

▼ Introduction

Before we begin with any data analysis, let us set up the business context first.

Company X offers home loans. The customer submits a loan application, and Company X wants to determine if the borrower will default on the loan. If Company X determines the borrower is likely to default on the loan, then Company X will deny the borrower's loan application.

```
from IPython.core.pylabtools import figsize

from collections import Counter
import math

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import entropy

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import OneHotEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import recall_score, make_scorer, confusion_matrix, classification_report, fbeta_score

import shap
shap.initjs() # Initialize SHAP js visualizations

# Define seed for repeatability
SEED = 42
np.random.seed(SEED)
```



```
# Read in the data
file_path = '../input/freddie-mac-singlefamily-loanlevel-dataset/loan_level_500k.csv'

data = pd.read_csv(file_path)
data.head()
```

	CREDIT_SCORE	FIRST_PAYMENT_DATE	FIRST_TIME_HOMEBUYER_FLAG	MATURITY_DATE	METROPOLITAN_STATISTICAL_AREA
0	669.0	200206	N	202901	
1	732.0	199904	N	202903	1
2	679.0	200208	N	202902	1
3	721.0	200209	N	202902	3
4	618.0	200210	N	202902	1

5 rows × 27 columns

```
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500137 entries, 0 to 500136
Data columns (total 27 columns):
 #   Column                                Non-Null Count  Dtype
---  -
 0   CREDIT_SCORE                         497426 non-null float64
 1   FIRST_PAYMENT_DATE                   500137 non-null int64
 2   FIRST_TIME_HOMEBUYER_FLAG           369578 non-null object
 3   MATURITY_DATE                       500137 non-null int64
 4   METROPOLITAN_STATISTICAL_AREA       429988 non-null float64
 5   MORTGAGE_INSURANCE_PERCENTAGE       449089 non-null float64
 6   NUMBER_OF_UNITS                     500134 non-null float64
 7   OCCUPANCY_STATUS                    500137 non-null object
 8   ORIGINAL_COMBINED_LOAN_TO_VALUE     500124 non-null float64
 9   ORIGINAL_DEBT_TO_INCOME_RATIO       485208 non-null float64
10   ORIGINAL_UPB                        500137 non-null int64
11   ORIGINAL_LOAN_TO_VALUE              500128 non-null float64
12   ORIGINAL_INTEREST_RATE              500137 non-null float64
13   CHANNEL                             500137 non-null object
14   PREPAYMENT_PENALTY_MORTGAGE_FLAG    494959 non-null object
15   PRODUCT_TYPE                       500137 non-null object
16   PROPERTY_STATE                     500137 non-null object
17   PROPERTY_TYPE                      500042 non-null object
```

```

18 POSTAL_CODE          500106 non-null float64
19 LOAN_SEQUENCE_NUMBER 500137 non-null object
20 LOAN_PURPOSE          500137 non-null object
21 ORIGINAL_LOAN_TERM   500137 non-null int64
22 NUMBER_OF_BORROWERS   499890 non-null float64
23 SELLER_NAME           500137 non-null object
24 SERVICER_NAME         500137 non-null object
25 PREPAID              500137 non-null bool
26 DELINQUENT           500137 non-null bool
dtypes: bool(2), float64(10), int64(4), object(11)
memory usage: 96.3+ MB

```

## ▼ Drop irrelevant columns

The dataset used contains information that is unavailable at the time of loan application. We will drop these columns before starting with our analysis. The columns we will drop are:

- FIRST\_PAYMENT\_DATE
- MATURITY\_DATE
- MORTGAGE\_INSURANCE\_PERCENTAGE
- ORIGINAL\_UPB
- ORIGINAL\_INTEREST\_RATE
- PREPAYMENT\_PENALTY\_MORTGAGE\_FLAG

Other columns we will drop are:

- We will drop PROPERTY\_STATE as this information is encoded in the MSA column.
- LOAN\_SEQUENCE\_NUMBER is a unique id assigned to each loan. As it provides no information we will drop this column.
- SELLER\_NAME and SERVICER\_NAME are dependent loan activity and since this information is not available at the time of loan request we will drop these columns.

```

# Drop columns
drop_cols = ['FIRST_PAYMENT_DATE', 'MATURITY_DATE', 'MORTGAGE_INSURANCE_PERCENTAGE', 'ORIGINAL_UPB',
             'ORIGINAL_INTEREST_RATE', 'PREPAYMENT_PENALTY_MORTGAGE_FLAG', 'PROPERTY_STATE',
             'LOAN_SEQUENCE_NUMBER', 'SELLER_NAME', 'SERVICER_NAME']
data.drop(columns=drop_cols, inplace=True)

```

## ▼ Helper functions

```

# Analyze categorical columns
def bar_plot(df, col, fig_size=(12, 7), title=None, rot=90):

    print("Number of unique values: {}".format(len(df[col].unique())))
    print("Number of missing values: {}".format(df[col].isnull().sum()))

    df = np.round(df[col].value_counts(normalize=True, ascending=False, dropna=False)*100, 2)
    if True in df.index.isnull():
        df.index = df.index.fillna("Missing Values")

    print(df)

    figsize(fig_size[0], fig_size[1])
    plt.title(title)
    ax = df.plot.bar()
    for p in ax.patches:
        ax.annotate(str(p.get_height()), (p.get_x() * 1.005, p.get_height() * 1.005))
    plt.xticks(rotation=rot)
    plt.show()

# Analyze numerical columns
def exp_num_cols(df, col, title="", fig_size=(12, 7), x_label=""):

    print("Summary statistics: \n")
    print(df[col].describe())
    print("\n\nNumber of missing values: {}".format(df[col].isnull().sum()))

    fig, ax = plt.subplots()
    plt.title(title)
    df.loc[df.DELINQUENT == True, col].plot.hist(bins=50, density=True,
                                                  ax=ax, alpha=0.4, label="Default")
    df.loc[df.DELINQUENT == False, col].plot.hist(bins=50, density=True,
                                                  ax=ax, alpha=0.4, label="No Default")
    plt.xlabel(x_label)

```

```

plt.ylabel("Density Values")
plt.legend()
plt.show()

# Theil's U
"""Source: https://github.com/shakedzy/dython/tree/eed757172fd58209bec353911b7301b9def7db32

"""
def _conditional_entropy(x,y):
    # entropy of x given y
    y_counter = Counter(y)
    xy_counter = Counter(list(zip(x,y)))
    total_occurrences = sum(y_counter.values())
    entropy = 0
    for xy in xy_counter.keys():
        p_xy = xy_counter[xy] / total_occurrences
        p_y = y_counter[xy[1]] / total_occurrences
        entropy += p_xy * math.log(p_y/p_xy)

    return entropy

def theil_u(x,y):
    s_xy = _conditional_entropy(x,y)
    x_counter = Counter(x)
    total_occurrences = sum(x_counter.values())
    p_x = list(map(lambda n: n/total_occurrences, x_counter.values()))
    s_x = entropy(p_x)
    if s_x == 0:
        return 1
    else:
        return (s_x - s_xy) / s_x

```

## ▼ Data Exploration

### ▼ Prepare the Delinquent label

For each loan we know if the loan has been prepaid or is delinquent. We can leverage the information available in these two columns to determine the delinquency label for each loan.

```

# Explore the prepaid column
data.PREPAID.value_counts(dropna=False, normalize=True)

True      0.961185
False     0.038815
Name: PREPAID, dtype: float64

# Explore the delinquent column
data.DELINQUENT.value_counts(dropna=False, normalize=True)

False     0.964028
True      0.035972
Name: DELINQUENT, dtype: float64

# Replace the true and false values of the Prepaid column with 1's and 0's
replace_dict = {True: 1,
                False: 0}
data.PREPAID.replace(replace_dict, inplace=True)
data.PREPAID.value_counts(normalize=True)

1      0.961185
0      0.038815
Name: PREPAID, dtype: float64

# Replace the true and false values of the Delinquent column with 1's and 0's
data.DELINQUENT.replace(replace_dict, inplace=True)
data.DELINQUENT.value_counts(normalize=True)

0      0.964028
1      0.035972
Name: DELINQUENT, dtype: float64

```

Let us check if there are loans where the status is prepaid and delinquent. If such values exist we will give preference to the prepaid status. Why? A loan is labeled as delinquent if a payment is late by 1 day. We will give the customer up to 30 days to make the payment. We will only regard a loan delinquent, if the payment is late by more than 30 days.

```
# We will mark these loans where the status is prepaid and delinquent as NOT delinquent
pre_del_sum = data.PREPAID + data.DELINQUENT
pre_del_sum.value_counts(dropna=False, normalize=True)
data.loc[pre_del_sum == 2, 'DELINQUENT'] = 0
data.DELINQUENT.value_counts(dropna=False, normalize=True)

0    0.978662
1    0.021338
Name: DELINQUENT, dtype: float64
```

```
# After we create the label column we can drop the PREPAID column
data.drop(columns=['PREPAID'], inplace=True)
```

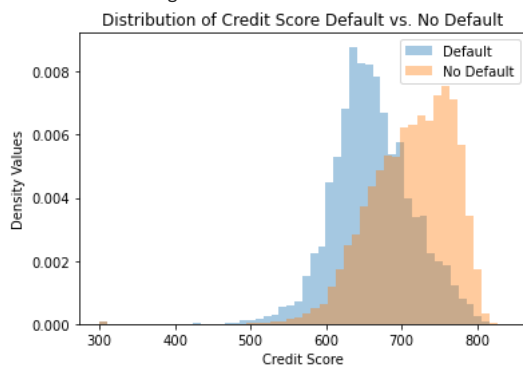
## ▼ Explore Credit Score

```
# Explore Credit Score
exp_num_cols(data, 'CREDIT_SCORE', title="Distribution of Credit Score Default vs. No Default",
              fig_size=(12, 7), x_label="Credit Score")
```

Summary statistics:

```
count    497426.000000
mean      712.536212
std        54.791262
min       300.000000
25%       676.000000
50%       719.000000
75%       756.000000
max       839.000000
Name: CREDIT_SCORE, dtype: float64
```

Number of missing values: 2711



- From the above plot we can see that as credit score increases the likelihood of defaulting on the loan decreases. This is expected. A higher credit score indicates the borrower is relatively more capable of paying off the loan.
- For credit score 500-690, the likelihood of defaulting on a loan is higher.
- For credit score 690-800, the likelihood of not defaulting on the loan is higher.

We do have some missing values. Can we drop these rows? Before we do that, let us check if we are losing any vital information.

```
# For rows with missing values, how many loans are delinquent
data.loc[data.CREDIT_SCORE.isnull(), 'DELINQUENT'].value_counts()

0    2613
1     98
Name: DELINQUENT, dtype: int64
```

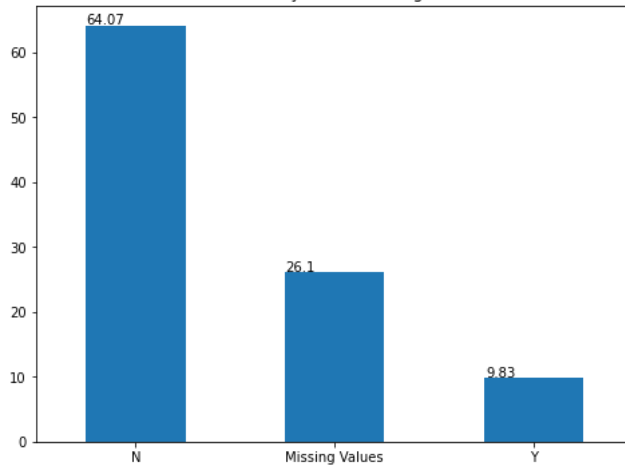
Some of the rows with missing values are loans that are delinquent. Since, there is a scarcity of this information in our dataset we cannot drop these rows. Instead, we will fill in the missing values with a missing value indicator, 0.

```
# We will replace missing values with a missing value indicator - 0
data.CREDIT_SCORE.fillna(value=0, inplace=True)
```

## ▼ Explore First Time Buyer Flag

```
bar_plot(data, col="FIRST_TIME_HOMEBUYER_FLAG", fig_size=(8, 6),
         title="First time buyer (Descending Order)", rot=0)
```

```
Number of unique values: 3
Number of missing values: 130559
N          64.07
Missing Values  26.10
Y           9.83
Name: FIRST_TIME_HOMEBUYER_FLAG, dtype: float64
First time buyer (Descending Order)
```

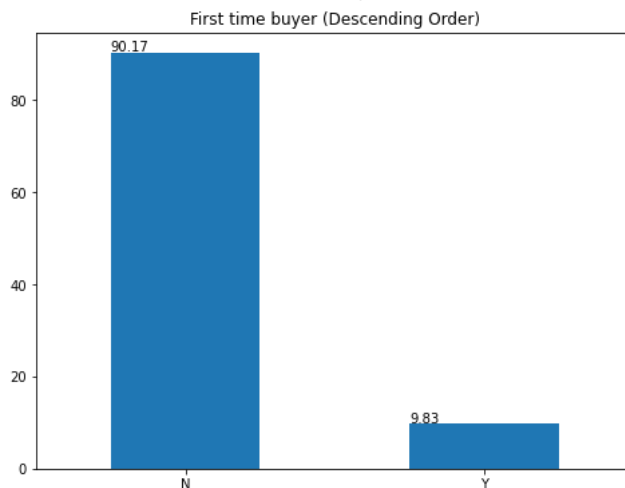


- Majority of the borrowers are not first-time home buyers.
- We also have some missing values. According to the data dictionary, if the loan transaction is for an investment property, for a second home and or refinancing then missing values are assigned. So, what is the purpose of N?
- We will fill the missing values with N because N and missing values are serving the same purpose.

```
# The missing values are for those loans where the buyer does not fall into the definition
# of a first home buyer. We will assign 'N'.
data.FIRST_TIME_HOMEBUYER_FLAG.fillna(value="N", inplace=True)
```

```
bar_plot(data, col="FIRST_TIME_HOMEBUYER_FLAG", fig_size=(8, 6),
         title="First time buyer (Descending Order)", rot=0)
```

```
Number of unique values: 2
Number of missing values: 0
N      90.17
Y       9.83
Name: FIRST_TIME_HOMEBUYER_FLAG, dtype: float64
First time buyer (Descending Order)
```



```
pd.crosstab(data.FIRST_TIME_HOMEBUYER_FLAG, data.DELINQUENT, normalize='index')
```

FIRST_TIME_HOMEBUYER_FLAG	DELINQUENT	
	0	1
N	0.978755	0.021245
Y	0.977807	0.022193

First-time home buyer is a weak predictor of loan delinquency. Whether the borrower is a first-time home buyer or not, the default rate is similar.

## ▼ Explore Metropolitan Statistical Area (MSA)

An MSA is generally with a population of 2.5 million or more. It may be divided into smaller groups of counties that the United States Office of Management and Budget refers to as Metropolitan Divisions.

```
print("Number of unique MSA codes: {}".format(len(data.METROPOLITAN_STATISTICAL_AREA.unique())))
print("Number of missing values: {}".format(data.METROPOLITAN_STATISTICAL_AREA.isnull().sum()))

Number of unique MSA codes: 391
Number of missing values: 70149
```

According to the data dictionary, a value is missing because the area is not a metropolitan area or the value is unknown. Since, we are not sure what the value could be, we will fill the missing values with `Unknown`.

```
# Fill the missing values in MSA with 'Unknown'
missing_val = 0
data.METROPOLITAN_STATISTICAL_AREA.fillna(value=missing_val, inplace=True)

cond = (data.METROPOLITAN_STATISTICAL_AREA != missing_val)
data.loc[cond, 'METROPOLITAN_STATISTICAL_AREA'] = data.loc[cond, 'METROPOLITAN_STATISTICAL_AREA'].astype(int)
```

We can use MSA to extract economic data associated to the metropolitan area. We could leverage external information such as average household income, population, average family size, current job rate, percentage of homes owned, average percentage of mortgage loans defaulted, average cost of single family homes, types of public transportation available, number of schools in the area, average rank of schools in the area, crime rate, and number of hospitals.

```
# Investigate areas with the highest loan default rates
pd.crosstab(data.METROPOLITAN_STATISTICAL_AREA, data.DELINQUENT, normalize='index')\
.sort_values(by=1, ascending=False).head()
```

	DELINQUENT	0	1
METROPOLITAN_STATISTICAL_AREA			
41900.0		0.500000	0.500000
11300.0		0.851852	0.148148
21940.0		0.875000	0.125000
11340.0		0.878676	0.121324
30020.0		0.888889	0.111111

Borrowers from MSA 41900 have a high likelihood to default on their loans. MSA 41900 is the San German-Cabo Rojo Metro Area in Puerto Rico.

## ▼ Explore Number of Units

```
# Investigate the unique values
print("Number unique values: {}".format(len(data.NUMBER_OF_UNITS.unique())))
print("Unique values: {}".format(data.NUMBER_OF_UNITS.unique()))
print("Number of missing values: {}".format(data.NUMBER_OF_UNITS.isnull().sum()))

Number unique values: 5
Unique values: [ 1.  2.  4.  3. nan]
Number of missing values: 3
```

```
# Investigate missing values
data.loc[data.NUMBER_OF_UNITS.isnull(), 'DELINQUENT'].value_counts()

0    3
Name: DELINQUENT, dtype: int64
```

Since we are losing vital information, we will drop the rows with missing values.

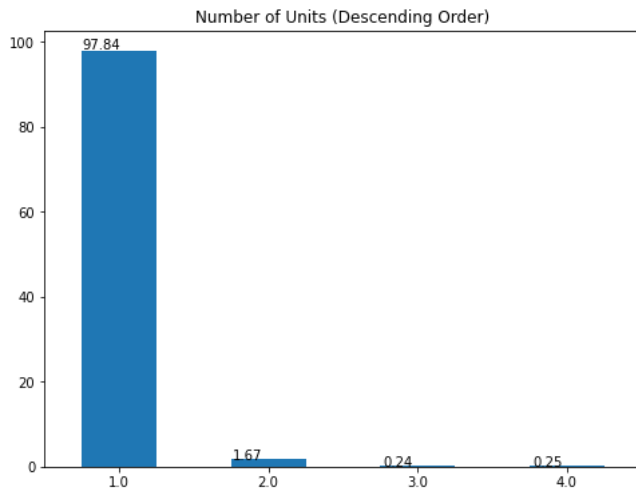
```
# Drop missing values
data = data[~data.NUMBER_OF_UNITS.isnull()].reset_index(drop=True, inplace=False)
```

```
bar_plot(data, col="NUMBER_OF_UNITS", fig_size=(8, 6),
         title="Number of Units (Descending Order)", rot=0)
```

```
Number of unique values: 4
Number of missing values: 0
```

```
1.0    97.84
2.0     1.67
4.0     0.25
3.0     0.24
```

```
Name: NUMBER_OF_UNITS, dtype: float64
```



- 97% of the loans are for houses with just 1 unit.

```
pd.crosstab(data.NUMBER_OF_UNITS, data.DELINQUENT, normalize='index').sort_values(by=1, ascending=False)
```

DELINQUENT	NUMBER_OF_UNITS	
	0	1
2.0	0.973203	0.026797
1.0	0.978747	0.021253
3.0	0.978796	0.021204
4.0	0.981511	0.018489

## ▼ Explore Occupancy Status

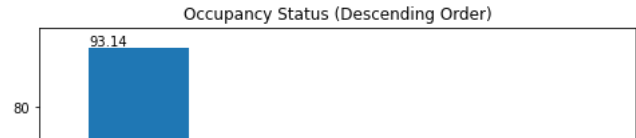
```
print("Number of unique values: {}".format(len(data.OCCUPANCY_STATUS.unique())))
print("Unique values: {}".format(data.OCCUPANCY_STATUS.unique()))
print("Number of missing values: {}".format(data.OCCUPANCY_STATUS.isnull().sum()))
```

```
Number of unique values: 3
Unique values: ['O' 'S' 'I']
Number of missing values: 0
```

- O: Owner occupied
- I: Investment property
- S: Second home

```
bar_plot(data, col="OCCUPANCY_STATUS", fig_size=(8, 6),
         title="Occupancy Status (Descending Order)", rot=0)
```

```
Number of unique values: 3
Number of missing values: 0
O    93.14
I     4.02
S     2.84
Name: OCCUPANCY_STATUS, dtype: float64
```



- 91% of the home mortgages are where the homes will be occupied by the owner.



```
pd.crosstab(data.OCCUPANCY_STATUS, data.DELINQUENT, normalize='index').sort_values(by=1, ascending=False)
```

	DELINQUENT	0	1
OCCUPANCY_STATUS			
I		0.965140	0.034860
O		0.978889	0.021111
S		0.990360	0.009640

- Loans for investment properties have the highest default rate.

▼ Explore relationship between Occupancy Status and Number of Units

```
# Distribution of occupancy statuses across number of units
pd.crosstab(data.OCCUPANCY_STATUS, data.NUMBER_OF_UNITS, normalize='index')
```

	NUMBER_OF_UNITS	1.0	2.0	3.0	4.0
OCCUPANCY_STATUS					
I		0.790989	0.143717	0.025014	0.040280
O		0.985982	0.011636	0.001451	0.000932
S		0.996552	0.003448	0.000000	0.000000

- Majority of the investment properties, owner occupied and second homes are single unit homes. This is expected, since single unit homes account for almost 98% of the data points.
- Borrowers who want to purchase their second home are looking for either single unit or double unit homes.

```
# Distribution of number of units across occupancy statuses
pd.crosstab(data.OCCUPANCY_STATUS, data.NUMBER_OF_UNITS, normalize='columns')
```

	NUMBER_OF_UNITS	1.0	2.0	3.0	4.0
OCCUPANCY_STATUS					
I		0.032504	0.345735	0.426633	0.651125
O		0.938555	0.648403	0.573367	0.348875
S		0.028940	0.005862	0.000000	0.000000

- 93% of the single unit homes are going to be occupied by the owner.
- 65% of the homes with 4 units are purchased primarily as investment properties.

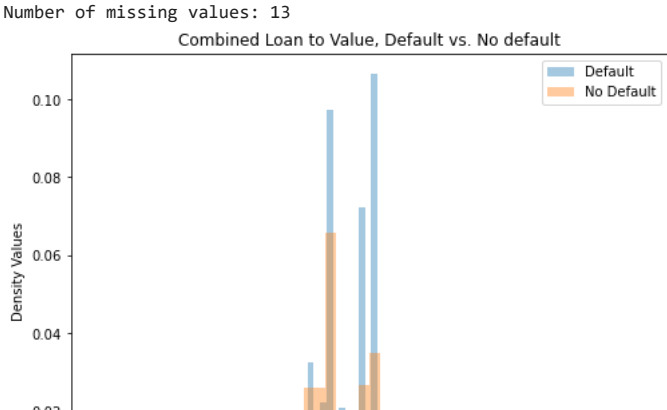
▼ Explore Combined Loan To Value

```
exp_num_cols(data, 'ORIGINAL_COMBINED_LOAN_TO_VALUE',
              title="Combined Loan to Value, Default vs. No default",
              x_label="Combined Loan to Value")
```



```
Summary statistics:

count      500121.000000
mean        76.053735
std         15.139871
min          6.000000
25%         70.000000
50%         80.000000
75%         88.000000
max        180.000000
Name: ORIGINAL_COMBINED_LOAN_TO_VALUE, dtype: float64
```



- From the above plot we can see that as the combined loan to value increases, the likelihood of defaulting on the loan increases. This is expected.

```
data.loc[data.ORIGINAL_COMBINED_LOAN_TO_VALUE > 90, 'DELINQUENT'].value_counts(normalize=True)

0    0.960002
1    0.039998
Name: DELINQUENT, dtype: float64
```

```
data.loc[data.ORIGINAL_COMBINED_LOAN_TO_VALUE <= 90, 'DELINQUENT'].value_counts(normalize=True)

0    0.981785
1    0.018215
Name: DELINQUENT, dtype: float64
```

When the combined loan to value is greater than 90% the loan default rate is twice that when the loan to value is less than 90%.

Next, we need to deal with missing values. Since we have only 12 missing rows, we will fill in the missing values with the median.

```
# Fill the missing values with the median
data.loc[data.ORIGINAL_COMBINED_LOAN_TO_VALUE.isnull(), 'ORIGINAL_COMBINED_LOAN_TO_VALUE'] = \
    data.ORIGINAL_COMBINED_LOAN_TO_VALUE.median()
```

▼ Explore the relationship between Combined Loan to Value and Number of Units

```
data.groupby(['NUMBER_OF_UNITS'])['ORIGINAL_COMBINED_LOAN_TO_VALUE'].describe()
```

	count	mean	std	min	25%	50%	75%	max
NUMBER_OF_UNITS								
1.0	489352.0	76.113550	15.155528	6.0	70.0	80.0	88.0	180.0
2.0	8359.0	74.471827	14.303496	7.0	68.0	78.0	82.0	100.0
3.0	1179.0	69.277354	13.027277	10.0	64.0	72.0	80.0	95.0
4.0	1244.0	69.617363	12.630266	7.0	65.0	71.0	80.0	95.0

The combined loan to value is more or less evenly spread through homes with units 1, 2, 3 and 4.

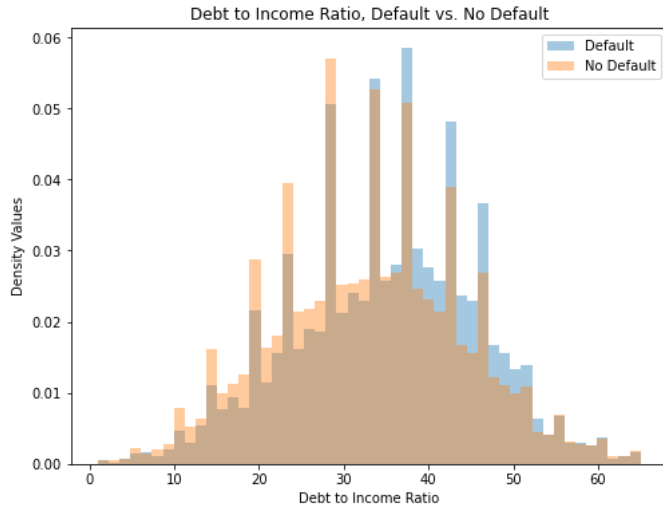
▼ Explore the Debt to Income Ratio

```
exp_num_cols(data, 'ORIGINAL_DEBT_TO_INCOME_RATIO',
              title="Debt to Income Ratio, Default vs. No Default",
              x_label='Debt to Income Ratio')
```

Summary statistics:

```
count    485206.000000
mean      32.917518
std       11.111817
min        1.000000
25%       25.000000
50%       33.000000
75%       41.000000
max       65.000000
Name: ORIGINAL_DEBT_TO_INCOME_RATIO, dtype: float64
```

Number of missing values: 14928



Higher the debt-to-income ratio, higher is the likelihood of a loan default. But for values between greater than 52% the likelihood of defaults and no defaults are similar.

We have over 14,000 missing values. From the data dictionary we learn that the missing values are where debt-to-income ratios are greater than 65%.

We will fill the missing values with an indicator value such as 999. Additionally, we will create a new column that indicates high debt-to-income ratio.

```
# Fill in missing values with '999'
dti_ind = 999
data.ORIGINAL_DEBT_TO_INCOME_RATIO.fillna(value=dti_ind, inplace=True)

# Create a separate column to indicate this information.
data['HIGH_DEBT_TO_INCOME_RATIO'] = 0
data.loc[data.ORIGINAL_DEBT_TO_INCOME_RATIO == dti_ind, 'HIGH_DEBT_TO_INCOME_RATIO'] = 1
data['HIGH_DEBT_TO_INCOME_RATIO'].value_counts(normalize=True)

0    0.970152
1    0.029848
Name: HIGH_DEBT_TO_INCOME_RATIO, dtype: float64
```

```
pd.crosstab(data.HIGH_DEBT_TO_INCOME_RATIO, data.DELINQUENT, normalize='index')
```

HIGH_DEBT_TO_INCOME_RATIO	DELINQUENT	
	0	1
0	0.978440	0.021560
1	0.985865	0.014135

I expected to see loans with a high debt-to-income ratio to have a higher than average default rate.

## ▼ Explore Loan To Value

Earlier, we had explored Combined Loan to Value. So how is that different from Loan to Value?

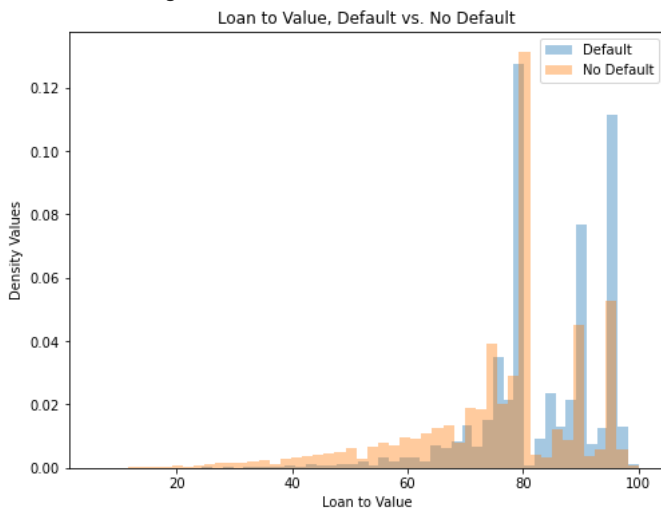
The main difference between the two lies in the numerator. Combined loan to value is the original mortgage loan amount on the note date plus any secondary mortgage loan amount disclosed by the Seller by the property's appraised value. Whereas, loan to value is the original mortgage loan amount on the note date by the property's appraised value

```
exp_num_cols(data, 'ORIGINAL_LOAN_TO_VALUE',
             title="Loan to Value, Default vs. No Default",
             x_label="Loan to Value")
```

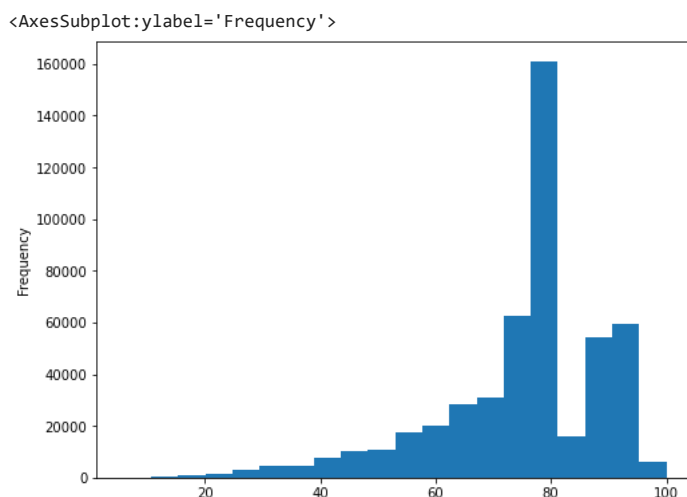
Summary statistics:

```
count    500125.000000
mean      75.710876
std       14.937603
min        6.000000
25%       70.000000
50%       80.000000
75%       85.000000
max      100.000000
Name: ORIGINAL_LOAN_TO_VALUE, dtype: float64
```

Number of missing values: 9



```
data.ORIGINAL_LOAN_TO_VALUE.plot.hist(bins=20)
```



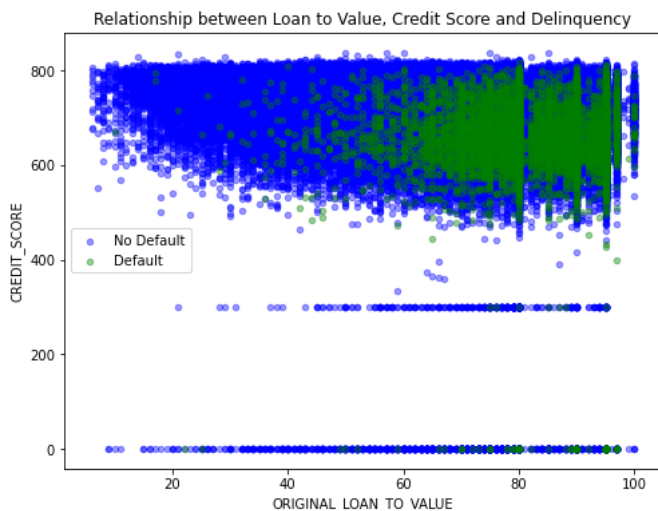
- On average, a borrower requests for a loan that is 75% of the value of the property.
- When loan to value is greater than 80%, the likelihood of a loan default increases. Borrowers who tend to take a loan the same amount as the value of the property tend to default more. This is interesting since 50% of the loans have values greater than 80%.
- The distribution is skewed to the left. Additionally, the distribution resembles a bi-modal distribution. We can see a peak at 78-80% and 90-95%.

We have 9 missing values. We will fill in the missing values with the median.

```
# We will fill the missing values with the median
data.loc[data.ORIGINAL_LOAN_TO_VALUE.isnull(), 'ORIGINAL_LOAN_TO_VALUE'] = \
    data.ORIGINAL_LOAN_TO_VALUE.median()
```

### ▼ Explore relationship between Loan to Value, Credit Score and Delinquency

```
fig, ax = plt.subplots()
plt.title("Relationship between Loan to Value, Credit Score and Delinquency")
data[data.DELINQUENT == 0].plot.scatter(x="ORIGINAL_LOAN_TO_VALUE", y="CREDIT_SCORE",
                                         c="blue", alpha=0.4, ax=ax, label="No Default")
data[data.DELINQUENT == 1].plot.scatter(x="ORIGINAL_LOAN_TO_VALUE", y="CREDIT_SCORE",
                                         c="green", alpha=0.4, ax=ax, label="Default")
plt.legend()
plt.show()
```

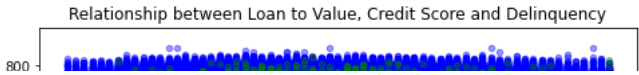


- Almost all of the loan defaults are when loan to value is greater than 50%. This makes intuitive sense. Customers who request for loans that are relatively a small percentage of the value of the property tend to be better borrowers because they have the capability (to accrue the required capital over time) to pay off the loan.
- As the loan to value decreases the variance in the credit score decreases as well. For loan to value less than 20%, the credit score ranges between 600 and 800.

### ▼ Explore relationship between Debt-to-income ratio, Credit Score and Delinquency

```
cond_default = ((data.DELINQUENT == 0) & (data.ORIGINAL_DEBT_TO_INCOME_RATIO < 200))
cond_no_default = ((data.DELINQUENT == 1) & (data.ORIGINAL_DEBT_TO_INCOME_RATIO < 200))

fig, ax = plt.subplots()
plt.title("Relationship between Loan to Value, Credit Score and Delinquency")
data[cond_default].plot.scatter(x="ORIGINAL_DEBT_TO_INCOME_RATIO", y="CREDIT_SCORE",
                               c="blue", alpha=0.4, ax=ax, label="No Default")
data[cond_no_default].plot.scatter(x="ORIGINAL_DEBT_TO_INCOME_RATIO", y="CREDIT_SCORE",
                                   c="green", alpha=0.4, ax=ax, label="Default")
plt.legend()
plt.show()
```



The loan defaults are quite spread through the debt-to-income ratio.

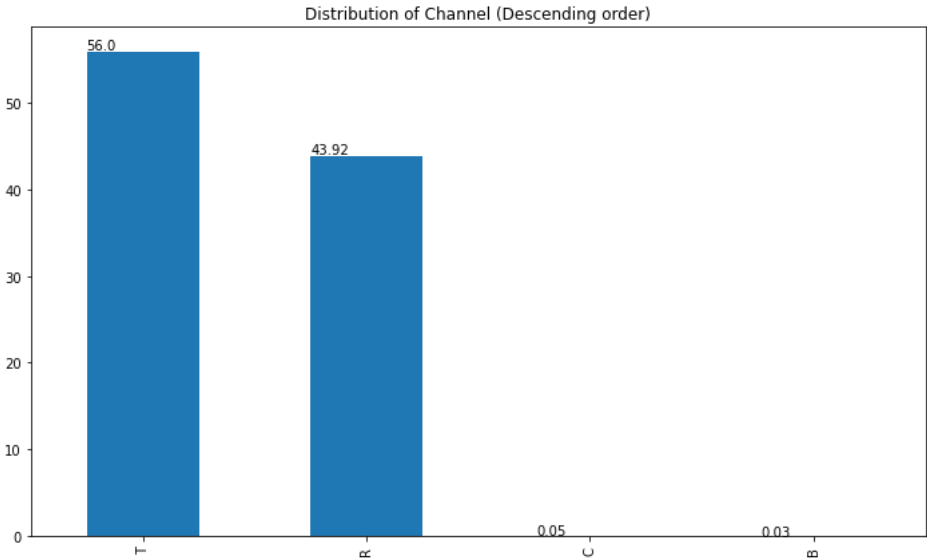


▼ Explore Channel



```
bar_plot(data, 'CHANNEL', title="Distribution of Channel (Descending order)")
```

Number of unique values: 4  
Number of missing values: 0  
T 56.00  
R 43.92  
C 0.05  
B 0.03  
Name: CHANNEL, dtype: float64



- R: Retail
- B: Broker
- C: Correspondent
- T: Third Party Organization Not Specified

Observations:

- Majority of the loans do not have the third party organization specified by the seller.
- The next popular option is for a borrower to directly go to the lender or one of its affiliates.

```
pd.crosstab(data.CHANNEL, data.DELINQUENT, normalize='index').sort_values(by=1, ascending=False)
```

DELINQUENT	0	1
CHANNEL		
B	0.914110	0.085890
T	0.975008	0.024992
C	0.977941	0.022059
R	0.983369	0.016631

- The broker channel has the highest loan default rate.
- This is interesting. If loans are requested from channels other than the lender, the default rates tend to be higher.

▼ Explore relationship between Channel and Loan to Value

```
data.groupby(['CHANNEL'])['ORIGINAL_LOAN_TO_VALUE'].describe()
```

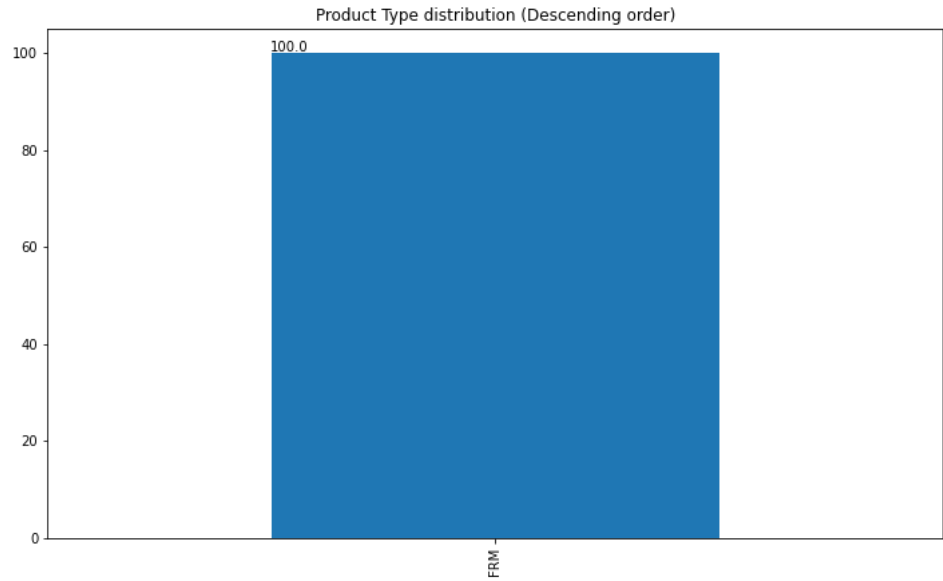
	count	mean	std	min	25%	50%	75%	max
CHANNEL								
B	163.0	79.711656	14.270199	26.0	74.0	80.0	93.0	97.0
C	272.0	71.783088	16.005557	15.0	65.0	75.0	80.0	97.0

- Brokers and correspondents tend to have a higher minimum for loan to value. Probably they want the loan to value percentage to be above a threshold that makes the business deal valuable for them.

▼ Explore Product Type

```
bar_plot(data, 'PRODUCT_TYPE',
         title="Product Type distribution (Descending order)")
```

Number of unique values: 1  
Number of missing values: 0  
FRM 100.0  
Name: PRODUCT\_TYPE, dtype: float64



Since this column has the same value for all rows, it provides no useful information to us. Let us drop this column.


```
data.drop(columns=['PRODUCT_TYPE'], inplace=True)
```

▼ Explore Property Type

```
bar_plot(data, 'PROPERTY_TYPE',
         title="Property Type distribution (Descending order)")
```

```
Number of unique values: 7
Number of missing values: 95
SF      82.10
PU      10.69
CO       6.73
MH       0.35
CP       0.08
LH       0.04
Missing Values    0.02
Name: PROPERTY_TYPE, dtype: float64
```

Property Type distribution (Descending order)



- CO: Condo
- PU: Planned Unit Development
- MH: Manufactured Housing
- SF: Single-Family
- CP: Cooperative share
- LH: Leasehold

Observations:

- Single family homes are the most popular option amongst borrowers by a large margin.

```
data.loc[data.PROPERTY_TYPE.isnull(), 'DELINQUENT'].value_counts()

0    93
1     2
Name: DELINQUENT, dtype: int64
```

We cannot remove the rows with missing values since we will be likely to lose vital information. Instead, we will replace the missing values with Not Available.

```
# Replace the missing values with 'Not Available'
data.PROPERTY_TYPE.fillna(value='Not Available', inplace=True)

pd.crosstab(data.PROPERTY_TYPE, data.DELINQUENT, normalize='index').sort_values(by=1, ascending=False)
```

DELINQUENT	0	1
PROPERTY_TYPE		
MH	0.878805	0.121195
LH	0.959391	0.040609
SF	0.976991	0.023009
Not Available	0.978947	0.021053
CO	0.987544	0.012456
PU	0.989131	0.010869
CP	0.992105	0.007895

Manufactured housing has the highest default rate. It is almost six times the average default rate.

▼ Explore the relationship between Number of Units and Property Type

```
# Distribution of number of units across property type
pd.crosstab(data.NUMBER_OF_UNITS, data.PROPERTY_TYPE, normalize='columns')
```

PROPERTY_TYPE	CO	CP	LH	MH	Not Available	PU	SF
NUMBER_OF_UNITS							
1.0	0.999554	1.0	0.923858	0.986215	0.842105	0.998335	0.974127
2.0	0.000297	0.0	0.060914	0.010339	0.136842	0.001291	0.020060
3.0	0.000059	0.0	0.010152	0.002872	0.021053	0.000056	0.002837
4.0	0.000089	0.0	0.005076	0.000574	0.000000	0.000318	0.002976

- Almost all of the condos are single unit condos.
- All cooperative share homes are single unit homes.

- Earlier, we learned that single unit homes are a very popular option. It is very evident for each property type.

## ▼ Explore Postal Code

Description of postal code from the data dictionary - ###00, where “###” represents the first three digits of the 5-digit postal code. We do not have the complete postal code.

```
print("Number of unique values: {}".format(len(data.POSTAL_CODE.unique())))
print("Number of missing values: {}".format(data.POSTAL_CODE.isnull().sum()))
```

```
Number of unique values: 893
Number of missing values: 31
```

Let us investigate the missing values.

```
data.loc[data.POSTAL_CODE.isnull(), 'DELINQUENT'].value_counts()
```

```
0    31
Name: DELINQUENT, dtype: int64
```

By removing the rows we are not be losing any vital information.

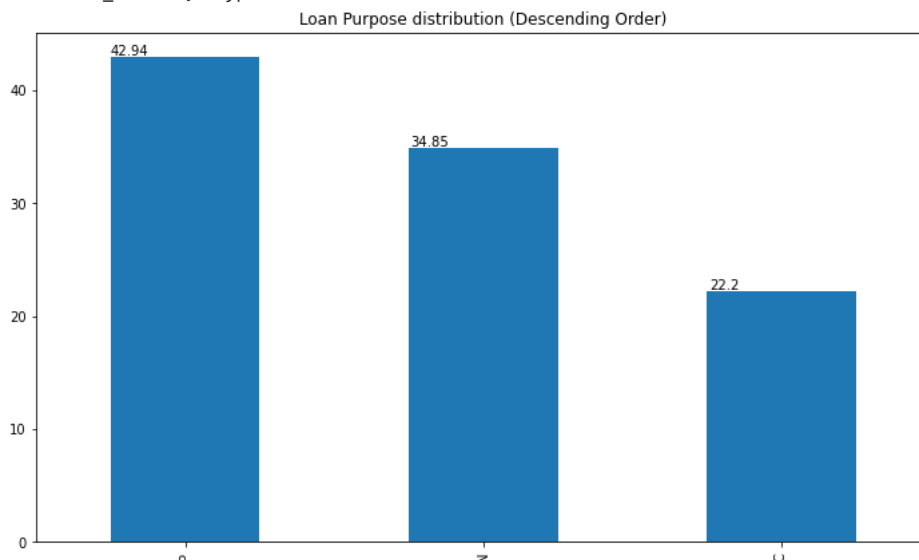
```
# We can drop the rows with missing values
data = data[~data.POSTAL_CODE.isnull()].reset_index(drop=True, inplace=False)
```

```
# Convert the type to int
data.POSTAL_CODE = data.POSTAL_CODE.astype('int')
```

## ▼ Explore Loan Purpose

```
bar_plot(data, 'LOAN_PURPOSE',
         title="Loan Purpose distribution (Descending Order)")
```

```
Number of unique values: 3
Number of missing values: 0
P    42.94
N    34.85
C    22.20
Name: LOAN_PURPOSE, dtype: float64
```



- P: Purchase
- C: Refinance - Cash Out
- N: Refinance - No Cash Out

Majority of the loans requested are for purchasing a property.

```
pd.crosstab(data.LOAN_PURPOSE, data.DELINQUENT, normalize='index').sort_values(by=1, ascending=False)
```



DELINQUENT	0	1
LOAN_PURPOSE		
C	0.976595	0.023405
N	0.976596	0.023404

The default rate is quite similar across loan purpose.

▼ Explore relationship between Loan Purpose and Occupancy Status

```
# Distribution of loan purpose across occupancy statuses
pd.crosstab(data.LOAN_PURPOSE, data.OCCUPANCY_STATUS, normalize='columns')
```

OCCUPANCY_STATUS	I	O	S
LOAN_PURPOSE			
C	0.223427	0.225843	0.095714
N	0.318926	0.353913	0.213104
P	0.457647	0.420244	0.691182

- Purchase is the popular option for all occupancy statuses.
- Refinance options are popular for investment properties and owner occupied properties.

▼ Explore relationship between Occupancy Status and Loan to Value

```
data.groupby(['OCCUPANCY_STATUS'])['ORIGINAL_LOAN_TO_VALUE'].describe()
```

	count	mean	std	min	25%	50%	75%	max
OCCUPANCY_STATUS								
I	20105.0	72.593136	13.195690	7.0	66.0	75.0	80.0	97.0
O	465789.0	75.885036	14.993814	6.0	70.0	80.0	85.0	100.0
S	14209.0	74.412837	14.855598	6.0	68.0	80.0	80.0	96.0

The loan to value is spread evenly across the various occupancy statuses.


▼ Create Refinance Indicator column

From occupancy status, we know which mortgages are refinance mortgages, let's create a new column to indicate this information.

```
data['REFINANCE_IND'] = 0
cond = (data.LOAN_PURPOSE != 'P')
data.loc[cond, 'REFINANCE_IND'] = 1

bar_plot(data, 'REFINANCE_IND',
          title="Refinance Indicator distribution")
```

```
Number of unique values: 2
Number of missing values: 0
1    57.06
0    42.94
Name: REFINANCE_IND, dtype: float64
```



```
pd.crosstab(data.REFINANCE_IND, data.DELINQUENT, normalize='index').sort_values(by=1, ascending=False)
```

DELINQUENT	0	1
REFINANCE_IND		
1	0.976596	0.023404
0	0.981403	0.018597

The default rates for refinanced and purchase loans are similar.

▼ Explore Loan Term

```
print("Number of unique values: {}".format(len(data.ORIGINAL_LOAN_TERM.unique())))
print("Number of missing values: {}".format(data.ORIGINAL_LOAN_TERM.isnull().sum()))
```

Number of unique values: 62  
Number of missing values: 0

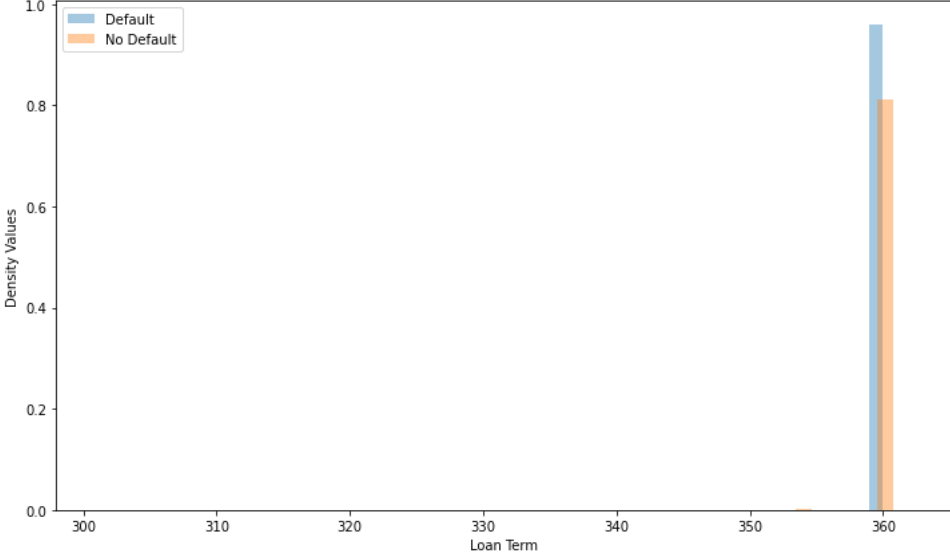
```
exp_num_cols(data, 'ORIGINAL_LOAN_TERM',
              title="Loan Term distribution, Default vs. No Default",
              x_label='Loan Term')
```

Summary statistics:

count	500103.000000
mean	359.855460
std	1.908315
min	301.000000
25%	360.000000
50%	360.000000
75%	360.000000
max	362.000000

Name: ORIGINAL\_LOAN\_TERM, dtype: float64

Number of missing values: 0



```
data.ORIGINAL_LOAN_TERM.value_counts(normalize=True, ascending=False)[:5]
```

360	0.990620
354	0.001214
348	0.000810
349	0.000564
336	0.000418

Name: ORIGINAL\_LOAN\_TERM, dtype: float64

- Almost all of the values are 360 months.
- The shortest loan duration is 301 months.

## ▼ Explore Number of Borrowers

```
print("Number of unique values: {}".format(len(data.NUMBER_OF_BORROWERS.unique())))
print("Unique values: {}".format(data.NUMBER_OF_BORROWERS.unique()))
print("Number of missing values: {}".format(data.NUMBER_OF_BORROWERS.isnull().sum()))
```

```
Number of unique values: 3
Unique values: [ 2.  1. nan]
Number of missing values: 247
```

```
# Investigate missing values
data.loc[data.NUMBER_OF_BORROWERS.isnull(), 'DELINQUENT'].value_counts()
```

```
0      245
1         2
Name: DELINQUENT, dtype: int64
```

If we decide to drop the rows we will be losing vital information on delinquent loans. We will fill in missing values with Not Available.

```
# Replace missing values with 'Not Available'
data.NUMBER_OF_BORROWERS.fillna(value='Not Available', inplace=True)
```

```
cond = (data.NUMBER_OF_BORROWERS != "Not Available")
data.loc[cond, 'NUMBER_OF_BORROWERS'] = data.loc[cond, 'NUMBER_OF_BORROWERS'].astype(int)
```

```
# Convert data type of column to 'category'
data.NUMBER_OF_BORROWERS = data.NUMBER_OF_BORROWERS.astype('str')
```

```
data.NUMBER_OF_BORROWERS.value_counts(normalize=True, ascending=False)
```

```
2          0.629988
1          0.369518
Not Available  0.000494
Name: NUMBER_OF_BORROWERS, dtype: float64
```

- Majority of the loans have two or more borrowers.

## ▼ Explore Number of Borrowers and Loan to Value

```
data.groupby(['NUMBER_OF_BORROWERS'])['ORIGINAL_LOAN_TO_VALUE'].describe()
```

	count	mean	std	min	25%	50%	75%	max
<b>NUMBER_OF_BORROWERS</b>								
1	184797.0	76.358117	15.127651	6.0	70.0	80.0	88.0	100.0
2	315059.0	75.329405	14.812089	6.0	69.0	80.0	83.0	100.0
<b>Not Available</b>	247.0	78.032389	13.980785	8.0	73.5	80.0	90.0	95.0

The loan to value is evenly spread across 1, 2, or more than 2 borrowers.

## ▼ Explore correlations between the numerical variables

```
# Get list of numerical columns
num_cols = data.select_dtypes('float64').columns
num_cols
```

```
Index(['CREDIT_SCORE', 'METROPOLITAN_STATISTICAL_AREA', 'NUMBER_OF_UNITS',
       'ORIGINAL_COMBINED_LOAN_TO_VALUE', 'ORIGINAL_DEBT_TO_INCOME_RATIO',
       'ORIGINAL_LOAN_TO_VALUE'],
      dtype='object')
```

```
# Correlation heatmap
plt.figure(figsize=(16, 6))
```

```
df_corr = data[num_cols].corr()
ax = sns.heatmap(df_corr, annot=True)
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
plt.yticks(rotation=0)
plt.show()
```



- Combined loan to value and loan to value are almost perfectly correlated. This is expected.
- As loan to value decreases there is a slight increase in credit score.
- The other variables are weakly correlated. Knowing one variable does not give much information about the other.

## ▼ Explore the correlations between the categorical features

We will use Thiel's U to quantify the relationship between the categorical features.

- Thiel's U is an uncertainty coefficient that is based on information theory.
- It essentially tells us the likelihood of x given y. This probability measure makes Thiel's U interpretable.
- Moreover, this measure is not symmetric,  $p(x|y) \neq p(y|x)$ . This gives us additional information about the relationship between x and y.

```
# Get list of categorical values
cat_cols = data.columns[~data.columns.isin(num_cols)]
cat_cols = cat_cols.drop(['POSTAL_CODE'])
cat_cols

Index(['FIRST_TIME_HOMEBUYER_FLAG', 'OCCUPANCY_STATUS', 'CHANNEL',
      'PROPERTY_TYPE', 'LOAN_PURPOSE', 'ORIGINAL_LOAN_TERM',
      'NUMBER_OF_BORROWERS', 'DELINQUENT', 'HIGH_DEBT_TO_INCOME_RATIO',
      'REFINANCE_IND'],
      dtype='object')
```

```
# Calculate Thiel's U
cat_corr = {}
for i in cat_cols:
    cat_corr[i] = []
    for j in cat_cols:
        cat_corr[i].append(np.round(theil_u(data[i], data[j]), 3))

cat_corr_df = pd.DataFrame(cat_corr, index=cat_cols)
```

```
# Thiel's U heatmap
plt.figure(figsize=(16, 6))
plt.title("Thiel's U")
ax = sns.heatmap(cat_corr_df, annot=True)
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
plt.yticks(rotation=0)
plt.show()
```



- None of the variables provide any information about delinquency. It is going to be a challenge for the model to predict delinquency given the variables.
- We can see a strong relationship between loan purpose and refinance indicator. This is expected because we created refinance indicator from loan purpose.

Great! So far, we have performed a deep-dive analysis of the dataset and found interesting patterns. Next, we will build a machine learning model to predict the likelihood a borrower will default on a loan at the time of requesting the loan.

## ▼ Predict loan default

## ▼ Encode categorical features

```
# Define the list of columns to one-hot encode
categorical_feats = ['FIRST_TIME_HOMEBUYER_FLAG', 'OCCUPANCY_STATUS', 'CHANNEL',
                    'PROPERTY_TYPE', 'LOAN_PURPOSE', 'NUMBER_OF_BORROWERS']

# Create a copy of the original dataset
feat_data = data.copy()

# One-hot encode the categorical features
enc = OneHotEncoder(sparse=False)
enc.fit(feat_data[categorical_feats])
encoded_data = enc.transform(feat_data[categorical_feats])

# Get encoded column names
enc_col_nm = pd.get_dummies(feat_data[categorical_feats],
                           prefix=categorical_feats).columns

enc_col_nm

Index(['FIRST_TIME_HOMEBUYER_FLAG_N', 'FIRST_TIME_HOMEBUYER_FLAG_Y',
      'OCCUPANCY_STATUS_I', 'OCCUPANCY_STATUS_O', 'OCCUPANCY_STATUS_S',
      'CHANNEL_B', 'CHANNEL_C', 'CHANNEL_R', 'CHANNEL_T', 'PROPERTY_TYPE_CO',
      'PROPERTY_TYPE_CP', 'PROPERTY_TYPE_LH', 'PROPERTY_TYPE_MH',
      'PROPERTY_TYPE_Not Available', 'PROPERTY_TYPE_PU', 'PROPERTY_TYPE_SF',
      'LOAN_PURPOSE_C', 'LOAN_PURPOSE_N', 'LOAN_PURPOSE_P',
      'NUMBER_OF_BORROWERS_1', 'NUMBER_OF_BORROWERS_2',
      'NUMBER_OF_BORROWERS_Not Available'],
      dtype='object')

encoded_data = pd.DataFrame(encoded_data, columns=enc_col_nm)
feat_data.drop(columns=categorical_feats, inplace=True)
feat_data = pd.concat([feat_data, encoded_data], axis=1)
```

```
feat_data.head()
```

	CREDIT_SCORE	METROPOLITAN_STATISTICAL_AREA	NUMBER_OF_UNITS	ORIGINAL_COMBINED_LOAN_TO_VALUE	ORIGINAL_DELINQUENCY
0	669.0	0.0	1.0	80.0	0
1	732.0	17140.0	1.0	25.0	0
2	679.0	15940.0	1.0	91.0	0
3	721.0	38060.0	1.0	39.0	0
4	618.0	10420.0	1.0	85.0	0

5 rows × 33 columns

```
# Split the dataset
test_size=0.2
train_data, test_data = train_test_split(feat_data, random_state=SEED, stratify=feat_data.DELINQUENCY,
                                          test_size=test_size)
```

```
# Prepare the train dataset
train_x = train_data[train_data.columns[~train_data.columns.isin(['DELINQUENCY'])]]
train_y = train_data.DELINQUENCY
```

```
# Prepare the test dataset
test_x = test_data[test_data.columns[~test_data.columns.isin(['DELINQUENCY'])]]
test_y = test_data.DELINQUENCY
```

## ▼ Model training

We will select the Decision Tree classifier as our model to predict Delinquency .

- The model is easy to interpret.
- It is scale invariant.

```
# Define the model
dc = DecisionTreeClassifier(max_depth=10, min_samples_split=10, min_samples_leaf=10,
                           random_state=SEED, class_weight="balanced")
```

## ▼ Define the scoring metric

We will use Recall to measure the performance of the model.

```
# Define a scorer
rs = make_scorer(recall_score)

# Cross validation
cv = cross_val_score(dc, train_x, train_y, cv=5, n_jobs=-1, scoring=rs)
print("%.2f recall with a standard deviation of %.2f" % (cv.mean(), cv.std()))
```

0.75 recall with a standard deviation of 0.01

```
# Fit the model
dc.fit(train_x, train_y)

DecisionTreeClassifier(class_weight='balanced', max_depth=10,
                      min_samples_leaf=10, min_samples_split=10,
                      random_state=42)
```

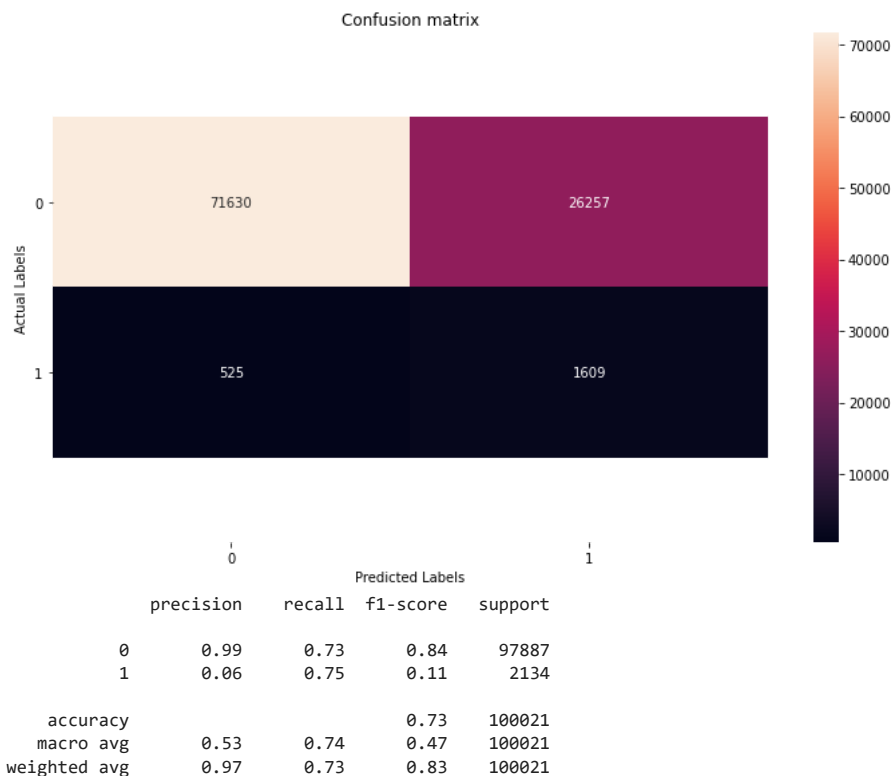
## ▼ Test the classifier

```
# Get predictions from the test dataset
pred = dc.predict(test_x)
print("The test recall score is {}".format(np.round(recall_score(test_y, pred), 2)))
```

The test recall score is 0.75

```
plt.title("Confusion matrix")
ax = sns.heatmap(confusion_matrix(test_y, pred), annot=True, fmt='d')
bottom, top = ax.get_ylim()
```

```
ax.set_ylim(bottom + 0.5, top - 0.5)
plt.xticks(rotation=0)
plt.xlabel("Predicted Labels")
plt.ylabel("Actual Labels")
plt.show()
print(classification_report(test_y, pred))
```



The test and cross-validation error are similar. This is good! There is no overfitting. We may be able to improve the model performance. We can try various methods like:

- hyperparameter tuning
- try other feature engineering techniques like binary encoding, frequency encoding, etc.
- train the model only on a subset of the features
- try other imputation techniques, or use missing value indicators
- use more complex models such as neural networks

The F-beta score is a weighted harmonic mean between precision and recall, and is used to weight precision and recall differently.

We assign `beta=2` to place more emphasis on recall. We are making the assumption that the cost associated with a false negative is significantly higher than the cost associated with a false positive. Therefore, we focus on maximizing the recall and minimizing losses.

```
# fbeta score
print("F-Beta Score: {}".format(np.round(fbeta_score(test_y, pred, beta=2, average='weighted'), 2)))

F-Beta Score: 0.76
```

## ▼ Explore the influence of variables on delinquency

Let's determine which factors influence the chances of a borrower defaulting on the loan. **We will perform this analysis using SHAP!**

What is SHAP???

SHAP (SHapley Additive exPlanations) leverages coalitional game theory concepts to help explain the feature contributions of any machine learning model. Below are resources to learn more about SHAP:

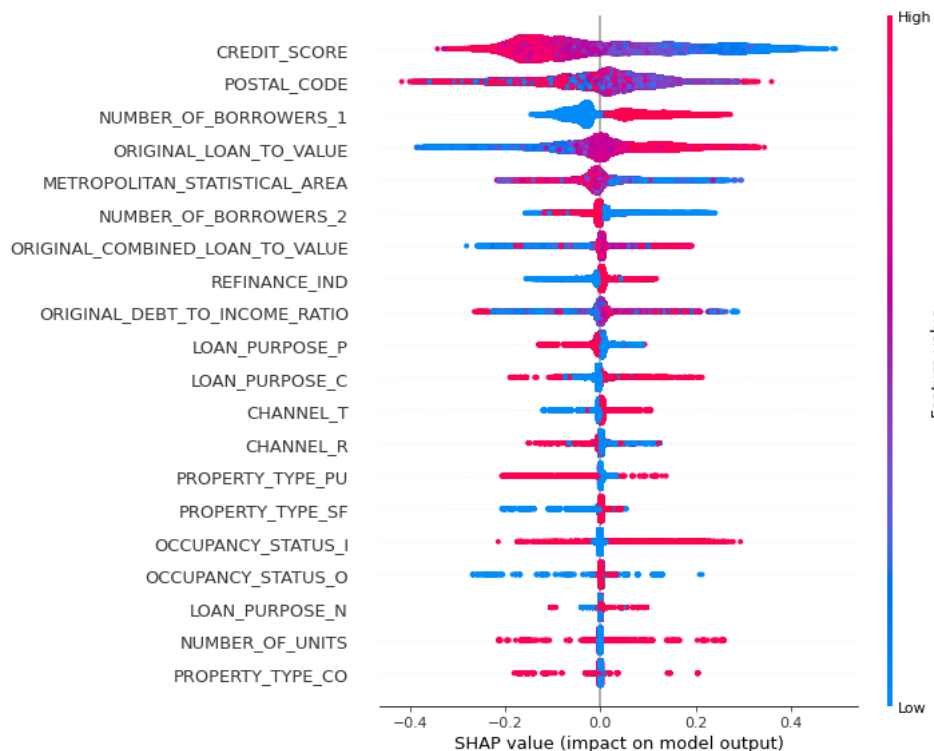
- [A Unified Approach to Interpreting Model Predictions](#)
- [SHAP - Github](#)
- [SHAP](#)
- [Shapley values](#)

Shapley values corresponding to a specific value of a feature is the contribution of that feature value towards the model prediction. We will use the shapley values to study the influence of various variables on predicting delinquency.

```
# Create the SHAP explainer instance
explainer = shap.TreeExplainer(dc, train_x)
shap_values = explainer.shap_values(train_x)

100%|=====| 797028/800164 [02:32<00:00]
```

```
# SHAP summary plot
fig, ax = plt.subplots()
fig.set_size_inches(18, 8)
shap.summary_plot(shap_values[1], train_x)
plt.show()
```



### ▼ How do we read the above plot?

We have a ranked list of features on the right-hand side of the plot. The dots on the plot represent each data point in our training dataset. The color of the dot represents the magnitude of the corresponding feature value. The x-axis is the SHAP values. SHAP values are the effect that each feature value of each data point has on the model's output.

### Observations

- Credit Score is the strongest predictor of loan delinquency. As the credit score decreases, it increases the likelihood of the borrower defaulting on the loan.
- Postal Code, loan to value, number of borrowers = 1 and metropolitan statistical area are other strong predictors of loan delinquency.
- The location of the property does have a considerable effect on loan delinquency. As next steps, we must investigate the economic conditions of the property locations that have a high default rate.
- As loan to value increases, the likelihood of loan delinquency increases. We observed this pattern earlier.
- If the number of borrowers is 2 or greater, it reduces the likelihood of loan delinquency. This makes intuitive sense - as the number of borrowers increases, the sources of capital are greater and hence the chance to pay off the loan also increases.
- I expected debt-to-income ratio to be one of the top predictors of loan delinquency.
- Property type, occupancy status and number of units are weak predictors of loan delinquency.

Let us sample a data point from the training dataset and observe the effects of each feature value on the model prediction.

```
# Sample a data point
sample_loan = train_x.sample(n=1)
sample_index = sample_loan.index
sample_loan.head()
```



CREDIT\_SCORE METROPOLITAN\_STATISTICAL\_AREA NUMBER\_OF\_UNITS ORIGINAL\_COMBINED\_LOAN\_TO\_VALUE

# Effect of feature values on model prediction

```
shap.force_plot(explainer.expected_value[1], shap_values[1][sample_index, :], train_x.iloc[sample_index,:])
```

**Visualization omitted, Javascript library not loaded!**

Have you run `!initjs()` in this notebook? If this notebook was from another user you must also trust this notebook (File -> Trust notebook). If you are viewing this notebook on github the Javascript has been stripped for security. If you are using JupyterLab this error is because a JupyterLab extension has not yet been written

## How to read the above figure?

The above figure displays the contribution of each feature value on the base value of our model predictions. Base value is the average model output over the training dataset. Feature values that push the prediction value higher are shown in red and those pushing the predictions lower are shown in blue.

Sample various data points to observe the effects of the features on the model prediction.

## ▼ What can we do to improve our solution as we advance?

### Data collection

The dataset provided to us has limited variables, but the model was able to perform reasonably well. It would be beneficial to the business if we can collect the following information -

- Specific information about the borrower such as current address, past loan information, dependents and occupation.
- Public information on the property areas such as average household income, population, average family size, current job rate, percentage of homes owned, average percentage of mortgage loans defaulted, average cost of single family homes, types of public transportation available, number of schools in the area, average rank of schools in the area, crime rate, and number of hospitals.

### Feature Engineering

- Instead of one-hot encoding, perform binary encoding, frequency encoding, weight of evidence, etc.
- Create features from summary statistics such as frequency for categorical data and mean, median, mode, skewness, and kurtosis for numerical features.
- Discretize credit score, loan to value and debt-to-income ratio.
- Apply PCA and append the principal components to the feature matrix.
- Pass the feature matrix through an autoencoder to encode the features into a latent space and append the encodings to the feature matrix.

### Improve model performance

- Hyperparameter tuning.
- Ensemble techniques such as boosting, bagging or stacking.
- Try more complex models such as neural networks.