To loan, or not to loan

Data Mining Project - ECAC

Group 27
Pietro Obbiso - up202001628
Tiago Ribeiro - up201605619

Domain Description

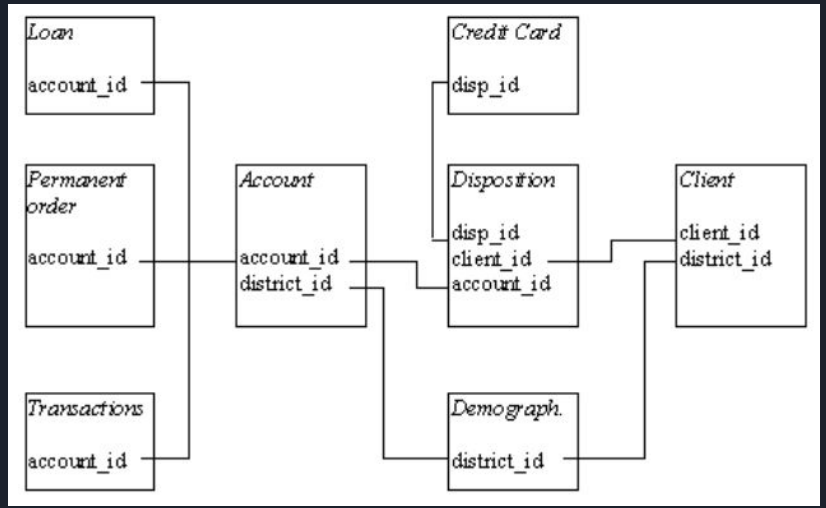
- The project, which consists in a predictive data mining task, is based on banking data. In particular, the target subject is a Czech bank that wants to improve their services.

- In order for the bank managers to understand who is a good client, that is, to whom to offer some additional services, and who is a bad client, that is, to whom to watch carefully to minimize the bank loses, we will try to predict whether a loan will end successfully or not.

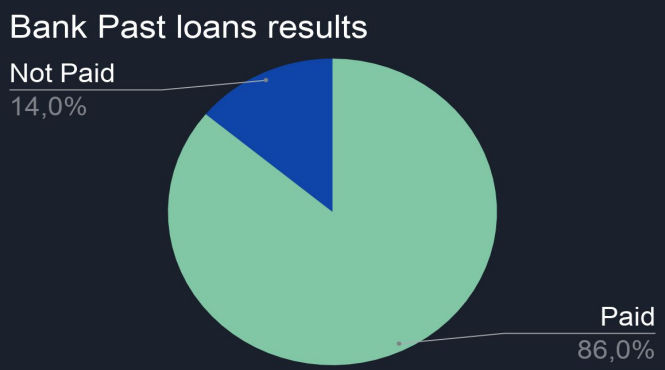
- To address this problem, the bank stored data, from 1993 to 1996, about their clients, including the loans already granted and the credit cards issued.

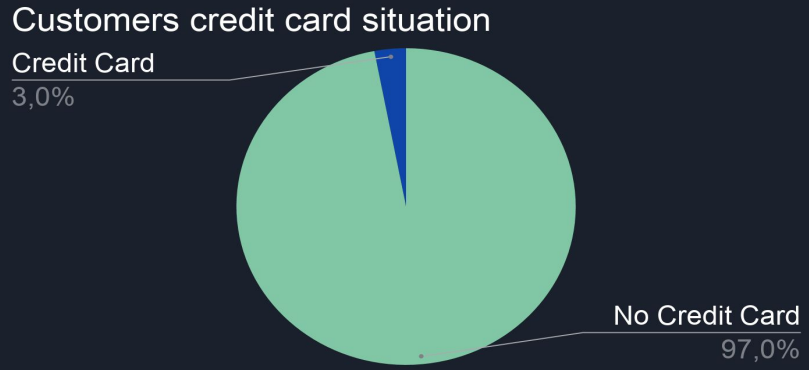
Exploratory Data Analysis

- Dataset comprises of multiple tables (account, client, disposition, permanent order, transaction, loan, credit card, demographic data) connected among each other
- Development data: **“loan_train”** table where **“status”** corresponds to the class; it assumes value 1 when the loan is paid, -1 when the loan is not paid
- Development data is strongly **unbalanced** towards the positive class (status = 1), with a large proportion
- No presence of **missing values** or **duplicates** in the development data



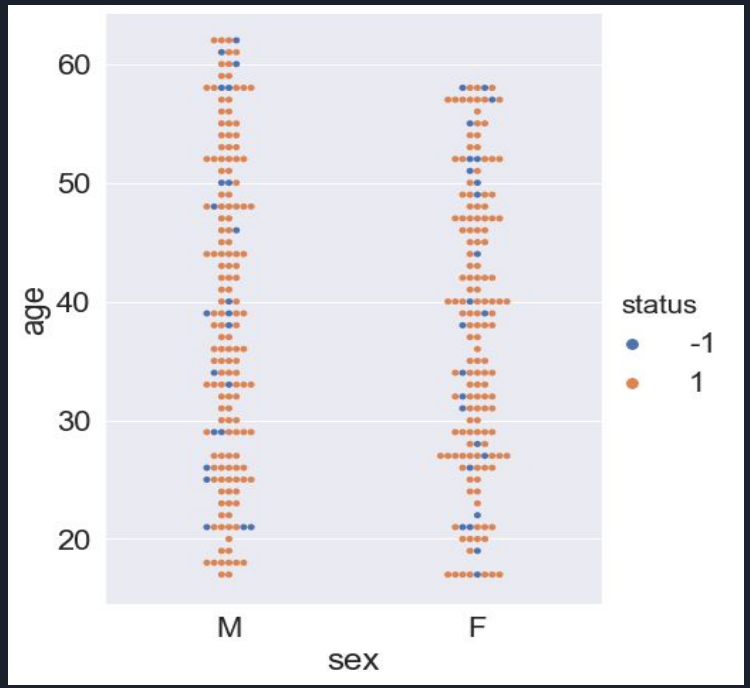
Exploratory Data Analysis

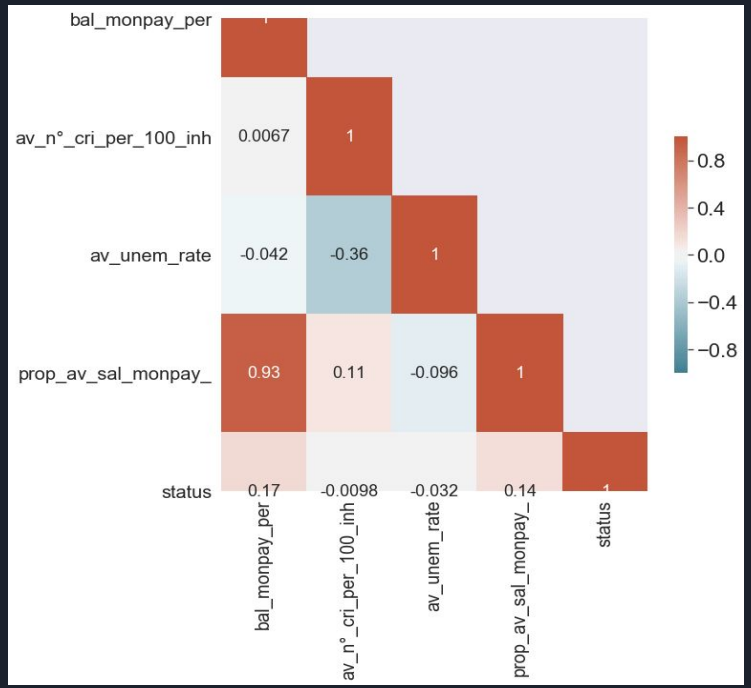




Credit card = {junior, classic, gold}

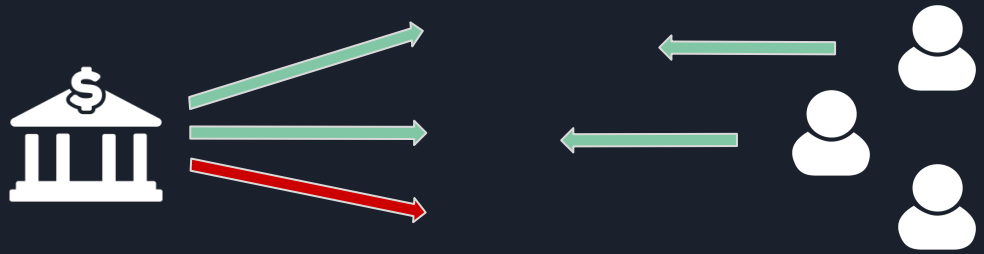
Exploratory Data Analysis





Problem Definition

- The goal of the Bank is to not lose the lended money, meaning that all customers need to pay back the money they got on a loan plus interest, so that a profit can be made. If too many customers don't pay, the bank runs out of money.
- In order for this to be possible, the Bank needs to make a selection of clients it can trust, based only the information it has available on their history as customers.
- So, considering all of this, the main goal of this Data Mining project is to predict which customers are more likely to successfully pay back a loan they are trying to request.
- This model will be built based on data from other customers that the Bank lended money in the past and that might or not payed back the loan. It should be possible to establish some patterns between different clients that will enable the model to predict the outcome of a future loan.



Data preparation

In order to get the bank data ready to be processed by the model, the customers data was cleanup up.

Feature selection:

- € amount, duration and payments from loan
- credit card type, sex an age of the customer

Also, some features were engineered from existing ones:

balance_monthlypayment_percentage - A customer should be able to comfortably afford the monthly payments of the loan. Based on the bank account history, the average monthly balance was calculated. Then, we try to analyze the percentage of this balance that would be “eaten up” to pay the new loan.

average_no_crimes_per_100_habitants - High crime rates are usually related with poorer and unstable areas, where granting a higher loan might be more difficult.

average_unemployment_rate - An area with an high unemployment rate might be an indication that employment is not very stable. A person living here might have an higher chance of being/becoming unemployed, seriously affecting her ability to pay back the loan.

proportion_avgsalary_monthlypayments - People living in areas with an higher average salary obviously have, on average, higher salaries. This means that they should be able to afford higher monthly payments.

Data preparation

Dataset Balancing:

The provided customer dataset was clearly unbalanced:

Customers who paid back the loan: 282 Customers who didn't paid back the loan: 46

By upsampling the minority class (not paid loan), we were able to balance the data and improve the model predictions.

Dataset Scaling:

To use some models like the State Vector Machine, the features with continuous numerical values need to be scaled to a [0,1] range.

Transformation of discrete text values:

Models usually deal better with numerical values than textual ones, so those features with text data were converted to multiple binary columns (one for each text category).

Experimental Setup

1. To extract the information from the provided .csv files, Python Pandas library was used.
2. Using python, this data is then organized into the features that will be used.
3. Using Pandas get_dummies, categorical columns are separated into binary ones.
4. Using SkLearn python libraries, the data is upsampled and then scaled.
5. Using again SkLearn libraries, a model is selected, alongside its hyperparameters.
6. This model is evaluated using a 5-fold cross-validation evaluation.
7. After a model is selected and refined, a prediction is made based on the challenge data.
8. This prediction is written in a .csv file following the Kaggle format.

python™ pandas scikit learn kaggle

Results

Several models were tried with several combinations of hyperparameters, to try to maximize the precision of the predictions:

1. **Decision Tree:** (SkLearn DecisionTreeClassifier library)
   - \+ Few requirements in terms of data preparation
   - \+ Easy to understand and visualize
   - \- Can can generate very complex trees, resulting in overfitting
   - \- Hard to adapt into a complex situation like this one (many features)

Best result on private Kaggle: **0.73045** Decision tree with (criterion="entropy",max_depth=7,min_impurity_decrease=0.015)

2. **Support Vector Machine:** (SkLearn SupportVectorMachine library)
   - \+ Effective in cases with a high number of features (exactly the type of case we are working)
   - \- Probabilities are not generated directly by the model. An additional calculation needs to exist (This might explain the obtained low results)

Best result on private Kaggle: **0.67901** State Vector Machine with (kernel='poly', probability=True)

Results

3. **K-nearest neighbors :** (SkLearn KNeighborsClassfier library)
   - \+ Simple algorithm and hence easy to interpret the prediction
   - \+ Training step is much faster compared to other machine learning algorithms
   - \- Computationally expensive as it searches the nearest neighbors for the new point in the prediction stage
   - \- Sensitive to outliers, accuracy is impacted by noise or irrelevant data

Best result on private Kaggle: **0.63539** KNeighborsClassifier(n_neighbors=5)

4. **Neural Networks:** (SkLearn Neural_NetworkMLPclassifier library)
   - \+ Once trained, the predictions are pretty fast
   - \+ Very flexible. Any data which can be made numeric can be used in the model
   - \- Neural networks depend a lot on training data. This leads to the problem of over-fitting and generalization.
   - \- Even with few attributes, the number of parameters to estimate is extremely large

No results on private Kaggle

Conclusions, limitations and future works

- Among all the models trained, the **Decision Tree** obtained the best prediction.
- The generation of too many variables might have made the model susceptible to the curse of dimensionality.
- By working with more observations, we could have the option of adopting more complex algorithms. Also, engineering new sets of features would allows us to explore different patterns in data.

Model	AUC Score
Decision Tree	0.73045
Support Vector Machine	0.67901
K-nearest neighbors	0.63509

Annexes

Python Code Overview

Libraries and methods

```
import pandas as pd
import numpy as np
import sys
import datetime
import seaborn as sns
from tqdm import tqdm
from sklearn import preprocessing
from sklearn import tree
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from keras import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt
from sklearn.utils import resample
```

Load data from csv files

```
df_train = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/loan_train.csv', sep=';', header=0)
df_test = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/loan_test.csv', sep=';', header=0)

df_dispositions = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/disp.csv', sep=';', header=0)
df_clients = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/client.csv', sep=';', header=0)
df_districts = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/district.csv', sep=';', header=0)
df_all_transactions = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/trans_train.csv', sep=';', header=0)
df_all_transactions_test = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/trans_test.csv', sep=';', header=0)
df_credit_cards = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/card_train.csv', sep=';', header=0)
df_credit_cards_test = pd.read_csv(filepath_or_buffer='/content/drive/My Drive/ECAC/card_test.csv', sep=';', header=0)


df_train_processed = pd.DataFrame(columns=['amount', 'duration', 'payments', 'balance_monthlypayment_percentage',
'average_no_crimes_per_100_habitants', 'average_unemployment_rate', 'proportion_avgsalary_monthlypayments','account_credit_Card_type','sex','age',
'status'])
df_test_processed = pd.DataFrame(columns=['amount', 'duration', 'payments', 'balance_monthlypayment_percentage',
'average_no_crimes_per_100_habitants', 'average_unemployment_rate', 'proportion_avgsalary_monthlypayments','account_credit_Card_type','sex','age',
'loan_id'])


for index_loan, loan in tqdm(df_train.iterrows()):
    df_train_processed.loc[index_loan] = [loan['amount'], loan['duration'], loan['payments'],
get_feature_balance_to_montlhypayment_percentage(loan,False),     get_feature_average_no_crimes_per_100_habitants(loan),
get_feature_average_unemployment_rate(loan),
get_feature_proportion_avgsalary_monthlypayments(loan),get_feature_account_credit_Card_type(loan,False),get_feature_sex(loan),get_feature_age(lo
an), loan['status']]
```

Feature Engineering

```
def get_feature_balance_to_montlhypayment_percentage(loan,test):
    df_cur_transactions = df_all_transactions_test if test else df_all_transactions
    df_loan_transactions = df_cur_transactions.loc[df_cur_transactions['account_id'] == loan['account_id']] #get all transactions for the
account of the loan

    year = []
    month = []

    for index, transaction in df_loan_transactions.iterrows(): #gets year and month for each transaction
        trans_date = datetime.datetime.strptime(str(transaction['date']), "%y%m%d")
        year.append(trans_date.year)
        month.append(trans_date.month)

    df_loan_transactions['year'] = year
    df_loan_transactions['month'] = month

    df_mean_balance_bymonth = df_loan_transactions.groupby(['year','month'])['balance'].mean().reset_index(name='balance')
    df_mean_balance_allmonth = df_mean_balance_bymonth['balance'].mean()

    return  df_mean_balance_allmonth / loan['payments']


def get_client_district_from_account(account_id):
    df_disposition = df_dispositions.loc[(df_dispositions['account_id'] == account_id) & (df_dispositions['type'] == 'OWNER')] #get the
disposition of the owner of the account
    df_client = df_clients.loc[df_clients['client_id'] == df_disposition.iloc[0]['client_id']] #get the info of the owner of the account
    return df_districts.loc[df_districts['code '] == df_client.iloc[0]['district_id']].iloc[0] #get the district info of the owner of the account
```

Feature Engineering

```
def get_feature_average_no_crimes_per_100_habitants(loan):

    district = get_client_district_from_account(loan['account_id'])

    no_crimes_95 = district['no. of commited crimes \'95 ']
    no_crimes_96 = district['no. of commited crimes \'96 ']

    no_crimes_95 = no_crimes_96 if no_crimes_95 == '?' else no_crimes_95
    no_crimes_96 = no_crimes_95 if no_crimes_96 == '?' else no_crimes_96

    return  ((int(no_crimes_95)+int(no_crimes_96))/2)/int(district['no. of inhabitants'])*100


def get_feature_average_unemployment_rate(loan):

    district = get_client_district_from_account(loan['account_id'])

    unemploymant_rate_95 = district['unemploymant rate \'95 ']
    unemploymant_rate_96 = district['unemploymant rate \'96 ']

    unemploymant_rate_95 = unemploymant_rate_96 if unemploymant_rate_95 == '?' else unemploymant_rate_95
    unemploymant_rate_96 = unemploymant_rate_95 if unemploymant_rate_96 == '?' else unemploymant_rate_96

    return  (float(unemploymant_rate_95)+float(unemploymant_rate_96))/2
```

Feature Engineering

```
def get_feature_proportion_avgsalary_monthlypayments(loan):

    district = get_client_district_from_account(loan['account_id'])

    return  int(district['average salary '])/int(loan['payments'])



def get_feature_account_credit_Card_type(loan,test):

    df_cur_credit_cards = df_credit_cards_test if test else df_credit_cards

    df_loan_disposition = df_dispositions.loc[(df_dispositions['account_id'] == loan['account_id'])& (df_dispositions['type'] == 'OWNER')]
    df_credit_card_disposition = df_cur_credit_cards.loc[df_cur_credit_cards['disp_id'] == df_loan_disposition.iloc[0]['disp_id']]
    if (len(df_credit_card_disposition.index) == 1):
        return df_credit_card_disposition.iloc[0]["type"]
    else:
        return "no credit card"
```

Feature Engineering

```
def get_feature_sex(loan):

    df_loan_disposition = df_dispositions.loc[df_dispositions['account_id'] == loan['account_id']]
    df_client_disposition = df_clients.loc[df_clients['client_id'] == df_loan_disposition.iloc[0]['client_id']]

    trans_date = list(str(df_client_disposition.iloc[0]['birth_number']))

    month = int(trans_date[2] + trans_date[3])
    #print(month)

    if (month > 12):
        return 'F'
    else:
        return 'M'


def get_feature_age(loan):

    df_loan_disposition = df_dispositions.loc[df_dispositions['account_id'] == loan['account_id']]
    df_client_disposition = df_clients.loc[df_clients['client_id'] == df_loan_disposition.iloc[0]['client_id']]


    trans_date = list(str(df_client_disposition.iloc[0]['birth_number']))

    year = int(trans_date[0] + trans_date[1])
    age = 97 - year

    return age
```

Transform categorical data in binary values

```
df_data = pd.get_dummies(df_train_processed.drop(columns=['status']), columns=['account_credit_Card_type','sex'])

df_test_target = pd.get_dummies(df_test_processed.drop(columns=['loan_id']), columns=['account_credit_Card_type','sex'])
```

Upsample minority class

```
df_merged = pd.concat([df_data,df_target],axis=1)

# Separate majority and minority classes
df_merged_majority = df_merged[df_merged.status==1]
df_merged_minority = df_merged[df_merged.status==-1]

df_minority_upsampled = resample(df_merged_minority,
                                 replace=True,     # sample with replacement
                                 n_samples=282,    # to match majority class
                                 random_state=123) # reproducible results


# Combine majority class with upsampled minority class
df_upsampled = pd.concat([df_merged_majority, df_minority_upsampled])

df_upsampled.status.value_counts()

df_upsample_data = df_upsampled.drop('status',axis=1)

df_upsample_target = df_upsampled[['status']]
df_upsample_target = df_upsample_target.astype(int)

df_data_upsampled=df_upsample_data
df_target_upsampled=df_upsample_target
```

Downsample majority class

```
df_majority_downsampled = resample(df_merged_majority,
                                   replace=False,    # sample with replacement
                                   n_samples=46,    # to match majority class
                                   random_state=123) # reproducible results

# Combine majority class with upsampled minority class
df_downsampled = pd.concat([df_majority_downsampled, df_merged_minority])


df_downsampled.status.value_counts()


df_downsampled_data = df_downsampled.drop('status',axis=1)


df_downsampled_target = df_downsampled[['status']]
df_downsampled_target = df_downsampled_target.astype(int)

df_data_downsampled=df_downsampled_data
df_target_downsampled=df_downsampled_target
```

Balancing combining the two approaches

```
df_majority_downsampled_balance = resample(df_merged_majority,
                    replace=False,    # sample with replacement
                    n_samples=164,    # to match minority class
                    random_state=123) # reproducible results

df_minority_upsampled_balance = resample(df_merged_minority,
                    replace=True,    # sample with replacement
                    n_samples=164,    # to match majority class
                    random_state=123) # reproducible results

# Combine majority class with upsampled minority class
df_balance = pd.concat([df_majority_downsampled_balance, df_minority_upsampled_balance])

df_balance.status.value_counts()

df_data_balance = df_balance.drop('status',axis=1)

df_target_balance = df_balance[['status']]
df_target_balance = df_target_balance.astype(int)
```

Scale values to [0,1] range

```
min_max_scaler = preprocessing.MinMaxScaler()

df_data_selected[['amount', 'duration', 'payments', 'balance_monthlypayment_percentage', 'average_no_crimes_per_100_habitants',
'average_unemployment_rate', 'proportion_avgsalary_monthlypayments','age']] = min_max_scaler.fit_transform(df_data_selected[['amount',
'duration', 'payments', 'balance_monthlypayment_percentage', 'average_no_crimes_per_100_habitants', 'average_unemployment_rate',
'proportion_avgsalary_monthlypayments','age']])

df_test_target[['amount', 'duration', 'payments', 'balance_monthlypayment_percentage', 'average_no_crimes_per_100_habitants',
'average_unemployment_rate', 'proportion_avgsalary_monthlypayments','age']] = min_max_scaler.fit_transform(df_test_target[['amount',
'duration', 'payments', 'balance_monthlypayment_percentage', 'average_no_crimes_per_100_habitants', 'average_unemployment_rate',
'proportion_avgsalary_monthlypayments','age']])
```

Model selection, tuning and evaluation

```
(examples of possible models)
model = tree.DecisionTreeClassifier(criterion="entropy",max_depth=20,min_samples_split=10,min_impurity_decrease=0.002)
model = tree.DecisionTreeClassifier(criterion="entropy")

model = svm.SVC(kernel='poly', probability=True)

model = KNeighborsClassifier(n_neighbors=5)

model = Sequential()
model.add(Dense(16, input_dim=14, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

(classify model with 5-fold-validation)
scores = cross_val_score(model, df_data_selected, df_target_selected.values.ravel(), scoring='roc_auc')
avg_scores = np.mean(scores)
print(avg_scores)
```

Model selection, tuning and evaluation

```
(fit the model and make a prediction)
model.fit(df_data_selected, df_target_selected)
y_predicted_train = model.predict(df_test_target)

y_predicted_train = model.predict_proba(df_test_target)[:,1]



(write the .csv file with all results to submit on kaggle)
original_stdout = sys.stdout  # Save a reference to the original standard output

with open('result.csv', 'w') as f:
  sys.stdout = f

  print('Id,Predicted')
  print()
  for i in range(0, len(df_test_id['loan_id'])):
    print(df_test_id['loan_id'][i], ',', y_predicted_train[i])
    print()

  sys.stdout = original_stdout
```

Hyperparameter tuning

```
#Automatically hyperparameter tuning (example for K-Nearest Neighbour)

leaf_size = list(range(1,50))
n_neighbors = list(range(1,30))
p=[1,2]

#Convert to dictionary
hyperparameters = dict(leaf_size=leaf_size, n_neighbors=n_neighbors, p=p)

#Create new KNN object
knn_2 = KNeighborsClassifier()

#Use GridSearch
clf = GridSearchCV(knn_2, hyperparameters, cv=10)

#Fit the model
best_model = clf.fit(df_upsample_data, df_upsample_target)

#Print The value of best Hyperparameters
print('Best leaf_size:', best_model.best_estimator_.get_params()['leaf_size'])
print('Best p:', best_model.best_estimator_.get_params()['p'])
print('Best n_neighbors:', best_model.best_estimator_.get_params()['n_neighbors'])
```

Plots

```
df_train_processed['status'].value_counts().plot(kind='pie', autopct='%1.0f%%',textprops={'fontsize': 20},title='Customers who paid vs not paid',label='')

df_train_processed['account_credit_Card_type'] = df_train_processed['account_credit_Card_type'].replace({'classic':'credit card', 'junior':'credit card','gold':'credit card'})
df_train_processed['account_credit_Card_type'].value_counts().plot(kind='pie', autopct='%1.0f%%',textprops={'fontsize': 20},title='Customers with credit card vs without credit card',label='')

plt.figure(figsize = (10,10))
corr = df_train_processed.corr()# plot the heatmap
x_axis_labels = ['bal_monpay_per', 'av_n°_cri_per_100_inh', 'av_unem_rate','prop_av_sal_monpay_', 'status'] # labels for x-axis
y_axis_labels = ['bal_monpay_per', 'av_n°_cri_per_100_inh', 'av_unem_rate' ,'prop_av_sal_monpay_', 'status']
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
mask[np.diag_indices_from(mask)] = False
res = sns.heatmap(corr, xticklabels=x_axis_labels, yticklabels=y_axis_labels, mask=mask, vmin=-1, vmax = 1, annot=True, annot_kws={"size": 18}, cmap=sns.diverging_palette(220, 20, as_cmap=True),cbar_kws={"shrink": .5})
res.set_xticklabels(res.get_xmajorticklabels(), fontsize = 20)
res.set_yticklabels(res.get_ymajorticklabels(), fontsize = 20)

sns.catplot(x="sex", y="age", hue="status",  data=df_train_processed, kind="swarm",  height=8, aspect=.8)
```

In-Depth model results analysis

Decision Tree

The initial results using a decision tree were very encouraging. Just by using applying the model without any hyperparameters, getting 0.67160 in Kaggle private classification.

The biggest problem arose when we tried to tune the hyperparameters to improve the classification. Although our 5-fold cross-validation scores were amazing (above 0.8), the actual kaggle scores could pass the 0.6 barrier.

This means that the model was getting extremely overfitted to the sample we had, even after applying our evaluation model made specifically to reduce this phenomenon.

We ended up setting for a couple of hyperparameters that actually improved the kaggle score up to 0.73045.

Those were the maximum depth and the minimum impurity decrease. Both are very useful in preventing a unecessarly deep tree that would be overfitted to the evaluation data.

Support Vector Machine

The initial results from the SVM were pretty disappointing (under 0.5).

After some data processing, including balancing,scaling and binary from categorical data, the results seemed very promising, especially considering the fact that we had a relatively high number of features.

Unfortunately, changing parameters like the kernel, degree,etc. didn't really improved the obtained results. We weren't able to surpass the result of the decision tree in Kaggle.

This might be due to the data not being easily separable, based on the plotted graphs. This definitely had a big impact in the model performance. Maybe with more adequate engineered features, the results would be drastically different.

K-Nearest Neighbors

First we tried with the most simple model: the original unbalanced model with 5 neighbors, by default, and p=2 (Euclidean distance). This combination provided us a disappointed result in terms of roc_auc score, precisely 0.55.

So we tried to add the parameter weight = ‘distance’ in order to address the unbalancing problem by not giving equal weight to all points in each neighborhood anymore.

Then we considered the upsampled, the downsampled and the mix of two datasets that we had already created before and, therefore, we repeated the same procedure for all of them. We obtain, respectively, scoring values equal to: 0.81, 0.61 and 0.70.

Before deciding to submit the predictions provided by our best model, the one using the upsampled model, we tried to tune the hyperparameters (number of neighbors, metric and leaf size) in order to find out the best combination that gave us the best score. The scores indeed improved becoming, respectively, 0.90, 0.77 and 0.86.

However, after the submission, the public kaggle score passed to 0.55 with the mixed dataset and 0.6 with the upsampled one. This means that the model was getting extremely overfitted to the sample we had, even after applying our evaluation model made specifically to reduce this phenomenon.

Neural Networks and Logistic Regression

Not happy about the results obtained, we also tried to make an attempt with other two models, the Neural Networks and the Logistic Regression.

For what concern the Neural Network, we created a simple model with 3 dense layers. The first one with activation = ‘relu’ and input dimension = 14; the intermediate one with just activation = ‘relu’ and the last one, the output layer, with activation = ‘sigmoid’, since we were working at a binary classification problem. However, knowing that even with few attributes, the number of parameters is extremely large, which leads to an increasing danger of overfitting, handled if we would had lots of data, we considered it not very useful for our project, hence we didn’t make any further improvements..

Regarding the Logistic Regression, instead, we made just one trial with the simplest model first, without any parameters, and then adding class_weight = ‘balanced’, but results were quite poor, respectively 0.58 and 0.57.

Eventually, we decided to leave these models and put our final forces in the DT and SVM models.

Individual factor

Tiago Ribeiro +1

Pietro Obbiso -1