Candidate Test - Modelling

With given dataset, predict the probability of the payment within 90 days!

This is a small sample of data (both in terms of observations and available variables) that is actually used for training our models in Intrum. All the IDs are replaced, and all datapoints have a slight noise added to them to avoid any chance of identification. Data is stored as a small SQLite database stored in attached `dataset.db` file. There are two tables - `dataset` and `metadata`. The data sample spans 1 year and 30k observations.

In [14]:
```
import sqlite3
import pandas as pd
con = sqlite3.connect('dataset.db')

df = pd.read_sql_query('SELECT * FROM metadata;', con)
DataFrame = pd.read_sql_query('SELECT * FROM dataset;', con)
```

Some background information on the data

The intended target for prediction is `ct090`, `case_id` is the unique identifier, `keydate` is the point in time when some event has happened in the lifecycle of a case, and also a date relative to which all of the backward looking variables and forward looking targets are calculated. In this case, it's a general purpose propensity to pay model, which means that it is a freshly registered case, where all the relevant data has been gathered and verified. In other words, `keydate` is set a few days after registration, and target `ct090` is checking for outcome in 90 days (`ap090` is a similar regression target), while all the rest of the data is only looking backwards!

Metadata gives some basic description of variables. The general naming convention is based on prefixes that define aggregation levels - `cXXXX` looking at the data of this case only, `dXXXX` looking at other cases of same debtor, `bXXXX` looking at all cases of the debtor, `aXXXX` looking at all the cases on the same address. This is not very relevant for this particular task, but gives some idea of our data setup here in Intrum! Note that this data selection has quite a few variables with the `dXXXX` prefix, which means that this selection is specifically looking at debtors that we already had worked with before, therefore, variable selection is much broader and models are generally better.

One more tip on interpretation of missing values: if variable is bound by time window, e.g. `d2112 NumberOfCustomerPaymentsInLast12Months`, the `NA` value implies that there never have been any values, while `0` would mean that have been no values within bounding period (in this case 12 months). In other words, `0` and `NA` have different interpretation. It may or may not be relevant, depending on the choice of the modelling approach.

Some tips on the task

There is no end to seeking the perfection, and countless ways to approach this task. However, try not to approach this like a competition to crank out the highest possible accuracy metrics. What matters is the sequence and thought process - show this in illustrations and comments!

Even if certain things might take too long to implement in a code, but you have a good idea where to go with this - write down your ideas! This is what matters. Generally, try not to spend more than a few hours on this.

Good luck!

—

My main objective will be to two clasiffication models, a Logistic Regression and a KNN_Classifier. Then, I will compare them.

Steps:

- Data Preparation.
- Analysis of the relation between the input variables and the target variable.
- Models construction and comparison.

Data Preparation:

Firstly, we can start changing the names of the DataFrame columns so that we can read them more easily:

In [15]:
```
colnames = []
for i in range(0, len(df)):
    tupla = []
    tupla.append(df['varcode'][i])
    tupla.append(df['name'][i])
    tupla = tuple(tupla)
    colnames.append(tupla)
```

In [16]:
```
colnames = dict(colnames)
#colnames

DataFrame.rename(columns = colnames, inplace=True)
```

In [23]:
```
DataFrame.head()
```

Out[23]:

	case_id	keydate	Target90Days	TargetAmount90Days	OriginalCapitalOfCaseInvoices	ClientNam
0	1	2017-08-12 00:00:00.0	0.0	0.0	221.68	
1	2	2017-02-03 00:00:00.0	0.0	0.0	151.36	
2	3	2017-02-17 00:00:00.0	0.0	0.0	48.84	

	case_id	keydate	Target90Days	TargetAmount90Days	OriginalCapitalOfCaseInvoices	ClientNam
3	4	2017-09-18 00:00:00.0	0.0	0.0	413.15	
4	5	2017-07-22 00:00:00.0	0.0	0.0	125.83	

In [24]:
```
DataFrame.describe()
```

Out[24]:

	case_id	Target90Days	TargetAmount90Days	OriginalCapitalOfCaseInvoices	NumberOfU
count	30000.000000	30000.000000	30000.000000	29975.000000	
mean	15000.500000	0.168733	49.215836	538.590694	
std	8660.398374	0.374522	240.063401	1248.533877	
min	1.000000	0.000000	0.000000	0.000000	
25%	7500.750000	0.000000	0.000000	145.100000	
50%	15000.500000	0.000000	0.000000	298.720000	
75%	22500.250000	0.000000	0.000000	638.645000	
max	30000.000000	1.000000	25000.000000	84561.840000	

With the describe function we see some information of the distribution of the variables (there are some negative values, for instance)

'IndustryCode' is a cualitative variable, we can make it cuantitative:

In [18]:
```
set(DataFrame.IndustryCode.to_list())

IndustryCodeNum = []

for a in DataFrame['IndustryCode'].to_list():
    if a == 'K6419':
        IndustryCodeNum.append(0)
    elif a == 'K6420':
        IndustryCodeNum.append(1)
    elif a == 'K6491':
        IndustryCodeNum.append(2)
    elif a == 'K6499':
        IndustryCodeNum.append(4)
    elif a == 'K6511':
        IndustryCodeNum.append(5)
    elif a == 'K6512':
        IndustryCodeNum.append(6)
    elif a == 'K6619':
        IndustryCodeNum.append(7)
    elif a == 'K6622':
        IndustryCodeNum.append(8)

DataFrame['IndustryCodeNum'] = IndustryCodeNum
DataFrame = DataFrame.drop('IndustryCode', axis = 1)
#DataFrame
```

Some variables like, 'case_id', are not going to give information to our model. We can build another Dataset from 'DataFrame' including only the variables we think that can work.

We name it 'DF':

In [21]:
```
DF = DataFrame.drop(['case_id', 'keydate', 'TargetAmount90Days', 'ClientName', 'Last
#DF
```

Out[21]:

	Target90Days	OriginalCapitalOfCaseInvoices	NumberOfUnsuccessfullyClosedCustomerCasesLast
0	0.0	221.68	
1	0.0	151.36	
2	0.0	48.84	
3	0.0	413.15	
4	0.0	125.83	
...	...	...	
29995	0.0	435.46	
29996	1.0	344.07	
29997	0.0	417.23	
29998	0.0	529.00	
29999	0.0	174.23	

30000 rows × 14 columns

After testing another aproaches like replacing the NaN values with the mean or median of each column/variable, I have found that this option gives the most correlation between the variables and the target:

As 0 has a different meaning than NAN, we can split the columns where there are some NAN into two:

- NoNANOfVariableWithNans: each register gets '1' if there is Not a NAN in that column/variable; 0 if it´s a NAN.
- VariableWithoutNans: each register gets its value if the variable is not a NAN (noNAN = 1) or 0 if there is a NAN

This way we create two categories for each column (if the column has any NaN), adding more information to the model, so if:

- noNAN = 1 --> Target = B1 + B2 *noNANoftheVariableWithNans(=1)* + *B3* VariableWithoutNans + ... --> Target = (B1 + B2) + B3 * VariableWithoutNans + ...
- noNAN= 0 --> Target = B1 + B2 *noNANoftheVariable(=0)* + *B3* VariableWithoutNans(=0) --> Target = B1 + ...

In order to not loosing the information as we have it right now, we can create another DataFrame with the split columns and name it DFSplit:

In [30]:
```
DFSplit = pd.DataFrame()

for columna in DF.columns.to_list():

    if len(DF[columna][DF[columna].isna() == True]) == 0: #If there are no Nan in th
        DFSplit[columna] = DF[columna]

    elif len(DF[columna][DF[columna].isna() == 0]) > 0: #If not, we create two new c
        nombrenuevacolumnanoNAN = 'noNAN' + columna
        nombrenuevacolumnaWithoutNAN = columna + 'WithoutNAN'
        noNAN = []
        variableWithoutNan = []
        for registro in DF[columna]:
            if (registro >= 0) or (registro < 0): #If 'registro' is not a Nan
                noNAN.append(1)
                variableWithoutNan.append(registro)
            else: #If 'registro' is NaN
                noNAN.append(0)
                variableWithoutNan.append(0)
        DFSplit[nombrenuevacolumnanoNAN] = noNAN
        DFSplit[nombrenuevacolumnaWithoutNAN] = variableWithoutNan

DFSplit
#DFSplit.describe() (the negative values continue where they were)
```

Out[30]:

	Target90Days	noNANOriginalCapitalOfCaseInvoices	OriginalCapitalOfCaseInvoicesWithoutNAN
0	0.0	1	221.68
1	0.0	1	151.36
2	0.0	1	48.84
3	0.0	1	413.15
4	0.0	1	125.83
...	...	...	...
29995	0.0	1	435.46
29996	1.0	1	344.07
29997	0.0	1	417.23
29998	0.0	1	529.00
29999	0.0	1	174.23

30000 rows × 21 columns

It´s important to see that our target variable seems unbalanced, we check it:

In [19]:
```
#% of repaid Loans:
(len(DataFrame[DataFrame['Target90Days'] == 0]) / len(DataFrame)) * 100
```

Out[19]:
```
83.12666666666667
```

83% of the Target Variable are '0', this is going to translate into our future model being prone to predict almost everything as '0'. To fix that, we can opt for balancing the sample so that our data becomes 50% of '1' and 50% of '0'. We keep all of our data registers with Target = 1, but we randomly choose around 5000 (the same number of registers with '1') registers with Target = 0.

This also easies our Model Validation. With an unbalanced sample, a high accuracy wouldn´t be necessarily a good indicator of how good the model behaves, as it could be predecting everything as '0', failing with all the '1's, and still be getting this high accuracy because of the high number of '0's compared to the '1's. In that case we would better check the sensibility/specificity as a better indicator of the real accuracy of our model.

With a balanced sample though, we can attend to Accuracy to analyse our model behaviour.

(This can work because 5000 is a big enough sample, and we should check that the variables of the balanced sample follows the same distribution of the original data).

We create a new DataFrame for this and we name it DFBalanced:

In [33]:
```
DFF = DFSplit[DFSplit['Target90Days'] == 0].sample(5062)
DFG = DFSplit[DFSplit['Target90Days'] == 1]

DFBalanced = pd.concat([DFF, DFG])

DFBalanced
```

Out[33]:

	Target90Days	noNANOriginalCapitalOfCaseInvoices	OriginalCapitalOfCaseInvoicesWithoutNAN
12266	0.0	1	775.51
29165	0.0	1	248.70
21606	0.0	1	520.80
3965	0.0	1	1126.32
19007	0.0	1	354.39
...	...	...	...
29980	1.0	1	237.74
29984	1.0	1	10500.00
29985	1.0	1	851.52
29991	1.0	1	250.07
29996	1.0	1	344.07

10124 rows × 21 columns

Descriptive Analysis of the Relation between the input variables and the target variable:

We can start adding a couple of control variables (random variables) to see their relation with the target (wich is none as they are random). Every variable with a relation with the target

variable weaker than the control variables relation with the target variable won´t be considered in our model.

In [34]:
```
import random

variable_aleatoria1 = []
for i in range(len(DFBalanced)):
    variable_aleatoria1.append(random.randint(0,100))

variable_aleatoria2 = []
for i in range(len(DFBalanced)):
    variable_aleatoria2.append(random.randint(0,100))
```

In [35]:
```
#we append them to our DFBalanced:

DFBalanced['VarControl1'] = variable_aleatoria1
DFBalanced['VarControl2'] = variable_aleatoria2
```

SPEARMAN COEFFICIENT:

We are going to attend to the Spearman Coefficient, wich is a measure of asotiation between two variables, regardless if they are continuous are discrete. It goes between -1 and 1, but we are going to compare its absolute value, as we do not care about the direction of the relation right now.

In [37]:
```
Spearman = DFBalanced.corr('spearman') #DataFrame of Correlations
Spearman['Target90Days'] = Spearman['Target90Days'].abs() #We set the column 'Target
Spearman = Spearman.sort_values('Target90Days', ascending=False) #We sort it
Spearman
```

Out[37]:

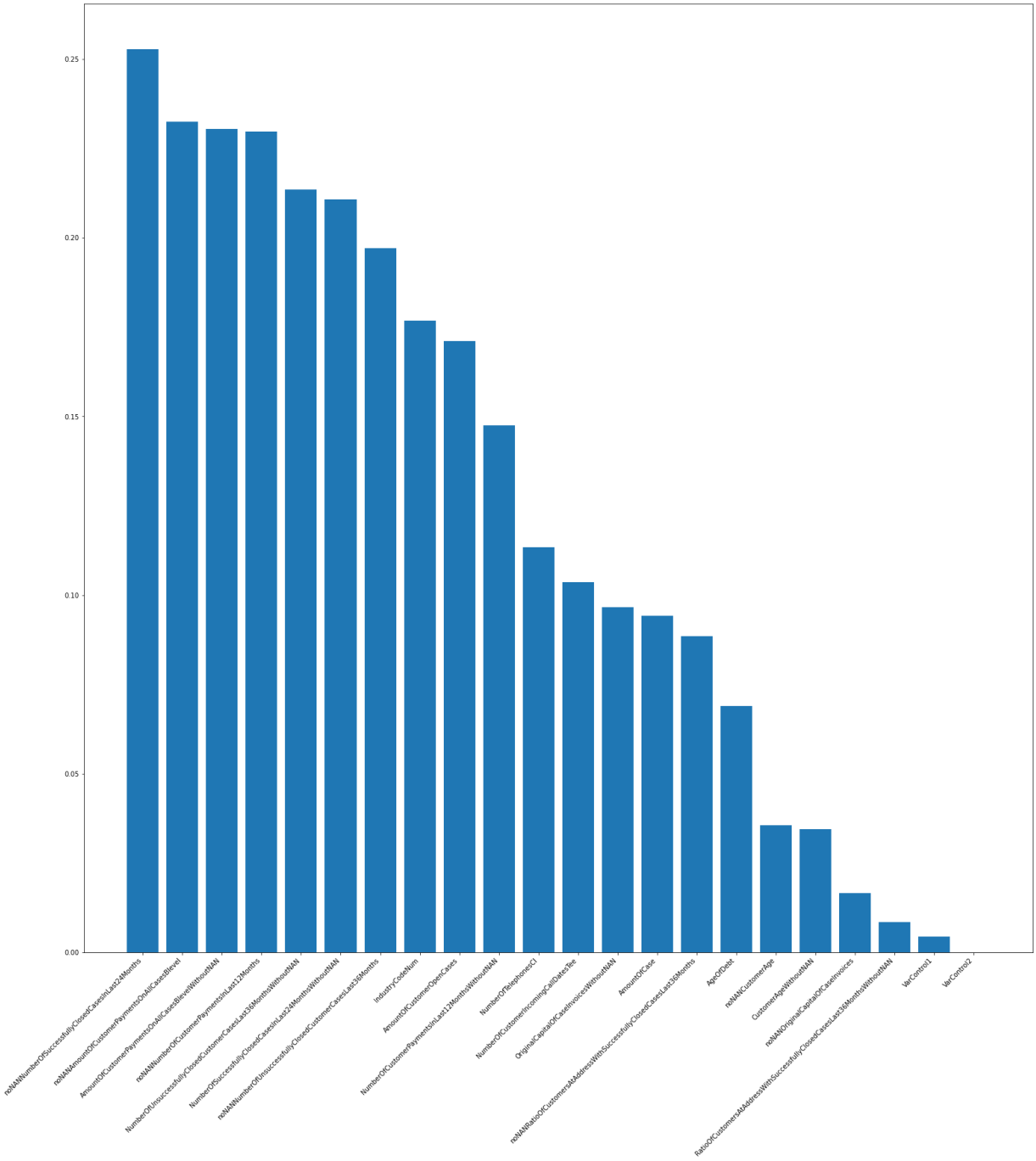
	Target90Days	noN
Target90Days	1.000000	
noNANNumberOfSuccessfullyClosedCasesInLast24Months	0.252847	
noNANAmountOfCustomerPaymentsOnAllCasesBlevel	0.232548	
AmountOfCustomerPaymentsOnAllCasesBlevelWithoutNAN	0.230552	
noNANNumberOfCustomerPaymentsInLast12Months	0.229769	
NumberOfUnsuccessfullyClosedCustomerCasesLast36MonthsWithoutNAN	0.213583	
NumberOfSuccessfullyClosedCasesInLast24MonthsWithoutNAN	0.210697	
noNANNumberOfUnsuccessfullyClosedCustomerCasesLast36Months	0.197062	
IndustryCodeNum	0.176849	
AmountOfCustomerOpenCases	0.171081	
NumberOfCustomerPaymentsInLast12MonthsWithoutNAN	0.147587	
NumberOfTelephonesCl	0.113425	
NumberOfCustomerIncomingCallDatesTee	0.103688	
OriginalCapitalOfCaseInvoicesWithoutNAN	0.096639	
AmountOfCase	0.094305	

	Target90Days	noN
noNANRatioOfCustomersAtAddressWithSuccessfullyClosedCasesLast36Months	0.088449	
AgeOfDebt	0.068972	
noNANCustomerAge	0.035626	
CustomerAgeWithoutNAN	0.034433	
noNANOriginalCapitalOfCaseInvoices	0.016572	
RatioOfCustomersAtAddressWithSuccessfullyClosedCasesLast36MonthsWithoutNAN	0.008528	
VarControl1	0.004412	
VarControl2	0.000003	

23 rows × 23 columns

In [41]:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(20, 20))
ax = fig.add_axes([0, 0, 1, 1])
ax.bar(Spearman.index.to_list()[1:], Spearman['Target90Days'].to_list()[1:])
#plt.xticks(rotation=45)
fig.autofmt_xdate(rotation=45)
plt.show()
```



There are not any big relation with the Target variable, but all the relations are bigger than the Control Variables Relation, so we can consider all of them as possible inputs for our model.

In [43]:
```
Inputs = DFBalanced.drop(['VarControl1', 'VarControl2', 'Target90Days'], axis = 1)
Inputs.head()
```

Out[43]:

	noNANOriginalCapitalOfCaseInvoices	OriginalCapitalOfCaseInvoicesWithoutNAN	noNANNumb
12266	1	775.51	
29165	1	248.70	
21606	1	520.80	
3965	1	1126.32	
19007	1	354.39	

We included many new variables in our model, when we split some columns into two, without

much checking of relation between them.

This could cause multicollinearity problems in our model. To control this, we attend to the VIFs (wich depending on the author indicates worrying Multicollinearity for values bigger than 5, 10...):

In [45]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor


X = Inputs

vif_data = pd.DataFrame() # VIF dataframe

variables = Inputs.columns.to_list()

vif_data["feature"] = variables

vif_data["VIF"] = [variance_inflation_factor(X.values, i) # calculating VIF for each
                           for i in range(len(X.columns))]

print(vif_data)
```

	feature	VIF
0	noNANOriginalCapitalOfCaseInvoices	33.169279
1	OriginalCapitalOfCaseInvoicesWithoutNAN	6.696349
2	noNANNumberOfUnsuccessfullyClosedCustomerCases...	4.053757
3	NumberOfUnsuccessfullyClosedCustomerCasesLast3...	3.035013
4	noNANAmountOfCustomerPaymentsOnAllCasesBlevel	259.052650
5	AmountOfCustomerPaymentsOnAllCasesBlevelWithou...	1.544099
6	AmountOfCustomerOpenCases	1.141876
7	NumberOfTelephonesCI	4.060239
8	noNANRatioOfCustomersAtAddressWithSuccessfully...	3.692219
9	RatioOfCustomersAtAddressWithSuccessfullyClose...	1.919484
10	AgeOfDebt	1.368664
11	NumberOfCustomerIncomingCallDatesTee	1.371871
12	noNANCustomerAge	19.334626
13	CustomerAgeWithoutNAN	11.194280
14	noNANNumberOfCustomerPaymentsInLast12Months	260.916618
15	NumberOfCustomerPaymentsInLast12MonthsWithoutNAN	1.572372
16	noNANNumberOfSuccessfullyClosedCasesInLast24Mo...	5.222440
17	NumberOfSuccessfullyClosedCasesInLast24MonthsW...	2.430387
18	AmountOfCase	7.151556
19	IndustryCodeNum	19.958665

We remove 'noNANNumberOfCustomerPaymentsInLast12Months' from our inputs:

In [46]:
```
Inputs = Inputs.drop('noNANNumberOfCustomerPaymentsInLast12Months', axis = 1)

X = Inputs

vif_data = pd.DataFrame() # VIF dataframe

variables = Inputs.columns.to_list()

vif_data["feature"] = variables

vif_data["VIF"] = [variance_inflation_factor(X.values, i) # calculating VIF for each
                           for i in range(len(X.columns))]

print(vif_data)
```

	feature	VIF
0	noNANOriginalCapitalOfCaseInvoices	33.153933
1	OriginalCapitalOfCaseInvoicesWithoutNAN	6.694903
2	noNANNumberOfUnsuccessfullyClosedCustomerCases...	4.053755
3	NumberOfUnsuccessfullyClosedCustomerCasesLast3...	3.034688
4	noNANAmountOfCustomerPaymentsOnAllCasesBlevel	5.281591
5	AmountOfCustomerPaymentsOnAllCasesBlevelWithou...	1.543883
6	AmountOfCustomerOpenCases	1.141870
7	NumberOfTelephonesCI	4.058344
8	noNANRatioOfCustomersAtAddressWithSuccessfully...	3.692218
9	RatioOfCustomersAtAddressWithSuccessfullyClose...	1.919452
10	AgeOfDebt	1.368485
11	NumberOfCustomerIncomingCallDatesTee	1.371426
12	noNANCustomerAge	19.333942
13	CustomerAgeWithoutNAN	11.193843
14	NumberOfCustomerPaymentsInLast12MonthsWithoutNAN	1.570275
15	noNANNumberOfSuccessfullyClosedCasesInLast24Mo...	5.179137
16	NumberOfSuccessfullyClosedCasesInLast24MonthsW...	2.430215
17	AmountOfCase	7.150538
18	IndustryCodeNum	19.940671

We remove 'noNANOriginalCapitalOfCaseInvoices' from our inputs:

In [47]:
```
Inputs = Inputs.drop('noNANOriginalCapitalOfCaseInvoices', axis = 1)

X = Inputs

vif_data = pd.DataFrame() # VIF dataframe

variables = Inputs.columns.to_list()

vif_data["feature"] = variables

vif_data["VIF"] = [variance_inflation_factor(X.values, i) # calculating VIF for each
                   for i in range(len(X.columns))]

print(vif_data)
```

	feature	VIF
0	OriginalCapitalOfCaseInvoicesWithoutNAN	6.694637
1	noNANNumberOfUnsuccessfullyClosedCustomerCases...	3.947900
2	NumberOfUnsuccessfullyClosedCustomerCasesLast3...	2.998869
3	noNANAmountOfCustomerPaymentsOnAllCasesBlevel	5.231749
4	AmountOfCustomerPaymentsOnAllCasesBlevelWithou...	1.541120
5	AmountOfCustomerOpenCases	1.141868
6	NumberOfTelephonesCI	4.003672
7	noNANRatioOfCustomersAtAddressWithSuccessfully...	3.607761
8	RatioOfCustomersAtAddressWithSuccessfullyClose...	1.916694
9	AgeOfDebt	1.323250
10	NumberOfCustomerIncomingCallDatesTee	1.368069
11	noNANCustomerAge	17.012218
12	CustomerAgeWithoutNAN	11.178004
13	NumberOfCustomerPaymentsInLast12MonthsWithoutNAN	1.569965
14	noNANNumberOfSuccessfullyClosedCasesInLast24Mo...	5.162607
15	NumberOfSuccessfullyClosedCasesInLast24MonthsW...	2.428761
16	AmountOfCase	7.126300
17	IndustryCodeNum	8.539145

We remove 'noNANCustomerAge' from our inputs:

In [50]:
```
Inputs = Inputs.drop('noNANCustomerAge', axis = 1)

X = Inputs
```

```
vif_data = pd.DataFrame() # VIF dataframe

variables = Inputs.columns.to_list()

vif_data["feature"] = variables

vif_data["VIF"] = [variance_inflation_factor(X.values, i) # calculating VIF for each
                           for i in range(len(X.columns))]

print(vif_data)
```

```
                                              feature       VIF
0             OriginalCapitalOfCaseInvoicesWithoutNAN  6.688345
1   noNANNumberOfUnsuccessfullyClosedCustomerCases...  3.931707
2   NumberOfUnsuccessfullyClosedCustomerCasesLast3...  2.998731
3       noNANAmountOfCustomerPaymentsOnAllCasesBlevel  5.227687
4   AmountOfCustomerPaymentsOnAllCasesBlevelWithou...  1.539599
5                           AmountOfCustomerOpenCases  1.141842
6                                NumberOfTelephonesCI  3.976736
7   noNANRatioOfCustomersAtAddressWithSuccessfully...  3.559276
8   RatioOfCustomersAtAddressWithSuccessfullyClose...  1.916228
9                                             AgeOfDebt  1.319938
10              NumberOfCustomerIncomingCallDatesTee  1.368039
11                             CustomerAgeWithoutNAN  4.772236
12   NumberOfCustomerPaymentsInLast12MonthsWithoutNAN  1.569166
13  noNANNumberOfSuccessfullyClosedCasesInLast24Mo...  5.162313
14  NumberOfSuccessfullyClosedCasesInLast24MonthsW...  2.428269
15                                       AmountOfCase  7.103784
16                                    IndustryCodeNum  7.376284
```

LOGIT:

If we are happy with the VIF values we can run a Logistic Regression

If we want to display statistical measures, we can run the regression with Statsmodels:

In [78]:

```
from sklearn.model_selection import train_test_split
import statsmodels.api as sm

X = Inputs.values
y = DFBalanced['Target90Days'].values

#We define the training part as 80% although that percentage could be optimised for
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)

X_trainWithConstant = sm.add_constant(X_train, prepend=True)
X_testWithConstant = sm.add_constant(X_test, prepend=True)

# building the model and fitting the data
model1 = sm.Logit(y_train, X_trainWithConstant).fit()

model1.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.623808
         Iterations 6
```

Out[78]:

Logit Regression Results

Dep. Variable:	y	**No. Observations:**	8099
Model:	Logit	**Df Residuals:**	8081
Method:	MLE	**Df Model:**	17

Date:	Tue, 15 Feb 2022	**Pseudo R-squ.:**	0.1000
Time:	17:12:31	**Log-Likelihood:**	-5052.2
converged:	True	**LL-Null:**	-5613.8
Covariance Type:	nonrobust	**LLR p-value:**	3.980e-228

	coef	std err	z	P>|z|	[0.025	0.975]
const	1.4753	0.137	10.780	0.000	1.207	1.743
x1	-0.0003	0.000	-2.993	0.003	-0.001	-0.000
x2	-0.0963	0.069	-1.396	0.163	-0.231	0.039
x3	-0.2742	0.035	-7.937	0.000	-0.342	-0.206
x4	0.5062	0.076	6.687	0.000	0.358	0.655
x5	-8.438e-06	4.88e-05	-0.173	0.863	-0.000	8.73e-05
x6	-7.841e-05	1.98e-05	-3.970	0.000	-0.000	-3.97e-05
x7	-0.0673	0.014	-4.794	0.000	-0.095	-0.040
x8	-0.4068	0.057	-7.103	0.000	-0.519	-0.295
x9	0.3972	0.092	4.302	0.000	0.216	0.578
x10	-6.602e-06	0.000	-0.051	0.959	-0.000	0.000
x11	0.0780	0.019	4.192	0.000	0.042	0.114
x12	0.0025	0.001	1.953	0.051	-9.46e-06	0.005
x13	0.0349	0.021	1.693	0.090	-0.006	0.075
x14	0.3094	0.090	3.444	0.001	0.133	0.486
x15	0.1156	0.053	2.165	0.030	0.011	0.220
x16	0.0002	0.000	1.399	0.162	-6.3e-05	0.000
x17	-0.2019	0.017	-11.688	0.000	-0.236	-0.168

We will use sklearn library though:

In [97]:
```
from sklearn.linear_model import LogisticRegression

import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

In [100…
```
model1 = LogisticRegression()

X = Inputs.values
y = DFBalanced['Target90Days'].values

#We define the training part as 80% although that percentage could be optimised for
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)

model1.fit(X = X_train, y = y_train)
```

```
print("Intercept:", model1.intercept_)
print("\nCoefficient:", list(model1.coef_))
print("\nAccuracy in the test sample:", model1.score(X_test, y_test))
```

```
Intercept: [0.03928543]

Coefficient: [array([-4.25914801e-04, -1.02880128e-01, -2.53544310e-01,  1.55147494e
-01,
        1.34507579e-04, -7.37968834e-05, -1.27702198e-01, -3.40564143e-02,
        1.43647642e-02,  2.22201294e-04,  1.13559371e-01,  6.15605114e-03,
        1.21774016e-01,  1.51492195e-01,  1.49577625e-01,  3.50600641e-04,
        3.17806756e-03])]

Accuracy in the test sample: 0.6370370370370371
```

REPEATED CROSS VALIDATION:

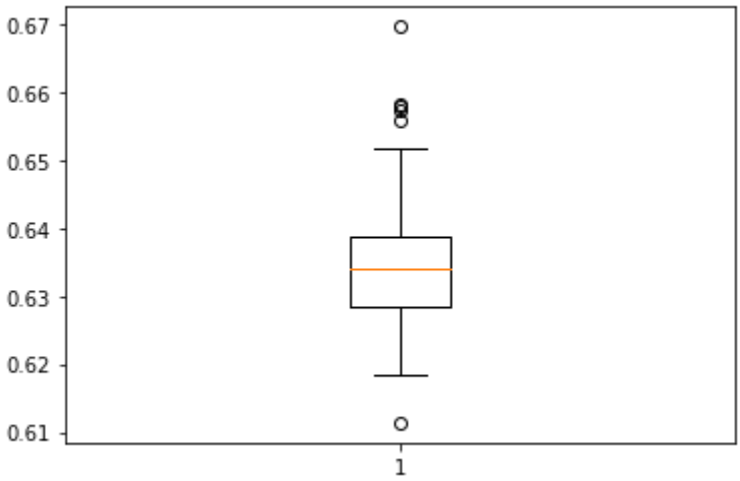
(We should do it with the original data, we´ll do it just with the balanced sample data)

In [101…

```
AccuraciesModel1 = []

for i in range(0, 100):

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)

    model1.fit(X = X_train, y = y_train)

    AccuraciesModel1.append(model1.score(X_test, y_test))
```

In [102…

```
fig, ax = plt.subplots()
ax.boxplot(AccuraciesModel1)
plt.show()
```



KNN_Classiffier:

In [103…

```
from sklearn.neighbors import KNeighborsClassifier
```

In [105…

```
#KNN:
k = 7 #this parameter should be optimised, we set 7 as an example

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

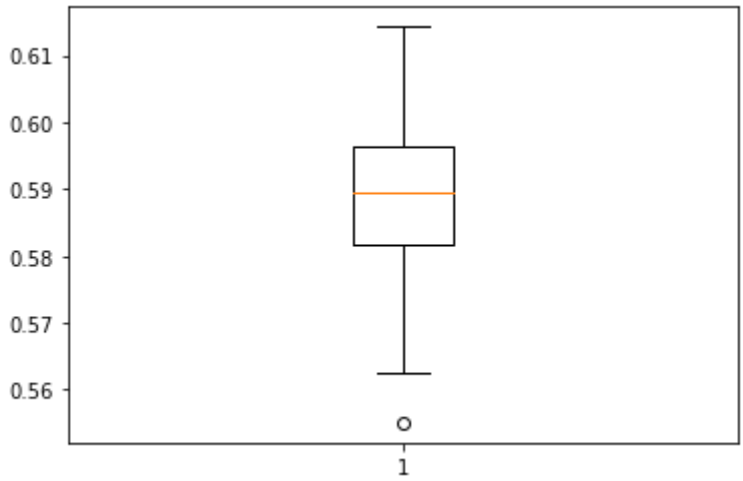
```
model2 = KNeighborsClassifier(k, weights='distance')

model2.fit(X_train, y_train)

model2.score(X_test, y_test)
```

Out[105… 0.582716049382716

REPEATED CROSS VALIDATION:

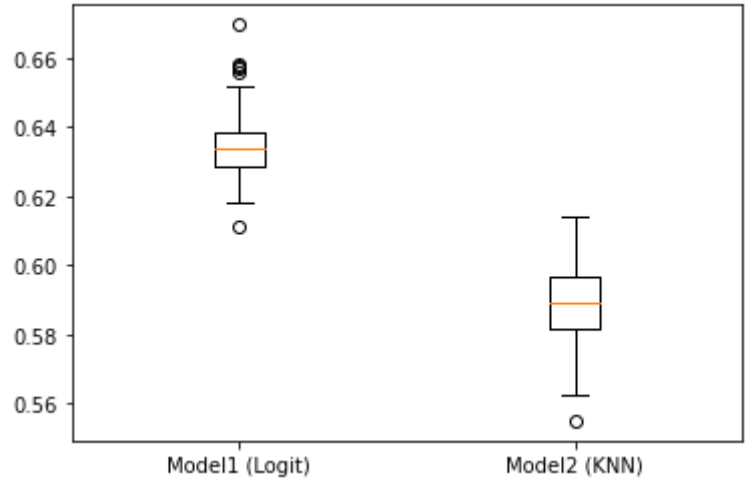
In [106…
```
AccuraciesModel2 = []

for i in range(0, 100):

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)

    model2.fit(X = X_train, y = y_train)

    AccuraciesModel2.append(model2.score(X_test, y_test))
```

In [107…
```
fig, ax = plt.subplots()
ax.boxplot(AccuraciesModel2)
plt.show()
```



Models Comparison:

In [110…
```
fig, ax = plt.subplots()
ax.boxplot([AccuraciesModel1, AccuraciesModel2], labels = ['Model1 (Logit)', 'Model2
plt.show()
```



We would choose Model1, with a median accuracy a bit bigger than 63%, instead of Model2, with a median accuracy of less than 60%

We can predict probabilities now.

For example, the probability of payment of the 47th individual of the current X_test is 65,93587 according to model1:

In [114…
```
model1.predict_proba([X_test[47]])
```

Out[114…
```
array([[0.6593587, 0.3406413]])
```