Survey for Term Deposit

Some details about the dataset regarding customer :

Age : Customers age (numeric)
Job : Type of job (categorical)
Marital Status : Marital status
Education : Level of education of the customer
Default : has credit in default? (binary: "yes","no")
Balance : average yearly balance, in euros (numeric)
Housing : has housing loan? (binary: "yes","no")
Loan : has personal loan? (binary: "yes","no")
Contact : contact communication type (categorical)
Day : last contact day of the month (numeric)
Month : last contact month of year (categorical)
Duration : last contact duration, in seconds (numeric),
Campaign :number of contacts performed during this campaign and for this client (numeric, includes last contact)
Pdays : number of days that passed by after the client was last contacted from a previous campaign (numeric, -1 means client was not previously contacted)
Previous : number of contacts performed before this campaign and for t his client (numeric)
Poutcome :outcome of the previous marketing campaign (categorical: "unknown","other","failure","success")
y : has the client subscribed a term deposit? (binary : 0, 1)

Aim:

Building a machine learning model to accurately classify whether or not the customer who is being targeted in the dataset is going to opt term deposit in that particular bank or not

Dataset Source:

This is a bank telemarketing data. Got this data set from UCI machine learning repository(as per kaggle source).
Link: https://www.kaggle.com/sharanmk/bank-marketing-term-deposit

Steps involved in project:

Importing necessary packages

Importing dataset csv files

Looking at the data set shape, number and types of variables, and the overall distribution of the numerical variables.

Data Cleaning (eg:drop duplicates (if any))

```
df.shape
```

```
Out[6]: (45211, 17)
```

```
In [ ]: #no duplicate values
```

```
In [9]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45211 entries, 0 to 45210
Data columns (total 17 columns):
age          45211 non-null int64
job          45211 non-null object
marital      45211 non-null object
education    45211 non-null object
default      45211 non-null object
balance      45211 non-null int64
housing      45211 non-null object
loan         45211 non-null object
contact      45211 non-null object
day          45211 non-null int64
month        45211 non-null object
duration     45211 non-null int64
campaign     45211 non-null int64
pdays        45211 non-null int64
previous     45211 non-null int64
poutcome     45211 non-null object
y            45211 non-null int64
dtypes: int64(8), object(9)
memory usage: 6.2+ MB
```

convert all variables of the type “object” into categorical variables, so that they are stored properly:

```
In [12]: df.info()
```

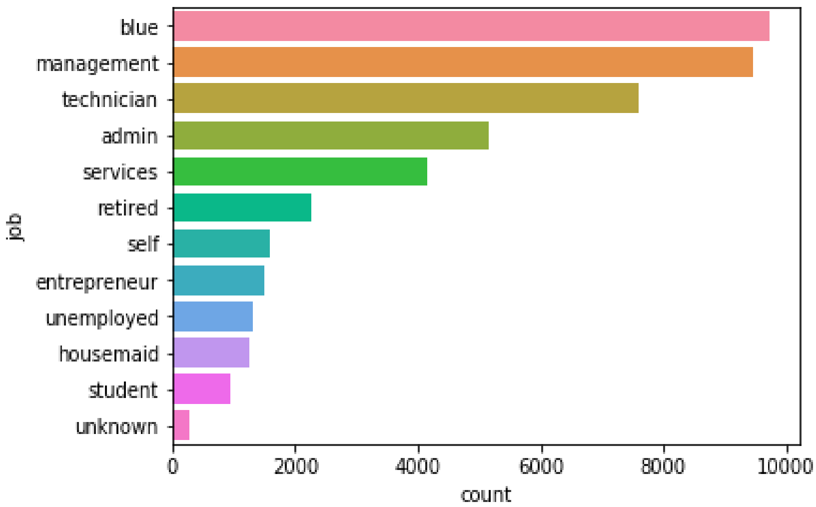
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45211 entries, 0 to 45210
Data columns (total 17 columns):
age          45211 non-null int64
job          45211 non-null category
marital      45211 non-null category
education    45211 non-null category
default      45211 non-null category
balance      45211 non-null int64
housing      45211 non-null category
loan         45211 non-null category
contact      45211 non-null category
day          45211 non-null int64
month        45211 non-null category
duration     45211 non-null int64
campaign     45211 non-null int64
pdays        45211 non-null int64
previous     45211 non-null int64
poutcome     45211 non-null category
y            45211 non-null int64
dtypes: category(9), int64(8)
memory usage: 3.5 MB
```

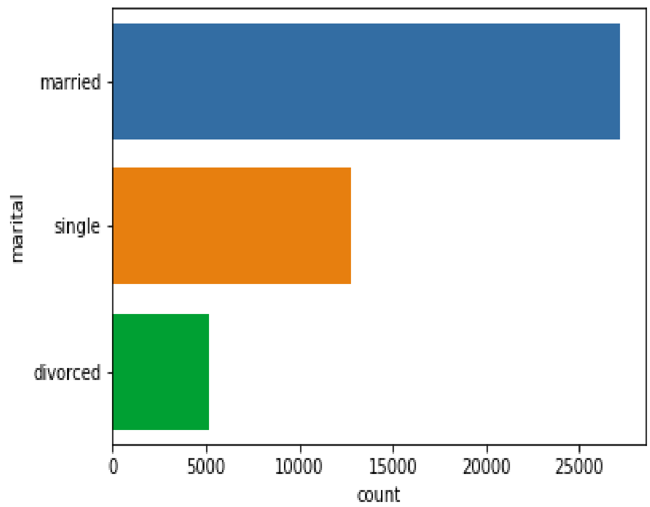
```
In [13]: df.head(15)
```

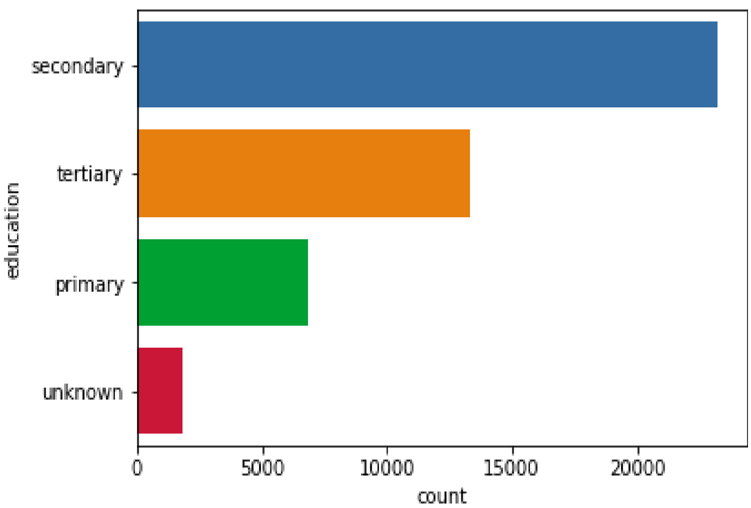
Out[13]:

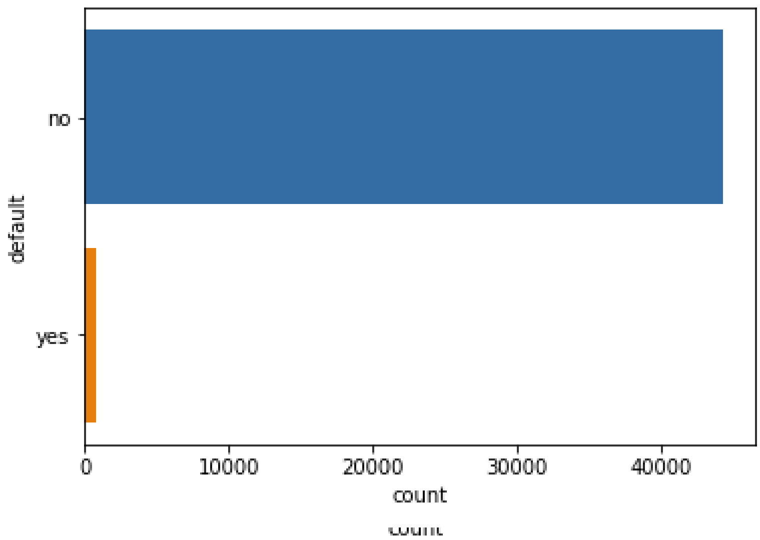
	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	0

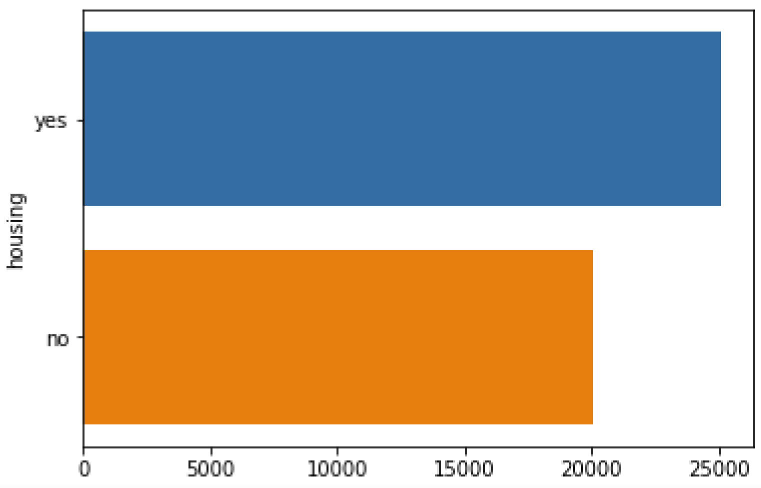
The easiest way to understand the distribution of the categorical variables would be to plot bar plots. I use value_counts() method to sort the bars in descending order.

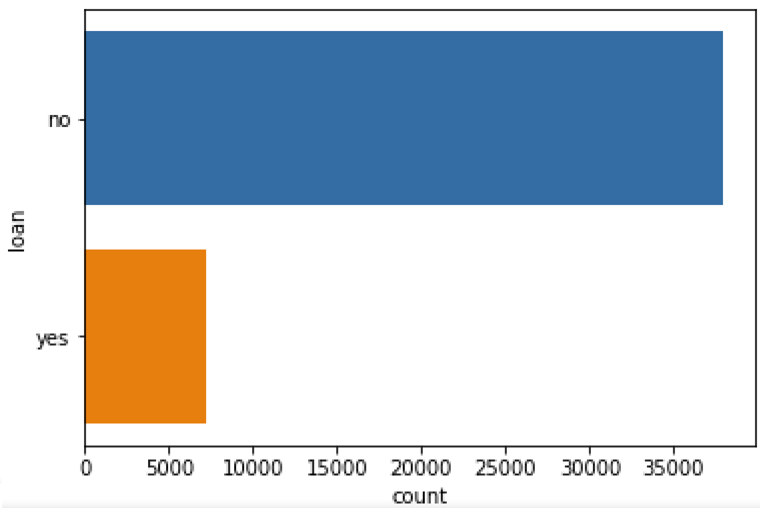


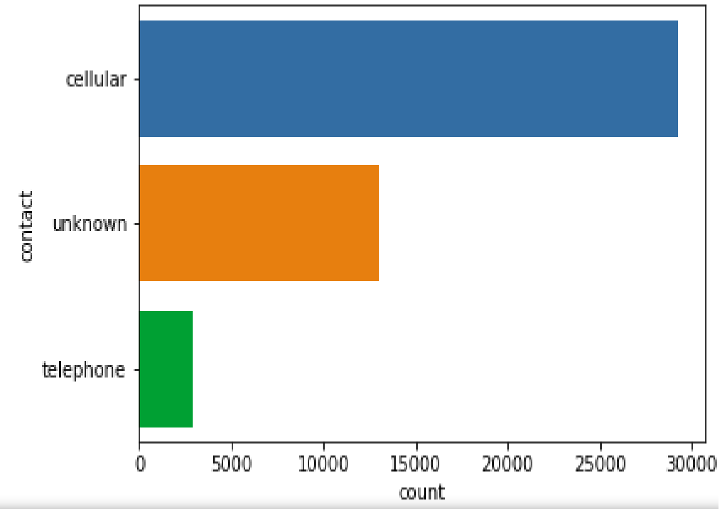


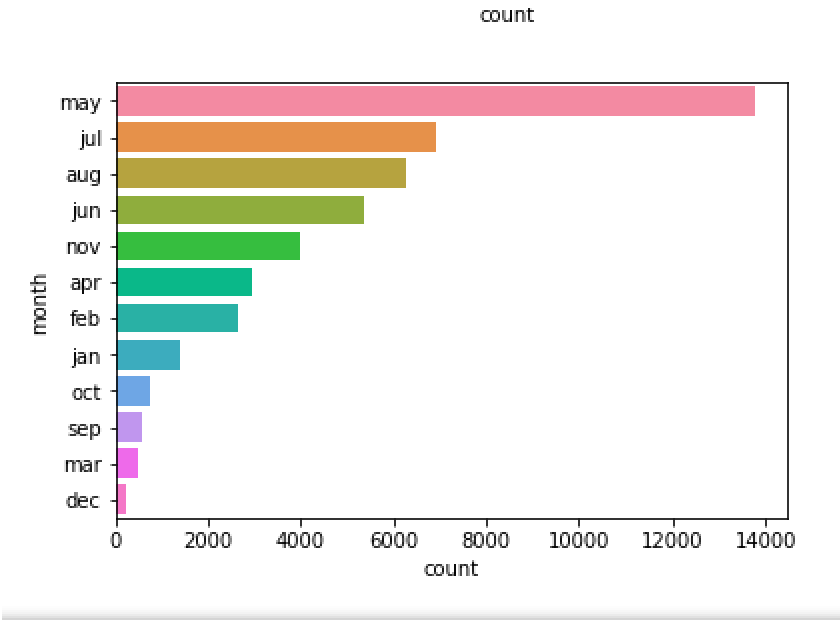


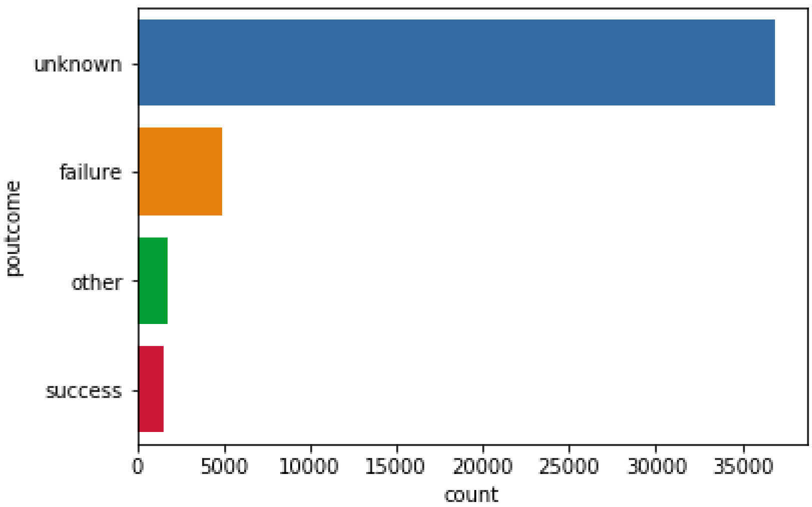












let’s look at the levels of the categorical variables as proportions:

```
In [19]: #job status as proportion of overall number of values
         df.job.value_counts()/45211

Out[19]: blue             0.215257
         management       0.209197
         technician       0.168034
         admin            0.114375
         services         0.091880
         retired          0.050076
         self             0.034925
         entrepreneur     0.032890
         unemployed       0.028820
         housemaid        0.027427
         student          0.020747
         unknown          0.006370
         Name: job, dtype: float64

In [25]: #default credit status as proportion of overall number of values
         df.default.value_counts()/45211

Out[25]: no      0.981973
         yes     0.018027
         Name: default, dtype: float64

In [26]: #housing loan status as proportion of overall number of values
         df.housing.value_counts()/45211

Out[26]: yes     0.555838
         no      0.444162
         Name: housing, dtype: float64
```

In [27]:
```
#personal status as proportion of overall number of values
df.loan.value_counts()/45211
```

Out[27]:
```
no     0.839774
yes    0.160226
Name: loan, dtype: float64
```

In [21]:
```
#maritial status as proportion of overall number of values
df.marital.value_counts()/45211
```

Out[21]:
```
married     0.601933
single      0.282896
divorced    0.115171
Name: marital, dtype: float64
```

In [22]:
```
#education status as proportion of overall number of values
df.education.value_counts()/45211
```

Out[22]:
```
secondary    0.513194
tertiary     0.294198
primary      0.151534
unknown      0.041074
Name: education, dtype: float64
```

In [23]:
```
#month status as proportion of overall number of values
df.month.value_counts()/45211
```

Out[23]:
```
may    0.304483
jul    0.152507
aug    0.138174
jun    0.118135
nov    0.087810
```

```
df.month.value_counts()/45211
```

Out[23]:

```
may    0.304483
jul    0.152507
aug    0.138174
jun    0.118135
nov    0.087810
apr    0.064851
feb    0.058592
jan    0.031032
oct    0.016323
sep    0.012807
mar    0.010551
dec    0.004733
Name: month, dtype: float64
```

In [24]:

```
#previous outcome status as proportion of overall number of values
df.poutcome.value_counts()/45211
```

Out[24]:

```
unknown    0.817478
failure    0.108403
other      0.040698
success    0.033421
Name: poutcome, dtype: float64
```

In [28]:

```
#Histogram grid
df.hist(figsize=(10,10), xrot=-45)
#Clear the text "residue"
plt.show()
```

categorical value : unknown in ‘Job’,’Marital’ and’Education’ hold very low proportion and hence can be eliminated

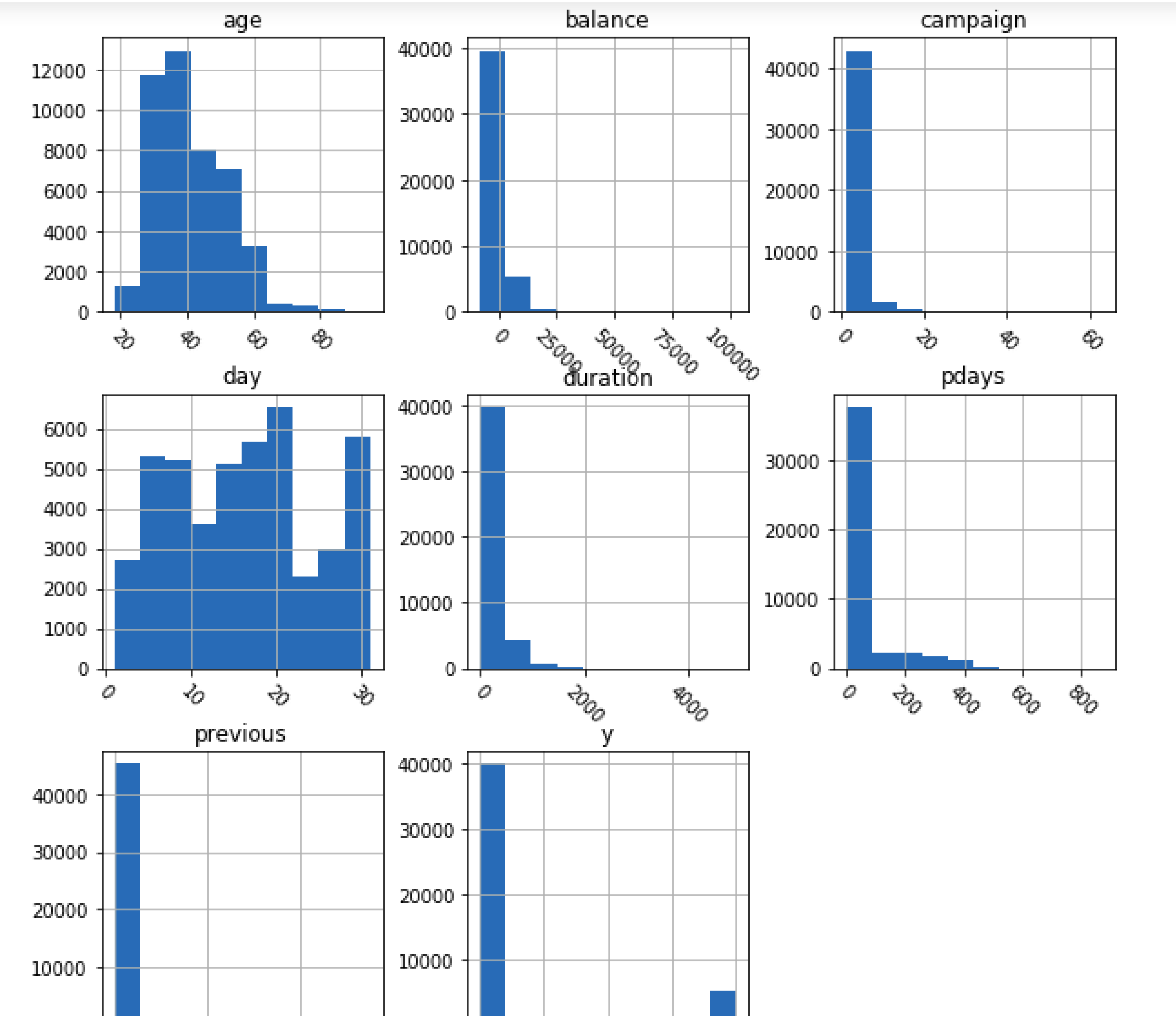
There is a very small number of respondents who defaulted on a credit, so this variable doesn’t look very useful for prediction purposes and can be dropped from the dataset.

Most of the respondents were contacted during the summer months, with more than 30% of all contacts happening in May. The month of contact can have a substantial impact on the desire to subscribe for a deposit (e.g., many people may be receiving salary bonuses at the end of the calendar year, which could be a good time to contact them about the deposit). This skewness of the previous campaigns’ efforts towards summer may potentially negatively impact the outcomes of the future campaigns, especially if the summer months prove to be negative predictors for campaign’s success.

Only ~11% of the respondents to the current campaign have actually subscribed for a deposit as a result of the campaign. This makes our data set highly imbalanced and requires application of special methods to compensate for it — a model built on this imbalanced data set without using any balancing approaches can be correct 88% of the time if it simply predicts "no" as a result and ignores the positive responses altogether.

Numerical Data

Distributions of the numerical variables:

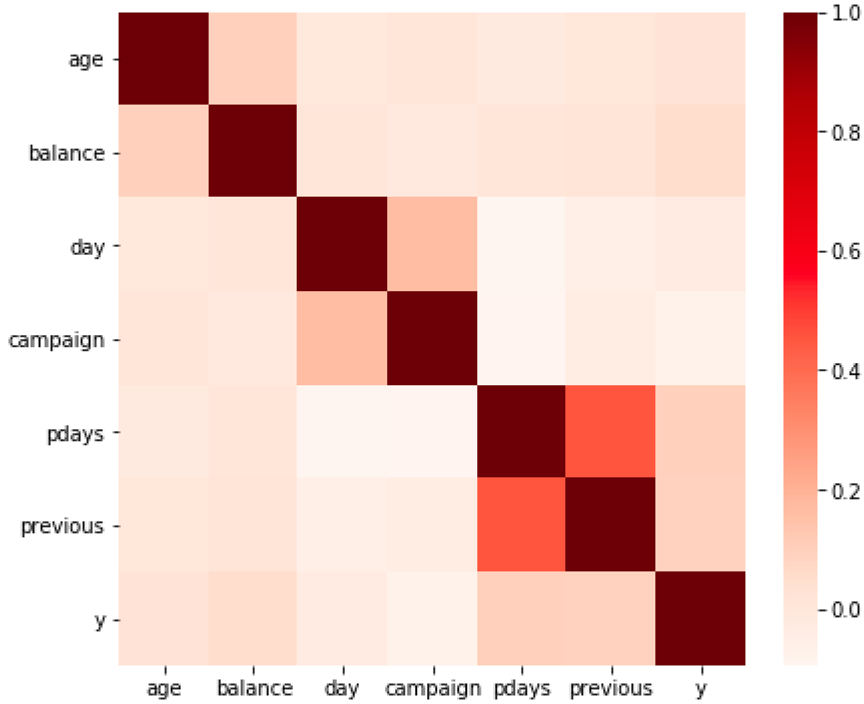


There are no obvious errors in the data. The economic indicators have several prominent peaks, but we don’t have any insight into what was causing those peaks.

To prepare the data set for using in a predictive model, we’ll remove the unknown values, redundant variables . The cleaned data set will be saved in the df_clean data frame.

First, let’s look at how the numeric variables in the dataset correlate with each other:

In [5]:
```
#Calculate correlations between numeric features
correlations = df_clean.corr()
#Make the figsize 7 x 6
plt.figure(figsize=(7,6))
_ = sns.heatmap(correlations, cmap="Reds")#heatmap
```



If the strong correlation between the above variables is statistically significant, we may potentially group or drop some of them, as having multiple strongly correlated variables in our prediction model will not substantially increase the accuracy of our model. Alternatively, we can use penalized prediction models to decrease the weights of the coefficients for the strongly correlated variables.

Fitting the Predictive Models

First, let's convert the levels of the categorical variables into dummy variables, using the standard function from Pandas. We will exclude one dummy variable level for each categorical variable to avoid collinearity (drop_first=True).

In [6]:
```
df_clean1 = pd.get_dummies(df_clean, drop_first=True)
df_clean1.head()
```

Out[6]:

	age	balance	day	campaign	pdays	previous	y	job_blue	job_entrepreneur	job_housemaid	...	month_jul	month_jun	month_mar	month_may	mont
0	58	2143	5	1	-1.0	0	0	0	0	0	...	0	0	0	1	
1	44	29	5	1	-1.0	0	0	0	0	0	...	0	0	0	1	
2	33	2	5	1	-1.0	0	0	0	1	0	...	0	0	0	1	
3	35	231	5	1	-1.0	0	0	0	0	0	...	0	0	0	1	
4	28	447	5	1	-1.0	0	0	0	0	0	...	0	0	0	1	

5 rows × 39 columns

The simplest and the most interpretable model to predict the categorical variable "y" would be the Logistic Regression. To identify how well this model works in our case, let's fit different types of the logistic regression model to identify the best model coefficients, using GradientSearchCV and pipelines. Since we have a highly imbalanced data classes (only ~10% of respondents subscribed for deposits), we will use the balanced class weight parameter (class_weight='balanced'):

```python
#Splitting variables into predictor and target variables
X=df_clean1.drop('y', axis=1)
y=df_clean1.y
#Setting up pipelines with a StandardScaler function to normalize variables
pipelines = {
    'log1':make_pipeline(StandardScaler(),LogisticRegression(penalty='l1',random_state=42,class_weight='balanced'))
    'log2' : make_pipeline(StandardScaler(), LogisticRegression(penalty='l2',random_state=42,class_weight='balanced
    #Setting the penalty for simple Logistic Regression as L2 to minimize the fitting time
    'log_reg' : make_pipeline(StandardScaler(), LogisticRegression(penalty='l2', random_state=42, class_weight='bal
}
#Setting up a very large hyperparameter C for the non-penalized Logistic Regression (to cancel the regularization)
log_reg_hyperparameters={
    'logisticregression__C':np.linspace(100000, 100001, 1),
    'logisticregression__fit_intercept':[True, False]
}
#Setting up hyperparameters for the Logistic Regression with log1 penalty
log1_hyperparameters = {
    'logisticregression__C' : np.linspace(1e-3, 1e3, 10),
    'logisticregression__fit_intercept' : [True, False]
}
#Setting up hyperparameters for the Logistic Regression with log2 penalty
log2_hyperparameters={
    'logisticregression__C':np.linspace(1e-3, 1e3, 10),
    'logisticregression__fit_intercept':[True, False]
}
#Creating the dictionary of hyperparameters
hyperparameters = {
    'log_reg' : log_reg_hyperparameters,
    'log1' : log1_hyperparameters,
    'log2' : log2_hyperparameters
}
#Splitting the data into train and test sets
```

In [8]:
```python
#Displaying best score for each fitted model
for name,model in fitted_logreg_models.items():
    print(name,model.best_score_)
```

```
log1 0.7613707783401145
log2 0.7613707783401145
log_reg 0.7613707783401145
```

In [19]:
```python
#defining the model with the highest accuracy score
max(predicted_logreg_models,key=lambda k:predicted_logreg_models[k])
```

Out[19]: 'log1'

Let’s calculate the confusion matrix and the classification report for the logistic regression model with the best accuracy score. Since the accuracy scores for log1- and log2-regularized models are practically the same, we’ll use the L2-model, as it requires much less computation

In [21]:
```python
#Creating the confusion matrix
pd.crosstab(y_test,fitted_logreg_models['log1'].predict(X_test),rownames=['True'],colnames=['Predict'],margins=True
```

```
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/pipeline.py:331: DataConversionWarning: Data with inpu
t dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```

Out[21]:

Predict	0	1	All
True			
0	9009	2458	11467
1	604	886	1490
All	9613	3344	12957

In [22]:
```python
#Creating the classification report
print(classification_report(y_test, fitted_logreg_models['log1'].predict(X_test)))
```

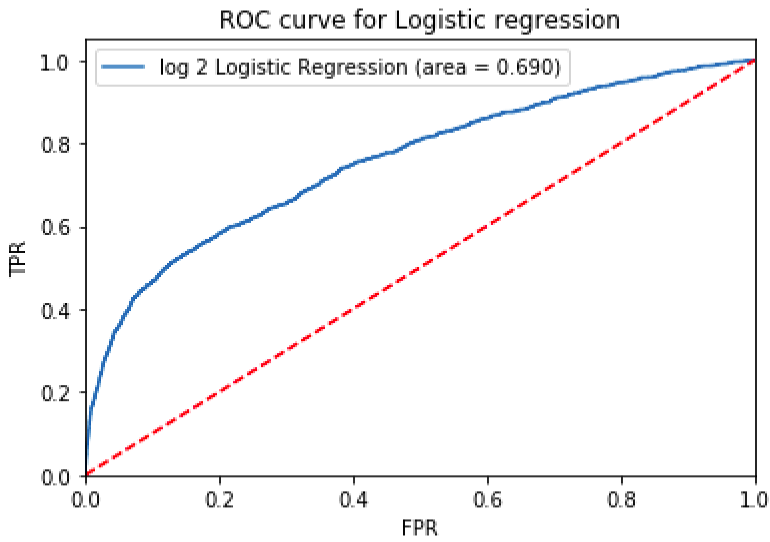
```
              precision    recall  f1-score   support

           0       0.94      0.79      0.85     11467
           1       0.26      0.59      0.37      1490

   micro avg       0.76      0.76      0.76     12957
   macro avg       0.60      0.69      0.61     12957
weighted avg       0.86      0.76      0.80     12957
```

The model's precision and recall for class 0 (respondents who didn't subscribe for a deposit) is pretty high, but for class 1 (the respondents who subscribed) it's low.

Let's also look at the ROC curve:



The area under the curve is ~0.69, which is substantially higher than the probability area for random guessing (0.5). Given the results of the classification report above, it could be assumed that the biggest contribution to the area under the ROC curve comes from the correctly identified class 0.

To summarize, the logistic regression doesn't seem to be a very good model to predict the respondents who may subscribe to a deposit. One of the reasons for bad prediction quality may be the highly imbalanced nature of the data set.

Testing other predictive models

Since the log1 logistic regression didn't perform very well, it makes sense to compare our model's performance with other classification models. Two of the most popular classification models are Random forest and Gradient boosting, which generally classify well. Let's fit these models and compare the accuracy scores:

In [9]:
```
#Setting up pipelines with a StandardScaler function to normalize the variables
pipelines = {
    'rf' : make_pipeline(StandardScaler(),RandomForestClassifier(random_state=42, class_weight='balanced')),
    'gb' : make_pipeline(StandardScaler(),GradientBoostingClassifier(random_state=42))
}
#Setting up the "rule of thumb" hyperparameters for the Random Forest
rf_hyperparameters = {
    'randomforestclassifier__n_estimators': [100, 200],
    'randomforestclassifier__max_features': ['auto', 'sqrt', 0.33]
}
#Setting up the "rule of thumb" hyperparameters for the Gradient Boost
gb_hyperparameters = {
    'gradientboostingclassifier__n_estimators': [100, 200],
    'gradientboostingclassifier__learning_rate': [0.05, 0.1, 0.2],
    'gradientboostingclassifier__max_depth': [1, 3, 5]
}
#Creating the dictionary of hyperparameters
hyperparameters = {
    'rf' : rf_hyperparameters,
    'gb' : gb_hyperparameters
}
#Creating an empty dictionary for fitted models
fitted_alternative_models = {}
# Looping through model pipelines, tuning each with GridSearchCV and saving it to fitted_logreg_models
for name, pipeline in pipelines.items():
    #Creating cross-validation object from pipeline and hyperparameters
    alt_model = GridSearchCV(pipeline, hyperparameters[name], cv=10, n_jobs=-1)

    #Fitting the model on X_train, y_train
    alt_model.fit(X_train, y_train)

    #Storing the model in fitted_logreg_models[name]
```

```
    #Fitting the model on X_train, y_train
    alt_model.fit(X_train, y_train)

    #Storing the model in fitted_logreg_models[name]
    fitted_alternative_models[name] = alt_model

    #Printing the status of the fitting
    print(name, 'fitted.')
#Displaying the best_score_ for each fitted model
for name, model in fitted_alternative_models.items():
    print(name, model.best_score_ )
```

```
rf fitted.
```

```
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/preprocessing/data.py:645: DataConversionWarning: Data
with input dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/base.py:467: DataConversionWarning: Data with input dt
ype uint8, int64, float64 were all converted to float64 by StandardScaler.
  return self.fit(X, y, **fit_params).transform(X)
```

```
gb fitted.
rf 0.8933545036551884
gb 0.8942807052363468
```

In [25]:
```
#Creating the confusion matrix for Random Forest
pd.crosstab(y_test, fitted_alternative_models['rf'].predict(X_test), rownames=['True'], colnames=['Predicted'], mar
```

```
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/pipeline.py:331: DataConversionWarning: Data with inpu
t dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```

In [25]:
```
#Creating the confusion matrix for Random Forest
pd.crosstab(y_test, fitted_alternative_models['rf'].predict(X_test), rownames=['True'], colnames=['Predicted'], mar
```
```
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/pipeline.py:331: DataConversionWarning: Data with inpu
t dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```
Out[25]:

Predicted	0	1	All
True			
0	11293	174	11467
1	1186	304	1490
All	12479	478	12957

In [26]:
```
#Creating the classification report for Gradient Boosting
print(classification_report(y_test, fitted_alternative_models['gb'].predict(X_test)))
```
```
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     11467
           1       0.66      0.20      0.31      1490

   micro avg       0.90      0.90      0.90     12957
   macro avg       0.78      0.59      0.63     12957
weighted avg       0.88      0.90      0.87     12957
```
```
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/pipeline.py:331: DataConversionWarning: Data with inpu
```

In [25]:
```
#Creating the confusion matrix for Gradient Boosting
pd.crosstab(y_test, fitted_alternative_models['gb'].predict(X_test), rownames=['True'], colnames=['Predicted'], mar
```
```
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/pipeline.py:331: DataConversionWarning: Data with inpu
t dtype uint8, int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```
Out[25]:

Predicted	0	1	All
True			
0	11314	153	11467
1	1188	302	1490
All	12502	455	12957

In [26]:
```
#Creating the classification report for Gradient Boosting
print(classification_report(y_test, fitted_alternative_models['gb'].predict(X_test)))
```
```
              precision    recall  f1-score   support

           0       0.90      0.99      0.94     11467
           1       0.66      0.20      0.31      1490

   micro avg       0.90      0.90      0.90     12957
   macro avg       0.78      0.59      0.63     12957
weighted avg       0.88      0.90      0.87     12957
```
```
/home/subarna/anaconda3/lib/python3.7/site-packages/sklearn/pipeline.py:331: DataConversionWarning: Data with inpu
```

Accuracy:

train data:
log1 0.7613707783401145
log2 0.7613707783401145
log_reg 0.7613707783401145
random forest: 0.8962654229102577

gradient boosting classifier: 0.8980516688167774

```
test data:
log1: 0.7636798641660878
log2: 0.7636798641660878
log_reg: 0.7636798641660878
```

random forest: 90%

Random Forest is best model