Table of Contents

Bank Loan Default

Goals

- Use past loan, bank account, and transaction data to predict whether a future customer will default the loan.
- The model should be interpretable to explain why the loan is rejected.

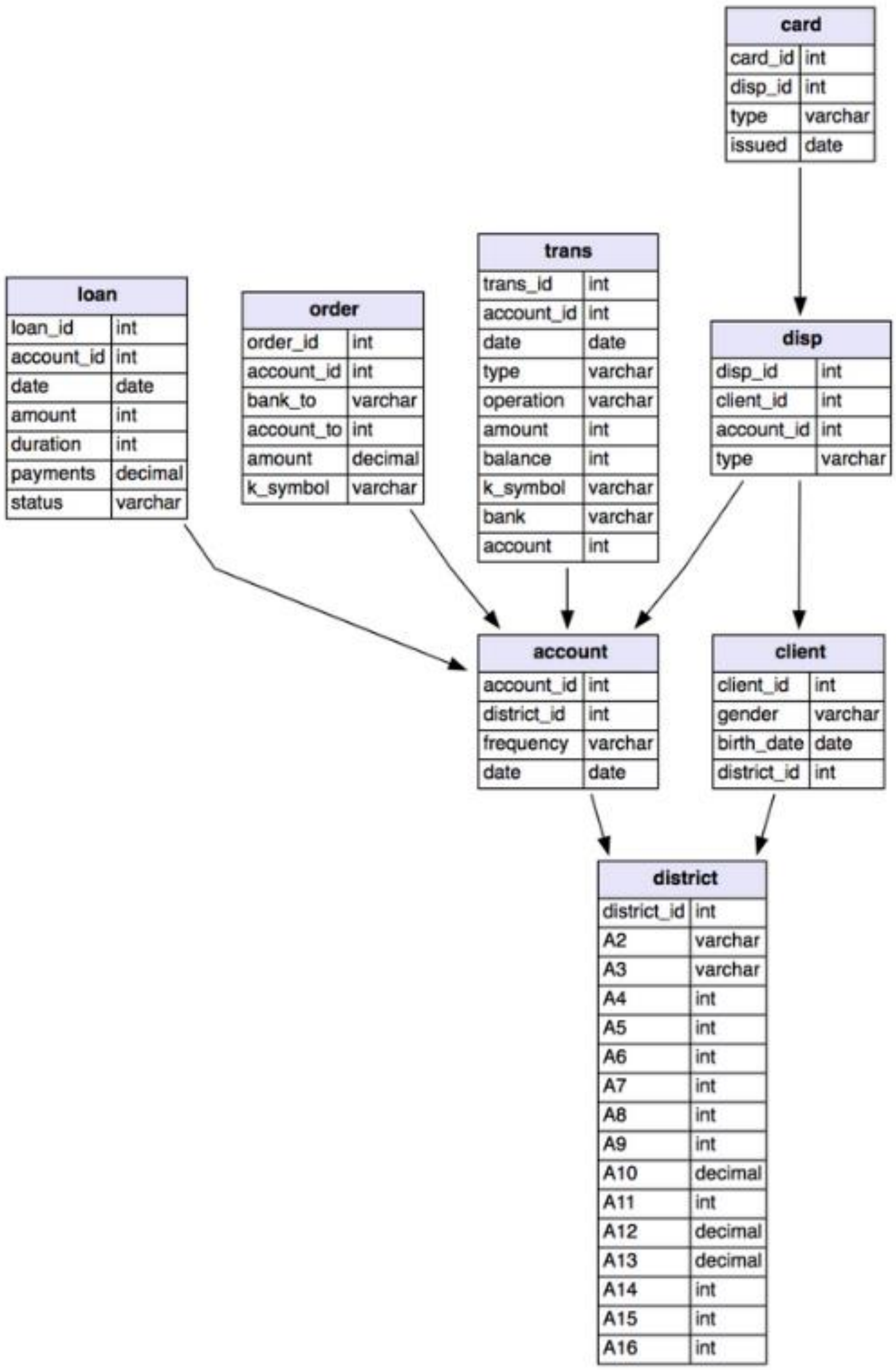
Tools Used

- MySQL
- Python
- Scikit learn

Data

- 8 datasets from a bank collected in 1999
- More than 1 Million Transactions
- 682 Accounts with past and current loans are used to train and test the model
- Imbalanced dataset, about 15% of loans are default

Database Schema



Project Outline

- Prepare the dataset for modeling
- Feature Exploration
- Problems and Solutions
- Train and Test Model

Prepare the dataset for modeling

Create tables

This code block shows an example of creating one of the tables from an ASC file. Statements like this are made for each of the 8 tables.

```
Create DATABASE bank;

use bank;

set global local_infile =1;
# ------------------------------------------
drop table if exists account;

create table account (
  accountId int PRIMARY KEY,
  districtId int,
  foreign key (districtId)
      references district(districtId),
  frequency varchar(20),
  date date
);

load data local infile
'bank_loan\\account.asc'
into table account
    CHARACTER SET 'utf8'
    fields terminated by ';' ENCLOSED BY '"'
    lines terminated by '\n' IGNORE 1 LINES
    (accountId,districtId,frequency,@date)
    set date = str_to_date(@date,'%y%m%d')
;
```

Avoid Data Leakage

It is important to remove information that is not available at prediction time when constructing the dataset for machine learning. In this case, the transaction data after the loan date is removed. A view is created to contain only the loan payment transactions that are before loan date. The balance data needs to be aggregated at account level on a later step.

```
create view transBeforeLoanView as
select t.*
from trans t
join loan l on l.accountId = t.accountId
and t.kSymbol !='UVER'
and t.date < l.date
;
```

Account data consolidated

A view is created to contain all the personal data for each account that took a loan. The features include: gender, age, has disponent, credit card tier, district info.

```
create view defaultView as
select l.*,
       c.gender as owner,
       1999 - year(c.birthdate) as ownerAge,
       If(c2.clientId is not null, 1,0) as hasDisponent,
       ca.type as creditCard,
       dis.*

from loan l
    join disposition d on l.accountId = d.accountId and d.type = 'OWNER'
        join client c on d.clientId = c.clientId
    left join disposition d2 on l.accountId = d2.accountId and d2.type = 'DISPONENT'
        left join client c2 on d2.clientId = c2.clientId
    left join card ca on d.dispId = ca.dispId
    join account a on a.accountId = l.accountId
        join district dis on dis.districtId = a.districtId
;
```

Data Cleaning and Feature Engineering

Load the data from MySQL

The password is saved in environment variable to avoid leaking credentials to GitHub.

```
[51]
from sqlalchemy import create_engine
import pymysql
import pandas as pd
from functools import reduce
```

```
[52]
# Compiling login info
pw = %env DB_PW
DB_TYPE = 'mysql'
DB_DRIVER = 'pymysql'
DB_USER = 'root' # your username in the mysql server
DB_PASS = pw # your password in the mysql server
DB_HOST = 'localhost' # change to hostname of your server if on cloud
DB_PORT = '3306' # change accordingly
DB_NAME = 'bank' # name of your database
POOL_SIZE = 50
```

```
[53]
SQLALCHEMY_DATABASE_URI = f'{DB_TYPE}+{DB_DRIVER}://{DB_USER}:{DB_PASS}@{DB_HOST}:{DB_PORT}/{DB_NAME}'
# Creating engine with login info
engine = create_engine(SQLALCHEMY_DATABASE_URI, pool_size=POOL_SIZE, max_overflow=0)

# this connects to the sql engine
connection = engine.connect()
```

Now Pandas can load the 2 views as dataframe.

```
loan = pd.read_sql('select * from defaultView',con=connection)
loan.head()
```

```
trans = pd.read_sql('select * from transBeforeLoanView',con=connection)
trans.head()
```

Close the connection

```
connection.close()
engine.dispose()
```

Aggregate the transaction data at account level.

5 features are created at this step:

- Minimum Balance
- Average Balance
- Count of times balance is under 5000$
- Count of times balance is under 1000$
- Count of times balance is negative

```
tmp = trans.groupby('accountId').balance.agg(['min','mean'])
tmp['balance5kCount'] = trans[trans['balance']<5000].groupby('accountId').balance.agg('count')
tmp['balance1kCount'] = trans[trans['balance']<1000].groupby('accountId').balance.agg('count')
tmp['balanceNegCount'] = trans[trans['balance']<0].groupby('accountId').balance.agg('count')
tmp.head()
```

Joining the aggregated transaction data to the loan dataframe on account

```
df = pd.merge(loan, tmp, on='accountId', how='left')
df.head()
```

All features

```
Index(['amount', 'duration', 'payments', 'status', 'ownerAge', 'hasDisponent',
       'creditCard', 'inhabitants', 'municipalities', 'municipalities500',
       'municipalities2k', 'municipalities10k', 'cities', 'urbanRatio',
       'avgSalary', 'unemploymentRate95', 'unemploymentRate96',
       'entrepreneurP1k', 'crimes95', 'crimes96', 'min', 'mean',
       'balance5kCount', 'balance1kCount', 'balanceNegCount', 'owner_Male'],
      dtype='object')
```

Remove ID and useless features

```
df = df.drop(columns=['loanId','accountId','date','districtId','name','region'])
```

Encode the target variable: Loan Status

0 = No issue with loan

1 = Defaulted loan

status	status of paying off the loan	'A' stands for contract finished, no problems, 'B' stands for contract finished, loan not payed, 'C' stands for running contract, OK so far, 'D' stands for running contract, client in debt

```
m = {"A": 0, "B": 1, "C": 0, "D": 1}
df['status'] = df['status'].map(m)
```

Label Encode Ordinal Categorical feature: Credit Card

```
df['creditCard'] = df['creditCard'].map({"gold": 3, "classic": 2, "junior": 1})
```

One-hot Encode Binary Categorical feature: Gender, Has Disponent

```
df = pd.get_dummies(df, drop_first=True)
```

Fill missing values

```
df.fillna(0, inplace=True)
df.head()
```

Split into X and y

```
X = df.loc[:, df.columns != "status"]
y = df.loc[:, "status"]
```

Split Train and Test

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, stratify=y,random_state=42)
```

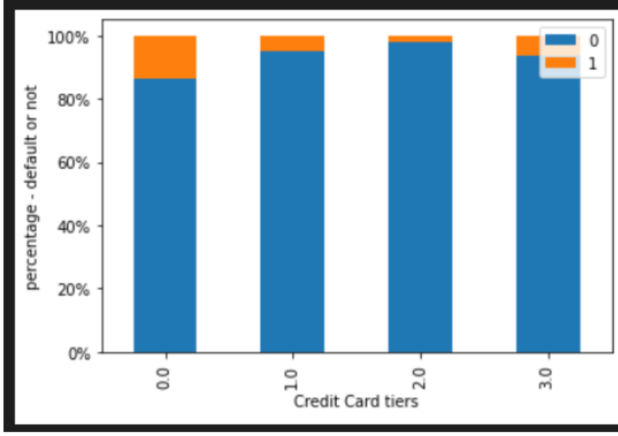
Scale the data

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Exploratory Data Analysis

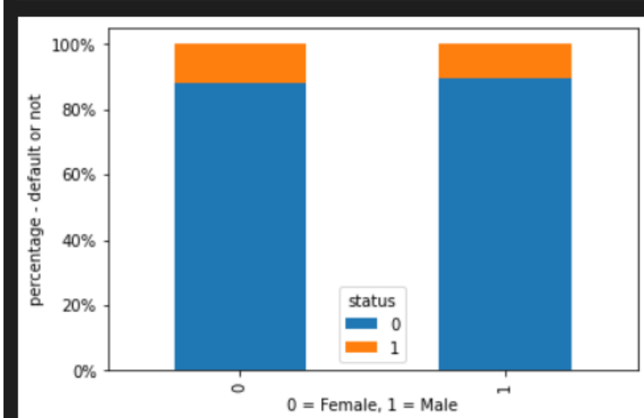
Credit Card: No credit card and highest credit card tier is more likely to default

```
df.groupby(["creditCard", "status"])["status"].size().groupby(level=0).apply(
    lambda x: 100 * x / x.sum()
).unstack().plot(kind="bar", stacked=True)
plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter())
plt.ylabel("percentage - default or not")
plt.xlabel("Credit Card tiers")
plt.legend(loc=1)
plt.show()
```



Gender: No impact

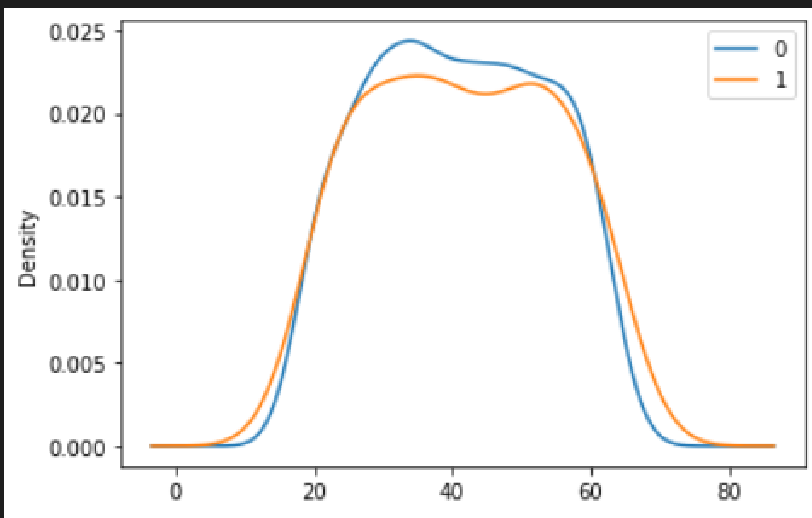
```
df.groupby(["owner_Male", "status"])["status"].size().groupby(level=0).apply(
    lambda x: 100 * x / x.sum()
).unstack().plot(kind="bar", stacked=True)
plt.xlabel("0 = Female, 1 = Male")
plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter())
plt.ylabel("percentage - default or not")
plt.show()
```



Age: No impact

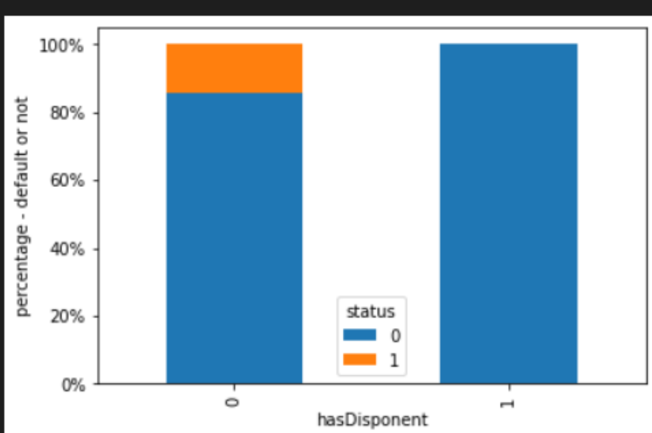
```
df.groupby('status')['ownerAge'].plot(kind='kde',legend=True)
```

```
status
0    AxesSubplot(0.125,0.125;0.775x0.755)
1    AxesSubplot(0.125,0.125;0.775x0.755)
Name: ownerAge, dtype: object
```



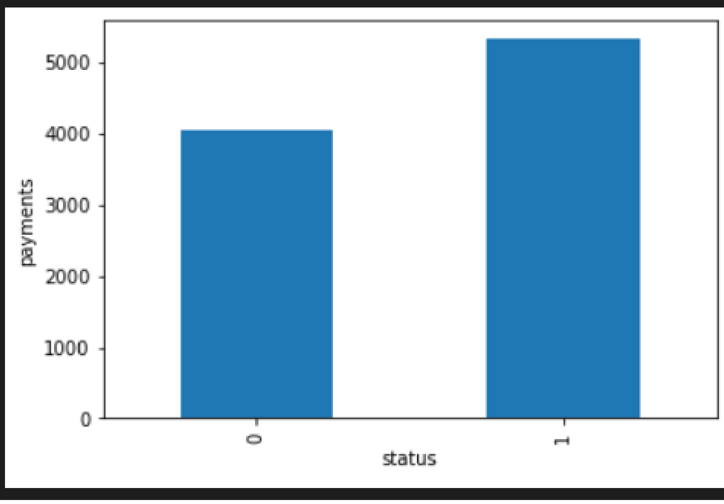
Has Disponent: disponent will not default

```
fig = df.groupby(["hasDisponent", "status"])["status"].size().groupby(level=0).apply(
    lambda x: 100 * x / x.sum()
).unstack().plot(kind="bar", stacked=True)
plt.xlabel("hasDisponent")
plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter())
plt.ylabel("percentage - default or not")
plt.show()
fig.figure.savefig('hasDisponent.png')
```



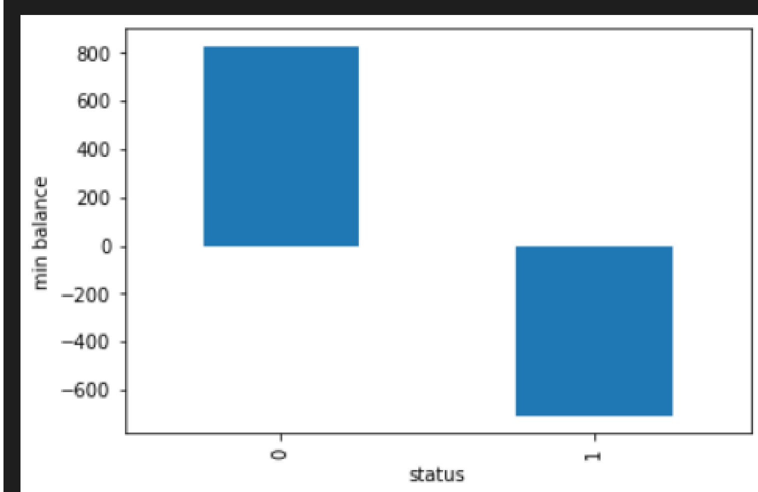
Monthly Payments: Higher payments more likely to default

```
a = df.groupby(
    by="status", axis=0, level=None, as_index=True, sort=True, group_keys=True
)
fig = a['payments'].mean().plot(kind="bar")
plt.ylabel("payments")
fig.figure.savefig('payments.png')
```



Minimum Balance: negative balance more likely to default

```
b = df.groupby(
    by="status", axis=0, level=None, as_index=True, sort=True, group_keys=True
)
fig = b['min'].mean().plot(kind="bar")
plt.ylabel("min balance")
fig.figure.savefig('min balance.png')
```



Benchmark model and Evaluation Metric

Evaluation function

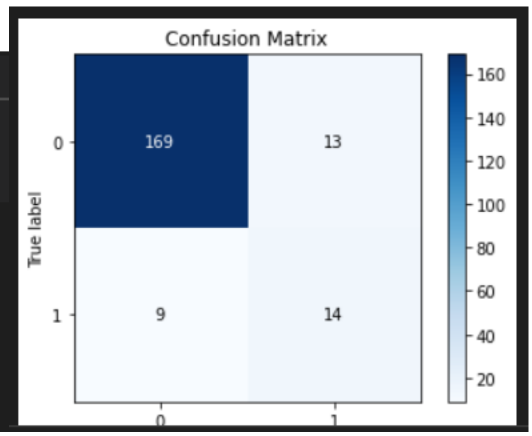
```
def eval(model):
    y_pred = model.predict(X_test_scaled)
    print(classification_report(y_test, y_pred))
    skplt.metrics.plot_confusion_matrix(y_test, y_pred)
```

Benchmark model: Decision Tree

The final model should have higher recall than 14/25 = 61%

```
base = DecisionTreeClassifier(random_state=42)
base.fit(X_train_scaled,y_train)
eval(base)
```

```
              precision    recall  f1-score   support

           0       0.95      0.93      0.94       182
           1       0.52      0.61      0.56        23

    accuracy                           0.89       205
   macro avg       0.73      0.77      0.75       205
weighted avg       0.90      0.89      0.90       205
```



Metric

The important metric is recall of class 1, because bank loses a lot of money when a customer defaults the loan

A potentially important metric is recall of class 0 which represent the loss of profit for rejecting a potential customer due to False Positive

Ideally, business stake holder can provide a number for the profit on loan to calculate the loss functon by combining the 2 metrics above.

Example, assume bank profit 1% on the loan, and a customer apply for 10k$ loan and only pays back 50%

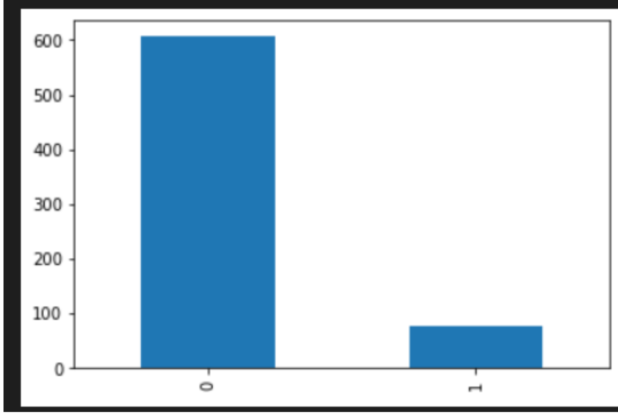
Loss when predict default on good loan = (100% - 93% recall 0) x 1% profit x 10k$ = 7$

Loss when approve a bad loan = (100% - 61% recall 1) x 10k$ x 50% payment = 1950$

Imbalanced Dataset

Most of the loans are good loans, only about 15% is defaulted. The classifiers can struggle with this dataset because it can achieve high accuracy by simply predicting all loans will be good. We are trying to classify the bad loans so we want as many bad loans as possible.

```python
# imbalanced dataset
fig = y.value_counts().plot(kind='bar');
fig.figure.savefig('status.png')
```



One solution to imbalanced data is Oversampling. The Imblearn library provides many algorithms. One popular method is Synthetic Minority Oversampling Technique, or SMOTE in short.

SMOTE first selects a minority class sample at random and then finds its K nearest neighbors. It then connects a line in the feature space between the sample and a random nearest neighbors. The new sample is generated as a random point on this line.

Another method is Adaptive Synthetic Sampling, ADASYN in short. ADASYN involves generating more samples in the regions of feature space where the density of minority class is low and fewer where the density is high.

Comparing 2 Oversampling method: ADASYN vs SMOTE

ADASYN: 65% > 61% base

```
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE, ADASYN

# define pipeline
over = ADASYN()
steps = [('over', over), ('model', base)]
pipeline = Pipeline(steps=[('over', over), ('model', base)])
# evaluate pipeline
pipeline.fit(X_train_scaled,y_train)
eval(pipeline)
```

```
              precision    recall  f1-score   support

           0       0.95      0.90      0.93       182
           1       0.45      0.65      0.54        23

    accuracy                           0.87       205
   macro avg       0.70      0.78      0.73       205
weighted avg       0.90      0.87      0.88       205
```

SMOTE: 70% > 65% ADASYN

```
# define pipeline
over = SMOTE()
steps = [('over', over), ('model', base)]
pipeline = Pipeline(steps=[('over', over), ('model', base)])
# evaluate pipeline
pipeline.fit(X_train_scaled,y_train)
eval(pipeline)
```

```
              precision    recall  f1-score   support

           0       0.96      0.92      0.94       182
           1       0.52      0.70      0.59        23

    accuracy                           0.89       205
   macro avg       0.74      0.81      0.77       205
weighted avg       0.91      0.89      0.90       205
```

Conclusion: SMOTE should be used for final model

```
sm = SMOTE(random_state=42)
X_res, y_res = sm.fit_resample(X_train_scaled, y_train)
```

Grid Search and K-fold validation

When tuning model with many hyper parameters, it is easy to find the optimal value with Grid Search which tries all the combinations of the hyper parameters chosen.

K-fold validation is used in conjunction with Grid Search to assure that the result of each Grid Search is tested multiple times on different subset of the train data.

However, this process can be very slow depending on the complexity of the algorithm chosen. Therefore it is important to set n_jobs to -1 to run it in parallel using all the available CPU cores.

Try out different algorithms

Logistic Regression: 83%

```
from sklearn.linear_model import LogisticRegression
lr =LogisticRegression(C=1000)
lr.fit(X_res,y_res)
eval(lr)
```

```
           0       0.97      0.87      0.92       182
           1       0.44      0.78      0.56        23

    accuracy                           0.86       205
   macro avg       0.70      0.83      0.74       205
weighted avg       0.91      0.86      0.88       205
```

Decision Tree: 74%

```
max_depth = range(2,20,2)

parameters = {
    'max_depth': max_depth,
    'criterion': ['gini', 'entropy'],
    'min_samples_split': range(2, 100,10)
}

tree = DecisionTreeClassifier(random_state=42,class_weight='balanced')
gs = GridSearchCV(tree, parameters, cv=StratifiedKFold(n_splits=5), scoring='recall', n_jobs=-1)
gs.fit(X_res,y_res)
print(gs.best_params_)
eval(gs.best_estimator_)
```

```
{'criterion': 'entropy', 'max_depth': 12, 'min_samples_split': 2}
              precision    recall  f1-score   support

           0       0.97      0.91      0.94       182
           1       0.52      0.74      0.61        23

    accuracy                           0.89       205
   macro avg       0.74      0.83      0.77       205
weighted avg       0.91      0.89      0.90       205
```

Random Forest: 70%

```
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}

rf = RandomForestClassifier(class_weight='balanced')
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid,scoring='f1', n_iter = 100, cv = StratifiedKFold
(n_splits=3), verbose=2, random_state=42, n_jobs = -1)
rf_random.fit(X_res, y_res)
print(rf_random.best_params_)
eval(rf_random.best_estimator_)
```

Gradient Boosting Classifier: 61%

```
parameters = {
    'n_estimators' : [10, 50, 100, 500, 1000, 5000]
}

gb = GradientBoostingClassifier(random_state=42)

gs = GridSearchCV(gb, parameters, cv=StratifiedKFold(n_splits=5), scoring='f1', n_jobs=-1)
gs.fit(X_res, y_res)
print(gs.best_params_)
eval(gs.best_estimator_)
```

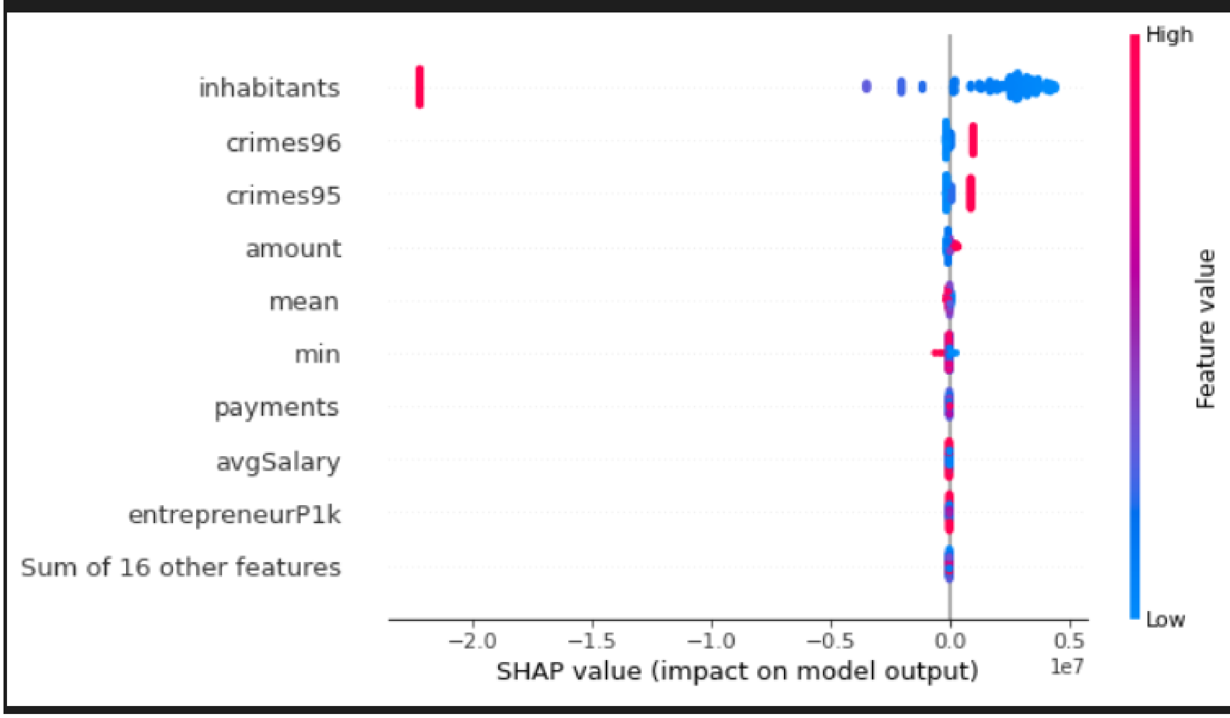
```
{'n_estimators': 500}
              precision    recall  f1-score   support

           0       0.95      0.94      0.94       182
           1       0.56      0.61      0.58        23

    accuracy                           0.90       205
   macro avg       0.76      0.77      0.76       205
weighted avg       0.91      0.90      0.90       205
```

Best Model: Logistic Regression

Model	Recall
Benchmark	61%
Oversample - ADASYN	65%
Oversample - SMOTE	70%
Logistic Regression	83%
Decision Tree	74%
Random Forest	70%
Gradient Boost	61%

Feature Importance: using SHAP

```
explainer = shap.Explainer(lr, X_train, feature_names=X.columns)
shap_values = explainer(X_test)
shap.plots.beeswarm(shap_values)
```



The top 3 features are inhabitants, crimes96 and crimes 95.