Predictive Data Mining Task

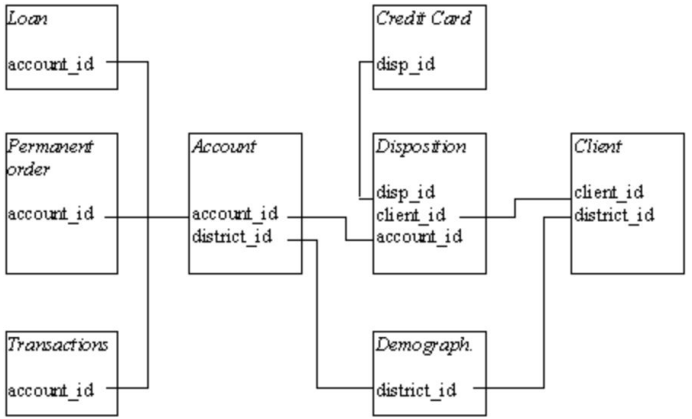
Class 1, Group 10:
Edgar Carneiro - *up201503784*
João Damas - *up201504088*

Course:
Knowledge Extraction and Machine Learning @FEUP

Domain Description

Dataset from the **records of a Czech bank** - as seen by regions in the demographic table -, dating from **1993 up to 1998**. Provided as set of *.csv* files, where each file represents a table of the displayed schema.

- **4500 accounts** that can be accessed / owned by more than one client.
- Demographic data for both the bank sucursal and the clients, constituted from **77 Czech districts**.
- **426 888 transactions**, including payments, cash credits and withdrawals, pensions, households, transactions from other banks, among others.
- **682 loans**, from which 329 the loan status was known.

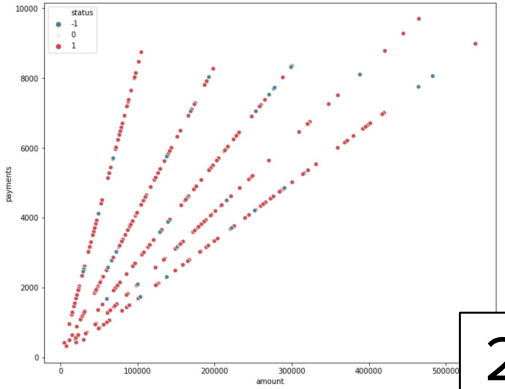


Exploratory Data Analysis

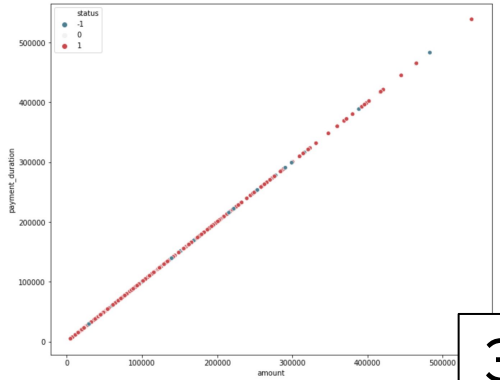
Results found:

1. On the labelled loans: 86% ended successfully (**1**) while 14% did not (**-1**). Hence, the status class is an unbalanced class.
2. Correlation between loan amount, payments and duration. (*amount = duration * payment*)
3. No interest rate from the bank reflected in the payments.
4. Correlation between large amounts loaned and the loan status being negative.
5. No User has more than one loan.
6. No correlation between gender and loan status.
7. The age distribution follows the expected pattern, with the majority of the loans being requested between 30 and 40 years old.
8. Districts with more inhabitants and higher ratio of urban inhabitants have higher average salaries.
9. 70% of the users have the Classic credit card type.
10. Transactions are always previous to loan.

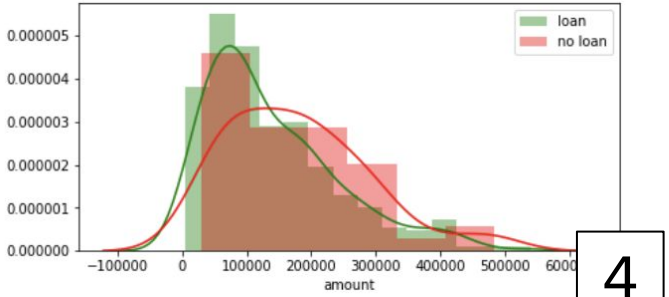
Referring Plots to previous slide



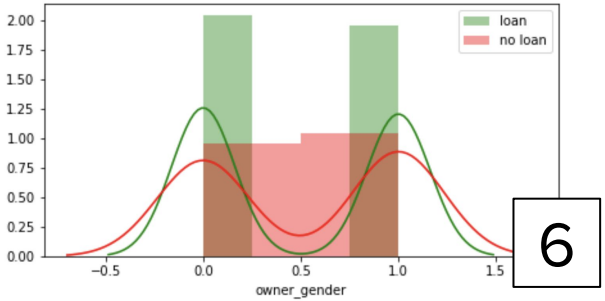
2



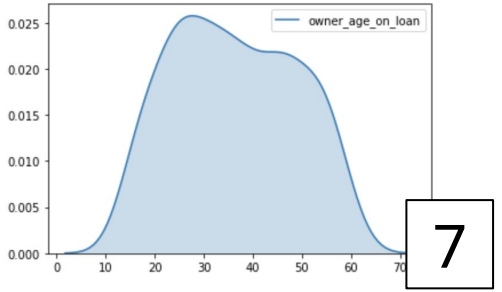
3



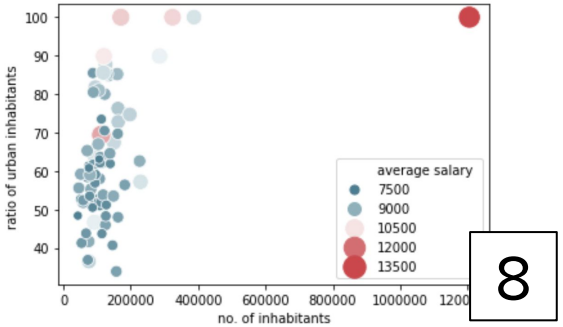
4



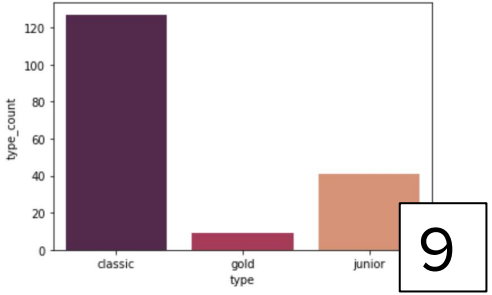
6



7



8



9

Problem Definition

Objective: Predict whether a loan will end successfully.

Nature: Classification problem were output variable is **1** or **-1**, with 1 being loan ends successfully and -1 the opposite.

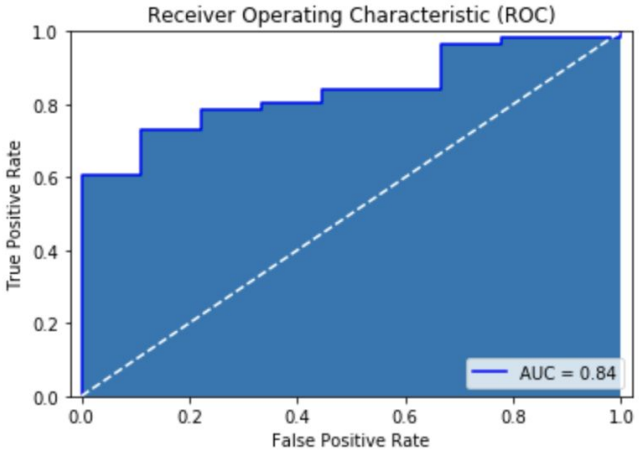
Evaluation metric: Area Under ROC Curve (**AUC**). The **ROC** (Receiver Operating Characteristic) curve plots **TPR** (True Positive Rate) vs **FPR** (False Positive Rate) at different classification thresholds. Since the output class is unbalanced (14% N - 86% P), the impact of False Positives is bigger than that of False Negatives, which is expected considering the project's context. For the bank, it's worse considering bad loans good as they will lose money, whilst considering good loans bad they only lose an opportunity.

True Positive Rate (TPR)

$$TPR = \frac{TP}{TP + FN}$$

False Positive Rate (FPR)

$$FPR = \frac{FP}{FP + TN}$$

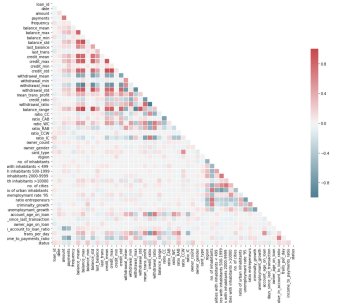


Data Preparation

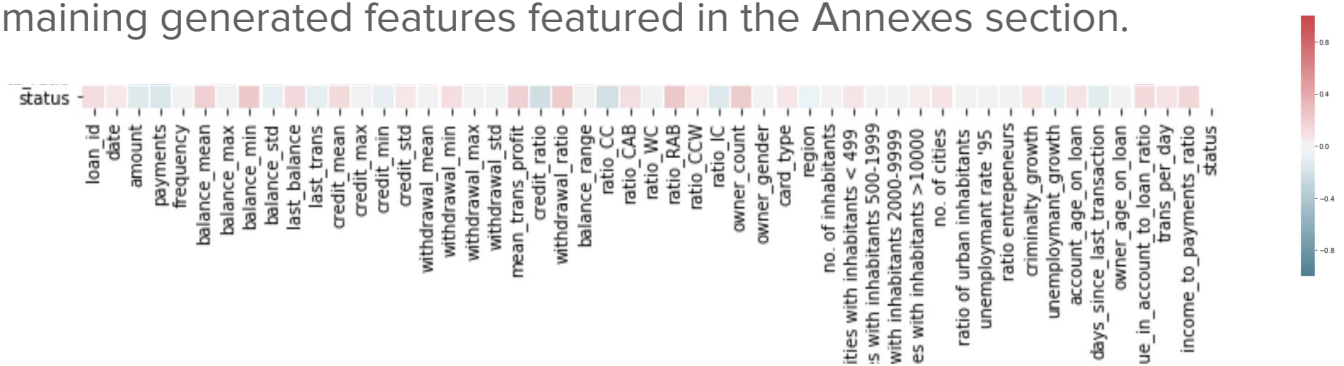
- Extract client birthdate and gender from *birth_number*
- Transform dates to format *yyyymmdd*
- Replace ‘*?*’ entries in both *no. of committed crimes \'95* and *unemployment rate \'95* with median values.
- Encoding of categorical classes such as credit card type, district name and region, gender, dates etc.
- Conversion of categorical variables into dummy/indicator variables, in transaction operations.
- Missing table permanent order.
- Detection of outliers (none detected).
- Remotion of features (**feature selection**) having no impact or mainly constituted by nulls
    - For example: bank and account columns in the transaction table (15% filled).
- Generation of new features (see next slide).

Feature Engineering

Generated features:



- **Statistic metrics** - max, min, mean, ... - for many features, such as balance, credits, etc.
- Using **logarithmic transformation** and data **discretization techniques.**
- Criminal growth and unemployment growth
- Owner age on loan date and account age on loan date
- Account ownership and expected **income to payments ratio**.
- Over 60 features were generated.
- Remaining generated features featured in the Annexes section.



Experimental Setup

Project as a pipeline of two steps:

- Preprocessing (seen in previous slides).
- Prediction:
  - Normalization of necessary columns.
  - Apply **undersampling**, **oversampling** or **both**, according to respective control flags.
  - Apply **Principal Component Analysis**, if the PCA flag is *True*.
  - Using **stratified cross validation** (80/20) split in train and test data.
  - Apply **grid search** on the chosen algorithms, for **hyper parameter tuning** - *Logistic Regression, Decision Tree, Random Forest & Gradient Boost* - find the one maximising AUC.
  - **Apply the best performing algorithm** and respective configuration and get predictions.

```
{   'bootstrap': [True, False],
    'class_weight': ['balanced', 'balanced_subsample', None],
    'criterion': ['gini', 'entropy'],
    'max_depth': [2, 6, 10, 14, None],
    'max_features': ['auto', 'sqrt'],
    'min_samples_leaf': [1, 2, 4],
    'min_samples_split': [2, 5, 10],
    'n_estimators': [2, 4, 6, 8, 10, 12]}
Fitting 5 folds for each of 6480 candidates, totalling 32400 fits

[Parallel(n_jobs=-1)]: Done 158 tasks      | elapsed:    1.6s
[Parallel(n_jobs=-1)]: Done 884 tasks      | elapsed:    8.2s
[Parallel(n_jobs=-1)]: Done 2102 tasks      | elapsed:   22.8s
[Parallel(n_jobs=-1)]: Done 3800 tasks      | elapsed:   44.9s
[Parallel(n_jobs=-1)]: Done 5990 tasks      | elapsed:  1.3min
[Parallel(n_jobs=-1)]: Done 8660 tasks      | elapsed:  1.7min
[Parallel(n_jobs=-1)]: Done 11822 tasks      | elapsed:  2.3min
[Parallel(n_jobs=-1)]: Done 15464 tasks      | elapsed:  2.9min
[Parallel(n_jobs=-1)]: Done 19598 tasks      | elapsed:  3.7min
[Parallel(n_jobs=-1)]: Done 24212 tasks      | elapsed:  4.7min
[Parallel(n_jobs=-1)]: Done 29318 tasks      | elapsed:  6.0min
[Parallel(n_jobs=-1)]: Done 32400 out of 32400 | elapsed:  6.6min finished
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default of the `iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This will change numeric results when test-set sizes are unequal.
  DeprecationWarning)
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/model_selection/_search.py:715: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using ravel().
  self.best_estimator_.fit(X, y, **fit_params)
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Best Score:  0.8432211435601199
Best Params:  {'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 2, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 10, 'n_estimators': 10}
```

Example grid Search output on Random Forest

Tools used: **Python**, namely Jupyter notebooks with scikit-learn, pandas, numpy, matplotlib and seaborn; Excel.

Results

- Locally getting a consistent score of 84% when using **Random Forest**.
- Performing slightly better on the competition - approx. 88%.
- **PCA decreases** all algorithms **performance** both locally and in kaggle (explanation in Annexes).
- Sampling techniques such as **oversampling** and a **dual approach** - both undersampling and oversampling - seem to increase performance.
- Seeing that we are working with temporal data, the behaviors might change over the train years and the test years, thus take our apparently good performing model with a pinch of salt.
- Major concern of having a too big of an error interval or overfitting, thus having the possibility of a much lower score in the private competition.
- A more thorough analysis and conclusions are presented in the Annexes.

```
1. Random Forest - 0.832330
---------
2. Gradient Boosting - 0.808303
---------
3. Decision Tree - 0.806183
---------
4. Logistic Regression - 0.785563
---------
Best algorithm: Random Forest
```

Ranking of used algorithms, without PCA

```
1. Logistic Regression - 0.768239
---------
2. Gradient Boosting - 0.695331
---------
3. Decision Tree - 0.689463
---------
4. Random Forest - 0.689277
---------
Best algorithm: Logistic Regression
```

Ranking of used algorithms, with PCA

Conclusions, Limitations and Future Work

Conclusions:

- Feature engineering and feature selection are, in our opinions, the most important and most difficult steps of the process.
- Learning algorithms such as the Gradient Boosting had bad performance, probably because there are low amounts of data, whilst Decision Tree based algorithms had the best performances.

Limitations:

- Reduced quantities of data, combined with the existence of many features (even after some selection), making it infeasible to apply deep learning algorithms (dimensionality issues).

Future Work:

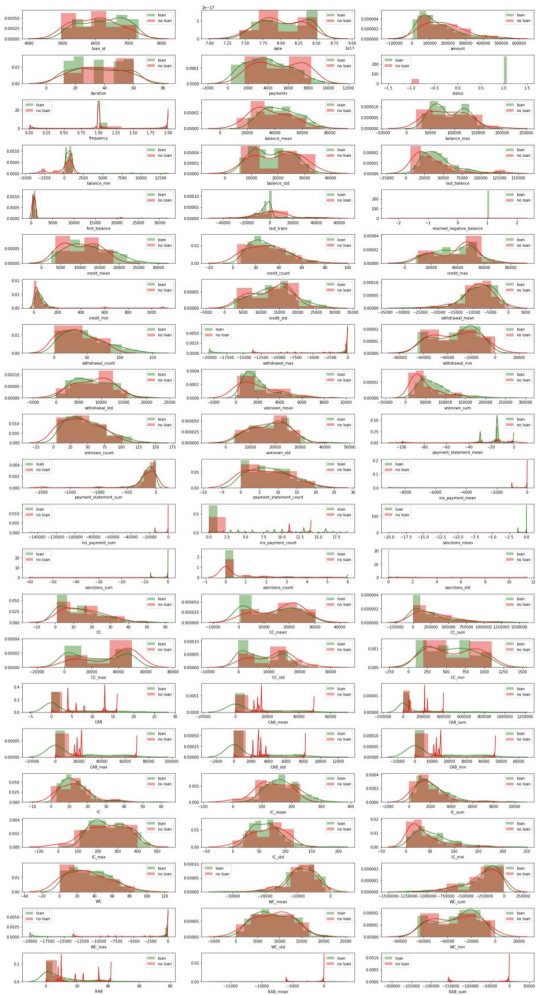
- Improve features being used, by feature engineering and feature selection (**key factor**)

Annexes

Data Analysis

For ensuring a constant analysis of the totality of the process, as well as visual representation of the data features, there is a set of **debug flags** across the entire Pipeline. Some of the automatic plots displayed, when the debug flag is set to True, are as follows:

Correlation matrix of all features



Plotting features histogram to understand impact on Y

Data Cleaning

Useful generic functions for data cleaning:

Imputation

```
def get_null_summary(dataset):
    '''Get a null summary display'''
    display(dataset.isnull().mean())
```

```
def clean_nulls(dataset, threshold=0.7):
    '''Clean nulls from the given table.
    If the nulls in a column are higher than the given threshold the entire column is deleted.
    If the nulls in a row are higher than the row, the row is also deleted.
    The threshold is a value between 0 and 1'''
    #Dropping columns with missing value rate higher than threshold
    dataset = dataset[dataset.columns[dataset.isnull().mean() < threshold]]

    #Dropping rows with missing value rate higher than threshold
    dataset = dataset.loc[dataset.isnull().mean(axis=1) < threshold]

    return dataset
```

```
def numerical_imputation(dataset, replacer=None):
    '''When null values exist, set them using the median of the colum,
    or a replacer, if one was given'''
    dataset = dataset.fillna(replacer if replacer is not None else dataset.median())

    return dataset
```

```
def categorical_imputation(dataset, column_name, replacer=None):
    '''Replace the inexistent values of the given column with the given replacer.
    If None replacer was ginve, use the column maximum value'''
    #Max fill function for categorical columns
    dataset[column_name].fillna(replacer if replacer is not None else \
                                dataset[column_name].value_counts()
                                                    .idxmax(),
                                inplace=True)

    return dataset
```

Useful functions for preprocessing

```
def convert_date(df, column, date_format='%y%m%d'):
    '''Convert the given column containg dates in the given format
    to the standard date format and type'''
    copy_df = df.copy()
    copy_df[column] = pd.to_datetime(copy_df[column], format=date_format)

    return copy_df
```

```
def encode_column(df, column):
    '''Encode the given column of the given dataframe.'''
    copy_df = df.copy()

    le = preprocessing.LabelEncoder()
    le.fit(df[column].unique())
    copy_df[column] = le.transform(copy_df[column])

    return copy_df
```

Handling Outliers

```
def display_to_drop_std(dataset, column, mult_factor=3):
    '''Display the rows that will be dropped using the std approach'''
    upper_lim = dataset[column].mean() + dataset[column].std() * mult_factor
    lower_lim = dataset[column].mean() - dataset[column].std() * mult_factor

    display(dataset[(dataset[column] >= upper_lim) & (dataset[column] <= lower_lim)])

def drop_outliers_std(dataset, column, mult_factor=3):
    '''Drop the outlier rows with standard deviation'''
    upper_lim = dataset[column].mean() + dataset[column].std() * mult_factor
    lower_lim = dataset[column].mean() - dataset[column].std() * mult_factor

    return dataset[(dataset[column] < upper_lim) & (dataset[column] > lower_lim)]
```

```
def display_to_drop_percentile(dataset, column):
    '''Display the rows that will be dropped with Percentiles approach'''
    upper_lim = dataset[column].quantile(.95)
    lower_lim = dataset[column].quantile(.05)

    display(dataset[(dataset[column] >= upper_lim) & (dataset[column] <= lower_lim)])

def drop_outliers_percentile(dataset, column):
    '''Drop the outlier rows with Percentiles approach'''
    upper_lim = dataset[column].quantile(.95)
    lower_lim = dataset[column].quantile(.05)

    data = dataset[(dataset[column] < upper_lim) & (dataset[column] > lower_lim)]
```

Data Cleaning

For preparing each of the tables representing the data, we chose to write a pre-processing function that would cleanse the data referent to each table:

- *process_loans()*
- *process_account()*
- *process_disposition()*
- *process_card()*
- *process_client()*
- *process_transactions()*
- *process_demographic()*

```python
def process_client(client_df, debug=False):
    '''Pre process the client table'''
    if debug:
        print(' > Raw client table representation')
        display(client_df)
        print(' > Raw client table correlations')
        get_df_correlation(client_df)

    processed_df = client_df.copy()

    # Getting year, day, and month+50 if women
    processed_df['year'] = 1900 + (processed_df['birth_number'] // 10000)
    processed_df['month_gender'] = (processed_df['birth_number'] % 10000) // 100
    processed_df['day'] = processed_df['birth_number'] % 100

    # Extracting gender and month
    processed_df['gender'] = np.where(processed_df['month_gender']>=50, 1, 0)
    processed_df['month'] = np.where(processed_df['month_gender']>=50, processed_df['month_gender']-50, processed_df

    # Composing data
    processed_df['birth_date'] = processed_df['year'] * 10000 +\
                                 processed_df['month'] * 100 +\
                                 processed_df['day']
    df = convert_date(processed_df, 'birth_date', '%Y%m%d')

    # Dropping useless columns
    df = df.drop(['birth_number', 'year', 'month_gender', 'month', 'day'], axis=1)

    if debug:
        print(' > After extracting the gender from the date we have:')
        display(df)
        print(' > Notice the gender representation:\n\t * 1 if female\n\t * 0 if male')
        get_df_correlation(df)

    return df
```

Data Transformation (semi-automatic)

- Usage of **statistical metrics** to aggregate data, such as:
  - Count of entries
  - Mean value
  - Standard deviation value
  - Maximum value
  - Minimum value
  - First value
  - Last value

```
# Aggregatting transaction balances
agg_ballance = processed_df.sort_values(by=['account_id', 'date'],
                                        ascending=[True, False])\
                           .groupby(['account_id'])\
                           .agg({
                                'balance': ['mean', 'max', 'min', 'std', get_first, get_last],
                                'date': get_first,
                                'amount': get_first
                           })\
                           .reset_index()
agg_ballance.columns = ['account_id', 'balance_mean', 'balance_max', 'balance_min', 'balance_std',
                        'last_balance', 'first_balance', 'last_trans_date', 'last_trans']
```

Example of aggregation of balances

Statistical Analysis

```
def get_col_max(df, col):
    '''Get the maximum value of a given column'''
    return df[col].max()

def get_col_min(df, col):
    '''Get the minimum value of a given column'''
    return df[col].min()

def get_col_count(df, col):
    '''Get the number of elements of a given column'''
    return df[col].count()

def get_col_avg(df, col):
    '''Get the average value of a given column'''
    return df[col].mean()

def get_col_std(df, col):
    '''Get the standard deviation value of a given column'''
    return df[col].std()

def get_first(df):
    '''Get the first entry of a dataframe'''
    return df.iloc[0]

def get_last(df):
    '''Get the last entry of a dataframe'''
    return df.iloc[-1]
```

Data Transformation (semi-automatic)

- Usage of **other methodologies**, such as:
  - Log Transform function
  - Data discretization

Log Transform

```
def log_transform(series):
    '''Get a log_transformation of the given series'''
    min_val = series.min()
    return (series - min_val + 1).transform(np.log)
```

```
agg_ballance['reached_negative_balance'] = agg_ballance['balance_min']
agg_ballance.loc[agg_ballance["balance_min"] >= 0, "reached_negative_balance"] = 1
agg_ballance.loc[agg_ballance["balance_min"] < 0, "reached_negative_balance"] = -1
```

Example of data discretization utilising minimum balance in account feature:
Find if an User ever reached negative balance

Feature Engineering

```
# Adding extra columns
df['mean_trans_profit'] = df['credit_mean'] + df['withdrawal_mean']
df['total_ops'] = df['CC'] + df['CAB'] + df['WC'] + df['RAB'] + df['CCW'] + df['IC']
df['credit_ratio'] = df['credit_count'] / df['total_ops']
df['withdrawal_ratio'] = df['withdrawal_count'] / df['total_ops']
df['balance_range'] = df['balance_max'] - df['balance_min']
df['last_first_balance_ratio'] = df['last_balance'] / df['first_balance']
df['last_max_balance_ratio'] = df['last_balance'] / df['balance_max']

# OPs as ratios
df['ratio_CC'] = df['CC'] / df['total_ops']
df['ratio_CAB'] = df['CAB'] / df['total_ops']
df['ratio_WC'] = df['WC'] / df['total_ops']
df['ratio_RAB'] = df['RAB'] / df['total_ops']
df['ratio_CCW'] = df['CCW'] / df['total_ops']
df['ratio_IC'] = df['IC'] / df['total_ops']
```

Feature Engineering using exclusively transactions table features

```
# Now lets create new features:
df['ratio entrepeneurs'] = df['no. of enterpreneurs per 1000 inhabitants '] / 1000
df['ratio of urban inhabitants '] = df['ratio of urban inhabitants '] / 100

# Growths
df['criminality_growth'] = (df['no. of commited crimes \'96 '] - df['no. of commited crimes \'95 ']) /\
                           df['no. of inhabitants']
df['unemploymant_growth'] = df['unemploymant rate \'96 '] - df['unemploymant rate \'95 ']
```

Feature Engineering using exclusively demographic table features

Feature Engineering

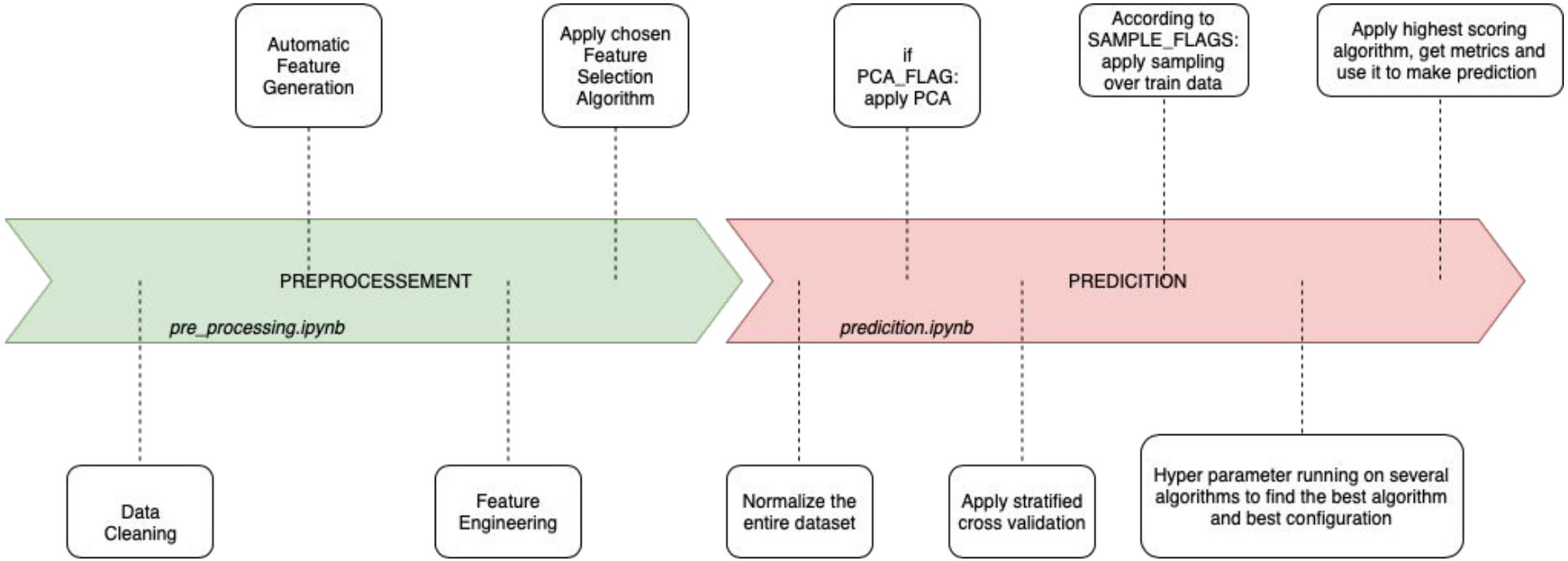
```
# Creating new columns using previous ones
df['account_age_on_loan'] = (df['date'] - df['account_creation_date']).dt.days
df['days_since_last_transaction'] = (df['date'] - df['last_trans_date']).dt.days
df['owner_age_on_loan'] = (df['date'] - df['owner_birthdate']).dt.days / 365
df["max_value_in_account_to_loan_ratio"] = df["balance_max"] / df["amount"]
df["last_value_in_account_to_loan_ratio"] = df["last_balance"] / df["amount"]
df["date"] = df["date"].astype(int)

# Stats per month
df['account_age_months'] = df['account_age_on_loan'] / 30
df["trans_per_month"] = df['total_ops'] / df["account_age_months"]
df['credit_per_month'] = df['credit_mean'] / df["account_age_months"]
df['withdrawal_per_month'] = df['withdrawal_mean'] / df["account_age_months"]
df['salary_from_month_records'] = df['credit_mean'] + df['withdrawal_mean']
df['withdrawal_to_credit_month_ratio'] = df['withdrawal_per_month'] / df['credit_per_month']

# Average salary, pension and household are anual- household value is negative
df['expected_month_income'] = (df['average salary '] + df['household']) / 12
df['actual_month_income'] = df['salary_from_month_records'] + (df['household'] / 12)
df.loc[df["pension"] > 0, "expected_month_income"] = (df["pension"] + df['household']) / 12
df.loc[df["pension"] > 0, "actual_month_income"] = df["actual_month_income"] + (df['pension'] / 12)
df['expected_income_to_payments_ratio'] = df['expected_month_income'] / df['payments']
df['actual_income_to_payments_ratio'] = df['actual_month_income'] / df['payments']
df['ratio_actual_salary_to_expected'] = df['actual_month_income'] / df['expected_month_income']
```

Feature Engineering using features from all tables

Experimental Setup - Pipeline visualization



Experimental Setup - Sampling

- For **oversampling**, we use *SMOTE oversampling*.
- For **undersampling**, we use *NeighbourhoodCleaningRule*.
- For both **oversampling & undersampling**, we use *SMOTETEEN* (seeing that in general it cleans more noisy data than *SMOTETomek*).
- Using *imblearn*'s Pipelines, we guarantee that sampling is only applied to train data.

```
def apply_sampling(algorithm, oversample, undersample):
    '''Apply sampling according to the control flags'''
    # Applying sampling techniques
    pipeline = Pipeline([
        ('classification', algorithm)
    ])

    if oversample:
        if undersample:
            pipeline = Pipeline([
                ('sampling', SMOTETomek(random_state = SEED)),
                ('classification', algorithm)
            ])
        else:
            pipeline = Pipeline([
                ('sampling', SMOTE(random_state = SEED)),
                ('classification', algorithm)
            ])
    elif undersample:
        pipeline = Pipeline([
        ('sampling', NeighbourhoodCleaningRule(random_state = SEED)),
        ('classification', algorithm)
    ])

    return pipeline
```

Experimental Setup - Principal Component Analysis

When running PCA, we first apply the PCA algorithm to the train data to obtain a **PCA model** [1]. Then, we use that same PCA model to transform the data we intend to predict [2].

```python
def apply_PCA(df, variance_val=0.95, debug=True):
    '''Apply the PCA algorithm to given dataframe,
    using the given variance val to trim the df'''
    # Necessary to normalize all data to use PCA
    scaler=StandardScaler()
    X_scaled=scaler.fit_transform(df)

    # PCA - keep, by default mode, 90% variance
    pca = PCA(variance_val)
    pca.fit(X_scaled)
    X_pca = pca.transform(X_scaled)

    if debug:
        ex_variance=np.var(X_pca,axis=0)
        ex_variance_ratio = ex_variance/np.sum(ex_variance)
        print(' > Impact in total variance of each generated feature by PCA:')
        print(ex_variance_ratio)

    principal_df = pd.DataFrame(data = X_pca, index = df.reset_index()['loan_id'])

    return (principal_df, pca)
```

1

```python
if USE_PCA:
    # Using train PCA and classifying
    scaler=StandardScaler()
    X_test_scaled=scaler.fit_transform(test_dataset)
    predictions_df = pd.DataFrame(data = pca.transform(X_test_scaled),
                                  index=test_dataset.reset_index()['loan_id'])
    display(predictions_df)

    predictions_df['Predicted'] = classifier.predict(predictions_df)
    final_df = predictions_df.reset_index()\
                    [['loan_id', 'Predicted']]\
                    .rename(columns={
                        'loan_id': 'Id'
                    })
```

2

Algorithms - Logistic Regression

```
def create_LR():
    '''Create a Logistic Regression model'''
    return LogisticRegression(random_state=SEED)
```

- Fast algorithm.
- Most **simple algorithm** in our Pipeline.
- Since the our output is binary, it is an adequate algorithm.
- Used as the **benchmark algorithm**, seeing that it is the most simple of the chosen algorithms.

```
def getLogisticRegressionBest(X, y, debug=True):
    '''Get the Logistic Regression Hyper Parameters'''

    # Maximum number of levels in tree
    max_depth = [int(x) for x in range(2, 20, 4)]
    max_depth.append(None)

    # Create the random grid
    grid = {'classification__penalty': ['l2', 'none'],
            'classification__C': [0.01, 0.05, 0.1, 0.2, 0.5, 1.0],
            'classification__solver': ['newton-cg', 'lbfgs', 'sag', 'saga'],
            'classification__class_weight': ["balanced", None]}

    if debug:
        pp.pprint(grid)

    # Applying sampling techniques
    lr = apply_sampling(create_LR(), OVERSAMPLE, UNDERSAMPLE)

    # Using the grid search for best hyperparameters
    lr_grid = GridSearchCV(estimator = lr,
                           param_grid = grid,
                           scoring=metrics.make_scorer(auc_scorer,
                                                       greater_is_better=True),
                           cv=StratifiedKFold(K_FOLD_NUM_SPLITS,
                                              random_state=SEED,
                                              shuffle=True),
                           verbose=2,
                           n_jobs = -1)

    # Fit the grid search model
    lr_grid = lr_grid.fit(X, y)

    if debug:
        print('Best Score: ', lr_grid.best_score_)
        print('Best Params: ', lr_grid.best_params_)

    # Return score, method & params tuple
    return (lr_grid.best_score_, 'Logistic Regression', lr_grid.best_params_)
```

Algorithms - Logistic Regression

```
{   'classification__C': [0.01, 0.05, 0.1, 0.2, 0.5, 1.0],
    'classification__class_weight': ['balanced', None],
    'classification__penalty': ['l2', 'none'],
    'classification__solver': ['newton-cg', 'lbfgs', 'sag', 'saga']}
Fitting 5 folds for each of 96 candidates, totalling 480 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  39 tasks      | elapsed:    2.4s
[Parallel(n_jobs=-1)]: Done 480 out of 480 | elapsed:    8.5s finished
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default of the `iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This will change numeric results when test-set sizes are unequal.
  DeprecationWarning)
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:1510: UserWarning: Setting penalty='none' will ignore the C and l1_ratio parameters
  "Setting penalty='none' will ignore the C and l1_ratio "
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/utils/validation.py:724: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/linear_model/sag.py:337: ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
```

```
Best Score:  0.8102752434949161
Best Params:  {'classification__C': 0.01, 'classification__class_weight': 'balanced', 'classification__penalty': 'none', 'classification__solver': 'saga'}
```

Grid searching hyper parameters for Logistic Regression output example

Algorithms - Decision Tree

- Easy to understand and great for **visual analysis.**
- **Feature selection** happens **automatically**: unimportant features will not influence the result.
- The presence of collinear features does not affect the result.
- However, it tends to **overfit.**

```
def create_DT():
    '''Create a new Decision Tree'''
    return DecisionTreeClassifier(random_state=SEED)
```

```
def getDecisionTreeBest(X, y, debug=True):
    '''Get the Decision Tree Hyper Parameters'''

    # Maximum number of levels in tree
    max_depth = [int(x) for x in range(2, 16, 2)]
    max_depth.append(None)

    # Create the random grid
    grid = {'classification__criterion': ['gini', 'entropy'],
            'classification__splitter': ['best'],
            'classification__max_features': ['auto', 'sqrt'],
            'classification__max_depth': max_depth,
            'classification__min_samples_split':  [2, 4, 6, 8],
            'classification__min_samples_leaf':  [1, 2, 4, 6],
            'classification__min_impurity_split': [0.05, 0.1, 0.23, 0.3],
            'classification__class_weight': ["balanced", None]}

    if debug:
        pp.pprint(grid)

    # Applying sampling techniques
    dt = apply_sampling(create_DT(), OVERSAMPLE, UNDERSAMPLE)

    # Using the grid search for best hyperparameters
    dt_grid = GridSearchCV(estimator = dt,
                           param_grid = grid,
                           scoring=metrics.make_scorer(auc_scorer,
                                                       greater_is_better=True),
                           cv=StratifiedKFold(K_FOLD_NUM_SPLITS,
                                              random_state=SEED,
                                              shuffle=True),
                           verbose=2,
                           n_jobs = -1)

    # Fit the grid search model
    dt_grid = dt_grid.fit(X, y)

    if debug:
        print('Best Score: ', dt_grid.best_score_)
        print('Best Params: ', dt_grid.best_params_)

    # Return score, method & params tuple
    return (dt_grid.best_score_, 'Decision Tree', dt_grid.best_params_)
```

Algorithms - Decision Tree

```
{   'classification__class_weight': ['balanced', None],
    'classification__criterion': ['gini', 'entropy'],
    'classification__max_depth': [2, 4, 6, 8, 10, 12, 14, None],
    'classification__max_features': ['auto', 'sqrt'],
    'classification__min_impurity_split': [0.05, 0.1, 0.23, 0.3],
    'classification__min_samples_leaf': [1, 2, 4, 6],
    'classification__min_samples_split': [2, 4, 6, 8],
    'classification__splitter': ['best']}
Fitting 5 folds for each of 4096 candidates, totalling 20480 fits
[Parallel(n_jobs=-1)]: Done 608 tasks      | elapsed:    3.8s
[Parallel(n_jobs=-1)]: Done 3512 tasks      | elapsed:   28.7s
[Parallel(n_jobs=-1)]: Done 6308 tasks      | elapsed:   43.4s
[Parallel(n_jobs=-1)]: Done 9704 tasks      | elapsed:   57.7s
[Parallel(n_jobs=-1)]: Done 14084 tasks      | elapsed:  1.3min
[Parallel(n_jobs=-1)]: Done 19424 tasks      | elapsed:  2.0min
[Parallel(n_jobs=-1)]: Done 20480 out of 20480 | elapsed:  2.1min finished
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default of the `iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This will change numeric results when test-set sizes are unequal.
  DeprecationWarning)
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/tree/tree.py:297: DeprecationWarning: The min_impurity_split parameter is deprecated. Its default value will change from 1e-7 to 0 in version 0.23, and it will be removed in 0.25. Use the min_impurity_decrease parameter instead.
  DeprecationWarning)
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Best Score:  0.7829368581514763
Best Params:  {'classification__class_weight': 'balanced', 'classification__criterion': 'gini', 'classification__max_depth': 4, 'classification__max_features': 'auto', 'classification__min_impurity_split': 0.23, 'classification__min_samples_leaf': 6, 'classification__min_samples_split': 2, 'classification__splitter': 'best'}
```

Grid searching hyper parameters for Decision Tree output example

Algorithms - Random Forest

- Harder to understand than Decision Tree but still acceptable for visual analysis.
- Since it inherently works with Decision Trees, it shares the same advantages.
- **More robust to overfit than Decision Trees,** thanks to the sampling of the data by its ***bag of trees***.
- Higher computational costs.

```python
def create_RF():
    '''Create a new Ranfom Forest model'''
    return RandomForestClassifier(random_state=SEED)
```

```python
def getRandomForestBest(X, y, debug=True):
    '''Get the Random Forest Hyper Parameters'''

    # Maximum number of levels in tree
    max_depth = [int(x) for x in range(2, 16, 4)]
    max_depth.append(None)

    # Create the random grid
    grid = {'classification__n_estimators': [int(x) for x in range(2, 14, 2)],
            'classification__max_features': ['auto', 'sqrt'],
            'classification__max_depth': max_depth,
            'classification__criterion': ['gini', 'entropy'],
            'classification__min_samples_split':  [2, 4, 6, 8],
            'classification__min_samples_leaf':  [1, 2, 4, 6],
            'classification__class_weight': ["balanced", "balanced_subsample", None]}

    if debug:
        pp.pprint(grid)

    # Applying sampling techniques
    rf = apply_sampling(create_RF(), OVERSAMPLE, UNDERSAMPLE)

    # Using the grid search for best hyperparameters
    rf_grid = GridSearchCV(estimator = rf,
                           param_grid = grid,
                           scoring=metrics.make_scorer(auc_scorer,
                                                       greater_is_better=True),
                           cv=StratifiedKFold(K_FOLD_NUM_SPLITS,
                                              random_state=SEED,
                                              shuffle=True),
                           verbose=2,
                           n_jobs = -1)

    # Fit the grid search model
    rf_grid = rf_grid.fit(X, y)

    if debug:
        print('Best Score: ', rf_grid.best_score_)
        print('Best Params: ', rf_grid.best_params_)

    # Return score, method & params tuple
    return (rf_grid.best_score_, 'Random Forest', rf_grid.best_params_)
```

Algorithms - Random Forest

```
{   'classification__class_weight': ['balanced', 'balanced_subsample', None],
    'classification__criterion': ['gini', 'entropy'],
    'classification__max_depth': [2, 6, 10, 14, None],
    'classification__max_features': ['auto', 'sqrt'],
    'classification__min_samples_leaf': [1, 2, 4, 6],
    'classification__min_samples_split': [2, 4, 6, 8],
    'classification__n_estimators': [2, 4, 6, 8, 10, 12]}
Fitting 5 folds for each of 5760 candidates, totalling 28800 fits
[Parallel(n_jobs=-1)]: Done 358 tasks      | elapsed:    5.0s
[Parallel(n_jobs=-1)]: Done 2052 tasks      | elapsed:   27.5s
[Parallel(n_jobs=-1)]: Done 4894 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 8156 tasks      | elapsed:  1.9min
[Parallel(n_jobs=-1)]: Done 10711 tasks      | elapsed:  2.5min
[Parallel(n_jobs=-1)]: Done 13826 tasks      | elapsed:  3.2min
[Parallel(n_jobs=-1)]: Done 17515 tasks      | elapsed:  4.1min
[Parallel(n_jobs=-1)]: Done 21764 tasks      | elapsed:  5.1min
[Parallel(n_jobs=-1)]: Done 23867 tasks      | elapsed:  5.7min
[Parallel(n_jobs=-1)]: Done 26174 tasks      | elapsed:  6.3min
[Parallel(n_jobs=-1)]: Done 28727 tasks      | elapsed:  6.9min
[Parallel(n_jobs=-1)]: Done 28800 out of 28800 | elapsed:  6.9min finished
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/sklearn/mod
el_selection/_search.py:814: DeprecationWarning: The default of the `iid` parameter will change from True to False
in version 0.22 and will be removed in 0.24. This will change numeric results when test-set sizes are unequal.
  DeprecationWarning)
/Users/edgarcarneiro/Documents/University/feup-ecac/project-competition/env/lib/python3.7/site-packages/imblearn/pi
peline.py:240: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the
shape of y to (n_samples,), for example using ravel().
  self._final_estimator.fit(Xt, yt, **fit_params)
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

Best Score:  0.8183838343317643
Best Params:  {'classification__class_weight': 'balanced_subsample', 'classification__criterion': 'entropy', 'class
ification__max_depth': 2, 'classification__max_features': 'auto', 'classification__min_samples_leaf': 1, 'classific
ation__min_samples_split': 2, 'classification__n_estimators': 12}
```

Grid searching hyper parameters for Random Forest output example

Algorithms - Gradient Boost

- **Learning Algorithm**, so the smaller the available data, the less efficient the algorithm.
- Works by building decision trees one-at-a-time, where each new tree corrects errors made by previous trees.
- More prone to **overfitting** than Random Forests.
- Most time-consuming algorithm.

```
def create_GB():
    '''Create a new Gradient Boosting model'''
    return GradientBoostingClassifier(random_state=SEED)
```

```
def getGradientBoostBest(X, y, debug=True):
    '''Get the Gradient Boost Hyper Parameters'''

    # Create the grid parameters
    grid = {'classification__n_estimators': [int(x) for x in range(2, 14, 2)],
            'classification__learning_rate': [0.1, 0.3, 0.5, 0.7],
            'classification__loss': ['deviance', 'exponential'],
            'classification__criterion': ['friedman_mse', 'mse', 'mae'],
            'classification__min_samples_split':  [4, 6, 8],
            'classification__min_samples_leaf':  [2, 4, 6]}

    if debug:
        pp.pprint(grid)

    # Applying sampling techniques
    gb = apply_sampling(create_GB(), OVERSAMPLE, UNDERSAMPLE)

    # Using the grid search for best hyperparameters
    gb_grid = GridSearchCV(estimator = gb,
                           param_grid = grid,
                           scoring=metrics.make_scorer(auc_scorer,
                                                       greater_is_better=True),
                           cv=StratifiedKFold(K_FOLD_NUM_SPLITS,
                                              random_state=SEED,
                                               shuffle=True),
                           verbose=2,
                           n_jobs = -1)

    # Fit the grid search model
    gb_grid = gb_grid.fit(X, y)

    if debug:
        print('Best Score: ', gb_grid.best_score_)
        print('Best Params: ', gb_grid.best_params_)

    # Return score, method & params tuple
    return (gb_grid.best_score_, 'Gradient Boosting', gb_grid.best_params_)
```

Algorithms - Gradient Boosting

```
{   'classification__criterion': ['friedman_mse', 'mse', 'mae'],
    'classification__learning_rate': [0.1, 0.3, 0.5, 0.7],
    'classification__loss': ['deviance', 'exponential'],
    'classification__min_samples_leaf': [2, 4, 6],
    'classification__min_samples_split': [4, 6, 8],
    'classification__n_estimators': [2, 4, 6, 8, 10, 12]}
Fitting 5 folds for each of 1296 candidates, totalling 6480 fits

[Parallel(n_jobs=-1)]: Done 308 tasks      | elapsed:    4.9s
[Parallel(n_jobs=-1)]: Done 1760 tasks      | elapsed:   26.0s
[Parallel(n_jobs=-1)]: Done 3926 tasks      | elapsed:   58.8s
[Parallel(n_jobs=-1)]: Done 4613 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 4978 tasks      | elapsed:  1.8min
[Parallel(n_jobs=-1)]: Done 5423 tasks      | elapsed:  2.2min
[Parallel(n_jobs=-1)]: Done 5950 tasks      | elapsed:  2.6min

Best Score:  0.7577567839823135
Best Params:  {'classification__criterion': 'friedman_mse', 'classification__learning_rate': 0.7, 'classification__
loss': 'exponential', 'classification__min_samples_leaf': 6, 'classification__min_samples_split': 4, 'classificatio
n__n_estimators': 8}
```

Grid searching hyper parameters for Gradient Boosting output example

Evaluation measures

- For the performance evaluation, we use the **Area under Curve** (AUC) value, seeing that it is the same metric used in the kaggle competition.
- For evaluation of the expected error interval, we observe how the AUC varies between different folds.

```
AUC scores: [0.8649122807017544, 0.8742690058479532, 0.8174603174603174, 0.8571428571428572, 0.8998015873015872]
> Average:  0.8627172096908939
            Predic NO       Predic YES
Actual NO       8               2
Actual YES      4               53
            Predic NO       Predic YES
Actual NO       8               1
Actual YES      8               49
            Predic NO       Predic YES
Actual NO       7               2
Actual YES      8               48
            Predic NO       Predic YES
Actual NO       9               0
Actual YES      16              40
            Predic NO       Predic YES
Actual NO       8               1
Actual YES      5               51
```

Example of scoring variation across folds

```python
def auc_scorer(y_true, y_pred):
    '''Scorer of Area Under Curve value'''
    fpr, tpr, _ = metrics.roc_curve(y_true, y_pred)
    return metrics.auc(fpr, tpr)
```

AUC score evaluation function

Result Analysis - Principal Component Analysis

Our thought regarding PCA performance decrease:

- Seeing that PCA is agnostic to Y, when reducing the dimensionality of our data, PCA may be throwing away information that even though it does not explain much of the data variance, it is important for r determining Y.

```
1. Random Forest - 0.832330
---------
2. Gradient Boosting - 0.808303
---------
3. Decision Tree - 0.806183
---------
4. Logistic Regression - 0.785563
---------
Best algorithm: Random Forest
```

Ranking of used algorithms, without PCA

```
1. Logistic Regression - 0.768239
---------
2. Gradient Boosting - 0.695331
---------
3. Decision Tree - 0.689463
---------
4. Random Forest - 0.689277
---------
Best algorithm: Logistic Regression
```

Ranking of used algorithms, with PCA

Result Analysis - local score

	Feature Selection Technique	None				Filter Method			
	Sampling	None	Over	Under	Both	None	Over	Under	Both
Algorithms	**Logistic Regression**	0.795	0.798	0.796	0.809	0.830	0.823	0.825	0.821
	Decision Tree	0.799	0.811	0.808	0.811	0.817	0.812	0.825	0.851
	Random Forest	0.813	0.832	0.817	0.832	0.844	**0.868**	0.853	0.865
	Gradient Boosting	0.744	0.813	0.755	0.813	0.763	0.842	0.756	0.849

Result Analysis - local score

	Feature Selection Technique	Backward Elimination				Random Forest				Recursive Feature Engineering			
	Sampling	None	Over	Under	Both	None	Over	Under	Both	None	Over	Under	Both
Algorithms	**Logistic Regression**	0.811	0.813	0.812	0.823	0.810	0.803	0.815	0.794	0.808	0.823	0.817	0.823
	Decision Tree	0.791	0.814	0.812	0.811	0.783	0.817	0.799	0.814	0.824	0.816	0.826	0.806
	Random Forest	0.819	0.835	0.809	0.837	0.818	0.830	0.802	0.824	0.830	0.832	0.830	0.807
	Gradient Boosting	0.796	0.813	0.787	0.804	0.753	0.849	0.758	0.845	0.696	0.828	0.760	0.837

Result Analysis - Conclusions

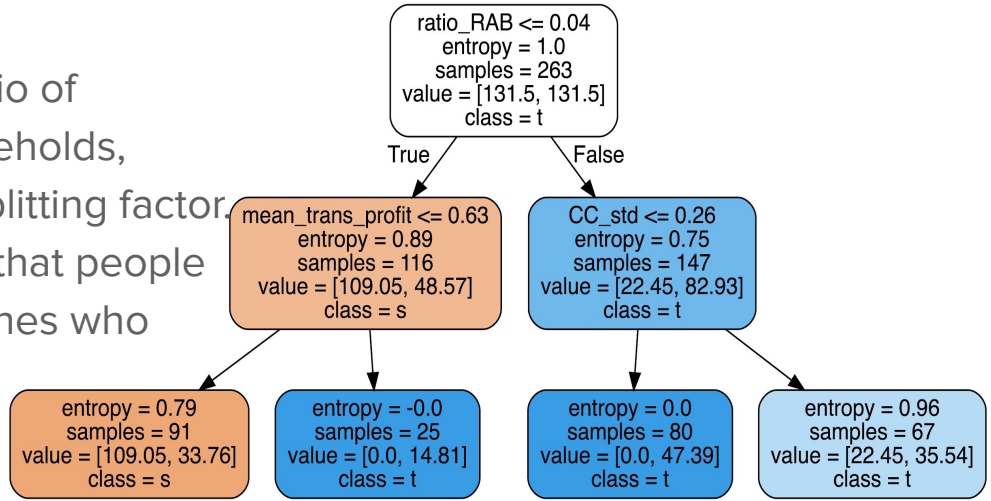
- **Logistic Regression tends to overfit**, having great local scores but having bad performances on kaggle.
- **Random Forest** is overall the **best performing algorithm** and proves to be robust to overfitting**.**
- Feature Selection is having an impact of about 5% in the algorithm's scoring ability.
- **Random Forest Feature Selection** technique is the one providing the **worst results.**
- **Filter Method** (using correlations) **Feature Selection** technique is the one providing the highest-scoring results.

Result Analysis - Conclusions

- **Gradient Boosting** algorithm scoring capability increases greatly when using **oversampling** - expected seeing that it is a learning algorithm, and so it works better by having more data.
- **Gradient Boosting tends to overfit**, having great local scores but having bad performances on kaggle.
- **Gradient Boosting** is the algorithm having the more varied scores.
- Regarding sampling, **SMOTE oversampling** tends to provide the best scores, with the mixed approach - **SMOTETEEN sampling** - having similar, although slightly worse results. **Undersampling** scores are usually worse than the results obtained when not using any sampling technique.

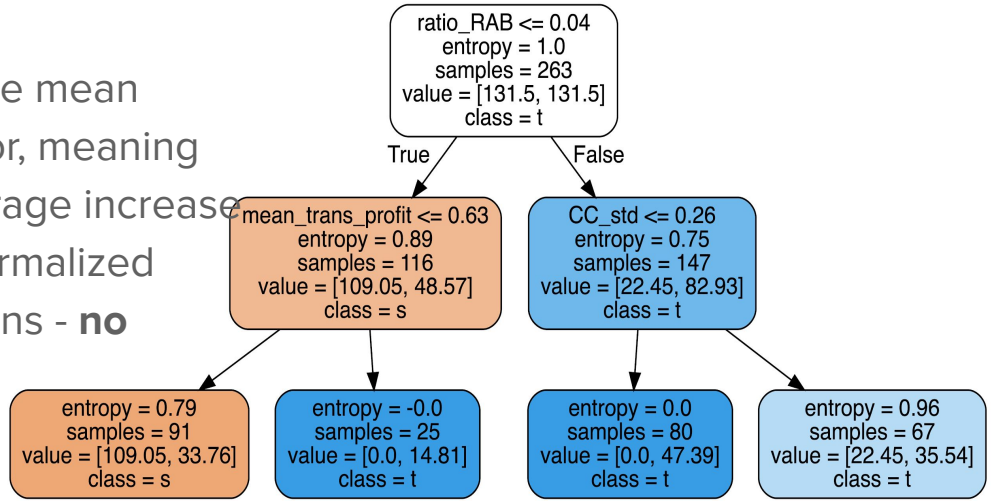
Model Analysis

- Most simple obtained Decision Tree that grants a solid score of around **80%,** without feature selection techniques.
- Difficult to analyse the splitting values since they are normalised.
- By its **analysis** we can see that the ratio of Remittances from another Bank (households, insurance payment, ...) are the main splitting factor. Looking at the context, it might mean that people that make more remittances are the ones who actually pay the bills.



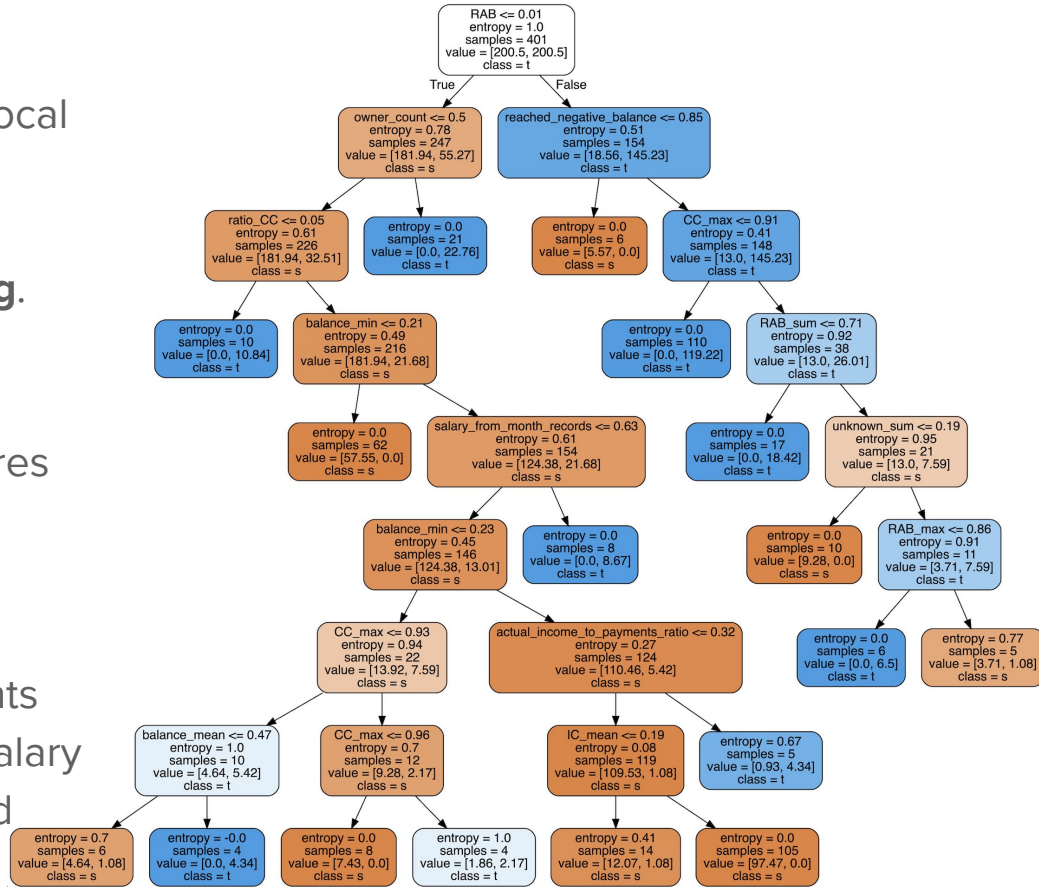
Model Analysis

- Considering the right subtree, the clients that have constant credits in cash values are assumed to pay its loans - **no entropy leaf** -, whilst the others are too, even though it is a leaf with still some entropy..
- Considering the left subtree, it uses the mean transaction profit as the deciding factor, meaning that clients whose transactions in average increase the account's balance (by a certain normalized amount) are considered to pay the loans - **no entropy leaf** -, whilst others do not.



Model Analysis

- **Decision Tree model** providing the local best scores (among Decision Trees). Resultant of **Filter Method** feature selection and **SMOTETEEN sampling**.
- Score: **0.851**
- Harder to analyse, since the tree is deeper. However, the decision features are now ones we naturally expect to relate with the loan output, such as: number of account owners, reached negative balance, income to payments ratio, balance minimum, computed salary from month records, interest credited mean, among others.



Used Tools

- **Python**: Development of the entire Pipeline. Single tool use with the intention of building a more automated process.
  - numpy: For numerical handling of data. Used by both *pandas* and the plotting libs.
  - *pandas*: For data manipulation.
  - *Imblearn:* For handling imbalanced data, sampling and Pipelines.
  - *scikit learn:* For the application of all the prediction algorithms, cross validation, hyper parameter tuning, Principal Component Analysis and data normalization.
  - *matplotlib & seaborn:* For plotting of data
- ***Excel****:* Mainly for visualization of the raw data provided through the *.csv* files.