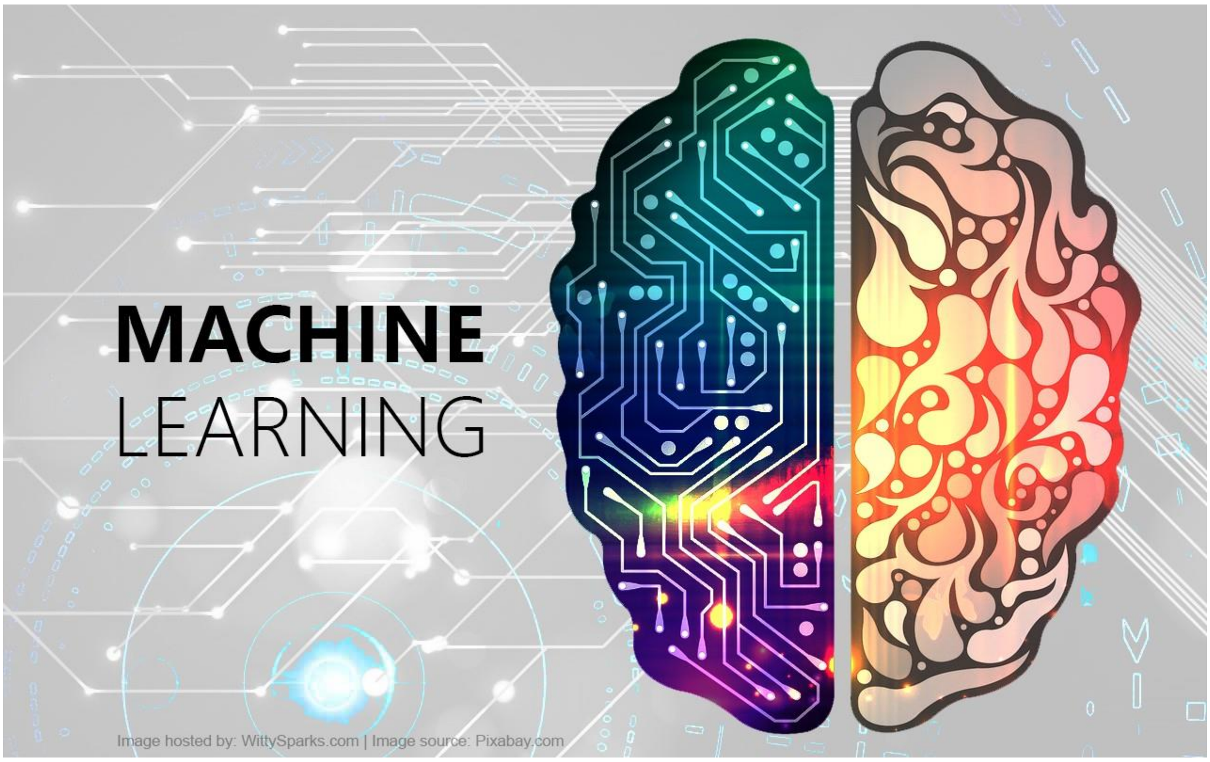
Home Credit Default Risk Prediction Using Machine Learning



IT17011662

Dissanayaka D.M.A.S.

Wednesday, June 10, 2020

Sri Lanka Institute of Information Technology

Table of Content

Contents

Table of Figures

Home Credit Default Risk Prediction Using Machine Learning

1 Introduction

A lot of people in the world struggling to get loans for various purposes. Numerous financial institutes offer loans to people with insufficient or non-existent. But the problem with this system is, those financial institutes cannot be sure about their clients with the repayments of the loans. In this case, the best option is to evaluate the customers before giving the loans to them. If we take a bank for an example, they evaluate hundreds of applications for a day. This method is time-consuming, and the result of the evaluation is not very accurate. As an answer to this problem, we can build a machine learning model with the data of previous applications to predict the likelihood of the loan repayments. With the implementation of this machine learning model, the financial institutes can ensure that the clients capable of repayment are not rejected, and the loans are given with the principals and repayment calendar that will empower their clients to be successful. The main objective of this model is to use previous loan application data to predict whether or not an applicant will be able to repay a loan.

2 Building the Machine Learning Model

This model will be a supervised model because the labels are included in the training data, and the goal is to train the model to predict the labels form the features of the dataset.

The label is a binary variable.

- 0 = will repay the loan in time
- 1 = will have difficulties to repay the loan

2.1 Data Preprocessing

2.1.1 Importing Libraries

Before preprocessing the data, we need to import the libraries. Each library has a different purpose. The libraries can be imported as follows.

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

#Removing unnecessary warnings
import warnings
warnings.filterwarnings("ignore")

#Display plots on the notebook
%matplotlib inline
```

Figure 2:1: Importing Libraries

- **Numpy** – for math calculations
- **Pandas** – for data manipulation
- **Matplotlib** – for plotting
- **Seaborn** – for more plotting options

2.1.2 Reading Dataset

The next step is to read the dataset. It can be done as follows.

```
train = pd.read_csv("../ML_Exam/data/application_train.csv")
test = pd.read_csv("../ML_Exam/data/application_test.csv")

new_test = pd.read_csv("../ML_Exam/data/new_test.csv")
```

Figure 2:2: Reading Dataset

The train is the training dataset, and the test dataset is for testing. The difference between these two is that the testing dataset does not have the column name TARGET, which we are going to predict. The new_test is another testing dataset with limited raws.

2.1.3 Data statistics

Now we can use various commands to look at the dataset statistics. The 'train' dataset can be described as follows.

```
train.describe()
```

	SK_ID_CURR	TARGET	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	AMT_GOODS_PRICE	REGION_POPULATI
count	307511.000000	307511.000000	307511.000000	3.075110e+05	3.075110e+05	307499.000000	3.072330e+05	3
mean	278180.518577	0.080729	0.417052	1.687979e+05	5.990260e+05	27108.573909	5.383962e+05	
std	102790.175348	0.272419	0.722121	2.371231e+05	4.024908e+05	14493.737315	3.694465e+05	
min	100002.000000	0.000000	0.000000	2.565000e+04	4.500000e+04	1615.500000	4.050000e+04	
25%	189145.500000	0.000000	0.000000	1.125000e+05	2.700000e+05	16524.000000	2.385000e+05	
50%	278202.000000	0.000000	0.000000	1.471500e+05	5.135310e+05	24903.000000	4.500000e+05	
75%	367142.500000	0.000000	1.000000	2.025000e+05	8.086500e+05	34596.000000	6.795000e+05	
max	456255.000000	1.000000	19.000000	1.170000e+08	4.050000e+06	258025.500000	4.050000e+06	

8 rows × 106 columns

Figure 2:3: Describing Train dataset

As in Figure 2:3: Describing Train dataset, We can see all the columns and data on the 'train' dataset. It is also possible to take each column to a plot using the library matplotlib as follows.

```
[5]: #Label is a binary variable. 0 = will pay loan on time. 1 = having difficulties to repay the loan
     sns.countplot(train.TARGET)

[5]: <matplotlib.axes._subplots.AxesSubplot at 0x1a697988>
```

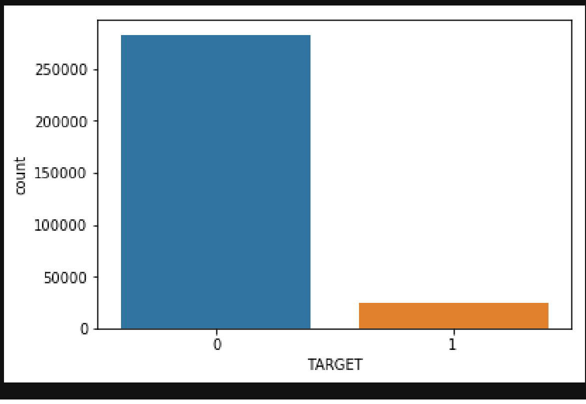


Figure 2:4: Plot diagram of TARGET column

As in Figure 2:4: Plot diagram of TARGET column, we can see the count of the people who will repay the loan amount without any trouble in blue color and the who will be having difficulties to pay the loan in yellow color.

Furthermore, we can get the count of the target column as the following figure.

```
#Train Target value count
train['TARGET'].value_counts()

0    282686
1     24825
Name: TARGET, dtype: int64
```

Figure 2:5: Getting count of columns

```
print("Train dataset dimensions: {}".format(train.shape))
print("Test dataset dimensions: {}".format(test.shape))
print("New_test dataset dimensions: {}".format(new_test.shape))

Train dataset dimensions: (307511, 122)
Test dataset dimensions: (48744, 121)
New_test dataset dimensions: (124, 121)
```

Figure 2:6: Dataset Dimensions

Using the command in Figure 2:6: Dataset Dimensions, it is possible to get the dimensions of all the datasets. As in the above figure, we can see the number of columns and rows in each dataset.

As in the above commands, it is possible to apply those to all the datasets as well, and we can look at each data using various libraries.

2.1.4 Finding Missing Data

We can create the following function to return the column names that have missing values.

```python
#This function is returning the dataframes that has missing column names and percent of missing values
def missing_columns(dataframe):
    missing_values = dataframe.isnull().sum().sort_values(ascending=False)

    missing_values_perc = 100 * missing_values/len(dataframe)

    concat_values = pd.concat([missing_values, missing_values/len(dataframe), missing_values_perc.round(1)],axis=1)

    concat_values.columns = ['Missing Count', 'Missing Count Ratio', 'Missing Count %']

    return concat_values[concat_values.iloc[:,1] != 0]
```

Figure 2:7: Function for retrive missing values

We can execute this function as follows, and it will retrieve all the column names that have missing values.

```python
#Executing Function
missing_columns(train)
```

	Missing Count	Missing Count Ratio	Missing Count %
COMMONAREA_MEDI	214865	0.698723	69.9
COMMONAREA_AVG	214865	0.698723	69.9
COMMONAREA_MODE	214865	0.698723	69.9
NONLIVINGAPARTMENTS_MODE	213514	0.694330	69.4
NONLIVINGAPARTMENTS_MEDI	213514	0.694330	69.4
NONLIVINGAPARTMENTS_AVG	213514	0.694330	69.4
FONDKAPREMONT_MODE	210295	0.683862	68.4

Figure 2:8: Executed function

As in Figure 2:8: Executed function, the function will retrieve the column name, missing count, missing count ratio, and the percentage of the missing count. Also, it is possible to execute this function to retrieve the values of all the datasets as well.

2.1.5 Data Types of Dataset

The data types are one of the essential parts when creating a machine learning model. All the datasets should have the same datatypes and the count of those data to create a very accurate machine learning model. We can check the data types of each dataset as follows.

```
print("Train Dataset: \n{}".format(train.dtypes.value_counts()))
print()
print("Test Dataset: \n{}".format(test.dtypes.value_counts()))
print()
print("New Test Dataset: \n{}".format(new_test.dtypes.value_counts()))
print()
```

```
Train Dataset:
float64    65
int64      41
object     16
dtype: int64
()
Test Dataset:
float64    65
int64      40
object     16
dtype: int64
()
New Test Dataset:
float64    61
int64      44
object     16
dtype: int64
()
```

Figure 2:9: Checking datatypes of datasets

As in Figure 2:9: Checking datatypes of datasets, the numbers are different compared to each dataset. To build the model, we must convert those testing dataset numbers similar to the 'train' dataset. With the following function, we can convert test data type values.

```
#This function converts dataframe to match columns in accordance with the training dataframe
def convert_dtypes(training_df, testing_df, target_name='TARGET'):
    for column_name in training_df.drop([target_name],axis=1).columns:
        testing_df[column_name]= testing_df[column_name].astype(train[column_name].dtype)

    return testing_df
```

Figure 2:10: Datatype conversion function

Now we can apply this function to the new_test dataset.

```
new_test = convert_dtypes(train,new_test)
```

Figure 2:11: Applying function to new test dataset

After executing the function, we can recheck the new dataset as in Figure 2:9: Checking datatypes of datasets. The result should be as follows.

```
Train Dataset:
float64    65
int64      41
object     16
dtype: int64
()
Test Dataset:
float64    65
int64      40
object     16
dtype: int64
()
New Test Dataset:
float64    65
int64      40
object     16
dtype: int64
()
New test dataset converted exacrtly like train dataset
```

Figure 2:12:Rechecked results

2.1.6 Different Classes in Categorical Columns

Another part of the data preprocessing is handling categorical data columns. Before handling them, we have to find each categorical columns. The finding can be done as follows.

```
train.select_dtypes('object').apply(pd.Series.nunique)

NAME_CONTRACT_TYPE             2
CODE_GENDER                    3
FLAG_OWN_CAR                   2
FLAG_OWN_REALTY                2
NAME_TYPE_SUITE                7
NAME_INCOME_TYPE               8
NAME_EDUCATION_TYPE            5
NAME_FAMILY_STATUS             6
NAME_HOUSING_TYPE              6
OCCUPATION_TYPE               18
WEEKDAY_APPR_PROCESS_START     7
ORGANIZATION_TYPE             58
FONDKAPREMONT_MODE             4
HOUSETYPE_MODE                 3
WALLSMATERIAL_MODE             7
EMERGENCYSTATE_MODE            2
dtype: int64
```

Figure 2:13: Finding Different Classes

The above command can be used to find different classes in categorical columns of other datasets as well.

2.1.7 Handling Categorical Variables

Most of the machine learning models cannot learn if the given data is in the text category. To learn from the data, the categorical data should convert into a numerical equivalent. These handlings can be done using **Label Encoding** and **One-Hot Encoding**.

- **Label Encoding:** Label encoding is the process of assigning each unique category in a categorical variable with an integer. This will not create new columns in the datasets. Label encoding function as follows.

```
# Label encode object creation to have less than or equal to 2 unique values
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
transform_counter = 0

#Going through all categorical column
for col in train.select_dtypes('object').columns:
    #selecting columns that have only less than or equest to 2 unique values.
    if pd.Series.nunique(train[col]) <= 2:
        train[col] = le.fit_transform(train[col].astype(str))
        test[col] = le.fit_transform(test[col].astype(str))
        new_test[col] = le.fit_transform(new_test[col].astype(str))

        transform_counter+=1

print("Label Encoded {} columns.".format(transform_counter))
```

Figure 2:14: Label encoding function

This function will go through each categorical column and select only columns where the number of unique values in the category is less than or equal to 2 and encode them.

- **One-Hot Encoding –** One hot encoding will create a new column for each unique category in the categorical variable. Each observation receives a 1 in the column for its corresponding category and a 0 in all other new columns. This encoding can be done as follows.

```
#This encoding method adding more columns
train = pd.get_dummies(train,drop_first=True)
test = pd.get_dummies(test,drop_first=True)
new_test = pd.get_dummies(new_test,drop_first=True)
```

Figure 2:15: One-Hot Encoding

If we view the result of the above command, we can see, this encoding method creates new columns as follows.

```
#Column check
print('Training shape: ', train.shape)
print('Testing shape: ', test.shape)
print('New Testing shape: ',new_test.shape)

('Training shape: ', (307511, 230))
('Testing shape: ', (48744, 226))
('New Testing shape: ', (124, 186))
```

Figure 2:16: Results of one hot encoding

This encoding method resolves one problem, but the column numbers are different after the encoding. The solution to that problem is aligning data.

2.1.8 Column Aligning

First, we should collect the target labels before aligning them as follows.

```
#Target labels collecting
target = train['TARGET']
```

Figure 2:17: Target Label Collecting

Then the aligning can be done as in the following figure.

```
train, test = train.align(test, axis=1, join='inner')

#Adding the stored target data to train dataset
train['TARGET'] = target

#This function adding the missing columns to test dataset and set them to 0
def match_cols(training_set, testing_set, target_label='TARGET'):

    for column in training_set.drop([target_label],axis=1).columns:
        if column not in testing_set.columns:
            testing_set[column]=0
    return testing_set
```

Figure 2:18: Aligning

The function will add missing columns to test dataset and set them to 0

When we execute the above function, we can see the results as follows.

```
#Executing function and checking new test column numbers
new_test=match_cols(train,new_test)
new_test.shape

(124, 226)
```

Figure 2:19: Aligning function results

After the alignment, we can recheck the datasets to check the column count as below figure.

```
print('Training shape: ', train.shape)
print('Testing shape: ', test.shape)
print('New Testing shape: ',new_test.shape)

('Training shape: ', (307511, 227))
('Testing shape: ', (48744, 226))
('New Testing shape: ', (124, 226))
```

Figure 2:20: Recheck column counts

2.1.9 Analyzing Anomalies in the datasets

In a dataset, having anomalies is a regular thing. Those anomalies can be available due to errors in measuring equipment, mistypes, or they could be valid but extreme measurements. One way to check the anomalies in the dataset is to check each column using the 'describe' method.

The first column I chose is the 'DAYS_BIRTH' column. We can see it as follows.

```
(train['DAYS_BIRTH']/-365).describe(

count    307511.000000
mean         43.936973
std          11.956133
min          20.517808
25%          34.008219
50%          43.150685
75%          53.923288
max          69.120548
Name: DAYS_BIRTH, dtype: float64
```

Figure 2:21: DAYS_BIRTH Column

We can get the chart plot as well.

```
#Graphing the result
fig, ax = plt.subplots(figsize =(12,7))
sns.distplot(train['DAYS_BIRTH']/-365,bins=5, kde=False)
plt.xlabel("Age of the Clients in Years")
```

```
Text(0.5,0,'Age of the Clients in Years')
```

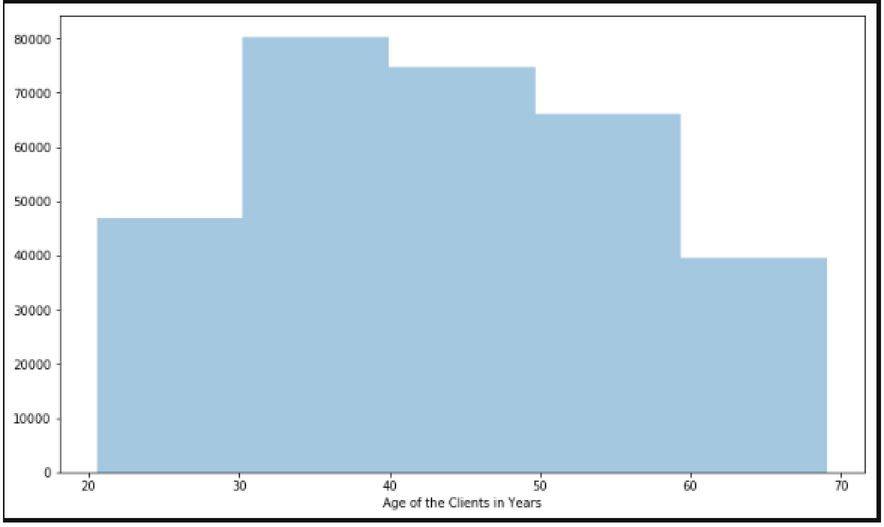


Figure 2:22: DAYS_BIRTH Chart plot

According to the above data, I cannot see any anomalies. Let's see the column DAYS_EMPLOYED.

```
(train['DAYS_EMPLOYED']/365).describe()

count    307511.000000
mean        174.835742
std         387.056895
min         -49.073973
25%          -7.561644
50%          -3.323288
75%          -0.791781
max        1000.665753
Name: DAYS_EMPLOYED, dtype: float64
```

Figure 2:23: DAYS_EMPLOYED describe

The above command says the maximum years of employment are 1000 years. This is definitely an anomaly.

So we found an anomaly, then we can look for how many are they here as follows.

```
#Counting anomalies in this section
thou_anomalies = train[(train['DAYS_EMPLOYED']/365>=900) & (train['DAYS_EMPLOYED']/365<=1100)]
len(thou_anomalies)

55374
```

Figure 2:24: Counting anomalies

These anomalies can be fixed as follows.

```
#Anomalous flag column creation
train['DAYS_EMPLOYED_ANOM'] = train["DAYS_EMPLOYED"] == 365243

#Replacing anomalous values with NaN
train['DAYS_EMPLOYED'] = train['DAYS_EMPLOYED'].replace({365243: np.nan})
```

Figure 2:25: Fixing anomalies

The above code is creating flag columns and replacing them with NaN values. As below, we can use this method to fix all the anomalies in all the datasets.

```
test['DAYS_EMPLOYED_ANOM'] = test["DAYS_EMPLOYED"] == 365243

test['DAYS_EMPLOYED'] = test['DAYS_EMPLOYED'].replace({365243: np.nan})

new_test['DAYS_EMPLOYED_ANOM'] = new_test["DAYS_EMPLOYED"] == 365243

new_test['DAYS_EMPLOYED'] = new_test['DAYS_EMPLOYED'].replace({365243: np.nan})
```

Figure 2:26: Fixing anomalies in all datasets

2.1.10 Correlation Features

Correlation is a statistical measure that indicates the extent to which two or more variables fluctuate together. A positive correlation indicates the extent to which those variables increase or decrease in parallel; a negative correlation indicates the extent to which one variable increases as the other decreases.

- Finding the most correlated features for the TARGET variable

```
print(corr_train.sort_values().tail(10))
corr_train.sort_values().head(10)
```

```
REG_CITY_NOT_WORK_CITY         0.050994
DAYS_ID_PUBLISH                0.051457
CODE_GENDER_M                  0.054713
DAYS_LAST_PHONE_CHANGE         0.055218
NAME_INCOME_TYPE_Working       0.057481
REGION_RATING_CLIENT           0.058899
REGION_RATING_CLIENT_W_CITY    0.060893
DAYS_EMPLOYED                  0.074958
DAYS_BIRTH                     0.078239
TARGET                         1.000000
Name: TARGET, dtype: float64
EXT_SOURCE_3                          -0.178919
EXT_SOURCE_2                          -0.160472
EXT_SOURCE_1                          -0.155317
NAME_EDUCATION_TYPE_Higher education  -0.056593
NAME_INCOME_TYPE_Pensioner            -0.046209
DAYS_EMPLOYED_ANOM                    -0.045987
ORGANIZATION_TYPE_XNA                 -0.045987
FLOORSMAX_AVG                         -0.044003
FLOORSMAX_MEDI                        -0.043768
FLOORSMAX_MODE                        -0.043226
Name: TARGET, dtype: float64
```

Figure 2:27: Top 10 most positively and negatively correlated features

Since EXT_SOURCE_3, EXT_SOURCE_2, EXT_SOURCE_1, and DAYS_BIRTH are highly correlated (Relatively), let's explore the possibility of having them as interaction variables.

Now we can start to fill up the missing values for the most correlated variables as follows.

```
from sklearn.preprocessing import Imputer

poly_fitting_vars = ['EXT_SOURCE_3', 'EXT_SOURCE_2', 'EXT_SOURCE_1', 'DAYS_BIRTH' ]

imputer = Imputer(missing_values='NaN', strategy='median')

train[poly_fitting_vars] = imputer.fit_transform(train[poly_fitting_vars])

train[poly_fitting_vars].shape

(307511, 4)

test[poly_fitting_vars] = imputer.transform(test[poly_fitting_vars])

test[poly_fitting_vars].shape

(48744, 4)
```

Figure 2:28: filling the missing vaulus

After that, we are able to generate interaction variables as follows.

```
from sklearn.preprocessing import PolynomialFeatures

poly_feat = PolynomialFeatures(degree=4)

poly_interaction_train = poly_feat.fit_transform(train[poly_fitting_vars])

poly_interaction_train.shape

(307511L, 70L)

poly_interaction_test = poly_feat.fit_transform(test[poly_fitting_vars])

poly_interaction_test.shape

(48744L, 70L)

poly_interaction_new_test = poly_feat.fit_transform(new_test[poly_fitting_vars])

poly_interaction_new_test.shape

(124L, 70L)
```

Figure 2:29: Generating interaction variables

The next step is to get the names of the columns which have the highest correlation – 1 & TARGET can be dropped. This is possible to get by running the following code.

```
set(interaction.head(15).index).union(interaction.tail(15).index).difference(set({'1','TARGET'}))

{'EXT_SOURCE_2',
 'EXT_SOURCE_2 DAYS_BIRTH',
 'EXT_SOURCE_2 EXT_SOURCE_1',
 'EXT_SOURCE_2 EXT_SOURCE_1 DAYS_BIRTH',
 'EXT_SOURCE_2^2 DAYS_BIRTH',
 'EXT_SOURCE_2^2 EXT_SOURCE_1',
 'EXT_SOURCE_2^2 EXT_SOURCE_1 DAYS_BIRTH',
 'EXT_SOURCE_2^3 DAYS_BIRTH',
 'EXT_SOURCE_3',
 'EXT_SOURCE_3 DAYS_BIRTH',
 'EXT_SOURCE_3 EXT_SOURCE_1',
 'EXT_SOURCE_3 EXT_SOURCE_1 DAYS_BIRTH',
 'EXT_SOURCE_3 EXT_SOURCE_2',
 'EXT_SOURCE_3 EXT_SOURCE_2 DAYS_BIRTH',
 'EXT_SOURCE_3 EXT_SOURCE_2 DAYS_BIRTH^2',
 'EXT_SOURCE_3 EXT_SOURCE_2 EXT_SOURCE_1',
 'EXT_SOURCE_3 EXT_SOURCE_2 EXT_SOURCE_1 DAYS_BIRTH',
 'EXT_SOURCE_3 EXT_SOURCE_2 EXT_SOURCE_1^2',
 'EXT_SOURCE_3 EXT_SOURCE_2^2',
 'EXT_SOURCE_3 EXT_SOURCE_2^2 DAYS_BIRTH',
 'EXT_SOURCE_3 EXT_SOURCE_2^2 EXT_SOURCE_1',
 'EXT_SOURCE_3 EXT_SOURCE_2^3',
 'EXT_SOURCE_3^2 DAYS_BIRTH',
 'EXT_SOURCE_3^2 EXT_SOURCE_1 DAYS_BIRTH',
 'EXT_SOURCE_3^2 EXT_SOURCE_2',
 'EXT_SOURCE_3^2 EXT_SOURCE_2 DAYS_BIRTH',
 'EXT_SOURCE_3^2 EXT_SOURCE_2 EXT_SOURCE_1',
```

Figure 2:30: Names of Columns which Have Highest Correlation - 1 & TARGET Can be Dropped

Then we can select the Selecting the columns which have the highest correlation to 'TARGET'. Columns '1' and 'TARGET' are not necessary as follows.

```
selected_inter_variables = list(set(interaction.head(15).index).union(interaction.tail(15).index).difference(set({'1','TARGE

# look at the selected features
poly_interaction_train[selected_inter_variables].head()
```

	EXT_SOURCE_3^2 EXT_SOURCE_2	EXT_SOURCE_3 EXT_SOURCE_2^2 DAYS_BIRTH	EXT_SOURCE_3^2 EXT_SOURCE_2^2	EXT_SOURCE_3 EXT_SOURCE_2^2	EXT_SOURCE_3 EXT_SOURCE_2^3	EXT_SOURCE_2^2 EXT_SOURCE_1	EXT_SOURCE_3 EXT_SOURCE_2 DAYS_BIRTH^2	EXT_SOURCE_3^ EXT_SOURCE_ EXT_SOURCE_
0	0.005108	-91.172960	0.001343	0.009637	0.002534	0.005741	3.280441e+06	0.00042
1	0.178286	-3474.605044	0.110938	0.207254	0.128963	0.120520	9.361535e+07	0.05549
2	0.295894	-4294.187521	0.164491	0.225464	0.125338	0.156373	1.471224e+08	0.14972
3	0.186365	-4303.904125	0.121220	0.226462	0.147300	0.214075	1.257541e+08	0.09430
4	0.092471	-1111.296208	0.029844	0.055754	0.017994	0.052705	6.863256e+07	0.04679

5 rows × 28 columns

Figure 2:31: the columns which have the highest correlation to 'TARGET'. Columns '1' and 'TARGET' are not necessary

As in the above figure, this method can be used for all the datasets.

2.1.11 Column Dropping

Now, as the next step, we can start to drop unnecessary columns.

We can get a list of columns that possible to drop as in the below image.

```
unselected_cols = [element for element in poly_interaction_train.columns if element not in selected_inter_variables]
```

Figure 2:32: Column list

First, I'm going to drop unselected columns of 'train' and 'test' data. It can be done as follows.

```
poly_interaction_train = poly_interaction_train.drop(unselected_cols,axis=1)

poly_interaction_test = poly_interaction_test.drop(list(set(unselected_cols).difference({'TARGET'})),axis=1)

poly_interaction_new_test = poly_interaction_new_test.drop(list(set(unselected_cols).difference({'TARGET'})),axis=1)
```

Figure 2:33: Dropping unselected columns

The next step is to merge polynomial features into the original dataset.

```
train = train.join(poly_interaction_train.drop(['EXT_SOURCE_2', 'EXT_SOURCE_3'],axis=1))

test = test.join(poly_interaction_test.drop(['EXT_SOURCE_2', 'EXT_SOURCE_3'],axis=1))

new_test = new_test.join(poly_interaction_new_test.drop(['EXT_SOURCE_2', 'EXT_SOURCE_3'],axis=1))
```

Figure 2:34: Merging polynomial features

Now we can check the merged data frame dimensions as follows.

```
print("The train dataset dimensions: {}".format(train.shape))
print("The test dataset dimensions: {}".format(test.shape))
print("The new test dataset dimensions: {}".format(new_test.shape))

The train dataset dimensions: (307511, 254)
The test dataset dimensions: (48744, 253)
The new test dataset dimensions: (124, 253)
```

Figure 2:35: Merged dataframe dimensions

Now we have cleaned, preprocessed datasets to create our machine learning model.

2.2 Building the Machine Learning Model

2.2.1 Feature Imputing

This is the process of filling the missing data on columns during captureing data. In this case imputation is done for the median value of every column

```
from sklearn.preprocessing import MinMaxScaler, Imputer

features = list(set(train.columns).difference({'TARGET'}))

imputer = Imputer(strategy="median")
```

Figure 2:36: Feature Imputing

2.2.2 Feature scaling

This means standardization of feature data or independent variables in data processing. Following code segment will do that.

```
new_test = new_test.replace(to_replace=np.inf,value=0)

scaler = MinMaxScaler(feature_range = (0, 1))

imputer.fit(train.drop(['TARGET'],axis=1))

Imputer(axis=0, copy=True, missing_values='NaN', strategy='median', verbose=0)

train_transformed = imputer.transform(train.drop(['TARGET'],axis=1))

test_transformed = imputer.transform(test)

new_test_transformed = imputer.transform(new_test)

train_transformed = scaler.fit_transform(train_transformed)

test_transformed = scaler.transform(test_transformed)

new_test_transformed = scaler.transform(new_test_transformed)

print("The train dataset dimensions: {}".format(train_transformed.shape))
print("The test dataset dimensions: {}".format(test_transformed.shape))
print("The new test dataset dimensions: {}".format(new_test_transformed.shape))

The train dataset dimensions: (307511L, 253L)
The test dataset dimensions: (48744L, 253L)
The new test dataset dimensions: (124L, 253L)
```

Figure 2:37: Feature scaling

As in Figure 2:37: Feature scaling, we can see the counts of coulmns of test data sets are the same.

2.2.3 Rogistic Regression

This algorithm measures the relationship between the dependant variable which is the target label to predict and the one or many indipendant varials as knowsn as features, by estimating probability using logistic function.

The task of the logistic function is to transform binary values to make a prediction.

This algorithm can be start as follows.

```
from sklearn.linear_model import LogisticRegression

logistic_regressor = LogisticRegression(C = 2)
```

```
logistic_regressor.fit(X_training_set,y_training_set)
```

```
LogisticRegression(C=2, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

```
log_regression_pred = logistic_regressor.predict(X_validation_set)
```

```
logistic_new = logistic_regressor.predict(new_test_transformed)
```

```
pd.DataFrame({'target':logistic_new})['target'].value_counts()
```

```
0    124
Name: target, dtype: int64
```

Figure 2:38: Starting algorithm

2.2.4 Accuracy Metrics

Accuracy metrics have 4 values which are True Positives, True Negatives, False Positives, False Negatives. To get a good accuracy from the machine learning model, these values are essential.

Now we can get the results from the model as follows.

```
from sklearn.metrics import accuracy_score,classification_report, roc_auc_score
print("The accuracy in general is : ", accuracy_score(y_validation_set,log_regression_pred))
print("\n")
print("The classification report is as follows:\n", classification_report(y_validation_set,log_regression_pred))
print("ROC AUC score is: ",roc_auc_score(y_validation_set,log_regression_pred))
```

```
('The accuracy in general is : ', 0.9200130076173395)


('The classification report is as follows:\n', u'              precision    recall  f1-score   support\n\n           0       0.92      1.00
0.96     93362\n           1       0.50      0.01      0.02      8117\n\n   micro avg       0.92      0.92      0.92    101479\n   macro avg
0.71      0.50      0.49    101479\nweighted avg       0.89      0.92      0.88    101479\n')
('ROC AUC score is: ', 0.5034871041311515)
```

Figure 2:39: Prediction

As in the above figure, we have an accuracy of 92 percent. And the ROC AUC score is 0.50. ROC AUC means the area under receiver operating characteristics. Simply ROC means a curve of true positives versus the false-positive rate at different thresholds. When it comes to the AUROC curve is the probability that a classifier will be more confidant that a randomly chosen positive example is actually positive than that a randomly chosen negative example is positive.

Our main objective is to predict the probability of not paying a loan, so we use the model predict_proba method. This returns an m x 2 array where m is the number of data points. The first column is the probability of the target being 0, and the second column is the probability of the target being 1. We want the probability the loan is not repaid, so we will select the second column as follows.

```
log_regression_pred_test = logistic_regressor.predict_proba(test_transformed)
```

```
# selecting the second column
log_regression_pred_test[:,1]
```

```
array([0.05342986, 0.20067074, 0.04960973, ..., 0.0489945 , 0.05287354,
       0.12652108])
```

Figure 2:40: Selecting second column

2.2.5 Output

Now our model is complete, and the output CSV file can get as follows.

```
submission_log_regression.to_csv("log_regression.csv",index=False)
```

Figure 2:41: Output CSV file