Introduction

Just1a2Noob

Table of contents

This is a Quarto document analyzing factors that contribute to credit risk. This analysis is mostly used for learning purposes only.

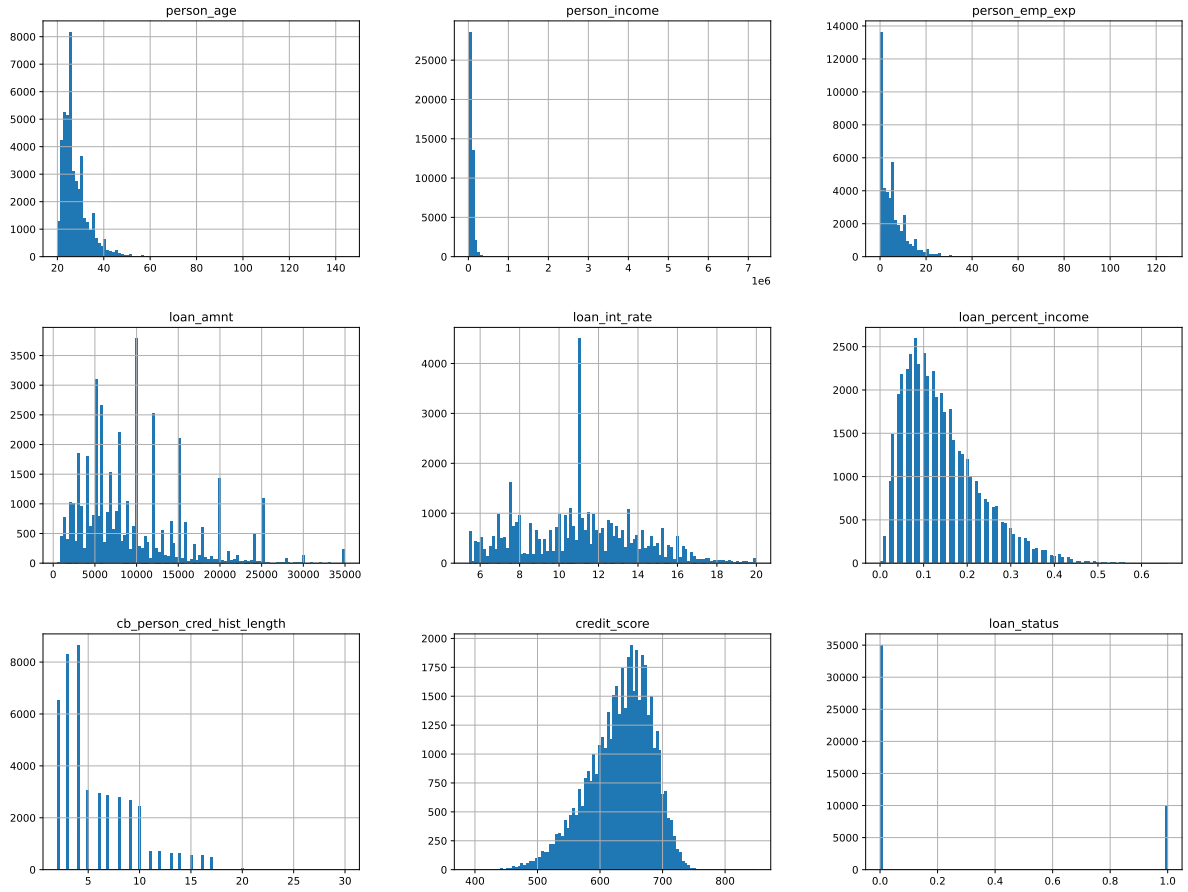
1 Setting Up Our Environment

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

df = pd.read_csv("loan_data.csv")
df.head()
```

	person_age	person_gender	person_education	person_income	person_emp_exp	person_home_ow
0	22.0	female	Master	71948.0	0	RENT
1	21.0	female	High School	12282.0	0	OWN
2	25.0	female	High School	12438.0	3	MORTGAGE

	person_age	person_gender	person_education	person_income	person_emp_exp	person_home_ow
3	23.0	female	Bachelor	79753.0	0	RENT
4	24.0	male	Master	66135.0	1	RENT

```
import matplotlib.pyplot as plt

df.hist(bins=100, figsize=(20, 15))
plt.show()
```



The graphs above gives us graph of all numerical columns in terms of frequency. From a glance we can see that the column's

```
from sklearn.model_selection import train_test_split, StratifiedShuffleSplit

# This if for splitting the data
```

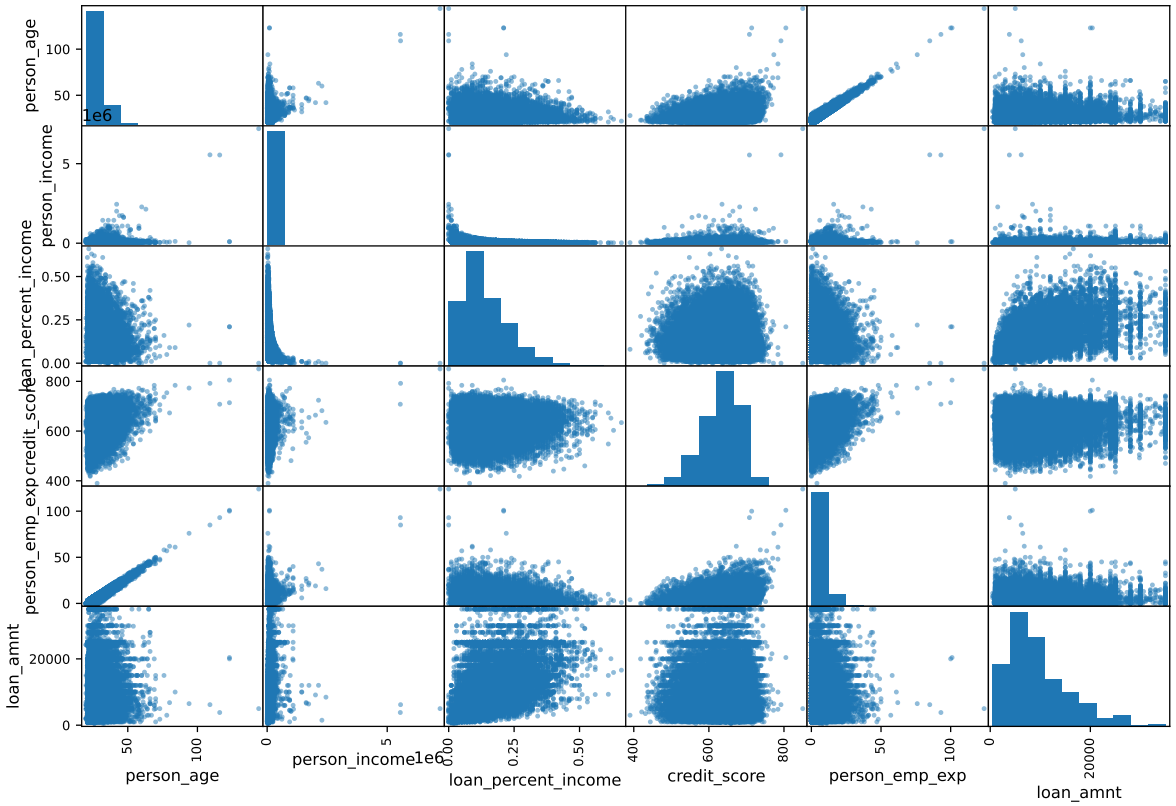
```
train_set, test_set = train_test_split(df, test_size=0.2)

# stratified sampling on the target
stratified_shuffle = StratifiedShuffleSplit(n_splits=1, test_size=0.2)
for train_index, test_index in stratified_shuffle.split(df, df["loan_status"]):
    strat_train_set = df.loc[train_index]
    strat_test_set = df.loc[test_index]

loan_status = strat_train_set.copy()
```

2 Exploratory Data Analysis

```
from pandas.plotting import scatter_matrix

attributes = [
    "person_age",
    "person_income",
    "loan_percent_income",
    "credit_score",
    "person_emp_exp",
    "loan_amnt",
]
scatter_matrix(loan_status[attributes], figsize=(12, 8))
```

```
array([[<Axes: xlabel='person_age', ylabel='person_age'>,
        <Axes: xlabel='person_income', ylabel='person_age'>,
        <Axes: xlabel='loan_percent_income', ylabel='person_age'>,
        <Axes: xlabel='credit_score', ylabel='person_age'>,
        <Axes: xlabel='person_emp_exp', ylabel='person_age'>,
        <Axes: xlabel='loan_amnt', ylabel='person_age'>],
       [<Axes: xlabel='person_age', ylabel='person_income'>,
        <Axes: xlabel='person_income', ylabel='person_income'>,
        <Axes: xlabel='loan_percent_income', ylabel='person_income'>,
        <Axes: xlabel='credit_score', ylabel='person_income'>,
        <Axes: xlabel='person_emp_exp', ylabel='person_income'>,
        <Axes: xlabel='loan_amnt', ylabel='person_income'>],
       [<Axes: xlabel='person_age', ylabel='loan_percent_income'>,
        <Axes: xlabel='person_income', ylabel='loan_percent_income'>,
        <Axes: xlabel='loan_percent_income', ylabel='loan_percent_income'>,
```

```
 <Axes: xlabel='credit_score', ylabel='loan_percent_income'>,
 <Axes: xlabel='person_emp_exp', ylabel='loan_percent_income'>,
 <Axes: xlabel='loan_amnt', ylabel='loan_percent_income'>],
[<Axes: xlabel='person_age', ylabel='credit_score'>,
 <Axes: xlabel='person_income', ylabel='credit_score'>,
 <Axes: xlabel='loan_percent_income', ylabel='credit_score'>,
 <Axes: xlabel='credit_score', ylabel='credit_score'>,
 <Axes: xlabel='person_emp_exp', ylabel='credit_score'>,
 <Axes: xlabel='loan_amnt', ylabel='credit_score'>],
[<Axes: xlabel='person_age', ylabel='person_emp_exp'>,
 <Axes: xlabel='person_income', ylabel='person_emp_exp'>,
 <Axes: xlabel='loan_percent_income', ylabel='person_emp_exp'>,
 <Axes: xlabel='credit_score', ylabel='person_emp_exp'>,
 <Axes: xlabel='person_emp_exp', ylabel='person_emp_exp'>,
 <Axes: xlabel='loan_amnt', ylabel='person_emp_exp'>],
[<Axes: xlabel='person_age', ylabel='loan_amnt'>,
 <Axes: xlabel='person_income', ylabel='loan_amnt'>,
 <Axes: xlabel='loan_percent_income', ylabel='loan_amnt'>,
 <Axes: xlabel='credit_score', ylabel='loan_amnt'>,
 <Axes: xlabel='person_emp_exp', ylabel='loan_amnt'>,
 <Axes: xlabel='loan_amnt', ylabel='loan_amnt'>]], dtype=object)
```



The graph above is a correlation matrix shown in terms of graphs. The columns used are age, income, loan percent income, credit score, employment experience, and loan amount. From the graph we can see that multiple graphs are scattered making it hard to find any meaningful patterns just from graphs alone. But we can see outliers and anomalies from these graphs.

We can see an anomaly in the data in the age group there are some people who are more than 80 years old.

```
df.loc[df['person_age'] >= 80]
```

	person_age	person_gender	person_education	person_income	person_emp_exp	person_hom
81	144.0	male	Bachelor	300616.0	125	RENT
183	144.0	male	Associate	241424.0	121	MORTGAG
575	123.0	female	High School	97140.0	101	RENT
747	123.0	male	Bachelor	94723.0	100	RENT
32297	144.0	female	Associate	7200766.0	124	MORTGAG
32416	94.0	male	High School	29738.0	76	RENT
32422	80.0	male	High School	77894.0	62	RENT

	person_age	person_gender	person_education	person_income	person_emp_exp	person_hom
32506	84.0	male	High School	114705.0	61	MORTGAG
37930	116.0	male	Bachelor	5545545.0	93	MORTGAG
38113	109.0	male	High School	5556399.0	85	MORTGAG

Also, we see an outlier in the person employed experience, some reaching above 100.

```
df.loc[df['person_emp_exp'] >= 100]
```

	person_age	person_gender	person_education	person_income	person_emp_exp	person_hom
81	144.0	male	Bachelor	300616.0	125	RENT
183	144.0	male	Associate	241424.0	121	MORTGAG
575	123.0	female	High School	97140.0	101	RENT
747	123.0	male	Bachelor	94723.0	100	RENT
32297	144.0	female	Associate	7200766.0	124	MORTGAG

And finally individuals who have an abnormally large income and abnormally low income we are going to ignore that and accept them because they are within the realm of possibilities but at the extreme level.

3 Preparing for Data For Machine Learning

I am gonna apply both OrdinalEncoder, OneHotEncoder, and StandardScaler to our inputs data set.

```
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder, StandardScaler


class Convert_Encoder(BaseEstimator, TransformerMixin):
    # Converts previous_loan_defaults_on_file column to binary
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        enc = OrdinalEncoder(categories=[["No", "Yes"]])
        X["previous_loan_defaults_on_file"] = enc.fit_transform(
```

```
            X[["previous_loan_defaults_on_file"]]
        ).astype(int)

        return X


class Convert_Categorical(BaseEstimator, TransformerMixin):
    # Apply OneHotEncoder to every categorical column
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        categorical_columns = [
            "person_gender",
            "person_education",
            "person_home_ownership",
            "loan_intent",
        ]
        ohe = OneHotEncoder()

        for col in categorical_columns:
            unique_values = X[col].unique()
            column_names = [str(value) for value in unique_values]

            matrix = ohe.fit_transform(X[[col]]).toarray()

            for i in range(len(matrix.T)):
                X[str(col) + "_" + column_names[i]] = matrix.T[i]

            X = X.drop([col], axis=1)

        return X


class CustomStandardScaler(BaseEstimator, TransformerMixin):
    # Applies StandardScaler and keeps the column names
    def __init__(self):
        # Initialize the standard scaler
        self.scaler = StandardScaler()
        # Store column names for later use
        self.columns = None
```

```
    def fit(self, X, y=None):
        # Store column names before scaling
        self.columns = X.columns
        # Fit the scaler
        self.scaler.fit(X)
        return self

    def transform(self, X):
        # Transform the data
        scaled_data = self.scaler.transform(X)
        # Convert back to DataFrame with original column names
        return pd.DataFrame(scaled_data, columns=self.columns, index=X.index)


X_inputs = strat_test_set.drop("loan_status", axis=1)
y_labels = strat_test_set["loan_status"].copy()

pipe = Pipeline(
    [
        ("Converting yes/no column", Convert_Encoder()),
        ("Converts categorical columns", Convert_Categorical()),
        ("Std scaler", CustomStandardScaler()),
    ]
)

X_transformed = pipe.fit_transform(X_inputs)

X_transformed.head()
```

	person_age	person_income	person_emp_exp	loan_amnt	loan_int_rate	loan_percent_incon
27055	0.495876	-0.851175	0.719680	-1.017791	0.286885	-0.441984
4149	-0.629166	0.308242	-0.729917	-0.808169	2.705893	-1.014899
25935	-0.147005	0.576010	0.075415	-0.396836	-1.866705	-0.900316
29819	2.103078	-0.467440	2.008211	-0.871451	0.757247	-0.671150
15480	-0.468446	1.576508	-0.085651	-0.238631	0.246568	-1.014899

4 Training Models

Our first model is linear regression. Below is the results of using Linear Regression model:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

lin_reg = LinearRegression()
lin_reg.fit(X_transformed, y_labels)
linear_pred = lin_reg.predict(X_transformed)
linear_rmse = mean_squared_error(y_labels, linear_pred)
print(f"The RMSE of Linear regression is: {linear_rmse}")
```

```
The RMSE of Linear regression is: 0.0902752783142164
```

And below is Decision Tree regression:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(X_transformed, y_labels)
tree_pred = tree_reg.predict(X_transformed)
tree_rmse = mean_squared_error(y_labels, tree_pred)
print(f"The RMSE of DecisionTree regression is: {tree_rmse}")
```

```
The RMSE of DecisionTree regression is: 0.0
```

We can instantly notice that the Decision Tree has an abnormal error rate of 0. This means it is overfitting that data to solve this we are going to use cross validation. For that we are going to use Scikit-Learn's K-fold cross validation feature.

```
from sklearn.model_selection import cross_val_score

lin_scores = cross_val_score(
    lin_reg, X_transformed, y_labels, scoring="neg_mean_squared_error", cv=10
)
tree_scores = cross_val_score(
  tree_reg, X_transformed, y_labels, scoring="neg_mean_squared_error", cv=10
)


def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
```

```
    print("Standard deviation:", scores.std())


linear_rmse_scores = np.sqrt(-lin_scores)
tree_rmse_scores = np.sqrt(-tree_scores)

print("------LinearRegression-------")
display_scores(linear_rmse_scores)
print("\n")
print("------DecisionTreeRegressor-------")
display_scores(tree_rmse_scores)
```

```
------LinearRegression-------
Scores: [0.30061631 0.30376203 0.30006671 0.29522971 0.29540808 0.3037271
 0.29445924 0.31356885 0.31015947 0.29458556]
Mean: 0.30115830557174683
Standard deviation: 0.006373365714970501


------DecisionTreeRegressor-------
Scores: [0.34801022 0.3197221  0.35276684 0.3197221  0.31622777 0.33499585
 0.3        0.34318767 0.31797973 0.30368112]
Mean: 0.3256293406078219
Standard deviation: 0.017287737522545706
```

4.1 Ensemble Models

We are going to use simple ensemble models with the purpose of seeing whether using ensemble methods are better compared to linear regression and decision tree.

The ensemble method we are going to use is Random Forests which works by training many Decision Trees on random subsets of the features, then averaging out their predictions.

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(X_transformed, y_labels)

forest_pred = forest_reg.predict(X_transformed)
forest_mse = mean_squared_error(y_labels, forest_pred)
forest_rsme = np.sqrt(forest_mse)
print(f"This is the RMSE of Forest Regression: {forest_rsme}")
```

```
This is the RMSE of Forest Regression: 0.08905659873236671
```

```
forest_scores = cross_val_score(forest_reg, X_transformed, y_labels, scoring="neg_mean_squar
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

```
Scores: [0.23834312 0.23653212 0.25256154 0.22132053 0.23961636 0.23809755
 0.23379882 0.25218181 0.2523177  0.23505602]
Mean: 0.23998255893290818
Standard deviation: 0.009424805757616995
```

From the looks of it Random Forests looks very promising. However, not that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set. We can fine-tune Forest Regressor using the `RandomizedSearchCV` class from Scikit-Learn which searches a range of parameters and finds the best parameters.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

parameter_distribs = {
  'n_estimators': randint(low=1, high=200),
  'max_features': randint(low=1, high=20)
}

forest_random_search = RandomizedSearchCV(
  forest_reg,
  param_distributions=parameter_distribs,
  n_iter=10,
  cv=5,
  scoring="neg_mean_squared_error",
)
forest_random_search.fit(X_transformed, y_labels)

cvres = forest_random_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
  print(np.sqrt(-mean_score), params)
```

```
0.2378803513148846 {'max_features': 11, 'n_estimators': 138}
0.2380446362513181 {'max_features': 8, 'n_estimators': 73}
0.24010974917111194 {'max_features': 19, 'n_estimators': 107}
0.23986856572029736 {'max_features': 6, 'n_estimators': 74}
```

```
0.30640183773567753 {'max_features': 1, 'n_estimators': 7}
0.27418827288449216 {'max_features': 1, 'n_estimators': 152}
0.2575673061653865 {'max_features': 2, 'n_estimators': 95}
0.2387747954970786 {'max_features': 12, 'n_estimators': 102}
0.24913440599357292 {'max_features': 8, 'n_estimators': 11}
0.23828648454737988 {'max_features': 12, 'n_estimators': 156}
```

From here we can see the best parameters of Random Forest model, we can also see each feature.

```
importances = forest_random_search.best_estimator_.feature_importances_

feature_importance_df = pd.DataFrame({
  'feature': X_transformed.columns,
  'importance': importances,
})
feature_importance_df = feature_importance_df.sort_values('importance', ascending=False)

feature_importance_df
```

	feature	importance
8	previous_loan_defaults_on_file	0.249553
4	loan_int_rate	0.171525
5	loan_percent_income	0.152376
1	person_income	0.114120
7	credit_score	0.060044
3	loan_amnt	0.042885
19	person_home_ownership_OTHER	0.038696
0	person_age	0.025213
2	person_emp_exp	0.021623
6	cb_person_cred_hist_length	0.020189
16	person_home_ownership_OWN	0.015074
18	person_home_ownership_MORTGAGE	0.012886
22	loan_intent_MEDICAL	0.010247
20	loan_intent_HOMEIMPROVEMENT	0.009333
25	loan_intent_VENTURE	0.007856
23	loan_intent_DEBTCONSOLIDATION	0.007742
24	loan_intent_PERSONAL	0.005677
21	loan_intent_EDUCATION	0.005352
11	person_education_Associate	0.005070
14	person_education_Master	0.004860

	feature	importance
12	person_education_Bachelor	0.004853
10	person_gender_male	0.004630
9	person_gender_female	0.004360
15	person_education_Doctorate	0.004071
13	person_education_High School	0.001396
17	person_home_ownership_RENT	0.000369

5 Evaluate Models

After cleaning and fine-tuning our Random Forest we now evaluate models using the test set prepared beforehand.

```
final_model = forest_random_search.best_estimator_

X_test = strat_test_set.drop("loan_status", axis=1)
Y_test = strat_test_set["loan_status"].copy()

X_test_prepared = pipe.fit_transform(X_test)
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(Y_test, final_predictions)
final_rmse = np.sqrt(final_mse)

final_rmse
```

```
0.08737499948040914
```

Here we can see the predicted RMSE of the Random Forest with the best parameters we had found. Additionally we can compute the a 95% confidence interval for the test RMSE:

```
from scipy import stats

confidence = 0.95

squared_errors = (final_predictions - Y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors),
```

```
                            )
        )
```

```
array([0.0848874 , 0.08979371])
```