AI & ML Exercise.

Peter Muwonge 2100060028

1. Find activation functions in neural networks?

- Sigmoid (logistic)
- ReLU function (hyperbolic tangent)
- Tanh
- Softmax function
- Binary.

2. List the various Applications of

Clustering

- Identification of cancer cells.
- Customer segmentation.
- Used in search engines
- Species Classification in biology

Neural networks:

- Marketing and Sales
- Health care Scenarios (oncology)
- Natural language processing

Scanned with AnyScanner

iii Types of machine Learning

- unsupervised learning
- Supervised learning
- reinforcement learning
- Semi supervised learning

Advantage and disadvantages of neural networks

merits

- It doesn't need reprogramming
- when an item of the network declines, it can continue working without issue
- It can implement complex tasks, unlike linear program.

Demerits

- needs ~~training~~ training to operate
- It needs high processing time for big networks

2100060028 5th/9/2023

AI&ML assignment 2

1. What are the use cases of artificial neural-networks?

→ Marketing and sales, help to study customer behaviour in order to influence purchases

→ Health care - enhances diagonistic ability hence leading to overall improvement in the quality of medical care.

→ Energy sector / grid, by predicting demand and guiding on regional supply preparedness.

— Process and quality control.

2. What are applications of Neural networks.

— ~~Deep learning~~

— facial recognition such as in premise access, airports

— Social media, studies behaviour pattern of a user ~~mind~~

— Aerospace, employed in modelling non-linear - time dynamic systems.

Defence, used in maritime and air patrol for controlling automatic/autonomous drone.

What are the learning techniques in neural networks

Solution

- Gradient descent.
- Newton's method.
- Conjugate gradient
- Quasi-newton method
- Back propagation.

Scanned with AnyScanner

Business Understanding

You work for a consumer finance company which specialises in lending various types of loans to urban customers. When the company receives a loan application, the company has to make a decision for loan approval based on the applicant’s profile. Two types of risks are associated with the bank’s decision:

If the applicant is likely to repay the loan, then not approving the loan results in a loss of business to the company

If the applicant is not likely to repay the loan, i.e. he/she is likely to default, then approving the loan may lead to a financial loss for the company

The dataset given contains the information about past loan applicants and whether they ‘defaulted’ or not. The aim is to identify patterns which indicate if a person is likely to default, which may be used for taking actions such as denying the loan, reducing the amount of loan, lending (to risky applicants) at a higher interest rate, etc.

In this case study, you will use EDA to understand how consumer attributes and loan attributes influence the tendency of default.

image.png

When a person applies for a loan, there are two types of decisions that could be taken by the company:

1. Loan accepted: If the company approves the loan, there are 3 possible scenarios described below:

   a. Fully paid: Applicant has fully paid the loan (the principal and the interest rate)

   b. Current: Applicant is in the process of paying the instalments, i.e. the tenure of the loan is not yet completed. These candidates are not labelled as 'defaulted'.

   c. Charged-off: Applicant has not paid the instalments in due time for a long period of time, i.e. he/she has defaulted on the loan

2. Loan rejected: The company had rejected the loan (because the candidate does not meet their requirements etc.). Since the loan was rejected, there is no transactional history of those applicants with the company and so this data is not available with the company (and thus in this dataset)

Case Study Objectives via EDA

• Determine the factors and attributes that contribute to Bad Loans.

- Determine grouped factors that contribute to Bad Loans.
- Analyse both Bad Loan Average and Number of Bad Loans.

```
pip install plotly
```

```
Requirement already satisfied: plotly in /usr/local/lib/python3.10/dist-packages (5.15.0)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from plotly) (8.2.3)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from plotly) (23.1)
```

```
## Ignore Warings
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)
pd.options.display.float_format = '{:.2f}'.format

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from datetime import datetime
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```
df = pd.read_csv("/content/loan.csv") #updating path name here --after data mounting to drive
type(df)
```

```
pandas.core.frame.DataFrame
```

DATA UNDERSTANDING

```
# Understanding what each column represents , and also what each row/data point represents
df.head()
```

	id	member_id	loan_amnt	funded_amnt	funded_amnt_inv	term	int_rate	installment	grade	sub_grade	emp_
0	1077501	1296599	5000	5000	4975.00	36 months	10.65%	162.87	B	B2	
1	1077430	1314167	2500	2500	2500.00	60 months	15.27%	59.83	C	C4	
2	1077175	1313524	2400	2400	2400.00	36 months	15.96%	84.33	C	C5	
3	1076863	1277178	10000	10000	10000.00	36 months	13.49%	339.31	C	C1	RESOU B
4	1075358	1311748	3000	3000	3000.00	60 months	12.69%	67.79	B	B5	Uni M

```
#rows x columns
df.shape
```

```
(39717, 111)
```

```
#data type of each variable
#categorical: object - strings
#numeric: float64, int64 - numbers
#time based variables : timestamp- datetime data type
#is the data type of the variable consistent with its variable description?
df.dtypes
```

```
id                              int64
member_id                       int64
loan_amnt                       int64
funded_amnt                     int64
funded_amnt_inv               float64
term                           object
int_rate                       object
installment                   float64
grade                          object
sub_grade                      object
emp_title                      object
emp_length                     object
home_ownership                 object
annual_inc                    float64
verification_status            object
issue_d                        object
loan_status                    object
pymnt_plan                     object
url                            object
desc                           object
purpose                        object
title                          object
zip_code                       object
addr_state                     object
dti                           float64
delinq_2yrs                     int64
earliest_cr_line               object
inq_last_6mths                  int64
mths_since_last_delinq        float64
mths_since_last_record        float64
open_acc                        int64
pub_rec                         int64
revol_bal                       int64
revol_util                     object
total_acc                       int64
initial_list_status            object
out_prncp                     float64
out_prncp_inv                 float64
total_pymnt                   float64
total_pymnt_inv               float64
total_rec_prncp               float64
total_rec_int                 float64
total_rec_late_fee            float64
recoveries                    float64
collection_recovery_fee       float64
last_pymnt_d                   object
last_pymnt_amnt               float64
next_pymnt_d                   object
last_credit_pull_d             object
collections_12_mths_ex_med    float64
```

```
mths_since_last_major_derog       float64
policy_code                         int64
application_type                   object
annual_inc_joint                  float64
dti_joint                         float64
verification_status_joint         float64
acc_now_delinq                      int64
tot_coll_amt                      float64
```

```
# Display columns missing values %
round(100*(df.isnull().sum()/len(df.index)),2)
```

```
id                              0.00
member_id                       0.00
loan_amnt                       0.00
funded_amnt                     0.00
funded_amnt_inv                 0.00
term                            0.00
int_rate                        0.00
installment                     0.00
grade                           0.00
sub_grade                       0.00
emp_title                       6.19
emp_length                      2.71
home_ownership                  0.00
annual_inc                      0.00
verification_status             0.00
issue_d                         0.00
loan_status                     0.00
pymnt_plan                      0.00
url                             0.00
desc                           32.58
purpose                         0.00
title                           0.03
zip_code                        0.00
addr_state                      0.00
dti                             0.00
delinq_2yrs                     0.00
earliest_cr_line                0.00
inq_last_6mths                  0.00
mths_since_last_delinq         64.66
mths_since_last_record         92.99
open_acc                        0.00
pub_rec                         0.00
revol_bal                       0.00
revol_util                      0.13
total_acc                       0.00
initial_list_status             0.00
out_prncp                       0.00
```

```
out_prncp_inv                   0.00
total_pymnt                     0.00
total_pymnt_inv                 0.00
total_rec_prncp                 0.00
total_rec_int                   0.00
total_rec_late_fee              0.00
recoveries                      0.00
collection_recovery_fee         0.00
last_pymnt_d                    0.18
last_pymnt_amnt                 0.00
next_pymnt_d                   97.13
last_credit_pull_d              0.01
collections_12_mths_ex_med      0.14
mths_since_last_major_derog   100.00
policy_code                     0.00
application_type                0.00
annual_inc_joint              100.00
dti_joint                     100.00
verification_status_joint     100.00
acc_now_delinq                  0.00
```

DATA CLEANING

```
print(df.id.nunique())
print(df.member_id.nunique())
# Both columns have same number of unique values; Drop one of the columns
df = df.drop(['member_id'], axis=1)
```

```
39717
39717
```

```
# Drop columns having all nans
df = df.dropna(how='all', axis=1)

# Drop columns - mths_since_last_delinq,mths_since_last_record,next_pymnt_d having more than 30% nans
df = df.drop(['mths_since_last_delinq', 'mths_since_last_record', 'next_pymnt_d'], axis=1)

# Drop pymnt_plan column; only value in that column is n
print(df['pymnt_plan'].value_counts(),"\n")
df = df.drop(['pymnt_plan'], axis=1)

# Drop collections_12_mths_ex_med column; only values in that column are 0 and NA
print(df['collections_12_mths_ex_med'].value_counts(),"\n")
df = df.drop(['collections_12_mths_ex_med'], axis=1)
```

```
# Drop policy_code column; only value in that column is 1
print(df['policy_code'].value_counts(),"\n")
df = df.drop(['policy_code'], axis=1)

# Drop application_type column; only value in that column is INDIVIDUAL
print(df['application_type'].value_counts(),"\n")
df = df.drop(['application_type'], axis=1)

# Drop acc_now_delinq column; only value in that column is 0
print(df['acc_now_delinq'].value_counts(),"\n")
df = df.drop(['acc_now_delinq'], axis=1)

# Drop corresponding delinq_amnt column; only value in that column is also 0
print(df['delinq_amnt'].value_counts(),"\n")
df = df.drop(['delinq_amnt'], axis=1)

# Drop tax_liens column; only values in that column are 0 and NA
print(df['tax_liens'].value_counts(),"\n")
df = df.drop(['tax_liens'], axis=1)

# Drop chargeoff_within_12_mths column; only values in that column is 0
print(df['chargeoff_within_12_mths'].value_counts(),"\n")
df = df.drop(['chargeoff_within_12_mths'], axis=1)

# Drop initial_list_status column; only value in that column is F
print(df['initial_list_status'].value_counts(),"\n")
df = df.drop(['initial_list_status'], axis=1)
```

```
n    39717
Name: pymnt_plan, dtype: int64

0.00    39661
Name: collections_12_mths_ex_med, dtype: int64

1    39717
Name: policy_code, dtype: int64

INDIVIDUAL    39717
Name: application_type, dtype: int64

0    39717
Name: acc_now_delinq, dtype: int64

0    39717
Name: delinq_amnt, dtype: int64
```

```
0.00    39678
Name: tax_liens, dtype: int64

0.00    39661
Name: chargeoff_within_12_mths, dtype: int64

f    39717
Name: initial_list_status, dtype: int64
```

```
# Drop url column; it does not hold significance in analysis
df = df.drop(['url'], axis=1)

# Drop desc column; Instead of desc, we can use purpose for analysis
df = df.drop(['desc'], axis=1)

# Drop title and emp_title column; Not significant for analysis
df = df.drop(['emp_title','title'], axis=1)
```

Derive target column which is loan_status

```
# Remove rows which have loan status as current as our target column values are Fully Paid and Charged Off
df = df.loc[(df.loan_status != 'Current'), :]

# Replace loan_status column values of Full Paid with 0 and Charged Off (or Defaulters) with 1
df.loan_status[df.loan_status == 'Fully Paid'] = 0
df.loan_status[df.loan_status == 'Charged Off'] = 1

df['loan_status'] = df['loan_status'].astype("int64")
```

```
# There are 3 values in home_ownership with value of NONE; We can impute NONE to OTHER
print(df['home_ownership'].value_counts(),"\n")
df['home_ownership'] = df['home_ownership'].replace('NONE', 'OTHER')

# Remove trailing xx in zip code column
print(df['zip_code'].head())
df['zip_code'] = df[['zip_code']].applymap(lambda x:str(x).rstrip('xx'))

# Remove trailing months from term column and rename column as term_in_months
print()
print(df['term'].head())
df['term'] = df[['term']].applymap(lambda x:str(x).rstrip('months'))
```

```
df['term'] = df['term'].astype('int64')
df.rename(columns={'term':'term_in_months'},inplace=True)

# Remove trailing % sign in int_rate and rename int_rate column as int_rate_percent
print()
print(df['int_rate'].head())
df['int_rate'] = df[['int_rate']].applymap(lambda x:str(x).rstrip('%'))
df['int_rate'] = df['int_rate'].astype('float64')
df.rename(columns={'int_rate':'int_rate_percent'},inplace=True)

# Remove trailing % sign in revol_util and rename column as revol_util_percent
print()
print(df['revol_util'].head())
df['revol_util'] = df[['revol_util']].applymap(lambda x:str(x).rstrip('%'))
df['revol_util'] = df['revol_util'].astype('float64')
df.rename(columns={'revol_util':'revol_util_percent'},inplace=True)
```

```
RENT        18480
MORTGAGE    17021
OWN          2975
OTHER          98
NONE            3
Name: home_ownership, dtype: int64

0    860xx
1    309xx
2    606xx
3    917xx
5    852xx
Name: zip_code, dtype: object

0     36 months
1     60 months
2     36 months
3     36 months
5     36 months
Name: term, dtype: object

0    10.65%
1    15.27%
2    15.96%
3    13.49%
5     7.90%
Name: int_rate, dtype: object

0    83.70%
1     9.40%
2    98.50%
```

```
3       21%
5    28.30%
Name: revol_util, dtype: object
```

```
# Change funded_amnt_inv type to int64
df['funded_amnt_inv'] = df['funded_amnt_inv'].astype("int64")

# Change grade and sub grades type to category
df['grade'] = df['grade'].astype("category")
df['sub_grade'] = df['sub_grade'].astype("category")
```

```
round(100*(df.isnull().sum()/len(df.index)),2)
```

```
id                        0.00
loan_amnt                 0.00
funded_amnt               0.00
funded_amnt_inv           0.00
term_in_months            0.00
int_rate_percent          0.00
installment               0.00
grade                     0.00
sub_grade                 0.00
emp_length                2.68
home_ownership            0.00
annual_inc                0.00
verification_status       0.00
issue_d                   0.00
loan_status               0.00
purpose                   0.00
zip_code                  0.00
addr_state                0.00
dti                       0.00
delinq_2yrs               0.00
earliest_cr_line          0.00
inq_last_6mths            0.00
open_acc                  0.00
pub_rec                   0.00
revol_bal                 0.00
revol_util_percent        0.13
total_acc                 0.00
out_prncp                 0.00
out_prncp_inv             0.00
total_pymnt               0.00
total_pymnt_inv           0.00
total_rec_prncp           0.00
total_rec_int             0.00
total_rec_late_fee        0.00
recoveries                0.00
```

```
collection_recovery_fee   0.00
last_pymnt_d              0.18
last_pymnt_amnt           0.00
last_credit_pull_d        0.01
pub_rec_bankruptcies      1.81
dtype: float64
```

```
df.shape
```

```
(38577, 40)
```

```
df.head(10)
```

	id	loan_amnt	funded_amnt	funded_amnt_inv	term_in_months	int_rate_percent	installment	grade	sub_grade
0	1077501	5000	5000	4975	36	10.65	162.87	B	B2
1	1077430	2500	2500	2500	60	15.27	59.83	C	C4
2	1077175	2400	2400	2400	36	15.96	84.33	C	C5
3	1076863	10000	10000	10000	36	13.49	339.31	C	C1
5	1075269	5000	5000	5000	36	7.90	156.46	A	A4
6	1069639	7000	7000	7000	60	15.96	170.08	C	C5
7	1072053	3000	3000	3000	36	18.64	109.43	E	E1
8	1071795	5600	5600	5600	60	21.28	152.39	F	F2
9	1071570	5375	5375	5350	60	12.69	121.45	B	B5
10	1070078	6500	6500	6500	60	14.65	153.45	C	C3

DERIVED METRICS

pub_rec_bankruptcies_b and delinq_2yrs_b

```
# Derive pub_rec_bankruptcies_b with values as NO if pub_rec_bankruptcies is 0 else YES
print(df['pub_rec_bankruptcies'].value_counts(),"\n")
df["pub_rec_bankruptcies"].fillna(0, inplace = True)
df['pub_rec_bankruptcies_b'] = df[['pub_rec_bankruptcies']].applymap(lambda x : 'NO' if x == 0 else 'YES')
```

```
df['pub_rec_bankruptcies_b'].describe()

# Derive delinq_2yrs_b with values as NO if delinq_2yrs is 0 else YES
print(df['delinq_2yrs'].value_counts(),"\n")
df['delinq_2yrs_b'] = df[['delinq_2yrs']].applymap(lambda x : 'NO' if x == 0 else 'YES')
```

```
0.00    36238
1.00     1637
2.00        5
Name: pub_rec_bankruptcies, dtype: int64

0     34386
1      3207
2       673
3       212
4        60
5        21
6        10
7         4
8         2
9         1
11        1
Name: delinq_2yrs, dtype: int64
```

Emp length

Bucketing emp length and standardizing the data. Assumption made, candidates with less than 1 year of exp also bucketed in 1 year.

```
df['emp_length'] = df[['emp_length']].applymap(lambda x:str(x).rstrip("year").rstrip("years").lstrip("<").strip())
df.loc[(df['emp_length'] == "nan"),['emp_length']] = "0"
```

Zip_Code

Zip Codes are orderly data , they are ordered numerically from east to west.Hence grouping them to infer the defaulter percentages for each group.

```
zip_mapping = {
  0: "1-100",
  1: "101-200",
  2: "201-300",
  3: "301-400",
  4: "401-500",
  5: "501-600",
  6: "601-700",
  7: "701-800",
```

```
    8: "801-900",
    9: "900-901"
}
df['zip_code'] = df['zip_code'].astype('int64')
def get_zip_group(zip_code):
    return zip_mapping[int(zip_code/100)]

df['zip_code_group'] = df[['zip_code']].applymap(lambda x : get_zip_group(x))
```

issue_d_year

issue_d derives issue_d_year

earliest_cr_line derives earliest_cr_line_year

```
dt_series = pd.to_datetime(df.issue_d.str.upper(), format='%b-%y', yearfirst=False)
df['issue_d_year'] = dt_series.dt.year

# We can do same as above for earliest_cr_line
dt_series = pd.to_datetime(df.earliest_cr_line.str.upper(), format='%b-%y', yearfirst=False)
df['earliest_cr_line_year'] = dt_series.dt.year
```

Derive Loan received amount bucket column

```
fq = df['funded_amnt_inv'].quantile(0.25)
sq = df['funded_amnt_inv'].quantile(0.50)
tq = df['funded_amnt_inv'].quantile(0.75)
print(fq, sq, tq)
```

```
5000.0 8733.0 14000.0
```

```
def get_loan_type(loan_amnt):
    if loan_amnt <= fq:
        return 'Small(<=5000)'
    if (loan_amnt > fq) & (loan_amnt <= sq):
        return 'Regular(>5000 & <=8733)'
    elif (loan_amnt > sq) & (loan_amnt <= tq):
        return 'Medium(>8733 & <=14000)'
    else:
        return 'Large(>14000)'
```

```
df['loan_type'] = df[['funded_amnt_inv']].applymap(lambda x : get_loan_type(x))
df['loan_type'].value_counts()
```

```
Small(<=5000)              10701
Medium(>8733 & <=14000)     9694
Large(>14000)               9593
Regular(>5000 & <=8733)     8589
Name: loan_type, dtype: int64
```

Derive annual income buckets based upon IQR of annual income

```
print(df.annual_inc.describe())
fq = df.annual_inc.quantile(0.25)
sq = df.annual_inc.quantile(0.50)
tq = df.annual_inc.quantile(0.75)
def get_annual_inc_type(income):
    if income <= fq:
        return 'Very Low(<=40K)'
    if (income > fq) & (income <= sq):
        return 'Low(>40K & <=58.86K)'
    elif (income > sq) & (income <= tq):
        return 'Medium(>58.86K & <=82K)'
    else:
        return 'High(>82K)'

df['annual_inc_type'] = df[['annual_inc']].applymap(lambda x : get_annual_inc_type(x))
df['annual_inc_type'].head(10)
df.head(10)
```

```
count     38577.00
mean      68777.97
std       64218.68
min        4000.00
25%       40000.00
50%       58868.00
75%       82000.00
max     6000000.00
Name: annual_inc, dtype: float64
```

	id	loan_amnt	funded_amnt	funded_amnt_inv	term_in_months	int_rate_percent	installment	grade	sub_grade
0	1077501	5000	5000	4975	36	10.65	162.87	B	B2
1	1077430	2500	2500	2500	60	15.27	59.83	C	C4
2	1077175	2400	2400	2400	36	15.96	84.33	C	C5
3	1076863	10000	10000	10000	36	13.49	339.31	C	C1

DATA SUMMARIZATION

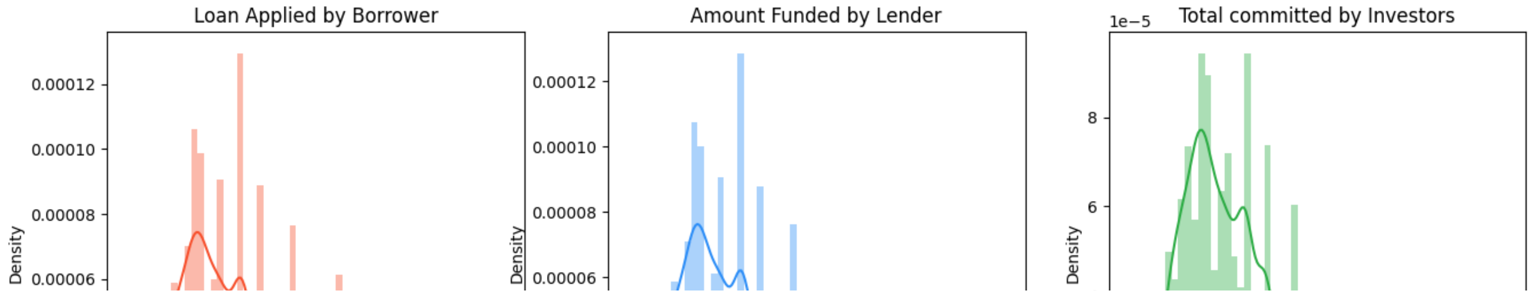
Problem Statement:

Understand what amount was mostly issued to borrowers

8	1071795	5600	5600	5600	60	21.28	152.39	F	F2

```
ig, ax = plt.subplots(1, 3, figsize=(16,5))

loan_amount = df["loan_amnt"].values
funded_amount = df["funded_amnt"].values
investor_funds = df["funded_amnt_inv"].values

sns.distplot(loan_amount, ax=ax[0], color="#F7522F")
ax[0].set_title("Loan Applied by Borrower", fontsize=12)
sns.distplot(funded_amount, ax=ax[1], color="#2F8FF7")
ax[1].set_title("Amount Funded by Lender", fontsize=12)
sns.distplot(investor_funds, ax=ax[2], color="#2EAD46")
ax[2].set_title("Total committed by Investors", fontsize=12)
```

```
Text(0.5, 1.0, 'Total committed by Investors')
```

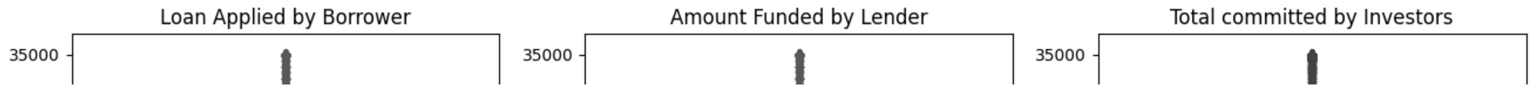


Summary:

Most of the loans issued were in the range of 5,000 to 14,000. The loans applied by potential borrowers, the amount issued to the borrowers and the amount funded by investors are similarly distributed, meaning that it is most likely that qualified borrowers are going to get the loan they had applied for.

```
ig, ax = plt.subplots(1, 3, figsize=(16,5))

loan_amount = df["loan_amnt"].values
funded_amount = df["funded_amnt"].values
investor_funds = df["funded_amnt_inv"].values

sns.boxplot(loan_amount, ax=ax[0], color="#F7522F")
ax[0].set_title("Loan Applied by Borrower", fontsize=12)
sns.boxplot(funded_amount, ax=ax[1], color="#2F8FF7")
ax[1].set_title("Amount Funded by Lender", fontsize=12)
sns.boxplot(investor_funds, ax=ax[2], color="#2EAD46")
ax[2].set_title("Total committed by Investors", fontsize=12)

df['funded_amnt_inv'].describe()
```

```
count   38577.00
mean    10222.39
std      7022.70
min         0.00
25%      5000.00
50%      8733.00
75%     14000.00
max     35000.00
Name: funded_amnt_inv, dtype: float64
```



Problem Statement:

Percentage of good loans and bad loans


```
print(df["loan_status"].value_counts())

# Slice data frame into good and bad loan status
df_good = df.loc[(df.loan_status == 0), :]
df_bad  = df.loc[(df.loan_status == 1), :]

plt.figure(figsize=(12, 6))
colors = ["#3791D7", "#D72626"]
labels = "Good Loans", "Bad Loans"
#df["loan_status"].value_counts().plot.pie(explode=[0,0.2], autopct='%1.2f%%', shadow=True, colors=colors, labels=labels, fontsize=12, startangle=70
plt.ylabel('% of Loan Status', fontsize=14)
plt.show()
```

```
0    32950
1     5627
Name: loan_status, dtype: int64
```

Summary:

Fully paid or good loans consist 85.41% of total loans in the cleaned data frame.

Charged off or bad loans consist 14.59% of total loans in the cleaned data frame.

Analysis of Loan Term vs Bad Loans

```
print(df.groupby("term_in_months").loan_status.count())
print(df.groupby("term_in_months").loan_status.mean())
plt.figure(figsize=(14, 8))
plt.subplot(2, 2, 1)
ax1 = sns.countplot(x="term_in_months", data = df)
ax1.set(xlabel='Term in months', ylabel="Count of Loans")
# subplot 2
plt.subplot(2, 2, 2)
ax2 = sns.barplot(x="term_in_months", y = "loan_status" , data = df)
ax2.set(xlabel='Term in months', ylabel='Average Bad Loan %')
plt.show()
```

```
term_in_months
36    29096
60     9481
Name: loan_status, dtype: int64
term_in_months
36   0.11
60   0.25
```

Summary:

If loan term is 60 months then it has 25% average who default on loans as compared to loan term of 36 months which has 11% average who default on loans.

Also from the count plot, one can infer that number of entries of 36 months is approx 3 times number of entries of 60 months. Still it has 11% average who default on loans.

Analysis of grouped variables - Loan Issued Year and Term vs Bad Loans


```
print(df.groupby(["issue_d_year", "term_in_months"]).loan_status.count())
print(df.groupby(["issue_d_year", "term_in_months"]).loan_status.mean())
plt.figure(figsize=(14, 8))
ax = sns.barplot(x="issue_d_year", y = "loan_status" , hue="term_in_months", data = df)
ax.set(xlabel='Loan Issue Year', ylabel='Average Bad Loan %')
plt.show()
```

```
issue_d_year  term_in_months
2007          36                  251
2008          36                 1562
2009          36                 4716
2010          36                 8466
              60                 3066
2011          36                14101
              60                 6415
Name: loan_status, dtype: int64
issue_d_year  term_in_months
2007          36               0.18
2008          36               0.16
2009          36               0.13
2010          36               0.10
              60               0.21
2011          36               0.11
              60               0.27
Name: loan_status, dtype: float64
```

Summary:

Between 2007-2009, there were only 36 months term loans.

2010-11 included 60 months term loans.

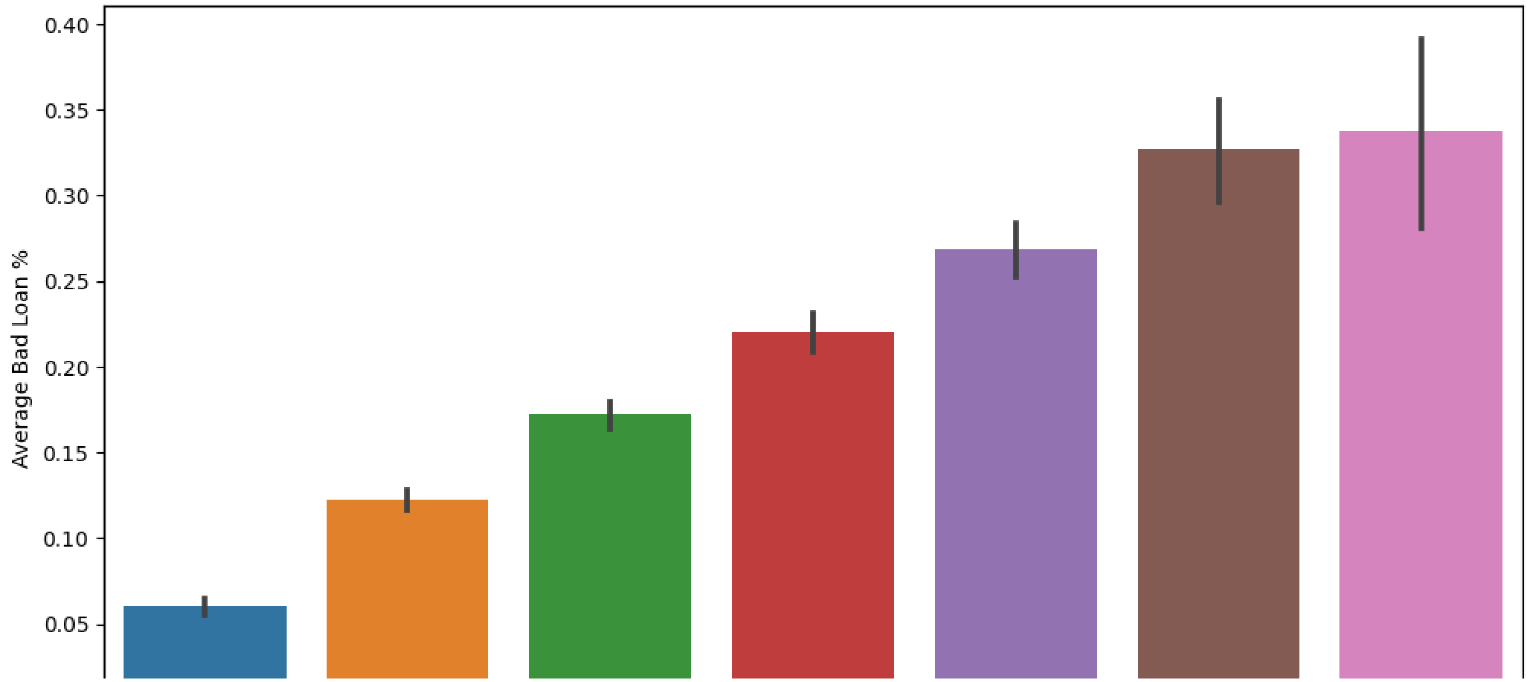
And in last 2 years of the data set, there are more default loans among the 60 months term loans.

27% of 60 months term loans defaulted in year 2011 and 21% of 60 months term loans defaulted in year 2010.

Analysis of Grade vs Bad Loans

```
print(df.groupby("grade").loan_status.mean())
plt.figure(figsize=(12, 6))
ax = sns.barplot(x="grade", y = "loan_status" , data = df)
ax.set(xlabel="Grade", ylabel='Average Bad Loan %')
plt.show()
```

```
grade
A    0.06
B    0.12
C    0.17
D    0.22
E    0.27
F    0.33
G    0.34
Name: loan_status, dtype: float64
```



Summary:

Grade G has highest 34% average who default on loans.

Grade F has second highest 33% average who default on loans.

Between Grades A to G - bad loan % increments in a consistent manner.

Analysis of number of loans and bad loans per grade in a table

```
grade = df.groupby('grade', as_index=False).agg({"loan_status": "mean"}).sort_values(by=['loan_status'], ascending=False)

def get_bad_loan_count_g(grade):
```

```
    return len(df[((df['grade']==grade) & (df['loan_status']==1))].index)

def get_good_loan_count_g(grade):
    return len(df[((df['grade']==grade) & (df['loan_status']==0))].index)

grade['Good Loan Count'] = grade[['grade']].applymap(lambda x:get_good_loan_count_g(x))
grade['Bad Loan Count'] = grade[['grade']].applymap(lambda x:get_bad_loan_count_g(x))
grade.rename(columns={'grade':'Grade','loan_status':'Bad Loan %'}, inplace=True)
grade.set_index('Grade', inplace=True)

grade.style.background_gradient('coolwarm')
```

Grade	Bad Loan %	Good Loan Count	Bad Loan Count
G	0.337793	198	101
F	0.326844	657	319
E	0.268494	1948	715
D	0.219862	3967	1118
C	0.171943	6487	1347
B	0.122056	10250	1425
A	0.059930	9443	602

Analysis of Sub Grade vs Bad Loans

```
df_grade = df.loc[df['grade'].isin(['A','B','C','D'])]
print(df_grade.groupby(["grade", "sub_grade"]).loan_status.mean())
plt.figure(figsize=(12, 6))
ax = sns.barplot(x="sub_grade", y = "loan_status", data = df)
ax.set(xlabel= "Sub Grade", ylabel='Average Bad Loan %')
plt.show()
```

```
grade  sub_grade
A      A1           0.03
       A2           0.05
       A3           0.06
       A4           0.06
       A5           0.08
       B1            NaN
       B2            NaN
       B3            NaN
       B4            NaN
       B5            NaN
       C1            NaN
       C2            NaN
       C3            NaN
       C4            NaN
       C5            NaN
       D1            NaN
       D2            NaN
       D3            NaN
       D4            NaN
       D5            NaN
       E1            NaN
       E2            NaN
       E3            NaN
       E4            NaN
       E5            NaN
       F1            NaN
       F2            NaN
       F3            NaN
       F4            NaN
       F5            NaN
       G1            NaN
       G2            NaN
       G3            NaN
       G4            NaN
       G5            NaN
B      A1            NaN
       A2            NaN
       A3            NaN
       A4            NaN
       A5            NaN
       B1           0.10
       B2           0.11
       B3           0.12
       B4           0.14
       B5           0.14
       C1            NaN
       C2            NaN
       C3            NaN
       C4            NaN
```

Summary:

Among the subgrades too of lower Grades A to D - bad loan % increments in a consistent manner between the sub grades in each grade.

Grades E to G already determined as higher risks.

Analysis of Home Ownership vs Bad Loans

```
print(df.groupby("home_ownership").loan_status.mean())
plt.figure(figsize=(12, 6))
ax = sns.barplot(x="home_ownership", y = "loan_status" , data = df)
ax.set(xlabel='Home Ownership', ylabel='Average Bad Loan %')
plt.show()
```

Summary:

There is less variance of bad loan % by the home ownership category.

All categories of home ownership (except others) have almost the same average of default loans.

Analysis of Annual Income vs Bad Loans

```
print(df.groupby("annual_inc_type").loan_status.mean())
plt.figure(figsize=(14, 6))
ax = sns.barplot(x="annual_inc_type", y = "loan_status", data = df)
ax.set(xlabel='Annual Income', ylabel='Average Bad Loan %')
plt.show()
```

Summary:

Those who have high income are less likely to default on loans. Income>82000 have 11% average of bad loans.

Whereas those who have very low income are more likely to default on loans. Income <=40000 have 18% average of bad loans.

```
Medium(>58.86K & <=82K)   0.14
```

Analysis of Income Verification Status and Annual Income vs Bad Loans

```
print("\nBad Loan Average grouped by verification status ->")
print(df.groupby("verification_status").loan_status.mean())
print("\nBad Loan Average grouped by annual income type and verification status ->")
print(df.groupby(["annual_inc_type","verification_status"]).loan_status.mean())
print("\nLoan Counts grouped by annual income type and verification status ->")
print(df.groupby(["annual_inc_type","verification_status"]).loan_status.count())
plt.figure(figsize=(18, 12))
plt.subplot(2, 2, 1)
ax1 = sns.barplot(y="verification_status", x = "loan_status", data = df)
ax1.set(ylabel='Annual Income Verification Status', xlabel='Average Bad Loan %')
# subplot 2
plt.subplot(2, 2, 3)
ax2 = sns.barplot(y="annual_inc_type", x = "loan_status", hue="verification_status", data = df)
ax2.set(ylabel='Annual Income', xlabel='Average Bad Loan %')
plt.show()
```

```
Bad Loan Average grouped by verification status ->
verification_status
Not Verified      0.13
Source Verified   0.15
Verified          0.17
Name: loan_status, dtype: float64

Bad Loan Average grouped by annual income type and verification status ->
annual_inc_type          verification_status
High(>82K)               Not Verified          0.09
                         Source Verified       0.11
                         Verified              0.13
Low(>40K & <=58.86K)     Not Verified          0.13
                         Source Verified       0.15
                         Verified              0.20
Medium(>58.86K & <=82K)  Not Verified          0.12
                         Source Verified       0.15
                         Verified              0.16
Very Low(<=40K)          Not Verified          0.16
                         Source Verified       0.18
                         Verified              0.23
Name: loan_status, dtype: float64

Loan Counts grouped by annual income type and verification status ->
annual_inc_type          verification_status
High(>82K)               Not Verified           2965
                         Source Verified        2216
                         Verified               4407
Low(>40K & <=58.86K)     Not Verified           4577
                         Source Verified        2490
                         Verified               2524
Medium(>58.86K & <=82K)  Not Verified           4069
       F5          NaN   Source Verified        2293
```

Summary:

Those who have annual income as not verified have 13% average of bad loans whereas those who have annual income as verified have 17% average of bad loans.

Also in each annual income category, the impact of income verification status seems strange. Those who have annual income verified or source verified have higher bad loan average than those who have annual income as not verified.

Analysis of Purpose of Loan vs Bad Loans

```
print(df.groupby("purpose").loan_status.mean().sort_values(ascending=False))
plt.figure(figsize=(14, 6))
ax = sns.barplot(y="purpose", x = "loan_status" , data = df)
ax.set(xlabel='Average Bad Loan %', ylabel="Purpose")
plt.show()
```

Summary:

Those who are approved loan for small business are more likely to default on loans with an average of 27%.

And those who are approved loans for purpose such as major purchase, wedding, car or credit card are least likely to default on loans with an average of around 10 to 11%.

```
purpose = df.groupby('purpose', as_index=False).agg({"loan_status": "mean"}).sort_values(by=['loan_status'], ascending=False)

def get_bad_loan_count_p(purpose):
    return len(df[((df['purpose']==purpose) & (df['loan_status']==1))].index)

def get_good_loan_count_p(purpose):
    return len(df[((df['purpose']==purpose) & (df['loan_status']==0))].index)

purpose['Good Loan Count'] = purpose[['purpose']].applymap(lambda x:get_good_loan_count_p(x))
purpose['Bad Loan Count'] = purpose[['purpose']].applymap(lambda x:get_bad_loan_count_p(x))
purpose.rename(columns={'purpose':'Purpose','loan_status':'Bad Loan %'}, inplace=True)
purpose.set_index('Purpose', inplace=True)

purpose.style.background_gradient('coolwarm')
```

	Bad Loan %	Good Loan Count	Bad Loan Count
Purpose			
small_business	0.270810	1279	475
renewable_energy	0.186275	83	19
educational	0.172308	269	56
other	0.163777	3232	633
house	0.160763	308	59
moving	0.159722	484	92
medical	0.155653	575	106
debt_consolidation	0.153254	15288	2767
vacation	0.141333	322	53
home_improvement	0.120696	2528	347
credit_card	0.107818	4485	542
car	0.106738	1339	160
wedding	0.103672	830	96
major_purchase	0.103256	1928	222

The heat map of the purpose against the bad loan percentage along with counts also reveals that "debt consolidation" is the worst performing purpose in terms of count.

Analysis of Interest Rate vs Bad Loans

```
fig = make_subplots(rows=1, cols=1)
trace = go.Histogram(x=df["int_rate_percent"], y=df["loan_status"], histfunc='avg', nbinsx=10, marker_color='#9D0B9F',
    opacity=0.90)
fig.append_trace(trace,1,1)
fig.update_layout(bargap=0.1,
                  title_text='Average Bad Loan vs Interest Rate',
                  xaxis_title_text='Interest rate %',
                  yaxis_title_text='Average Bad Loan %')
fig.show()
```

Plot reveals that bad loan % increases steadily as the interest rate increases.

Analysis of Public Record Bankruptcies vs Bad Loans

```
print(df.groupby("pub_rec_bankruptcies_b").loan_status.mean().sort_values(ascending=False))
plt.figure(figsize=(14, 6))
ax = sns.barplot(x="pub_rec_bankruptcies_b", y = "loan_status" , data = df)
ax.set(ylabel='Average Bad Loan %', xlabel="Public Record Bankruptcies")
plt.show()
```

Summary:

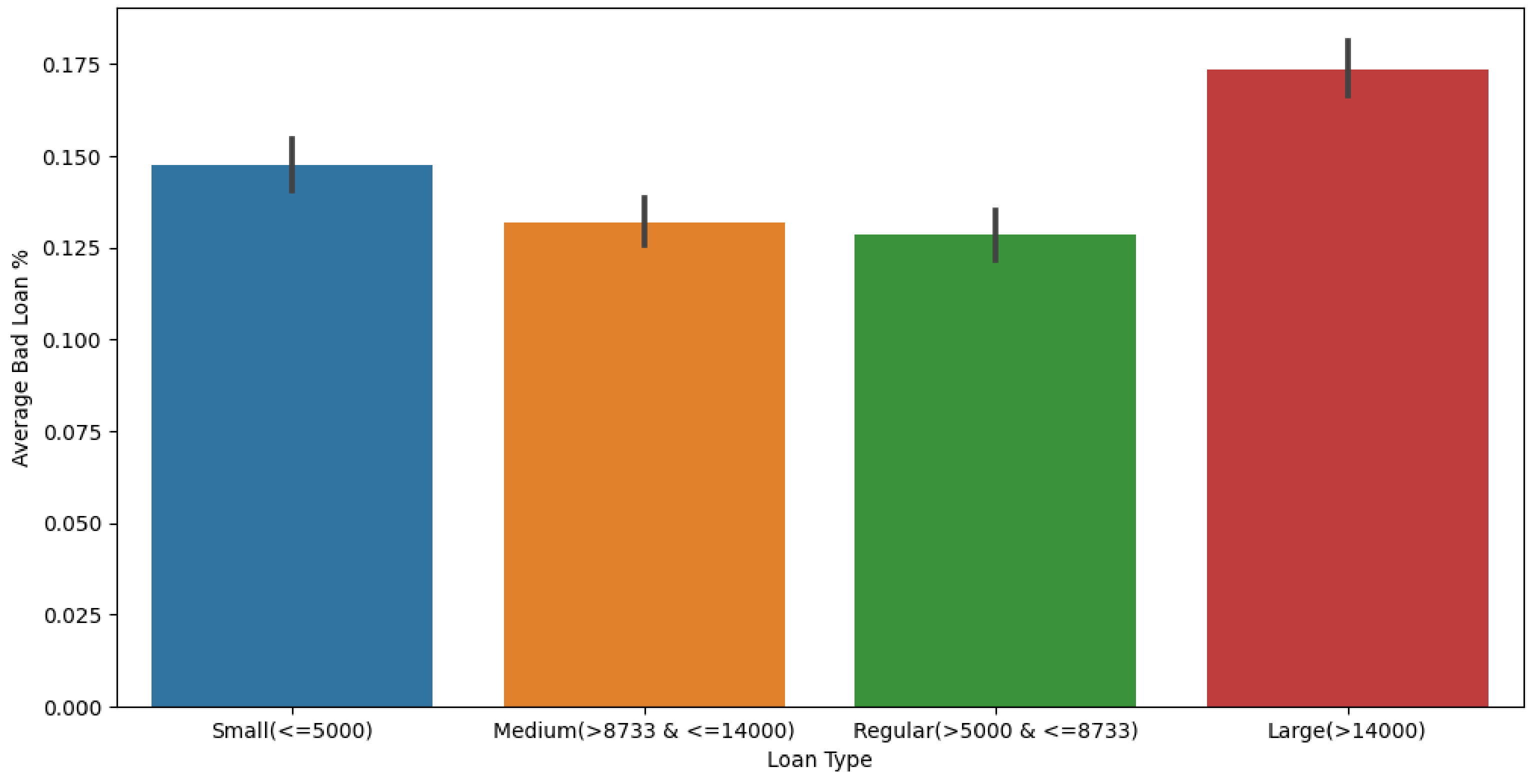
Those who have public record bankrupt record are more likely to default on loans with an average of 22%.

Analysis of Loan Amount vs Bad Loans

```
Name: loan_status, dtype: float64
```

```
print(df.groupby("loan_type").loan_status.mean())
plt.figure(figsize=(12, 6))
ax = sns.barplot(x="loan_type", y = "loan_status" , data = df)
ax.set(xlabel='Loan Type',ylabel='Average Bad Loan %')
plt.show()
```

```
loan_type
Large(>14000)             0.17
Medium(>8733 & <=14000)   0.13
Regular(>5000 & <=8733)   0.13
Small(<=5000)             0.15
Name: loan_status, dtype: float64
```

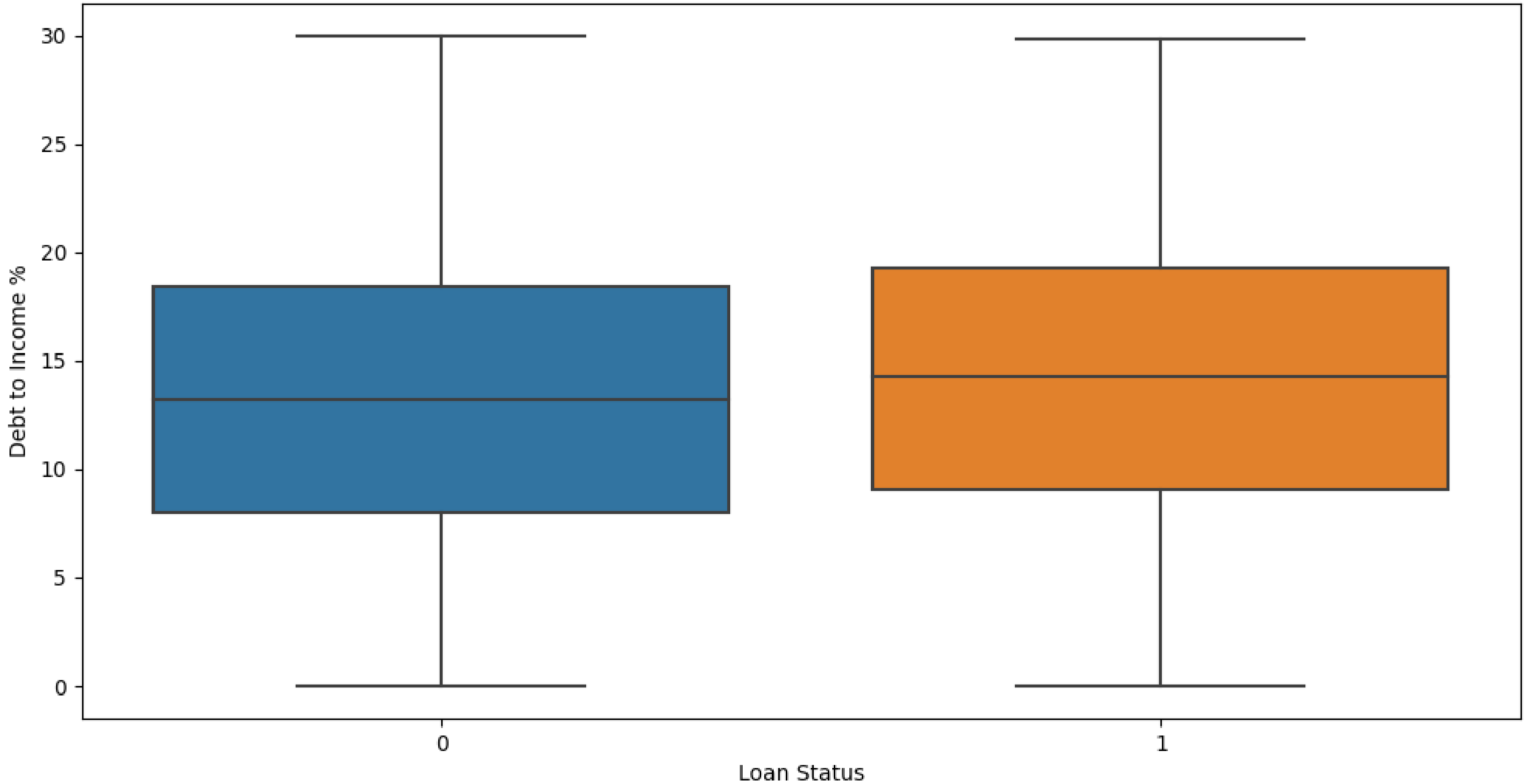


Summary: From the plot we can infer that the variance is very low among different categories hence it alone is unable predict bad loans %.

Analysis of Debt to Income Percentage vs Bad Loans

```
plt.figure(figsize=(12, 6))
ax = sns.boxplot(x="loan_status", y="dti", data=df)
ax.set(xlabel='Loan Status',ylabel='Debt to Income %')
```

```
[Text(0.5, 0, 'Loan Status'), Text(0, 0.5, 'Debt to Income %')]
```



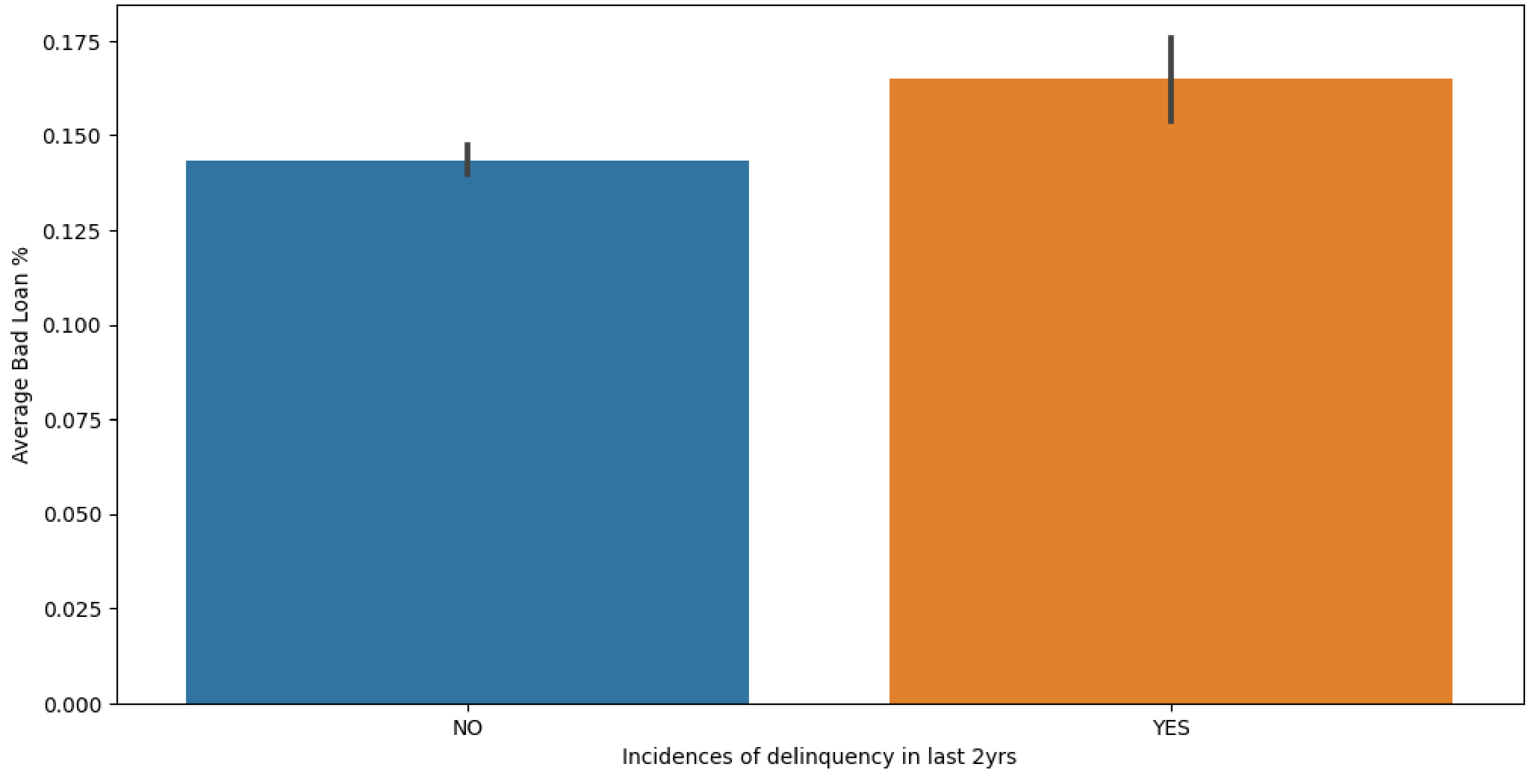
From the box plot we can infer that plots are similar for both good and bad loans, which signifies this field alone does not have much of impact on the bad loan %.

Analysis of delinq in last 2yrs vs Bad Loans

```
print(df.groupby("delinq_2yrs_b").loan_status.count())
print(df.groupby("delinq_2yrs_b").loan_status.mean())
plt.figure(figsize=(12, 6))
ax = sns.barplot(x="delinq_2yrs_b", y = "loan_status" , data = df)
```

```
ax.set(xlabel='Incidences of delinquency in last 2yrs',ylabel='Average Bad Loan %')
plt.show()
```

```
delinq_2yrs_b
NO     34386
YES     4191
Name: loan_status, dtype: int64
delinq_2yrs_b
NO    0.14
YES   0.16
Name: loan_status, dtype: float64
```



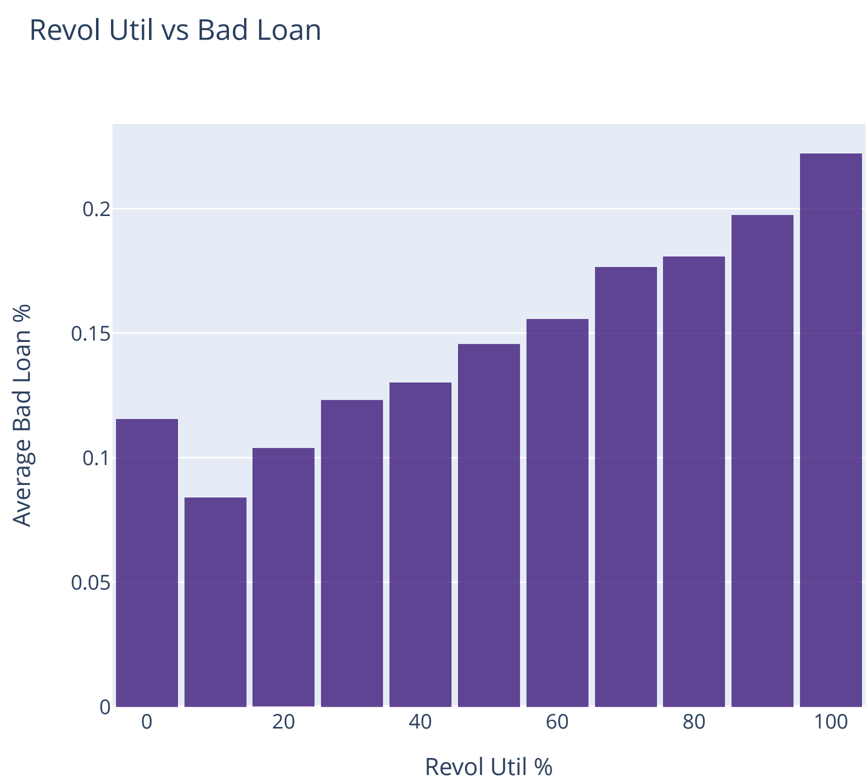
Summary:

Those who have incidences of delinquency in last 2 years default on loans with an average of 16%.

Those who don't have any incidences of delinquency in last 2 years default on loans with an average of 14%.

Analysis of Revolving Utilization vs Bad Loans

```
fig = make_subplots(rows=1, cols=1)
trace = go.Histogram(x=df["revol_util_percent"], y=df["loan_status"], histfunc='avg', nbinsx=10, marker_color='#330C73',
    opacity=0.75)
fig.append_trace(trace,1,1)
fig.update_layout(bargap=0.1,
                  title_text='Revol Util vs Bad Loan',
                  xaxis_title_text='Revol Util %',
                  yaxis_title_text='Average Bad Loan %')
fig.show()
```

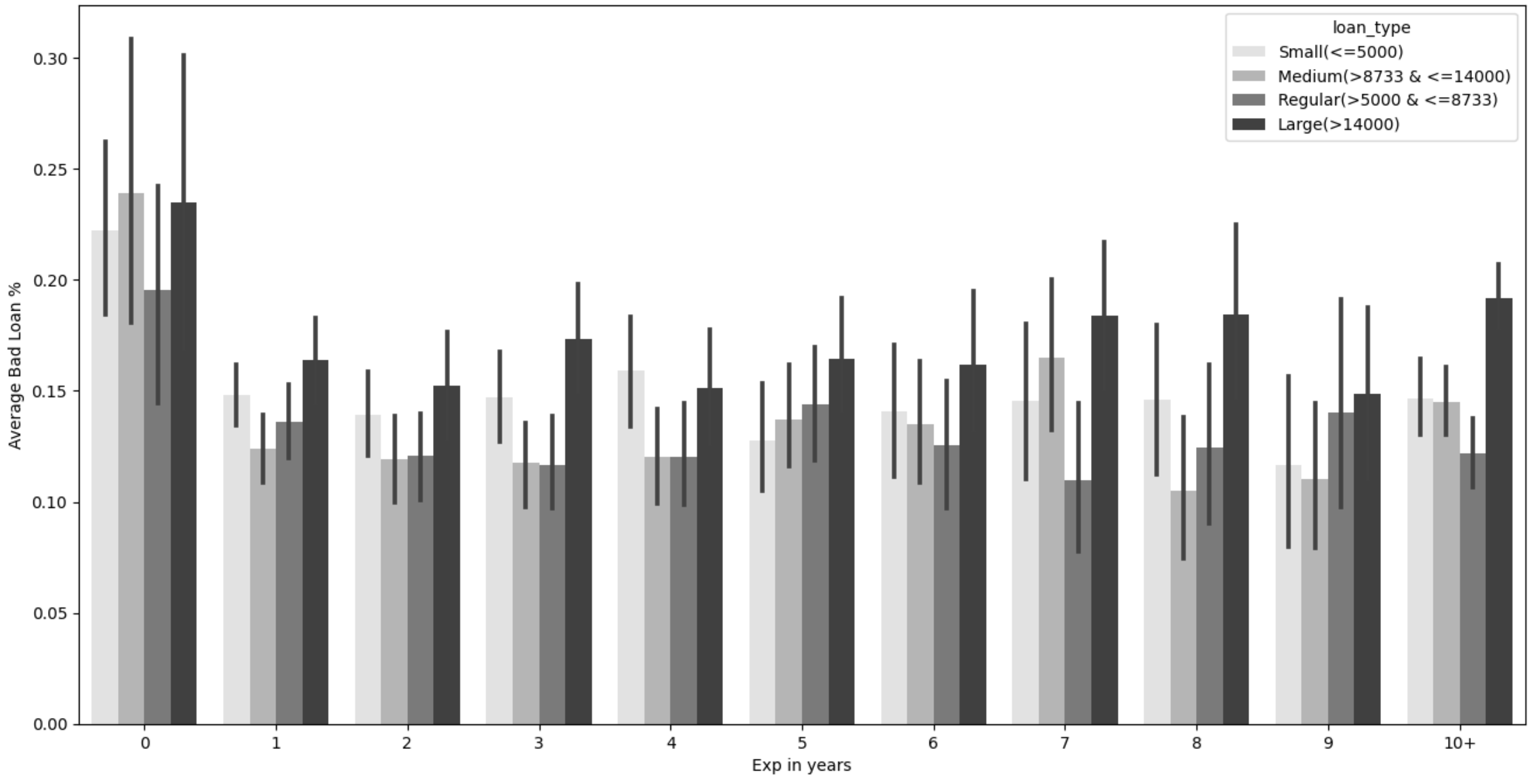


Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.

Plot reveals that bad loan % increases steadily as the Revolving line utilization rate increases.

Analysis of Employee Length vs Bad Loans

```
plt.figure(figsize=(16, 8))
ax = sns.barplot(x="emp_length", palette = sns.color_palette("Greys",4), hue="loan_type", y = "loan_status" , data = df,
                order=["0","1","2","3","4","5","6","7","8","9","10+"])
ax.set(xlabel='Exp in years', ylabel='Average Bad Loan %')
plt.show()
```



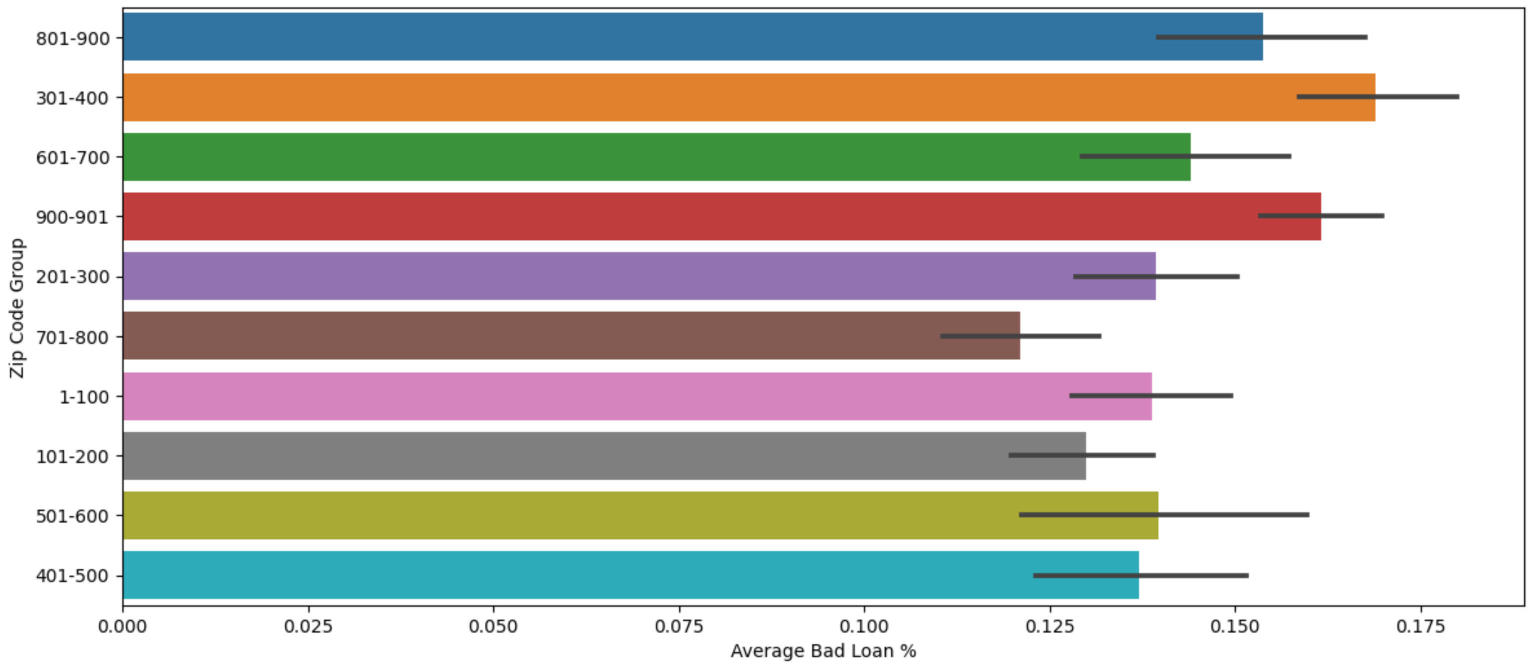
Among the employee length, applicants with 0 years of exp are having higher probability of bad loans that is 25%.

When grouped with loan amount we can also observe large amount loans are performing badly across all employee lengths. Also in 10+ category the variance is significant.

Analysis of Zip Code vs Bad Loans

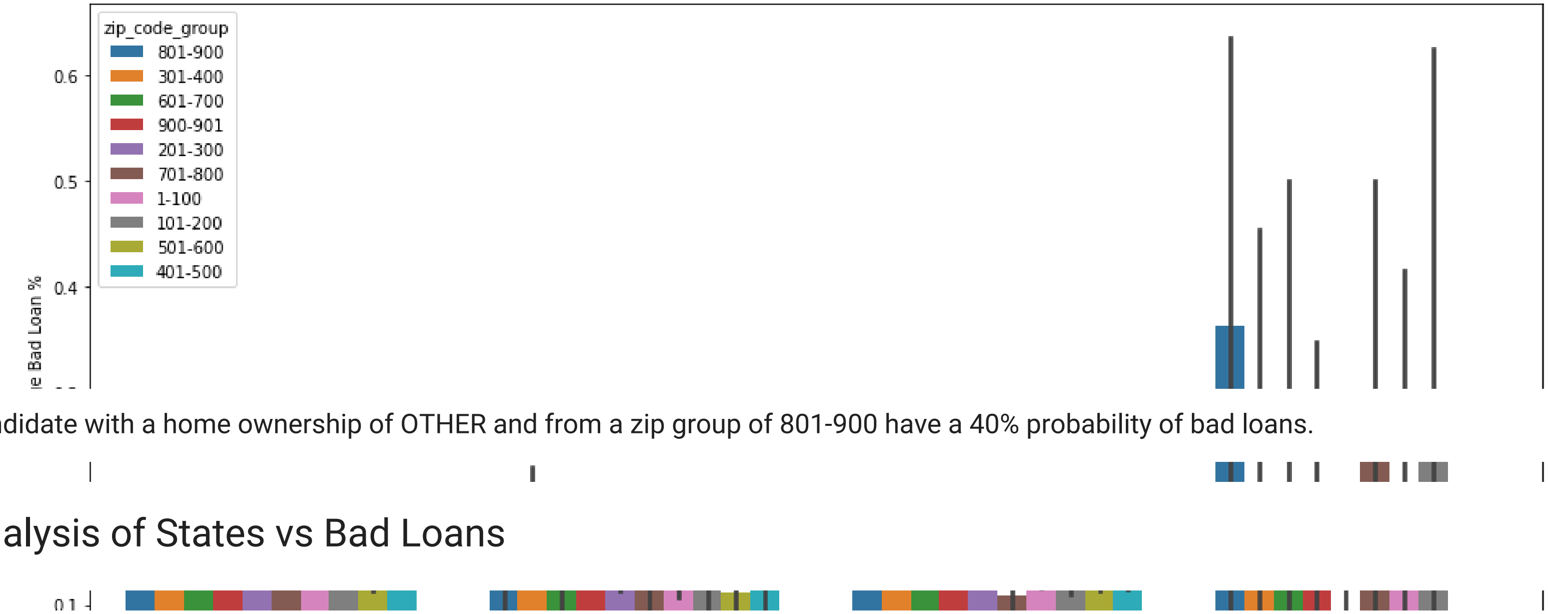
```
plt.figure(figsize=(14, 6))
ax = sns.barplot(y="zip_code_group", x = "loan_status" , data = df)
```

```
ax.set(xlabel='Average Bad Loan %', ylabel="Zip Code Group")
plt.show()
```



Zip Codes 301-400 and 900-901 have higher bad loan % as compared to other zip codes groups.

```
plt.figure(figsize=(16, 8))
ax = sns.barplot(x="home_ownership", y = "loan_status" , hue="zip_code_group", data = df)
ax.set(xlabel='Zip Group', ylabel='Average Bad Loan %')
plt.show()
```



Candidate with a home ownership of OTHER and from a zip group of 801-900 have a 40% probability of bad loans.

Analysis of States vs Bad Loans

```
statewise = df.groupby('addr_state', as_index=False).agg({"loan_status": "mean"}).sort_values(by=['loan_status'], ascending=False)

def get_bad_loan_count(addr_state):
    return len(df[((df['addr_state']==addr_state) & (df['loan_status']==1))].index)

def get_good_loan_count(addr_state):
    return len(df[((df['addr_state']==addr_state) & (df['loan_status']==0))].index)

statewise['Good Loan Count'] = statewise[['addr_state']].applymap(lambda x:get_good_loan_count(x))
statewise['Bad Loan Count'] = statewise[['addr_state']].applymap(lambda x:get_bad_loan_count(x))

statewise.rename(columns={'addr_state':'State','loan_status':'Bad Loan %'}, inplace=True)
statewise.set_index('State', inplace=True)

statewise.style.background_gradient('coolwarm')
```

State	Bad Loan %	Good Loan Count	Bad Loan Count
NE	0.600000	2	3
NV	0.225470	371	108
SD	0.193548	50	12
AK	0.192308	63	15
FL	0.181230	2277	504
MO	0.170149	556	114
HI	0.168675	138	28
ID	0.166667	5	1
NM	0.163934	153	30
OR	0.163218	364	71
CA	0.161894	5824	1125
UT	0.158730	212	40
MD	0.158358	861	162
GA	0.158205	1144	215
NJ	0.155307	1512	278
WA	0.155257	691	127
NC	0.152000	636	114
NH	0.150602	141	25
MI	0.146307	601	103
AZ	0.144876	726	123
KY	0.144695	266	45
SC	0.143791	393	66
WI	0.143182	377	63
OK	0.139373	247	40

Statewise analysis of loan percentage.

From the gradient we can clearly make out 'CA' is one of the worst performing states in terms of number of bad loans.

And 'NE' has the highest bad loan% (60%).

OH	0.131579	1023	155
CT	0.129477	632	94
VA	0.129291	1192	177
RI	0.128866	169	25
CO	0.127937	668	98

```
# import library
import numpy as np
import pandas as pd
from sklearn import preprocessing
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
```

Load data

```
# read dataset
df = pd.read_csv("/heart.csv")
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Age             918 non-null    int64  
 1   Sex             918 non-null    object 
 2   ChestPainType   918 non-null    object 
 3   RestingBP       918 non-null    int64  
 4   Cholesterol     918 non-null    int64  
 5   FastingBS       918 non-null    int64  
 6   RestingECG      918 non-null    object 
 7   MaxHR           918 non-null    int64  
 8   ExerciseAngina  918 non-null    object 
 9   Oldpeak         918 non-null    float64
 10  ST_Slope        918 non-null    object 
 11  HeartDisease    918 non-null    int64  
dtypes: float64(1), int64(6), object(5)
memory usage: 86.2+ KB
```

```
df.describe()
```

	Age	RestingBP	Cholesterol	FastingBS	MaxHR	Oldpeak	HeartDisease
count	918.000000	918.000000	918.000000	918.000000	918.000000	918.000000	918.000000
mean	53.510893	132.396514	198.799564	0.233115	136.809368	0.887364	0.553377
std	9.432617	18.514154	109.384145	0.423046	25.460334	1.066570	0.497414
min	28.000000	0.000000	0.000000	0.000000	60.000000	-2.600000	0.000000
25%	47.000000	120.000000	173.250000	0.000000	120.000000	0.000000	0.000000
50%	54.000000	130.000000	223.000000	0.000000	138.000000	0.600000	1.000000
75%	60.000000	140.000000	267.000000	0.000000	156.000000	1.500000	1.000000
max	77.000000	200.000000	603.000000	1.000000	202.000000	6.200000	1.000000

Data Pre-processing

Handling Missing Value

```
df.isnull().sum()
```

```
Age               0
Sex               0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
HeartDisease      0
dtype: int64
```

no missing values

Handling Duplicated Rows

```
# check shape before drop
df.shape
```

```
(918, 12)
```

```
# drop duplicated row
df = df.drop_duplicates()

# check shape after drop
df.shape
```

```
(918, 12)
```

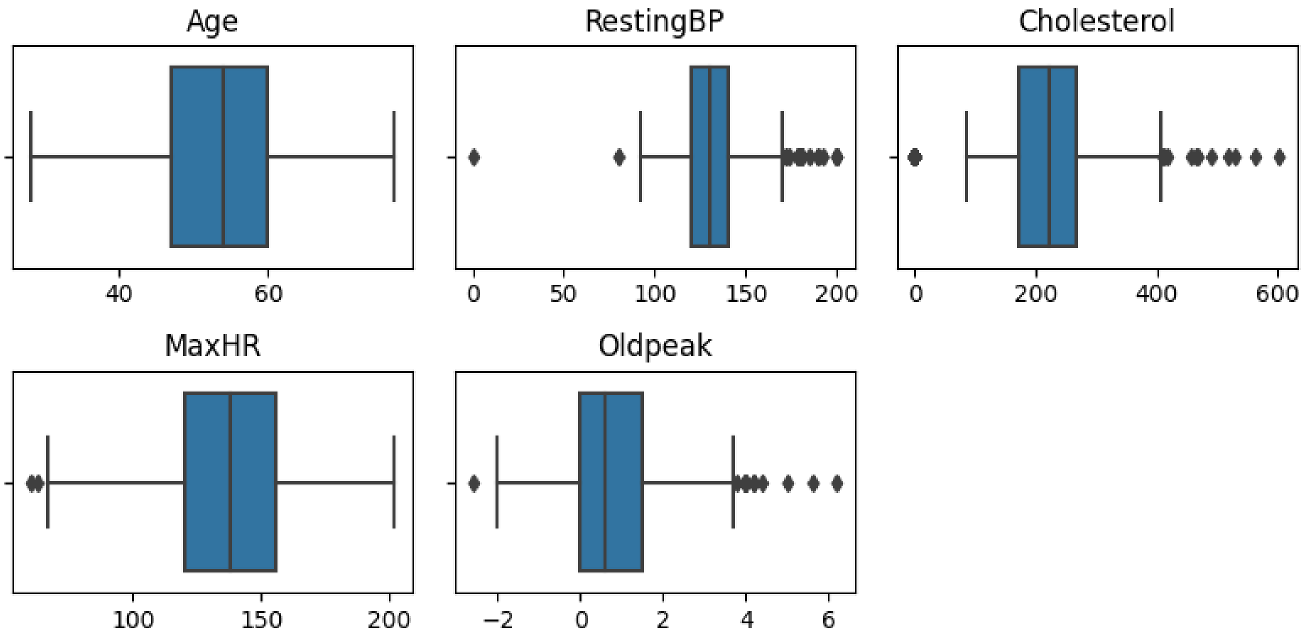
no duplicated rows

Handling Outliers

- Only perform on continuous data only, i.e. Age, RestingBP, Cholesterol, MaxHR and Oldpeak.
- Box plot is used to identify the potential outliers visually.

```
# List of variables to include in the boxplot
cont_var = ["Age", "RestingBP", "Cholesterol", "MaxHR", "Oldpeak"]
df1 = df[cont_var]

# Create a 2x3 grid of subplots
fig, axes = plt.subplots(2, 3, figsize=(8, 4))
axes = axes.flatten()

# Create a boxplot for each continuous variable
for i, column in enumerate(df1.columns):
    sns.boxplot(data=df1, x=column, ax=axes[i])
    axes[i].set_title(f"{column}")
    axes[i].set_xlabel("")  # Remove x-axis title

# Remove empty subplot
for i in range(len(df1.columns), len(axes)):
    fig.delaxes(axes[i])
```

```
# Adjust layout
plt.tight_layout()
plt.show()
```



It can be seen that the RestingBP, Cholesterol and Oldpeak are having a number of observations that are exceed that maximum range, while for RestingBP, Cholesterol and MaxHR are observed to have outliers that stay at below minimum range.

Outliers are to be removed via IQR methods.

```
# Remove outlier using IQR method

# Copy df for modification
df2 = df.copy()

# Iterate over continuous features only
for col in ["Age", "RestingBP", "Cholesterol", "MaxHR", "Oldpeak"]:

    # Calculate Q1, Q3 & IQR
    q1 = df2[col].quantile(0.25)
    q3 = df2[col].quantile(0.75)
    iqr = q3 - q1
```

```
    # Define the lower bound and upper bound
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    print(f"{col}: lower bound is {round(lower_bound, 3)}, upper bound is {round(upper_bound, 3)}")

    # Remove outliers by filtering based on lower & upper bounds
    df2 = df2[(df2[col] >= lower_bound) & (df2[col] <= upper_bound)]
```

```
Age: lower bound is 27.5, upper bound is 79.5
RestingBP: lower bound is 90.0, upper bound is 170.0
Cholesterol: lower bound is 37.0, upper bound is 403.0
MaxHR: lower bound is 65.0, upper bound is 217.0
Oldpeak: lower bound is -2.25, upper bound is 3.75
```

```
# check no of row removed under outlier removal
row_rm = df.shape[0] - df2.shape[0]
print(f"rows removed: {row_rm}")
```

```
rows removed: 217
```

```
# Set df2 (cleaned) to df (original)
df = df2
```

Label Encoding

- Only perform on character-type data only, i.e. Sex, ChestPainType, RestingECG, ExerciseAngina, ST_Slope.

```
# confirm unique value of string object
for col in df.columns:
    if df[col].dtype == 'object':
        value_counts = df[col].value_counts()
        print(f"'{col}':\n{value_counts}\n")
```

```
'Sex':
M    725
F    193
Name: Sex, dtype: int64

'ChestPainType':
ASY    496
NAP    203
ATA    173
TA      46
Name: ChestPainType, dtype: int64
```

```
'RestingECG':
Normal    552
LVH       188
ST        178
Name: RestingECG, dtype: int64

'ExerciseAngina':
N    547
Y    371
Name: ExerciseAngina, dtype: int64

'ST_Slope':
Flat    460
Up      395
Down     63
Name: ST_Slope, dtype: int64
```

```
# Label encoding for string object
label_encoder = preprocessing.LabelEncoder()

for col in ['Sex', 'ChestPainType', 'RestingECG', 'ExerciseAngina', 'ST_Slope']:
    if col in df.columns:
        df[col]= label_encoder.fit_transform(df[col])
        print(f"'{col}':\n{df[col].value_counts()}\n")
```

```
'Sex':
1    725
0    193
Name: Sex, dtype: int64

'ChestPainType':
0    496
2    203
1    173
3     46
Name: ChestPainType, dtype: int64

'RestingECG':
1    552
0    188
2    178
Name: RestingECG, dtype: int64

'ExerciseAngina':
0    547
1    371
```

```
Name: ExerciseAngina, dtype: int64

'ST_Slope':
1    460
2    395
0     63
Name: ST_Slope, dtype: int64
```

- Sex: M → 1; F → 0
- ChestPainType: ASY → 0; ATA → 1; NAP → 2; TA → 3
- RestingECG: LVH → 0; Normal → 1; ST → 2
- ExerciseAngina: N → 0; Y → 1
- ST_Slope: Down → 0; Flat → 1; Up → 2

```
# confirm datatype after encode
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Age             918 non-null    int64  
 1   Sex             918 non-null    int64  
 2   ChestPainType   918 non-null    int64  
 3   RestingBP       918 non-null    int64  
 4   Cholesterol     918 non-null    int64  
 5   FastingBS       918 non-null    int64  
 6   RestingECG      918 non-null    int64  
 7   MaxHR           918 non-null    int64  
 8   ExerciseAngina  918 non-null    int64  
 9   Oldpeak         918 non-null    float64
 10  ST_Slope        918 non-null    int64  
 11  HeartDisease    918 non-null    int64  
dtypes: float64(1), int64(11)
memory usage: 86.2 KB
```

Data Splitting

- Using 70-30 split

```
df_features = df.drop(['HeartDisease'], axis=1)
df_target = df['HeartDisease']

# split df at 70-30 ratio
X_train, X_test, y_train, y_test = train_test_split(df_features, df_target, test_size=0.3, random_state=123)
```

Data Transformation

- Conducted on train_set and test_set separately
- In this case, scaling and normalization is performed, by scaling all the continuous variables into a common scale to ease the comparison between variables with various units and ranges
- Continuous data variables: Age, RestingBP, Cholesterol, MaxHR and Oldpeak
- Formula employed: x=(x−mean)/stdev

```
# Initialize the StandardScaler object
scaler = StandardScaler()

# Fit and transform the train_set & test_set, features only
scaled_X_train = scaler.fit_transform(X_train)
scaled_X_test = scaler.fit_transform(X_test)
```

```
scaled_X_train = pd.DataFrame(scaled_X_train, columns = X_train.columns, index=X_train.index)
scaled_X_train.head()
```

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak
132	0.290767	0.518435	-0.814065	2.148213	1.718808	-0.523268	1.610581	-0.608935	1.227930	1.025585
9	-0.557743	-1.928883	0.248191	-0.699394	0.782760	-0.523268	0.017371	-0.687737	-0.814379	-0.834112
254	0.184703	0.518435	-0.814065	0.724410	0.458743	-0.523268	0.017371	-1.633361	1.227930	1.025585
787	1.457468	0.518435	-0.814065	-1.838436	0.917767	-0.523268	-1.575838	-0.490733	1.227930	0.002752
82	1.033213	0.518435	-0.814065	1.009171	0.233732	-0.523268	0.017371	-0.884742	-0.814379	-0.834112

```
scaled_X_test = pd.DataFrame(scaled_X_test, columns = X_test.columns, index=X_test.index)
scaled_X_test.head()
```

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	ExerciseAngina	Oldpeak	S
349	-1.925419	0.510171	-0.824873	-1.103911	-1.933420	1.622214	0.01699	-0.403646	1.183216	0.129534	-(
653	-1.287082	0.510171	1.198816	-0.129872	-0.218323	-0.616441	0.01699	0.575319	-0.845154	-0.828947	
7	-0.010408	0.510171	0.186971	-1.103911	0.048470	-0.616441	0.01699	0.262050	-0.845154	-0.828947	
571	1.585435	0.510171	-0.824873	0.357148	-0.885305	1.622214	0.01699	-1.030183	1.183216	0.608774	-(
171	-1.499861	0.510171	1.198816	0.357148	0.305735	-0.616441	0.01699	2.063345	-0.845154	-0.828947	

Data Modelling

Modeling algorithms considered include:

- Logistic Regression
- Decision Tree
- Random Forest
- Gradient Boosting Classifier
- K-Nearest Neighbours (KNN)
- Support Vector Machines (SVM)

Logistic regression

```
# Logistic regression
log_reg = LogisticRegression(random_state=123)

# Train the LR model on the train data
log_reg.fit(X_train, y_train)

# Predict on test data
y_pred_lr = log_reg.predict(X_test)

# Calculate evaluation metrics
accuracy_lr = accuracy_score(y_test, y_pred_lr)
precision_lr = precision_score(y_test, y_pred_lr)
recall_lr = recall_score(y_test, y_pred_lr)
f1_lr = f1_score(y_test, y_pred_lr)

# Print results
print("Accuracy:", round(accuracy_lr, 3))
print("Precision:", round(precision_lr, 3))
```

```
print("Recall:", round(recall_lr, 3))
print("F1-Score:", round(f1_lr, 3))
```

```
Accuracy: 0.833
Precision: 0.835
Recall: 0.878
F1-Score: 0.856
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Decision Tree

```
# Decision tree
d_tree = DecisionTreeClassifier(random_state=123)

# Train the DT model on the train data
d_tree.fit(X_train, y_train)

# Predict on test data
y_pred_dt = d_tree.predict(X_test)

# Calculate evaluation metrics
accuracy_dt = accuracy_score(y_test, y_pred_dt)
precision_dt = precision_score(y_test, y_pred_dt)
recall_dt = recall_score(y_test, y_pred_dt)
f1_dt = f1_score(y_test, y_pred_dt)

# Print results
print("Accuracy:", round(accuracy_dt, 3))
print("Precision:", round(precision_dt, 3))
print("Recall:", round(recall_dt, 3))
print("F1-Score:", round(f1_dt, 3))
```

```
Accuracy: 0.772
Precision: 0.808
Recall: 0.782
F1-Score: 0.795
```

Random Forest

```
# Random forest
r_forest = RandomForestClassifier(random_state=123)

# Train the DT model on the train data
r_forest.fit(X_train, y_train)

# Predict on test data
y_pred_rf = r_forest.predict(X_test)

# Calculate evaluation metrics
accuracy_rf = accuracy_score(y_test, y_pred_rf)
precision_rf = precision_score(y_test, y_pred_rf)
recall_rf = recall_score(y_test, y_pred_rf)
f1_rf = f1_score(y_test, y_pred_rf)

# Print results
print("Accuracy:", round(accuracy_rf, 3))
print("Precision:", round(precision_rf, 3))
print("Recall:", round(recall_rf, 3))
print("F1-Score:", round(f1_rf, 3))
```

```
Accuracy: 0.87
Precision: 0.857
Recall: 0.923
F1-Score: 0.889
```

Gradient Boosting Classifier

```
# Gradient boosting classifier
g_boost = GradientBoostingClassifier(random_state=123)

# Train the GBC on the train data
g_boost.fit(X_train, y_train)

# Predict on test data
y_pred_gbc = g_boost.predict(X_test)

# Calculate evaluation metrics
accuracy_gbc = accuracy_score(y_test, y_pred_gbc)
precision_gbc = precision_score(y_test, y_pred_gbc)
```

```
recall_gbc = recall_score(y_test, y_pred_gbc)
f1_gbc = f1_score(y_test, y_pred_gbc)

# Print results
print("Accuracy:", round(accuracy_gbc, 3))
print("Precision:", round(precision_gbc, 3))
print("Recall:", round(recall_gbc, 3))
print("F1-Score:", round(f1_gbc, 3))
```

```
Accuracy: 0.877
Precision: 0.863
Recall: 0.929
F1-Score: 0.895
```

K-Nearest Neighbours (KNN)

```
# K neighbors classifier
knn = KNeighborsClassifier()

# Train KNN on the train data
knn.fit(X_train, y_train)

# Predict on test data
y_pred_knn = knn.predict(X_test)

# Calculate evaluation metrics
accuracy_knn = accuracy_score(y_test, y_pred_knn)
precision_knn = precision_score(y_test, y_pred_knn)
recall_knn = recall_score(y_test, y_pred_knn)
f1_knn = f1_score(y_test, y_pred_knn)

# Print results
print("Accuracy:", round(accuracy_knn, 3))
print("Precision:", round(precision_knn, 3))
print("Recall:", round(recall_knn, 3))
print("F1-Score:", round(f1_knn, 3))
```

```
Accuracy: 0.75
Precision: 0.774
Recall: 0.788
F1-Score: 0.781
```

Support Vector Machines (SVM)

```
# Support vector machines classifier
svm = SVC(random_state=123)

# Train SVC on train data
svm.fit(X_train, y_train)

# Predict on the test data
y_pred_svm = svm.predict(X_test)

# Calculate evaluation metrics
accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm)
recall_svm = recall_score(y_test, y_pred_svm)
f1_svm = f1_score(y_test, y_pred_svm)

# Print results
print("Accuracy:", round(accuracy_svm, 3))
print("Precision:", round(precision_svm, 3))
print("Recall:", round(recall_svm, 3))
print("F1-Score:", round(f1_svm, 3))
```

```
Accuracy: 0.721
Precision: 0.762
Recall: 0.737
F1-Score: 0.749
```

Model Evaluation

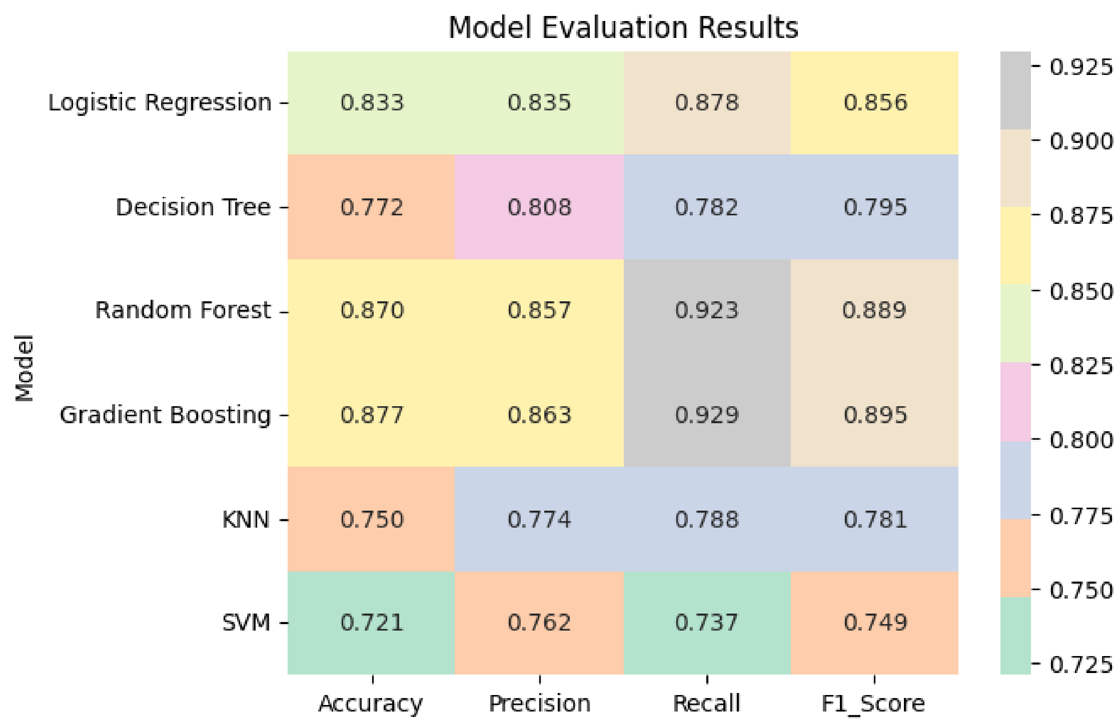
```
# Define the models and their corresponding evaluation metrics
models = ["Logistic Regression", "Decision Tree", "Random Forest", "Gradient Boosting", "KNN", "SVM"]
accuracy = [accuracy_lr, accuracy_dt, accuracy_rf, accuracy_gbc, accuracy_knn, accuracy_svm]
precision = [precision_lr, precision_dt, precision_rf, precision_gbc, precision_knn, precision_svm]
recall = [recall_lr, recall_dt, recall_rf, recall_gbc, recall_knn, recall_svm]
f1_score = [f1_lr, f1_dt, f1_rf, f1_gbc, f1_knn, f1_svm]

# Create summary table in df
summary_table = pd.DataFrame({
    "Model": models,
    "Accuracy": accuracy,
    "Precision": precision,
    "Recall": recall,
    "F1_Score": f1_score
```

```
})
summary_table.round(3)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-340e2c8e5251> in <cell line: 3>()
      1 # Define the models and their corresponding evaluation metrics
      2 models = ["Logistic Regression", "Decision Tree", "Random Forest", "Gradient Boosting", "KNN", "SVM"]
----> 3 accuracy = [accuracy_lr, accuracy_dt, accuracy_rf, accuracy_gbc, accuracy_knn, accuracy_svm]
      4 precision = [precision_lr, precision_dt, precision_rf, precision_gbc, precision_knn, precision_svm]
      5 recall = [recall_lr, recall_dt, recall_rf, recall_gbc, recall_knn, recall_svm]

NameError: name 'accuracy_lr' is not defined
```

SEARCH STACK OVERFLOW

```
# Create summary table in heatmap
sns.heatmap(data=summary_table.set_index('Model').iloc[:, :], annot=True, fmt=".3f", cmap='Pastel2')
plt.title("Model Evaluation Results")
plt.show()
```



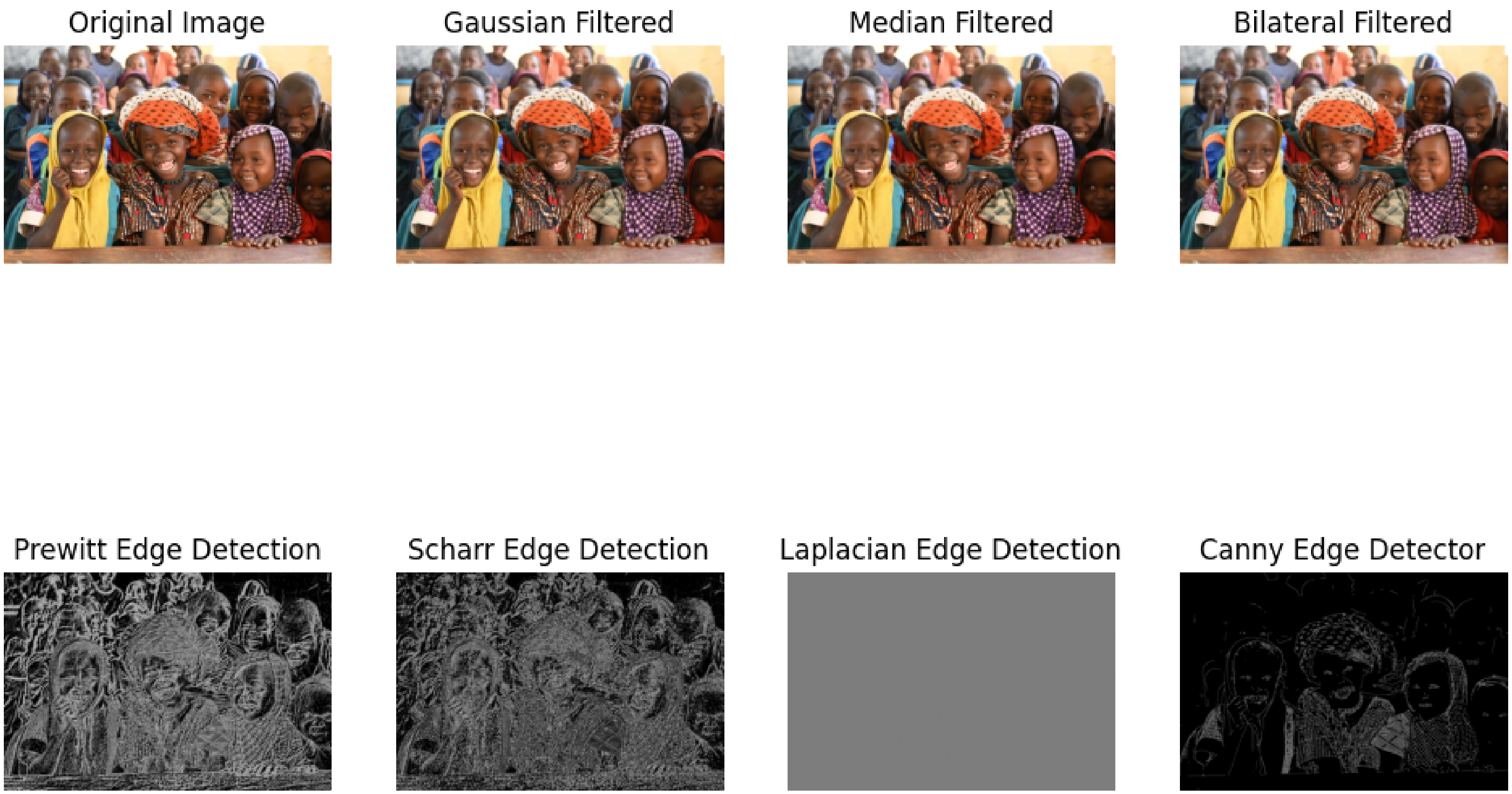
Best 2 models in predicting heart failure:

- Random forest
- Gradient boosting

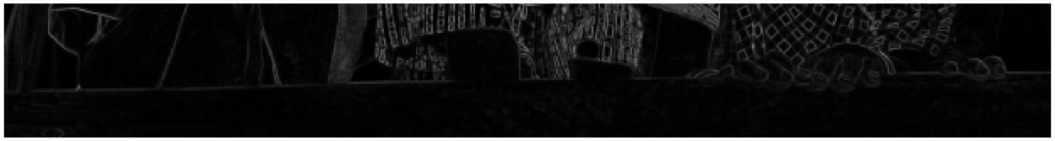
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read the color image
img = cv2.imread('/content/qwa.jpg')

# Split the image into color channels (BGR)
blue, green, red = cv2.split(img)

# Apply a larger Gaussian filter for smoothing
gaussian_blue = cv2.GaussianBlur(blue, (15, 15), 0)
gaussian_green = cv2.GaussianBlur(green, (15, 15), 0)
gaussian_red = cv2.GaussianBlur(red, (15, 15), 0)

# Merge the Gaussian filtered color channels back into an RGB image
gaussian_filtered_img = cv2.merge((gaussian_blue, gaussian_green, gaussian_red))

# Apply a larger Median filter for noise reduction
median_blue = cv2.medianBlur(blue, 15)
median_green = cv2.medianBlur(green, 15)
median_red = cv2.medianBlur(red, 15)

# Merge the median filtered color channels back into an RGB image
median_filtered_img = cv2.merge((median_blue, median_green, median_red))

# Apply a bilateral filter with larger parameters for edge-preserving smoothing
bilateral_blue = cv2.bilateralFilter(blue, d=15, sigmaColor=75, sigmaSpace=75)
bilateral_green = cv2.bilateralFilter(green, d=15, sigmaColor=75, sigmaSpace=75)
bilateral_red = cv2.bilateralFilter(red, d=15, sigmaColor=75, sigmaSpace=75)

# Merge the bilateral filtered color channels back into an RGB image
bilateral_filtered_img = cv2.merge((bilateral_blue, bilateral_green, bilateral_red))

# Convert the image to grayscale
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply the Prewitt filter for edge detection
prewitt_x = cv2.filter2D(gray_img, -1, np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]))
prewitt_y = cv2.filter2D(gray_img, -1, np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]]))
prewitt_magnitude = np.sqrt(prewitt_x**2 + prewitt_y**2)

# Apply the Scharr filter for edge detection
```

```
# Apply the Scharr filter for edge detection
scharr_x = cv2.Scharr(gray_img, -1, 1, 0)
scharr_y = cv2.Scharr(gray_img, -1, 0, 1)
scharr_magnitude = np.sqrt(scharr_x**2 + scharr_y**2)

# Apply the Laplacian filter for edge detection
laplacian = cv2.Laplacian(gray_img, cv2.CV_64F)

# Apply the Canny edge detector with optimized thresholds
canny = cv2.Canny(gray_img, 100, 200)

# Display the original and filtered color images
plt.figure(figsize=(12, 12))

plt.subplot(3, 4, 1)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

plt.subplot(3, 4, 2)
plt.imshow(cv2.cvtColor(gaussian_filtered_img.astype(np.uint8), cv2.COLOR_BGR2RGB))
plt.title('Gaussian Filtered')
plt.axis('off')

plt.subplot(3, 4, 3)
plt.imshow(cv2.cvtColor(median_filtered_img.astype(np.uint8), cv2.COLOR_BGR2RGB))
plt.title('Median Filtered')
plt.axis('off')

plt.subplot(3, 4, 4)
plt.imshow(cv2.cvtColor(bilateral_filtered_img.astype(np.uint8), cv2.COLOR_BGR2RGB))
plt.title('Bilateral Filtered')
plt.axis('off')

plt.subplot(3, 4, 5)
plt.imshow(prewitt_magnitude, cmap='gray')
plt.title('Prewitt Edge Detection')
plt.axis('off')

plt.subplot(3, 4, 6)
plt.imshow(scharr_magnitude, cmap='gray')
plt.title('Scharr Edge Detection')
plt.axis('off')

plt.subplot(3, 4, 7)
plt.imshow(laplacian, cmap='gray')
plt.title('Laplacian Edge Detection')
```

```
plt.axis('off')

plt.subplot(3, 4, 8)
plt.imshow(canny, cmap='gray')
plt.title('Canny Edge Detector')
plt.axis('off')

# Convert the image to grayscale
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Apply Sobel filter for edge detection
sobel_x = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=3)
gradient_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)

# Normalize the gradient magnitude for better visualization
gradient_magnitude = cv2.normalize(gradient_magnitude, None, 0, 255, cv2.NORM_MINMAX, cv2.CV_8U)

plt.figure(figsize=(8, 8))
plt.imshow(gradient_magnitude, cmap='gray')
plt.title('Sobel Filtered Image (Edge Detection)')
plt.axis('off')

plt.show()
```


Original Image
Gaussian Filtered
Median Filtered
Bilateral Filtered
Prewitt Edge Detection
Scharr Edge Detection
Laplacian Edge Detection
Canny Edge Detector


Sobel Filtered Image (Edge Detection)



```
#long short term memoryy-----> STOCK PRICER-ANTICPATER
```

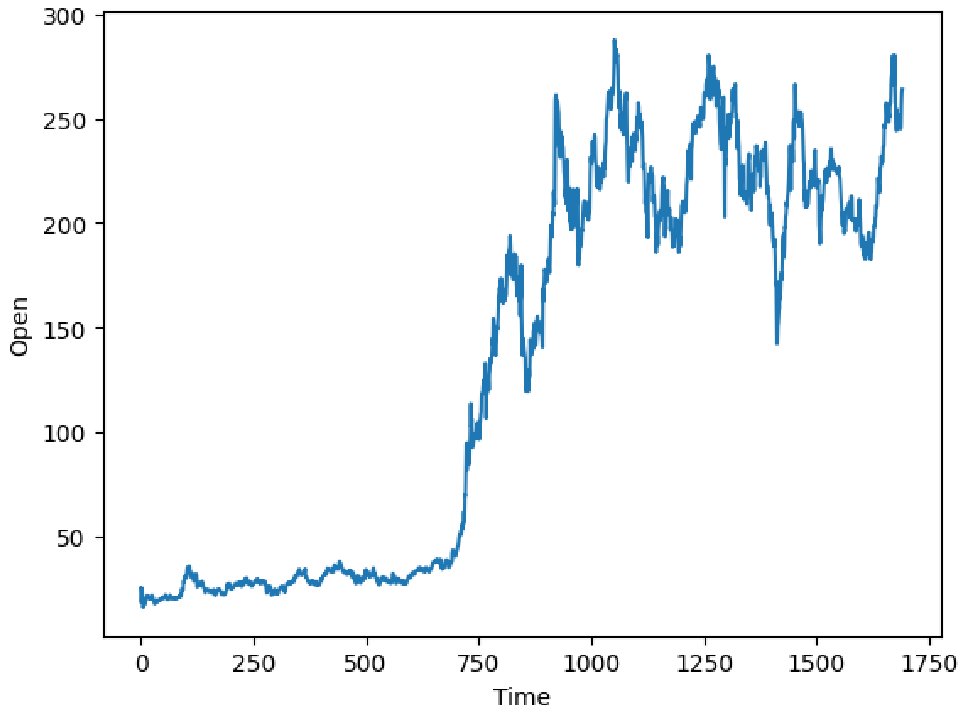
```
import numpy as np # Linear algebra.
import pandas as pd # Data processing.
import matplotlib.pyplot as plt # Visualize
import math
from keras.models import Sequential # Create Model
from keras.layers import Dense # Neurons
from keras.layers import LSTM # Long Short Term Memory
from sklearn.preprocessing import MinMaxScaler # Normalize
from sklearn.metrics import mean_squared_error # Loss Function
from sklearn.model_selection import train_test_split
```

```
data = pd.read_csv("/content/Tesla.csv - Tesla.csv (1).csv") # Import data
data.head(12)
```

	Date	Open	High	Low	Close	Volume	Adj Close
0	6/29/2010	19.000000	25.000000	17.540001	23.889999	18766300	23.889999
1	6/30/2010	25.790001	30.420000	23.299999	23.830000	17187100	23.830000
2	7/1/2010	25.000000	25.920000	20.270000	21.959999	8218800	21.959999
3	7/2/2010	23.000000	23.100000	18.709999	19.200001	5139800	19.200001
4	7/6/2010	20.000000	20.000000	15.830000	16.110001	6866900	16.110001
5	7/7/2010	16.400000	16.629999	14.980000	15.800000	6921700	15.800000
6	7/8/2010	16.139999	17.520000	15.570000	17.459999	7711400	17.459999
7	7/9/2010	17.580000	17.900000	16.549999	17.400000	4050600	17.400000
8	7/12/2010	17.950001	18.070000	17.000000	17.049999	2202500	17.049999
9	7/13/2010	17.389999	18.639999	16.900000	18.139999	2680100	18.139999
10	7/14/2010	17.940001	20.150000	17.760000	19.840000	4195200	19.840000
11	7/15/2010	19.940001	21.500000	19.000000	19.889999	3739800	19.889999

```
df = data.iloc[:,1].values # We use "Open" column.
plt.plot(df)
plt.xlabel("Time")
```

```
plt.ylabel("Open")
plt.show()
```



DATA NORMALISATION--NW

```
df = df.reshape(-1,1)

scaler = MinMaxScaler(feature_range = (0,1)) # Normalize data
df = scaler.fit_transform(df)
np.max(df)
```

```
1.0
```

```
#DIVIDE DATA INTO TRAINAND TEST STRIP DATA   # 75% AND 25% RESPECTIVELY
```

```
# Test - Train Split
train_size = int(len(df) * 0.75) # % 75 Train
test_size = len(df) - train_size # % 25 Test
print("Train Size :",train_size,"Test Size :",test_size)
```

```
train = df[0:train_size,:]
test = df[train_size:len(df),:]
```

```
Train Size : 1269 Test Size : 423
```

```
#PREPARE DATA

time_stemp = 10

datax = []
datay = []
for i in range(len(train)-time_stemp-1):
    a = train[i:(i+time_stemp), 0]
    datax.append(a)
    datay.append(train[i + time_stemp, 0])
trainx = np.array(datax)
trainy = np.array(datay)


datax = []
datay = []
for i in range(len(test)-time_stemp-1):
    a = test[i:(i+time_stemp), 0]
    datax.append(a)
    datay.append(test[i + time_stemp, 0])
testx = np.array(datax)
testy = np.array(datay)

trainx = np.reshape(trainx, (trainx.shape[0], 1, trainx.shape[1])) # For Keras
testx = np.reshape(testx, (testx.shape[0], 1,testx.shape[1])) # For Keras
print(trainx.shape)
testx.shape
```

```
(1258, 1, 10)
(412, 1, 10)
```

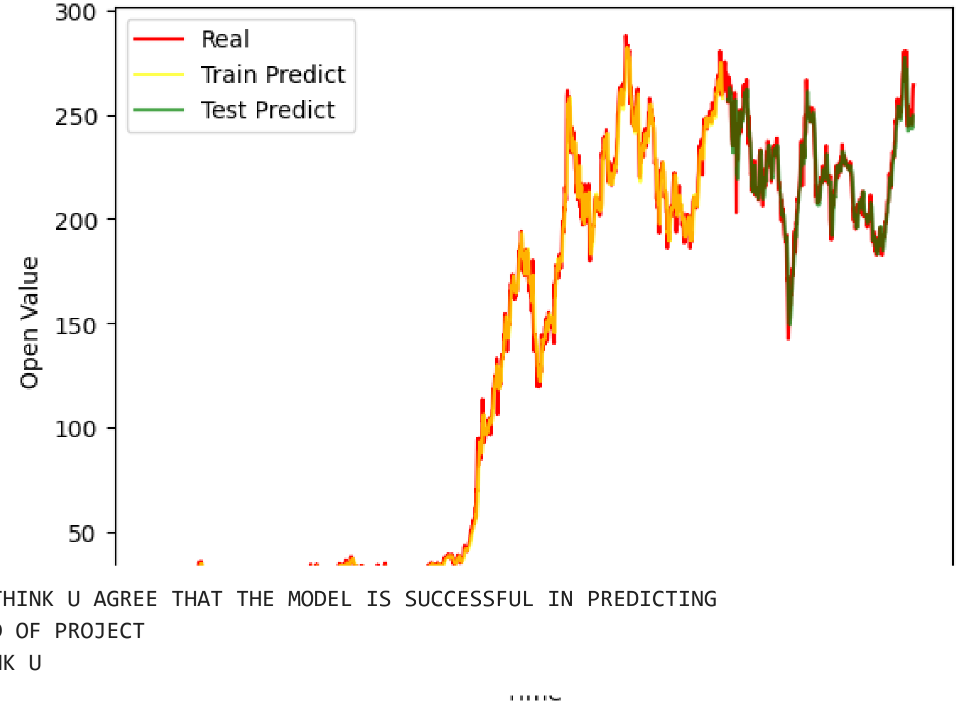
```
#WE CREATE A MODEL

epochs = 200
model = Sequential()
model.add(LSTM(10, input_shape = (1, time_stemp)))
model.add(Dense(1)) # Output Layer
model.compile(loss = "mean_squared_error", optimizer = "adam")
```

```
history = model.fit(trainx,trainy, epochs = epochs, batch_size = 50, verbose=0)
# As you can see, Loss is very little


#LOSS VISUALISATION
epoch = np.arange(0, epochs, 10)
losses = []
for i in epoch:
    if i % 10 == 0:
        losses.append(history.history["loss"][i])

data = {"epoch":epoch,"loss":losses}
data = pd.DataFrame(data) # Create dataframe for visualize with plotly


import plotly.express as px

fig = px.line(data,x="epoch",y="loss",width = 1200, height = 500)
fig.show()
```

```
# LETS TRAIN AND TEST

train_predict = model.predict(trainx)
test_predict = model.predict(testx)

train_predict = scaler.inverse_transform(train_predict)
test_predict = scaler.inverse_transform(test_predict)
trainy = scaler.inverse_transform([trainy])
testy = scaler.inverse_transform([testy])

train_score = math.sqrt(mean_squared_error(trainy[0], train_predict[:,0])) # mean_squared_error -> Loss Function
print("Train Score : %2.f RMSE" % (train_score))
test_score = math.sqrt(mean_squared_error(testy[0], test_predict[:,0]))
print("Test Score : %2.f RMSE" % (test_score))
```

```
40/40 [==============================] - 1s 2ms/step
13/13 [==============================] - 0s 2ms/step
Train Score :  5 RMSE
Test Score :  7 RMSE
```

```
#VISUALISING AGAIN

train_predict_plot = np.empty_like(df)
train_predict_plot[:,:] = np.nan
train_predict_plot[time_stemp:len(train_predict)+time_stemp, :] = train_predict

test_predict_plot = np.empty_like(df)
test_predict_plot[:, :] = np.nan
test_predict_plot[len(train_predict)+(time_stemp*2)+1:len(df)-1, :] = test_predict

plt.plot(scaler.inverse_transform(df),color = "red",label = "Real")
plt.plot(train_predict_plot,label = "Train Predict",color = "yellow",alpha = 0.7)
plt.plot(test_predict_plot,label = "Test Predict",color = "green", alpha = 0.7)
plt.legend()
plt.xlabel("Time")
plt.ylabel("Open Value")
plt.show()
```




```
# I THINK U AGREE THAT THE MODEL IS SUCCESSFUL IN PREDICTING
# END OF PROJECT
#THANK U
```