_random = rf_random.best_estimator_
prediction_RF = rf_best_random.predict(x_test)
print(classification_report(y_test, prediction_RF,target_names=["no","yes"]))
CM_RF=confusion_matrix(y_test, prediction_RF)
df_cm = pd.DataFrame(CM_RF, index = ["Predicted No","Predicted Yes"],
                  columns = ["Actual No","Actual Yes"])
plt.figure()
sns.heatmap(df_cm,cmap=cmap, annot=True)
#AUC
probs_RF = rf_random.predict_proba(x_test)
preds_RF = probs_RF[:,1]
fprrfc, tprrfc, thresholdrfc = metrics.roc_curve(y_test, preds_RF)
roc_aucrfc = metrics.auc(fprrfc, tprrfc)
```

```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': 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': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
{'bootstrap': [True, False],
 'max_depth': [3, 6, 8, None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 8],
 'n_estimators': [100, 200, 300]}
Fitting 3 folds for each of 100 candidates, totalling 300 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed:   25.8s
[Parallel(n_jobs=-1)]: Done 154 tasks      | elapsed:  2.0min
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed:  3.6min finished

best parameters are:
{'n_estimators': 200, 'min_samples_split': 5, 'min_samples_leaf': 2, 'max_
features': 'sqrt', 'max_depth': None, 'bootstrap': False}
              precision    recall  f1-score   support

          no       0.97      1.00      0.98      5463
         yes       0.83      0.36      0.50       291

    accuracy                           0.96      5754
   macro avg       0.90      0.68      0.74      5754
weighted avg       0.96      0.96      0.96      5754
```

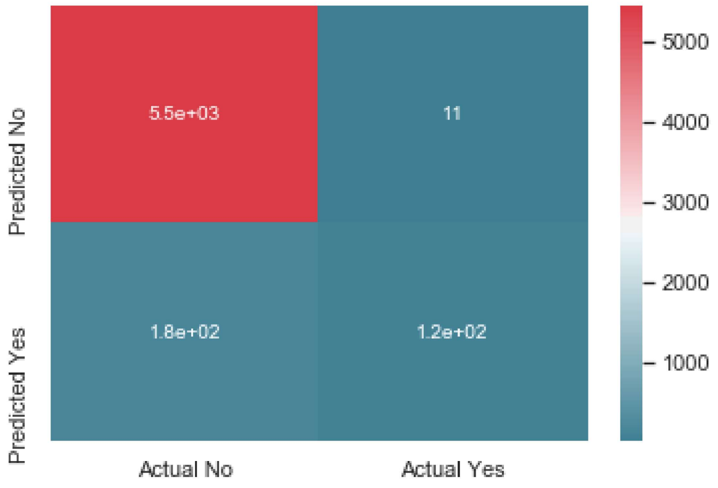


## 6.2. ADABOOST

In [200]:

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
ADA = AdaBoostClassifier()
#explore the hyperparameters of this classifier
pprint(ADA.get_params())
#learning rate shrinks the contribution of each tree by learning_rate.
learning_rate = [0.05,0.1,0.2]
#algorithm ==================================================================
# If ‘SAMME.R’ then use the SAMME.R real boosting algorithm.
# base_estimator must support calculation of class probabilities.
# If ‘SAMME’ then use the SAMME discrete boosting algorithm.
# The SAMME.R algorithm typically converges faster than SAMME,
# achieving a lower test error with fewer boosting iterations.
# ===========================================================================
algorithm = ["SAMME", "SAMME.R"]
n_estimators = [50,100,200]
#The base estimator from which the boosted ensemble is built
base_estimator= [DecisionTreeClassifier(max_depth=int(x)) for x in [3,6,8]]
base_estimator.append(None)
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'algorithm': algorithm,
               'base_estimator': base_estimator,
               'learning_rate':learning_rate}
pprint(random_grid)
# search across 100 different combinations, and use all available cores
ADA_random = RandomizedSearchCV(estimator = ADA,
                               param_distributions = random_grid,
                               n_iter = 100, cv = kfold, verbose=2,
                               random_state=42, n_jobs = -1)
ADA_random.fit(x_train, y_train)
print("Best parameters are:")
print(ADA_random.best_params_)
ADA_best_random = ADA_random.best_estimator_
predictions_ADA = ADA_best_random.predict(x_test)
CM_ADA=confusion_matrix(y_test, predictions_ADA)
df_cm = pd.DataFrame(CM_ADA, index = ["Predicted No","Predicted Yes"],
                  columns = ["Actual No","Actual Yes"])
plt.figure()
sns.heatmap(df_cm,cmap=cmap, annot=True)
print(classification_report(y_test, predictions_ADA,target_names=["no","yes"]))
#AUC
probs_ADA = ADA_random.predict_proba(x_test)
preds_ADA = probs_ADA[:,1]
fprADA, tprADA, thresholdADA = metrics.roc_curve(y_test, preds_ADA)
roc_aucADA = metrics.auc(fprADA, tprADA)
```

```
{'algorithm': 'SAMME.R',
 'base_estimator': None,
 'learning_rate': 1.0,
 'n_estimators': 50,
 'random_state': None}
{'algorithm': ['SAMME', 'SAMME.R'],
 'base_estimator': [DecisionTreeClassifier(ccp_alpha=0.0, class_weight=Non
e, criterion='gini',
                       max_depth=3, max_features=None, max_leaf_nodes=Non
e,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best'),
                    DecisionTreeClassifier(ccp_alpha=0.0, class_weight=Non
e, criterion='gini',
                       max_depth=6, max_features=None, max_leaf_nodes=Non
e,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best'),
                    DecisionTreeClassifier(ccp_alpha=0.0, class_weight=Non
e, criterion='gini',
                       max_depth=8, max_features=None, max_leaf_nodes=Non
e,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best'),
                    None],
 'learning_rate': [0.05, 0.1, 0.2],
 'n_estimators': [50, 100, 200]}
Fitting 3 folds for each of 72 candidates, totalling 216 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed:   37.7s
[Parallel(n_jobs=-1)]: Done 154 tasks      | elapsed:  4.1min
[Parallel(n_jobs=-1)]: Done 216 out of 216 | elapsed:  6.2min finished

Best parameters are:
{'n_estimators': 100, 'learning_rate': 0.05, 'base_estimator': DecisionTre
eClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=8, max_features=None, max_leaf_nodes=Non
e,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best'), 'algorithm':
'SAMME.R'}
              precision    recall  f1-score   support

          no       0.97      1.00      0.98      5463
         yes       0.91      0.40      0.55       291

    accuracy                           0.97      5754
   macro avg       0.94      0.70      0.77      5754
weighted avg       0.97      0.97      0.96      5754
```




## 6.3. Gradient Boosting

In [201]:

```
from sklearn.ensemble import GradientBoostingClassifier
grd = GradientBoostingClassifier()
#explore the hyperparameters
pprint(grd.get_params())
# Number of trees in random forest
n_estimators = [100,200,300]
# Number of features to consider at every split
max_features = ['auto', 'sqrt',"log2"]
# Maximum number of levels in tree
max_depth = [3,6,8]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 8]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
#learning rate shrinks the contribution of each tree by learning_rate.
learning_rate=[0.05,0.1,0.2]
#loss function to be optimized
loss=["deviance", "exponential"]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
                'max_features': max_features,
                'max_depth': max_depth,
                'min_samples_split': min_samples_split,
                'min_samples_leaf': min_samples_leaf,
                'loss' : loss,
                'learning_rate': learning_rate}
pprint(random_grid)
#search across 100 different combinations, and use all available cores
grd_random = RandomizedSearchCV(estimator = grd,
                                param_distributions = random_grid,
                                n_iter = 100, cv = kfold, verbose=2,
                                random_state=42, n_jobs = -1)
grd_random.fit(x_train, y_train)
grd_random.best_params_
grd_best_random = grd_random.best_estimator_
predictions_grd = grd_best_random.predict(x_test)
print("for Gradient Boosting we get " +str(round(accuracy_score(y_test, predictions_grd
),5)))
CM_grd=confusion_matrix(y_test, predictions_grd)
df_cm = pd.DataFrame(CM_grd, index = ["Predicted No","Predicted Yes"],
                  columns = ["Actual No","Actual Yes"])
print(classification_report(y_test, predictions_grd,target_names=["no","yes"]))
#AUC
probs_grd = grd_random.predict_proba(x_test)
preds_grd = probs_grd[:,1]
fprgrd, tprgrd, thresholdgrd = metrics.roc_curve(y_test, preds_grd)
roc_aucgrd = metrics.auc(fprgrd, tprgrd)
```

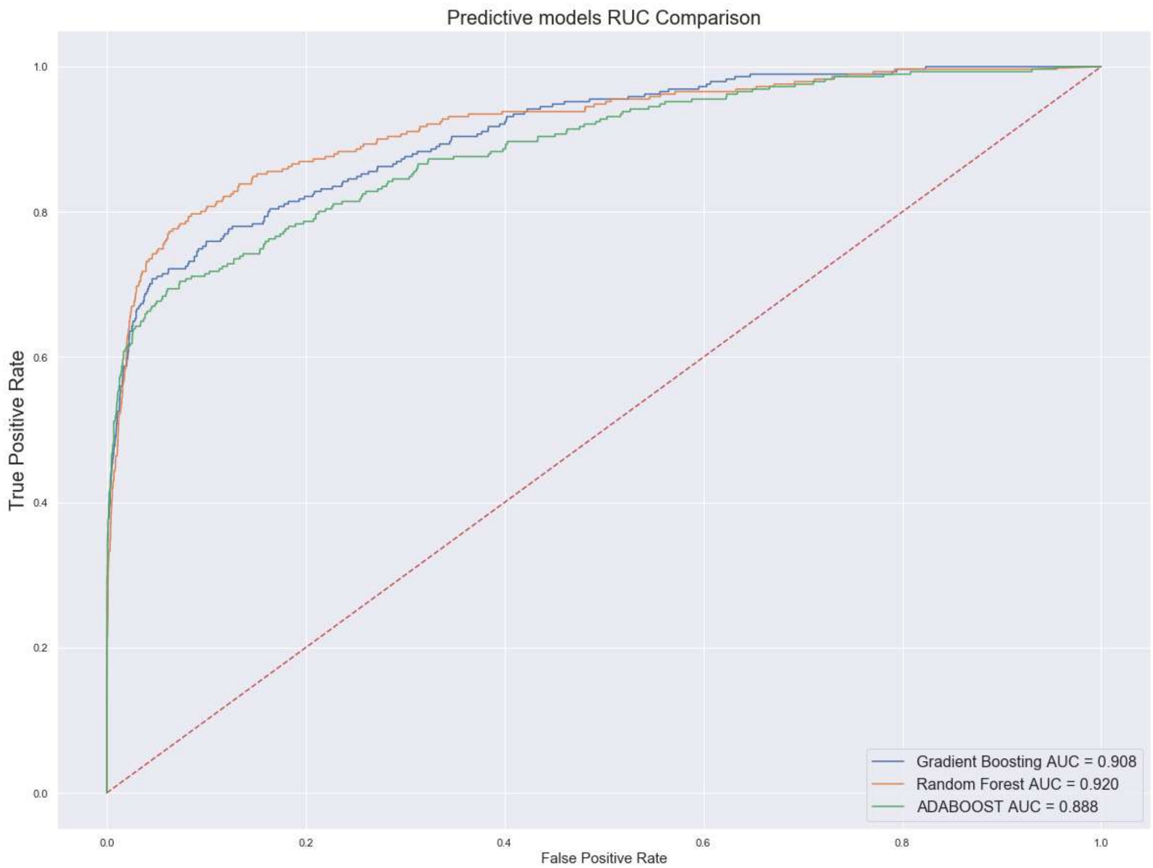
```
{'ccp_alpha': 0.0,
 'criterion': 'friedman_mse',
 'init': None,
 'learning_rate': 0.1,
 'loss': 'deviance',
 'max_depth': 3,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_iter_no_change': None,
 'presort': 'deprecated',
 'random_state': None,
 'subsample': 1.0,
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
{'learning_rate': [0.05, 0.1, 0.2],
 'loss': ['deviance', 'exponential'],
 'max_depth': [3, 6, 8, None],
 'max_features': ['auto', 'sqrt', 'log2'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 8],
 'n_estimators': [100, 200, 300]}
Fitting 3 folds for each of 100 candidates, totalling 300 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed:  4.0min
[Parallel(n_jobs=-1)]: Done 154 tasks      | elapsed: 10.8min
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed: 22.9min finished

for Gradient Boosting we get 0.96715
              precision    recall  f1-score   support

          no       0.97      1.00      0.98      5463
         yes       0.93      0.38      0.54       291

    accuracy                           0.97      5754
   macro avg       0.95      0.69      0.76      5754
weighted avg       0.97      0.97      0.96      5754
```

### AUC Curves

In [203]:

```
sns.set()
plt.figure(figsize=(20, 15))
plt.plot([0, 1], [0, 1],'r--')
plt.title('Predictive models RUC Comparison',fontsize=20)
plt.ylabel('True Positive Rate',fontsize=20)
plt.xlabel('False Positive Rate',fontsize=15)
plt.plot(fprgrd, tprgrd, label = 'Gradient Boosting AUC = %0.3f' % roc_aucgrd)
plt.plot(fprrfc, tprrfc, label = 'Random Forest AUC = %0.3f' % roc_aucrfc)
plt.plot(fprADA, tprADA, label = 'ADABOOST AUC = %0.3f' % roc_aucADA)
plt.legend(loc = 'lower right', prop={'size': 16})
print("--- %s minutes ---" % (round(time.time()/60 - start_time/60,2)))
```

```
--- 399.59 minutes ---
```




In [ ]: