To loan or not to loan

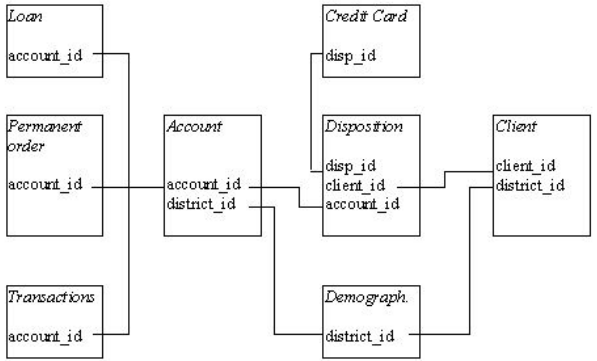
Data Mining Project

Beatriz Mendes - up201806551@up.pt
Henrique Pereira - up201806538@up.pt
Hugo Guimarães - up201806490@up.pt

Domain Description

Each table of the schema is correspondent to one of the provided files

- **4500 accounts**, that can be accessed by clients, which can be owners or disponents, with different types of credit cards
- Demographic data about the district, regarding the bank (accounts) and the clients
- **426885 transactions**, characterised by date, type, amount and operation.
- **682 loans**, 328 of which with known loan status

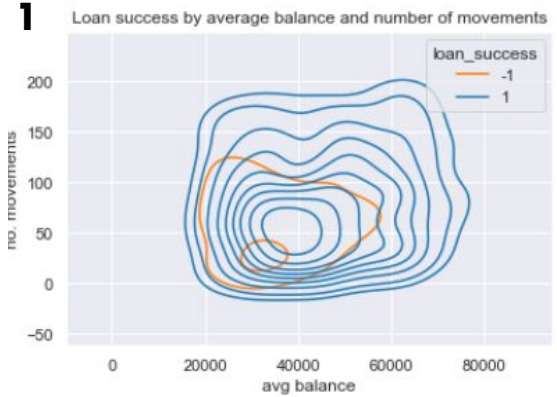


Exploratory Data Analysis

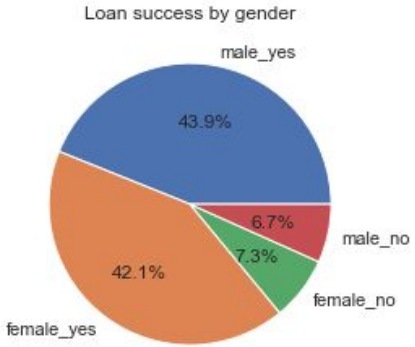
1. There is a very positive correlation between the pair of attributes *no. movements* and *avg balance* and the success of a loan.
2. Gender is not correlated to loan success
3. Loan success rate is lower when the loan amount is higher
4. There is no correlation between account age on loan and loan success
5. Over 80% of clients that ask for loans have no credit card
6. Whenever a client has a credit card, their loan is always successful
7. From the distribution of the status on the loans where it is known - 86% ended successfully - we can conclude that this attribute is unbalanced.
8. No client has more than one loan
9. No transactions occur on an account after a loan is made
10. The condition amount = payment * duration is always true

Exploratory Data Analysis

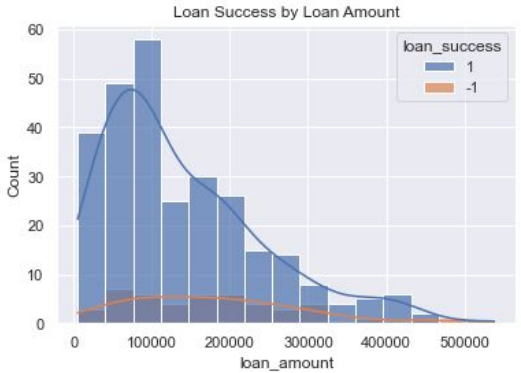
1



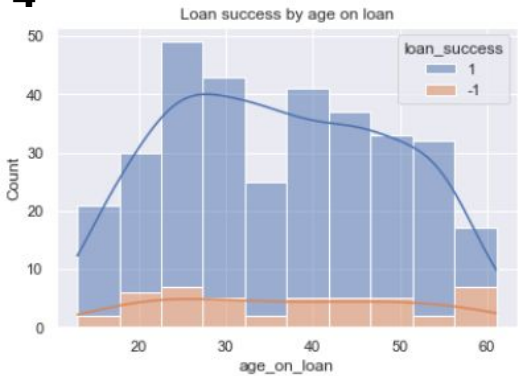
2



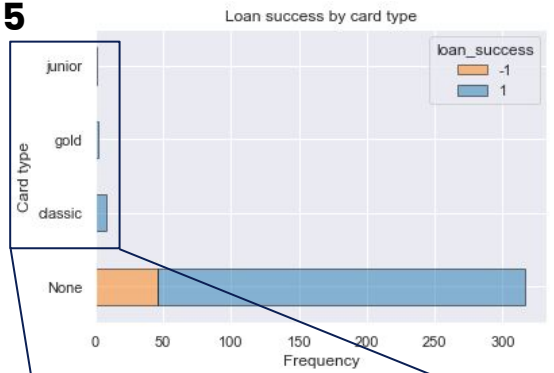
3



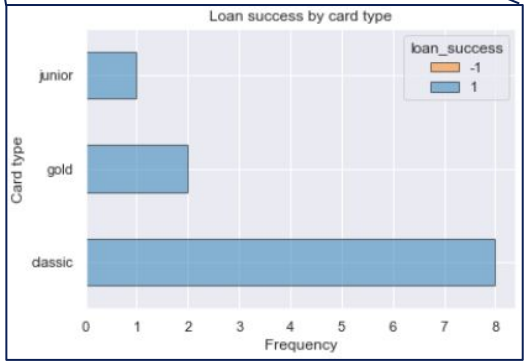
4



5



6

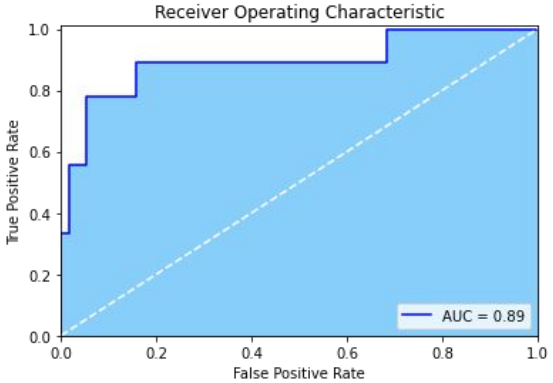


Problem Definition

Goal: Predict when loan will be unsuccessful (-1)

Problem Type: Classification problem where the variable to be predicted is either 1 or -1 (successful or unsuccessful, respectively)

Evaluation: Area Under the Curve (AUC) is a performance measurement for the classification problems at various threshold settings. It shows the rate of false positives and true positives, giving an insight on the performance of the model.



Data Preparation

Data Cleaning

- Transform dates into the format DD-MM-YYYY
- Replace empty values in *no. of commited crimes /'95* and *unemployment rate /'95* with the values from the next year ('96)
- Deal with categorical classes:
    - Label encoding to numerical in the same column
    - Creating dummy columns with binary (0 or 1) values for each category
- Deal with outliers (no outliers were detected, hence none were removed)
- Drop columns (**feature selection**) from *transactions* that have too many missing values (more than 80%)
- Replace missing values in the *operation* column of *transactions* with the most common occurrence.
- Consider only the owner for each account (drop disponents)

Data Preparation

Feature Engineering

- Client birthdate and gender calculated from *birth_number*
- Client age on loan calculated from birthdate and loan date
- District crime growth and unemployment growth between ‘95 and ‘96
- Total crime in ‘95 and ‘96
- Transactions statistics:
    - Minimum, maximum and average balance after transaction
    - Minimum, maximum and average transaction amount
    - Total number of transactions from each account

Experimental Setup

After the pre-processing of the data, the project has the following steps, for any algorithm:

- Flag Selection
  a. Boolean values - OVERSAMPLE, DUMMIES, CATEGORY_ENCONDING, MIN_MAX_SCALER
  b. Numerical values - SPLIT_RATIO, N_COLUMNS, N_SPLITS
- Prediction
  a. Select the most influencing attributes using the **selectKbest** function from sklearn (number of selected attributes can be changed by a flag)
  b. Split the datasets, using one of the following options:
     - **train_test_split** (default 80/20, changed by the flag)
     - **stratifiedKFold** - Split the data into K folds (value can be changed by a flag)
  c. Apply oversample technique using **SMOTE** (enabled/disabled by a flag)
  d. Apply **grid search** to evaluate the best parameters for the algorithm
  e. Apply the **chosen algorithm** with the obtained grid search parameters to get the predicted values
  f. Evaluate the model Score and AUC curve

Results

- When using the ***RandomForest*** algorithm, an average score of 86% was obtained
- The final result of the *Kaggle* competition was slightly lower, with approximately 84%
- After several experimentations, we have concluded that the ***Oversampling*** technique has resulted in better performances
- Several algorithms were used, such as ***RandomForest***, ***Logistic Regression***, ***Decision Tree*** and ***KNN***, and better results were obtained by using the first two, with the ***Random Forest*** achieving better predictions in the public leaderboard, while the best result in the private leaderboard was obtained through the ***Logistic Regression***
- The two splitting methods used belong to the *sklearn* library: ***train_test_split*** and ***stratifiedKfold***. The first one provided higher maximum scores, but the second obtained more consistent results. It is plausible to conclude that the first method was causing an overfit, maybe due to the provided split ratio.

Conclusions, Limitations and Future Work

- Conclusions
  - Feature Engineering and selection were the most important steps that lead to a better AUC curve
  - There are many algorithms that can be applied as a classification model, and each has their different parameters and require different pre-processing
  - We cannot blindly trust in an AUC curve drawn without knowledge about what are doing, because we can easily enter a state of overfitting

Conclusions, Limitations and Future Work

- Limitations
  - The loan training dataset was too small and imbalanced, which inadvertently lead to some increased overfitting, making us believe that our result were better than they actually were
- Future Work
  - Clustering
  - More feature Engineering/Selection - We believe that adding more features to our project may lead us to better results. The transactions table can be further explored.
  - Experiment other algorithms
  - Try different parameters in each algorithm

Annexes

Business Understanding

Business Goals

This project's goal was to improve the bank services by using the stored data about the customers to improve the bank's understanding of them, which was done through a Data Mining project which contains 2 problems:

- The predictive problem, whose goal was to predict whether a loan given to a client would end successfully
  a. Do a prediction for all loans (8)
  b. Interpret the results obtained (13)
  c. Improve the results to at least 90% correct predictions (21)
- The descriptive problem, which was related to the data analysis, and consisted of the process of observing and interpreting the data in order to find the relevance of each data component for the predictive model and the correlation between attributes
  a. Identify correlations between attributes (2)
  b. Determine the most relevant attributes (5)
  c. Explore the available data (3)
  d. Interpret the results obtained (5)

Business Understanding

Data Mining Goals

- Data preparation, by analyzing the provided data and its quality. These were studied by analyzing their outliers, missing, duplicate and inconsistent values, and deciding how to deal with them.
    a. Replace missing values with mean / previous year values (3)
    b. Remove outliers (5)
- Data understanding, by applying a statistical analysis to the data and plotting the appropriate graphs
    a. Plot the data using several correlated attributes (8)
    b. Interpret the plots (13)
    c. Analyse attribute statistics (minimum, maximum, mean) (5)
    d. Aggregate data into clusters (13)

Database

- We implemented a **Database** to make the reading of the data provided more efficient, since the reading of CSV files is known for not being the best approach (considerably slower).

```
con = sqlite3.connect("banking_data")
cur = con.cursor()

create_tables()
insert_into_tables()

con.close()
```

```
con = sqlite3.connect("../database/banking_data")
cur = con.cursor()

accounts = cleaning.clean_account(con)
cards = cleaning.clean_card(con, test)
clients = cleaning.clean_client(con)
districts = cleaning.clean_district(con)
loans = cleaning.clean_loans(con, test)
trans = cleaning.clean_trans(con, test)
disp = cleaning.pd_disp(con)
```

```
if test:
    loan_final.to_sql('loan_united_test', con=con, if_exists='replace')
else:
    loan_final.to_sql('loan_united_train', con=con, if_exists='replace')

con.close()
```

Data Cleaning

```
### Outlier removal using the z-score method
def remove_outliers_zscore(df,column,factor = 3):
    return df[(np.abs(stats.zscore(df[column])) < factor)]

### Outlier removel according to percentile
def remove_outliers_percentile(df,column):
    lower_bound = df[column].quantile(.95)
    upper_bound = df[column].quantile(.05)

    return df[(df[column] > lower_bound) & (df[column] < upper_bound)]
```

Outlier Removal

The remove outliers functions were not used since we did not discover any outlier

```
def replace_null(df):
    return df.fillna(df.median())

def replace_null_non_numeric(df,column):
    return df[column].fillna(df[column].value_counts().idxmax())

# %%
### Drop columns with percentage of nulls that surpasses the provided limit
def drop_null_columns(df,limit = 0.7):
    return df[df.columns[df.isnull().mean() < limit]]

### Drop rows with percentage of nulls that surpasses the provided limit
def drop_null_rows(df,limit = 0.5):
    return df.loc[df.isnull().mean() < limit]
```

Null Cleaning

Data Preparation

Feature Engineering - Dates

```
def convert_dates(df):
    copy = df.copy()
    columns = ["loan_date","account_creation","birth_number"]

    for column in columns:
        copy[column] = copy[column].apply(lambda x: datetime.datetime.strptime(x, '%d-%m-%Y').strftime('%Y')).astype(int)

    copy["age_on_loan"] = copy["loan_date"] - copy["birth_number"]
    copy = copy.drop(columns = ["loan_date","account_creation","birth_number"])

    copy['card_issued'] = pd.to_numeric(copy["card_issued"].astype(str), errors='coerce').fillna(1).astype(int)

    return copy
```

Date Conversion for only including years (year of birth)

```
### Convert numerical date to datetime formats
def date_conversion(df,column,dt_format = '%d-%m-%Y'):
    df_copy = df.copy()
    date = df_copy[column]
    date = date.astype(str)
    date = '19' + date
    df_copy[column] = date.apply(lambda x: (datetime.strptime(x,'%Y%m%d')).strftime(dt_format))

    return df_copy
```

Date Conversion

```
def date_conversion_genders(df,column,dt_format = '%d-%m-%Y'):
    df_copy = df.copy()
    date = df_copy[column]                                    # Get the actual column
    date = date.astype(str)                                   # Convert to string to add the century
    date = '19' + date                                        # Add the century value

    ### Get the monts value with gender
    lst = []
    for item in date:
        lst.append(item[4:6])

    months = []
    gender = []
    for item in lst:
        if(int(item) > 50):
            months.append(int(item)- 50)
            gender.append('female')
        else:
            months.append(int(item))
            gender.append('male')

    ### Replace the old month values
    for i in range(len(date)):
        if gender[i] == 'female':
            date[i] = str(int(date[i])-5000)

    df_copy[column] = date.apply(lambda x: (datetime.strptime(x,'%Y%m%d')).strftime(dt_format))
    df_copy["gender"] = gender

    return df_copy
```

Gender Date Conversion

Data Preparation

Feature Engineering

```
# Get the values for each column of the new dataframe
no_movements = trans_test.shape[0]
min_no_trans = min(pd.to_numeric(trans_test["amount"]))
max_no_trans = max(pd.to_numeric(trans_test["amount"]))
avg_no_trans = mean(pd.to_numeric(trans_test["amount"]))
min_balance = min(pd.to_numeric(trans_test["balance"]))
max_balance = max(pd.to_numeric(trans_test["balance"]))
avg_balance = mean(pd.to_numeric(trans_test["balance"]))
```

Minimum, maximum and average amount of a transaction and minimum, maximum and average balance of an account after a transaction

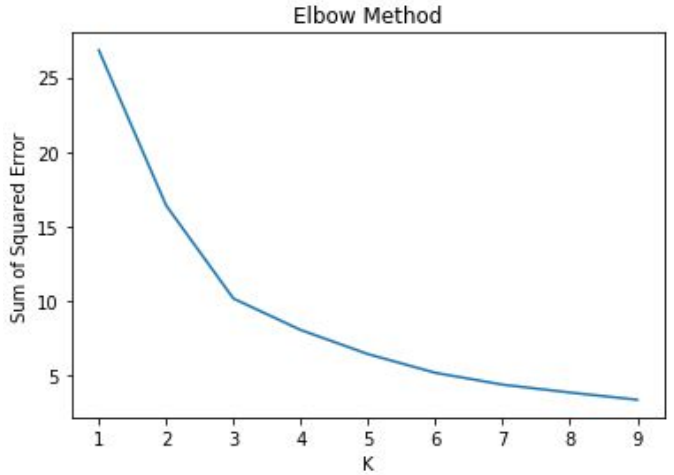
```
### Added columns for crime analysis
districts['crime_growth'] = pd.to_numeric(districts["no. of commited crimes '96"]) - pd.to_numeric(districts["no. of commited crimes '95"])
districts['total_crime'] = pd.to_numeric(districts["no. of commited crimes '96"]) + pd.to_numeric(districts["no. of commited crimes '95"])

### Added column for uneployment growth analysis
districts['unemploymant_growth'] = pd.to_numeric(districts["unemploymant rate '96"]) - pd.to_numeric(districts["unemploymant rate '95"])
```

Crime Growth and Total Crime (‘95 and ‘96)

Clustering

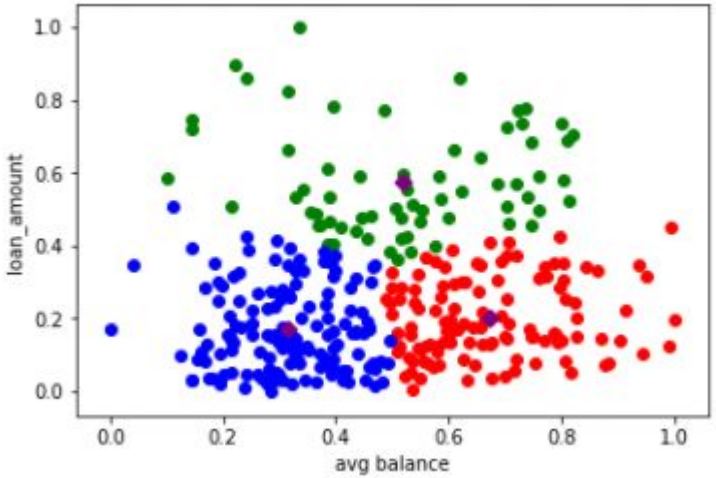
- *Sklearn* provides several clustering methods such as *Agglomerative* and *Mean-shift* clustering, but the chosen method was ***K-Means***, which separates samples in groups of equal variance and scales well to large number of samples
- To select the number of clusters, the *Elbow Method* heuristic was used. The method consists of plotting the sum of squared errors as a function of the number of clusters, and picking the elbow of the curve as the number of clusters to use.

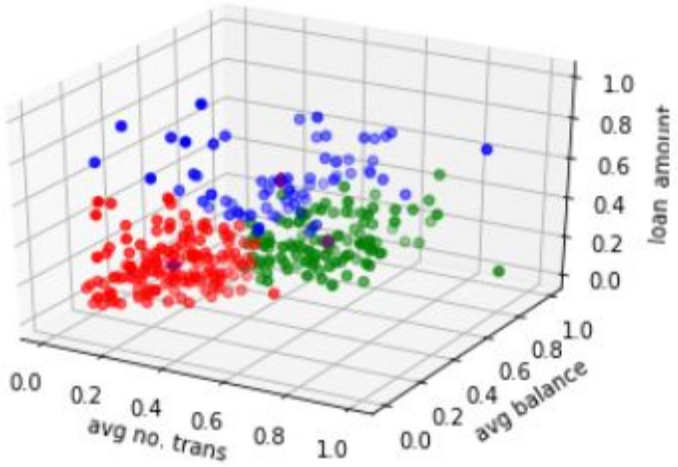
- In the Following example, the number of chosen clusters was 3



Clustering

- 2D and 3D graphs of clusters were plotted (2 or 3 selected features)
- Each cluster is represented by a coloured circle
- The Purple Diamonds represent the cluster centers





Data Transformation

- Transforming categorical columns to a single numeric column
- MinMax Scaler to convert numerical values between 0 and 1
- Create dummy columns to convert categorical columns into several boolean columns of each attribute

```
def category_encoding(df,column):

    copy = df.copy()
    encoder = preprocessing.LabelEncoder()
    encoder.fit(copy[column].unique())
    copy[column] = encoder.transform(copy[column])

    return copy
```

```
if DUMMIES:
    columns = ["account_frequency","gender","card_type"]
    df = add_dummy(df, columns)
```

```
if MIN_MAX_SCALER:
    scaler = MinMaxScaler()
    copy = df.copy()
    y = copy["status"]
    X = copy.drop(columns=["status"])
    transf = scaler.fit_transform(X)
    copy = pd.DataFrame(transf,index=X.index,columns=X.columns)
    copy["status"] = y
    df = copy
```

Feature Selection

- Since our final dataset contained at least 51 columns (57 with the DUMMIES flag activated) there was a need to select the most important features.
- The feature selection was done through the *SelectKBest* function from *sklearn*
- Several number of selected features were attempted, but the best results obtained were around 15 features.
- The method requires a score function. The chosen score function was *f_classif*, which selects attributes based on the variance of their group means

```
select = SelectKBest(f_classif, k=N_COLUMNS)
X_new = select.fit_transform(X, y)

split_filter = select.get_support()
features = X.columns[split_filter]
```

Data Splitting

Train_test_split

- This splitting method was attempted with several SPLIT_RATIOs, with the train size being approximately 80%.
- The method obtained better local results than *StratifiedKFold*, but had worse performances in the Kaggle Competition, due to overfitting

```
### Seperate the precition columns from output

X = df.drop(columns=['status'])
y = df['status']

select = SelectKBest(f_classif, k= N_COLUMNS)
X_new = select.fit_transform(X, y)

split_filter = select.get_support()
features = X.columns

### Apply splitting
X_train, X_test, y_train, y_test = train_test_split(X_new,y,train_size=SPLIT_RATIO,test_size=1-SPLIT_RATIO)

return X_train,X_test,y_train,y_test, features[split_filter]
```

Data Splitting

StratifiedKFold

- This method splits the data into K folds. The chosen value of K was 3, given that the dataset is small and larger K values would often create samples with no unsuccessful loans
- The method obtained was more consistent than *train_test_split* and caused less overfitting, thus, this method was chosen as the main splitting criteria

```
skf = StratifiedKFold(n_splits=N_SPLITS, random_state=True, shuffle=True)

for train_index, test_index in skf.split(X_new, y):

    X_train, X_test = X_new[train_index], X_new[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

Data Sampling

- SMOTE was used for oversampling
- The usage of *imblearn*'s pipelines allows the assembly of several steps to build our model

```
def build_pipeline(algorithm):

    if(OVERSAMPLE):
        return Pipeline([
            ('sampling',SMOTE()),
            ('classification',algorithm)
        ])
    else:
        return  Pipeline([
            ('classification',algorithm)
        ])
```

Grid Search

- The performance of each algorithm is heavily dependant on their provided parameters
- A *GridSearchCV* was used on every algorithm to find their best parameters on our dataset
- The search evaluates each combination of parameters based on the *AUC* curve of each attempt
- The same Pipeline used in the estimator is used in the the model training
- The Cross Validation (cv) method used is *stratifiedKFold*

```
grid_search = GridSearchCV(estimator = alg,
                           param_grid = grid,
                           scoring=metrics.make_scorer(utils.get_auc, greater_is_better=True),
                           cv=StratifiedKFold(3,random_state=True,shuffle=True),
                           n_jobs = -1,
                           verbose = 2)

model = grid_search.fit(X,y)
```

Performance Evaluation

- To evaluate algorithm performance, the AUC (Area Under the Curve) was used, given that it is the same metric used in the kaggle competition

```
def get_auc(y_test,y_predicted,label=-1):
    fpr, tpr, _ = metrics.roc_curve(pd.to_numeric(y_test), pd.to_numeric(y_predicted),pos_label=label)
    return metrics.auc(fpr, tpr)
```

- The *StratifiedKFold* method was used to split the data into 3 folds. The AUC curve of the model of each fold was calculated and the best one was selected

Run with StratifiedKFold

```
final_CV(algorithm)
```

```
AUC=0.6875
AUC=0.8184397163120568
AUC=0.724822695035461
Average AUC = 0.7435874704491727
Best AUC = 0.8184397163120568
```

Training an algorithm

- Choose an algorithm
- Get the Training Data
- Select Best Features of the dataset (15 features)
- Apply *StratifiedKfold* splitting (3 splits)
- Train the Model
- Predict the outcome with the test data and calculate the *AUC* curve
- Store the best model out of the provided splits

```
def final_CV(algorithm):
    train = get_df()

    X = train.drop(columns=['status'])
    y = train['status']

    select = SelectKBest(f_classif, k=N_COLUMNS)
    X_new = select.fit_transform(X, y)

    split_filter = select.get_support()
    features = X.columns[split_filter]

    skf = StratifiedKFold(n_splits=N_SPLITS, random_state=True, shuffle=True)

    model_list = []
    auc_list = []

    for train_index, test_index in skf.split(X_new, y):

        X_train, X_test = X_new[train_index], X_new[test_index]
        y_train, y_test = y[train_index], y[test_index]

        ### Train the model
        pipe = build_pipeline(algorithm())
        model = pipe.fit(X_train, y_train)

        ### Predict the outcome with the test data
        y_pred = model.predict_proba(X_test)
        y_final = y_pred.transpose()[0]

        auc = utils.get_auc(y_test, y_final)
        auc_list.append(auc)
        model_list.append(model)
        print(f"AUC={auc}")

    ### Get the best model

    best_score = max(auc_list)
```

Testing an algorithm

- Get the testing data
- Select the previously obtained best model from the training phase
- Make a prediction
- Save the predicted values in a dataframe

```
### Use the best model to get a prediction
test = get_df(test=True)

X2 = test.drop(columns=['status'])
X2 = X2[features]

y_predicted = best_model.predict_proba(X2)
y_final = y_predicted.transpose()[0]

final_df = pd.DataFrame()
final_df['Id'] = test["loan_id"]
final_df['Predicted'] = y_final
```

Algorithms

Random Forest

- This algorithm combines several decision trees, therefore sharing it's advantages
- Slower than Decision trees and requires rigorous training
- More Robust to overfit than Decision Trees
- Dummy columns were created to convert categorical data to several numerical columns

```
def get_random_forest():
    return RandomForestClassifier(bootstrap = False,
                                  max_depth =110,
                                  max_features = 2,
                                  min_samples_leaf = 3,
                                  min_samples_split = 8,
                                  n_estimators = 300)
```

```
param_grid = {
    'classification__max_depth': [80, 90, 100, 110],
    'classification__max_features': [2, 3],
    'classification__min_samples_leaf': [3, 4, 5],
    'classification__min_samples_split': [8, 10, 12],
    'classification__n_estimators': [100, 200, 300, 1000]
}

getBestSearch("RF",param_grid)
```

```
(0.6018347209374036,
 {'classification__class_weight': 'balanced',
  'classification__criterion': 'gini',
  'classification__max_depth': 30,
  'classification__max_features': 'log2',
  'classification__min_impurity_decrease': 0.05,
  'classification__min_samples_leaf': 6,
  'classification__min_samples_split': 6,
  'classification__splitter': 'best'})
```

Algorithms

Logistic Regression

- Works with small datasets such as the one provided
- Tends to overfit, which led to better results in the local score, but worse in the Kaggle competition
- Works better while using *MinMaxScaler*

```
def get_logistic_regression():
    return LogisticRegression(random_state=30,
                              solver='sag',
                              max_iter=400,
                              dual=False,
                              multi_class='auto',
                              penalty='none',
                              tol=0.1)
```

```
param_grid = {
    'classification__penalty': ['none', 'l1', 'l2', 'elasticnet'],
    'classification__dual': [True, False],
    'classification__tol': [0.1, 0.01, 0.001, 1e-4, 1e-5, 1e-6],
    'classification__random_state': [8, 10, 12, 20, 30, 50],
    'classification__solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'],
    'classification__max_iter': [100, 200, 300, 400],
    'classification__multi_class': ['auto', 'ovr', 'multinomial']
    }

getBestSearch("LR",param_grid)
```

```
(0.5750077089115018,
 {'classification__dual': False,
  'classification__max_iter': 400,
  'classification__multi_class': 'multinomial',
  'classification__penalty': 'l1',
  'classification__random_state': 30,
  'classification__solver': 'saga',
  'classification__tol': 0.1})
```

Algorithms

K-Nearest Neighbors

- Usually simple and efficient
- Not efficient for small sets of data, which presented itself as a limitation in this project

```
def get_knn():
    return KNeighborsClassifier(algorithm="ball_tree",
                                leaf_size=150,
                                metric="chebyshev",
                                n_neighbors=10,
                                p=3,
                                weights="distance")
```

```
param_grid = {
    'classification__n_neighbors': [2, 3, 4, 5, 8, 10],
    'classification__algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'classification__weights': ['uniform', 'distance'],
    'classification__metric': ['euclidean', 'manhattan', 'chebyshev',
        'minkowski', 'wminkowski', 'seuclidean', 'mahalanobis', 'haversine',
        'hamming', 'canberra', 'braycurtis', 'cityblock', 'infinity',
        'l1', 'l2', 'p'],
    'classification__leaf_size': [10, 20, 30, 50, 80, 150],
    'classification__p': [1, 2, 3, 5, 10]
}

getBestSearch("KNN",param_grid)
```

```
(0.5868023435090965,
 {'classification__algorithm': 'ball_tree',
  'classification__leaf_size': 150,
  'classification__metric': 'chebyshev',
  'classification__n_neighbors': 10,
  'classification__p': 3,
  'classification__weights': 'distance'})
```

Algorithms

Decision Tree

- Fast to implement
- Not accurate at predicting the results, and tends to overfit
- Dummy columns were created to convert categorical data to several numerical columns

```
def get_decision_tree():
    return DecisionTreeClassifier(class_weight='balanced',
                                  criterion='gini',
                                  max_depth=30,
                                  max_features='log2',
                                  min_impurity_decrease=0.05,
                                  min_samples_leaf=6,
                                  min_samples_split=6,
                                  splitter='best')
```

```
param_grid = {
    'classification__criterion':['gini', 'entropy'],
    'classification__splitter': ["best", "random"],
    'classification__max_depth': [5, 10, 20, 30, 40],
    'classification__min_samples_split': [2, 4, 6, 8],
    'classification__min_samples_leaf': [1, 2, 4, 6],
    'classification__max_features': ["auto", "sqrt", "log2"],
    'classification__min_impurity_decrease': [0.05, 0.1, 0.2, 0.3],
    'classification__class_weight': ["balanced", None]
}

getBestSearch("DT",param_grid)
```

```
(0.6018347209374036,
 {'classification__class_weight': 'balanced',
  'classification__criterion': 'gini',
  'classification__max_depth': 30,
  'classification__max_features': 'log2',
  'classification__min_impurity_decrease': 0.05,
  'classification__min_samples_leaf': 6,
  'classification__min_samples_split': 6,
  'classification__splitter': 'best'})
```

Result Analysis

Local Score

Features	15		20		25		All	
Oversample	Yes	No	Yes	No	Yes	No	Yes	No
Random Forest	*0.823*	0.796	0.804	0.793	0.778	0.804	0.713	0.741
Logistic Regression	***0.841***	0.815	0.803	0.816	0.767	0.777	0.618	0.588
Decision Tree	*0.725*	0.692	0.705	0.656	0.694	0.661	0.65	0.611
KNN	0.758	0.687	0.814	0.764	*0.838*	0.764	0.754	0.698

Result Analysis

Conclusions

- **Random Forest** obtained the more consistent results locally (average of 0.782)
- **Logistic Regression** got the best result when using oversampling and 15 features (0.841), locally
- The worst algorithm is the **Decision Tree**, obtaining an average of score of 0.674, locally
- When we use **15 features**, we get the more consistent results. When using less than 15, the model tends to overfit and when using more than 15, the AUC starts getting lower
- **Oversampling** generally gets the best results

Used Tools

- **Python**: Development of the Pipeline. Most important libraries:
  - ***pandas***: data manipulation
  - ***numpy***: numerical manipulation of data
  - ***sklearn***: prediction algorithms, cross validation, hyper parameter tuning, data normalization…
  - ***sqlite3***: database where the data is stored (more efficient)
  - ***matplotlib***, ***seaborn***: data plotting
- **Github/Github Project**: maintenance of the project and planning of issues that needed to be resolved
- **Rapidminer**: initial visualization and study of the data provided
- **Discord**: group meetings and share of useful materials
- **Excel**: visualization of the raw data provided

Used Tools

Github Projects