Credit Risk Project

by Mikhail Mitrofanov, Aleksandr Egoshin

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 61169 entries, 0 to 61168
Data columns (total 22 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   issue_d                 61169 non-null  object 
 1   purpose                 61169 non-null  object 
 2   addr_state              61169 non-null  object 
 3   sub_grade               61169 non-null  object 
 4   home_ownership          61169 non-null  object 
 5   emp_title               57304 non-null  object 
 6   dti                     61169 non-null  float64
 7   funded_amnt             61169 non-null  int64  
 8   annual_inc              61169 non-null  float64
 9   emp_length              58495 non-null  float64
 10  term                    61169 non-null  int64  
 11  inq_last_6mths          61169 non-null  int64  
 12  mths_since_recent_inq   47640 non-null  float64
 13  delinq_2yrs             61169 non-null  int64  
 14  chargeoff_within_12_mths  61169 non-null  int64  
 15  num_accts_ever_120_pd   49228 non-null  float64
 16  num_tl_90g_dpd_24m      49228 non-null  float64
 17  acc_open_past_24mths    53283 non-null  float64
 18  avg_cur_bal             49224 non-null  float64
 19  tot_hi_cred_lim         49228 non-null  float64
 20  delinq_amnt             61169 non-null  int64  
 21  def                     61169 non-null  int64  
dtypes: float64(9), int64(7), object(6)
memory usage: 10.3+ MB
```

Variable Name Description

- **issue_d** The month which the loan was funded
- **purpose** A category provided by the borrower for the loan request
- **addr_state** The state provided by the borrower in the loan application
- **sub_grade** External assigned loan subgrade
- **home_ownership** The home ownership status provided by the borrower during registration or obtained from the credit report. Our values are: RENT, OWN, MORTGAGE, OTHER
- **emp_title** The job title supplied by the Borrower when applying for the loan
- **dti** A ratio calculated using the borrower's total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan, divided by the borrower's self-reported monthly income
- **funded_amnt** The total amount committed to that loan at that point in time
- **annual_inc** The self-reported annual income provided by the borrower during registration
- **emp_length** Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years
- **term** The number of payments on the loan. Values are in months and can be either 36 or 60
- **inq_last_6mths** The number of inquiries in past 6 months (excluding auto and mortgage inquiries)
- **mths_since_recent_inq** Months since most recent inquiry
- **delinq_2yrs** The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years
- **chargeoff_within_12_mths** Number of charge-offs within 12 months
- **num_accts_ever_120_pd** Number of accounts ever 120 or more days past due
- **num_tl_90g_dpd_24m** Number of accounts 90 or more days past due in last 24 months
- **acc_open_past_24mths** Number of trades opened in past 24 months
- **avg_cur_bal** Average current balance of all accounts
- **tot_hi_cred_lim** Total high credit/credit limit
- **delinq_amnt** The past-due amount owed for the accounts on which the borrower is now delinquent
- **def** Default indicator

Our variable of interst is a binary outcome variable. When dealing with such a case, there are several machine learning algorithms commonly used for prediction. Here are some of the most commonly used algorithms for binary classification tasks:

1. Logistic Regression
2. Decision Trees and Random Forest
3. Support Vector Machines (SVM)
4. K-Nearest Neighbors (KNN)
5. Naive Bayes
6. Neural Networks (Deep Learning)
7. Gradient Boosting Algorithms
8. Logistic Regression with Regularization

The choice of the most suitable algorithm will depend on the specific characteristics of our dataset, the complexity of the problem. It's often a good practice to try multiple algorithms and compare their performance using cross-validation to select the one that works best for our binary classification task.

We are going to start with comparing the following three algorithms:

- **Logistic Regression**

Logistic regression is a classic algorithm for binary classification. It models the probability of the binary outcome based on a linear combination of predictor variables and applies the logistic function to the result.

- **a Decision Tree** (most likely a Random Forest)

Decision trees can be used for binary classification by splitting the dataset into subsets based on feature values and creating a tree-like structure to make predictions. Random Forest is an ensemble method based on several decision trees.

- **a Gradient Boosting Algorithms**

In addition to decision trees, gradient boosting algorithms like XGBoost, LightGBM, and CatBoost can be used directly for binary classification tasks. They are known for their high predictive accuracy and robustness.

```
[ ] income_brackets = [-1, 50000, 100000, 150000, float('inf')]
    income_labels = ['Low', 'Medium', 'High', 'Very High']
    df['income_category'] = pd.cut(df['annual_inc'], bins=income_brackets, labels=income_labels)
    df['income_category'].value_counts()
```

```
Medium       30397
Low          20821
High          7351
Very High     2600
Name: income_category, dtype: int64
```

```
[ ] emp_length_brackets = [-1, 1, 3, 5, 10, float('inf')]
    emp_length_labels = ['<1yr', '1-3yrs', '3-5yrs', '5-10yrs', '>10yrs']
    df['emp_length_category'] = pd.cut(df['emp_length'], bins=emp_length_brackets, labels=emp_length_labels)
    df['emp_length_category'].value_counts()
```

```
5-10yrs    32044
1-3yrs     10122
<1yr        8322
3-5yrs      8007
>10yrs         0
Name: emp_length_category, dtype: int64
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 61169 entries, 0 to 61168
Data columns (total 24 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   issue_d                     61169 non-null  object
 1   purpose                     61169 non-null  object
 2   addr_state                  61169 non-null  object
 3   sub_grade                   61169 non-null  object
 4   home_ownership              61169 non-null  object
 5   emp_title                   57304 non-null  object
 6   dti                         61169 non-null  float64
 7   funded_amnt                 61169 non-null  int64
 8   annual_inc                  61169 non-null  float64
 9   emp_length                  58495 non-null  float64
 10  term                        61169 non-null  int64
 11  inq_last_6mths              61169 non-null  int64
 12  mths_since_recent_inq       47640 non-null  float64
 13  delinq_2yrs                 61169 non-null  int64
 14  chargeoff_within_12_mths    61169 non-null  int64
 15  num_accts_ever_120_pd       49228 non-null  float64
 16  num_tl_90g_dpd_24m          49228 non-null  float64
 17  acc_open_past_24mths        53283 non-null  float64
 18  avg_cur_bal                 49224 non-null  float64
 19  tot_hi_cred_lim             49228 non-null  float64
 20  delinq_amnt                 61169 non-null  int64
 21  def                         61169 non-null  int64
 22  income_category             61169 non-null  category
 23  emp_length_category         58495 non-null  category
dtypes: category(2), float64(9), int64(7), object(6)
memory usage: 10.4+ MB
```

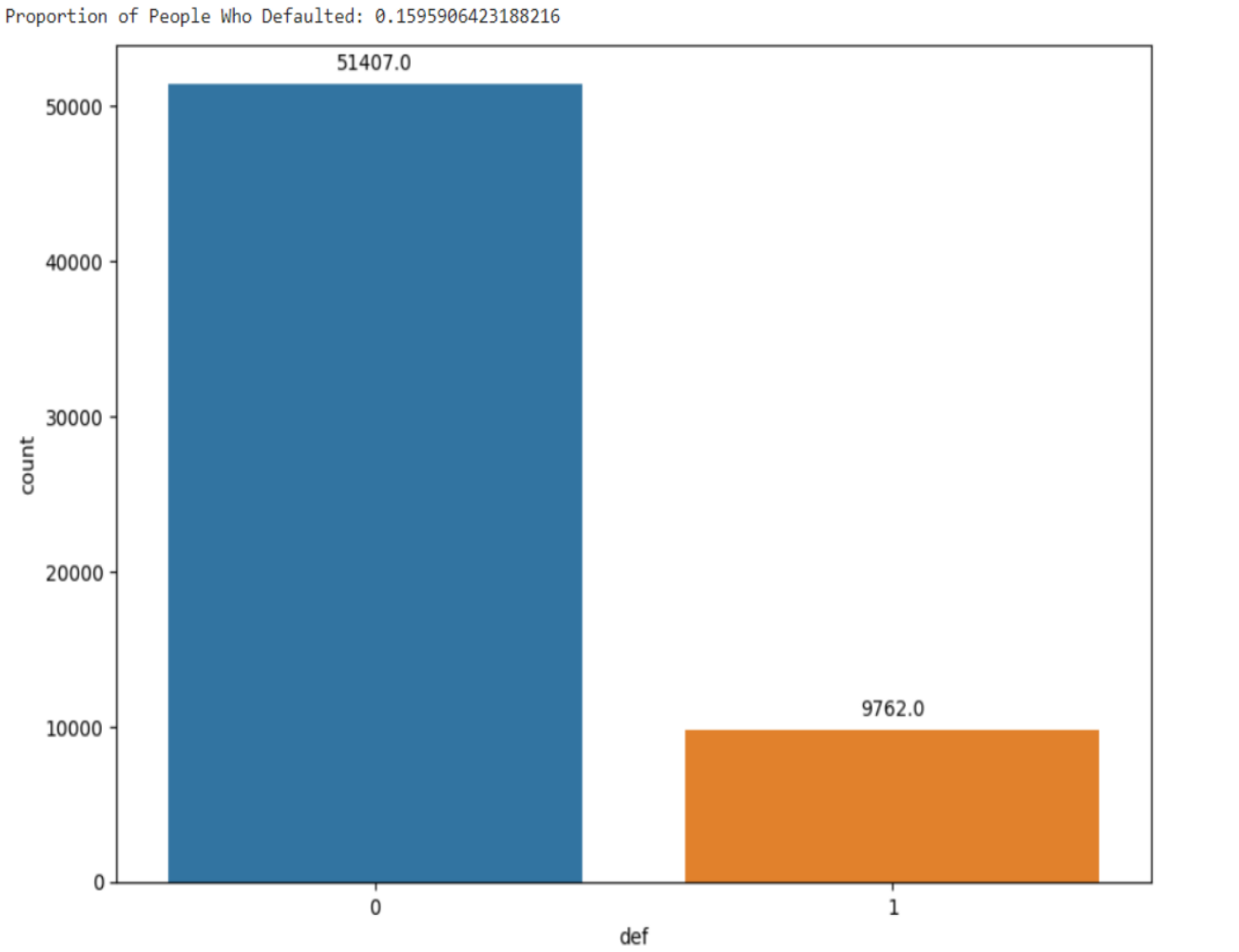
```
# Target's Mean
mean_def = df['def'].mean()
mean_def
```

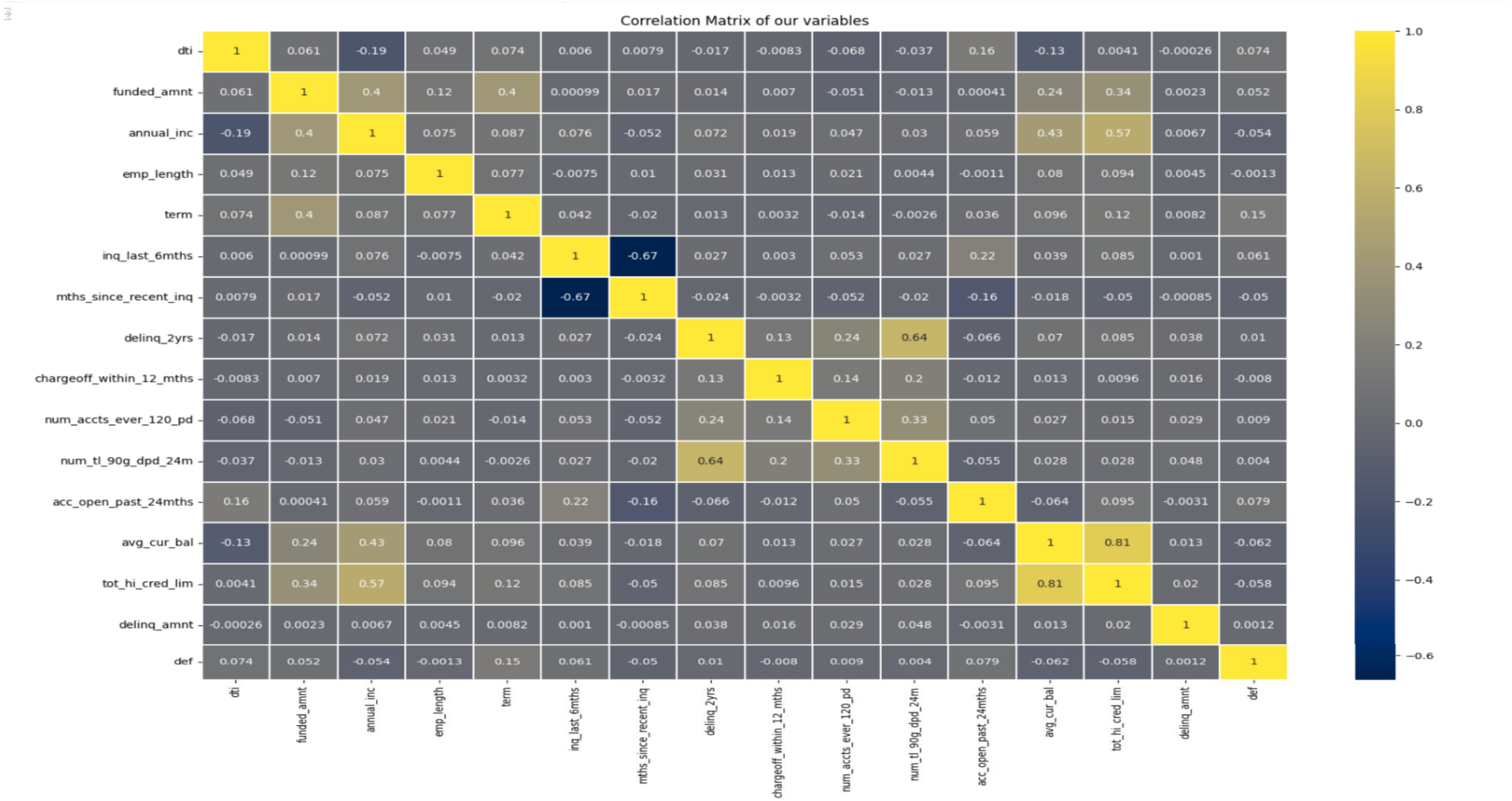
```
0.1595906423188216
```

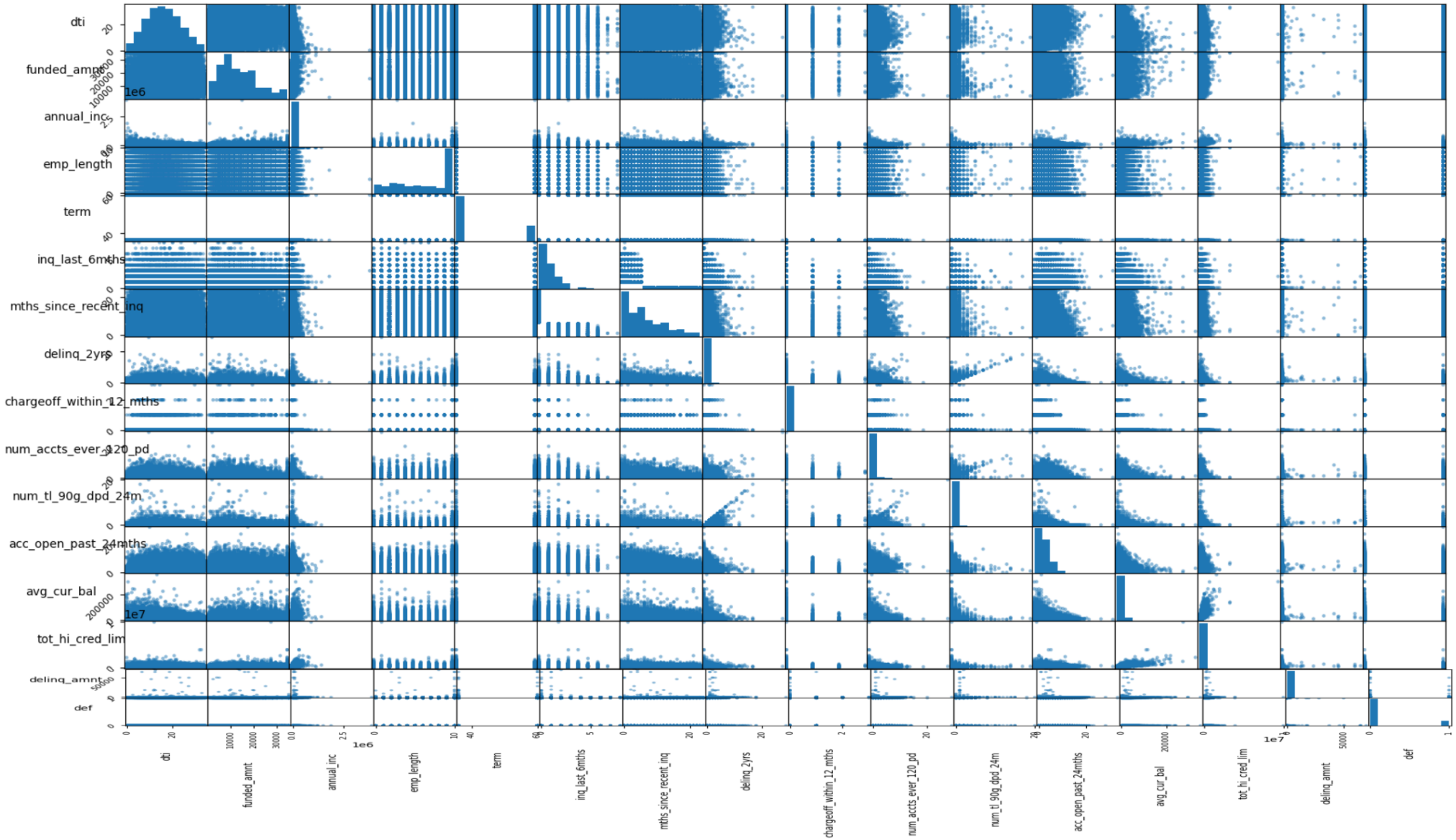
Feature Selection

- Feature selection is deemed unnecessary, primarily because the extensive number of observations (61169) significantly surpasses the relatively modest count of features (21). This substantial difference effectively diminishes the risk of overfitting.
- Furthermore, mostly, the correlation between the features is minimal, indicating an absence of redundant features. In the hypothetical scenario where redundancy exists, we could address it by either removing or reducing it through dimensionality reduction techniques.

Rule of Thumb for Feature Count:
A common rule of thumb in machine learning is the "N:10" or "N:5" rule, where "N" is the number of samples. This rule suggests having at least 10 (or 5 in some versions) times as many observations as features to reduce the risk of overfitting.




Correlation Matrix of our variables
dti 1 0.061 -0.19 0.049 0.074 0.006 0.0079 -0.017 -0.0083 -0.068 -0.037 0.16 -0.13 0.0041 -0.00026 0.074
funded_amnt 0.061 1 0.4 0.12 0.4 0.00099 0.017 0.014 0.007 -0.051 -0.013 0.00041 0.24 0.34 0.0023 0.052
annual_inc -0.19 0.4 1 0.075 0.087 0.076 -0.052 0.072 0.019 0.047 0.03 0.059 0.43 0.57 0.0067 -0.054
emp_length 0.049 0.12 0.075 1 0.077 -0.0075 0.01 0.031 0.013 0.021 0.0044 -0.0011 0.08 0.094 0.0045 -0.0013
term 0.074 0.4 0.087 0.077 1 0.042 -0.02 0.013 0.0032 -0.014 -0.0026 0.036 0.096 0.12 0.0082 0.15
inq_last_6mths 0.006 0.00099 0.076 -0.0075 0.042 1 -0.67 0.027 0.003 0.053 0.027 0.22 0.039 0.085 0.001 0.061
mths_since_recent_inq 0.0079 0.017 -0.052 0.01 -0.02 -0.67 1 -0.024 -0.0032 -0.052 -0.02 -0.16 -0.018 -0.05 -0.00085 -0.05
delinq_2yrs -0.017 0.014 0.072 0.031 0.013 0.027 -0.024 1 0.13 0.24 0.64 -0.066 0.07 0.085 0.038 0.01
chargeoff_within_12_mths -0.0083 0.007 0.019 0.013 0.0032 0.003 -0.0032 0.13 1 0.14 0.2 -0.012 0.013 0.0096 0.016 -0.008
num_accts_ever_120_pd -0.068 -0.051 0.047 0.021 -0.014 0.053 -0.052 0.24 0.14 1 0.33 0.05 0.027 0.015 0.029 0.009
num_tl_90g_dpd_24m -0.037 -0.013 0.03 0.0044 -0.0026 0.027 -0.02 0.64 0.2 0.33 1 -0.055 0.028 0.028 0.048 0.004
acc_open_past_24mths 0.16 0.00041 0.059 -0.0011 0.036 0.22 -0.16 -0.066 -0.012 0.05 -0.055 1 -0.064 0.095 -0.0031 0.079
avg_cur_bal -0.13 0.24 0.43 0.08 0.096 0.039 -0.018 0.07 0.013 0.027 0.028 -0.064 1 0.81 0.013 -0.062
tot_hi_cred_lim 0.0041 0.34 0.57 0.094 0.12 0.085 -0.05 0.085 0.0096 0.015 0.028 0.095 0.81 1 0.02 -0.058
delinq_amnt -0.00026 0.0023 0.0067 0.0045 0.0082 0.001 -0.00085 0.038 0.016 0.029 0.048 -0.0031 0.013 0.02 1 0.0012
def 0.074 0.052 -0.054 -0.0013 0.15 0.061 -0.05 0.01 -0.008 0.009 0.004 0.079 -0.062 -0.058 0.0012 1
dti
funded_amnt
annual_inc
emp_length
term
inq_last_6mths
mths_since_recent_inq
delinq_2yrs
chargeoff_within_12_mths
num_accts_ever_120_pd
num_tl_90g_dpd_24m
acc_open_past_24mths
avg_cur_bal
tot_hi_cred_lim
delinq_amnt
def
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6


dti
funded_amnt
annual_inc
emp_length
term
inq_last_6mths
mths_since_recent_inq
delinq_2yrs
chargeoff_within_12_mths
num_accts_ever_120_pd
num_tl_90g_dpd_24m
acc_open_past_24mths
avg_cur_bal
tot_hi_cred_lim
delinq_amnt
def
1e6
1e7

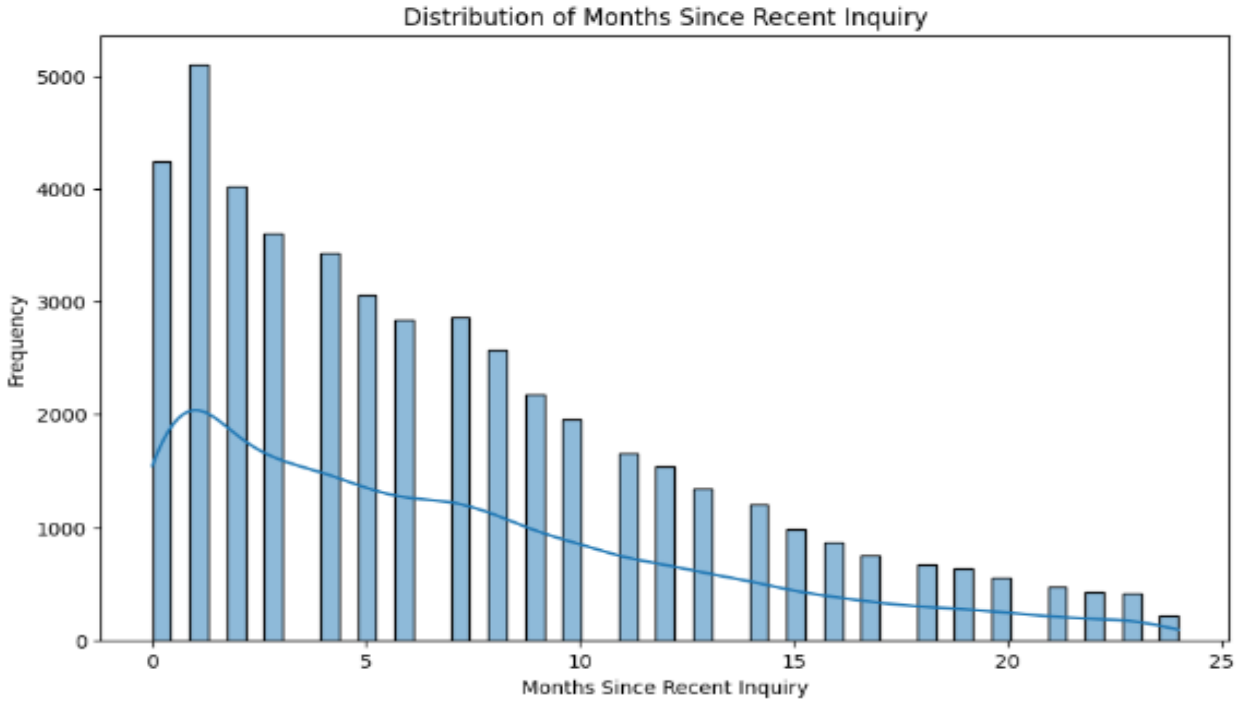
Looking at the null values

```
null_val_sums = df.isnull().sum()
pd.DataFrame({"Column": null_val_sums.index, "Number of Null Values": null_val_sums.values,
              "Proportion": null_val_sums.values / len(df) })
```

	Column	Number of Null Values	Proportion
0	issue_d	0	0.000000
1	purpose	0	0.000000
2	addr_state	0	0.000000
3	sub_grade	0	0.000000
4	home_ownership	0	0.000000
5	emp_title	3865	0.063186
6	dti	0	0.000000
7	funded_amnt	0	0.000000
8	annual_inc	0	0.000000
9	emp_length	2674	0.043715
10	term	0	0.000000
11	inq_last_6mths	0	0.000000
12	mths_since_recent_inq	13529	0.221174
13	delinq_2yrs	0	0.000000
14	chargeoff_within_12_mths	0	0.000000
15	num_accts_ever_120_pd	11941	0.195213
16	num_tl_90g_dpd_24m	11941	0.195213
17	acc_open_past_24mths	7886	0.128922
18	avg_cur_bal	11945	0.195279
19	tot_hi_cred_lim	11941	0.195213
20	delinq_amnt	0	0.000000
21	def	0	0.000000
22	income_category	0	0.000000
23	emp_length_category	2674	0.043715

```
# Calculating the percentage of missing values for each column
missing_percentage = (df.isnull().sum() / len(df)) * 100

# Filtering out columns with a high percentage of missing values (threshold: 20%)
high_missing_columns = missing_percentage[missing_percentage > 20].index.tolist()

# Displaying columns with a high percentage of missing values
print(high_missing_columns, missing_percentage[high_missing_columns])
```

```
['mths_since_recent_inq'] mths_since_recent_inq    22.117412
dtype: float64
```



The distribution of mths_since_recent_inq (Months Since Recent Inquiry) is somewhat skewed, with a concentration of values at lower months. Based on this distribution, imputation using the median might be more appropriate than the mean, as the median is less affected by skewness. For the other columns with missing values, a similar approach can be applied: evaluate the distribution and decide whether to use mean, median, or mode for imputation. But to save our time we will use Median value for NA imputation.

Weight of Evidence (WoE) Encoding:

1. **Purpose and Use Case**:
   - Primarily used in credit scoring and risk modeling.
   - Effective for binary classification problems, especially with a binary target variable.
2. **How It Works**:
   - Transforms a categorical variable into a continuous one by calculating the log of the ratio of the distribution of the target variable.
   - Provides a measure of the "strength" of a grouping in separating good cases from bad cases.
3. **Output**:
   - A single continuous variable for each categorical variable.
4. **Advantages**:
   - Handles categories with different levels of risk naturally.
   - Does not expand the feature space significantly.
   - Captures the relationship between the categorical variable and the binary target.
5. **Disadvantages**:
   - Not suitable for non-binary classification or regression tasks.
   - Can be sensitive to rare events.
   - Requires careful handling of zero-frequency cases.

```
print(f"1. Number of unique values in 'issue_d': {df['issue_d'].nunique()}")
print(f"2. Number of unique values in 'purpose': {df['purpose'].nunique()}")
print(f"3. Number of unique values in 'addr_state': {df['addr_state'].nunique()}")
print(f"4. Number of unique values in 'sub_grade': {df['sub_grade'].nunique()}")
print(f"5. Number of unique values in 'home_ownership': {df['home_ownership'].nunique()}")
print(f"6. Number of unique values in 'emp_title': {df['emp_title'].nunique()}")
print(f"7. Number of unique values in 'income_category': {df['income_category'].nunique()}")
print(f"8. Number of unique values in 'emp_length_category': {df['emp_length_category'].nunique()}")
```

```
1. Number of unique values in 'issue_d': 50
2. Number of unique values in 'purpose': 14
3. Number of unique values in 'addr_state': 47
4. Number of unique values in 'sub_grade': 35
5. Number of unique values in 'home_ownership': 5
6. Number of unique values in 'emp_title': 38576
7. Number of unique values in 'income_category': 4
8. Number of unique values in 'emp_length_category': 4
```

One-Hot Encoding:

1. **Purpose and Use Case**:
   - Used in various types of machine learning problems, including regression, classification, etc.
   - Suitable for nominal (non-ordinal) categorical variables.
2. **How It Works**:
   - Creates a new binary (0 or 1) column for each level of the categorical variable.
   - Each observation gets a 1 in the column of its category and 0 in all others.
3. **Output**:
   - Expands the feature space, with one binary feature for each category level.
4. **Advantages**:
   - Simple and effective, widely used and understood.
   - Preserves information about the category.
   - Suitable for non-linear relationships.
5. **Disadvantages**:
   - Can result in a high-dimensional feature space.
   - Not suitable for categorical variables with many levels.

Choosing Between WoE and One-Hot Encoding:

- **Use WoE** for binary classification, particularly in risk modeling or credit scoring.
- **Use One-Hot Encoding** for general purposes, non-binary classification problems, or regression.

Thus, we will chose WOE over OneHotEncoding in this project

```
# Split the data into training and testing sets
X = df.drop(['def'], axis=1)
y = df['def'].copy()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Automatically get the list of categorical and numerical columns
categorical_cols = X.select_dtypes(include=['object', 'category']).columns.tolist()
numerical_cols = X.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Create transformers for numerical and categorical columns
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),  # Impute with median for numerical columns
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('woe', WOEEncoder(cols=categorical_cols))  # WOE encoding for categorical columns
])

# Combine transformers using ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# Create a final pipeline that includes the preprocessing steps
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor)
])

# Fit and transform the training data
X_train_transformed_numpy = pipeline.fit_transform(X_train, y_train)

# Transform the testing data (do not fit again)
X_test_transformed_numpy = pipeline.transform(X_test)


# Create DataFrames from the transformed NumPy arrays
X_train_scaled = pd.DataFrame(X_train_transformed_numpy, columns=X_train.columns)
X_test_scaled = pd.DataFrame(X_test_transformed_numpy, columns=X_test.columns)
```

```
X_train_scaled.head()
```

	issue_d	purpose	addr_state	sub_grade	home_ownership	emp_title	dti	funded_amnt	annual_inc	emp_length	...	delinq_2yrs	chargeoff_within_12_mths	num_accts_ever_120_pd
0	-1.329164	-0.587671	-0.348496	1.130326	-0.595359	1.088608	-0.917011	-0.339011	-0.069496	-0.322044	...	-0.833353	-0.017991	-0.064617
1	-1.870414	-0.935818	-0.388202	-0.573957	-0.595359	1.088608	-0.144807	-0.339011	-0.069496	-0.322044	...	-0.290734	-0.017991	-0.104564
2	0.855732	0.795754	0.086759	1.130326	1.679659	2.951591	-1.110062	-0.339011	-0.069496	-0.322044	...	0.045936	-0.017991	-0.019748
3	0.307850	2.191394	2.364488	0.278185	-0.595359	1.088608	-1.110062	-0.339011	-0.069496	-0.322044	...	1.632035	-0.017991	0.055557
4	-1.196505	-0.996896	-0.689061	0.846279	-0.595359	-0.774374	0.820448	-0.339011	-0.069496	-0.322044	...	-0.162515	-0.017991	-0.010878

5 rows × 23 columns

```
X_test_scaled.head()
```

	issue_d	purpose	addr_state	sub_grade	home_ownership	emp_title	dti	funded_amnt	annual_inc	emp_length	...	delinq_2yrs	chargeoff_within_12_mths	num_accts_ever_120_pd
0	0.070390	0.713298	2.235721	0.278185	-0.595359	0.157117	-0.917011	-0.339011	-0.069496	-0.322044	...	-0.027970	-0.017991	-0.022635
1	-1.459170	-1.119053	-0.410561	1.130326	-0.595359	-0.774374	0.820448	-0.339011	-0.069496	-0.322044	...	-0.924905	-0.017991	-0.050660
2	-1.100991	-0.605995	-0.649275	-0.858004	-0.595359	0.157117	-0.144807	-0.339011	-0.069496	-0.322044	...	-0.290734	-0.017991	-0.107417
3	-0.135232	-0.263956	-0.151955	-0.858004	1.679659	0.157117	-0.917011	-0.339011	-0.069496	-0.322044	...	0.336320	-0.017991	0.009835
4	0.427243	0.175808	-0.589597	1.130326	-0.595359	-0.774374	-0.144807	-0.339011	-0.069496	-0.322044	...	-0.290734	-0.017991	0.031238

5 rows × 23 columns

```
LR_model = LogisticRegression(C=0.01)
LR_model.fit(X_train_scaled, y_train)
```

```
LogisticRegression
LogisticRegression(C=0.01)
```

```
predicted_probabilities = LR_model.predict_proba(X_test_scaled)[:, 1]
predicted_probabilities
```

```
array([0.05261935, 0.13230018, 0.07751043, ..., 0.16179638, 0.06731672,
       0.21826046])
```

```
y_pred = LR_model.predict(X_test_scaled)
y_pred
```

```
array([0, 0, 0, ..., 0, 0, 0])
```

```
cv = StratifiedKFold(n_splits=5)

LR_model = LogisticRegression(C=0.01)

scores = cross_val_score(LR_model, X_train_scaled, y_train, scoring='roc_auc')

print('Scores on CV folds:', scores, '\n')
print('Average score on CV folds: {:.4f}'.format(np.mean(scores)))
print('Std of score on CV folds: {:.4f}'.format(np.std(scores)))
```

```
Scores on CV folds: [0.7061003  0.71953635 0.71535419 0.71454105 0.71130754]

Average score on CV folds: 0.7134
Std of score on CV folds: 0.0045
```

```
params = {
    'C': np.logspace(-3, 0, 20),
}

LR_model = LogisticRegression()

GS_LR = GridSearchCV(cv=cv, estimator=LR_model, param_grid=params, scoring='roc_auc')

GS_LR.fit(X_train_scaled, y_train)
```

```
GridSearchCV
GridSearchCV(cv=StratifiedKFold(n_splits=5, random_state=None, shuffle=False),
             estimator=LogisticRegression(),
             param_grid={'C': array([0.001     , 0.00143845, 0.00206914, 0.00297635, 0.00428133,
       0.00615848, 0.00885867, 0.01274275, 0.01832981, 0.02636651,
       0.0379269 , 0.05455595, 0.078476  , 0.11288379, 0.16237767,
       0.23357215, 0.33598183, 0.48329302, 0.6951928 , 1.        ])},
             scoring='roc_auc')
estimator: LogisticRegression
LogisticRegression()
LogisticRegression
```

```
print('Best C {:.3f}'.format(GS_LR.best_params_['C']))
print('Best ROC AUC {:.4f}'.format(GS_LR.best_score_))
```

```
Best C 1.000
Best ROC AUC 0.7158
```

Pros:

1. **Interpretability**: Decision trees are straightforward and easy to understand, making them highly interpretable.
2. **No Need for Feature Scaling**: They don't require feature scaling like normalization or standardization. (you can pass scaled data to a Decision Tree Classifier, but in most cases, it's not necessary, and it won't impact the performance of the model)
3. **Flexibility**: Capable of handling both linear and non-linear problems effectively.

Cons:

1. **Performance with Small Datasets**: Tend to underperform on very small datasets.
2. **Overfitting Risks**: Prone to overfitting, especially with complex trees.

Key Hyperparameters

1. **Criterion**: Choice between 'gini' or 'entropy' for measuring the quality of a split.
2. **Max Depth** (`max_depth`): The maximum depth of the tree.
3. **Min Samples Split** (`min_samples_split`): The minimum number of samples required to split an internal node.
4. **Min Samples Leaf** (`min_samples_leaf`): The minimum number of samples required to be at a leaf node.

```
dtc = DecisionTreeClassifier(random_state=123)

dtc.fit(X=X_train_scaled, y=y_train)
```

```
DecisionTreeClassifier(random_state=123)
```

```
score = dtc.score(X_test_scaled, y_test)
print('ROC AUC {:.3f}'.format(score))
```

```
ROC AUC 0.738
```

```
# Defining the hyperparameter grid
param_grid = {
    'max_depth': [3, 5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5, 10],
    'max_features': [None, 'auto', 'sqrt', 'log2', 0.5, 0.7]
}

# Setting up the Grid Search
grid_search = GridSearchCV(dtc, param_grid, cv=5, scoring='roc_auc', n_jobs=1)

# Fitting the Grid Search model
grid_search.fit(X_train_scaled, y_train)

# Printing the best parameters and the best score
print("Best parameters:", grid_search.best_params_)
print("Best ROC AUC score:", grid_search.best_score_)
```

```
Best parameters: {'max_depth': 5, 'max_features': 0.7, 'min_samples_leaf': 10, 'min_samples_split': 2}
Best ROC AUC score: 0.6967521178743834
```

Cross Validation

```
cv_scores = cross_val_score(dtc, X_train_scaled, y_train, cv=5, scoring='roc_auc')
# Print the ROC AUC scores for each fold and their mean
print("ROC AUC scores for each fold:", cv_scores)
print("Average ROC AUC score:", cv_scores.mean())
```

```
ROC AUC scores for each fold: [0.55693453 0.54945811 0.56237294 0.56617328 0.55604487]
Average ROC AUC score: 0.5581967473256837
```

Model 3: Random Forset

Random Forest Classifier

Pros:

- **Better Performance**: Generally provides more accurate predictions than a single decision tree.
- **Overfitting Reduction**: Less prone to overfitting compared to a single decision tree due to averaging of multiple trees.
- **Handles Unbalanced Data**: Works well with unbalanced and missing data.
- **Feature Importance**: Provides insights into feature importance, helpful for feature selection.

Cons:

- **Complexity and Size**: More complex and requires more computational resources and memory.
- **Model Interpretability**: Less interpretable than a single decision tree due to the complexity of multiple trees.
- **Training Time**: Longer training time because of multiple trees.
- **Performance with Noisy Data**: Can overfit on certain noisy classification/regression tasks.

Random Forest classifiers generally outperform single Decision Tree classifiers in terms of accuracy and are less prone to overfitting, but they lose out in terms of simplicity and interpretability. The choice between the two depends on the specific requirements of the problem, such as the need for accuracy versus the need for a clear, interpretable model.

```
# Creating the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=123)

# Training the model
rf_model.fit(X_train_scaled, y_train)
```

```
score = rf_model.score(X_test_scaled, y_test)
print('ROC AUC {:.3f}'.format(score))
```

```
ROC AUC 0.835
```

```
param_dist = {
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': sp_randint(2, 11),
    'min_samples_leaf': sp_randint(1, 5),
    'max_features': [None, 'auto', 'sqrt', 'log2', 0.5, 0.7]
}

# Set up the Randomized Search
n_iter_search = 20  # Number of parameter settings that are sampled
random_search = RandomizedSearchCV(rf_model, param_distributions=param_dist,
                                   n_iter=n_iter_search, cv=5, scoring='roc_auc', n_jobs=1)

# Fit the Randomized Search model
random_search.fit(X_train_scaled, y_train)

# Print the best parameters and the best ROC AUC score
print("Best parameters:", random_search.best_params_)
print("Best ROC AUC score:", random_search.best_score_)
```

```
Best parameters: {'max_depth': 10, 'max_features': None, 'min_samples_leaf': 3, 'min_samples_split': 7}
Best ROC AUC score: 0.7298004017926777
```

+ Код + Текст

Cross Validation

```
cv_scores = cross_val_score(rf_model, X_train_scaled, y_train, cv=5, scoring='roc_auc')
# Print the ROC AUC scores for each fold and their mean
print("ROC AUC scores for each fold:", cv_scores)
print("Average ROC AUC score:", cv_scores.mean())
```

```
ROC AUC scores for each fold: [0.69910889 0.70021251 0.71231707 0.71235515 0.70462175]
Average ROC AUC score: 0.7057230726243157
```

Model 4: k Nearest Neighbours (kNN)

k-Nearest Neighbors (kNN) Summary

k-Nearest Neighbors (kNN) is a fundamental algorithm in machine learning, used for both classification and regression tasks. Below is an overview of its key aspects:

Concept

- **Proximity-Based**: Assumes similar data points are close to each other.
- **Instance-Based Learning**: Retains the training dataset for making predictions.

Working

- **Classification**: Assigns a class based on the majority class of the 'k' nearest neighbors.
- **Regression**: Predicts a value based on the average of the values of the 'k' nearest neighbors.
- **Distance Metric**: Utilizes metrics like Euclidean or Manhattan distance to find closest neighbors.

Hyperparameters

- **Number of Neighbors (k)**: Crucial parameter influencing the model's sensitivity to noise and computation cost.
- **Distance Metric**: Choice of metric (Euclidean, Manhattan, Minkowski, etc.) can impact performance.

Features

- **No Training Phase**: Does not learn a function but stores the dataset.
- **Lazy Learning**: Approximates the function locally, with computation deferred until evaluation.

Advantages

- **Simple and Versatile**: Easy to implement, suitable for classification and regression.
- **Effective with Sufficient Data**: Performs well with large, representative datasets.

Disadvantages

- **Scalability and Efficiency**: Inefficient with large datasets due to storage and computation needs.
- **Curse of Dimensionality**: Performance degrades with increasing number of features.
- **Sensitivity to Features**: Requires careful feature selection or dimensionality reduction.

```
kNN_model = KNeighborsClassifier()
cv_scores = cross_val_score(kNN_model, X_train_scaled, y_train, cv=5, scoring='roc_auc')
# Print the ROC AUC scores for each fold and their mean
print("ROC AUC scores for each fold:", cv_scores)
print("Average ROC AUC score:", cv_scores.mean())
```

```
ROC AUC scores for each fold: [0.595236   0.60846864 0.59902685 0.61113466 0.59656528]
Average ROC AUC score: 0.6020862870911932
```

```
cv = StratifiedKFold(n_splits=5)

kNN_model = KNeighborsClassifier()

# Define the hyperparameter distribution
params = {
    'n_neighbors': sp_randint(1, 100)
}

# Set up Randomized Search
n_iter_search = 20
RS_kNN = RandomizedSearchCV(
    kNN_model,
    param_distributions=params,
    n_iter=n_iter_search,
    cv=cv,
    scoring='roc_auc',
    random_state=42,
    n_jobs=1
)

# Fit the Randomized Search model
RS_kNN.fit(X_train_scaled, y_train)

# Print the best parameters
print("Best Parameters:", RS_kNN.best_params_)
print("Best ROC AUC Score:", RS_kNN.best_score_)
```

```
Best Parameters: {'n_neighbors': 93}
Best ROC AUC Score: 0.6855055533988963
```

Model 5: Blending. Voting Classifier

A Voting Classifier is an ensemble machine learning model that combines the predictions from multiple other models. It is typically used to improve the overall performance by leveraging the strengths of various base models.

Concept

- **Ensemble Approach**: Combines the decisions from multiple models to make a final prediction.
- **Types of Voting**:
  - **Hard Voting**: Uses the mode of the predictions (majority vote) from the base models.
  - **Soft Voting**: Predicts the class label based on the average of probabilities given by the base models.

Working

- **Combining Models**: Incorporates diverse models, often with different algorithms, to make a collective decision.
- **Final Decision Rule**: Depends on the voting strategy (hard or soft).

Advantages

- **Improved Performance**: Often achieves higher performance than individual models alone.
- **Reduced Overfitting**: By averaging out biases, it reduces the likelihood of overfitting.

Disadvantages

- **Complexity**: More complex to implement and understand compared to a single model.
- **Computationally Intensive**: Requires training multiple models, which can be resource-intensive.

Applications

- Commonly used in competitions and practical applications where performance improvement is critical.
- Effective in scenarios where individual models have different strengths and weaknesses.

Considerations

- **Model Diversity**: It is crucial to combine models that are diverse and have different error patterns.
- **Tuning**: Requires careful tuning of the base models and the selection of an appropriate voting strategy.

```
[ ]  # pass our models to VotingClassifier

     kNN_model = KNeighborsClassifier(n_neighbors=RS_kNN.best_params_['n_neighbors'])

     LR_model = LogisticRegression(C=GS_LR.best_params_['C'])

     VC = VotingClassifier([('kNN', kNN_model), ('LR', LR_model)], voting='soft')
```

CV and HP tuning

```
[ ]  cv_scores = cross_val_score(VC, X_train_scaled, y_train, cv=5, scoring='roc_auc')
     # Print the ROC AUC scores for each fold and their mean
     print("ROC AUC scores for each fold:", cv_scores)
     print("Average ROC AUC score:", cv_scores.mean())
```

```
ROC AUC scores for each fold: [0.70803471 0.71803085 0.71498194 0.71401298 0.7105917 ]
Average ROC AUC score: 0.7131304346631137
```

```
# Define the distribution for weights
def generate_random_weights():
    i = np.random.uniform(0, 1)
    return [i, 1 - i]

# Set up Randomized Search
n_iter_search = 20
RS_VC = RandomizedSearchCV(
    estimator=VC,
    param_distributions={
        'weights': [generate_random_weights() for _ in range(200)]
    },
    n_iter=n_iter_search,
    cv=CV,
    scoring='roc_auc',
    random_state=123,
    n_jobs=1
)

# Fit the Randomized Search model
RS_VC.fit(X_train_scaled, y_train)

# Print the best parameters
print("Best Parameters:", RS_VC.best_params_)
print("Best ROC AUC Score:", RS_VC.best_score_)
```

```
Best Parameters: {'weights': [0.1747179771935461, 0.8252820228064539]}
Best ROC AUC Score: 0.7166709524501609
```

Model 6: LightGBM (Light Gradient Boosting Machine)

LightGBM and XGBoost: Overview of Gradient Boosting Frameworks

LightGBM (Light Gradient Boosting Machine)

- **Developed By**: Microsoft.

Key Features:

- **Efficient Handling of Large Data**: Utilizes a histogram-based algorithm, enhancing speed and efficiency, especially with large datasets.
- **Lower Memory Usage**: More memory-efficient compared to traditional gradient boosting frameworks.
- **Gradient-based One-Side Sampling (GOSS)**: A feature sampling method focusing on the most informative instances and minimizing the influence of less significant ones.
- **Exclusive Feature Bundling (EFB)**: Reduces the number of features by bundling mutually exclusive features, with minimal information loss.
- **Support for Categorical Features**: Natively handles categorical features, eliminating the need for one-hot encoding.

Usage:

- Ideal for handling large datasets efficiently and speedily, often chosen in scenarios with limited computational resources.

```
lgbm_model = lgb.LGBMClassifier(random_state=123)
lgbm_model.fit(X_train_scaled, y_train)
```

```
[LightGBM] [Info] Number of positive: 7823, number of negative: 41112
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.010368 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1741
[LightGBM] [Info] Number of data points in the train set: 48935, number of used features: 23
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.159865 -> initscore=-1.659232
[LightGBM] [Info] Start training from score -1.659232
LGBMClassifier
LGBMClassifier(random_state=123)
```

Differences from Random Forest

- **Model Building**:
  - Random Forest builds trees independently.
  - LightGBM and XGBoost build trees sequentially.
- **Bias-Variance Trade-off**:
  - Random Forest reduces variance without increasing bias.
  - Gradient boosting methods (LightGBM, XGBoost) can reduce bias but may increase variance.
- **Performance**:
  - LightGBM and XGBoost often outperform Random Forest in terms of prediction accuracy, particularly in structured data scenarios and Kaggle competitions.
- **Speed and Scalability**:
  - LightGBM is designed for speed and efficiently handles larger datasets compared to Random Forest and XGBoost.
- **Interpretability**:
  - Random Forest is generally more interpretable than boosting methods, owing to the simplicity of averaging multiple decision trees.

CV

```
cv_scores = cross_val_score(lgbm_model, X_train_scaled, y_train, cv=5, scoring='roc_auc')
# Print the ROC AUC scores for each fold and their mean
print("ROC AUC scores for each fold:", cv_scores)
print("Average ROC AUC score:", cv_scores.mean())
```

```
[LightGBM] [Info] Total Bins 1735
[LightGBM] [Info] Number of data points in the train set: 39148, number of used features: 23
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.159855 -> initscore=-1.659308
[LightGBM] [Info] Start training from score -1.659308
ROC AUC scores for each fold: [0.74135923 0.73965482 0.73951527 0.7388156  0.73415041]
Average ROC AUC score: 0.7386990656491859
```

HP tunning

```
# Define the LightGBM model
lgbm = lgb.LGBMClassifier(random_state=123)

# Define the hyperparameter space to search
param_dist = {
    'num_leaves': sp_randint(20, 50),
    'min_child_samples': sp_randint(100, 500),
    'min_child_weight': sp_uniform(0.01, 0.1),
    'subsample': sp_uniform(0.6, 0.4),
    'colsample_bytree': sp_uniform(0.6, 0.4),
    'reg_alpha': [0, 1e-1, 1, 2, 5, 7, 10, 50, 100],
    'reg_lambda': [0, 1e-1, 1, 5, 10, 20, 50, 100]
}

# Set up Randomized Search
n_iter_search = 20
RS_lgbm = RandomizedSearchCV(lgbm, param_distributions=param_dist,
                             n_iter=n_iter_search, scoring='roc_auc',
                             cv=5, random_state=123, n_jobs=1)

# Fit Randomized Search
RS_lgbm.fit(X_train_scaled, y_train)

# Print the best parameters and the best score
print("Best Parameters:", RS_lgbm.best_params_)
print("Best ROC AUC Score:", RS_lgbm.best_score_)
```

```
[LightGBM] [Info] Total Bins 1707
[LightGBM] [Info] Number of data points in the train set: 48935, number of used features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.159865 -> initscore=-1.659232
[LightGBM] [Info] Start training from score -1.659232
Best Parameters: {'colsample_bytree': 0.6523579802656323, 'min_child_samples': 140, 'min_child_weight': 0.05602384244022097, 'num_leaves': 32, 'reg_alpha': 0.1,
Best ROC AUC Score: 0.7441914669926959
```

Summary of the results so far

Based on the extracted performance metrics from the notebook, here are the ROC AUC scores for the different models:

1. **Logistic Regression (Cross-Validation)**: Average ROC AUC = 0.7134 **Logistic Regression (GridSearchCV)**: Best ROC AUC = 0.7158
2. **Decision Tree Classifier (Cross-Validation)**: Average ROC AUC = 0.5582 **Decision Tree Classifier (GridSearchCV)**: Best ROC AUC = 0.69675
3. **Random Forest Classifier (Cross-Validation)**: Average ROC AUC = 0.70572 **Random Forest Classifier (RandomizedSearchCV)**: Best ROC AUC = 0.72980
4. **K-Nearest Neighbors (Cross-Validation)**: Average ROC AUC = 0.60208 **K-Nearest Neighbors (RandomizedSearchCV)**: Best ROC AUC = 0.68551
5. **Voting Classifier (Cross-Validation)**: Average ROC AUC = 0.71313

Voting Classifier (RandomizedSearchCV): Best ROC AUC = 0.716671

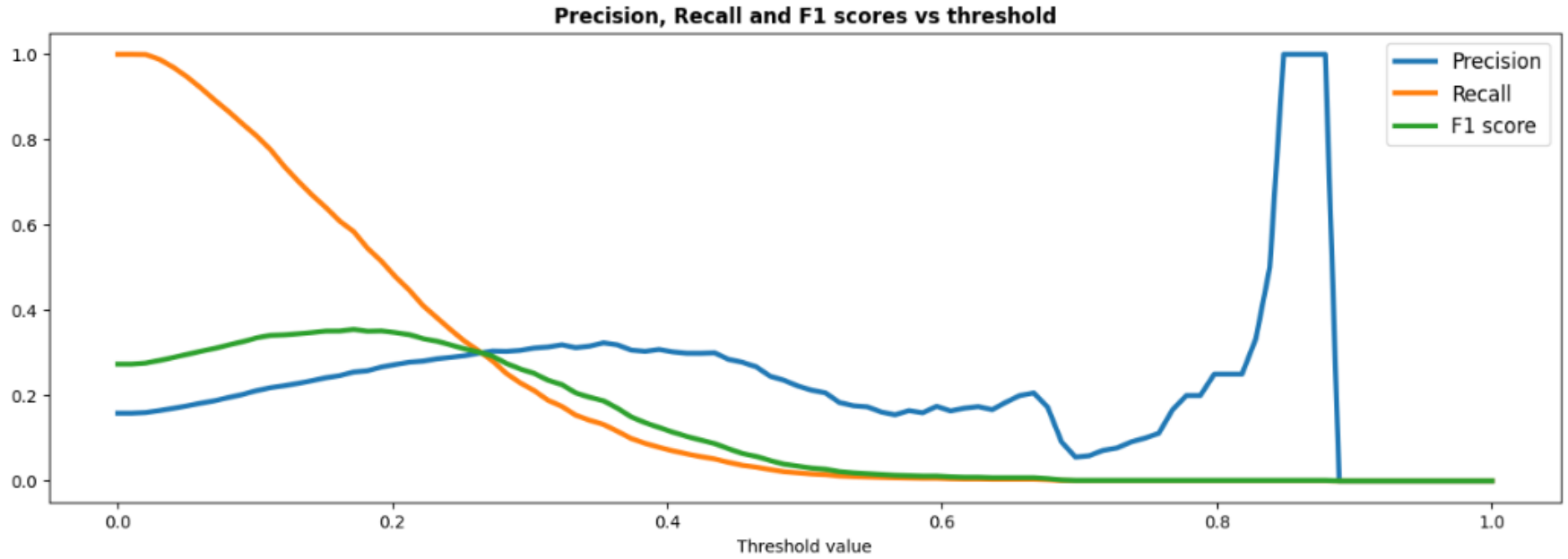
6. **LightGBM (Cross-Validation)** Average ROC AUC = 0.73869 **LightGBM (RandomizedSearchCV)** Best ROC AUC Score = 0.74419

From these results, the **Light Gradient Boosting Machine with RandomizedSearchCV used to tune hyperparameters** appears to be the best model with the highest ROC AUC score of **0.74419**. This model outperforms the others in terms of the ROC AUC metric, which is a key indicator of performance in binary classification tasks.

It's important to note that while ROC AUC is a crucial metric, other factors like model interpretability, computational efficiency, and how the model will be used in practice should also be considered when finalizing the best model for deployment.

Here are the Gini coefficients for each model, calculated from their respective ROC AUC scores:

1. **Logistic Regression (Cross-Validation)**: Gini = 0.4268
2. **Logistic Regression (GridSearchCV)**: Gini = 0.4316
3. **Decision Tree Classifier (Cross-Validation)**: Gini = 0.1164
4. **Decision Tree Classifier (GridSearchCV)**: Gini = 0.3935
5. **Random Forest Classifier (Cross-Validation)**: Gini = 0.41144
6. **Random Forest Classifier (RandomizedSearchCV)**: Gini = 0.4596
7. **K-Nearest Neighbors (Cross-Validation)**: Gini = 0.20416
8. **K-Nearest Neighbors (RandomizedSearchCV)**: Gini = 0.37102
9. **Voting Classifier (Cross-Validation)**: Gini = 0.42626
10. **Voting Classifier (RandomizedSearchCV)**: Gini = 0.43334
11. **LightGBM (Cross-Validation)**: Gini = 0.47738
12. **LightGBM (RandomizedSearchCV)**: Gini = 0.48838



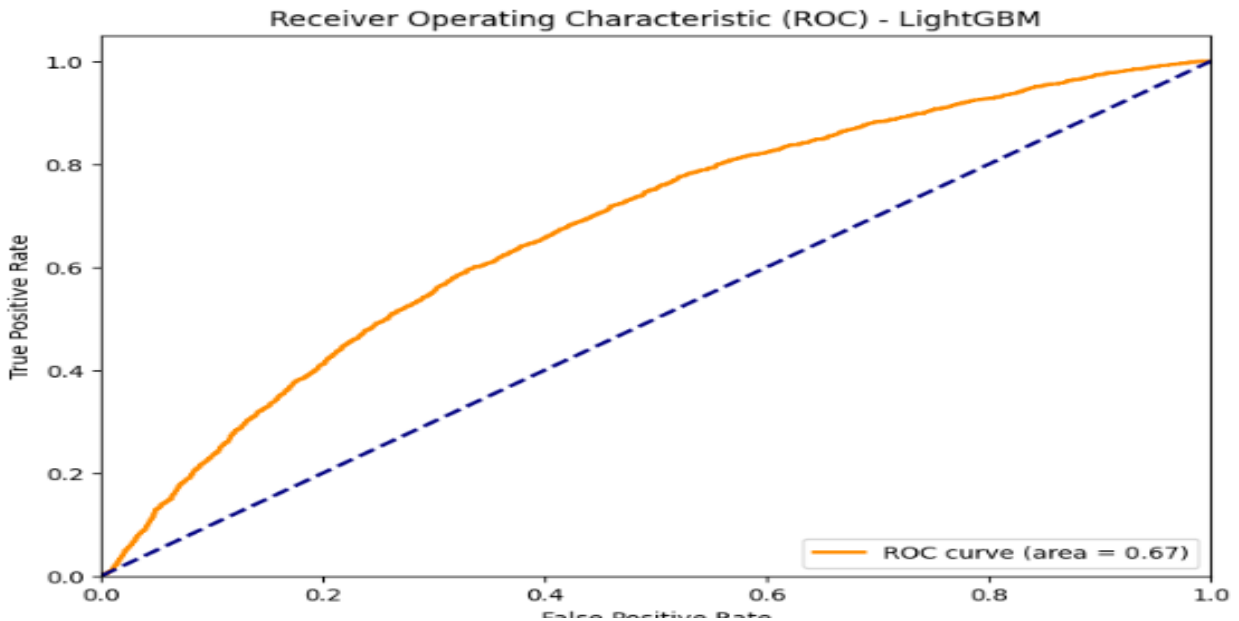
From the graph we can see that it is fairly reasinble to leave the threshhold at the level Target_mean level of 0.1595906423188216

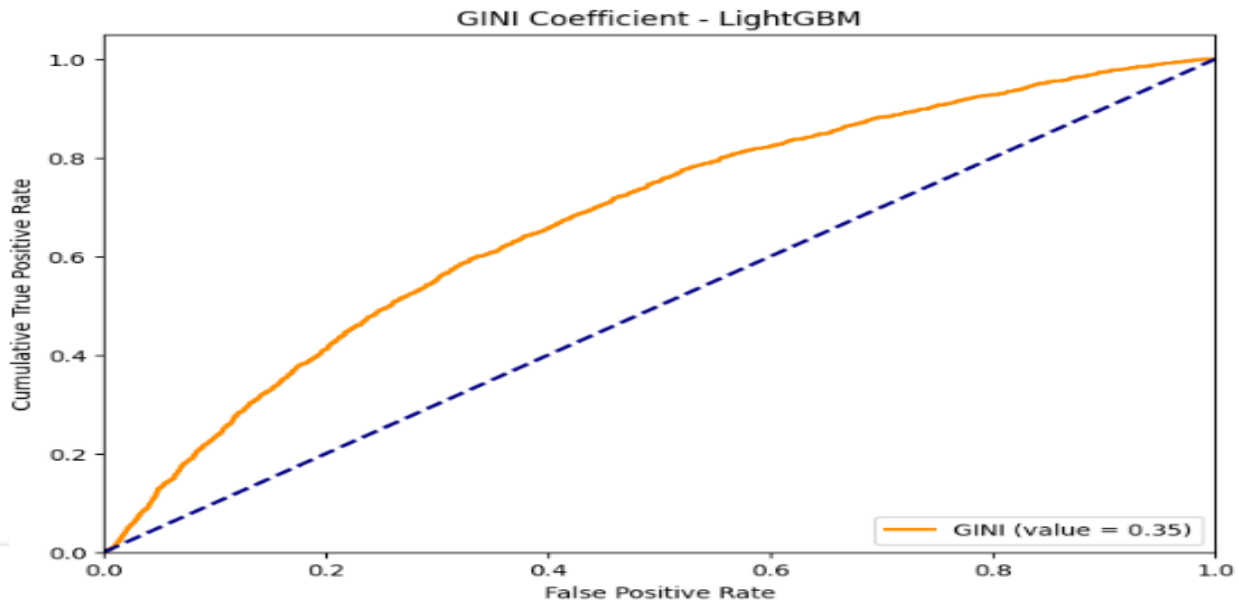
```
# By default threshhold is 0.5 BUT to increase the RECALL in OUR CREDIT SCORING we've chosen it to be rqual to Target mean level.
# Get probability predictions for the positive class
y_pred_proba = best_lgbm_model.predict_proba(X_test_scaled)[:, 1]

# Define your custom threshold
custom_threshold = 0.1595906423188216

# Apply the custom threshold to determine class labels
y_pred_custom = np.where(y_pred_proba >= custom_threshold, 1, 0)

y_pred_custom
```

```
array([0, 0, 0, ..., 0, 0, 1])
```

+ Код + Текст

```python
# Obtain probability scores using the best estimator
y_scores = RS_lgbm.best_estimator_.predict_proba(X_test_scaled)[:, 1]

# Calculating True Positive Rate and False Positive Rate for ROC
fpr, tpr, _ = roc_curve(y_test, y_scores)
roc_auc = auc(fpr, tpr)

# Plotting the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) - LightGBM')
plt.legend(loc="lower right")
plt.show()

# GINI calculation
gini = 2 * roc_auc - 1

# Plotting the GINI curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='GINI (value = %0.2f)' % gini)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('Cumulative True Positive Rate')
plt.title('GINI Coefficient - LightGBM')
plt.legend(loc="lower right")
plt.show()
```





LightGBM's results on the test set are not so great. BUt that's the best you've achieved so far given time constraints. So, let's use it as it is.

Optimal Rate Calculation

```
[ ]  nc_2['term'].value_counts()
```

```
36    40718
60    19615
Name: term, dtype: int64
```

```
from scipy.optimize import minimize_scalar

# Define constants
r0 = f = 0.08  # 8% annual risk-free and borrowing interest rate

# Function to calculate r_n
def calculate_r_n(r):
    return (1 + r) ** (1/12) - 1

def calculate_e_pnl(r, pd, s, term):
    r_n = calculate_r_n(r)
    r0_n = calculate_r_n(r0)
    f_n = calculate_r_n(f)

    term1 = (1 - 0.6 * pd) * r_n * (1 + r_n)**term / ((1 + r_n)**term - 1) * s
    term2 = ((1 + r0_n)**term - 1) / r0_n
    term3 = (1 + f_n)**term * s

    return term1 * term2 - term3

# Optimization function for a single client with term length
def optimize_rate_for_client(pd, s, term):
    obj_func = lambda r: -calculate_e_pnl(r, pd, s, term)
    bounds = (0.05, 0.30)  # Bounds for the interest rate
    result = minimize_scalar(obj_func, bounds=bounds, method='bounded')
    optimal_rate = result.x if calculate_e_pnl(result.x, pd, s, term) > 0 else 0
    return optimal_rate

# Apply the optimization for each client
nc_2['optimal_rate'] = nc_2.apply(lambda x: optimize_rate_for_client(x['PD'], x['funded_amnt'], x['term']), axis=1)
```

$$E(PNL) = FV - PD * EAD * LGD =$$
$$= (1 - 0.6 * PD) * \frac{r_n * (1 + r_n)^n}{(1 + r_n)^n - 1} * S$$
$$* \frac{(1 + r_{0_n})^n - 1}{r_{0_n}} - (1 + f_n)^n * S$$

```
[ ]  nc_2['optimal_rate'].value_counts()
```

```
0.299994    58915
0.000000     1418
Name: optimal_rate, dtype: int64
```

We see that an additional constraint should be included to account for PD.

we take minimal rate as 9% to be higher that risk free and to cover costs of the bank at least to some extent.

Optimization procedure was created with the following constraints:

- Maximum and minimum rates of 9% and 30% (the latter one was achieved during first stage of optimization on the previous slide) respectively to avoid potential losses and excessively high rates;
- Non – linear risk term structure, penalizing clients with high default rate by higher optimal rates в

To conclude, such algorithm allows to calculate optimal rate (for banks) for each client, given their characteristics

$$E(PNL) = FV - PD * EAD * LGD =$$
$$= (1 - 0.6 * PD) * \frac{r_n * (1 + r_n)^n}{(1 + r_n)^n - 1} * S$$
$$* \frac{(1 + r_{0_n})^n - 1}{r_{0_n}} - (1 + f_n)^n * S$$

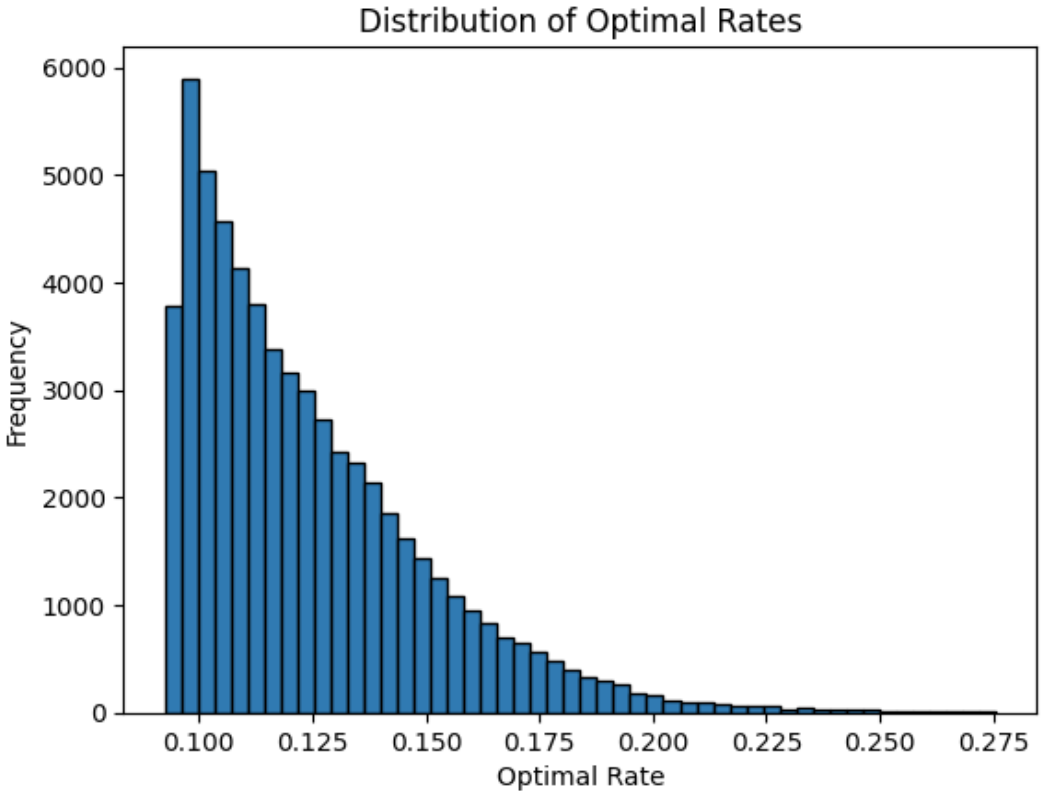
$$r_n = \sqrt[12]{1 + r} - 1$$

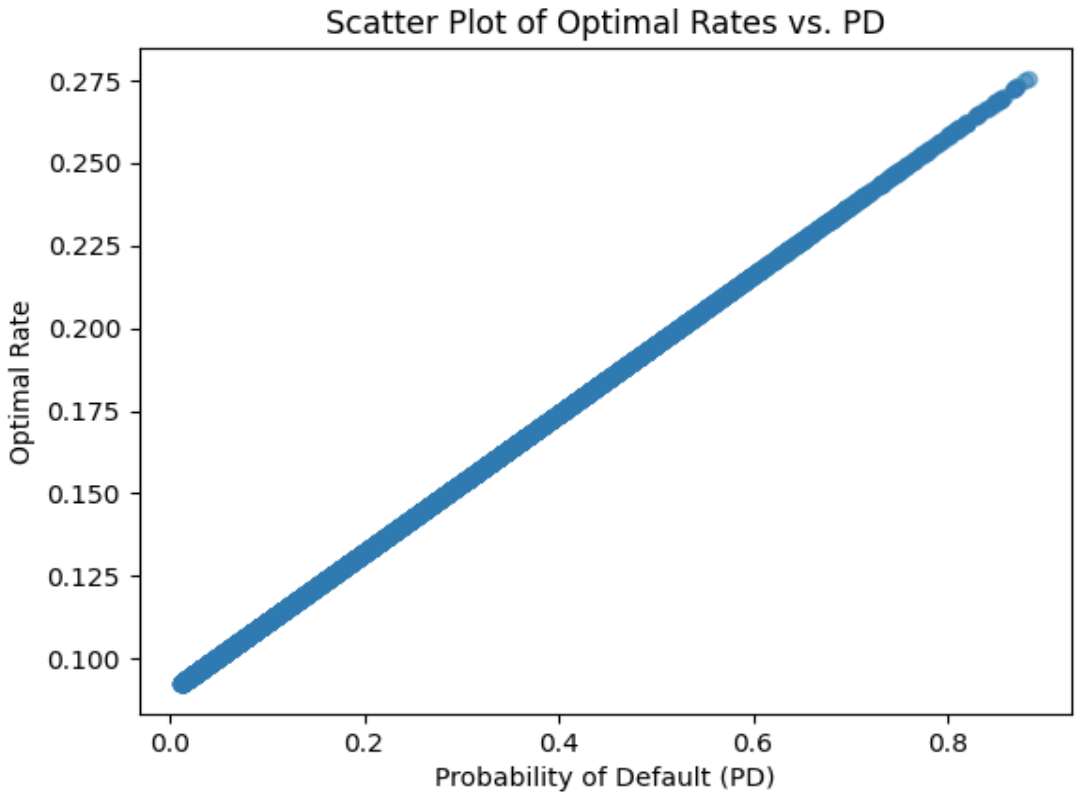
```python
# Constants
r0 = f = 0.08  # Annual risk-free and borrowing interest rate
min_rate = 0.09  # Minimum rate
max_rate = 0.30  # Maximum rate

# Monthly rate from annual rate
def calculate_r_n(r):
    return (1 + r) ** (1/12) - 1

# E(PNL) with term length as a parameter, incorporating non-linearity
def calculate_e_pnl(r, pd, s, term):
    r_n = calculate_r_n(r)
    r0_n = calculate_r_n(r0)
    f_n = calculate_r_n(f)

    if (1 + r_n)**term - 1 == 0:
        return -np.inf

    # Non-linear risk term that penalizes high PD with higher rates
    risk_term = pd * r ** 2

    term1 = (1 - risk_term) * r_n * (1 + r_n)**term / ((1 + r_n)**term - 1) * s
    term2 = ((1 + r0_n)**term - 1) / r0_n
    term3 = (1 + f_n)**term * s

    return term1 * term2 - term3

# Rate based on PD, increasing with PD
def rate_based_on_pd(pd):
    # Linear increase from min_rate at PD = 0 to max_rate at PD = 1
    rate_increase = max_rate - min_rate
    return min_rate + pd * rate_increase

# Optimization function for a single client
def optimize_rate_for_client(pd, s, term):
    obj_func = lambda r: -calculate_e_pnl(r, pd, s, term)
    bounds = (min_rate, rate_based_on_pd(pd))
    result = minimize_scalar(obj_func, bounds=bounds, method='bounded')
    return result.x if result.fun < 0 else np.nan  # fun is the minimized value of E(PNL)

# Assuming nc_2 is your DataFrame with new client data

# Apply the optimization to find the optimal rate for each client
nc_2['optimal_rate'] = nc_2.apply(
    lambda x: optimize_rate_for_client(x['PD'], x['funded_amnt'], x['term']), axis=1
)
```

```
Team4_answer
```

	client_id	PD	rate_offered
0	0	0.175818	0.126915
1	1	0.223877	0.137009
2	2	0.063100	0.103245
3	3	0.138454	0.119071
4	4	0.067899	0.104252
...	...	...	...
60328	60328	0.193275	0.130584
60329	60329	0.247910	0.142055
60330	60330	0.159613	0.123513
60331	60331	0.123956	0.116027
60332	60332	0.048947	0.100275

60333 rows × 3 columns





The proposed algorithm does not reject anyone, but assigns greater optimal rates for more risky clients


HSE UNIVERSITY
EST 1992