Final_Project

July 27, 2022

1 Credit Card Fraud Detection

The following project aims to create two different Machine Learning models which should be able to recognize a credit card fraud.
The different parameters taken into account are: * distancefromhome - the distance from home where the transaction happened. * distancefromlast_transaction - the distance from last transaction happened. * ratiotomedianpurchaseprice - Ratio of purchased price transaction to median purchase price. * repeat_retailer - Is the transaction happened from same retailer. * used_chip - Is the transaction through chip (credit card). * usedpinnumber - Is the transaction happened by using PIN number. * online_order - Is the transaction an online order. * fraud - Is the transaction fraudulent. The values reported as 0 and 1 have the following meaning: * 0 –> no * 1 –> yes
For example: used_pin_number == 0 states that the pin number has not been used.

2 Downloading the Dataset

The dataset has been dwonloaded from the following link: https://www.kaggle.com/datasets/dhanushnarayananr/credit-card-fraud in a .csv format.

[145]:
```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('darkgrid')
%matplotlib inline
```

[2]:
```
raw_df = pd.read_csv('card_transdata.csv')
```

[3]:
```
raw_df
```

[3]:
```
        distance_from_home  distance_from_last_transaction  \
0                57.877857                        0.311140
1                10.829943                        0.175592
2                 5.091079                        0.805153
3                 2.247564                        5.600044
4                44.190936                        0.566486
…                      …                             …
999995            2.207101                        0.112651
```

```
999996            19.872726                        2.683904
999997             2.914857                        1.472687
999998             4.258729                        0.242023
999999            58.108125                        0.318110

        ratio_to_median_purchase_price  repeat_retailer  used_chip  \
0                             1.945940              1.0        1.0
1                             1.294219              1.0        0.0
2                             0.427715              1.0        0.0
3                             0.362663              1.0        1.0
4                             2.222767              1.0        1.0
…                                  …              …        …
999995                        1.626798              1.0        1.0
999996                        2.778303              1.0        1.0
999997                        0.218075              1.0        1.0
999998                        0.475822              1.0        0.0
999999                        0.386920              1.0        1.0

        used_pin_number  online_order  fraud
0                   0.0           0.0    0.0
1                   0.0           0.0    0.0
2                   0.0           1.0    0.0
3                   0.0           1.0    0.0
4                   0.0           1.0    0.0
…                 …           …    …
999995              0.0           0.0    0.0
999996              0.0           0.0    0.0
999997              0.0           1.0    0.0
999998              0.0           1.0    0.0
999999              0.0           1.0    0.0

[1000000 rows x 8 columns]
```

- fraud == 0 –> no fraud
- fraud == 1 –> fraud

[4]:
```python
fraud_dict = {1:'yes', 0:'no'}
str_fraud = raw_df.fraud.map(fraud_dict)
```

[5]:
```python
raw_df.drop(columns='fraud', inplace=True)
```

[6]:
```python
raw_df['fraud'] = str_fraud
```

[7]:
```python
raw_df
```

[7]:
```
        distance_from_home  distance_from_last_transaction  \
0                57.877857                        0.311140
1                10.829943                        0.175592
```

```
2                5.091079                        0.805153
3                2.247564                        5.600044
4               44.190936                        0.566486
…                  …                              …
999995           2.207101                        0.112651
999996          19.872726                        2.683904
999997           2.914857                        1.472687
999998           4.258729                        0.242023
999999          58.108125                        0.318110

        ratio_to_median_purchase_price  repeat_retailer  used_chip  \
0                             1.945940              1.0        1.0
1                             1.294219              1.0        0.0
2                             0.427715              1.0        0.0
3                             0.362663              1.0        1.0
4                             2.222767              1.0        1.0
…                                  …               …         …
999995                        1.626798              1.0        1.0
999996                        2.778303              1.0        1.0
999997                        0.218075              1.0        1.0
999998                        0.475822              1.0        0.0
999999                        0.386920              1.0        1.0

        used_pin_number  online_order fraud
0                   0.0           0.0    no
1                   0.0           0.0    no
2                   0.0           1.0    no
3                   0.0           1.0    no
4                   0.0           1.0    no
…                  …            …   …
999995              0.0           0.0    no
999996              0.0           0.0    no
999997              0.0           1.0    no
999998              0.0           1.0    no
999999              0.0           1.0    no

[1000000 rows x 8 columns]
```

[127]:
```
raw_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 8 columns):
 #   Column                          Non-Null Count    Dtype
---  ------                          --------------    -----
 0   distance_from_home              1000000 non-null  float64
 1   distance_from_last_transaction  1000000 non-null  float64
 2   ratio_to_median_purchase_price  1000000 non-null  float64
```

```
 3   repeat_retailer                 1000000 non-null  float64
 4   used_chip                       1000000 non-null  float64
 5   used_pin_number                 1000000 non-null  float64
 6   online_order                    1000000 non-null  float64
 7   fraud                           1000000 non-null  object
dtypes: float64(7), object(1)
memory usage: 61.0+ MB
```

[120]:
```
raw_df.shape
```

[120]:
```
(1000000, 8)
```

[121]:
```
raw_df.describe()
```

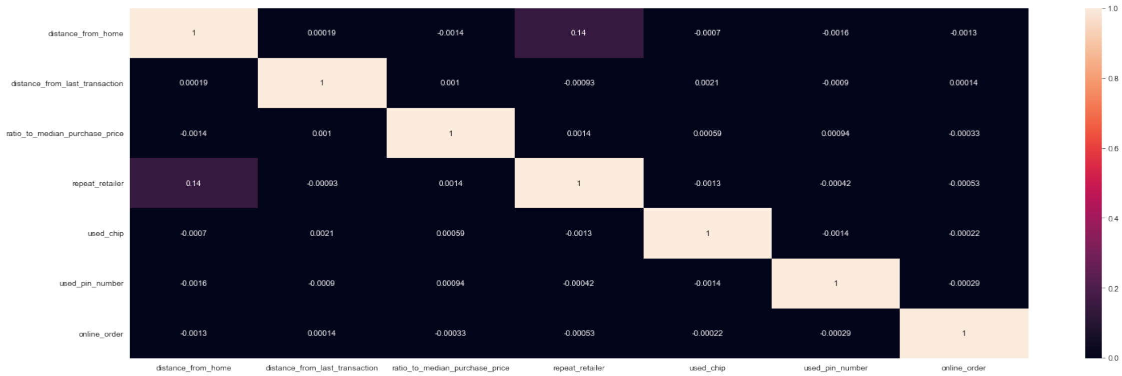
[121]:
```
       distance_from_home  distance_from_last_transaction  \
count      1000000.000000                  1000000.000000
mean            26.628792                        5.036519
std             65.390784                       25.843093
min              0.004874                        0.000118
25%              3.878008                        0.296671
50%              9.967760                        0.998650
75%             25.743985                        3.355748
max          10632.723672                    11851.104565

       ratio_to_median_purchase_price  repeat_retailer       used_chip  \
count                  1000000.000000   1000000.000000  1000000.000000
mean                         1.824182         0.881536        0.350399
std                          2.799589         0.323157        0.477095
min                          0.004399         0.000000        0.000000
25%                          0.475673         1.000000        0.000000
50%                          0.997717         1.000000        0.000000
75%                          2.096370         1.000000        1.000000
max                        267.802942         1.000000        1.000000

       used_pin_number    online_order
count   1000000.000000  1000000.000000
mean          0.100608        0.650552
std           0.300809        0.476796
min           0.000000        0.000000
25%           0.000000        0.000000
50%           0.000000        1.000000
75%           0.000000        1.000000
max           1.000000        1.000000
```

[125]:
```
raw_df.corr()
```

[125]:
```
                                distance_from_home  \
distance_from_home                        1.000000
```

```
distance_from_last_transaction           0.000193
ratio_to_median_purchase_price          -0.001374
repeat_retailer                          0.143124
used_chip                               -0.000697
used_pin_number                         -0.001622
online_order                            -0.001301

                                distance_from_last_transaction  \
distance_from_home                                    0.000193
distance_from_last_transaction                        1.000000
ratio_to_median_purchase_price                        0.001013
repeat_retailer                                      -0.000928
used_chip                                             0.002055
used_pin_number                                      -0.000899
online_order                                          0.000141

                                ratio_to_median_purchase_price  \
distance_from_home                                   -0.001374
distance_from_last_transaction                        0.001013
ratio_to_median_purchase_price                        1.000000
repeat_retailer                                       0.001374
used_chip                                             0.000587
used_pin_number                                       0.000942
online_order                                         -0.000330

                                repeat_retailer  used_chip  used_pin_number  \
distance_from_home                     0.143124  -0.000697        -0.001622
distance_from_last_transaction        -0.000928   0.002055        -0.000899
ratio_to_median_purchase_price         0.001374   0.000587         0.000942
repeat_retailer                        1.000000  -0.001345        -0.000417
used_chip                             -0.001345   1.000000        -0.001393
used_pin_number                       -0.000417  -0.001393         1.000000
online_order                          -0.000532  -0.000219        -0.000291

                                online_order
distance_from_home                 -0.001301
distance_from_last_transaction      0.000141
ratio_to_median_purchase_price     -0.000330
repeat_retailer                    -0.000532
used_chip                          -0.000219
used_pin_number                    -0.000291
online_order                        1.000000
```

[146]:
```
plt.figure(figsize=(25,8))
sns.heatmap(raw_df.corr(), annot=True);
```



3 Defining Inputs and Outputs

[8]:
```
input_cols = list(raw_df.columns)[0:7]
target_col = 'fraud'
```

[9]:
```
input_cols
```

[9]:
```
['distance_from_home',
 'distance_from_last_transaction',
 'ratio_to_median_purchase_price',
 'repeat_retailer',
 'used_chip',
 'used_pin_number',
 'online_order']
```

[10]:
```
target_col
```

[10]:
```
'fraud'
```

Define the Input and Target Data Frame

[11]:
```
input_df = raw_df[input_cols].copy()
```

[12]:
```
input_df
```

[12]:
```
        distance_from_home  distance_from_last_transaction  \
0                57.877857                        0.311140
1                10.829943                        0.175592
2                 5.091079                        0.805153
3                 2.247564                        5.600044
4                44.190936                        0.566486
…                      …                              …
999995            2.207101                        0.112651
999996           19.872726                        2.683904
```

```
999997          2.914857                     1.472687
999998          4.258729                     0.242023
999999         58.108125                     0.318110

        ratio_to_median_purchase_price  repeat_retailer  used_chip  \
0                             1.945940              1.0        1.0
1                             1.294219              1.0        0.0
2                             0.427715              1.0        0.0
3                             0.362663              1.0        1.0
4                             2.222767              1.0        1.0
…                                  …              …        …
999995                        1.626798              1.0        1.0
999996                        2.778303              1.0        1.0
999997                        0.218075              1.0        1.0
999998                        0.475822              1.0        0.0
999999                        0.386920              1.0        1.0

        used_pin_number  online_order
0                   0.0           0.0
1                   0.0           0.0
2                   0.0           1.0
3                   0.0           1.0
4                   0.0           1.0
…                   …            …
999995              0.0           0.0
999996              0.0           0.0
999997              0.0           1.0
999998              0.0           1.0
999999              0.0           1.0

[1000000 rows x 7 columns]
```

[13]:
```
target_df = raw_df[target_col].copy()
```

[14]:
```
target_df
```

[14]:
```
0         no
1         no
2         no
3         no
4         no
          ..
999995    no
999996    no
999997    no
999998    no
999999    no
```

```
Name: fraud, Length: 1000000, dtype: object
```

Scale the Values There are only numerical columns.

[15]:
```python
numerical_cols = list(input_df.columns)
```

[16]:
```python
numerical_cols
```

[16]:
```
['distance_from_home',
 'distance_from_last_transaction',
 'ratio_to_median_purchase_price',
 'repeat_retailer',
 'used_chip',
 'used_pin_number',
 'online_order']
```

Check if there are missing values

[17]:
```python
input_df.isna().sum()
```

[17]:
```
distance_from_home                0
distance_from_last_transaction    0
ratio_to_median_purchase_price    0
repeat_retailer                   0
used_chip                         0
used_pin_number                   0
online_order                      0
dtype: int64
```

Good, there are no missing values, but it is still necessary to put this value in a range between 0 and 1.

[18]:
```python
from sklearn.preprocessing import MinMaxScaler
```

[19]:
```python
scaler = MinMaxScaler().fit(input_df[numerical_cols])
```

[20]:
```python
input_df[numerical_cols] = scaler.transform(input_df[numerical_cols])
```

[21]:
```python
input_df
```

[21]:
```
        distance_from_home  distance_from_last_transaction  \
0                 0.005443                        0.000026
1                 0.001018                        0.000015
2                 0.000478                        0.000068
3                 0.000211                        0.000473
4                 0.004156                        0.000048
…                    …                               …
999995            0.000207                        0.000009
999996            0.001869                        0.000226
```

```
999997          0.000274                        0.000124
999998          0.000400                        0.000020
999999          0.005465                        0.000027

        ratio_to_median_purchase_price  repeat_retailer  used_chip  \
0                             0.007250              1.0        1.0
1                             0.004816              1.0        0.0
2                             0.001581              1.0        0.0
3                             0.001338              1.0        1.0
4                             0.008284              1.0        1.0
…                                  …               …         …
999995                        0.006058              1.0        1.0
999996                        0.010358              1.0        1.0
999997                        0.000798              1.0        1.0
999998                        0.001760              1.0        0.0
999999                        0.001428              1.0        1.0

        used_pin_number  online_order
0                   0.0           0.0
1                   0.0           0.0
2                   0.0           1.0
3                   0.0           1.0
4                   0.0           1.0
…                   …            …
999995              0.0           0.0
999996              0.0           0.0
999997              0.0           1.0
999998              0.0           1.0
999999              0.0           1.0

[1000000 rows x 7 columns]
```

4 Create a Train and a Validation Set

[22]:
```python
from sklearn.model_selection import train_test_split
```

[23]:
```python
train_inputs, val_inputs, train_targets, val_targets =␣
↪train_test_split(input_df[numerical_cols], target_df, test_size=0.25,␣
↪random_state=42)
```

[24]:
```python
train_inputs
```

[24]:
```
        distance_from_home  distance_from_last_transaction  \
570606          0.005705                    1.088382e-03
756283          0.000692                    1.740870e-05
738227          0.000795                    2.476203e-04
554038          0.004589                    5.165978e-04
```

```
712266           0.002926                      2.943333e-06
…                    …                               …
259178           0.000050                      2.935927e-04
365838           0.010703                      1.391652e-05
131932           0.001925                      3.592453e-07
671155           0.000936                      1.416755e-05
121958           0.000066                      2.116144e-04

        ratio_to_median_purchase_price  repeat_retailer  used_chip  \
570606                        0.000564              1.0        1.0
756283                        0.006263              1.0        0.0
738227                        0.005740              1.0        0.0
554038                        0.005348              1.0        0.0
712266                        0.000285              1.0        0.0
…                                  …              …         …
259178                        0.000467              0.0        0.0
365838                        0.005673              1.0        0.0
131932                        0.003401              1.0        0.0
671155                        0.004368              1.0        0.0
121958                        0.005071              0.0        1.0

        used_pin_number  online_order
570606              0.0           1.0
756283              0.0           1.0
738227              0.0           1.0
554038              0.0           1.0
712266              0.0           0.0
…                   …           …
259178              0.0           0.0
365838              0.0           0.0
131932              0.0           0.0
671155              0.0           1.0
121958              0.0           0.0

[750000 rows x 7 columns]
```

[25]:
```
val_inputs
```

[25]:
```
        distance_from_home  distance_from_last_transaction  \
987231            0.000087                        0.000109
79954             0.000057                        0.000018
567130            0.000372                        0.000045
500891            0.002050                        0.000002
55399             0.000311                        0.000144
…                    …                               …
619805            0.001124                        0.000771
513543            0.007748                        0.000315
```

```
283945          0.000515                         0.000016
498596          0.000091                         0.000275
635068          0.000610                         0.000537

        ratio_to_median_purchase_price  repeat_retailer  used_chip  \
987231                        0.001332              0.0        0.0
79954                         0.011630              0.0        0.0
567130                        0.005883              1.0        0.0
500891                        0.042616              1.0        0.0
55399                         0.007560              1.0        0.0
…                                  …              …        …
619805                        0.004578              1.0        1.0
513543                        0.045899              1.0        1.0
283945                        0.002236              1.0        0.0
498596                        0.003650              0.0        1.0
635068                        0.010309              1.0        0.0

        used_pin_number  online_order
987231              0.0           1.0
79954               0.0           1.0
567130              0.0           0.0
500891              0.0           0.0
55399               0.0           0.0
…                  …            …
619805              0.0           0.0
513543              0.0           0.0
283945              1.0           0.0
498596              0.0           0.0
635068              0.0           0.0

[250000 rows x 7 columns]
```

[26]:
```
train_targets
```

[26]:
```
570606    no
756283    no
738227    no
554038    no
712266    no
          ..
259178    no
365838    no
131932    no
671155    no
121958    no
Name: fraud, Length: 750000, dtype: object
```

[27]:
```python
val_targets
```

[27]:
```
987231    no
79954     no
567130    no
500891    no
55399     no
          ..
619805    no
513543    no
283945    no
498596    no
635068    no
Name: fraud, Length: 250000, dtype: object
```

5 Training the First Model: Logistic Regression

[28]:
```python
from sklearn.linear_model import LogisticRegression
```

Training

[29]:
```python
%%time
logistic_reg_model = LogisticRegression(solver='liblinear')
logistic_reg_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 4.08 s
Wall time: 4.1 s
```

[29]:
```
LogisticRegression(solver='liblinear')
```

Train Accuracy

[30]:
```python
log_reg_train_accuracy = logistic_reg_model.score(train_inputs[numerical_cols],␣
↪train_targets)
```

[31]:
```python
log_reg_train_accuracy
```

[31]:
```
0.9438373333333333
```

[32]:
```python
train_probs = logistic_reg_model.predict_proba(train_inputs[numerical_cols])
```

[33]:
```python
train_probs
```

[33]:
```
array([[0.96326231, 0.03673769],
       [0.89572206, 0.10427794],
       [0.90102388, 0.09897612],
       …,
       [0.99847642, 0.00152358],
       [0.9167462 , 0.0832538 ],
```

```
        [0.99893163, 0.00106837]])
```

Validation Accuracy

[34]:
```python
log_reg_val_accuracy = logistic_reg_model.score(val_inputs[numerical_cols],␣
 ↪val_targets)
```

[35]:
```python
log_reg_val_accuracy
```

[35]:
```
0.944284
```

[36]:
```python
val_probs = logistic_reg_model.predict_proba(val_inputs[numerical_cols])
```

[37]:
```python
val_probs
```

[37]:
```
array([[9.31292761e-01, 6.87072386e-02],
       [7.61307650e-01, 2.38692350e-01],
       [9.98081199e-01, 1.91880054e-03],
       …,
       [9.99999475e-01, 5.25492160e-07],
       [9.99121948e-01, 8.78051989e-04],
       [9.96297887e-01, 3.70211312e-03]])
```

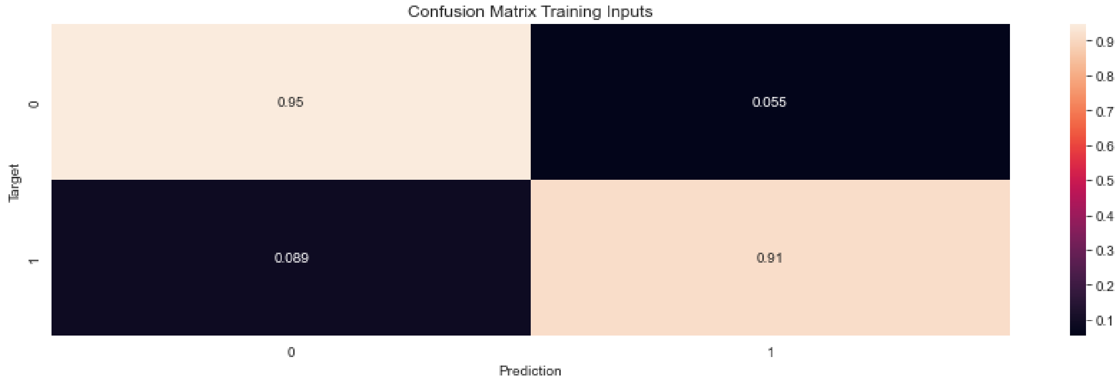
Visualization It will be used a confusion matrix to better visualize the situation.

For training:

[38]:
```python
from sklearn.metrics import confusion_matrix


conf_mx_train = confusion_matrix(logistic_reg_model.
 ↪predict(train_inputs[numerical_cols]), train_targets, normalize='true')
conf_mx_train
```

[38]:
```
array([[0.94514012, 0.05485988],
       [0.08898202, 0.91101798]])
```

[147]:
```python
plt.figure(figsize=(15, 4))
sns.heatmap(conf_mx_train, annot=True)
plt.xlabel('Prediction')
plt.ylabel('Target')
plt.title('Confusion Matrix Training Inputs');
```

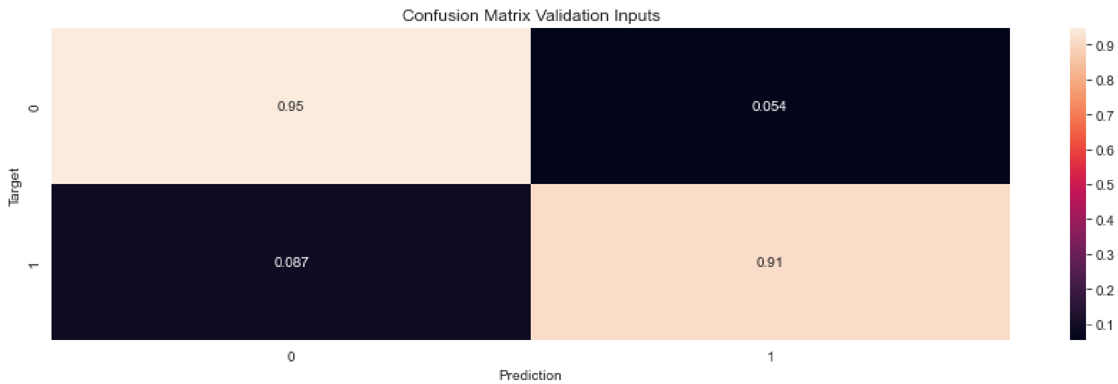


For validation:

[40]:
```
conf_mx_val = confusion_matrix(logistic_reg_model.
  ↪predict(val_inputs[numerical_cols]), val_targets, normalize='true')
conf_mx_val
```

[40]:
```
array([[0.94551078, 0.05448922],
       [0.08691974, 0.91308026]])
```

[148]:
```
plt.figure(figsize=(15, 4))
sns.heatmap(conf_mx_val, annot=True)
plt.xlabel('Prediction')
plt.ylabel('Target')
plt.title('Confusion Matrix Validation Inputs');
```



6 Weights Importance

It is important to start by seeing the most important weights.

[42]:
```
log_regr_weights = logistic_reg_model.coef_
log_regr_weights = log_regr_weights.reshape(7,)
log_regr_weights
```

[42]:
```
array([ 77.57255434,  33.27357992, 141.02183367,  -0.28304987,
        -0.73681525,  -7.70059214,   4.02612461])
```

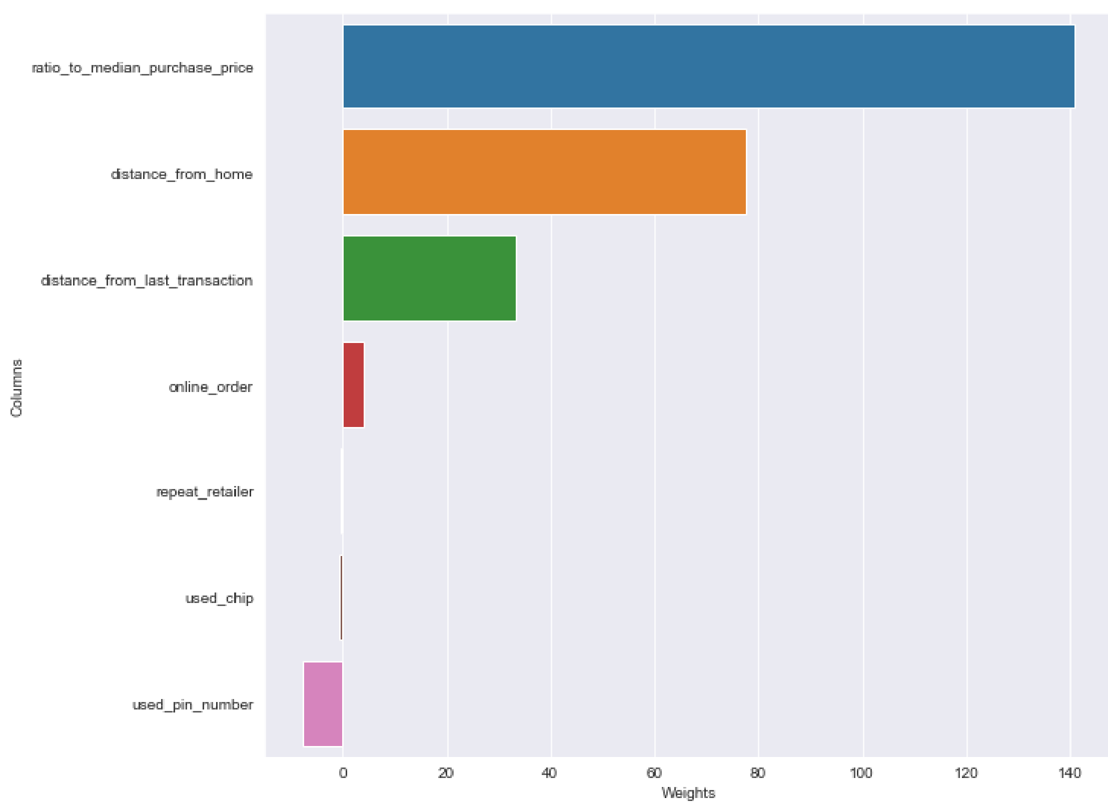
[43]:
```
log_regr_weights_df = pd.DataFrame({'Columns': train_inputs.columns, 'Weights':
 ↪log_regr_weights})
```

[44]:
```
log_regr_weights_df.sort_values('Weights', ascending=False)
```

[44]:

	Columns	Weights
2	ratio_to_median_purchase_price	141.021834
0	distance_from_home	77.572554
1	distance_from_last_transaction	33.273580
6	online_order	4.026125
3	repeat_retailer	-0.283050
4	used_chip	-0.736815
5	used_pin_number	-7.700592

[150]:
```
plt.figure(figsize=(10, 9))
sns.barplot(data=log_regr_weights_df.sort_values('Weights', ascending=False),
 ↪x='Weights', y='Columns');
```



The ratio to median purchase price seems to be the most important parameter by far.

7 Hypertuning

To make better predictions it is possible to change the so-called 'hyperparameters'. In particular: * solver; * class_weight.

Start by reporting the basic accuracies.

[45]:
```
base_accs = (log_reg_train_accuracy, log_reg_val_accuracy)
base_accs
```

[45]:
```
(0.9438373333333333, 0.944284)
```

Changing the solver from 'liblinear' to 'lbfgs'.

[46]:
```
%%time
log_regr_sol = LogisticRegression(solver='lbfgs')
log_regr_sol.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 10.4 s
Wall time: 7.7 s
```

[46]:
```
LogisticRegression()
```

[47]:
```
log_regr_sol_train_acc =log_regr_sol.score(train_inputs[numerical_cols],␣
 ↪train_targets)
log_regr_sol_val_acc = log_regr_sol.score(val_inputs[numerical_cols],␣
 ↪val_targets)

(log_regr_sol_train_acc, log_regr_sol_val_acc), base_accs
```

[47]:
```
((0.9439093333333334, 0.944316), (0.9438373333333333, 0.944284))
```

There is a little improvement.

Now, it is time for class_weight.

[48]:
```
logistic_reg_model.classes_
```

[48]:
```
array(['no', 'yes'], dtype=object)
```

[49]:
```
%%time
log_regr_cl_w = LogisticRegression(class_weight={'no':1, 'yes':3},␣
 ↪solver='liblinear')
log_regr_cl_w.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 4.33 s
Wall time: 4.29 s
```

[49]:
```
LogisticRegression(class_weight={'no': 1, 'yes': 3}, solver='liblinear')
```

[50]:
```python
log_regr_cl_w_train_acc = log_regr_cl_w.score(train_inputs[numerical_cols],␣
↪train_targets)
log_regr_cl_w_val_acc = log_regr_cl_w.score(val_inputs[numerical_cols],␣
↪val_targets)

(log_regr_cl_w_train_acc, log_regr_cl_w_val_acc), base_accs
```

[50]:
```
((0.962328, 0.962556), (0.9438373333333333, 0.944284))
```

[51]:
```python
def optimal_class(number):
    log_regr_cl_w = LogisticRegression(class_weight={'no':1, 'yes':number},␣
↪solver='liblinear')
    log_regr_cl_w.fit(train_inputs[numerical_cols], train_targets)
    log_regr_cl_w_train_acc = log_regr_cl_w.score(train_inputs[numerical_cols],␣
↪train_targets)
    log_regr_cl_w_val_acc = log_regr_cl_w.score(val_inputs[numerical_cols],␣
↪val_targets)
    return {'Class Number': number, 'Train Accuracy':log_regr_cl_w_train_acc,␣
↪'Validation Accuracy': log_regr_cl_w_val_acc}
```

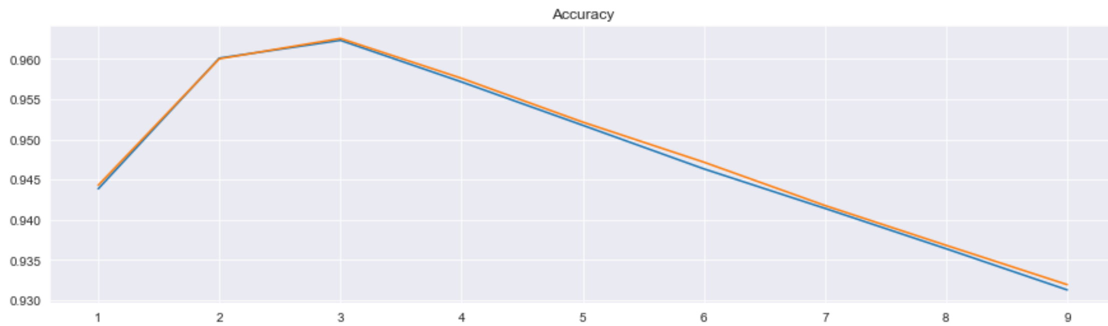
[52]:
```python
classes_df = pd.DataFrame([optimal_class(number) for number in range(1,10)])
```

[53]:
```python
classes_df
```

[53]:

	Class Number	Train Accuracy	Validation Accuracy
0	1	0.943837	0.944284
1	2	0.960108	0.960024
2	3	0.962328	0.962556
3	4	0.957145	0.957596
4	5	0.951755	0.952148
5	6	0.946329	0.947164
6	7	0.941413	0.941768
7	8	0.936395	0.936816
8	9	0.931265	0.931920

[142]:
```python
plt.figure(figsize=(15, 4))
plt.plot(classes_df['Class Number'], classes_df['Train Accuracy'])
plt.plot(classes_df['Class Number'], classes_df['Validation Accuracy'])
plt.title('Accuracy');
```



As it is possible to see by the graph, the class 'yes' should have a value of 3, while the 'no' class 1.

Try merging the two changed hyperparameters.

[55]:
```
%%time
final_regr_model = LogisticRegression(class_weight={'no':1, 'yes':3},␣
↪solver='lbfgs')
final_regr_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 12.6 s
Wall time: 8.11 s
```

[55]:
```
LogisticRegression(class_weight={'no': 1, 'yes': 3})
```

[56]:
```
final_regr_model.score(train_inputs[numerical_cols], train_targets),␣
↪final_regr_model.score(val_inputs[numerical_cols], val_targets)
```

[56]:
```
(0.9623226666666667, 0.962536)
```

[57]:
```
base_accs
```

[57]:
```
(0.9438373333333333, 0.944284)
```

Definitely better.

8 Making some Predictions

Let's see the original data frame before.

[58]:
```
raw_df
```

[58]:
```
        distance_from_home  distance_from_last_transaction  \
0                57.877857                        0.311140
1                10.829943                        0.175592
2                 5.091079                        0.805153
3                 2.247564                        5.600044
4                44.190936                        0.566486
```

```
…                   …                              …
999995       2.207101                       0.112651
999996      19.872726                       2.683904
999997       2.914857                       1.472687
999998       4.258729                       0.242023
999999      58.108125                       0.318110

        ratio_to_median_purchase_price  repeat_retailer  used_chip  \
0                             1.945940              1.0        1.0
1                             1.294219              1.0        0.0
2                             0.427715              1.0        0.0
3                             0.362663              1.0        1.0
4                             2.222767              1.0        1.0
…                                  …              …         …
999995                        1.626798              1.0        1.0
999996                        2.778303              1.0        1.0
999997                        0.218075              1.0        1.0
999998                        0.475822              1.0        0.0
999999                        0.386920              1.0        1.0

        used_pin_number  online_order fraud
0                   0.0           0.0    no
1                   0.0           0.0    no
2                   0.0           1.0    no
3                   0.0           1.0    no
4                   0.0           1.0    no
…                  …            …   …
999995              0.0           0.0    no
999996              0.0           0.0    no
999997              0.0           1.0    no
999998              0.0           1.0    no
999999              0.0           1.0    no

[1000000 rows x 8 columns]
```

Creating a new Input

[59]:
```
new_regr_input1 = {'distance_from_home':25.347021,
'distance_from_last_transaction': 3.146031, 'ratio_to_median_purchase_price': 5.
↪602498,
'repeat_retailer':1.0, 'used_chip':0.0, 'used_pin_number':1.0, 'online_order':1.
↪0}
```

[60]:
```
def input_try(input):
    new_input_df = pd.DataFrame([input])
    new_input_df[numerical_cols] = scaler.
↪transform(new_input_df[numerical_cols])
    pred = final_regr_model.predict(new_input_df[numerical_cols])[0]
```

```
    prob = final_regr_model.
↪predict_proba(new_input_df[numerical_cols])[0][list(final_regr_model.
↪classes_).index(pred)]
    return pred, prob
```

[61]:
```
input_try(new_regr_input1)
```

[61]: ('no', 0.9971788420081894)

Try another input.

[62]:
```
new_regr_input2 = {'distance_from_home':25.347021,
'distance_from_last_transaction': 3.146031, 'ratio_to_median_purchase_price': 5.
↪602498,
'repeat_retailer':1.0, 'used_chip':0.0, 'used_pin_number':0.0, 'online_order':1.
↪0}
```

[63]:
```
input_try(new_regr_input2)
```

[63]: ('yes', 0.7874864207116093)

The model is quite good, in particular as an alert system, for example, in this case, there is almost the 80% probability that we are facing a fraud.

9 Saving the model

[64]:
```
import joblib
```

[65]:
```
credit_card_fraud_det_model = {'model': final_regr_model, 'scaler': scaler,
               'input_cols': input_cols, 'target_col': target_col,␣
↪'numeric_cols': numerical_cols}

joblib.dump(credit_card_fraud_det_model, 'credit_card_fraud_detection')
```

[65]: ['credit_card_fraud_detection']

Load it again.

[66]:
```
credit_card_fraud_det_model2 = joblib.load('credit_card_fraud_detection')
```

[67]:
```
credit_card_fraud_det_model2
```

[67]:
```
{'model': LogisticRegression(class_weight={'no': 1, 'yes': 3}),
 'scaler': MinMaxScaler(),
 'input_cols': ['distance_from_home',
  'distance_from_last_transaction',
  'ratio_to_median_purchase_price',
  'repeat_retailer',
```

```
  'used_chip',
  'used_pin_number',
  'online_order'],
 'target_col': 'fraud',
 'numeric_cols': ['distance_from_home',
  'distance_from_last_transaction',
  'ratio_to_median_purchase_price',
  'repeat_retailer',
  'used_chip',
  'used_pin_number',
  'online_order']}
```

10 The Second Model: Random Forest

[68]:
```
from sklearn.ensemble import RandomForestClassifier
```

[69]:
```
%%time
ran_for_model = RandomForestClassifier(n_jobs=1, random_state=42)
ran_for_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 52.8 s
Wall time: 55 s
```

[69]:
```
RandomForestClassifier(n_jobs=1, random_state=42)
```

It takes quite a lot, but there are 1 mln rows.

[70]:
```
ran_forest_base_accs = ran_for_model.score(train_inputs[numerical_cols],
 ↪train_targets), ran_for_model.score(val_inputs[numerical_cols], val_targets)
```

[71]:
```
ran_forest_base_accs
```

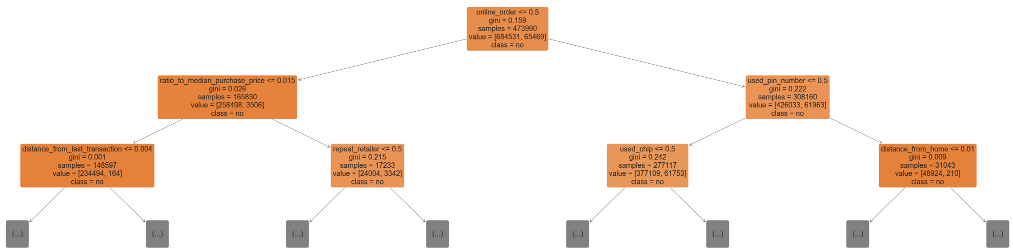
[71]:
```
(1.0, 0.999992)
```

The model seems to be good, but is it possible to improve it?

How is the model structured?

11 Visualization

[143]:
```
from sklearn.tree import plot_tree

plt.figure(figsize=(80,20))
plot_tree(ran_for_model.estimators_[0], max_depth=2,
 ↪feature_names=train_inputs[numerical_cols].columns, filled=True,
 ↪rounded=True, class_names=ran_for_model.classes_);
```



12 Feature Importance

[73]:
```
feature_importance_df = pd.DataFrame({'Feature': train_inputs[numerical_cols].
 ↪columns, 'Importance': ran_for_model.feature_importances_})
```

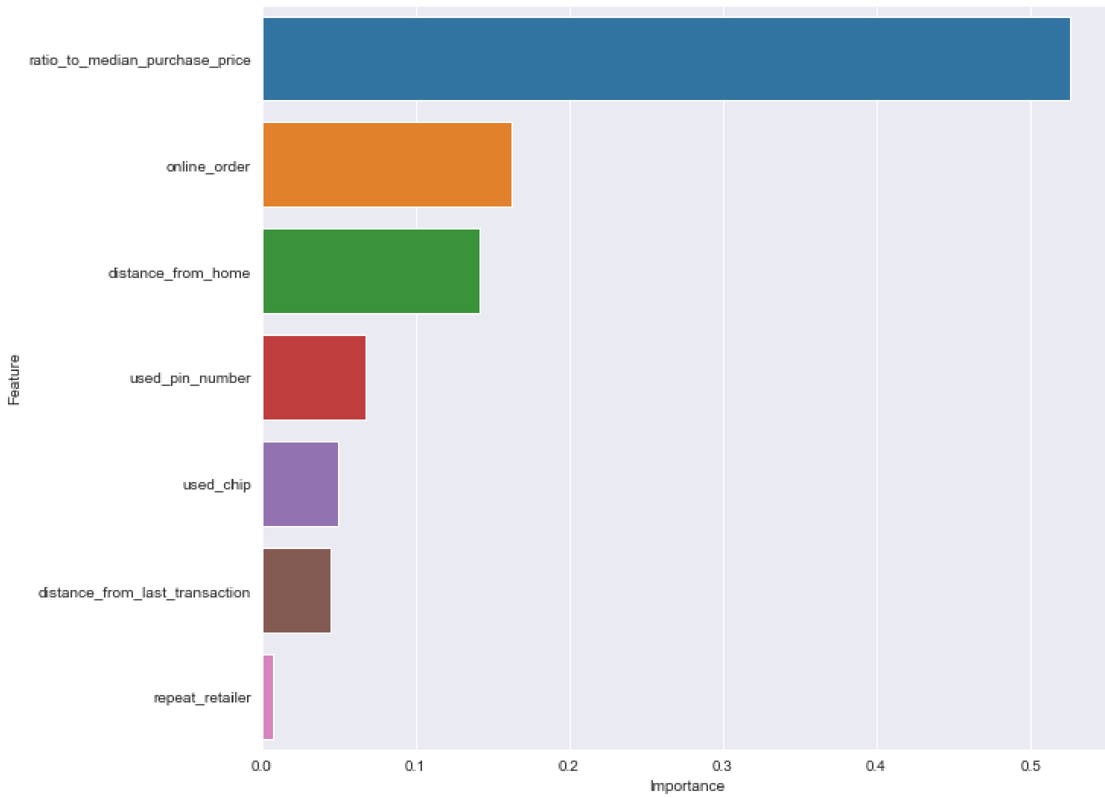
[74]:
```
feature_importance_df.sort_values('Importance', ascending=False)
```

[74]:

	Feature	Importance
2	ratio_to_median_purchase_price	0.525864
6	online_order	0.162705
0	distance_from_home	0.141537
5	used_pin_number	0.067906
4	used_chip	0.050073
1	distance_from_last_transaction	0.045195
3	repeat_retailer	0.006719

Plot them to better visualize the situation.

[151]:
```
plt.figure(figsize=(10,9))
sns.barplot(data=feature_importance_df.sort_values('Importance',␣
 ↪ascending=False), x='Importance', y='Feature');
```



It is confirmed the situation with the previous model.

13 Hypertuning

As already seen before, it would be right to change the class_weight parameter and to intervene on other two parameters: * max_depth * min_impurity_decrease

max_depth

[76]:
```
%%time
ran_for_md_model = RandomForestClassifier(n_jobs=1, random_state=42,␣
↪class_weight={'no':1, 'yes':3}, max_depth=5)
ran_for_md_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 45.9 s
Wall time: 47.6 s
```

[76]:
```
RandomForestClassifier(class_weight={'no': 1, 'yes': 3}, max_depth=5, n_jobs=1,
                       random_state=42)
```

[77]:
```
ran_for_md_model.score(train_inputs[numerical_cols], train_targets),␣
↪ran_for_md_model.score(val_inputs[numerical_cols], val_targets)
```

[77]: (0.999964, 0.999936)

[78]:
```
ran_forest_base_accs
```

[78]: (1.0, 0.999992)

It is not an improvement, but it has reduced the overfitting problem. Try with other max_depth.

[79]:
```
%%time
ran_for_md_model = RandomForestClassifier(n_jobs=1, random_state=42,␣
↪class_weight={'no':1, 'yes':3}, max_depth=7)
ran_for_md_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 52.6 s
Wall time: 52.7 s
```

[79]:
```
RandomForestClassifier(class_weight={'no': 1, 'yes': 3}, max_depth=7, n_jobs=1,
                       random_state=42)
```

[80]:
```
ran_for_md_model.score(train_inputs[numerical_cols], train_targets),␣
↪ran_for_md_model.score(val_inputs[numerical_cols], val_targets)
```

[80]: (0.9999986666666667, 0.999988)

[81]:
```
ran_forest_base_accs
```

[81]: (1.0, 0.999992)

Much better now.

min_impurity_decrease

[82]:
```
%%time
ran_for_mid_model = RandomForestClassifier(n_jobs=1, random_state=42,␣
↪class_weight={'no':1, 'yes':3}, min_impurity_decrease=1e-7)
ran_for_mid_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 57.4 s
Wall time: 57.4 s
```

[82]:
```
RandomForestClassifier(class_weight={'no': 1, 'yes': 3},
                       min_impurity_decrease=1e-07, n_jobs=1, random_state=42)
```

[83]:
```
ran_for_mid_model.score(train_inputs[numerical_cols], train_targets),␣
↪ran_for_mid_model.score(val_inputs[numerical_cols], val_targets)
```

[83]: (1.0, 0.999992)

[84]:
```
ran_forest_base_accs
```

[84]: (1.0, 0.999992)

The result is the same.

[85]:
```
%%time
ran_for_mid_model = RandomForestClassifier(n_jobs=1, random_state=42,␣
↪class_weight={'no':1, 'yes':3}, min_impurity_decrease=1e-6)
ran_for_mid_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 56.4 s
Wall time: 56.4 s
```

[85]:
```
RandomForestClassifier(class_weight={'no': 1, 'yes': 3},
                       min_impurity_decrease=1e-06, n_jobs=1, random_state=42)
```

[86]:
```
ran_for_mid_model.score(train_inputs[numerical_cols], train_targets),␣
↪ran_for_mid_model.score(val_inputs[numerical_cols], val_targets)
```

[86]:
```
(0.9999986666666667, 0.999992)
```

It has reduced the overfitting and kept the validation value equal.

Now, put this two parameters together.

[87]:
```
%%time
ran_for_final_model = RandomForestClassifier(n_jobs=1, random_state=42,␣
↪class_weight={'no':1, 'yes':3}, min_impurity_decrease=1e-6, max_depth=7)
ran_for_final_model.fit(train_inputs[numerical_cols], train_targets)
```

```
CPU times: total: 49.2 s
Wall time: 49.3 s
```

[87]:
```
RandomForestClassifier(class_weight={'no': 1, 'yes': 3}, max_depth=7,
                       min_impurity_decrease=1e-06, n_jobs=1, random_state=42)
```

[88]:
```
ran_for_final_model.score(train_inputs[numerical_cols], train_targets),␣
↪ran_for_final_model.score(val_inputs[numerical_cols], val_targets)
```

[88]:
```
(0.999996, 0.999996)
```

[89]:
```
ran_forest_base_accs
```

[89]:
```
(1.0, 0.999992)
```

At the end, the validation is higher and the overfitting is lower.

14 Making some Predictions

[113]:
```
def input_ran_for_try(input):
    new_input_df = pd.DataFrame([input])
    new_input_df[numerical_cols] = scaler.
↪transform(new_input_df[numerical_cols])
```

```
    pred = ran_for_final_model.predict(new_input_df[numerical_cols])[0]
    prob = ran_for_final_model.
 ↪predict_proba(new_input_df[numerical_cols])[0][list(ran_for_final_model.
 ↪classes_).index(pred)]
    return pred, prob
```

Try with the same inputs of the previous model.

[114]:
```
input_ran_for_try(new_regr_input1), input_ran_for_try(new_regr_input2)
```

[114]:
```
(('no', 0.6588253407008182), ('yes', 0.8768984406397538))
```

At the end, the output is the same, but probabilities have changed. The first one decreased, the second one increased.

15 Save the Model

[92]:
```
import joblib
```

[93]:
```
credit_card_ran_for_fraud_det_model = {'model': ran_for_final_model, 'scaler':␣
 ↪scaler,
                'input_cols': input_cols, 'target_col': target_col,␣
 ↪'numeric_cols': numerical_cols}

joblib.dump(credit_card_ran_for_fraud_det_model,␣
 ↪'credit_card_ran_for_fraud_detection')
```

[93]:
```
['credit_card_ran_for_fraud_detection']
```

Loading it.

[94]:
```
credit_card_ran_for_fraud_det_model2 = joblib.
 ↪load('credit_card_ran_for_fraud_detection')
```

[95]:
```
credit_card_ran_for_fraud_det_model2
```

[95]:
```
{'model': RandomForestClassifier(class_weight={'no': 1, 'yes': 3}, max_depth=7,
                        min_impurity_decrease=1e-06, n_jobs=1, random_state=42),
 'scaler': MinMaxScaler(),
 'input_cols': ['distance_from_home',
  'distance_from_last_transaction',
  'ratio_to_median_purchase_price',
  'repeat_retailer',
  'used_chip',
  'used_pin_number',
  'online_order'],
 'target_col': 'fraud',
 'numeric_cols': ['distance_from_home',
```

```
'distance_from_last_transaction',
'ratio_to_median_purchase_price',
'repeat_retailer',
'used_chip',
'used_pin_number',
'online_order']}
```

16 Conclusions

The work mainly focused on two different Machine Learning types:
* Logistic Regression * Random Forest
The results can be updated in the future by using more accurate models and by studying in more details the hyperparameters of each model.
The models developed today can be helpful for future works.