How to Develop a Credit Risk Model and Scorecard

A walkthrough of statistical credit risk modeling, probability of default prediction, and credit scorecard development with Python

Asad Mumtaz Aug 13, 2020 · 17 min read ★

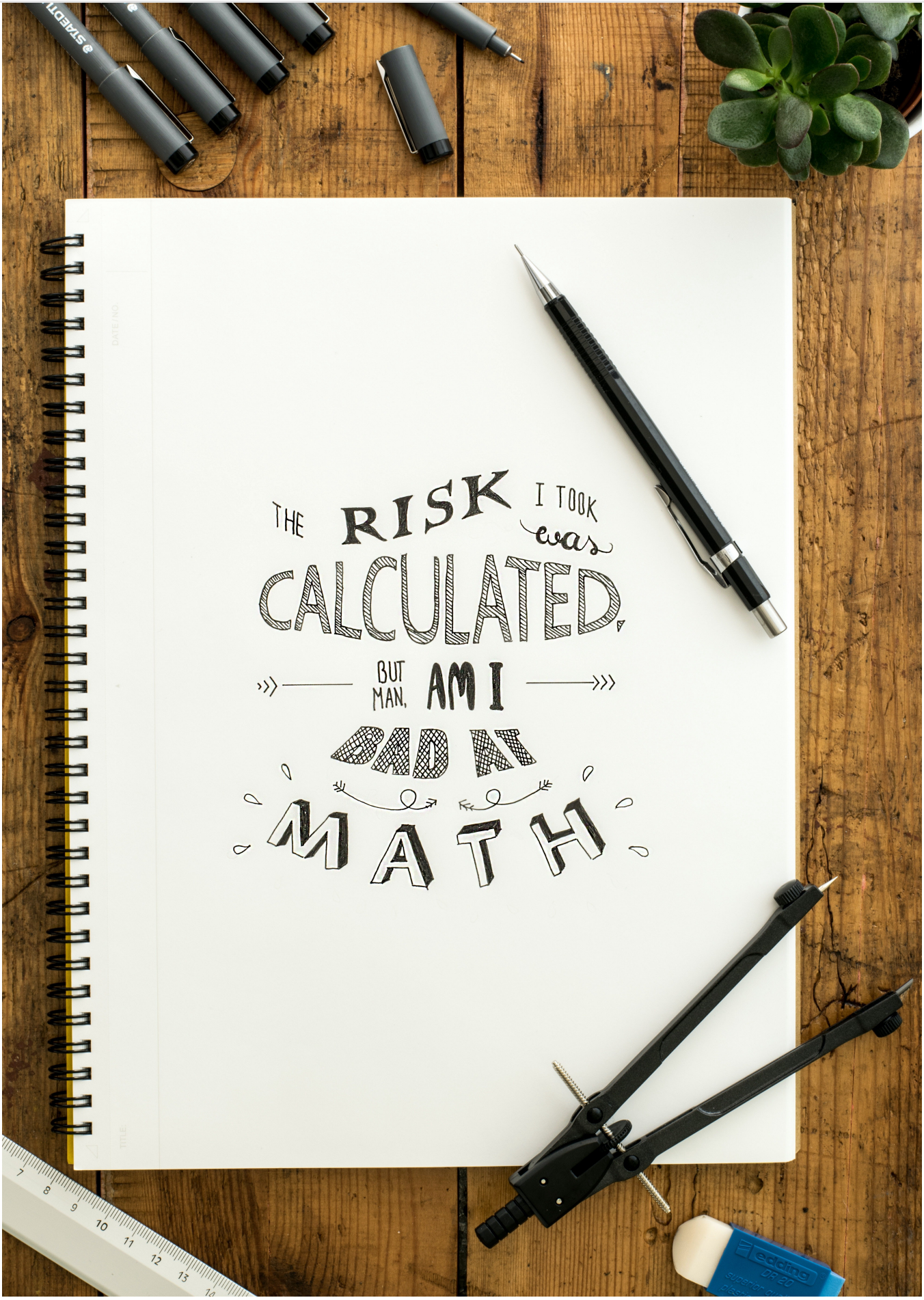


Photo by Lum3n from Pexels

suppose we all also have a basic intuition of how a credit score is calculated, or which factors affect it. Refer to my previous article for some further details on what a credit score is.

In this article, we will go through detailed steps to develop a data-driven credit risk model in Python to predict the probabilities of default (PD) and assign credit scores to existing or potential borrowers. We will determine credit scores using a highly interpretable, easy to understand and implement scorecard that makes calculating the credit score a breeze.

I will assume a working Python knowledge and a basic understanding of certain statistical and credit risk concepts while working through this case study.

We have a lot to cover, so let's get started.

Preliminary Data Exploration & Splitting

We will use a dataset made available on Kaggle that relates to consumer loans issued by the Lending Club, a US P2P lender. The raw data includes information on over 450,000 consumer loans issued between 2007 and 2014 with almost 75 features, including the current loan status and various attributes related to both borrowers and their payment behavior. Refer to the data dictionary for further details on each column.

The concepts and overall methodology, as explained here, are also applicable to a corporate loan portfolio.

Initial data exploration reveals the following:

- 18 features with more than 80% of missing values. Given the high proportion of missing values, any technique to impute them will most likely result in inaccurate results
- Certain static features not related to credit risk, e.g., `id`, `member_id`, `url`, `title`
- Other forward-looking features that are expected to be populated only once the borrower has defaulted, e.g., `recoveries`, `collection_recovery_fee`. Since our

event has occurred

We will drop all the above features.

Identify Target Variable

Based on the data exploration, our target variable appears to be `loan_status` . A quick look at its unique values and their proportion thereof confirms the same.

```
# explore the unique values in loan_status column
loan_data['loan_status'].value_counts(normalize = True)
```

```
Current                                                 0.480878
Fully Paid                                              0.396193
Charged Off                                             0.091092
Late (31-120 days)                                      0.014798
In Grace Period                                         0.006747
Does not meet the credit policy. Status:Fully Paid      0.004263
Late (16-30 days)                                       0.002612
Default                                                 0.001784
Does not meet the credit policy. Status:Charged Off     0.001632
Name: loan_status, dtype: float64
```

Image 1: Distribution of defaults

Based on domain knowledge, we will classify loans with the following `loan_status` values as being in default (or 0):

- Charged Off
- Default
- Late (31–120 days)
- Does not meet the credit policy. Status:Charged Off

All the other values will be classified as good (or 1).

Data Split

Let us now split our data into the following sets: training (80%) and test (20%). We will perform Repeated Stratified k Fold testing on the training test to preliminary evaluate our model while the test set will remain untouched till final model evaluation. This approach follows the best model evaluation practice.

train/test split so that the distribution of good and bad loans in the test set is the same as that in the pre-split data. This is achieved through the `train_test_split` function's `stratify` parameter.

Splitting our data before any data cleaning or missing value imputation prevents any data leakage from the test set to the training set and results in more accurate model evaluation. Refer to my previous article for further details.

A code snippet for the work performed so far follows:

```
# import the required libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, RepeatedStratifiedKFold, cross_va
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, roc_auc_score, confusion_matrix, precision_recal
from sklearn.feature_selection import f_classif
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
from scipy.stats import chi2_contingency

# import data
loan_data = pd.read_csv('.../loan_data_2007_2014.csv')

# drop columns with more than 80% null values
loan_data.dropna(thresh = loan_data.shape[0]*0.2, how = 'all', axis = 1, inplace = True

#drop all redundant and forward-looking columns
loan_data.drop(columns = ['id', 'member_id', 'sub_grade', 'emp_title', 'url', 'desc', '
                          'zip_code', 'next_pymnt_d', 'recoveries', 'collection_recover
                          'total_rec_prncp', 'total_rec_late_fee'], inplace = True)

# explore the unique values in loan_status column
loan_data['loan_status'].value_counts(normalize = True)

# create a new column based on the loan_status column that will be our target variable
loan_data['good_bad'] = np.where(loan_data.loc[:, 'loan_status'].isin(['Charged Off', '
                                                                       'Late (31-120 da
                                                                       'Does not meet t
                                 0, 1)

```

```
# split data into 80/20 while keeping the distribution of bad loans in test set same as
X = loan_data.drop('good_bad', axis = 1)
y = loan_data['good_bad']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    random_state = 42, stratify = y)

# hard copy the X datasets to avoid Pandas' SetttingWithCopyWarning when we play around
# this is currently an open issue between Pandas and Scikit-Learn teams
X_train, X_test = X_train.copy(), X_test.copy()
```

credit_risk_data_exploration.py hosted with ❤️ by GitHub view raw

Data Cleaning

Next comes some necessary data cleaning tasks as follows:

- Remove text from the `emp_length` column (e.g., years) and convert it to numeric
- For all columns with dates: convert them to Python's `datetime` format, create a new column as a difference between model development date and the respective date feature and then drop the original feature
- Remove text from the `term` column and convert it to numeric

We will define helper functions for each of the above tasks and apply them to the training dataset. Having these helper functions will assist us with performing these same tasks again on the test dataset without repeating our code.

```
# function to clean up the emp_length column, assign 0 to NANs, and convert to numeric
def emp_length_converter(df, column):
    df[column] = df[column].str.replace('\+ years', '')
    df[column] = df[column].str.replace('< 1 year', str(0))
    df[column] = df[column].str.replace(' years', '')
    df[column] = df[column].str.replace(' year', '')
    df[column] = pd.to_numeric(df[column])
    df[column].fillna(value = 0, inplace = True)

'''
function to convert date columns to datetime format and
```

```
    # store current month
    today_date = pd.to_datetime('2020-08-01')
    # convert to datetime format
    df[column] = pd.to_datetime(df[column], format = "%b-%y")
    # calculate the difference in months and add to a new column
    df['mths_since_' + column] = round(pd.to_numeric((today_date - df[column]) /
                                                     np.timedelta64(1, 'M')))
    # make any resulting -ve values to be equal to the max date
    df['mths_since_' + column] = df['mths_since_' + column].apply(
                lambda x: df['mths_since_' + column].max() if x < 0 else x)
    # drop the original date column
    df.drop(columns = [column], inplace = True)

# function to remove 'months' string from the 'term' column and convert it to numeric
def loan_term_converter(df, column):
    df[column] = pd.to_numeric(df[column].str.replace(' months', ''))

# apply these functions to X_train
date_columns(X_train, 'earliest_cr_line')
date_columns(X_train, 'issue_d')
date_columns(X_train, 'last_pymnt_d')
date_columns(X_train, 'last_credit_pull_d')
emp_length_converter(X_train, 'emp_length')
loan_term_converter(X_train, 'term')
```

Feature Selection

Next up, we will perform feature selection to identify the most suitable features for our binary classification problem using the Chi-squared test for categorical features and ANOVA F-statistic for numerical features. Refer to my previous article for further details on these feature selection techniques and why different techniques are applied to categorical and numerical variables.

The p-values, in ascending order, from our Chi-squared test on the categorical features are as below:

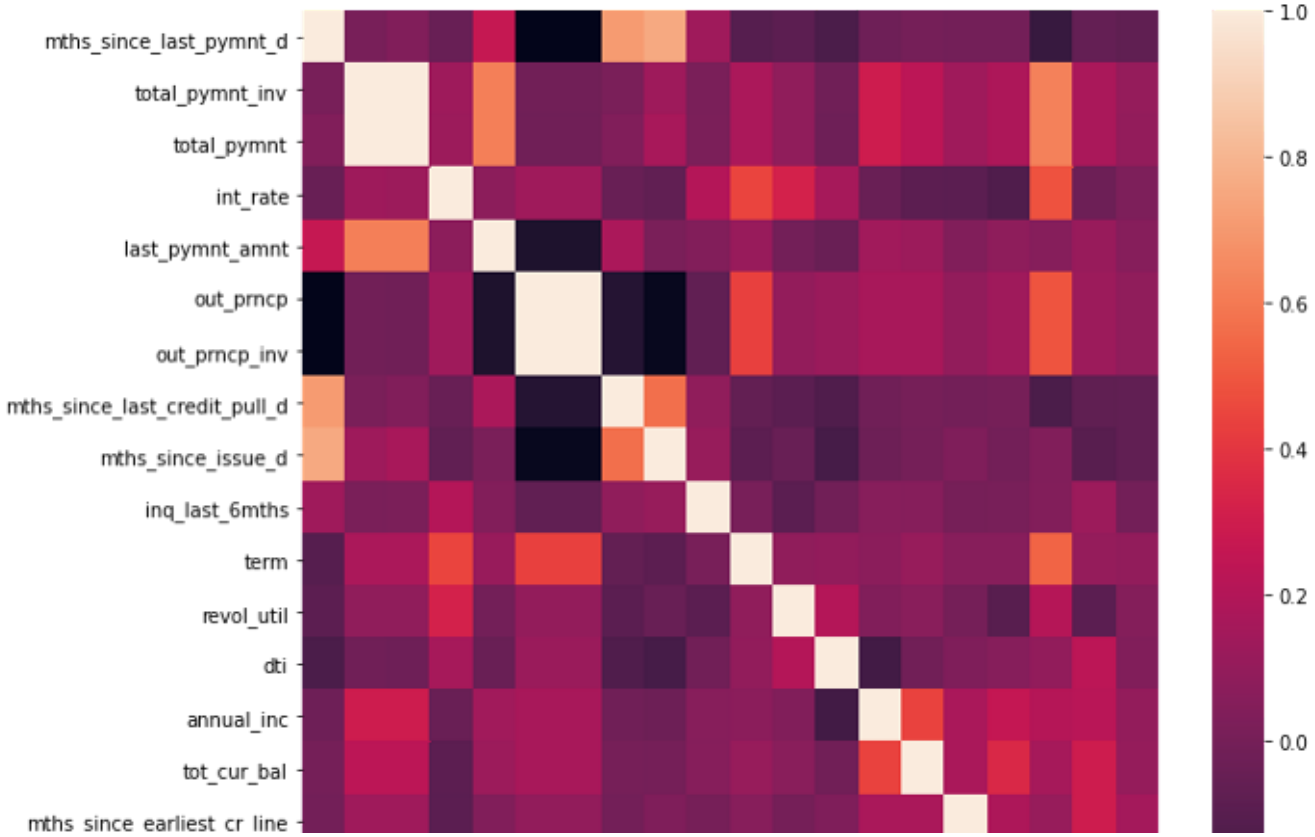
1	home_ownership	0.000000
2	verification_status	0.000000
3	purpose	0.000000
4	addr_state	0.000000
5	initial_list_status	0.000000
6	pymnt_plan	0.000923
7	application_type	1.000000

Image 2: p-values from Chi-squared test

For the sake of simplicity, we will only retain the top four features and drop the rest.

The ANOVA F-statistic for 34 numeric features shows a wide range of F values, from 23,513 to 0.39. We will keep the top 20 features and potentially come back to select more in case our model evaluation results are not reasonable enough.

Next, we will calculate the pair-wise correlations of the selected top 20 numerical features to detect any potentially multicollinear variables. A heat-map of these pair-wise correlations identifies two features (`out_prncp_inv` and `total_pymnt_inv`) as highly correlated. Therefore, we will drop them also for our model.



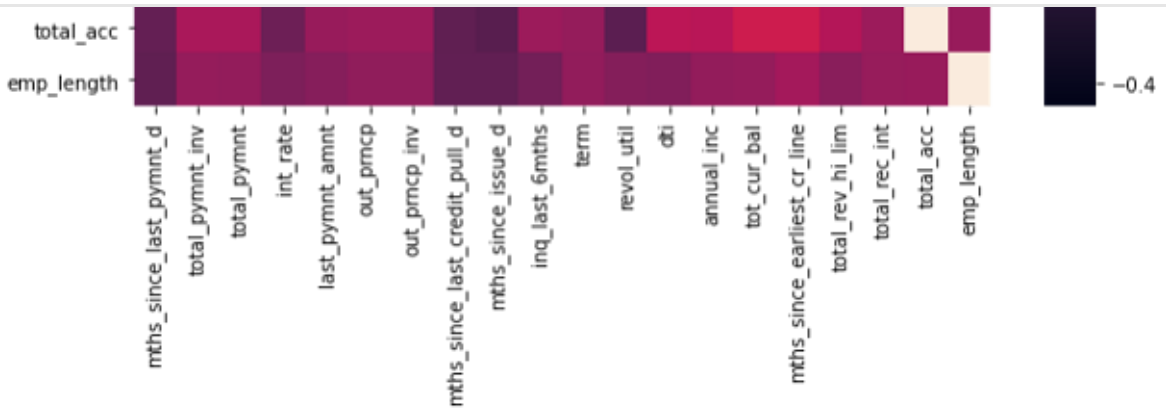


Image 3: Pair-wise correlations

Next, we will simply save all the features to be dropped in a list and define a function to drop them. The code for these feature selection techniques follows:

```
# first divide training data into categorical and numerical subsets
X_train_cat = X_train.select_dtypes(include = 'object').copy()
X_train_num = X_train.select_dtypes(include = 'number').copy()

# define an empty dictionary to store chi-squared test results
chi2_check = {}

# loop over each column in the training set to calculate chi-statistic with the target
for column in X_train_cat:
    chi, p, dof, ex = chi2_contingency(pd.crosstab(y_train, X_train_cat[column]))
    chi2_check.setdefault('Feature',[]).append(column)
    chi2_check.setdefault('p-value',[]).append(round(p, 10))

# convert the dictionary to a DF
chi2_result = pd.DataFrame(data = chi2_check)
chi2_result.sort_values(by = ['p-value'], ascending = True, ignore_index = True, inplac

# since f_class_if does not accept missing values, we will do a very crude imputation c
X_train_num.fillna(X_train_num.mean(), inplace = True)

# Calculate F Statistic and corresponding p values
F_statistic, p_values = f_classif(X_train_num, y_train)

# convert to a DF
ANOVA_F_table = pd.DataFrame(data = {'Numerical_Feature': X_train_num.columns.values,
                                     'F-Score': F_statistic, 'p values': p_values.ro
ANOVA_F_table.sort_values(by = ['F-Score'], ascending = False, ignore_index = True, inp

# save the top 20 numerical features in a list
```

```
corrmat = X_train_num[top_num_features].corr()
plt.figure(figsize=(10,10))
sns.heatmap(corrmat)

# save the names of columns to be dropped in a list
drop_columns_list = ANOVA_F_table.iloc[20:, 0].to_list()
drop_columns_list.extend(chi2_result.iloc[4:, 0].to_list())
drop_columns_list.extend(['out_prncp_inv', 'total_pymnt_inv'])

# function to drop these columns
def col_to_drop(df, columns_list):
    df.drop(columns = columns_list, inplace = True)

# apply to X_train
col_to_drop(X_train, drop_columns_list)
```

credit_risk_feature_selection.py hosted with ❤️ by GitHub view raw

One-Hot Encoding and Update Test Dataset

Next, we will create dummy variables of the four final categorical variables and update the test dataset through all the functions applied so far to the training dataset.

Note a couple of points regarding the way we create dummy variables:

- We will use a particular naming convention for all variables: original variable name, colon, category name
- Generally speaking, in order to avoid multicollinearity, one of the dummy variables is dropped through the `drop_first` parameter of `pd.get_dummies`. However, we will not do so at this stage as we require all the dummy variables to calculate the Weight of Evidence (WoE) and Information Values (IV) of our categories — more on this later. We will drop one dummy variable for each category later on
- We will also not create the dummy variables directly in our training data, as doing so would drop the categorical variable, which we require for WoE calculations. Therefore, we will create a new dataframe of dummy variables and then concatenate it to the original training/test dataframe.

dummy variables. Let me explain this by a practical example.

Consider a categorical feature called `grade` with the following unique values in the pre-split data: A, B, C, and D. Suppose that the proportion of D is very low, and due to the random nature of train/test split, none of the observations with D in the `grade` category is selected in the test set. Therefore, `grade`'s dummy variables in the training data will be grade:A, grade:B, grade:C, and grade:D, but grade:D will not be created as a dummy variable in the test set. We will be unable to apply a fitted model on the test set to make predictions, given the absence of a feature expected to be present by the model. Therefore, we reindex the test set to ensure that it has the same columns as the training data, with any missing columns being added with 0 values. A 0 value is pretty intuitive since that category will never be observed in any of the test samples.

```
# function to create dummy variables
def dummy_creation(df, columns_list):
    df_dummies = []
    for col in columns_list:
        df_dummies.append(pd.get_dummies(df[col], prefix = col, prefix_sep = ':'))
    df_dummies = pd.concat(df_dummies, axis = 1)
    df = pd.concat([df, df_dummies], axis = 1)
    return df

# apply to our final four categorical variables
X_train = dummy_creation(X_train, ['grade', 'home_ownership', 'verification_status', 'p

# update the test data with all functions defined so far
emp_length_converter(X_test, 'emp_length')
date_columns(X_test, 'earliest_cr_line')
date_columns(X_test, 'issue_d')
date_columns(X_test, 'last_pymnt_d')
date_columns(X_test, 'last_credit_pull_d')
loan_term_converter(X_test, 'term')
col_to_drop(X_test, drop_columns_list)
X_test = dummy_creation(X_test, ['grade', 'home_ownership', 'verification_status', 'pur
# reindex the dummied test set variables to make sure all the feature columns in the tr
X_test = X_test.reindex(labels=X_train.columns, axis=1, fill_value=0)
```

credit_risk_dummy_and_test.py hosted with ❤️ by GitHub view raw

WoE is one of the most critical steps before developing a credit risk model, and also quite time-consuming. There are specific custom Python packages and functions available on GitHub and elsewhere to perform this exercise. However, I prefer to do it manually as it allows me a bit more flexibility and control over the process.

But What is WoE and IV?

Weight of Evidence (WoE) and Information Value (IV) are used for feature engineering and selection and are extensively used in the credit scoring domain.

WoE is a measure of the predictive power of an independent variable in relation to the target variable. It measures the extent a specific feature can differentiate between target classes, in our case: good and bad customers.

IV assists with ranking our features based on their relative importance.

According to Baesens et al.[1] and Siddiqi[2], WOE and IV analyses enable one to:

- Consider each variable's independent contribution to the outcome
- Detect linear and non-linear relationships
- Rank variables in terms of its univariate predictive strength
- Visualize the correlations between the variables and the binary outcome
- Seamlessly compare the strength of continuous and categorical variables without creating dummy variables
- Seamlessly handle missing values without imputation. (Note that we have not imputed any missing values so far, this is the reason why. Missing values will be assigned a separate category during the WoE feature engineering step)
- Assess the predictive power of missing values

Weight of Evidence (WoE)

The formula to calculate WoE is as follow:

$$WoE = \ln\left(\frac{\%\ of\ good\ customers}{\%\ of\ bad\ customers}\right)$$

customers and vice versa for a negative WoE value.

Steps for WoE feature engineering

1. Calculate WoE for each unique value (bin) of a categorical variable, e.g., for each of grad:A, grad:B, grad:C, etc.
2. Bin a continuous variable into discrete bins based on its distribution and number of unique observations, maybe using `pd.cut` (called fine classing)
3. Calculate WoE for each derived bin of the continuous variable
4. Once WoE has been calculated for each bin of both categorical and numerical features, combine bins as per the following rules (called coarse classing)

Rules related to combining WoE bins

1. Each bin should have at least 5% of the observations
2. Each bin should be non-zero for both good and bad loans
3. The WOE should be distinct for each category. Similar groups should be aggregated or binned together. It is because the bins with similar WoE have almost the same proportion of good or bad loans, implying the same predictive power
4. The WOE should be monotonic, i.e., either growing or decreasing with the bins
5. Missing values are binned separately

The above rules are generally accepted and well documented in academic literature[3].

Why discretize numerical features

Discretization, or binning, of numerical features, is generally not recommended for machine learning algorithms as it often results in loss of data. However, our end objective here is to create a scorecard based on the credit scoring model eventually. A scorecard is utilized by classifying a new untrained observation (e.g., that from the test dataset) as per the scorecard criteria.

would be given the same score in this category, irrespective of their income. If, however, we discretize the income category into discrete classes (each with different WoE) resulting in multiple categories, then the potential new borrowers would be classified into one of the income categories according to their income and would be scored accordingly.

WoE binning of continuous variables is an established industry practice that has been in place since FICO first developed a commercial scorecard in the 1960s, and there is substantial literature out there to support it. Some of the other rationales to discretize continuous features from the literature are:

- A scorecard is usually legally required to be easily interpretable by a layperson (a requirement imposed by the Basel Accord, almost all central banks, and various lending entities) given the high monetary and non-monetary misclassification costs. This is easily achieved by a scorecard that does not has any continuous variables, with all of them being discretized. Reasons for low or high scores can be easily understood and explained to third parties. All of this makes it easier for scorecards to get 'buy-in' from end-users compared to more complex models
- Another legal requirement for scorecards is that they should be able to separate low and high-risk observations[4]. WoE binning takes care of that as WoE is based on this very concept
- Monotonicity. It is expected from the binning algorithm to divide an input dataset on bins in such a way that if you walk from one bin to another in the same direction, there is a monotonic change of credit risk indicator, i.e., no sudden jumps in the credit score if your income changes. This arises from the underlying assumption that a predictor variable can separate higher risks from lower risks in case of the global non-monotonous relationship[5]
- An underlying assumption of the logistic regression model is that all features have a linear relationship with the log-odds (logit) of the target variable. Is there a difference between someone with an income of $38,000 and someone with $39,000? Most likely not, but treating income as a continuous variable makes this assumption. By categorizing based on WoE, we can let our model decide if there is a statistical difference; if there isn't, they can be combined in the same category

missing values or handle outliers

Information Value (IV)

IV is calculated as follows:

$$IV = \sum (\%\ of\ good\ customers - \%\ of\ bad\ customers) \times WoE$$

According to Siddiqi[2], by convention, the values of IV in credit scoring is interpreted as follows:

Information Value	Variable Predictiveness
Less than 0.02	Not useful for prediction
0.02 to 0.1	Weak predictive Power
0.1 to 0.3	Medium predictive Power
0.3 to 0.5	Strong predictive Power
>0.5	Suspicious Predictive Power

Note that IV is only useful as a feature selection and importance technique when using a binary logistic regression model.

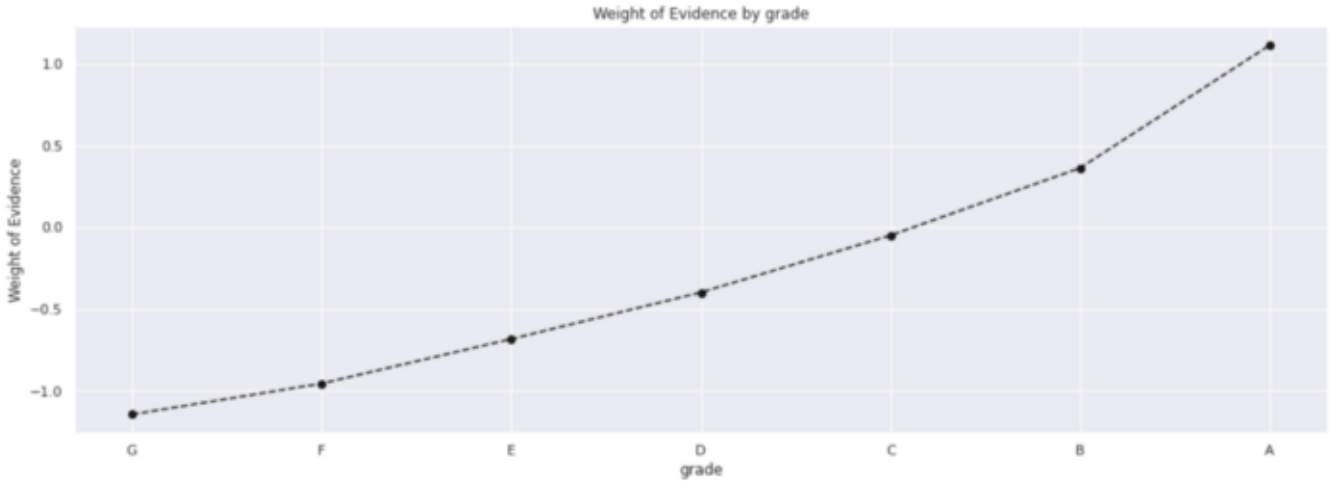
WoE Feature Engineering and IV Calculation for our Data

Enough with the theory, let's now calculate WoE and IV for our training data and perform the required feature engineering. We will define three functions as follows, each one to:

- calculate and display WoE and IV values for categorical variables
- calculate and display WoE and IV values for numerical variables
- plot the WoE values against the bins to help us in visualizing WoE and combining similar WoE bins

	grade	n_obs	prop_good	prop_n_obs	n_good	n_bad	prop_n_good	prop_n_bad	WoE	diff_prop_good	diff_WoE	IV
0	G	2623	0.721693	0.007032	1893.0	730.0	0.005697	0.017904	-1.144981	NaN	NaN	0.292145
1	F	10606	0.758061	0.028432	8040.0	2566.0	0.024198	0.062932	-0.955774	0.036369	0.189207	0.292145
2	E	28590	0.804477	0.076643	23000.0	5590.0	0.069224	0.137097	-0.683340	0.046416	0.272434	0.292145
3	D	61713	0.845527	0.165438	52180.0	9533.0	0.157049	0.233801	-0.397915	0.041050	0.285425	0.292145
4	C	100342	0.885870	0.268993	88890.0	11452.0	0.267536	0.280865	-0.048620	0.040343	0.349295	0.292145
5	B	109344	0.921422	0.293125	100752.0	8592.0	0.303238	0.210723	0.363975	0.035552	0.412595	0.292145
6	A	59810	0.961361	0.160336	57499.0	2311.0	0.173057	0.056678	1.116232	0.039939	0.752257	0.292145

WoE and IV values



WoE plot

Once we have calculated and visualized WoE and IV values, next comes the most tedious task to select which bins to combine and whether to drop any feature given its IV. The shortlisted features that we are left with until this point will be treated in one of the following ways:

- There is no need to combine WoE bins or create a separate missing category given the discrete and monotonic WoE and absence of any missing values: `grade`, `verification_status`, `term`
- Combine WoE bins with very low observations with the neighboring bin: `home_ownership`, `purpose`
- Combine WoE bins with similar WoE values together, potentially with a separate missing category: `int_rate`, `annual_inc`, `dti`, `inq_last_6mths`, `revol_util`,

- Ignore features with a low or very high IV value: `emp_length`, `total_acc`, `last_pymnt_amnt`, `tot_cur_bal`, `mths_since_last_pymnt_d_factor`

Note that for certain numerical features with outliers, we will calculate and plot WoE after excluding them that will be assigned to a separate category of their own.

Once we have explored our features and identified the categories to be created, we will define a custom 'transformer' class using sci-kit learn's `BaseEstimator` and `TransformerMixin` classes. Like other sci-kit learn's ML models, this class can be fit on a dataset to transform it as per our requirements. Another significant advantage of this class is that it can be used as part of a sci-kit learn's `Pipeline` to evaluate our training data using Repeated Stratified k-Fold Cross-Validation. Using a Pipeline in this structured way will allow us to perform cross-validation without any potential data leakage between the training and test folds.

Remember that we have been using all the dummy variables so far, so we will also drop one dummy variable for each category using our custom class to avoid multicollinearity.

The code for our three functions and the transformer class related to WoE and IV follows:

```
# function to calculate WoE and IV of categorical features
# The function takes 3 arguments: a dataframe (X_train_prepr), a string (column name),
def woe_discrete(df, cat_variabe_name, y_df):
    df = pd.concat([df[cat_variabe_name], y_df], axis = 1)
    df = pd.concat([df.groupby(df.columns.values[0], as_index = False)[df.columns.valu
                    df.groupby(df.columns.values[0], as_index = False)[df.columns.valu
    df = df.iloc[:, [0, 1, 3]]
    df.columns = [df.columns.values[0], 'n_obs', 'prop_good']
    df['prop_n_obs'] = df['n_obs'] / df['n_obs'].sum()
    df['n_good'] = df['prop_good'] * df['n_obs']
    df['n_bad'] = (1 - df['prop_good']) * df['n_obs']
    df['prop_n_good'] = df['n_good'] / df['n_good'].sum()
    df['prop_n_bad'] = df['n_bad'] / df['n_bad'].sum()
    df['WoE'] = np.log(df['prop_n_good'] / df['prop_n_bad'])
    df = df.sort_values(['WoE'])
    df = df.reset_index(drop = True)
    df['diff_prop_good'] = df['prop_good'].diff().abs()
```

```
    return df

'''
function to calculate WoE & IV of continuous variables
This is same as the function we defined earlier for discrete variables
The only difference are the 2 commented lines of code in the function that results in
being sorted by continuous variable values
'''
def woe_ordered_continuous(df, continuous_variabe_name, y_df):
    df = pd.concat([df[continuous_variabe_name], y_df], axis = 1)
    df = pd.concat([df.groupby(df.columns.values[0], as_index = False)[df.columns.valu
                    df.groupby(df.columns.values[0], as_index = False)[df.columns.valu
    df = df.iloc[:, [0, 1, 3]]
    df.columns = [df.columns.values[0], 'n_obs', 'prop_good']
    df['prop_n_obs'] = df['n_obs'] / df['n_obs'].sum()
    df['n_good'] = df['prop_good'] * df['n_obs']
    df['n_bad'] = (1 - df['prop_good']) * df['n_obs']
    df['prop_n_good'] = df['n_good'] / df['n_good'].sum()
    df['prop_n_bad'] = df['n_bad'] / df['n_bad'].sum()
    df['WoE'] = np.log(df['prop_n_good'] / df['prop_n_bad'])
    #df = df.sort_values(['WoE'])
    #df = df.reset_index(drop = True)
    df['diff_prop_good'] = df['prop_good'].diff().abs()
    df['diff_WoE'] = df['WoE'].diff().abs()
    df['IV'] = (df['prop_n_good'] - df['prop_n_bad']) * df['WoE']
    df['IV'] = df['IV'].sum()
    return df

# We set the default style of the graphs to the seaborn style.
sns.set()
# function to plot WoE value
def plot_by_woe(df_WoE, rotation_of_x_axis_labels = 0):
    x = np.array(df_WoE.iloc[:, 0].apply(str))
    y = df_WoE['WoE']
    plt.figure(figsize=(18, 6))
    plt.plot(x, y, marker = 'o', linestyle = '--', color = 'k')
    plt.xlabel(df_WoE.columns[0])
    plt.ylabel('Weight of Evidence')
    plt.title(str('Weight of Evidence by ' + df_WoE.columns[0]))
    plt.xticks(rotation = rotation_of_x_axis_labels)

# create a list of all the reference categories, i.e. one category from each of the gl
ref_categories = ['mths_since_last_credit_pull_d:>75', 'mths_since_issue_d:>122', 'mth
                  'total_rec_int:>7,260', 'total_pymnt:>25,000', 'out_prncp:>15,437',
                  'annual_inc:>150K', 'int_rate:>20.281', 'term:60', 'purpose:major_pu
```

```
class WoE_Binning(BaseEstimator, TransformerMixin):
    def __init__(self, X): # no *args or *kargs
        self.X = X
    def fit(self, X, y = None):
        return self #nothing else to do
    def transform(self, X):
        X_new = X.loc[:, 'grade:A': 'grade:G']
        X_new['home_ownership:OWN'] = X.loc[:,'home_ownership:OWN']
        X_new['home_ownership:MORTGAGE'] = X.loc[:,'home_ownership:MORTGAGE']
        X_new['home_ownership:OTHER_NONE_RENT'] = sum([X['home_ownership:OTHER'], X['h
        X_new = pd.concat([X_new, X.loc[:, 'verification_status:Not Verified':'verific
        X_new['purpose:debt_consolidation'] = X.loc[:,'purpose:debt_consolidation']
        X_new['purpose:credit_card'] = X.loc[:,'purpose:credit_card']
        X_new['purpose:major_purch__car__home_impr'] = sum([X['purpose:major_purchase'
        X_new['purpose:educ__ren_en__sm_b__mov'] = sum([X['purpose:educational'], X['p
                                                        X['purpose:moving']])
        X_new['purpose:vacation__house__wedding__med__oth'] = sum([X['purpose:vacatior
                                                                   X['purpose:medical'
        X_new['term:36'] = np.where((X['term'] == 36), 1, 0)
        X_new['term:60'] = np.where((X['term'] == 60), 1, 0)
        X_new['int_rate:<7.071'] = np.where((X['int_rate'] <= 7.071), 1, 0)
        X_new['int_rate:7.071-10.374'] = np.where((X['int_rate'] > 7.071) & (X['int_ra
        X_new['int_rate:10.374-13.676'] = np.where((X['int_rate'] > 10.374) & (X['int_
        X_new['int_rate:13.676-15.74'] = np.where((X['int_rate'] > 13.676) & (X['int_r
        X_new['int_rate:15.74-20.281'] = np.where((X['int_rate'] > 15.74) & (X['int_ra
        X_new['int_rate:>20.281'] = np.where((X['int_rate'] > 20.281), 1, 0)
        X_new['annual_inc:missing'] = np.where(X['annual_inc'].isnull(), 1, 0)
        X_new['annual_inc:<28,555'] = np.where((X['annual_inc'] <= 28555), 1, 0)
        X_new['annual_inc:28,555-37,440'] = np.where((X['annual_inc'] > 28555) & (X['a
        X_new['annual_inc:37,440-61,137'] = np.where((X['annual_inc'] > 37440) & (X['a
        X_new['annual_inc:61,137-81,872'] = np.where((X['annual_inc'] > 61137) & (X['a
        X_new['annual_inc:81,872-102,606'] = np.where((X['annual_inc'] > 81872) & (X['
        X_new['annual_inc:102,606-120,379'] = np.where((X['annual_inc'] > 102606) & (X
        X_new['annual_inc:120,379-150,000'] = np.where((X['annual_inc'] > 120379) & (X
        X_new['annual_inc:>150K'] = np.where((X['annual_inc'] > 150000), 1, 0)
        X_new['dti:<=1.6'] = np.where((X['dti'] <= 1.6), 1, 0)
        X_new['dti:1.6-5.599'] = np.where((X['dti'] > 1.6) & (X['dti'] <= 5.599), 1, 0
        X_new['dti:5.599-10.397'] = np.where((X['dti'] > 5.599) & (X['dti'] <= 10.397)
        X_new['dti:10.397-15.196'] = np.where((X['dti'] > 10.397) & (X['dti'] <= 15.19
        X_new['dti:15.196-19.195'] = np.where((X['dti'] > 15.196) & (X['dti'] <= 19.19
        X_new['dti:19.195-24.794'] = np.where((X['dti'] > 19.195) & (X['dti'] <= 24.79
        X_new['dti:24.794-35.191'] = np.where((X['dti'] > 24.794) & (X['dti'] <= 35.19
        X_new['dti:>35.191'] = np.where((X['dti'] > 35.191), 1, 0)
        X_new['inq_last_6mths:missing'] = np.where(X['inq_last_6mths'].isnull(), 1, 0)
        X_new['inq_last_6mths:0'] = np.where((X['inq_last_6mths'] == 0), 1, 0)
```

```
X_new['inq_last_6mths:>4'] = np.where((X['inq_last_6mths'] > 4), 1, 0)
X_new['revol_util:missing'] = np.where(X['revol_util'].isnull(), 1, 0)
X_new['revol_util:<0.1'] = np.where((X['revol_util'] <= 0.1), 1, 0)
X_new['revol_util:0.1-0.2'] = np.where((X['revol_util'] > 0.1) & (X['revol_uti
X_new['revol_util:0.2-0.3'] = np.where((X['revol_util'] > 0.2) & (X['revol_uti
X_new['revol_util:0.3-0.4'] = np.where((X['revol_util'] > 0.3) & (X['revol_uti
X_new['revol_util:0.4-0.5'] = np.where((X['revol_util'] > 0.4) & (X['revol_uti
X_new['revol_util:0.5-0.6'] = np.where((X['revol_util'] > 0.5) & (X['revol_uti
X_new['revol_util:0.6-0.7'] = np.where((X['revol_util'] > 0.6) & (X['revol_uti
X_new['revol_util:0.7-0.8'] = np.where((X['revol_util'] > 0.7) & (X['revol_uti
X_new['revol_util:0.8-0.9'] = np.where((X['revol_util'] > 0.8) & (X['revol_uti
X_new['revol_util:0.9-1.0'] = np.where((X['revol_util'] > 0.9) & (X['revol_uti
X_new['revol_util:>1.0'] = np.where((X['revol_util'] > 1.0), 1, 0)
X_new['out_prncp:<1,286'] = np.where((X['out_prncp'] <= 1286), 1, 0)
X_new['out_prncp:1,286-6,432'] = np.where((X['out_prncp'] > 1286) & (X['out_pr
X_new['out_prncp:6,432-9,005'] = np.where((X['out_prncp'] > 6432) & (X['out_pr
X_new['out_prncp:9,005-10,291'] = np.where((X['out_prncp'] > 9005) & (X['out_p
X_new['out_prncp:10,291-15,437'] = np.where((X['out_prncp'] > 10291) & (X['out
X_new['out_prncp:>15,437'] = np.where((X['out_prncp'] > 15437), 1, 0)
X_new['total_pymnt:<10,000'] = np.where((X['total_pymnt'] <= 10000), 1, 0)
X_new['total_pymnt:10,000-15,000'] = np.where((X['total_pymnt'] > 10000) & (X[
X_new['total_pymnt:15,000-20,000'] = np.where((X['total_pymnt'] > 15000) & (X[
X_new['total_pymnt:20,000-25,000'] = np.where((X['total_pymnt'] > 20000) & (X[
X_new['total_pymnt:>25,000'] = np.where((X['total_pymnt'] > 25000), 1, 0)
X_new['total_rec_int:<1,089'] = np.where((X['total_rec_int'] <= 1089), 1, 0)
X_new['total_rec_int:1,089-2,541'] = np.where((X['total_rec_int'] > 1089) & (X
X_new['total_rec_int:2,541-4,719'] = np.where((X['total_rec_int'] > 2541) & (X
X_new['total_rec_int:4,719-7,260'] = np.where((X['total_rec_int'] > 4719) & (X
X_new['total_rec_int:>7,260'] = np.where((X['total_rec_int'] > 7260), 1, 0)
X_new['total_rev_hi_lim:missing'] = np.where(X['total_rev_hi_lim'].isnull(), 1
X_new['total_rev_hi_lim:<6,381'] = np.where((X['total_rev_hi_lim'] <= 6381), 1
X_new['total_rev_hi_lim:6,381-19,144'] = np.where((X['total_rev_hi_lim'] > 638
X_new['total_rev_hi_lim:19,144-25,525'] = np.where((X['total_rev_hi_lim'] > 19
X_new['total_rev_hi_lim:25,525-35,097'] = np.where((X['total_rev_hi_lim'] > 25
X_new['total_rev_hi_lim:35,097-54,241'] = np.where((X['total_rev_hi_lim'] > 35
X_new['total_rev_hi_lim:54,241-79,780'] = np.where((X['total_rev_hi_lim'] > 54
X_new['total_rev_hi_lim:>79,780'] = np.where((X['total_rev_hi_lim'] > 79780),
X_new['mths_since_earliest_cr_line:missing'] = np.where(X['mths_since_earliest
X_new['mths_since_earliest_cr_line:<125'] = np.where((X['mths_since_earliest_c
X_new['mths_since_earliest_cr_line:125-167'] = np.where((X['mths_since_earlies
X_new['mths_since_earliest_cr_line:167-249'] = np.where((X['mths_since_earlies
X_new['mths_since_earliest_cr_line:249-331'] = np.where((X['mths_since_earlies
X_new['mths_since_earliest_cr_line:331-434'] = np.where((X['mths_since_earlies
X_new['mths_since_earliest_cr_line:>434'] = np.where((X['mths_since_earliest_c
X_new['mths_since_issue_d:<79'] = np.where((X['mths_since_issue_d'] <= 79), 1,
```

```
        X_new['mths_since_issue_d:>122'] = np.where((X['mths_since_issue_d'] > 122), 1
        X_new['mths_since_last_credit_pull_d:missing'] = np.where(X['mths_since_last_c
        X_new['mths_since_last_credit_pull_d:<56'] = np.where((X['mths_since_last_cred
        X_new['mths_since_last_credit_pull_d:56-61'] = np.where((X['mths_since_last_cr
        X_new['mths_since_last_credit_pull_d:61-75'] = np.where((X['mths_since_last_cr
        X_new['mths_since_last_credit_pull_d:>75'] = np.where((X['mths_since_last_cred
        X_new.drop(columns = ref_categories, inplace = True)
        return X_new
```

Model Training

Finally, we come to the stage where some actual machine learning is involved. We will fit a logistic regression model on our training set and evaluate it using `RepeatedStratifiedKFold`. Note that we have defined the `class_weight` parameter of the `LogisticRegression` class to be `balanced`. This will force the logistic regression model to learn the model coefficients using cost-sensitive learning, i.e., penalize false negatives more than false positives during model training. Cost-sensitive learning is useful for imbalanced datasets, which is usually the case in credit scoring. Refer to my previous article for further details on imbalanced classification problems.

Our evaluation metric will be Area Under the Receiver Operating Characteristic Curve (AUROC), a widely used and accepted metric for credit scoring. RepeatedStratifiedKFold will split the data while preserving the class imbalance and perform k-fold validation multiple times.

After performing k-folds validation on our training set and being satisfied with AUROC, we will fit the pipeline on the entire training set and create a summary table with feature names and the coefficients returned from the model.

```
# define modeling pipeline
reg = LogisticRegression(max_iter=1000, class_weight = 'balanced')
woe_transform = WoE_Binning(X)
pipeline = Pipeline(steps=[('woe', woe_transform), ('model', reg)])

# define cross-validation criteria
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=1)

```

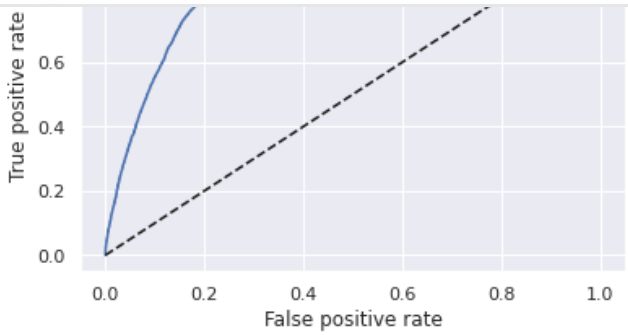
```
AUROC = np.mean(scores)
GINI = AUROC * 2 - 1

# print the mean AUROC score and Gini
print('Mean AUROC: %.4f' % (AUROC))
print('Gini: %.4f' % (GINI))

# fit the pipeline on the whole training set
pipeline.fit(X_train, y_train)

# create a summary table
# first create a transformed training set through our WoE_Binning custom class
X_train_woe_transformed = woe_transform.fit_transform(X_train)
# Store the column names in X_train as a list
feature_name = X_train_woe_transformed.columns.values
# Create a summary table of our logistic regression model
summary_table = pd.DataFrame(columns = ['Feature name'], data = feature_name)
# Create a new column in the dataframe, called 'Coefficients'
summary_table['Coefficients'] = np.transpose(pipeline['model'].coef_)
# Increase the index of every row of the dataframe with 1 to store our model intercept
summary_table.index = summary_table.index + 1
# Assign our model intercept to this new row
summary_table.loc[0] = ['Intercept', pipeline['model'].intercept_[0]]
# Sort the dataframe by index
summary_table.sort_index(inplace = True)
```

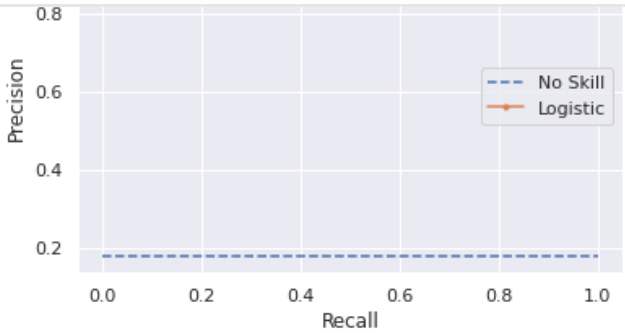
credit_risk_model.py hosted with ❤ by GitHub view raw

Prediction Time

It all comes down to this: apply our trained logistic regression model to predict the probability of default on the test set, which has not been used so far (other than for the generic data cleaning and feature selection tasks). We will save the predicted probabilities of default in a separate dataframe together with the actual classes.

Next, we will draw a ROC curve, PR curve, and calculate AUROC and Gini. Our AUROC on test set comes out to 0.866 with a Gini of 0.732, both being considered as quite acceptable evaluation scores. Our ROC and PR curves will be something like this:





Code for predictions and model evaluation on the test set is:

```
# make preditions on our test set
y_hat_test = pipeline.predict(X_test)
# get the predicted probabilities
y_hat_test_proba = pipeline.predict_proba(X_test)
# select the probabilities of only the positive class (class 1 - default)
y_hat_test_proba = y_hat_test_proba[:][: , 1]

# we will now create a new DF with actual classes and the predicted probabilities
# create a temp y_test DF to reset its index to allow proper concaternation with y_hat_
y_test_temp = y_test.copy()
y_test_temp.reset_index(drop = True, inplace = True)
y_test_proba = pd.concat([y_test_temp, pd.DataFrame(y_hat_test_proba)], axis = 1)
# Rename the columns
y_test_proba.columns = ['y_test_class_actual', 'y_hat_test_proba']
# Makes the index of one dataframe equal to the index of another dataframe.
y_test_proba.index = X_test.index

# get the values required to plot a ROC curve
fpr, tpr, thresholds = roc_curve(y_test_proba['y_test_class_actual'],
                                 y_test_proba['y_hat_test_proba'])
# plot the ROC curve
plt.plot(fpr, tpr)
# plot a secondary diagonal line, with dashed line style and black color to represent a
plt.plot(fpr, fpr, linestyle = '--', color = 'k')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve');

# Calculate the Area Under the Receiver Operating Characteristic Curve (AUROC) on our t
AUROC = roc_auc_score(y_test_proba['y_test_class_actual'], y_test_proba['y_hat_test_pro
# calculate Gini from AUROC
Gini = AUROC * 2 - 1
# print AUROC and Gini
```

```
# draw a PR curve
# calculate the no skill line as the proportion of the positive class
no_skill = len(y_test[y_test == 1]) / len(y)
# plot the no skill precision-recall curve
plt.plot([0, 1], [no_skill, no_skill], linestyle='--', label='No Skill')
# get the values required to plot a PR curve
precision, recall, thresholds = precision_recall_curve(y_test_proba['y_test_class_actua
                                                       y_test_proba['y_hat_test_proba']
# plot PR curve
plt.plot(recall, precision, marker='.', label='Logistic')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.title('PR curve');
```

credit_risk_predictions.py hosted with ❤️ by GitHub view raw

Scorecard Development

The final piece of our puzzle is creating a simple, easy-to-use, and implement credit risk scorecard that can be used by any layperson to calculate an individual's credit score given certain required information about him and his credit history.

Remember the summary table created during the model training phase? We will append all the reference categories that we left out from our model to it, with a coefficient value of 0, together with another column for the original feature name (e.g., `grade` to represent `grade:A`, `grade:B`, etc.).

We will then determine the minimum and maximum scores that our scorecard should spit out. As a starting point, we will use the same range of scores used by FICO: from 300 to 850.

The coefficients returned by the logistic regression model for each feature category are then scaled to our range of credit scores through simple arithmetic. An additional step here is to update the model intercept's credit score through further scaling that will then be used as the starting point of each scoring calculation.

Feature name	Coefficients	Original feature name	Score - Calculation	Score - Preliminary
Intercept	2.948892	Intercept	598.515609	599.0
grade:A	0.980200	grade	24.463996	24.0
grade:B	0.792926	grade	19.789993	20.0
grade:C	0.609305	grade	15.207145	15.0
grade:D	0.487386	grade	12.164269	12.0
grade:E	0.331929	grade	8.284354	8.0
grade:F	0.190615	grade	4.757402	5.0
home_ownership:OWN	-0.048374	home_ownership	-1.207327	-1.0
home_ownership:OTHER_NONE_RENT	-0.104195	home_ownership	-2.600515	-3.0
verification_status:Source Verified	-0.281427	verification_status	-7.023912	-7.0
verification_status:Verified	-0.459417	verification_status	-11.466206	-11.0
purpose:debt_consolidation	-0.315579	purpose	-7.876281	-8.0
purpose:credit_card	-0.222064	purpose	-5.542316	-6.0
purpose:educ__ren_en__sm_b__mov	-0.481318	purpose	-12.012820	-12.0
purpose:vacation__house__wedding__med__oth	0.003981	purpose	0.099355	0.0
term:36	-0.094122	term	-2.349118	-2.0
int_rate:<7.071	0.955408	int_rate	23.845255	24.0
int_rate:7.071-10.374	0.285426	int_rate	7.123724	7.0
int_rate:10.374-13.676	0.056631	int_rate	1.413405	1.0
int_rate:13.676-15.74	0.039281	int_rate	0.980393	1.0

Depending on your circumstances, you may have to manually adjust the Score for a random category to ensure that the minimum and maximum possible scores for any given situation remains 300 and 850. Some trial and error will be involved here.

Calculate Credit Scores for Test Set

Once we have our final scorecard, we are ready to calculate credit scores for all the observations in our test set. Remember, our training and test sets are a simple collection of dummy variables with 1s and 0s representing whether an observation belongs to a specific dummy variable. For example, in the image below, observation 395346 had a C grade, owns its own home, and its verification status was Source Verified.

	Intercept	grade:A	grade:B	grade:C	grade:D	grade:E	grade:F	home_ownership:OWN	home_ownership:OTHER_NONE_RENT	verification_status:Source Verified	verification_status:Verified
395346	1	0	0	1	0	0	0	1	0	1	0
376583	1	1	0	0	0	0	0	0	1	0	0
297790	1	0	0	1	0	0	0	0	1	1	0
47347	1	0	1	0	0	0	0	0	1	0	1
446772	1	0	0	0	1	0	0	0	0	1	0

for each category. Consider the above observations together with the following final scores for the intercept and grade categories from our scorecard:

Feature name	Final Score
Intercept	598
grade:A	24
grade:B	20
grade:C	15
grade:D	12
grade:E	8
grade:F	5

Intuitively, observation 395346 will start with the intercept score of 598 and receive 15 additional points for being in the grade:C category. Similarly, observation 3766583 will be assigned a score of 598 plus 24 for being in the grade:A category. We will automate these calculations across all feature categories using matrix dot multiplication. The final credit score is then a simple sum of individual scores of each feature category applicable for an observation.

Setting Loan Approval Cut-offs

So how do we determine which loans should we approve and reject? What is the ideal credit score cut-off point, i.e., potential borrowers with a credit score higher than this cut-off point will be accepted and those less than it will be rejected? This cut-off point should also strike a fine balance between the expected loan approval and rejection rates.

To find this cut-off, we need to go back to the probability thresholds from the ROC curve. Remember that a ROC curve plots FPR and TPR for all probability thresholds between 0 and 1. Since we aim to minimize FPR while maximizing TPR, the top left corner probability threshold of the curve is what we are looking for. This ideal threshold is calculated using the Youden's J statistic that is a simple difference between TPR and FPR.

The ideal probability threshold in our case comes out to be 0.187. All observations with a predicted probability higher than this should be classified as in Default and vice versa. At first, this ideal threshold appears to be counterintuitive compared to a more intuitive probability threshold of 0.5. But remember that we used the `class_weight` parameter

We then calculate the scaled score at this threshold point. As shown in the code example below, we can also calculate the credit scores and expected approval and rejection rates at each threshold from the ROC curve. This can help the business to further manually tweak the score cut-off based on their requirements.

All the code related to scorecard development is below:

```
# create a new dataframe with one column with values from the 'reference_categories' l
df_ref_categories = pd.DataFrame(ref_categories, columns = ['Feature name'])
# We create a second column, called 'Coefficients', which contains only 0 values.
df_ref_categories['Coefficients'] = 0

# Concatenates two dataframes
df_scorecard = pd.concat([summary_table, df_ref_categories])
# reset the index
df_scorecard.reset_index(inplace = True)

# create a new column, called 'Original feature name', which contains the value of the
df_scorecard['Original feature name'] = df_scorecard['Feature name'].str.split(':').st

# Define the min and max threshholds for our scorecard
min_score = 300
max_score = 850

# calculate the sum of the minimum coefficients of each category within the original f
min_sum_coef = df_scorecard.groupby('Original feature name')['Coefficients'].min().sum
# calculate the sum of the maximum coefficients of each category within the original f
max_sum_coef = df_scorecard.groupby('Original feature name')['Coefficients'].max().sum
# create a new column that has the imputed calculated Score based scaled from the coef
df_scorecard['Score - Calculation'] = df_scorecard['Coefficients'] * (max_score - min_
        max_sum_coef - min_sum_coef)
# update the calculated score of the Intercept
df_scorecard.loc[0, 'Score - Calculation'] = (
        (df_scorecard.loc[0,'Coefficients'] - min_sum_coef) /
        (max_sum_coef - min_sum_coef
        )) * (max_score - min_score) + min_score
# round the values of the 'Score - Calculation' column and store them in a new column
df_scorecard['Score - Preliminary'] = df_scorecard['Score - Calculation'].round()

# check the min and max possible scores of our scorecard
min_sum_score_prel = df_scorecard.groupby('Original feature name')['Score - Preliminar
max_sum_score_prel = df_scorecard.groupby('Original feature name')['Score - Preliminar
print(min_sum_score_prel)
```

```
# so both our min and max scores are out by +1. we need to manually adjust this
# Which one? We'll evaluate based on the rounding differences of the minimum category
pd.options.display.max_rows = 102
df_scorecard['Difference'] = df_scorecard['Score - Preliminary'] - df_scorecard['Score

# look like we can get by deducting 1 from the Intercept
df_scorecard['Score - Final'] = df_scorecard['Score - Preliminary']
df_scorecard.loc[0, 'Score - Final'] = 598

# Recheck min and max possible scores
print(df_scorecard.groupby('Original feature name')['Score - Final'].min().sum())
print(df_scorecard.groupby('Original feature name')['Score - Final'].max().sum())

# calculate credit scores for test set
# first create a transformed test set through our WoE_Binning custom class
X_test_woe_transformed = woe_transform.fit_transform(X_test)
# insert an Intercept column in its beginning to align with the # of rows in scorecard
X_test_woe_transformed.insert(0, 'Intercept', 1)

# get the list of our final scorecard scores
scorecard_scores = df_scorecard['Score - Final']
# check the shapes of test set and scorecard before doing matrix dot multiplication
print(X_test_woe_transformed.shape)
print(scorecard_scores.shape)

# we can see that the test set has 17 less columns than the rows in scorecard due to t
# since the reference categories will always be scored as 0 based on the scorecard,
# it is safe to add these categories to the end of test set with 0 values
X_test_woe_transformed = pd.concat([X_test_woe_transformed,
                                    pd.DataFrame(dict.fromkeys(ref_categories, [0]
                                                               * len(X_test_woe_transf
                                                          index = X_test_woe_transformed.index)
# Need to reshape scorecard_scores so that it is (102,1) to allow for matrix dot multi
scorecard_scores = scorecard_scores.values.reshape(102, 1)
print(X_test_woe_transformed.shape)
print(scorecard_scores.shape)

# matrix dot multiplication of test set with scorecard scores
y_scores = X_test_woe_transformed.dot(scorecard_scores)

# Score cutoff for loan approvals
# Calculate Youden's J-Statistic to identify the best threshhold
J = tpr - fpr
# locate the index of the largest J
ix = np.argmax(J)
```

```
# create a new DF comprising of the thresholds from the ROC output
df_cutoffs = pd.DataFrame(thresholds, columns = ['thresholds'])
# calcue Score corresponding to each threshold
df_cutoffs['Score'] = ((np.log(df_cutoffs['thresholds'] / (1 - df_cutoffs['thresholds'
                       ((max_score - min_score) / (max_sum_coef - min_sum_coef)) + mir

# define a function called 'n_approved' which assigns a value of 1 if a predicted prob
# is greater than the parameter p, which is a threshold, and a value of 0, if it is nc
# Then it sums the column.
# Thus, for given any percentage values, the function will return
# the number of rows wih estimated probabilites greater than the threshold.
def n_approved(p):
    return np.where(y_test_proba['y_hat_test_proba'] >= p, 1, 0).sum()
# Assuming that all credit applications above a given probability of being 'good' will
# when we apply the 'n_approved' function to a threshold, it will return the number of
# Thus, here we calculate the number of approved appliations for al thresholds.
df_cutoffs['N Approved'] = df_cutoffs['thresholds'].apply(n_approved)
# Then, we calculate the number of rejected applications for each threshold.
# It is the difference between the total number of applications and the approved appli
df_cutoffs['N Rejected'] = y_test_proba['y_hat_test_proba'].shape[0] - df_cutoffs['N A
# Approval rate equalts the ratio of the approved applications and all applications.
df_cutoffs['Approval Rate'] = df_cutoffs['N Approved'] / y_test_proba['y_hat_test_prob
# Rejection rate equals one minus approval rate.
df_cutoffs['Rejection Rate'] = 1 - df_cutoffs['Approval Rate']

# let's have a look at the approval and rejection rates at our ideal threshold
df_cutoffs[df_cutoffs['thresholds'].between(0.18657, 0.18658)]
```

Conclusion

Well, there you have it — a complete working PD model and credit scorecard! The complete notebook is available here on GitHub. Feel free to play around with it or comment in case of any clarifications required or other queries.

As always, feel free to reach out to me if you would like to discuss anything related to data analytics, machine learning, financial analysis, or financial analytics.

Till next time, rock on!

References

Weight of Evidence and Information Value Explained

[1] Baesens, B., Roesch, D., & Scheule, H. (2016). Credit risk analytics: Measurement techniques, applications, and examples in SAS. John Wiley & Sons.

[2] Siddiqi, N. (2012). Credit risk scorecards: developing and implementing intelligent credit scoring. John Wiley & Sons.

[3] Thomas, L., Edelman, D. & Crook, J. (2002). Credit Scoring and its Applications.

[4] Mays, E. (2001). Handbook of Credit Scoring. Glanelake Publishing Company.

[5] Mironchyk, P. & Tchistiakov, V. (2017). Monotone optimal binning algorithm for credit risk modeling.