CONTENTS

1. Abstract

Given some information of a marketing campaign, the goal of this project is to *predict whether or not they end up subscribing for a term deposit*.

2. Data Inspection

df.shape

```
df.shape
```

```
(41188, 21)
```

Interpretation: There are 41188 rows and 21 features.

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 and banking industries often have to deal with datasets that are massively imbalanced. Considering the reality surrounding these problems, addressing the class balance anomaly is not a major priority, for now.

3 - EDA

df.describe()

Using describe() on the dataframe, for 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

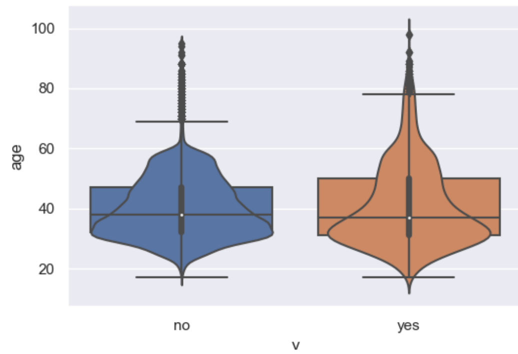
The standard deviations of 'duration', 'pdays' and 'employees' are very large compared to other variables. These variables should be investigated to understand the reasons for this variability.

As part of Graphical EDA, I plot two graphs

- Histograms
- Violin plot

3.1 EDA with Numeric Variables:

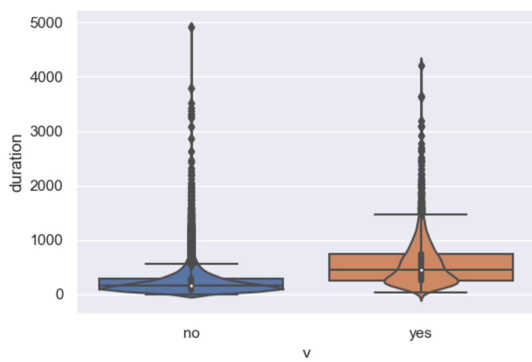
'age':



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 accepted. 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':

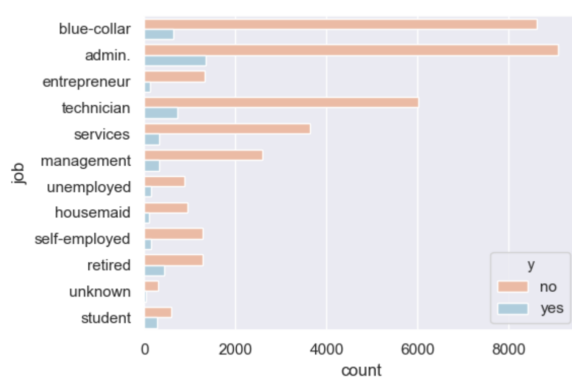


Interpretation: The variance of class 'no' of the response variable is less compared to that of 'yes' class. Outliers for 'no' are widespread than the outliers of 'yes'.

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

3.2 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	admin.	blue-collar	entrepreneur	housemaid	management	retired	self-employed	services	student	technician	unemployed	unknown
y												
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.

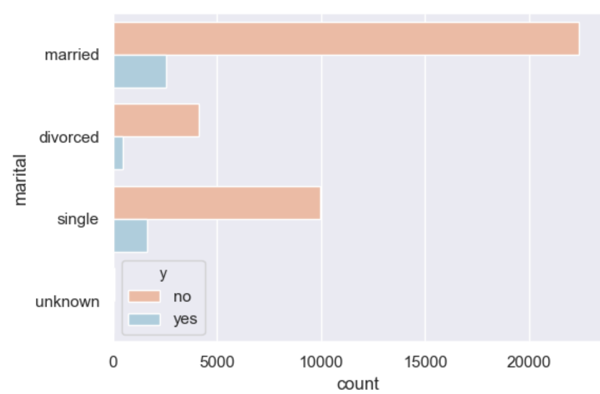
```
pd.crosstab(df.job, df.y, normalize='index').transpose()
```

job	admin.	blue-collar	entrepreneur	housemaid	management	retired	self-employed	services	student	technician	unemployed	unknown
y												
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')
```

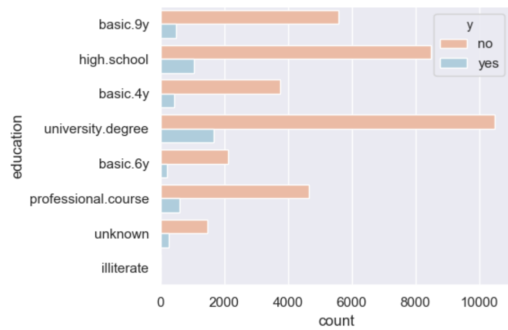
y	no	yes
marital		
divorced	0.893209	0.106791
married	0.897718	0.102282
single	0.859706	0.140294
unknown	0.838710	0.161290

```
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.

4. Feature Engineering

4.1 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'],
                  ['j114', 'j114', 'j114', 'j113', 'j113', 'j113', 'j113', 'j112', 'j112', 'j112', 'j1
11', 'j111'],
                  inplace=True)

z['education'].replace(['basic.9y','basic.6y','basic.4y','high.school','professional.course','universit
y.degree','unknown','illiterate'],
                      ['e114','e114','e113','e113','e113','e112','e112','e111'],
                      inplace=True)

z['month'].replace(['may','jul','nov','aug','jun','apr','oct','sep','dec','mar'],
                      ['m113','m113','m113','m113','m113','m112','m111','m111','m111','m111'],
                      inplace=True)
```

4.2 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'

4.3. 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)
```

4.4 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
```

4.5 Inclusion and Exclusion of 'duration' column:

‘duration’ highly affects the output, I create two dataframes (one with 'duration' column, one without).

4.6 Treating Outliers:

Idea:

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

4.6.1. 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 )
```

4.6.2. 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)))
```

4.6.3. 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

```
# 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])
```

5. Ready for Machine Learning

5.1 Standardization and Normalization

Two popular data scaling methods are normalization and standardization.

1. Data Normalization
2. Data Standardization

```
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)
```

5.2 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
```

5.3 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]
```

6. Machine Learning

6.1 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 – 0.912, Precision – 0.44, Recall – 0.67, F1 – 0.53, AUC - 518

6.2 Randomized Search CV:

Efficient approach is to narrow our search to evaluate a wide range of values for each hyperparameter.

```python
# 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.

```python
# 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())
```

Bootstrap – False, max_depth – 110, max_features – 0.2, min_samples_leaf – 5, min_samples_split – 5, n_estimators - 800

6.3 Grid Search CV:

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

```python
# 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.

```python
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_
```

Bootstrap – False, 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

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.

	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.