By Jeheon Kim
Side-project: Loan Prediction

Problem Statement

The finance company deals in all home loans. They have presence across all urban, semi urban, and rural areas. Customer first apply for home loan after that company validates the customer eligibility for loan.

The company wants to automate the loan eligibility process (real time) based on customer detail provided while filling online application form. These details are Gender, Marital Status, Education, Number of Dependents, Income, Loan Amount, Credit History, and others. To automate this process, they have given a problem to identify the customers segments, those are eligible for loan amount so that they can specifically target these customers.

Variable	Description
Loan_ID	Unique Loan ID
Gender	Male/ Female
Married	Applicant married (Y/N)
Dependents	Number of dependents
Education	Applicant Education (Graduate/ Under Graduate)
Self_Employed	Self employed (Y/N)
ApplicantIncome	Applicant income
CoapplicantIncome	Coapplicant income
LoanAmount	Loan amount in thousands
Loan_Amount_Term	Term of loan in months
Credit_History	credit history meets guidelines
Property_Area	Urban/ Semi Urban/ Rural
Loan_Status	Loan approved (Y/N)

In brief, it is a classification problem where we have to predict whether a loan would be approved or not.

Data Description

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome
0	LP001002	Male	No	0	Graduate	No	5849
1	LP001003	Male	Yes	1	Graduate	No	4583
2	LP001005	Male	Yes	0	Graduate	Yes	3000
3	LP001006	Male	Yes	0	Not Graduate	No	2583
4	LP001008	Male	No	0	Graduate	No	6000

CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status
0.0	NaN	360.0	1.0	Urban	Y
1508.0	128.0	360.0	1.0	Rural	N
0.0	66.0	360.0	1.0	Urban	Y
2358.0	120.0	360.0	1.0	Urban	Y
0.0	141.0	360.0	1.0	Urban	Y

The last categorical data, ‘Loan_Status’, whose values are either Y(Yes) or N(No) is the target column of this project.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Loan_ID            614 non-null    object
 1   Gender             601 non-null    object
 2   Married            611 non-null    object
 3   Dependents         599 non-null    object
 4   Education          614 non-null    object
 5   Self_Employed      582 non-null    object
 6   ApplicantIncome    614 non-null    int64
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object
 12  Loan_Status        614 non-null    object
dtypes: float64(4), int64(1), object(8)
```

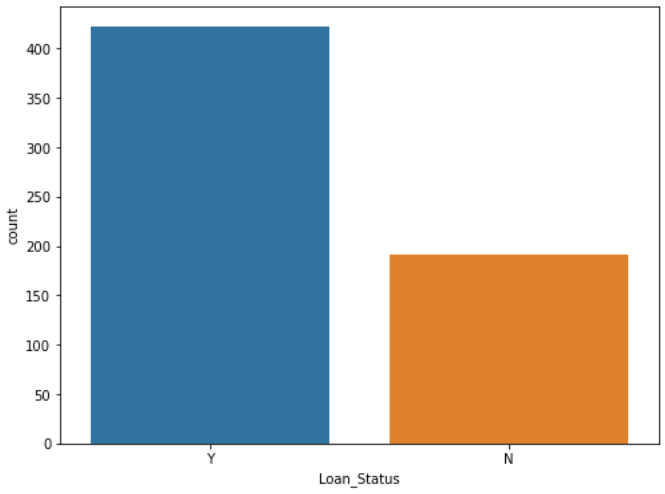
From the DataFrame.info, we can see that data contains some categorical and numerical data and quite many of them have missing values, which need to be handled.

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
count	614.000000	614.000000	592.000000	600.00000	564.000000
mean	5403.459283	1621.245798	146.412162	342.00000	0.842199
std	6109.041673	2926.248369	85.587325	65.12041	0.364878
min	150.000000	0.000000	9.000000	12.00000	0.000000
25%	2877.500000	0.000000	100.000000	360.00000	1.000000
50%	3812.500000	1188.500000	128.000000	360.00000	1.000000
75%	5795.000000	2297.250000	168.000000	360.00000	1.000000
max	81000.000000	41667.000000	700.000000	480.00000	1.000000

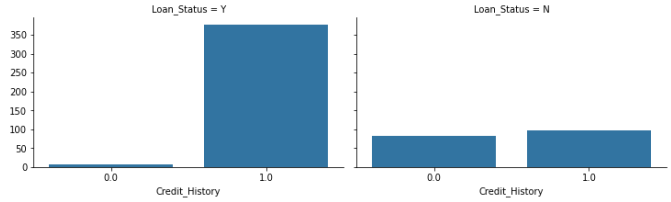
We can drop the Loan_ID (identification number of each customer) as it is not required for classification. To reduce the noise and increase the accuracy, it is important for us to eliminate duplicate or irrelevant observations from our dataset. This process is called “Data Cleaning”.

Univariate Analysis

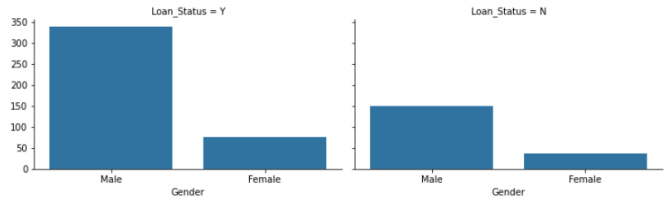
Let us explore each variable in a dataset, separately. This process might give us some insights whether or not to keep the feature for training later.



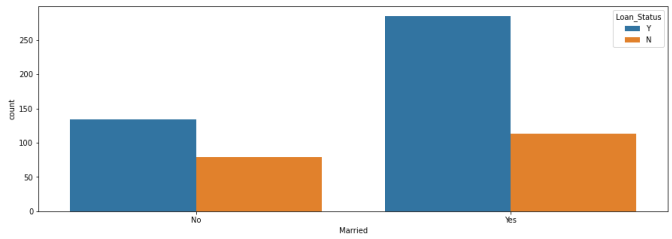
We can see that the case of loan approved is more than the case of loan rejected. Our final prediction will be based on this target data divided into a training set.



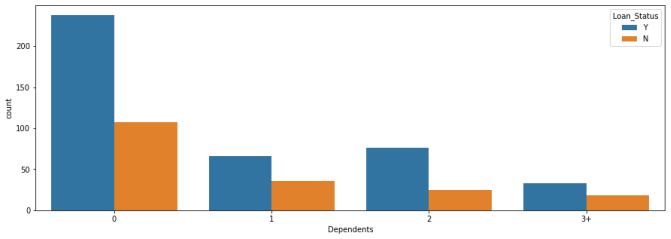
Credit history is a record of a borrower's responsible repayment of debts, from a number of sources, including banks, credit card companies, collection agencies, and governments. From the 'Credit_History', we can see most of applicants with Credit_History = 0 (Meaning, the applicant's credit history does not meet guidelines) could not get a loan. On the other hand, most of applicants with Credit_History = 1 got their loan request approved. Therefore, it should be reflected in the prediction that the applicant with Credit_History = 1 have a better chance to get a loan. Thus, this is an important feature we would like to keep.



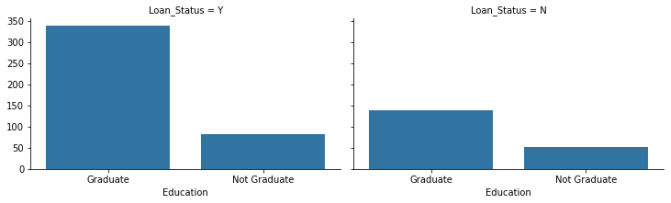
We can see that there is no clear pattern for the gender. The proportion of approved and rejected for both genders are pretty similar. Traditionally, female entrepreneurs report either difficulties or higher costs in accessing bank credit. However, the problem is not evident in this dataset and also no longer prevail in real world as it is considered a gender discrimination.



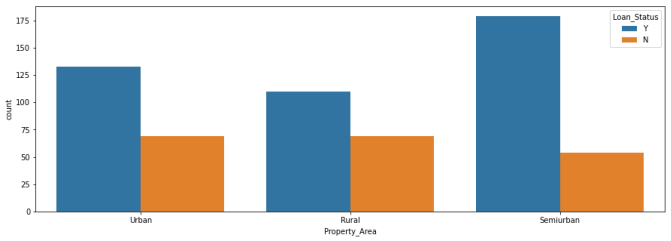
Here, we can see that many of applicants who got married did get a loan. It is because married couples have flexibility when it is time to apply for a mortgage. If spouses apply for a loan together, they can use both of their incomes, which might allow lenders to be able to approve them for even a larger loan. Therefore, we can say that marriage gives an application a better chance to get a loan. (Only if they are in good relationship.) Thus, this feature can be considered important for prediction and we keep it.



Having dependents means that the applicant has higher commitments, which in turn lower his/her disposable income. Lenders will take this into consideration when approving a loan since it affects his/her living average cost of living. And here in the dataset, we can clearly see from the plot above that people with 0 dependents have a very high chance to get a loan, compared to those who have more than one dependent.



The proportion of approved and rejected for two cases: Graduate and Not Graduate are quite similar in this dataset. It shows lower percentage of approvement for applicants who did not graduated, compared to those who graduated, but the difference is not significant.



Here, we can see that the applicants who live in semi-urban property area got more approvement of their loan request. In general, the further the applicant live away from the metropolitan city, the lower chance of getting their loan approved as following: Urban > Semi-Urban > Rural Areas. However, in this dataset, Semi-Urban turned out to have a highest approval rate. Because it shows somewhat significant distinction between three different geographical area, we can consider it a good feature.

Loan_Status	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
N	3833.5	268.0	129.0	360.0
Y	3812.5	1239.5	126.0	360.0

Lastly, let us take a look at the median (Chose Median over the Mean here to avoid the influence of outliers) of numerical data. We can see that co-applicant income has clear distinction between two Loan_Status while the other being almost identical. A co-applicant is an additional person considered in the underwriting and approval of a

loan application. Applying for a loan with a co-applicant can help to improve the chances and amount of loan approval since it involves additional sources of income, credit, or assets. Thus, we can consider this CoapplicantIncome an important feature for the prediction.

Data Processing

Handling missing values

As we checked earlier, this dataset contains quite a lot of missing values as following:

```
df.isnull().sum().sort_values(ascending=False)
```

```
Credit_History       50
Self_Employed        32
LoanAmount           22
Dependents           15
Loan_Amount_Term     14
Gender               13
Married               3
Loan_Status           0
Property_Area         0
CoapplicantIncome     0
ApplicantIncome       0
Education             0
dtype: int64
```

We need to fill them up one way or another for better prediction and, as a first step, we will separate the dataset into numerical data and categorical data.

	Gender	Married	Dependents	Education	Self_Employed	Credit_History	Property_Area	Loan_Status
0	Male	No	0	Graduate	No	1	Urban	Y
1	Male	Yes	1	Graduate	No	1	Rural	N
2	Male	Yes	0	Graduate	Yes	1	Urban	Y
3	Male	Yes	0	Not Graduate	No	1	Urban	Y
4	Male	No	0	Graduate	No	1	Urban	Y

This is a collection of categorical data.

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
0	5849.0	0.0	NaN	360.0
1	4583.0	1508.0	128.0	360.0
2	3000.0	0.0	66.0	360.0
3	2583.0	2358.0	120.0	360.0
4	6000.0	0.0	141.0	360.0

And this is a collection of numerical data.

There are many ways to fill in missing values:
- Mean Imputation
- Linear Imputation
- Regression Imputation

But here, we will simply use the Median (Most frequent value) for the missing values in categorical data and use the previous value in the same column to fill in missing values in numerical data.

Converting categorical to numerical variable

Regression analysis requires numerical variables. So, in order to include a categorical variable in a regression model, supplementary steps are required to make the results interpretable. We are going to use 'LabelEncoder' to encode labels with value between 0 and n_classes-1.

```
>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1]...)
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']
```

Figure. Example of the LabelEncoder (Scikit-learn)

	Gender	Married	Dependents	Education	Self_Employed	Credit_History	Property_Area	Loan_Status
0	Male	No	0	Graduate	No	1.0	Urban	Y
1	Male	Yes	1	Graduate	No	1.0	Rural	N
2	Male	Yes	0	Graduate	Yes	1.0	Urban	Y
3	Male	Yes	0	Not Graduate	No	1.0	Urban	Y
4	Male	No	0	Graduate	No	1.0	Urban	Y

We transform all our categorical data into constant values as below:

	Gender	Married	Dependents	Education	Self_Employed	Credit_History	Property_Area
0	1	0	0	0	0	1	2
1	1	1	1	0	0	1	0
2	1	1	0	0	1	1	2
3	1	1	0	1	0	1	2
4	1	0	0	0	0	1	2

Lastly, we concatenate categorical data and numerical data, and we are finally ready to train the data.

Data Processing

Sampling

Sampling is the selection of s subset of individuals from with a statistical population to estimate characteristics of the whole population.

Splitting our data sets into training sets and test set can be largely done under the two sampling techniques:

1. Random Sampling:
   This is a sampling technique in which a subset of a statistical population is taken out, where each member of the population has an equal opportunity of being selected. It is ideal when there is not much information about a population or when the data is diverse and not easily grouped.

2. Stratified Sampling:
   This is a sampling technique that is best used when a statistical population can easily be

broken down into distinctive sub-groups. Then samples are taken from each sub-group based on the ratio of the sub-group's size to the total population. Using Stratified Sampling technique ensures that there will be selection from each sub-group and prevents the chance of omitting one sub-group leading to sampling bias.

For this project, we are going to use 'sklearn.model_selection.StratifiedShuffleSplit', which returns stratified randomized folds. The folds are made by preserving the percentage of samples for each class.

```
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

for train, test in sss.split(X, y):
    X_train, X_test = X.iloc[train], X.iloc[test]
    y_train, y_test = y.iloc[train], y.iloc[test]
```

Figure 1. Code for StratifiedShuffleSplit

Then, we get:

```
X_train shape (491, 11)
y_train shape (491,)
X_test shape (123, 11)
y_test shape (123,)

ratio of target in y_train : [0.68635438 0.31364562]
ratio of target in y_test : [0.69105691 0.30894309]
ratio of target in original_data : [0.68729642 0.31270358]
```

where all separate target data (y_train, y_test) has the same proportion of target data as in original data.

Training the data

Machine Learning Classification Models

In this project, we are going to use four different ML-based classification models for training and compare their performances.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

models = {
    'LogisticRegression': LogisticRegression(random_state=42),
    'KNeighborsClassifier': KNeighborsClassifier(),
    'SVC': SVC(random_state=42),
    'DecisionTreeClassifier': DecisionTreeClassifier(max_depth=1, random_state=42)
}
```

1. Logistic Regression:
   Even though, the name 'Regression' comes up, it is not a regression model, but a classification model. It uses a logistic function to frame binary output model. The output of the logistic regression will be a probability ($0 \leq x \leq 1$) and can be used to predict the binary 0 or 1 as the output (if $x<0.5$, output= 0, else output=1). A commonly used model is a sigmoid function. In the sigmoid function, outputs are contained between the boundaries of 0 and 1.

   $$y = e^{(b0 + b1*x)} / (1 + e^{(b0 + b1*x)})$$

2. K-nearest Neighbor:
   A simple supervised machine learning algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions). It's easy to implement and understand but has a major drawback of becoming significantly slows as the size of that data in use grows.

3. Support Vector Classification (SVC)
   A Support Vector Machine (SVM) performs classification by finding the hyperplane that maximizes the margin between the two classes. The vectors (cases) that define the hyperplane are the support vectors.

4. Decision Tree
   It builds classification or regression models in the form of a tree structure. It breaks down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed.

Build Functions

Three different types of functions to evaluate models:

1. Performance Metrics:
   - Precision
   - Recall
   - F1
   - Log_Loss
   - Accuracy_score

2. train_eval_train function:
   To evaluate models in the same data that we train the model on.

3. train_eval_cross function:
   To evaluate models using different data that we train the model on (e.g., StratifiedKFold)

These functions are needed to improve the performance our models over the process of modification.

```
from sklearn.metrics import precision_score, recall_score, f1_score, log_loss, accuracy_score

def loss(y_true, y_pred, retu=False):
    pre = precision_score(y_true, y_pred)
    rec = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    loss = log_loss(y_true, y_pred)
    acc = accuracy_score(y_true, y_pred)

    if retu:
        return pre, rec, f1, loss, acc
    else:
        print('  pre: %.3f%n  rec: %.3f%n  f1: %.3f%n  loss: %.3f%n  acc: %.3f' % (pre, rec, f1, loss, acc))
```

Figure 3. Code for loss function

```
def train_eval_train(models, X, y):
    for name, model in models.items():
        print(name,':')
        model.fit(X, y)
        loss(y, model.predict(X))
        print('-'*30)

train_eval_train(models, X_train, y_train)
```

Figure 4. Code for train_eval_train function

```python
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=10, random_state=42, shuffle=True)

def train_eval_cross(models, X, y, folds):
    X = pd.DataFrame(X)
    y = pd.DataFrame(y)
    idx = [' pre', ' rec', ' f1', ' loss', ' acc']
    for name, model in models.items():
        ls = []
        print(name,':')

        for train, test in folds.split(X, y):
            model.fit(X.iloc[train], y.iloc[train])
            y_pred = model.predict(X.iloc[test])
            ls.append(loss(y.iloc[test], y_pred, retu=True))
        print(pd.DataFrame(np.array(ls).mean(axis=0), index=idx)[0])
        print('-'*30)

train_eval_cross(models, X_train, y_train, skf)
```

Figure 5. Code for train_eval_cross function

train_eval_train function returns:

```
LogisticRegression :
  pre: 0.930
  rec: 0.429
  f1: 0.587
  loss: 6.542
  acc: 0.811
------------------------------
KNeighborsClassifier :
  pre: 0.667
  rec: 0.364
  f1: 0.471
  loss: 8.863
  acc: 0.743
------------------------------
SVC :
  pre: 1.000
  rec: 1.000
  f1: 1.000
  loss: 0.000
  acc: 1.000
------------------------------
DecisionTreeClassifier :
  pre: 0.929
  rec: 0.422
  f1: 0.580
  loss: 6.612
  acc: 0.809
------------------------------
```

We can see that the LogisticRegression is the best model in terms of all validation metrics. On the contrary, SVC is over-fitting by having a value of 1 for precision, recall, and f1. If precision equals to 1, It means that all positive sample are classified as positive samples and none of the positive samples are classified as negative. This often signifies overfitting, which is defined as “The production of an analysis that corresponds too closely or exactly to a particular set of data and may therefore fail to fit additional data or predict future observations reliably”.

Then, let us check train_eval_cross:

```
LogisticRegression :
 pre     0.915595
 rec     0.418333
 f1      0.559679
 loss    6.745938
 acc     0.804685
Name: 0, dtype: float64
------------------------------
KNeighborsClassifier :
 pre      0.374145
 rec      0.207083
 f1       0.255135
 loss    12.028209
 acc      0.651750
Name: 0, dtype: float64
------------------------------
SVC :
 pre      0.000000
 rec      0.000000
 f1       0.000000
 loss    10.830902
 acc      0.686413
Name: 0, dtype: float64
------------------------------
DecisionTreeClassifier :
 pre     0.928095
 rec     0.424583
 f1      0.569373
 loss    6.604903
 acc     0.808769
Name: 0, dtype: float64
------------------------------
```

Here, it turns out that Decision Tree shows slightly better performance than the Logistic Regression. And, SVC is still having a problem with memorizing the data.

In repeated cross-validation, the cross-validation procedure is repeated n times, yielding n random partitions of the original sample. The n results are again averaged (or otherwise combined) to produce a single estimation. The main reason of using k-fold cross-validation is because it has a lower variance than a single hold0out set estimator, which can be very important if the amount of data available is limited.

Therefore, these values of evaluation metrics above are the mean (average) value across 10 folds. For example, the precision score above represents the overall precision for all folds. The performance for each individual fold of Logistic function is:

	pre	rec	f1	loss	acc
0	0.857143	0.375000	0.521739	7.598547	0.780000
1	1.000000	0.187500	0.315789	8.980082	0.740000
2	1.000000	0.312500	0.476190	7.598531	0.780000
3	1.000000	0.375000	0.545455	6.907755	0.800000
4	1.000000	0.466667	0.636364	5.638984	0.836735
5	1.000000	0.733333	0.846154	2.819492	0.918367
6	0.857143	0.400000	0.545455	7.048746	0.795918
7	0.900000	0.600000	0.720000	5.036922	0.854167
8	0.666667	0.266667	0.380952	9.354285	0.729167
9	0.875000	0.466667	0.608696	6.476037	0.812500

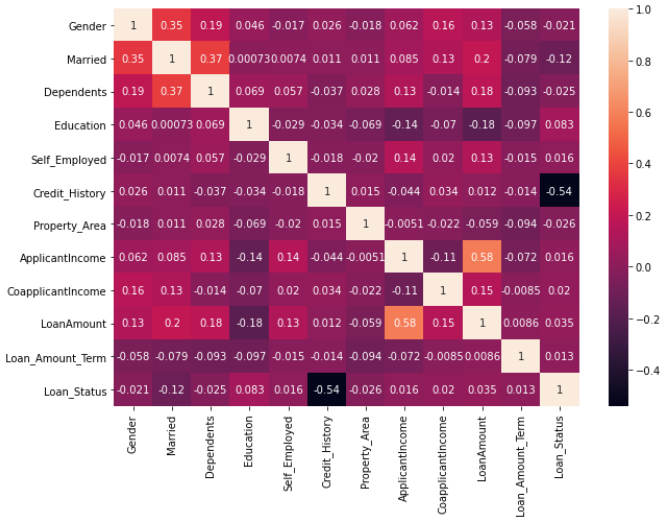
Figure 6. Logistic function's performance for each fold

Improving the performance

Feature Engineering

Feature engineering is the process of using domain knowledge to extract features from raw data via data mining techniques. These features can be used to improve the performance of machine learning algorithms.

We are going to use the heatmap to further investigate the relationship and correlation of our features:



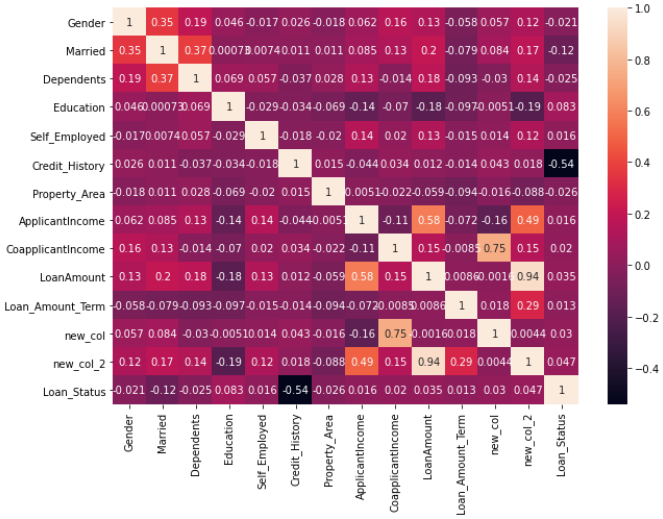
In the heatmap, correlation is ranged from -1 to +1. Values closer to zero means there is no linear trend between the two variables. The close to 1, the correlation is the more positively correlated they are; that is as one increases so does the other and the closer to 1 the stronger this relationship is. A correlation closer to -1 is similar, but instead of both increasing one variable will decrease as the other increases.

According to our heatmap, as we anticipated in the early stage, Credit_History and Married are good features for prediction of Loan_Status. We can see that Credit_History has strong negative correlation with Loan_Status.

Noted that, LoanAmount and ApplicantIncome has high similarity of 58%. (Can be considered a duplicate)

We can distinguish these two data by manipulating them. One way is the arithmetic manipulation with other related columns.

For the Coapplicantincome, we are going to divide it by Applicantincome and create a new column called 'new_col'. And for the LoanAmount, we are going to multiply it by Loan_Amount_Term and create a new column called 'new_col_2'.



We can see that the correlation between new_col (previously, Coapplicantincome) and new_col2 (previously, LoanAmount) are no longer in high similarity.

So, we can drop the previous columns: Coapplicantincome, Applicantincome, LoanAmount, and Loan_Amount_Term. Now we we evaluate the models again with this modified data, then we get:

```
LogisticRegression :
 pre       0.000000
 rec       0.000000
 f1        0.000000
 loss     10.830902
 acc       0.686413
Name: 0, dtype: float64
------------------------------
KNeighborsClassifier :
 pre       0.319301
 rec       0.181667
 f1        0.223081
 loss     13.366435
 acc       0.613005
Name: 0, dtype: float64
------------------------------
```

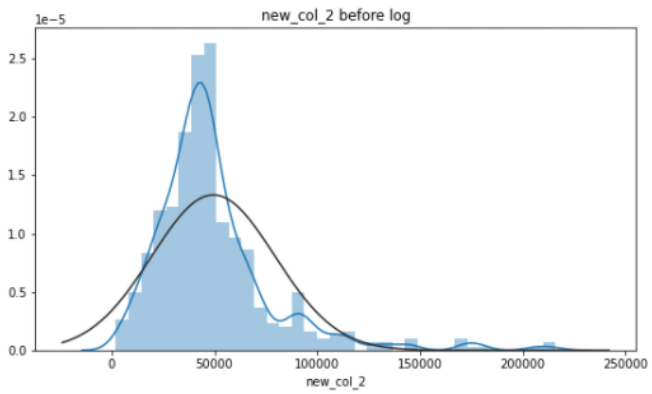
```
SVC :
 pre       0.373571
 rec       0.090417
 f1        0.139719
 loss     11.756151
 acc       0.659626
Name: 0, dtype: float64
------------------------------
DecisionTreeClassifier :
 pre       0.928095
 rec       0.424583
 f1        0.569373
 loss      6.604903
 acc       0.808769
Name: 0, dtype: float64
------------------------------
```

Unfortunately, Logistic Regression is now suffering from the overfitting, but we can see that the SVC has overcome the overfitting.

As a next step for improving our model, let us take a look at the skewness each feature:

```
------------------------------------------------
43200.0    18
39600.0    13
36000.0    11
57600.0    11
46080.0     9
           ..
3000.0      1
12000.0     1
9072.0      1
25920.0     1
7560.0      1
Name: new_col_2, Length: 211, dtype: int64
------------------------------------------------
```

And we can see that new_col_2 is highly skewed as:



We can solve this right-skewness by taking the logarithm of all the values in new_col_2 column to make the data conform to normality. This process is called a Log normalization.

```python
from scipy.stats import norm

fig, ax = plt.subplots(1,2,figsize=(20,5))

sns.distplot(X_train['new_col_2'], ax=ax[0], fit=norm)
ax[0].set_title('new_col_2 before log')

X_train['new_col_2'] = np.log(X_train['new_col_2'])  # logarithm of all the values

sns.distplot(X_train['new_col_2'], ax=ax[1], fit=norm)
ax[1].set_title('new_col_2 after log');
```

Figure 7. Code for Log normalization

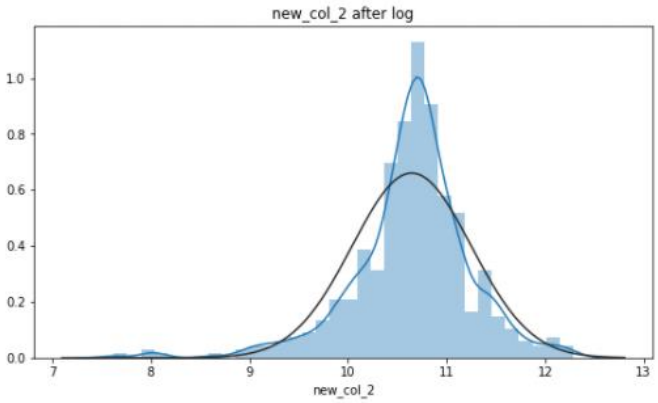


Figure 7. new_col_2's distribution after normalization

As a result, we get more normally distributed data. Normal distribution, also known as the Gaussian distribution, is a probability distribution that is symmetric about the mean, showing that data near the mean are more frequent in occurrence than data far from the mean.

A large portion of the filed of statistics is concerned with methods that assume a Normal distribution, and therefore, having our data's distribution normally distributed can help improving the performance of our models.

Again, the train_eval_cross function returns:

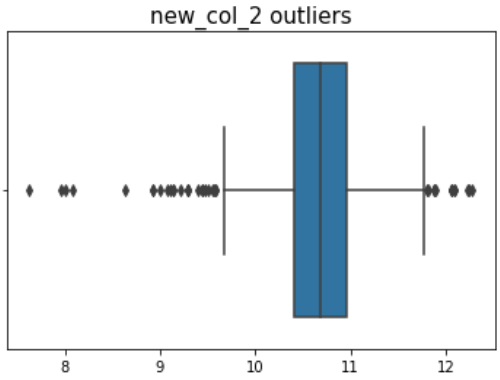
```
LogisticRegression :
 pre       0.918571
 rec       0.424583
 f1        0.567641
 loss      6.676861
 acc       0.806685
Name: 0, dtype: float64
------------------------------
KNeighborsClassifier :
 pre       0.745595
 rec       0.345833
 f1        0.465661
 loss      8.507145
 acc       0.753694
Name: 0, dtype: float64
------------------------------
```

```
SVC :
 pre       0.932857
 rec       0.424583
 f1        0.570013
 loss      6.607781
 acc       0.808685
Name: 0, dtype: float64
------------------------------
DecisionTreeClassifier :
 pre       0.928095
 rec       0.424583
 f1        0.569373
 loss      6.604903
 acc       0.808769
Name: 0, dtype: float64
------------------------------
```

We can see that all of our models have improved significantly after fixing the skewness in new_col_2 variable.

Identifying and removing outliers

In this project, we are going to use simple Interquartile Range (IQR) to identify any outliers in our dataset. Using the boxplot, we can get:



We can see that there are quite a lot of outliers in the new_col_2 variable. By removing these outliers, we should be able to further improve the performance of our models.

To detect the outliers using this method, we define a new range, let us call it decision range, and any data point lying outside this range is considered as outlier and is accordingly dealt with. A commonly used rule says that a data point is an outlier if it is more than 1.5 * IQR above the third quartile or below the first quartile. Said differently, low outliers are below Q1 – 1.5 IQR and high outliers are above Q3 + 1.5 IQR.

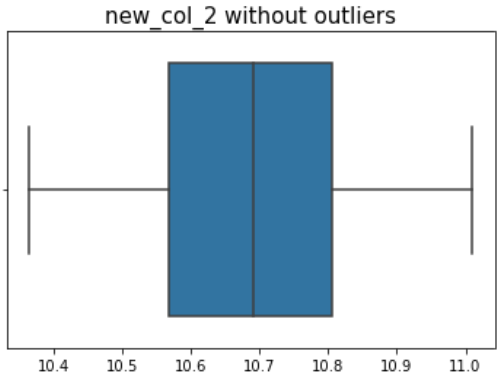
Here, the number 1.5 controls the sensitivity of the range and hence the decision rule. A bigger scale would make the outlier(s) to be considered as data point(s) while a smaller one would make some of the data point(s) to be perceived as outlier(s). And we are quite sure, none of these cases is desirable.

However, surprisingly, for this new_col_2, it turns out that 0.1 gave the best result. Although, it is considered very low which causes the loss of important information from the data by considering data points as outliers.

```
new_col_2_out = X_train['new_col_2']
q25, q75 = np.percentile(new_col_2_out, 25), np.percentile(new_col_2_out, 75) # Q25, Q75
print('Quartile 25: {} , Quartile 75: {}'.format(q25, q75))

iqr = q75 - q25
print('iqr: {}'.format(iqr))

cut = iqr * threshold
lower, upper = q25 - cut, q75 + cut
print('Cut Off: {}'.format(cut))
print('Lower: {}'.format(lower))
print('Upper: {}'.format(upper))

outliers = [x for x in new_col_2_out if x < lower or x > upper]

data_outliers = pd.concat([X_train, y_train], axis=1)
data_outliers = data_outliers.drop(data_outliers[(data_outliers['new_col_2'] > upper)
                                                 | (data_outliers['new_col_2'] < lower)].index)
```

Figure 8. Code for detecting and removing outliers.

If we check the boxplot again, then we get:



We can see that we successfully removed all outliers in the new_col_2 column. Now, we test our models again:

```
LogisticRegression :
 pre       0.941667
 rec       0.535714
 f1        0.671587
 loss      4.543108
 acc       0.868464
Name: 0, dtype: float64
------------------------------
KNeighborsClassifier :
 pre       0.806667
 rec       0.407143
 f1        0.519561
 loss      6.470769
 acc       0.812653
Name: 0, dtype: float64
------------------------------
```

```
SVC :
 pre      0.941667
 rec      0.535714
 f1       0.671587
 loss     4.543108
 acc      0.868464
Name: 0, dtype: float64
------------------------------
DecisionTreeClassifier :
 pre      0.941667
 rec      0.535714
 f1       0.671587
 loss     4.543108
 acc      0.868464
Name: 0, dtype: float64
------------------------------
```

Finally, we got desirable performances for all models. Logistic Regression, SVC, and Decision Tree achieved over 94 for precision and 53 for recall.

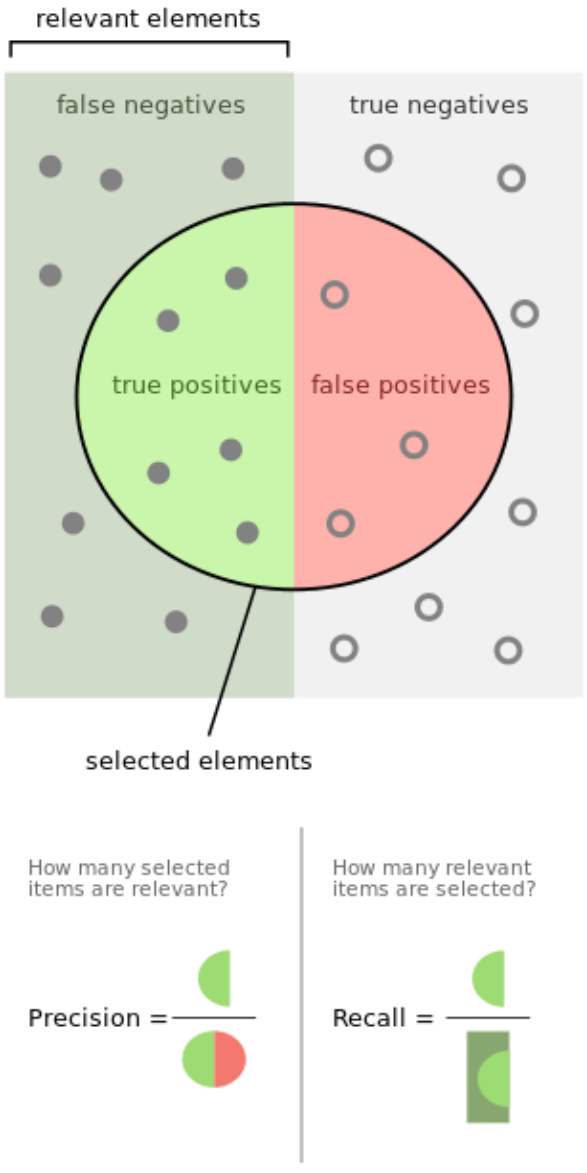
As a last stage, we can finally evaluate our trained models on the Test data. And we got:

```
LogisticRegression:
  pre: 0.895
  rec: 0.447
  f1: 0.596
  loss: 6.458
  acc: 0.813
------------------------------
KNeighborsClassifier:
  pre: 0.647
  rec: 0.289
  f1: 0.400
  loss: 9.267
  acc: 0.732
------------------------------
SVC:
  pre: 0.895
  rec: 0.447
  f1: 0.596
  loss: 6.458
  acc: 0.813
------------------------------
DecisionTreeClassifier:
  pre: 0.895
  rec: 0.447
  f1: 0.596
  loss: 6.458
  acc: 0.813
------------------------------
```

The prediction result on the test data turns out to be satisfactory with almost 90 score for precision and 44 score for recall.

Extra information about the performance metrics:

Precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances, while recall (also known as sensitivity) is the fraction of relevant instances that were retrieved. Both precision and recall are therefore based on relevance.



When we have imbalanced class and we need high true positives, precision is preferred over recall. because precision has no false negative in its formula, which can impact.