Importing Python modules for analysis

In [1]:
```
# import necessary python modules for analysis
import pandas as pd
import numpy as np
import seaborn as sns
from sklearn import preprocessing
import matplotlib.pyplot as plt
import copy

from sklearn.linear_model import LinearRegression


%matplotlib inline
```

Step 1 : Cleaning and Preparing the Data

Reading all the input files

In [2]:
```
# read the train data
train = pd.read_csv("Data for Cleaning & Modeling.csv", low_memory=False)

# read the test data
test = pd.read_csv("Holdout for Testing.csv", low_memory=False)

# read the metadata
metadata = pd.read_csv("Metadata.csv", encoding = "utf-8")
print(metadata)
```

```
   Variable                                         Definition
0        X1                            Interest Rate on the loan
1        X2                            A unique id for the loan.
2        X3                  A unique id assigned for the borrower.
3        X4                               Loan amount requested
4        X5                                  Loan amount funded
5        X6                       Investor-funded portion of loan
6        X7                         Number of payments (36 or 60)
7        X8                                          Loan grade
8        X9                                       Loan subgrade
9       X10                  Employer or job title (self-filled)
10      X11  Number of years employed (0 to 10; 10 = 10 or ...
11      X12  Home ownership status: RENT, OWN, MORTGAGE, OT...
12      X13                              Annual income of borrower
13      X14  Income verified, not verified, or income sourc...
14      X15                                Date loan was issued
15      X16                  Reason for loan provided by borrower
16      X17                Loan category, as provided by borrower
17      X18                   Loan title, as provided by borrower
18      X19                           First 3 numbers of zip code
19      X20                                   State of borrower
20      X21  A ratio calculated using the borrower's total ...
21      X22  The number of 30+ days past-due incidences of ...
22      X23  Date the borrower's earliest reported credit l...
23      X24  Number of inquiries by creditors during the pa...
24      X25  Number of months since the borrower's last del...
25      X26     Number of months since the last public record.
26      X27  Number of open credit lines in the borrower's ...
27      X28                  Number of derogatory public records
28      X29                        Total credit revolving balance
29      X30  Revolving line utilization rate, or the amount...
30      X31  The total number of credit lines currently in ...
31      X32  The initial listing status of the loan. Possib...
```

Understanding the data

In [3]:
```
train.shape
```

Out[3]:
```
(400000, 32)
```

The train data has 400,000 rows X 32 columns. The Target is Variable X1 which is Interest Rate on the loan along with 31 features X2 through X32.

In [4]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400000 entries, 0 to 399999
Data columns (total 32 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   X1      338990 non-null  object
 1   X2      399999 non-null  float64
 2   X3      399999 non-null  float64
 3   X4      399999 non-null  object
 4   X5      399999 non-null  object
 5   X6      399999 non-null  object
 6   X7      399999 non-null  object
 7   X8      338730 non-null  object
 8   X9      338730 non-null  object
 9   X10     376014 non-null  object
 10  X11     382462 non-null  object
 11  X12     338639 non-null  object
 12  X13     338972 non-null  float64
 13  X14     399999 non-null  object
 14  X15     399999 non-null  object
 15  X16     123560 non-null  object
 16  X17     399999 non-null  object
 17  X18     399981 non-null  object
 18  X19     399999 non-null  object
 19  X20     399999 non-null  object
 20  X21     399999 non-null  float64
 21  X22     399999 non-null  float64
 22  X23     399999 non-null  object
 23  X24     399999 non-null  float64
 24  X25     181198 non-null  float64
 25  X26     51155 non-null   float64
 26  X27     399999 non-null  float64
 27  X28     399999 non-null  float64
 28  X29     399999 non-null  float64
 29  X30     399733 non-null  object
 30  X31     399999 non-null  float64
 31  X32     399999 non-null  object
dtypes: float64(12), object(20)
memory usage: 97.7+ MB
```

In [5]:
```
test.shape
```

Out[5]:
```
(80000, 32)
```

The train data has 80,000 rows X 32 columns.

In [6]:
```
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 80000 entries, 0 to 79999
Data columns (total 32 columns):
```

```
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   X1      0 non-null      float64
 1   X2      80000 non-null  int64  
 2   X3      80000 non-null  int64  
 3   X4      80000 non-null  object 
 4   X5      80000 non-null  object 
 5   X6      80000 non-null  object 
 6   X7      80000 non-null  object 
 7   X8      80000 non-null  object 
 8   X9      80000 non-null  object 
 9   X10     75606 non-null  object 
 10  X11     75618 non-null  object 
 11  X12     80000 non-null  object 
 12  X13     80000 non-null  float64
 13  X14     80000 non-null  object 
 14  X15     80000 non-null  object 
 15  X16     15 non-null     object 
 16  X17     80000 non-null  object 
 17  X18     80000 non-null  object 
 18  X19     80000 non-null  object 
 19  X20     80000 non-null  object 
 20  X21     80000 non-null  float64
 21  X22     80000 non-null  int64  
 22  X23     80000 non-null  object 
 23  X24     80000 non-null  int64  
 24  X25     41296 non-null  float64
 25  X26     13839 non-null  float64
 26  X27     80000 non-null  int64  
 27  X28     80000 non-null  int64  
 28  X29     80000 non-null  int64  
 29  X30     79970 non-null  object 
 30  X31     80000 non-null  int64  
 31  X32     80000 non-null  object 
dtypes: float64(5), int64(8), object(19)
memory usage: 19.5+ MB
```

In [7]:
```
train.head()
```

Out[7]:

	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	...	X23	X24	X
0	11.89%	54734.0	80364.0	$25,000	$25,000	$19,080	36 months	B	B4	NaN	...	Feb-94	0.0	N
1	10.71%	55742.0	114426.0	$7,000	$7,000	$673	36 months	B	B5	CNN	...	Oct-00	0.0	N
2	16.99%	57167.0	137225.0	$25,000	$25,000	$24,725	36 months	D	D3	Web Programmer	...	Jun-00	0.0	4
3	13.11%	57245.0	138150.0	$1,200	$1,200	$1,200	36 months	C	C2	city of beaumont texas	...	Jan-85	0.0	6
4	13.57%	57416.0	139635.0	$10,800	$10,800	$10,692	36 months	C	C3	State Farm Insurance	...	Dec-96	1.0	5

5 rows × 32 columns

In [8]:
```
train.iloc[0].T
```

Out[8]:
```
X1                                                11.89%
X2                                               54734.0
X3                                               80364.0
X4                                               $25,000
X5                                               $25,000
```

```
X6                                                  $19,080
X7                                                36 months
X8                                                        B
X9                                                       B4
X10                                                     NaN
X11                                                < 1 year
X12                                                    RENT
X13                                                 85000.0
X14                                      VERIFIED - income
X15                                                  Aug-09
X16      Due to a lack of personal finance education an...
X17                                      debt_consolidation
X18                        Debt consolidation for on-time payer
X19                                                   941xx
X20                                                      CA
X21                                                   19.48
X22                                                     0.0
X23                                                  Feb-94
X24                                                     0.0
X25                                                     NaN
X26                                                     NaN
X27                                                    10.0
X28                                                     0.0
X29                                                 28854.0
X30                                                  52.10%
X31                                                    42.0
X32                                                       f
Name: 0, dtype: object
```

Checking Categorical features

In [9]:
```python
train.describe(include = 'object')
```

Out[9]:

	X1	X4	X5	X6	X7	X8	X9	X10	X11	X12	X
count	338990	399999	399999	399999	399999	338730	338730	376014	382462	338639	3999
unique	482	1339	1342	7036	2	7	35	187821	11	6	
top	10.99%	$10,000	$10,000	$10,000	36 months	B	B3	Teacher	10+ years	MORTGAGE	VERIF - inco
freq	11082	28417	28324	24319	292369	101668	24009	4222	128060	172112	1496

In [10]:
```python
test.describe(include = 'object')
```

Out[10]:

	X4	X5	X6	X7	X8	X9	X10	X11	X12	X14	X15
count	80000	80000	80000	80000	80000	80000	75606	75618	80000	80000	80000
unique	1236	1236	1283	2	7	35	31557	11	3	3	3
top	$10,000	$10,000	$10,000	36 months	C	C2	Teacher	10+ years	MORTGAGE	VERIFIED - income source	15-Jan
freq	5380	5380	5157	53630	22869	5003	1557	26723	38994	35712	30830

Cleaning the remaining features

We can see that features X10, X15, X16, X18 and X23 are not useful features for our analysis as they have a large number of unique values and hence we can drop these features

Feature X16 has 50% missing values in train set and it is a unique text data type that will most likely not be useful for our model. Also, our test set only has 15 out 80000 values available for this column, so we can drop X16.

In [11]:
```
print("Target variable X1 is missing {0} values".format(train[train['X1'].isna()].shape[
```

```
Target variable X1 is missing 61010 values
```

X1, our dependent variable (Interest Rate on the loan), is missing 61010 values and we can handle this by dropping the null values in our train set

In [12]:
```
# drop the rows with missing interest rate values in X1
train.dropna(subset=['X1'], inplace=True)

# there is one row in the data with nulls besides the loan interest rate value. we proce
train.dropna(subset=['X4'], inplace=True)
```

In [13]:
```
# Number of payments (36 or 60)
train['X7'].value_counts()
```

Out[13]:
```
 36 months    247791
 60 months     91198
Name: X7, dtype: int64
```

In [14]:
```
# Loan grade
train['X8'].value_counts()
```

Out[14]:
```
B    86121
C    76446
D    46984
A    45525
E    21628
F     8395
G     2024
Name: X8, dtype: int64
```

In [15]:
```
# Loan subgrade
train['X9'].value_counts()
```

Out[15]:
```
B3    20352
B4    19137
B2    16767
C1    16342
C2    16310
B5    15521
C3    15425
C4    14646
B1    14344
C5    13723
A5    13086
A4    11806
D1    11720
D2    10498
D3     9091
D4     8573
A3     7653
D5     7102
A2     6496
A1     6484
E1     5447
E2     5246
E3     4230
E4     3640
E5     3065
F1     2490
```

```
F2      1873
F3      1712
F4      1331
F5       989
G1       677
G2       511
G3       378
G4       252
G5       206
Name: X9, dtype: int64
```

In [16]:

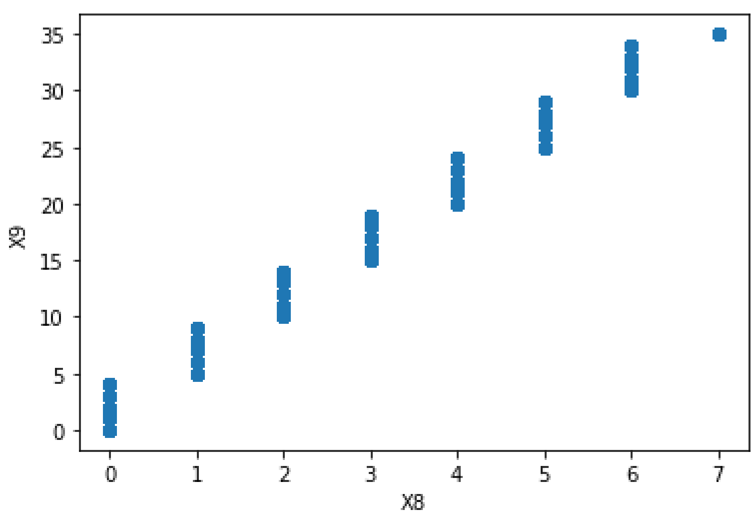
```
print("Categorical features X8 and X9 - loan grades and sub-grades have {0} missing valu
      .format(train[train['X8'].isna()].shape[0]))
```

```
Categorical features X8 and X9 - loan grades and sub-grades have 51866 missing values an
d hence we can drop nulls in these columns
```

In [17]:

```
le = preprocessing.LabelEncoder()
from scipy.stats import pearsonr

print("The correlation between X8 and X9 is:",
      pearsonr(le.fit_transform(train['X8'].values),le.fit_transform(train['X9'].values)

# Plot the lists as a scatter plot
plt.scatter(le.fit_transform(train['X8'].values), le.fit_transform(train['X9'].values))
# Label the axes
plt.xlabel('X8')
plt.ylabel('X9')

# Show the plot
plt.show()
```

```
The correlation between X8 and X9 is: PearsonRResult(statistic=0.9924288772525864, pvalu
e=0.0)
```



We can see that X8 and X9 have a strong positive linear relationship between the two variables and so we can drop feature X8.

In [18]:

```
# Employer or job title (self-filled)
print("Unique values is {0}".format(train['X10'].nunique()))
print("Missing values is {0}".format(train[train['X10'].isna()].shape[0]))
```

```
Unique values is 163395
Missing values is 20256
```

Since categorical feature X10 has numerous unique categories with a large number of missing values we can drop this feature.

In [19]:
```
# Number of years employed (0 to 10; 10 = 10 or more)
train['X11'].value_counts()
```

Out[19]:
```
10+ years    108491
2 years       30117
3 years       26670
< 1 year      26003
5 years       23072
1 year        21432
4 years       20259
6 years       19601
7 years       19445
8 years       16212
9 years       12893
Name: X11, dtype: int64
```

We can convert the feature X1, Number of years to numeric and impute nulls with least number of years (i.e., 1)

In [20]:
```
# Home ownership status: RENT, OWN, MORTGAGE, OTHER
train['X12'].value_counts()
```

Out[20]:
```
MORTGAGE    145958
RENT        115958
OWN          24976
OTHER          107
NONE            30
ANY              1
Name: X12, dtype: int64
```

In [21]:
```
test['X12'].value_counts()
```

Out[21]:
```
MORTGAGE    38994
RENT        32778
OWN          8228
Name: X12, dtype: int64
```

In [22]:
```
print("Missing values in feature X12 is : {0}".format(train[train['X12'].isna()].shape[0
```

```
Missing values in feature X12 is : 51959
```

In [23]:
```
print("We can see that only about {0}% of the total values are in classes (OTHER, NONE a
      .format(round(138/train['X12'].count()*100,2)))
print("Hence we can create dummy variables for categorical values MORTGAGE, RENT, OWN an
```

```
We can see that only about 0.05% of the total values are in classes (OTHER, NONE and AN
Y)
Hence we can create dummy variables for categorical values MORTGAGE, RENT, OWN and then
drop the feature X12
```

In [24]:
```
# Annual income of borrower
train[train['X13'].isna()].shape[0]
```

Out[24]:
```
51751
```

Since feature X13 has 51751 null values we can impute these missing values. Going with the assumption that people in similar Annual income brackets would qualify for similar loan grades, we can impute missing values with average Annual income of people falling in the same loan subgrade bucket (X9).

In [25]:
```
# Income verified, not verified, or income source was verified
train['X14'].value_counts()
```

Out[25]:
```
VERIFIED - income           127040
not verified                107873
```

```
VERIFIED - income source    104076
Name: X14, dtype: int64
```

We can create encoded integer values for the categorical feature X14 as it has 3 groups and does not contain any nulls

In [26]:
```
# Loan category, as provided by borrower
train['X17'].value_counts()
```

Out[26]:
```
debt_consolidation    198226
credit_card            75680
home_improvement       19625
other                  17154
major_purchase          7312
small_business          5359
car                     4115
medical                 3329
moving                  2138
wedding                 1934
vacation                1848
house                   1723
educational              279
renewable_energy         267
Name: X17, dtype: int64
```

In [27]:
```
test['X17'].value_counts()
```

Out[27]:
```
debt_consolidation    49884
credit_card           18660
home_improvement       3920
other                  3383
major_purchase         1232
small_business          668
medical                 619
car                     573
moving                  393
vacation                359
house                   266
renewable_energy         42
wedding                   1
Name: X17, dtype: int64
```

The top 2 famous Loan categories in both train and test sets are debt_consolidation and credit_card. We create dummy variables for these 2 categories and drop the rest of the values in feature X17.

In [28]:
```
# First 3 numbers of zip code
train['X19'].value_counts()
```

Out[28]:
```
945xx    3922
750xx    3703
112xx    3689
606xx    3419
100xx    3216
         ...
643xx       1
528xx       1
522xx       1
663xx       1
938xx       1
Name: X19, Length: 874, dtype: int64
```

We convert categorical variable X19 to type numeric by extracting the first 3 characters.

In [29]:
```
# State of borrower
```

```
train['X20'].value_counts().head()
```

Out[29]:

```
CA    52835
NY    29226
TX    26493
FL    22756
IL    13483
Name: X20, dtype: int64
```

Since we already have a more granular location feature : First 3 numbers of zip code, we can drop this State feature X20.

In [30]:

```
# Number of months since the borrower's last delinquency
train['X32'].value_counts()
```

Out[30]:

```
f    232600
w    106389
Name: X32, dtype: int64
```

We can convert the categorical values to numeric for feature X32.

In [31]:

```
# X25 : The initial listing status of the loan. Possible values are W, F
# X26 : Number of months since the last public record
# X30 : Revolving line utilization rate

print("The number of missing values in X25, X26 and X30 are {0}, {1}, and {2} respective
```

```
The number of missing values in X25, X26 and X30 are 185456, 295589, and 224 respectivel
y
```

We have more than 50% of values missing in features X25 and X26 in both the train and test data. Hence, we will remove the columns altogether as imputing these nulls would'nt be very effective.

However, we can input the missing values in feature X30 using Linear Regression.

Data pre-processing

In [32]:

```
def cleaning_and_preprocessing(df, type):
    # drop X2 and X3 from as they are unique identifiers and hence not useful for our mo
    df.drop(['X2', 'X3'], axis=1, inplace=True)

    # drop unnecessary categorical features
    df.drop(['X10','X15','X16','X18','X23'], axis=1, inplace=True)

    # clean X4, X5, X6 to remove ($ and ,) and X1, X30 to remove (%)
    for i in ('X4','X5','X6'):
        df[i]=df[i].map(lambda x:str(x).replace('$',''))
        df[i]=df[i].map(lambda x:str(x).replace(',',''))
        # convert to float
        df[i]=df[i].astype(float)

    if type == 'test' :
        #drop the target column in test data set
        df.drop(['X1'], axis=1, inplace=True)
        df['X30']=df['X30'].str.replace('%', '')
        # convert percentage to float
        df['X30']=df['X30'].map(lambda x: round(float(x)/100,4))
    else:
        for i in ('X1','X30'):
            df[i]=df[i].str.replace('%', '')
            # convert percentage to float
```

```
        df[i]=df[i].map(lambda x: round(float(x)/100,4))

# convert X7 with values 36, 60 from str to numeric
df['X7'] = pd.to_numeric(df['X7'].str.replace(' months', ''))

# drop nulls in X8 and X9 columns: loan grade and subgrade
df = df.copy()
df.dropna(subset=['X8', 'X9'], inplace=True)

# instatiate sklearn's labelencoder
le = preprocessing.LabelEncoder()

# drop feature X8
df.drop(['X8'], axis=1, inplace=True)

# create integer labels for categorical string features X9
df['X9'] = le.fit_transform(df['X9'].values)

# convert X11 to numeric and impute nulls
df['X11'] = df['X11'].replace('\D+','',regex=True)
df['X11']  = df["X11"].astype(float)
df['X11'].fillna(1.0, inplace=True)

# create dummy variables for X12 and drop OTHER, NONE and ANY
dummies_12  = pd.get_dummies(df['X12'])
dummies_12  = dummies_12.drop(['NONE','ANY','OTHER'], axis=1, errors='ignore')
df = df.join(dummies_12)
df.drop(['X12'], axis=1,inplace=True)

# impute nulls in X13 with average value of X9
df['X13'].fillna(df.groupby('X9')['X13'].transform(lambda x: round(x.mean())), inpla

# create integer labels for categorical feature X14
df['X14'] = le.fit_transform(df['X14'].values)

# create dummy variables for X17 and keep only debt_consolidation and credit_card
dummies_17  = pd.get_dummies(df['X17'])
dummies_17  = dummies_17[['debt_consolidation',  'credit_card']]
df = df.join(dummies_17)
df.drop(['X17'], axis=1,inplace=True)

# convert X19 to numeric type
df['X19'] = df['X19'].astype(str).str[:3]
df['X19'] = df['X19'].astype(float)

# drop feature X20
df.drop(['X20'], axis=1, inplace=True)

# create integer labels for categories in feature X32
df['X32'] = le.fit_transform(df['X32'].values)

# drop feature X25, X26
df.drop(['X25', 'X26'], axis=1, inplace=True)

# impute nulls in X30 using Linear Regression
lr = LinearRegression()

testdf = df[df['X30'].isnull()==True]
traindf = df[df['X30'].isnull()==False]
y = traindf['X30']
traindf = traindf.copy()
traindf.drop('X30', axis=1, inplace=True)
lr.fit(traindf,y)
testdf = testdf.copy()
testdf.drop('X30', axis=1, inplace=True)
pred = lr.predict(testdf)
```

```python
    testdf['X30'] = copy.copy(pred)
    traindf['X30'] = y

    frames = [traindf, testdf]
    lr_df = pd.concat(frames).sort_index()

    return lr_df
```

In [33]:

```python
# cleaning the train data
df = train.copy()
clean_train = cleaning_and_preprocessing(df, 'train')
clean_train.head()
```

Out[33]:

	X1	X4	X5	X6	X7	X9	X11	X13	X14	X19	...	X28	X29	X31	X32	M
0	0.1189	25000.0	25000.0	19080.0	36	8	1.0	85000.0	0	941.0	...	0.0	28854.0	42.0	0	
1	0.1071	7000.0	7000.0	673.0	36	9	1.0	65000.0	2	112.0	...	0.0	33623.0	7.0	0	
2	0.1699	25000.0	25000.0	24725.0	36	17	1.0	70000.0	0	100.0	...	0.0	19878.0	17.0	0	
3	0.1311	1200.0	1200.0	1200.0	36	11	10.0	54000.0	2	777.0	...	0.0	2584.0	31.0	0	
4	0.1357	10800.0	10800.0	10692.0	36	12	6.0	32000.0	2	67.0	...	0.0	3511.0	40.0	0	

5 rows × 24 columns

In [34]:

```python
clean_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 287123 entries, 0 to 399999
Data columns (total 24 columns):
 #   Column              Non-Null Count   Dtype  
---  ------              --------------   -----  
 0   X1                  287123 non-null  float64
 1   X4                  287123 non-null  float64
 2   X5                  287123 non-null  float64
 3   X6                  287123 non-null  float64
 4   X7                  287123 non-null  int64  
 5   X9                  287123 non-null  int64  
 6   X11                 287123 non-null  float64
 7   X13                 287123 non-null  float64
 8   X14                 287123 non-null  int64  
 9   X19                 287123 non-null  float64
 10  X21                 287123 non-null  float64
 11  X22                 287123 non-null  float64
 12  X24                 287123 non-null  float64
 13  X27                 287123 non-null  float64
 14  X28                 287123 non-null  float64
 15  X29                 287123 non-null  float64
 16  X31                 287123 non-null  float64
 17  X32                 287123 non-null  int64  
 18  MORTGAGE            287123 non-null  uint8  
 19  OWN                 287123 non-null  uint8  
 20  RENT                287123 non-null  uint8  
 21  debt_consolidation  287123 non-null  uint8  
 22  credit_card         287123 non-null  uint8  
 23  X30                 287123 non-null  float64
dtypes: float64(15), int64(4), uint8(5)
memory usage: 45.2 MB
```

In [35]:

```python
df = test.copy()
clean_test = cleaning_and_preprocessing(df, 'test')
clean_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
Int64Index: 80000 entries, 0 to 79999
Data columns (total 23 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   X4                  80000 non-null  float64
 1   X5                  80000 non-null  float64
 2   X6                  80000 non-null  float64
 3   X7                  80000 non-null  int64  
 4   X9                  80000 non-null  int64  
 5   X11                 80000 non-null  float64
 6   X13                 80000 non-null  float64
 7   X14                 80000 non-null  int64  
 8   X19                 80000 non-null  float64
 9   X21                 80000 non-null  float64
 10  X22                 80000 non-null  int64  
 11  X24                 80000 non-null  int64  
 12  X27                 80000 non-null  int64  
 13  X28                 80000 non-null  int64  
 14  X29                 80000 non-null  int64  
 15  X31                 80000 non-null  int64  
 16  X32                 80000 non-null  int64  
 17  MORTGAGE            80000 non-null  uint8  
 18  OWN                 80000 non-null  uint8  
 19  RENT                80000 non-null  uint8  
 20  debt_consolidation  80000 non-null  uint8  
 21  credit_card         80000 non-null  uint8  
 22  X30                 80000 non-null  float64
dtypes: float64(8), int64(10), uint8(5)
memory usage: 12.0 MB
```

In [36]:
```
clean_train.corr(method='pearson', min_periods=1)
```

Out[36]:

	X1	X4	X5	X6	X7	X9	X11	X
X1	1.000000	0.178780	0.179757	0.182324	0.456793	0.976371	0.035614	-0.0347
X4	0.178780	1.000000	0.998286	0.994048	0.411542	0.185626	0.135405	0.3264
X5	0.179757	0.998286	1.000000	0.996125	0.409183	0.185148	0.135841	0.3257
X6	0.182324	0.994048	0.996125	1.000000	0.410010	0.183821	0.137930	0.3239
X7	0.456793	0.411542	0.409183	0.410010	1.000000	0.480509	0.091401	0.0624
X9	0.976371	0.185626	0.185148	0.183821	0.480509	1.000000	0.027749	-0.0249
X11	0.035614	0.135405	0.135841	0.137930	0.091401	0.027749	1.000000	0.0849
X13	-0.034799	0.326455	0.325759	0.323976	0.062413	-0.024950	0.084987	1.0000
X14	-0.234582	-0.364182	-0.363382	-0.365268	-0.272915	-0.219160	-0.030525	-0.0941
X19	-0.004947	-0.010005	-0.009939	-0.009525	-0.033043	-0.009133	-0.007011	-0.0098
X21	0.157820	0.059974	0.061531	0.065712	0.085997	0.143438	0.039530	-0.1677
X22	0.092090	0.008841	0.009477	0.010380	0.005285	0.092090	0.032886	0.0543
X24	0.209053	-0.001913	-0.002134	-0.003835	0.026408	0.210626	-0.003188	0.0587
X27	0.020067	0.204486	0.205374	0.206444	0.072992	0.018573	0.056123	0.1416
X28	0.072836	-0.075370	-0.074805	-0.073140	-0.018998	0.072493	0.021082	-0.016
X29	0.009964	0.343620	0.343174	0.341424	0.097248	0.011800	0.101830	0.2892
X31	-0.026944	0.237797	0.237668	0.238120	0.098345	-0.024505	0.116739	0.2038
X32	-0.009242	0.052969	0.055471	0.060929	0.039136	0.003554	0.023598	0.0196
MORTGAGE	-0.055628	0.174716	0.174548	0.174287	0.100115	-0.054735	0.164384	0.1440

OWN	0.008065	-0.029611	-0.029333	-0.028682	-0.015602	0.007654	-0.002858	-0.0362
RENT	0.050638	-0.166421	-0.166316	-0.166738	-0.097540	0.050298	-0.170326	-0.1294
debt_consolidation	0.076687	0.121373	0.122022	0.123340	0.076936	0.069039	0.036366	-0.0156
credit_card	-0.148264	0.023050	0.023844	0.025304	-0.055361	-0.156551	-0.012264	-0.0087
X30	0.343261	0.118543	0.119898	0.122381	0.085016	0.323337	0.050991	0.0286

24 rows × 24 columns

In [37]:
```
clean_train.describe()
```

Out[37]:

	X1	X4	X5	X6	X7	X9	
count	287123.000000	287123.000000	287123.000000	287123.000000	287123.000000	287123.000000	287
mean	0.139450	14271.874423	14242.514793	14172.461381	42.448303	11.045043	
std	0.043772	8257.961456	8243.548262	8263.798986	10.638564	6.526639	
min	0.054200	500.000000	500.000000	0.000000	36.000000	0.000000	
25%	0.109900	8000.000000	8000.000000	8000.000000	36.000000	6.000000	
50%	0.136800	12000.000000	12000.000000	12000.000000	36.000000	10.000000	
75%	0.167800	20000.000000	20000.000000	19900.000000	60.000000	15.000000	
max	0.260600	35000.000000	35000.000000	35000.000000	60.000000	34.000000	

8 rows × 24 columns

Step 2 : Build Machine Learning model in Python to predict the interest rates assigned to loans.

We start by standardising the data. We use MinMaxScaler in order to scale the date to values that lie between 0 and 1. This scaling technique us not sensitive to ouliers and is suitable for datasets with extreme values and hence we can use this instead of the StandardScaler

The MinMaxScaler calculates the minimum and maximum values of each feature and subtracts the minimum from each feature and divides the resulting values by the range (max - min). The formula used is (x - min(x)) / (max(x) - min(x)).

We further take a look at the Feature importance for each of the following implemented models.

Feature importance refers to a class of techniques for assigning scores to input features to a predictive model that indicates the relative importance of each feature when making a prediction.

The scores are useful and can be used in a range of situations in a predictive modeling problem, such as:

1. **Feature importance scores can provide insight into the dataset.** The relative scores can highlight which features may be most relevant to the target, and the converse, which features are the least relevant.

2. **Feature importance scores can provide insight into the model.** Inspecting the importance score provides insight into that specific model and which features are the most important and least important to the model when making a prediction.

3. **Feature importance can be used to improve a predictive model.** This can be achieved by using the importance scores to select those features to delete (lowest scores) or those features to keep (highest scores). This process of deleting features is called dimensionality reduction, and in some cases, can improve the performance of the model.

Predict interest rates on training and test data

Modeling

The choice of machine learning model depends on the specific problem and the characteristics of the data. Each model has its own strengths and weaknesses, and it's important to carefully evaluate their performance on a given dataset before choosing one.

A. LinearRegression

A simple, linear model that predicts a continuous output based on one or more input features. It works by finding the best-fitting line through the data.

Pros:

- Simple and easy to implement.
- Can be trained quickly on large datasets.
- Provides interpretable results and can help identify important features.

Cons:

- Assumes a linear relationship between the features and the target variable.
- Not suitable for datasets with nonlinear relationships.
- Can be sensitive to outliers and noise in the data.

In [38]:

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn import preprocessing

error_metrics = pd.DataFrame(columns=['Model', 'R-Squared', 'Adjusted R-Squared', 'RMSE'

# Split the target variable and features
X = clean_train.drop("X1", axis=1).values
y = clean_train["X1"].values

# Scaling the data
min_max = preprocessing.MinMaxScaler()
X = min_max.fit_transform(X)

# Split the train data into training and validation sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42

# Define the model
LR = LinearRegression()

# Define the stratified k-fold cross-validation object
k_fold = KFold(n_splits=5, shuffle=True, random_state=0)

# Calculate the cross-validation scores
```

```
score = cross_val_score(LR, X_train, y_train, cv=k_fold, scoring='r2').mean()
print("LR score : ", score)
```

```
LR score :  0.957821987997348
```

Cross - Validation

Cross-validation is a statistical method used to estimate the performance and accuracy of our machine learning models. It is used to protect against overfitting in a predictive model.

There are different types of cross validation methods, and they could be classified into two broad categories – Non-exhaustive and Exhaustive Methods.

Non-exhaustive Methods Non-exhaustive cross validation methods, as the name suggests do not compute all ways of splitting the original data.

Holdout method This is a quite basic and simple approach in which we divide our entire dataset into two parts viz- training data and testing data. As the name, we train the model on training data and then evaluate on the testing set.

K fold cross validation This is one way to improve the holdout method. This method guarantees that the score of our model does not depend on the way we picked the train and test set. The data set is divided into k number of subsets and the holdout method is repeated k number of times.

Exhaustive Methods Exhaustive cross validation methods and test on all possible ways to divide the original sample into a training and a validation set. Some exhaustive methods are Leave-P-out Cross validation and Leave-one-out Cross validation

I have implemented the **Holdout** and **K fold Cross validation methods** for model validation in order to avoid overfitting.

Accuracy Metrics

R-squared is a relative measure of fit. Adjusted R squared is a modified version of R square, where it is adjusted for the number of independent variables in the model. RMSE is an absolute measure of fit.

R-squared is a scale-free score i.e. irrespective of the values being small or large, the value of R square will be less than one. The lower value of MAE, MSE, and RMSE implies higher accuracy of a regression model. However, a higher value of R square is considered desirable.

The **RMSE** tells how well a regression model can predict the value of a response variable in absolute terms while R- Squared tells how well the predictor variables can explain the variation in the response variable. The RMSE is particularly useful for comparing the fit of different regression models.

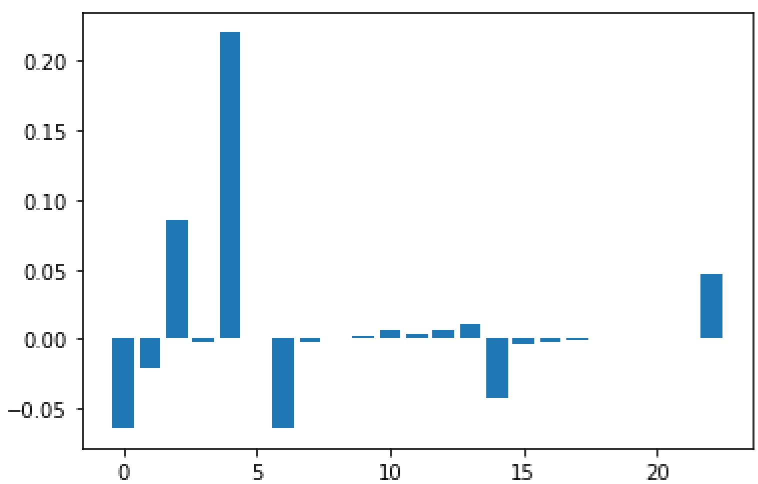
In [39]:
```
LR.fit(X_train, y_train)
pred = LR.predict(X_test)
r2 = r2_score(y_test, pred)
adj_r2 = 1 - (1-LR.score(X_test, y_test))*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-1
rmse = np.sqrt(mean_squared_error(y_test, pred))
error_metrics = pd.concat([error_metrics, pd.DataFrame.from_records([{'Model': 'LinearRe
                                                  'R-Squared': r2,
                                                  'Adjusted R-Squared': adj_r2,
                                                  'RMSE': rmse}])])

print (" R-squared : {0} \n Adjusted R-squared : {1} \n RMSE : {2}".format(r2, adj_r2, r
```

```
R-squared : 0.9577897958818697
Adjusted R-squared : 0.9577785219198116
RMSE : 0.0089999057712539
```

In [40]:

```python
# Feature importance of Linear Regression Model

# get importance
importance = LR.coef_
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
plt.bar([x for x in range(len(importance))], importance)
plt.show()
```

```
Feature: 0, Score: -0.06388
Feature: 1, Score: -0.02120
Feature: 2, Score: 0.08623
Feature: 3, Score: -0.00158
Feature: 4, Score: 0.22039
Feature: 5, Score: 0.00091
Feature: 6, Score: -0.06391
Feature: 7, Score: -0.00258
Feature: 8, Score: 0.00025
Feature: 9, Score: 0.00142
Feature: 10, Score: 0.00649
Feature: 11, Score: 0.00314
Feature: 12, Score: 0.00681
Feature: 13, Score: 0.01086
Feature: 14, Score: -0.04252
Feature: 15, Score: -0.00403
Feature: 16, Score: -0.00149
Feature: 17, Score: -0.00015
Feature: 18, Score: 0.00006
Feature: 19, Score: -0.00001
Feature: 20, Score: 0.00097
Feature: 21, Score: 0.00079
Feature: 22, Score: 0.04741
```



In [41]:

```python
clean_train.head()
```

Out[41]:

	X1	X4	X5	X6	X7	X9	X11	X13	X14	X19	...	X28	X29	X31	X32
0	0.1189	25000.0	25000.0	19080.0	36	8	1.0	85000.0	0	941.0	...	0.0	28854.0	42.0	0
1	0.1071	7000.0	7000.0	673.0	36	9	1.0	65000.0	2	112.0	...	0.0	33623.0	7.0	0
2	0.1699	25000.0	25000.0	24725.0	36	17	1.0	70000.0	0	100.0	...	0.0	19878.0	17.0	0
3	0.1311	1200.0	1200.0	1200.0	36	11	10.0	54000.0	2	777.0	...	0.0	2584.0	31.0	0

4	0.1357	10800.0	10800.0	10692.0	36	12	6.0	32000.0	2	67.0	...	0.0	3511.0	40.0	0

5 rows × 24 columns

B. RandomForestRegressor

A type of ensemble model that combines multiple decision trees to make predictions. It works by randomly selecting subsets of the data and features, and building a decision tree on each subset. The predictions from each tree are then averaged to produce the final output.

Pros:

- Can handle both numerical and categorical features.
- Can capture complex nonlinear relationships in the data.
- Can handle missing values and noisy data.

Cons:

- Can be slow to train on large datasets with many features.
- Difficult to interpret compared to simpler models like Linear Regression.
- May overfit if the number of trees is too high.

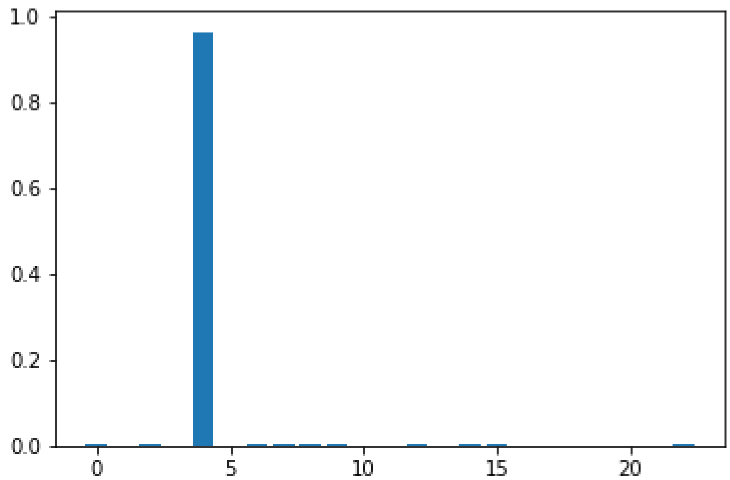
In [42]:
```
from sklearn.ensemble import RandomForestRegressor

# Define the model
RF = RandomForestRegressor(n_estimators=20, random_state=0)

# Fit the Random Forest Model
RF.fit(X_train, y_train)
pred = RF.predict(X_test)
r2 = r2_score(y_test, pred)
adj_r2 = 1 - (1-RF.score(X_test, y_test))*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-1
rmse = np.sqrt(mean_squared_error(y_test, pred))
error_metrics = pd.concat([error_metrics, pd.DataFrame.from_records([{'Model': 'RandomFo
                                      'R-Squared': r2,
                                      'Adjusted R-Squared': adj_r2,
                                      'RMSE': rmse}])])


print (" R-squared : {0} \n Adjusted R-squared : {1} \n RMSE : {2}".format(r2, adj_r2, r
```

```
 R-squared : 0.9697064306285292
 Adjusted R-squared : 0.9696983394913543
 RMSE : 0.007624450555355222
```

In [43]:
```
# Feature importance of Random Forest Regressor Model

# get importance
importance = RF.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
plt.bar([x for x in range(len(importance))], importance)
plt.show()
```

```
Feature: 0, Score: 0.00178
Feature: 1, Score: 0.00132
Feature: 2, Score: 0.00368
Feature: 3, Score: 0.00082
```

```
Feature: 4, Score: 0.96404
Feature: 5, Score: 0.00129
Feature: 6, Score: 0.00268
Feature: 7, Score: 0.00153
Feature: 8, Score: 0.00311
Feature: 9, Score: 0.00375
Feature: 10, Score: 0.00046
Feature: 11, Score: 0.00093
Feature: 12, Score: 0.00214
Feature: 13, Score: 0.00034
Feature: 14, Score: 0.00370
Feature: 15, Score: 0.00239
Feature: 16, Score: 0.00100
Feature: 17, Score: 0.00031
Feature: 18, Score: 0.00019
Feature: 19, Score: 0.00032
Feature: 20, Score: 0.00032
Feature: 21, Score: 0.00024
Feature: 22, Score: 0.00366
```



C. DecisionTreeRegressor

A tree-based model that makes predictions by splitting the data into smaller and smaller subsets, based on the values of the input features. The final predictions are made based on the values of the target variable in the leaf nodes of the tree.

Pros:

- Easy to interpret and understand.
- Can handle both numerical and categorical features.
- Can capture nonlinear relationships in the data.

Cons:

- Can overfit the training data.
- Prone to instability and high variance.
- Can be sensitive to small changes in the training data.

In [44]:
```
from sklearn.tree import DecisionTreeRegressor

# Define the model
DT = DecisionTreeRegressor(random_state=0)

# Fit the Decision Tree Model
DT.fit(X_train, y_train)
```

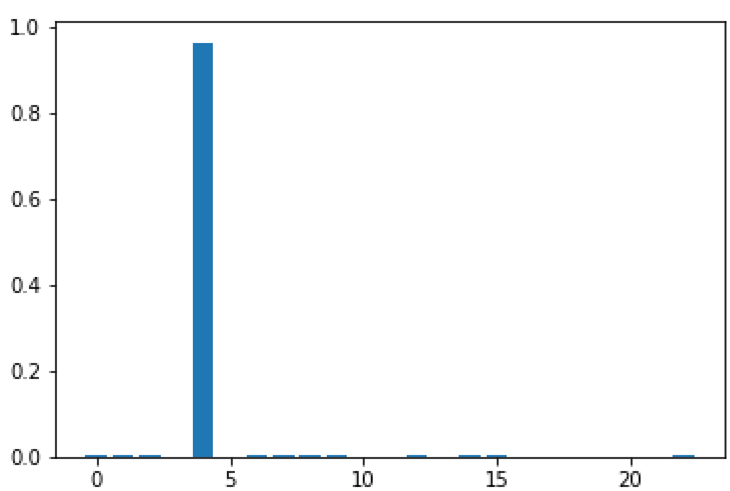
```
pred = DT.predict(X_test)
r2 = r2_score(y_test, pred)
adj_r2 = 1 - (1-DT.score(X_test, y_test))*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-1
rmse = np.sqrt(mean_squared_error(y_test, pred))
error_metrics = pd.concat([error_metrics, pd.DataFrame.from_records([{'Model': 'Decision
                                        'R-Squared': r2,
                                        'Adjusted R-Squared': adj_r2,
                                        'RMSE': rmse}])])

print (" R-squared : {0} \n Adjusted R-squared : {1} \n RMSE : {2}".format(r2, adj_r2, r
```

```
 R-squared : 0.9411102859067279
 Adjusted R-squared : 0.9410945569990815
 RMSE : 0.01063048432684732
```

In [45]:

```
# Feature importance of Decision Tree Regressor Model

# get importance
importance = DT.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
plt.bar([x for x in range(len(importance))], importance)
plt.show()
```

```
Feature: 0, Score: 0.00178
Feature: 1, Score: 0.00141
Feature: 2, Score: 0.00382
Feature: 3, Score: 0.00074
Feature: 4, Score: 0.96410
Feature: 5, Score: 0.00127
Feature: 6, Score: 0.00271
Feature: 7, Score: 0.00150
Feature: 8, Score: 0.00310
Feature: 9, Score: 0.00364
Feature: 10, Score: 0.00047
Feature: 11, Score: 0.00091
Feature: 12, Score: 0.00218
Feature: 13, Score: 0.00034
Feature: 14, Score: 0.00368
Feature: 15, Score: 0.00248
Feature: 16, Score: 0.00100
Feature: 17, Score: 0.00028
Feature: 18, Score: 0.00018
Feature: 19, Score: 0.00034
Feature: 20, Score: 0.00029
Feature: 21, Score: 0.00024
Feature: 22, Score: 0.00354
```



D. GradientBoostingRegressor

A type of ensemble model that combines multiple weak learners, typically decision trees, to make predictions. It works by iteratively fitting new models to the residuals of the previous models, with the goal of minimizing the overall error.

Pros:

- Can handle both numerical and categorical features.
- Can capture complex nonlinear relationships in the data.
- Can handle missing values and noisy data.

Cons:

- Can be computationally expensive and slow to train on large datasets.
- Can be sensitive to hyperparameters and prone to overfitting.
- Difficult to interpret compared to simpler models like Linear Regression.

In [46]:

```
from sklearn.ensemble import GradientBoostingRegressor

# Define the model
GB = GradientBoostingRegressor(random_state=0)

# Fit the XGBoost Model
GB.fit(X_train, y_train)
pred = GB.predict(X_test)
r2 = r2_score(y_test, pred)
adj_r2 = 1 - (1-GB.score(X_test, y_test))*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-1
rmse = np.sqrt(mean_squared_error(y_test, pred))
error_metrics = pd.concat([error_metrics, pd.DataFrame.from_records([{'Model': 'Gradient
                                       'R-Squared': r2,
                                       'Adjusted R-Squared': adj_r2,
                                       'RMSE': rmse}])])

print (" R-squared : {0} \n Adjusted R-squared : {1} \n RMSE : {2}".format(r2, adj_r2, r
```

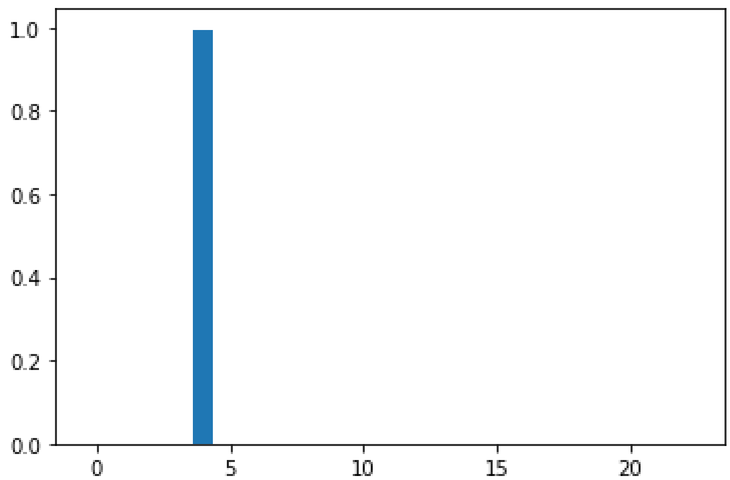
```
 R-squared : 0.9689947349247574
 Adjusted R-squared : 0.9689864537001255
 RMSE : 0.007713492345900862
```

In [47]:

```
# Feature importance of Gradient Boosting Regressor Model

# get importance
importance = GB.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
plt.bar([x for x in range(len(importance))], importance)
plt.show()
```

```
Feature: 0, Score: 0.00053
Feature: 1, Score: 0.00000
Feature: 2, Score: 0.00112
Feature: 3, Score: 0.00032
Feature: 4, Score: 0.99507
Feature: 5, Score: 0.00006
Feature: 6, Score: 0.00005
Feature: 7, Score: 0.00101
Feature: 8, Score: 0.00000
Feature: 9, Score: 0.00023
Feature: 10, Score: 0.00000
```

```
Feature: 11, Score: 0.00005
Feature: 12, Score: 0.00027
Feature: 13, Score: 0.00002
Feature: 14, Score: 0.00028
Feature: 15, Score: 0.00002
Feature: 16, Score: 0.00058
Feature: 17, Score: 0.00000
Feature: 18, Score: 0.00000
Feature: 19, Score: 0.00000
Feature: 20, Score: 0.00000
Feature: 21, Score: 0.00000
Feature: 22, Score: 0.00039
```



E. XGBRegressor

An implementation of the gradient boosting algorithm that is optimized for speed and efficiency. It uses a combination of parallel processing, memory optimization, and approximation techniques to speed up the training process.

Pros:

- Very fast and efficient implementation.
- Can handle large datasets with many features.
- Can capture complex nonlinear relationships in the data.

Pros:

- Can be sensitive to hyperparameters and prone to overfitting.
- Difficult to interpret compared to simpler models like Linear Regression.
- Requires careful tuning of hyperparameters to achieve good performance.

In [48]:
```python
from xgboost import XGBRegressor

# Define the model
XGB = XGBRegressor(random_state=0)

# Fit the XGBoost Model
XGB.fit(X_train, y_train)
pred = XGB.predict(X_test)
r2 = r2_score(y_test, pred)
adj_r2 = 1 - (1-XGB.score(X_test, y_test))*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-
rmse = np.sqrt(mean_squared_error(y_test, pred))
error_metrics = pd.concat([error_metrics, pd.DataFrame.from_records([{'Model': 'XGBRegre
                                      'R-Squared': r2,
                                      'Adjusted R-Squared': adj_r2,
```

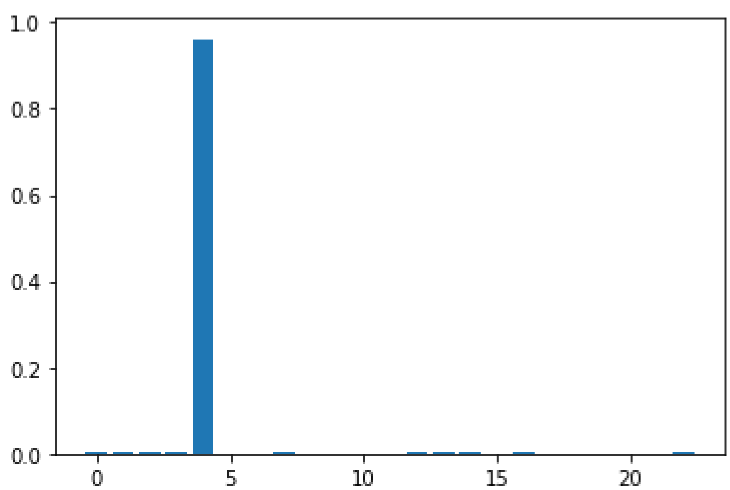
```
                                        'RMSE': rmse}]))])

print (" R-squared : {0} \n Adjusted R-squared : {1} \n RMSE : {2}".format(r2, adj_r2, r
```

```
 R-squared : 0.9741285181163061
 Adjusted R-squared : 0.9741216080785263
 RMSE : 0.007046021778499427
```

In [49]:

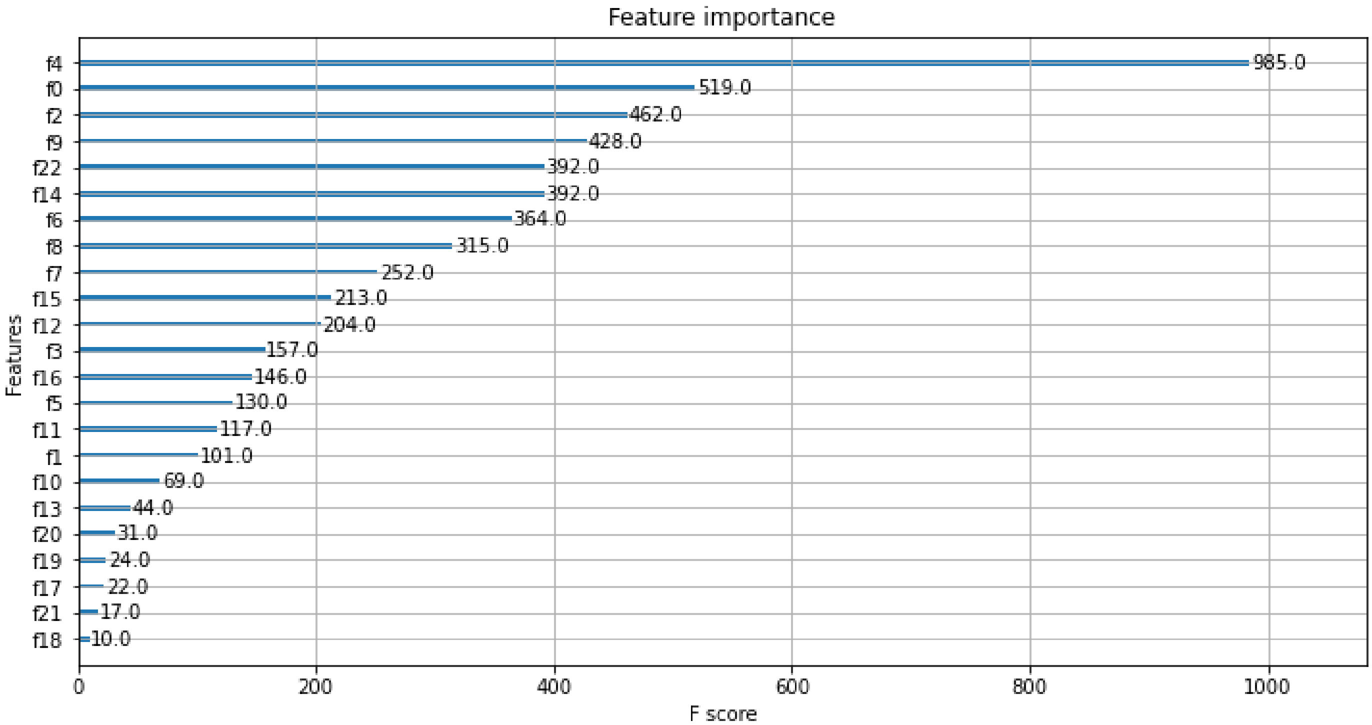
```
# Feature importance of XGBoost Regressor Model

# get importance
importance = XGB.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
plt.bar([x for x in range(len(importance))], importance)
plt.show()
```

```
Feature: 0, Score: 0.00315
Feature: 1, Score: 0.00175
Feature: 2, Score: 0.00558
Feature: 3, Score: 0.00409
Feature: 4, Score: 0.96099
Feature: 5, Score: 0.00113
Feature: 6, Score: 0.00100
Feature: 7, Score: 0.00386
Feature: 8, Score: 0.00064
Feature: 9, Score: 0.00139
Feature: 10, Score: 0.00069
Feature: 11, Score: 0.00104
Feature: 12, Score: 0.00167
Feature: 13, Score: 0.00162
Feature: 14, Score: 0.00155
Feature: 15, Score: 0.00078
Feature: 16, Score: 0.00448
Feature: 17, Score: 0.00062
Feature: 18, Score: 0.00067
Feature: 19, Score: 0.00057
Feature: 20, Score: 0.00057
Feature: 21, Score: 0.00069
Feature: 22, Score: 0.00147
```



In [50]:

```
from xgboost import plot_importance

# plot F-score of each feature using xgboost's .plot_importance() method
plt.rcParams["figure.figsize"] = (12,6)
plt.show(plot_importance(XGB))
```



F. AdaBoostRegressor

Another type of ensemble model that combines multiple weak learners, typically decision trees, to make predictions. It works by iteratively adjusting the weights of the training data, to focus on the examples that the previous models got wrong. The final predictions are made by averaging the outputs of all the models.

Pros:

- Can handle both numerical and categorical features.
- Can capture complex nonlinear relationships in the data.
- Can handle missing values and noisy data.

Cons:

- Can be sensitive to outliers and noise in the data.
- Prone to overfitting if the weak learners are too complex.
- Can be computationally expensive and slow to train on large datasets.

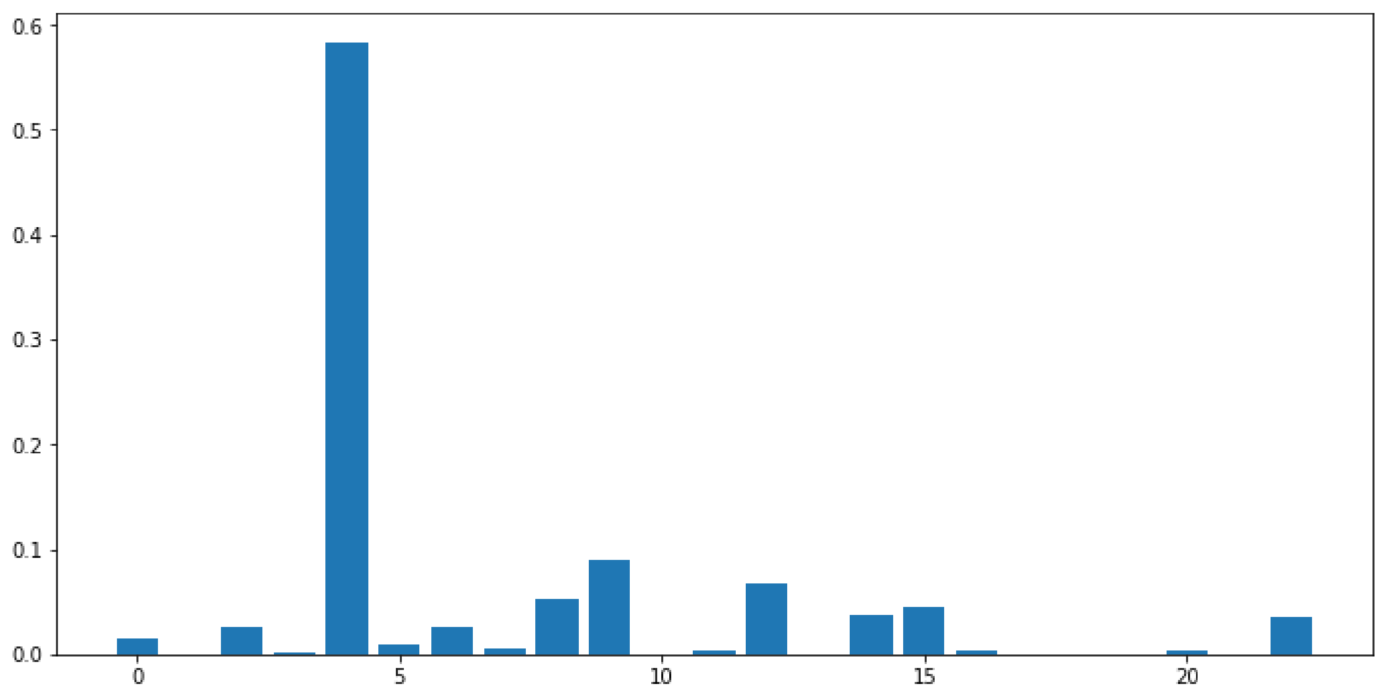
In [51]:

```
from sklearn.ensemble import AdaBoostRegressor

# Define the model
AB = AdaBoostRegressor(random_state=0)

# Fit the AdaBoostRegressor Model
AB.fit(X_train, y_train)
pred = AB.predict(X_test)
r2 = r2_score(y_test, pred)
adj_r2 = 1 - (1-AB.score(X_test, y_test))*(len(y_test)-1)/(len(y_test)-X_test.shape[1]-1
rmse = np.sqrt(mean_squared_error(y_test, pred))
error_metrics = pd.concat([error_metrics, pd.DataFrame.from_records([{'Model': 'AdaBoost
                                      'R-Squared': r2,
                                      'Adjusted R-Squared': adj_r2,
                                      'RMSE': rmse}])])

print (" R-squared : {0} \n Adjusted R-squared : {1} \n RMSE : {2}".format(r2, adj_r2, r
```

```
R-squared : 0.8732274859951589
Adjusted R-squared : 0.8731936262083426
RMSE : 0.015597172667412045
```

In [52]:
```python
# Feature importance of AdaBoost Regressor Model

# get importance
importance = AB.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
plt.bar([x for x in range(len(importance))], importance)
plt.show()
```

```
Feature: 0, Score: 0.01466
Feature: 1, Score: 0.00019
Feature: 2, Score: 0.02502
Feature: 3, Score: 0.00195
Feature: 4, Score: 0.58292
Feature: 5, Score: 0.00859
Feature: 6, Score: 0.02659
Feature: 7, Score: 0.00527
Feature: 8, Score: 0.05281
Feature: 9, Score: 0.08975
Feature: 10, Score: 0.00000
Feature: 11, Score: 0.00248
Feature: 12, Score: 0.06682
Feature: 13, Score: 0.00009
Feature: 14, Score: 0.03700
Feature: 15, Score: 0.04483
Feature: 16, Score: 0.00267
Feature: 17, Score: 0.00000
Feature: 18, Score: 0.00000
Feature: 19, Score: 0.00000
Feature: 20, Score: 0.00273
Feature: 21, Score: 0.00000
Feature: 22, Score: 0.03563
```



Selecting the best model by comparing model accuracy and predicting the Target for the Test set

Across all the models implemented, we can observe that clearly **Feature X4 - Loan Amount Requested** is the most important feature.

Let us comapre the accuracy scores and select the best model in order to predict the **Target - X1 Interest Rate on the loan** on our given test set.

In [53]:
```
error_metrics
```

Out[53]:

	Model	R-Squared	Adjusted R-Squared	RMSE
0	LinearRegression	0.95779	0.957779	0.009
0	RandomForestRegressor	0.969706	0.969698	0.007624
0	DecisionTreeRegressor	0.94111	0.941095	0.01063
0	GradientBoostingRegressor	0.968995	0.968986	0.007713
0	XGBRegressor	0.974129	0.974122	0.007046
0	AdaBoostRegressor	0.873227	0.873194	0.015597

We select the XGBoost model as it has the best R-Squared values and the least RMSE compared to all other models used. We can also see that the model is not susceptible to underfitting or overfitting by looking at the cross-validation scores.

In [54]:
```
# Define the model
XGB = XGBRegressor(random_state=0)

# Fit the XGBoost Model
XGB.fit(X, y)


# Define the stratified k-fold cross-validation object
k_fold = KFold(n_splits=5, shuffle=True, random_state=0)

# Calculate the cross-validation scores
score = cross_val_score(LR, X, y, cv=k_fold, scoring='r2')
print("Cross Val scores for XGBoost are :", score)
print("Mean LR score : ", score.mean())
```

```
Cross Val scores for XGBoost are : [0.95871383 0.95683594 0.95776537 0.95692262 0.958840
2 ]
Mean LR score :  0.9578155910441465
```

In [55]:
```
# Using out hold out test data
holdout_X = clean_test.values

# Scaling the data
min_max = preprocessing.MinMaxScaler()
holdout_X = min_max.fit_transform(holdout_X)
```

In [56]:
```
result_pred = XGB.predict(holdout_X)

result_df = pd.DataFrame(data = result_pred, columns = ['x1'])
```

In [57]:
```
result_df.describe()
```

Out[57]:

	x1
count	80000.000000
mean	0.135625

std	0.043559
min	0.033076
25%	0.104990
50%	0.133726
75%	0.161445
max	0.270279

In [58]:
```
result_df.to_csv('Results from SwathiGanesan_12372237.csv', index=False)
```