Data features

- **Revolving Utilization Of Unsecured Lines**: Total balance on credit cards and personal lines of credit except for real estate, and no installment debt, like car loans, divided by the sum of credit limits
- **Age**: Age of borrower in years
- **The number of times 30-59 days past due not worse**: Number of times borrower has been 30-59 days past due, but no worse, in the last 2 years
- **Debt Ratio**: Monthly debt payments, alimony, living costs divided by monthly gross income
- **Monthly Income**: Monthly income per person
- **The number of open credit lines and loans**: Number of Open loans (installment like car loan or mortgage) and lines of credit (e.g. credit cards)
- **The number of times 90 days Late**: Number of times borrower has been 90 days or more past due
- **The number of real estate loans or lines**: Number of mortgage and real estate loans including home equity lines of credit
- **The number of times 60-89 days past due not worse**: Number of times borrower has been 60-89 days past due, but no worse, in the last 2 years
- **The number of dependents**: Number of dependents in family excluding themselves (spouse, children, etc.)

Target

- **Serious Delinquency in 2 years**: Person experienced 90 days past due delinquency or worse

The target indicates the delinquency of a debtor measured in a time window of two years. A value of 1 means that the borrower is delinquent and has defaulted on his loans for the last 2 years. Value of 0 means that the borrower is a good customer and has repaid debts on time for the last two years.

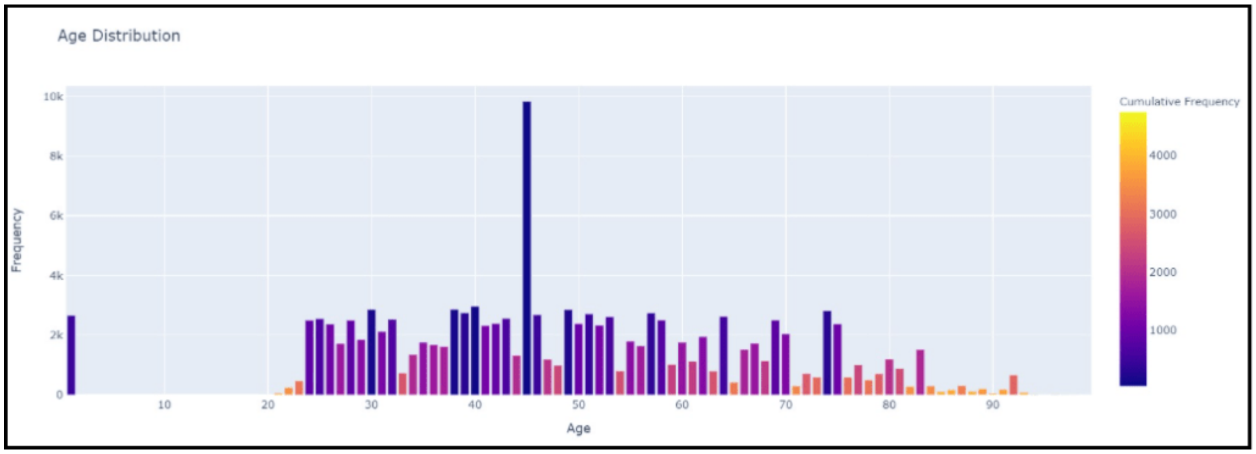
Commonly, most financial industry data contains missing values, or values that don’t make sense for a particular characteristic. This particular dataset is no exception, with inconsistencies regarding the debt and credit balance ratios, and values way over what should be admitted.

So, we’ll apply data transformations to get rid of all discrepancies that would alter the results in the modelization and training stages.

For now, let’s focus on the descriptive analysis:

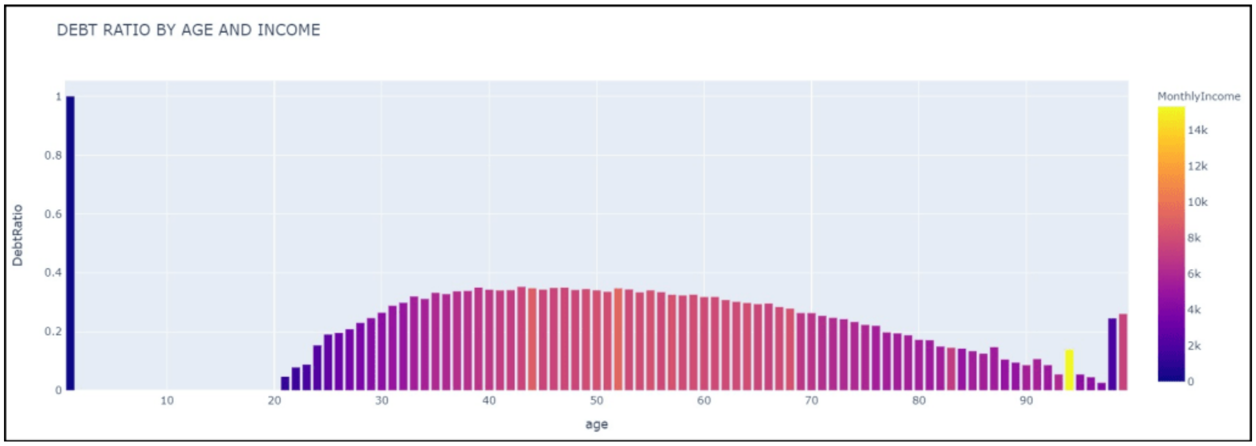
Unsurprisingly, the target characteristic is highly unbalanced, which will lead to serious problems in model training. Having **86.3**% of debtors classified as bad payers will induce the model to overfit for this particular class, and completely disregard the other.

By looking at the **Age** distribution, we clearly observe that the age group between the 40s and 50s contains most samples, and the other remaining groups are more or less equally balanced.



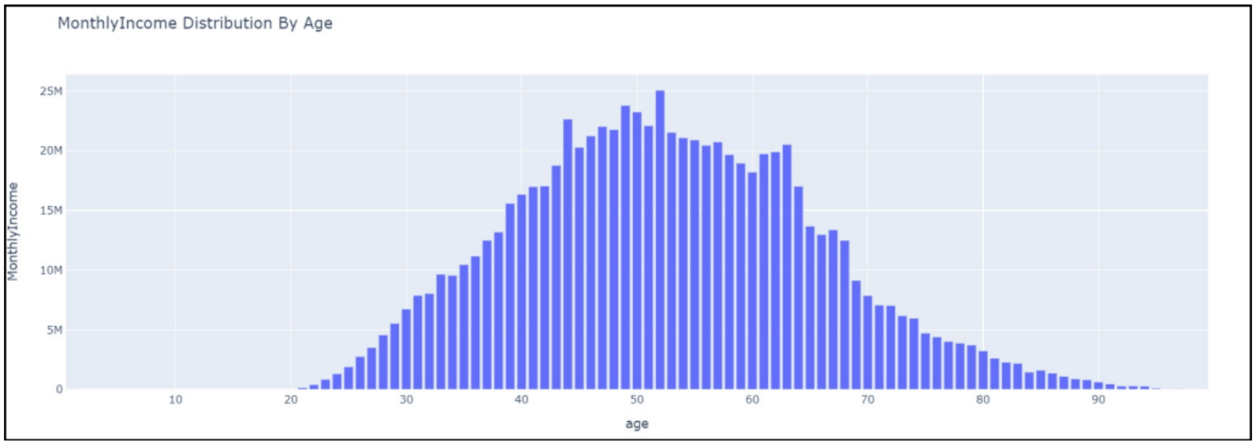
Age distribution regarding Frequency and Cumulative Frequency

Age analyses show that the most indebted segment of the population is between 35 and 70 years of age. On top of that, debt is always higher for populations with the lowest monthly salaries.



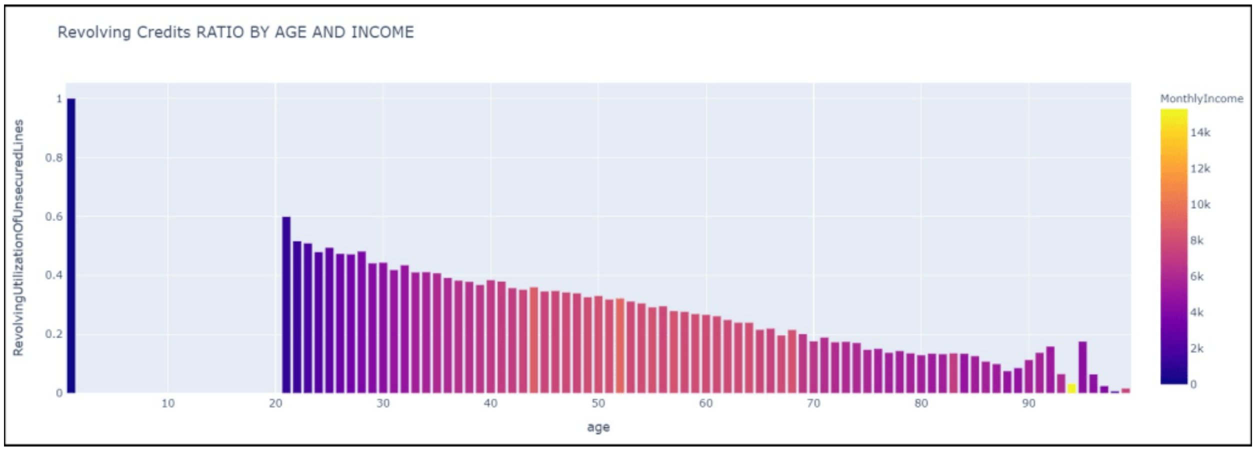
Debt ratio distribution regarding Age and Monthly Income

The segment of the population that accumulates the most financial resources is between 45 and 60, which is quite logical and plausible.



Monthly Income distribution by Age

Revolving debts follow a trend quite similar to long-term debts, with populations that are quite young, and always more and more indebted. The most obvious correlations in the data are those between short and long term debts, age group and salary.



Revolving credits ratio by Age and Monthly Income

Now that we have fairly more knowledge and understanding about the dataset, we can proceed to the ETL processes for data extraction and transformation.

Build and automate your ETL process

In this section, we'll implement an ETL workflow that extracts data from a csv file to a usable pandas dataframe, and we'll apply all the required transformations to have a clean and prepared dataset for the ML models. To make things more efficient, we'll try to simulate an automation cycle for the process to run through.

This section will introduce concepts such as Airflow Directed Acyclic Graphs for process automation, and Factor Analysis procedures to implement data transformations.

Data extraction

We want to extract the data from the *csv* file, and make it usable for our experimental purposes. To do, first we create a small *Python Data Manager* class that will take care of parsing the csv, extract, and format any relevant data for our analysis.

```
class DataETLManager:
    def __init__(self, root_dir: str, csv_file: str):
        if os.path.exists(root_dir):
            if csv_file.endswith('.csv'):
                self.csv_file = os.path.join(root_dir, csv_file)
            else:
                logging.error('The file is not in csv format')
                exit(1)
        else:
            logging.error('The root dir path does not exist')
            exit(1)

        self.credit_scoring_df = pd.read_csv(self.csv_file, sep=',',
encoding='ISO-8859-1')


    def extract_data(self):
        return self.credit_scoring_df

    def fetch_columns(self):
        return self.credit_scoring_df.columns.tolist()

    def data_description(self):
        return self.credit_scoring_df.describe()

    def fetch_categorical(self, categorical=False):
        if categorical:
            categorical_columns = list(set(self.credit_scoring_df.columns) -
set(self.credit_scoring_df._get_numerical_data().columns))
            categorical_df = self.credit_scoring_df[categorical_columns]
            return categorical_df
        else:
            non_categorical =
list(set(self.credit_scoring_df._get_numerical_data().columns))
            return self.credit_scoring_df[non_categorical]
```

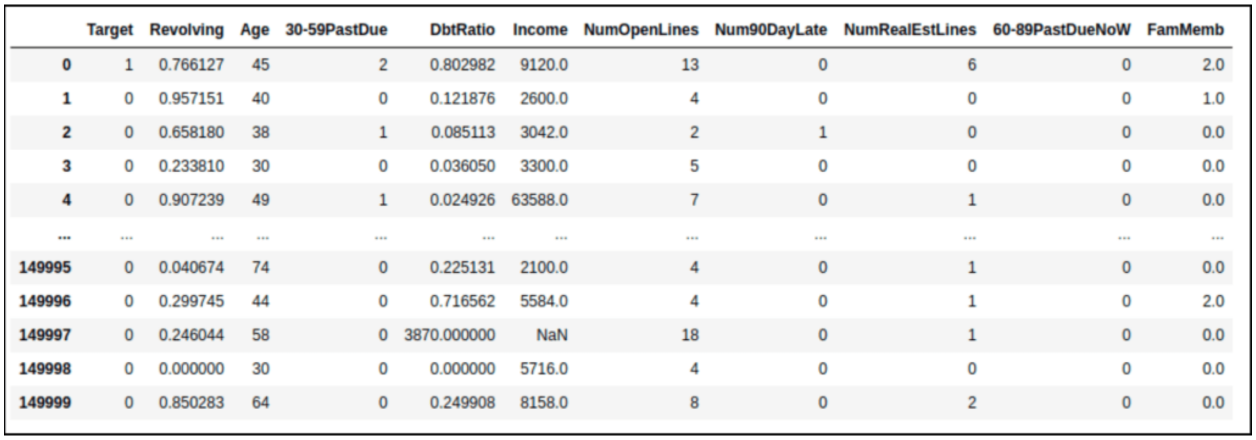
Three important methods to take a closer look at:

- extract_data(): returns the credit scoring data frame we've just created
- fetch_columns(): returns all the columns in the data
- data_description(): useful if we want a have a quick glance at the structure of the data
- fetch_categorical(): returns the categorical values in the data frame

We'll change the column names to be shorter:

```
credit_df=credit_df.drop('Unnamed: 0', axis=1)
credit_df.columns = ['Target', 'Revolving', 'Age', '30-59PastDue',
'DbtRatio', 'Income', 'NumOpenLines', 'Num90DayLate', 'NumRealEstLines', '60-
89PastDueNoW', 'FamMemb']
```

The result looks like this:



	Target	Revolving	Age	30-59PastDue	DbtRatio	Income	NumOpenLines	Num90DayLate	NumRealEstLines	60-89PastDueNoW	FamMemb
0	1	0.766127	45	2	0.802982	9120.0	13	0	6	0	2.0
1	0	0.957151	40	0	0.121876	2600.0	4	0	0	0	1.0
2	0	0.658180	38	1	0.085113	3042.0	2	1	0	0	0.0
3	0	0.233810	30	0	0.036050	3300.0	5	0	0	0	0.0
4	0	0.907239	49	1	0.024926	63588.0	7	0	1	0	0.0
...	...	...	...	...	...	...	...	...	...	...	...
149995	0	0.040674	74	0	0.225131	2100.0	4	0	1	0	0.0
149996	0	0.299745	44	0	0.716562	5584.0	4	0	1	0	2.0
149997	0	0.246044	58	0	3870.000000	NaN	18	0	1	0	0.0
149998	0	0.000000	30	0	0.000000	5716.0	4	0	0	0	0.0
149999	0	0.850283	64	0	0.249908	8158.0	8	0	2	0	0.0

Credit Scoring datatset

Data transformation

We'll focus on two transformation phases:

- Preprocessing transformations
- Analytics transformations

The idea is that we absolutely need to preprocess incoming raw data, eliminate duplicates, drop null and missing values. Furthermore, doing univariate analysis we'll soon observe that many of the samples had out of range values for ratio variables. Typically, we need to detect and delete outliers.

With usable data, we'll start implementing Factor Analysis to extract insightful features that best explain variance and correlation.

Preprocessing transformations

Dropping duplicates and missing values:

```
def transform_data(self):
        # 1 Drop Duplicates:
        self.credit_scoring_df.drop_duplicates(keep='last', inplace=True)

        # 2 Drop null values:
        self.credit_scoring_df.dropna(how='all', inplace=True)
```

Removing outliers:

- For the age feature which is a continuous variable from 0 to a maximum of 100. There are certain records, with a value of zero, that don't make sense and can't be qualified as a borrower, the person must be an adult of 18 years.

- Checking for upper bounding and top-coding methods for the debt and revolving ratios with values greater than 1, which means that all values above the upper band will be dropped.
- Exclude characteristics or records that have significant (more than 50%) missing values from the model, especially if the level of missing is important enough concerning the rate of data unbalancing, which is quite high.

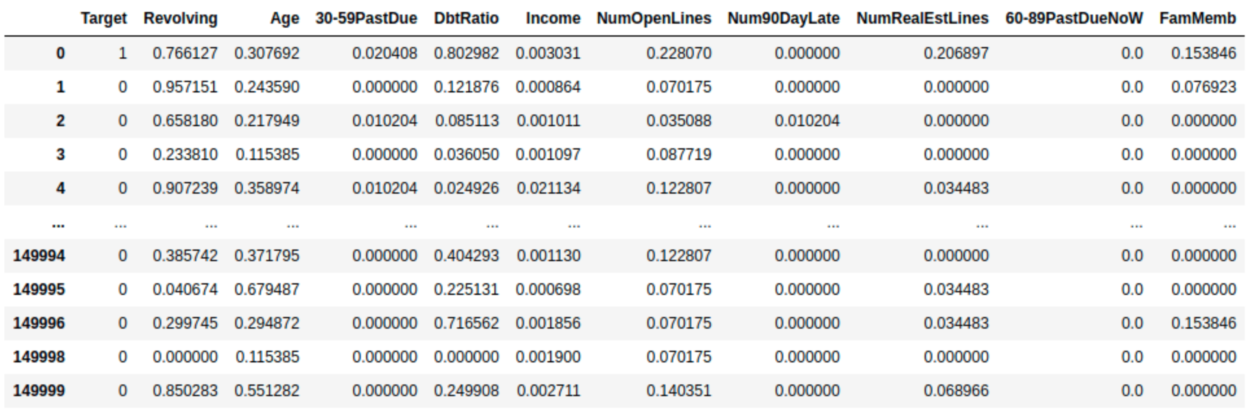
```
clean_credit = self.credit_scoring_df.loc[self.credit_scoring_df['Revolving']
<= 1]
clean_credit = clean_credit.loc[clean_credit['DbtRatio'] <= 1]
clean_credit = clean_credit.loc[clean_credit['Age'] <= 100]
clean_credit = clean_credit.loc[clean_credit['Age'] >= 18]
clean_credit = clean_credit.loc[clean_credit['FamMemb'] < 20]
```

Analytics transformations

After cleaning the data, we'll standardize the data with **mean=0** and **std deviation=1**, except for the binary dependent variable which is the Target.

```
def normalize(dataset):
    dataNorm=((dataset-dataset.min())/(dataset.max()-dataset.min()))
    dataNorm["Target"]=dataset["Target"]
    return dataNorm

clean_scaled_df = normalize(clean_credit)
```

The data values range from 0 to 1:



	Target	Revolving	Age	30-59PastDue	DbtRatio	Income	NumOpenLines	Num90DayLate	NumRealEstLines	60-89PastDueNoW	FamMemb
0	1	0.766127	0.307692	0.020408	0.802982	0.003031	0.228070	0.000000	0.206897	0.0	0.153846
1	0	0.957151	0.243590	0.000000	0.121876	0.000864	0.070175	0.000000	0.000000	0.0	0.076923
2	0	0.658180	0.217949	0.010204	0.085113	0.001011	0.035088	0.010204	0.000000	0.0	0.000000
3	0	0.233810	0.115385	0.000000	0.036050	0.001097	0.087719	0.000000	0.000000	0.0	0.000000
4	0	0.907239	0.358974	0.010204	0.024926	0.021134	0.122807	0.000000	0.034483	0.0	0.000000
...	...	...	...	...	...	...	...	...	...	...	...
149994	0	0.385742	0.371795	0.000000	0.404293	0.001130	0.122807	0.000000	0.000000	0.0	0.000000
149995	0	0.040674	0.679487	0.000000	0.225131	0.000698	0.070175	0.000000	0.034483	0.0	0.000000
149996	0	0.299745	0.294872	0.000000	0.716562	0.001856	0.070175	0.000000	0.034483	0.0	0.153846
149998	0	0.000000	0.115385	0.000000	0.000000	0.001900	0.070175	0.000000	0.000000	0.0	0.000000
149999	0	0.850283	0.551282	0.000000	0.249908	0.002711	0.140351	0.000000	0.068966	0.0	0.000000

Normalized data values

Start with factor analysis

Before segregating the data for training phases, we want to check correlation between all the variables, to get an idea on how the variables will be combined.

Factor analysis is a linear statistical model. It's used to explain the variance among observed variables, and condense a set of observed variables into the *unobserved variable*, called factors. Observed variables are modeled as a linear combination of factors and error terms.

First, let's run an adequacy test to check whether the dataset is suitable for Factor Analysis or not. We'll perform Barlett's test of sphericity.

```
from scipy.stats import chi2, pearsonr
import numpy as np

def barlett_test(frame: pd.DataFrame):
    # print dataframe info:
    frame.info()

    col, row = frame.shape
    x_corr = frame.corr()

    corr_det = np.linalg.det(x_corr)
    chi_measure = -np.log(corr_det) * (col - 1 - (2 * row + 5) / 6)
    degrees_of_freedom = row * (row - 1) / 2
    p_value = chi2.sf(statistic, degrees_of_freedom)
    return chi_measure, p_value
chi_square, p_value = barlett_test(clean_scaled_df)
(1003666.113272278, 0.0)
```

Highly Suitable: *The observed correlation is different from the identity matrix, H0 and H1 hypothesis are verified.*

We'll use the factor_analyzer python package, install it with the following command:

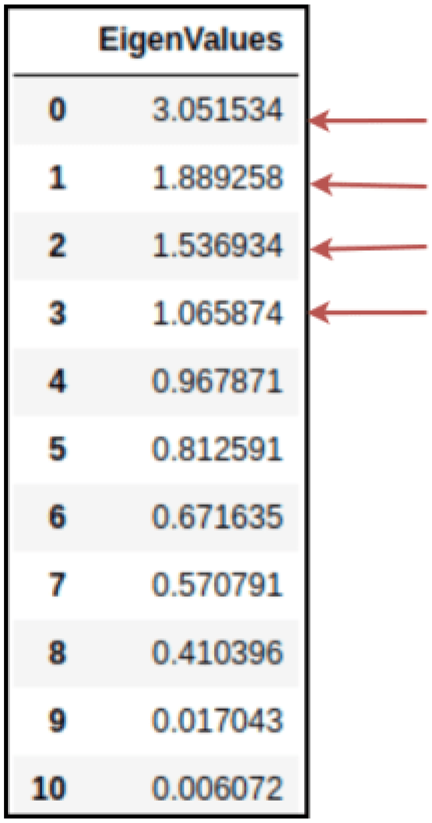
```
pip install factor_analyzer
```

Fit the data to the FactorAnalyzer class that we'll run Kaiser criterion internal statistics to come up with EigenValues in the data.

```
from factor_analyzer import FactorAnalyzer

fa = FactorAnalyzer()
fa.fit(clean_scaled_df)

# EigenValues:
ev, v = fa.get_eigenvalues()
eigen_values = pd.DataFrame(ev)
eigen_values
```

Original EigenValues:

There are four principal components whose eigenvalue is more than 1:

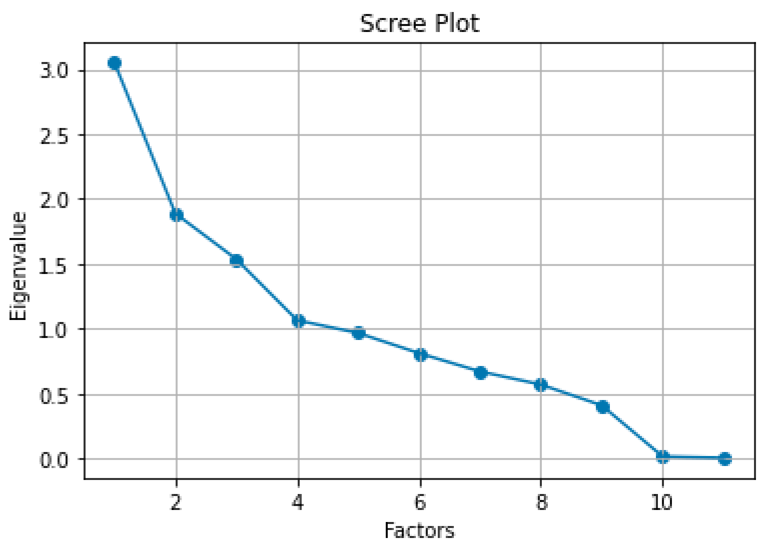


	EigenValues
0	3.051534
1	1.889258
2	1.536934
3	1.065874
4	0.967871
5	0.812591
6	0.671635
7	0.570791
8	0.410396
9	0.017043
10	0.006072

Eigen Values after Running Factor Analysis

Plotting a Scree plot we can easily visualize the four relevant factors we need:

```python
import matplotlib.pyplot as plt

plt.scatter(range(1,clean_scaled_df.shape[1]+1),ev)
plt.plot(range(1,clean_scaled_df.shape[1]+1),ev)
plt.title('Scree Plot')
plt.xlabel('Factors')
plt.ylabel('Eigenvalue')
plt.grid()
plt.show()
```



Scree plot for the 4 factors

Learn more

Check how to log charts generated in matplotlib to Neptune.

Let’s perform Factor Analysis rotation on those 4 factors to have a better interpretation. Rotation can be orthogonal or oblique. It helps re-distributing the communalities between the observed variables with a clear pattern of loadings.

```
fac_rotation = FactorAnalyzer(n_factors=4, rotation='varimax')
fac_rotation.fit(clean_scaled_df)

fac_rotation.get_factor_variance()
```

Characteristics
Factor 1
Factor 2
Factor 3
Factor 4

SS Loading

2.94778507

1.42332453

0.9133116

0.36775979

Prop Variance

0.26798046

0.12939314

0.08302833

0.03343271

Cum Variance

0.26798046

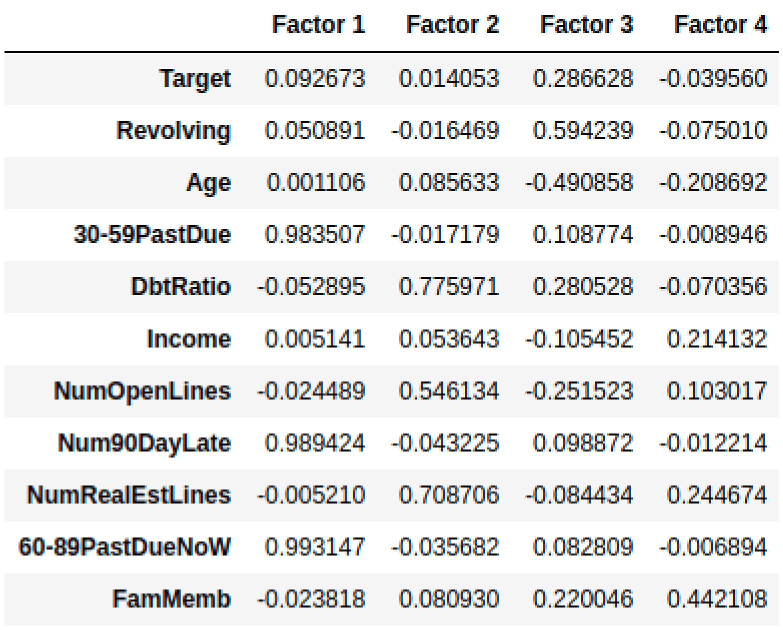
0.3973736

0.48040193

0.51383464

We conclude that the variance explained by these four components is **51.4%**, which is satisfactory. Based on this, we decide to use four factors for further analysis.

Displaying the 4 Factors with their respective observable features, we see that:



	Factor 1	Factor 2	Factor 3	Factor 4
Target	0.092673	0.014053	0.286628	-0.039560
Revolving	0.050891	-0.016469	0.594239	-0.075010
Age	0.001106	0.085633	-0.490858	-0.208692
30-59PastDue	0.983507	-0.017179	0.108774	-0.008946
DbtRatio	-0.052895	0.775971	0.280528	-0.070356
Income	0.005141	0.053643	-0.105452	0.214132
NumOpenLines	-0.024489	0.546134	-0.251523	0.103017
Num90DayLate	0.989424	-0.043225	0.098872	-0.012214
NumRealEstLines	-0.005210	0.708706	-0.084434	0.244674
60-89PastDueNoW	0.993147	-0.035682	0.082809	-0.006894
FamMemb	-0.023818	0.080930	0.220046	0.442108

Factors and Observable Features side by side

I Observation

- The variables *30-59PastDue*, *60-89PastDueNoW* and *Num90DayLate* load high on **Factor1**. This means that this factor causes the borrower to be more and more delinquent. Hence we can name this factor as ***Financial Struggle***.

II Observation

- The variables *DbtRatio*, *NumOpenLines* and *NumRealEstLines* load high on **Factor2**. Which means that this factor causes the borrower to engage in more debt and more credit lines. We can name it ***Finance Requirements***.

III Observation

- Variables *Age*, *FamMemb* and *Revolving* load high on **Factor3** with Age being indirectly proportional to Factor 3. This factor has been named as ***Expendable Income*** because as age increases, expendable income increases.

IV Observation

- The Variables *Income* and *FamMemb* load high on **Factor4**. So we can easily name it ***Behavioral LifeStyle*** because as income increases, so does our lifestyle and the lifestyle of our family members.

So, now we have the four factors for ML analysis:

- Financial Struggle
- Finance Requirements
- Expendable Income
- Behavioral Lifestyle

Automate the ETL process with DAGs

Once the Extract and Transform pipelines are formed and ready for deployment, we can start thinking of a way to store the previous data in a database.

The Load process comes into picture, to simplify and for the purposes of illustration, we're just going to load the previous transformed data frame into a local MySQL database using the Sql_Alchemy python package.

```
pip install SQLAlchemy
from sqlalchemy.engine import create_engine

def load_data(self):
        database_config = {
            'username': 'your_username',
            'password': 'your_pass',
            'host': '127.0.0.1',
            'port':'3306',
            'database':'db_name'
        }

        # create database connection using engine:
        engine =
create_engine('mysql+mysqlconnector://{}:{}@{}:{}/{}'.format(
            database_config['username'],
            database_config['password'],
            database_config['host'],
            database_config['port'],
            database_config['database']
        ))

        data_to_load = type(pd.DataFrame())(self.credit_scoring_df)
        try:
            data_to_load.to_sql('Credit Scoring', con=engine,
if_exists='append', index=False)
        except Exception as err:
            print(err)
```

Airflow DAGs: Directed Acyclic Graphs

Airflow schedules automated data workflows that include multiple tasks or processes that share specific dependencies. The Airflow documentation defines DAGs as follows:

"*In Airflow, a DAG – or a Directed Acyclic Graph – is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies. A DAG is defined in a Python script, which represents the DAGs structure (tasks and their dependencies) as code*"

To install and start using Airflow, you can check the website, which explains each step thoroughly: AirFlow Doc.

In our case, we'll code a small DAG file to simulate the automation of our ETL. We'll schedule the DAG to run daily starting from the 2021/03/25. The DAG will have three PythonOperators representing the Extract, Transform and Load functions.

```
from datetime import timedelta, datetime
from airflow import DAG
from airflow.operators.python import PythonOperator
from etl_process import DataETLManager, DATA_PATH

default_dag_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2021, 3, 9),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 2,
    'retry_delay': timedelta(minutes=1),
}

etl_dag = DAG(
    'etl_retail',
    default_args=default_dag_args,
    description='DAG for ETL retail process',
    schedule_interval=timedelta(days=1),
    tags=['retail']
)
```

The PyhtonOperators in charge of running each ETL process:

```
etl_manager = DataETLManager(DATA_PATH, 'OnlineRetail.csv')

extract = PythonOperator(
    task_id='extract_data',
    python_callable=etl_manager.extract_data,
    dag=etl_dag
)

transform = PythonOperator(
    task_id='transform_data',
    python_callable=etl_manager.transform_data,
    dag=etl_dag
)

load = PythonOperator(
    task_id='load_data',
    python_callable=etl_manager.load_data,
    dag=etl_dag
)
```

Finally, we define tasks correlation: Extract, then Transform, then Load into the Database.

```
extract >> transform >gt; load
```

Run predictions and benchmark results

In this part, we'll segregate the transformed and reduced data we've obtained into training and testing sets, and use three ensemble algorithms to predict serious delinquency using the 4 factors that we previously pulled out.

We'll be integrating all our ML development with Neptune, to keep track and compare all the metrics that we're getting along the way.

Please note that due to the recent API update, this post needs some changes as well – we're working on it! In the meantime, please check the Neptune documentation, where everything is up to date!

Using three ML models, we'll be able to compare the results and see where each approach can perform better. We'll be testing three ML models for this particular task:

- Logistic Regression
- Decision Trees
- Extreme Gradient Boosting

Set Neptune for development

Install the required neptune client libraries to start integrating your code with Neptune:

Install neptune library:

```
pip  install neptune-client
```

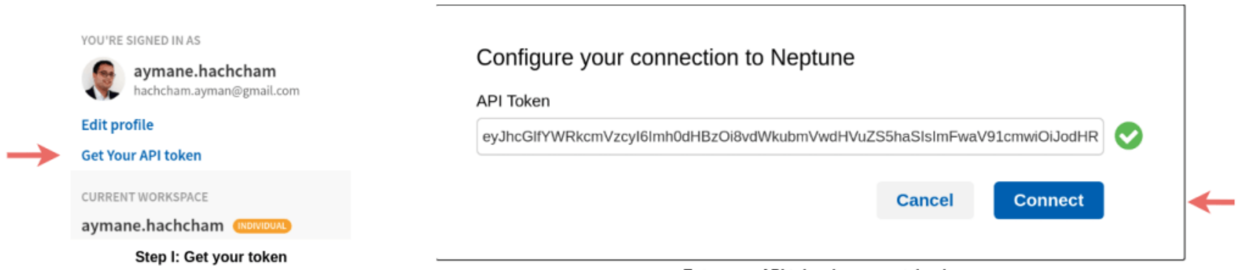
Install Neptune Notebooks, so that can save all our work to Neptune website

```
pip install -U neptune-notebooks
```

Enable jupyter integration by installing the following extension

```
jupyter nbextension enable --py neptune-notebooks
```

Get your api Key and connect your notebook with your Neptune session:



API Token

eyJhcGlfYWRkcmVzcyI6Imh0dHBzOi8vdWkubmVwdHVuZS5haSIsImFwaV91cmwiOiJodHR

Step I: Get your token

Enter your API token in your notebook

Set up your Neptune Token from the website

To complete the setup, import the neptune client library in your notebook and initialize the connection calling the neptune.init() method:

```
import neptune
neptune.init(project_qualified_name='aymane.hachcham/CreditScoring')
```

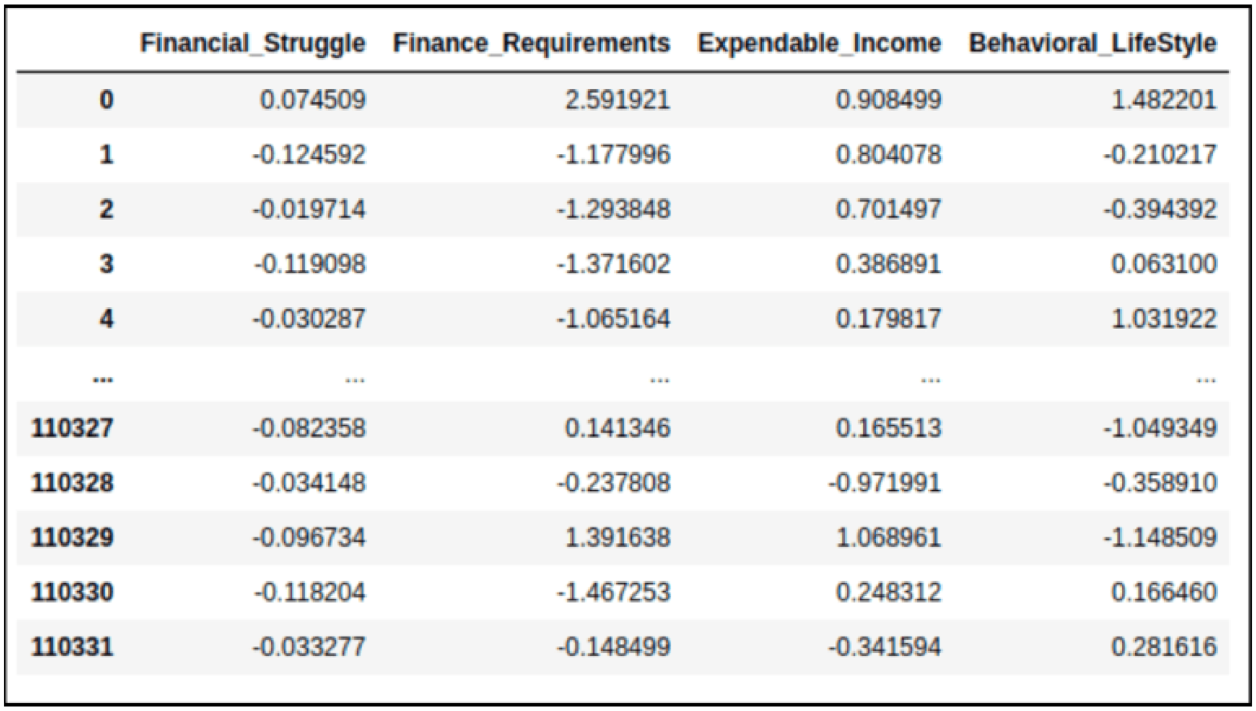
Logistic regression

The main goal of regression models is to find a combination between the predictors that best fit a set of data according to a specific mathematical criterion.

The most common regression model used in the case of a categorical response data problem is logistic regression. Typically, logistic regression uses, in its basic form, a logistic function to model a binary dependent variable based on various predictors.

The training of the model is notably straightforward. We used the well-known machine learning library *SciKit-Learn* to implement the logistic model. The effort made previously on the data processing will help us quite a lot during this part.

The table with previously defined factors looks like this:



	Financial_Struggle	Finance_Requirements	Expendable_Income	Behavioral_LifeStyle
0	0.074509	2.591921	0.908499	1.482201
1	-0.124592	-1.177996	0.804078	-0.210217
2	-0.019714	-1.293848	0.701497	-0.394392
3	-0.119098	-1.371602	0.386891	0.063100
4	-0.030287	-1.065164	0.179817	1.031922
...	...	...	...	...
110327	-0.082358	0.141346	0.165513	-1.049349
110328	-0.034148	-0.237808	-0.971991	-0.358910
110329	-0.096734	1.391638	1.068961	-1.148509
110330	-0.118204	-1.467253	0.248312	0.166460
110331	-0.033277	-0.148499	-0.341594	0.281616

Table that aggregates the values for the 4 factors

```
credit_scoring_final = pd.DataFrame(new_data_frame,
columns=['Financial_Struggle', 'Finance_Requirements', 'Expendable_Income',
'Behavioral_LifeStyle'])
credit_scoring_final
```

Segregate the data in training and testing sets:

```
X = credit_scoring_final
x_train, x_test, y_train, y_test = train_test_split(X, target,
test_size=0.25, random_state=56)
```

Train with Logistic Regression:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(C=0.00026366508987303583, class_weight=None,
dual=False, max_iter=100, multi_class='auto', n_jobs=None, penalty='l1',
random_state=None, solver='saga')
model.fit(x_train, y_train)
```

Test Results:

```
from sklearn.metrics import accuracy_score, classification_report

predictions = model.predict(x_test)
test_accuracy = accuracy_score(y_test, predictions)
```

Accuracy
Recall
Precision
F1-Score
Factor 4

94.23%

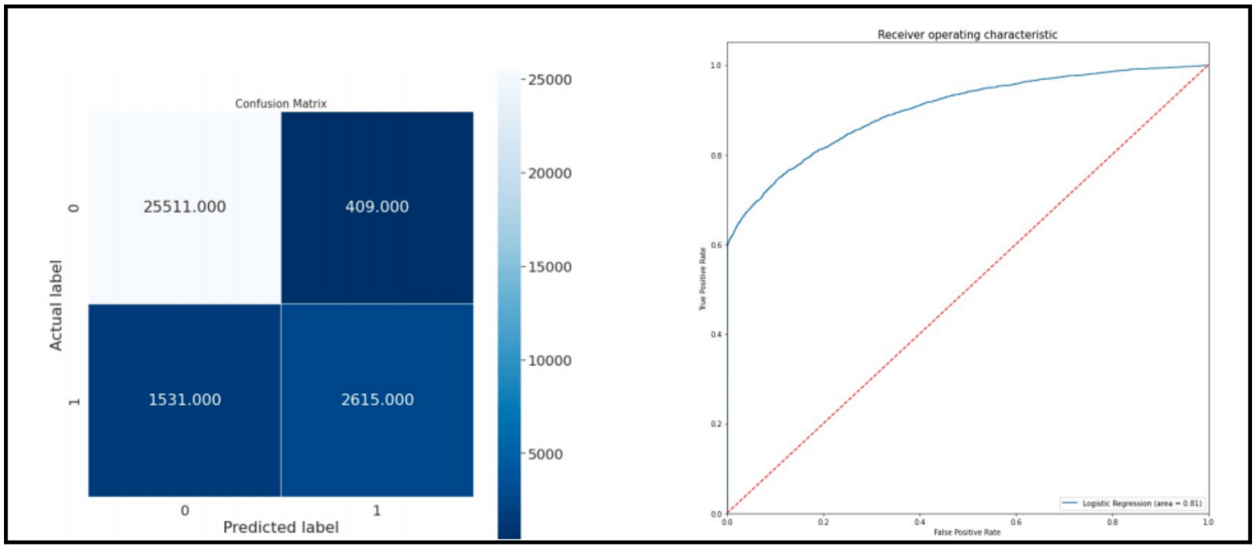
57.89%

74.19%

74.45%

0.36775979

ROC Curve and Confusion Matrix:



ROC Curve and Confusion Matrix for the Logistic Regression

The model will always predict largely better bad debtors over good debtors, because of their proportion in the data. It's inevitable, and regression models can't surpass this limitation, no matter the level of engineering involved.

XGBoost

EXtreme Gradient Boosting (XGBoost) is an optimized and parallelized open source implementation of gradient boosting created by Tianqi Chen, a PhD student at the University of Washington.

XGBoost uses decision trees (like random forest) to solve classification (binary & multi-class), ranking, and regression problems. So, we're in the area of supervised learning algorithms here.

Start by initializing the neptune experiment:

```
import neptune
from neptunecontrib.monitoring.xgboost import neptune_callback

params_attempt3 = {
    'max_depth':10,
    'learning_rate':0.001,
    'colsample_bytree': 0.7,
    'subsample':0.8,
    'gamma': 0.3,
    'alpha':0.35,
    'n_estimator': 100,
    'objective': 'binary:logistic',
    'eval_metric': 'error'
}

neptune.create_experiment(
    name='CreditScoring XGB',
    tags=['XGBoost', 'Credit Scoring'],
    params=params
)
```

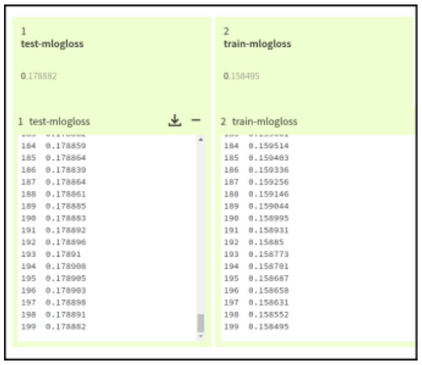
Split the data and Instantiate the DMatrix data loaders:

```
x_train, x_test, y_train, y_test = train_test_split(X, target,
test_size=0.25, random_state=56)

dtrain = xgb.DMatrix(x_train, label=y_train)
dtest = xgb.DMatrix(x_test, label=y_test)
```

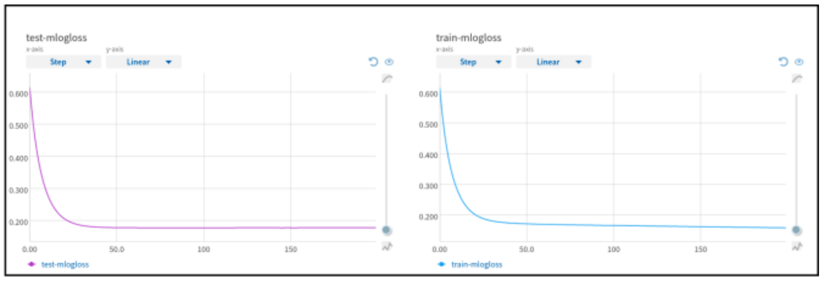
Let's start training the model and track each metric using Neptune XGBoost integration.

```
import xgboost as xgb
import neptune
from neptunecontrib.monitoring.xgboost import neptune_callback

xgb_classifer = xgb.XGBClassifier(**params_attempt3)
xgb_classifer.fit(
    x_train,
    y_train,
    eval_set=[(x_test, y_test)],
    callbacks=[neptune_callback(log_tree=[0, 1])])

neptune.stop()
```

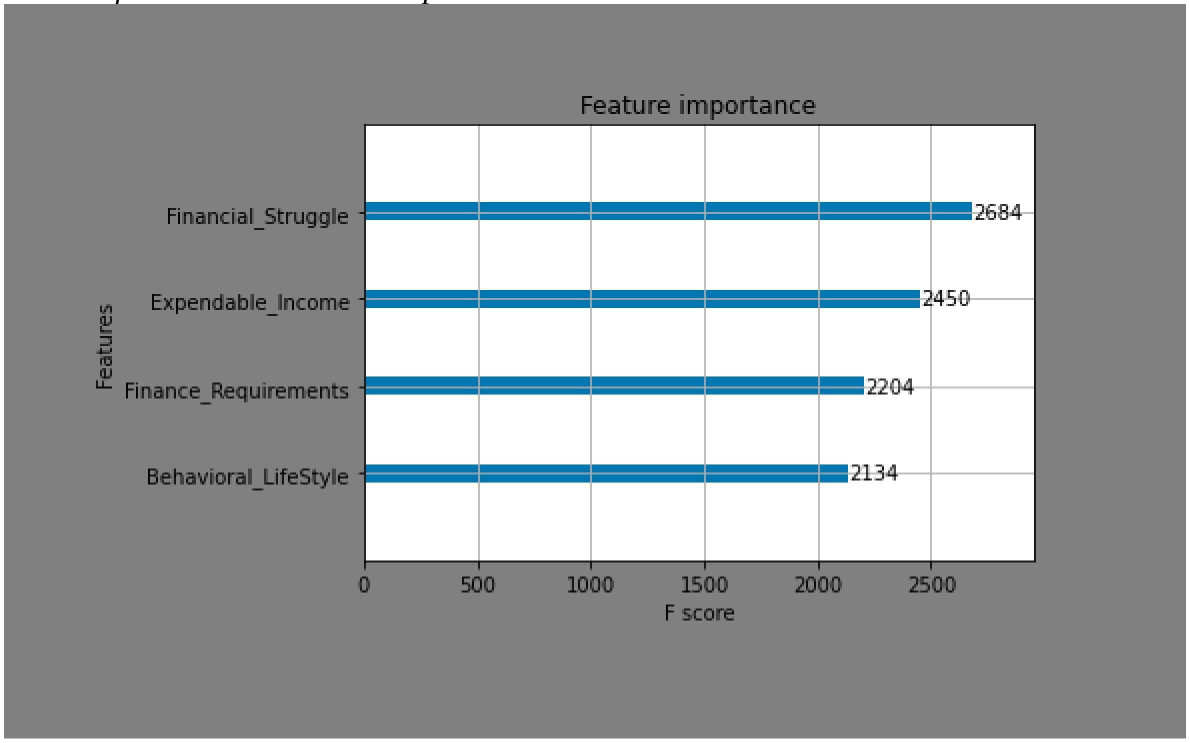
Head back to Neptune to check the loss metrics and the Feature importance graph:



```
1                               2
test-mlogloss                   train-mlogloss

0.178882                        0.158495

1  test-mlogloss                2  train-mlogloss
184  0.178859                   184  0.159514
185  0.178864                   185  0.159403
186  0.178839                   186  0.159336
187  0.178864                   187  0.159256
188  0.178861                   188  0.159146
189  0.178885                   189  0.159044
190  0.178883                   190  0.158995
191  0.178892                   191  0.158931
192  0.178896                   192  0.15885
193  0.17891                    193  0.158773
194  0.178908                   194  0.158781
195  0.178905                   195  0.158687
196  0.178903                   196  0.158658
197  0.178898                   197  0.158631
198  0.178891                   198  0.158552
199  0.178882                   199  0.158495
```



Charts of train and eval loss: Neptune web UI



Feature importance graph for the 4 factors

We can even take a look at the Internal estimators of the model: the Graph of trees that XGBoost uses internally.

Internal XGBoost estimators (click to enlarge)

XGBoost Results:

Accuracy
Recall

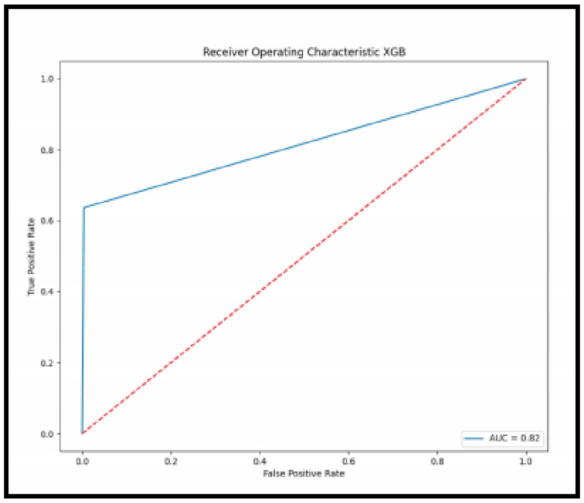
Precision
F1-Score

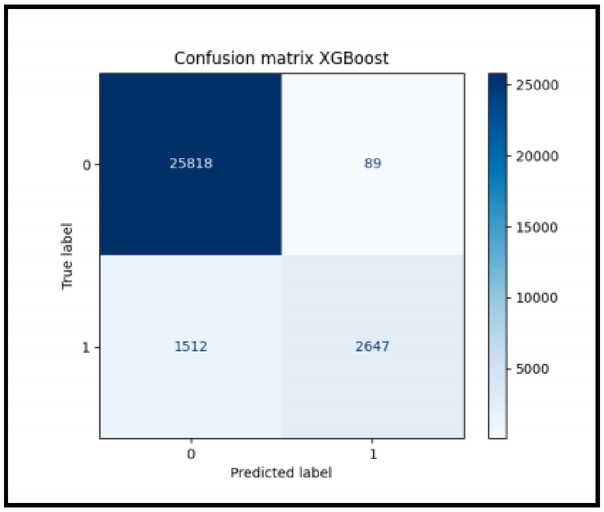
94.68%

55.70%

72.19%

76.78%





ROC Curve and Confusion Matrix for the XGBoost performer

Decision tree

The advantage of decision trees is that they're simple to interpret, very quick to train, non-parametric, and require very little data preprocessing.They can be calculated automatically, by supervised learning algorithms capable of automatically selecting the discriminating variables within unstructured and potentially large data.

We'll be testing a light implementation of Decision Trees for this prepared dataset, and then benchmark the three models to see which performs better.

Train the model

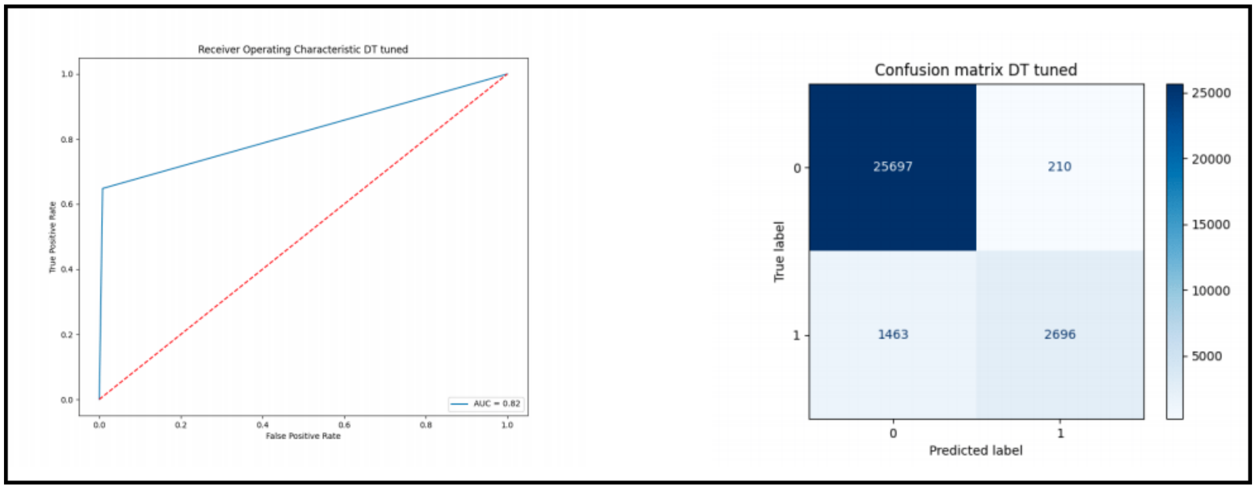
```
from sklearn.tree import DecisionTreeClassifier

classifier = DecisionTreeClassifier()
classifier.fit(x_train, y_train)
```

Results:

```
preds = classifier.predict(x_test)
```

```
print('Accuracy Score: ', metrics.accuracy_score(y_test, preds))
```

Accuracy	Recall	Precision	F1-Score
92.80%	68.84%	69.19%	72.57%



ROC Curve and Confusion Matrix for the Decision Tree model

Benchmark of the three models

After training Logistic Regression (LR), Decision Tree (DT) and eXtreme Gradient Boosting (XGBoost) multiple times to avoid the single-result bias, the best performing iteration of each model will be used to do the comparison.

Model	Accuracy	F1-Score
Logistic regression	94.23%	74.45%
Decision tree		

94.50%

76.73%

XGBoost

94.68%

76.78%

We see that all the models are pretty close in terms of accuracy, even though **XGBoost** did better overall. The most interesting metric is the F1-Score which is better at evaluating the level of confusion of the models in terms of identifying the right classes. **XGBoost** did better at predicting the right classes and differentiating the "*bad*" and "*good*" debtors.