keras and tf project2

February 21, 2023

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from scipy.stats import skew, norm
from sklearn.impute import SimpleImputer
from sklearn import preprocessing
from sklearn.feature_selection import chi2
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE
from sklearn.preprocessing import OrdinalEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import accuracy_score, confusion_matrix
import keras_tuner as kt
```

[2]:
```
data = pd.read_csv('loan_data.csv')
```

[3]:
```
# Checking the first five variables of the dataframe
data.head()
```

[3]:
```
   credit.policy             purpose  int.rate  installment  log.annual.inc  \
0              1  debt_consolidation    0.1189       829.10       11.350407
1              1         credit_card    0.1071       228.22       11.082143
2              1  debt_consolidation    0.1357       366.86       10.373491
3              1  debt_consolidation    0.1008       162.34       11.350407
4              1         credit_card    0.1426       102.92       11.299732

     dti  fico  days.with.cr.line  revol.bal  revol.util  inq.last.6mths  \
0  19.48   737        5639.958333      28854        52.1               0
```

```
1  14.29   707        2760.000000      33623        76.7                  0
2  11.63   682        4710.000000       3511        25.6                  1
3   8.10   712        2699.958333      33667        73.2                  1
4  14.97   667        4066.000000       4740        39.5                  0

   delinq.2yrs  pub.rec  not.fully.paid
0            0        0               0
1            0        0               0
2            0        0               0
3            0        0               0
4            1        0               0
```

[4]:
```python
# Checking the size of the dataframe
data.shape
```

[4]:
```
(9578, 14)
```

[5]:
```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9578 entries, 0 to 9577
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   credit.policy      9578 non-null   int64
 1   purpose            9578 non-null   object
 2   int.rate           9578 non-null   float64
 3   installment        9578 non-null   float64
 4   log.annual.inc     9578 non-null   float64
 5   dti                9578 non-null   float64
 6   fico               9578 non-null   int64
 7   days.with.cr.line  9578 non-null   float64
 8   revol.bal          9578 non-null   int64
 9   revol.util         9578 non-null   float64
 10  inq.last.6mths     9578 non-null   int64
 11  delinq.2yrs        9578 non-null   int64
 12  pub.rec            9578 non-null   int64
 13  not.fully.paid     9578 non-null   int64
dtypes: float64(6), int64(7), object(1)
memory usage: 1.0+ MB
```

[6]:
```python
# Checking the target variable
data['not.fully.paid'].value_counts()
```

[6]:
```
0    8045
1    1533
Name: not.fully.paid, dtype: int64
```

[7]:
```
#handling imbalanced dataset
not_fully_paid_0 = data[data['not.fully.paid'] == 0]
not_fully_paid_1 = data[data['not.fully.paid'] == 1]

print('not_fully_paid_0', not_fully_paid_0.shape)
print('not_fully_paid_1', not_fully_paid_1.shape)
```

```
not_fully_paid_0 (8045, 14)
not_fully_paid_1 (1533, 14)
```

[8]:
```
#handling imbalanced data
from sklearn.utils import resample
df_minority_upsampled = resample(not_fully_paid_1, replace = True, n_samples =␣
↪8045)
df = pd.concat([not_fully_paid_0, df_minority_upsampled])

from sklearn.utils import shuffle
df = shuffle(df)
```

[9]:
```
#imbalanced data handling
df['not.fully.paid'].value_counts()
```

[9]:
```
1    8045
0    8045
Name: not.fully.paid, dtype: int64
```

[10]:
```
#The data contained in the dataframe is comprised of float64, int64 and object␣
↪values.
```

[11]:
```
# Separating data to include numerical data only
num_data = df[["int.rate", "installment", "log.annual.inc", "dti", "fico",␣
↪"days.with.cr.line", "revol.bal",
               "revol.util", "not.fully.paid"]]
num_data
```

[11]:

	int.rate	installment	log.annual.inc	dti	fico	days.with.cr.line	\
4640	0.1670	710.03	11.264464	24.60	677	1748.958333	
4694	0.1913	734.42	11.050890	8.86	677	7350.000000	
3013	0.1379	67.30	11.332602	13.42	677	5579.958333	
1125	0.0863	142.33	11.277152	7.53	722	5761.000000	
5611	0.0788	312.81	10.858999	9.09	762	5791.041667	
…	…	…	…	…	…	…	
8271	0.1261	228.69	11.141862	23.62	712	7440.041667	
5519	0.1287	201.80	10.968198	18.12	687	3480.000000	
4914	0.1183	457.25	10.915088	20.86	717	4470.000000	
878	0.1324	265.41	12.206073	15.62	707	4680.000000	
6533	0.1183	795.22	11.918391	0.70	812	11702.000000	

```
      revol.bal  revol.util  not.fully.paid
4640          0       49.63               1
4694       9881       99.80               0
3013      11386       55.00               0
1125      20237       32.70               0
5611       7386       33.90               0
…           …          …              …
8271      92929       97.70               0
5519       6184       85.90               0
4914      21981       59.60               0
878       36293       62.30               1
6533        346        0.70               0

[16090 rows x 9 columns]
```

[12]:
```python
# Checking the features in the numerical data
num_data_features = num_data.columns
num_data_features
```

[12]:
```
Index(['int.rate', 'installment', 'log.annual.inc', 'dti', 'fico',
       'days.with.cr.line', 'revol.bal', 'revol.util', 'not.fully.paid'],
      dtype='object')
```

[13]:
```python
# Separating data to include categorical data only
cat_data = df[["credit.policy", "purpose", "inq.last.6mths", "delinq.2yrs",␣
 ↪"not.fully.paid"]]
cat_data
```

[13]:
```
      credit.policy             purpose  inq.last.6mths  delinq.2yrs  \
4640              1    home_improvement               1            0
4694              1  debt_consolidation               0            0
3013              1           all_other               2            2
1125              1  debt_consolidation               2            0
5611              1           all_other               0            0
…               …                   …             …           …
8271              0  debt_consolidation               0            0
5519              1         credit_card               1            0
4914              1         credit_card               1            0
878               1           all_other               3            0
6533              1           all_other               0            0

      not.fully.paid
4640               1
4694               0
3013               0
1125               0
```

```
5611              0
…              …
8271              0
5519              0
4914              0
878               1
6533              0

[16090 rows x 5 columns]
```

[14]:
```
# Checking the features in the numerical data
cat_data_features = cat_data.columns
cat_data_features
```

[14]:
```
Index(['credit.policy', 'purpose', 'inq.last.6mths', 'delinq.2yrs',
       'not.fully.paid'],
      dtype='object')
```

[15]:
```
#DATA EXPLORATION
```

[16]:
```
#Exploration of statistical analysis such as the identification of standards␣
↪deviations and central tendecies, the quantiles and minimums and maximums of␣
↪data variables.
```

[17]:
```
# Checking the statistics of the numerical data
num_data.describe()
```

[17]:

	int.rate	installment	log.annual.inc	dti	fico \
count	16090.000000	16090.000000	16090.000000	16090.000000	16090.000000
mean	0.126622	329.677479	10.916284	12.846860	705.669981
std	0.026946	215.347795	0.641872	6.926944	36.836101
min	0.060000	15.670000	7.547502	0.000000	612.000000
25%	0.110300	166.500000	10.518889	7.430000	677.000000
50%	0.125400	276.300000	10.915088	12.950000	702.000000
75%	0.143800	468.140000	11.289782	18.220000	730.750000
max	0.216400	940.140000	14.528354	29.960000	827.000000

	days.with.cr.line	revol.bal	revol.util	not.fully.paid
count	16090.000000	1.609000e+04	16090.000000	16090.000000
mean	4486.125888	1.862119e+04	48.876132	0.500000
std	2473.576419	4.116208e+04	29.127270	0.500016
min	178.958333	0.000000e+00	0.000000	0.000000
25%	2789.000000	3.168750e+03	25.200000	0.000000
50%	4080.000000	8.712000e+03	49.300000	0.500000
75%	5699.041667	1.913275e+04	73.200000	1.000000
max	17639.958330	1.207359e+06	119.000000	1.000000

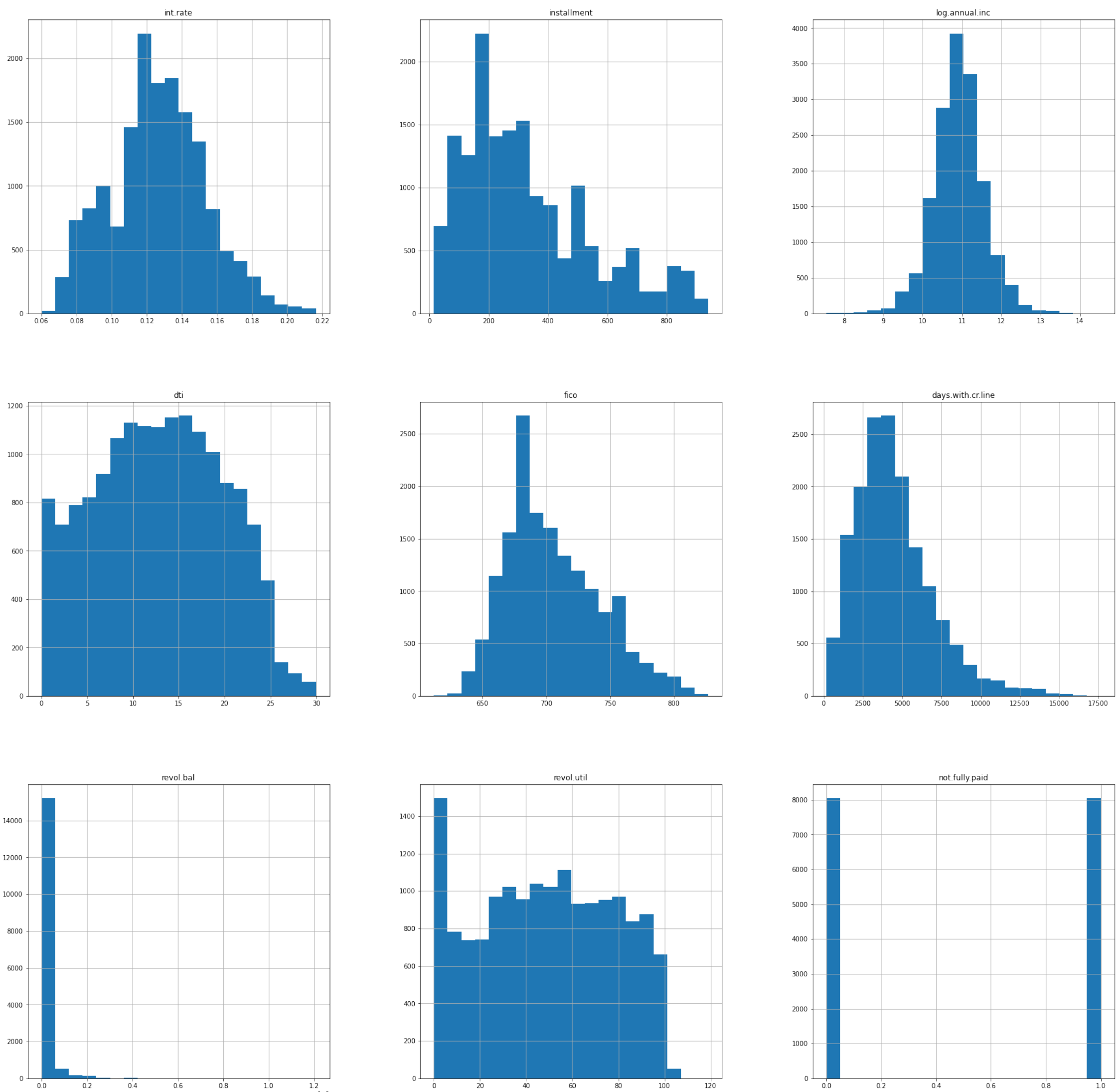
[18]:
```
#Summary of the statistical data above
```

[19]:
```
#The feature revol.bal (The borrower's revolving line utilization rate) has the highest standard deviation and so, it expected that this variable will contain outliers.
```

[20]:
```
#Other features such as days.with.cr.line, installment, fico, and revol.util also show high standard deviations, as such, outliers in this data have to be detect and handled.
```

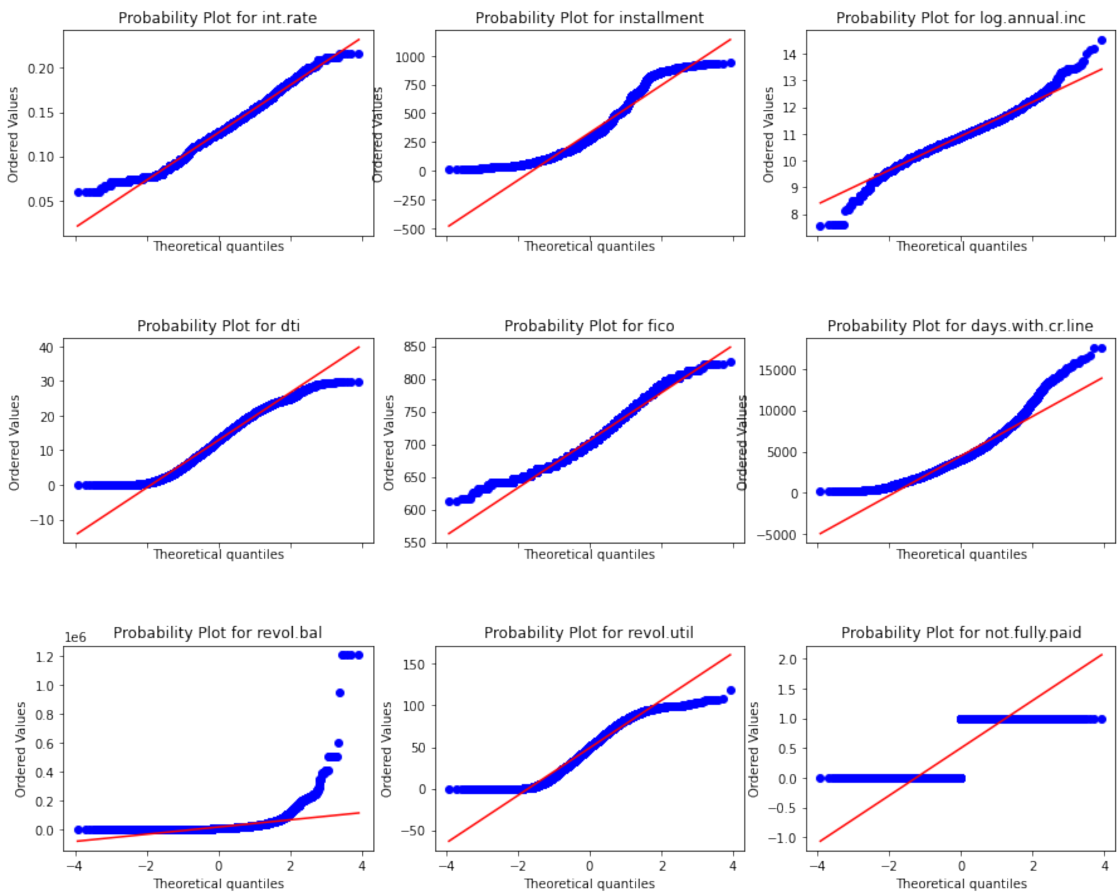
[21]:
```
#The highest number of days the borrower has had a credit line (days.with.cr.line) was 17640 days.
```

[22]:
```
# Checking the distribution of the numerical continous data
num_data.hist(figsize = (30, 30), bins = 20, legend = False)
plt.show()
```



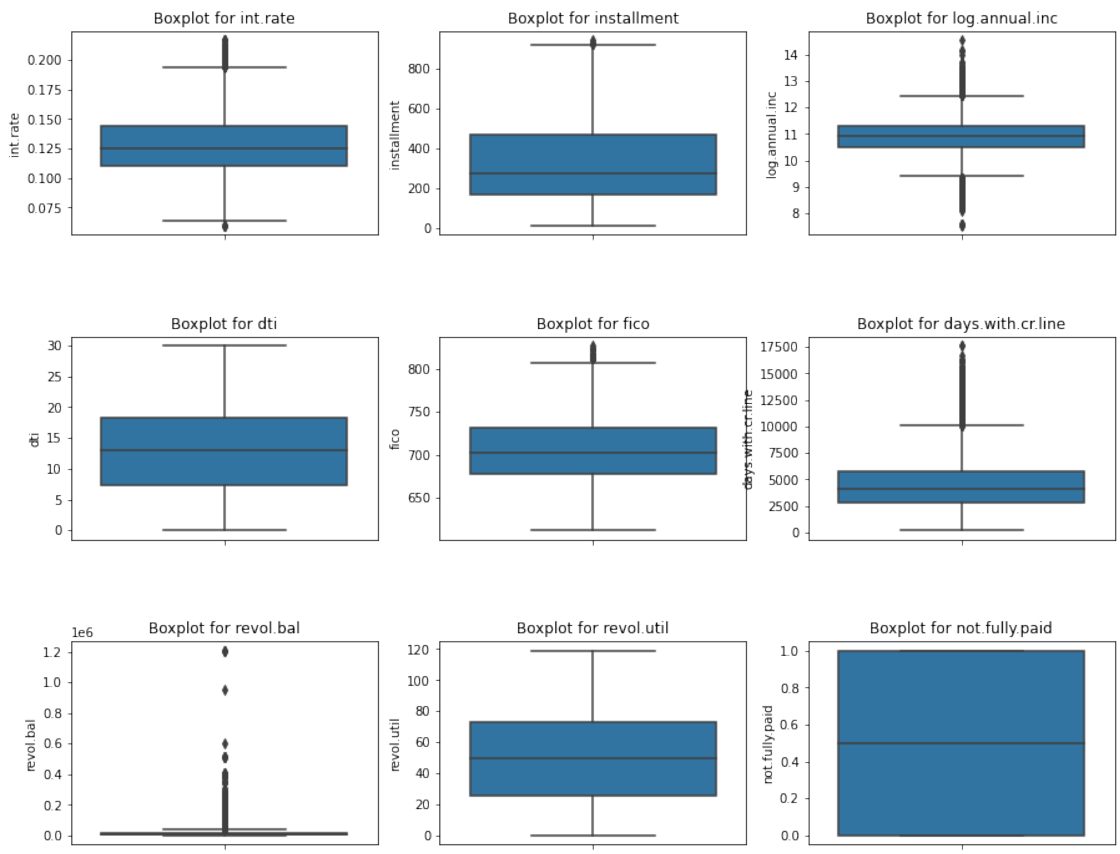
[23]:
```
#revol.bal, days.with.cr.line, installment, fico, and revol.util may contain␣
↪outliers because they are all positively skewed.
```

[24]:
```
# Developing a probability plot to find out how compacted or degress of␣
↪normalisation the data is.

# Defining subplot grid
fig, axs = plt.subplots(nrows = 3, ncols = 3, figsize = (15, 12), sharex = True)
fig.subplots_adjust(hspace = 0.5)


for i, col in enumerate(num_data):
    ax = plt.subplot(3, 3, i+1)
    stats.probplot(num_data[col], plot = ax)
    ax.set_title(f"Probability Plot for {col}")
```



[25]:
```
# The variables such as revol.bal, days.with.cr.line, installment, fico, and␣
↪revol.util may contain outliers because the values in these variables do not␣
↪fall well around the best fit line.
```

[26]:
```
# Creating plots showing the uncertainity in the data and the outliers.

# Defining subplot grid
fig, axs = plt.subplots(nrows = 3, ncols = 3, figsize = (15, 12), sharex = True)
fig.subplots_adjust(hspace = 0.5)


for i, col in enumerate(num_data):
    ax = plt.subplot(3, 3, i+1)
    sns.boxplot(y = df[col])
    ax.set_title(f"Boxplot for {col}")
plt.show()
```



[27]:
```
#From the above graphs, it can be seen that the outliers exist in the variables␣
↪such as the following: int.rate, installment, log.annual.inc, fico, days.
↪with.cr.line and revol.bal.
```

[28]:
```
# Converting categorical feature into numerical feature
cat_data = cat_data.copy()
le = preprocessing.LabelEncoder()
cat_data["purpose"] = le.fit_transform(cat_data["purpose"].astype(str))
cat_data.head()
```

[28]:

	credit.policy	purpose	inq.last.6mths	delinq.2yrs	not.fully.paid
4640	1	4	1	0	1
4694	1	2	0	0	0
3013	1	0	2	2	0
1125	1	2	2	0	0
5611	1	0	0	0	0

[29]:
```
# Checking the statistics of the numerical data
cat_data.describe()
```

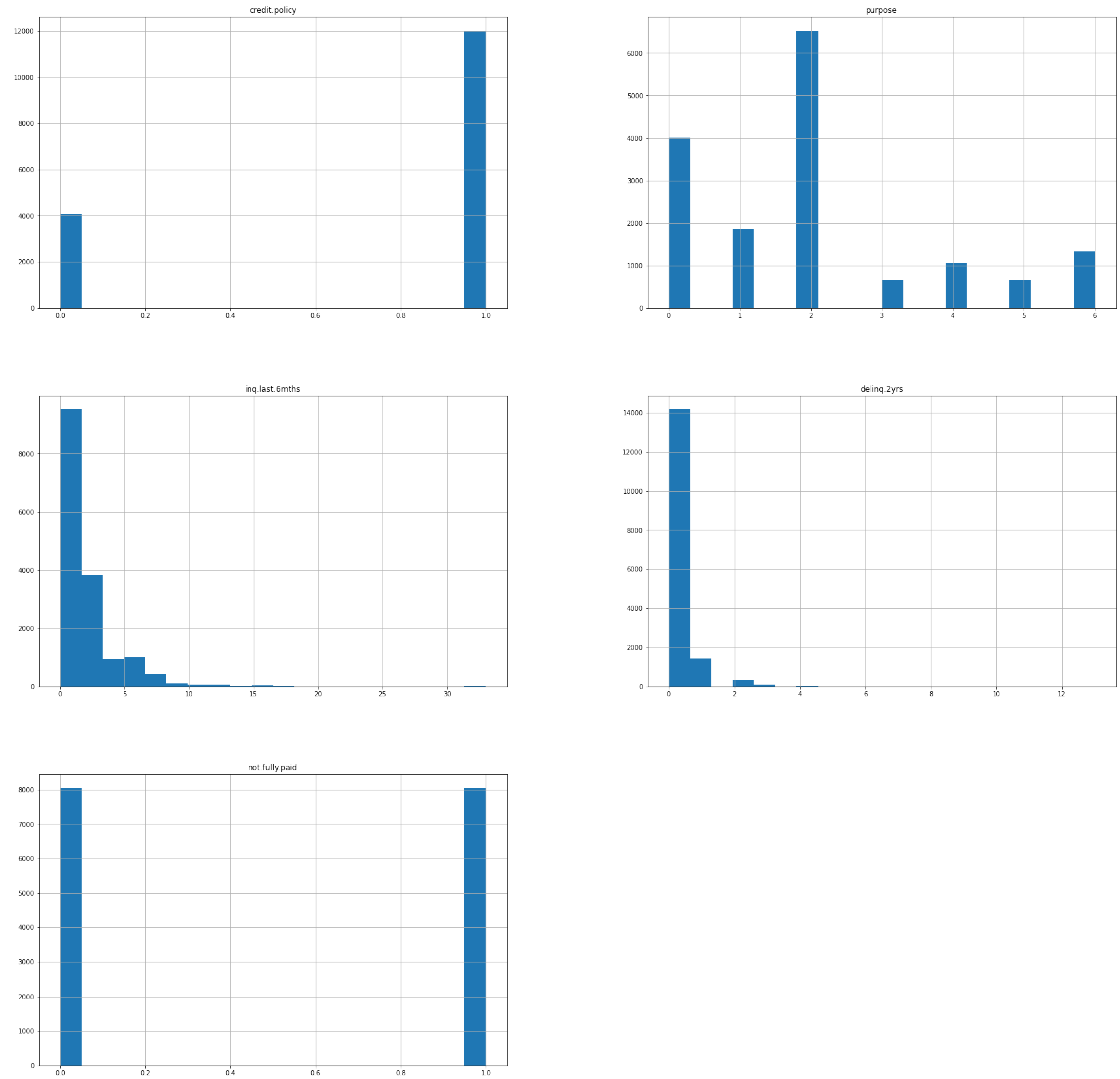
[29]:

	credit.policy	purpose	inq.last.6mths	delinq.2yrs	\
count	16090.000000	16090.000000	16090.000000	16090.000000	
mean	0.746613	2.010690	1.872281	0.163580	
std	0.434964	1.760722	2.538173	0.527063	
min	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	1.000000	0.000000	0.000000	
50%	1.000000	2.000000	1.000000	0.000000	
75%	1.000000	2.000000	3.000000	0.000000	
max	1.000000	6.000000	33.000000	13.000000	

	not.fully.paid
count	16090.000000
mean	0.500000
std	0.500016
min	0.000000
25%	0.000000
50%	0.500000
75%	1.000000
max	1.000000

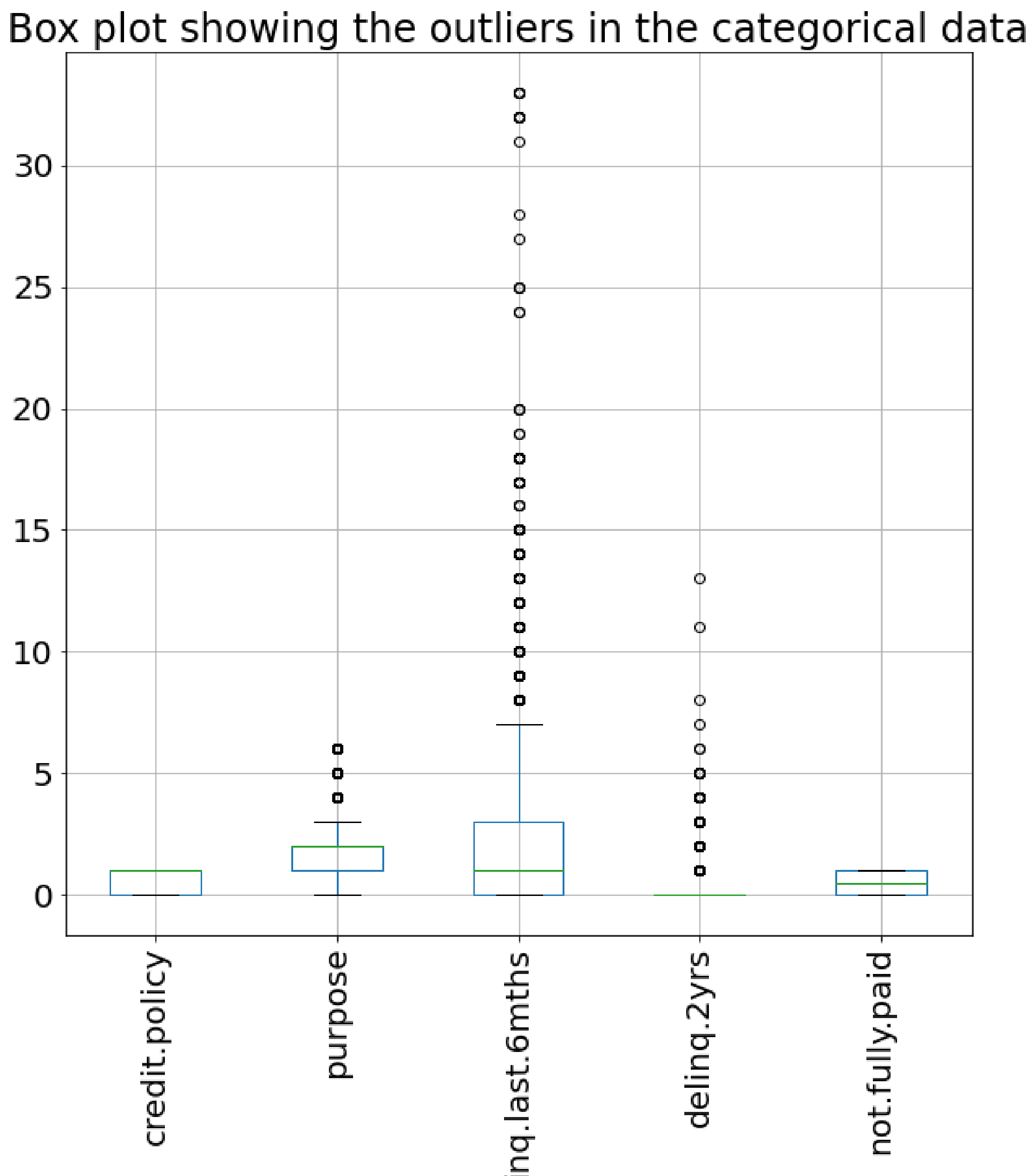
[30]:
```
#The standard deviation in all the variables is small because the data ranges␣
↪from either 0 to 1 and 0 to 6 or 0 to 33 and 0 to 13.
```

[31]:
```
# Checking the distribution of the categorical data
cat_data.hist(figsize = (30, 30), bins = 20, legend = False)
plt.rcParams["font.size"] = "20"
plt.show()
```



[32]:
```python
#It can be seen that most of the categorical data is positively skewed.
#Most clients satisfied the credit policy.
#Most clients decided to take the loan for purposes of loan consolidation.
```

[33]:
```python
# Creating plots showing the uncertainity in the categorical data and the outliers.
plt.figure(figsize = (10, 10))
cat_data.boxplot()
plt.xticks(rotation = 90)
plt.title("Box plot showing the outliers in the categorical data")
plt.show()
```



[34]:
```
#The graph shown above indicates that outliers exist in purpose, inq.last.
↪6mths, delinq.2yrs
```

[35]:
```
#DATA WRANGLING
```

[36]:
```
#In this section, data will be wrangled in the sense that all missing values␣
↪will handled in both numerical and categorical data.
```

```
#Outliers in the numerical and categorical data will be eliminated so that the␣
↪data can produce an improved result in the prediction model at a faster time.
```

[37]:
```
#Handling missing values in the data frame
```

[38]:
```
# Converting the categorical feature in the data set into a numerical feature
le = preprocessing.LabelEncoder()
df["purpose"] = le.fit_transform(df["purpose"].astype(str))
df.head()
```

[38]:
```
      credit.policy  purpose  int.rate  installment  log.annual.inc    dti  \
4640              1        4    0.1670       710.03       11.264464  24.60
4694              1        2    0.1913       734.42       11.050890   8.86
3013              1        0    0.1379        67.30       11.332602  13.42
1125              1        2    0.0863       142.33       11.277152   7.53
5611              1        0    0.0788       312.81       10.858999   9.09

      fico  days.with.cr.line  revol.bal  revol.util  inq.last.6mths  \
4640   677        1748.958333          0       49.63               1
4694   677        7350.000000       9881       99.80               0
3013   677        5579.958333      11386       55.00               2
1125   722        5761.000000      20237       32.70               2
5611   762        5791.041667       7386       33.90               0

      delinq.2yrs  pub.rec  not.fully.paid
4640            0        0               1
4694            0        0               0
3013            2        1               0
1125            0        0               0
5611            0        0               0
```

[39]:
```
# Checking for missing values in the data frame
df.isnull().sum()
```

[39]:
```
credit.policy        0
purpose              0
int.rate             0
installment          0
log.annual.inc       0
dti                  0
fico                 0
days.with.cr.line    0
revol.bal            0
revol.util           0
inq.last.6mths       0
delinq.2yrs          0
pub.rec              0
```

```
not.fully.paid      0
dtype: int64
```

[40]:
```
#There are no missing values in the given dataframe.
```

[41]:
```
#Handling outliers and skewness in the numerical variable of our data set.
```

[42]:
```
# Detecting outliers in combined data set
def detect_outlier(feature):
    outliers = []
    data = df[feature]
    mean = np.mean(data)
    std =np.std(data)

    for y in data:
        z_score= (y - mean)/std
        if np.abs(z_score) > 3:
            outliers.append(y)
    print(f"\nOutlier caps for {feature}")
    print('  --95p: {:.1f} / {} values exceed that'.format(data.quantile(.95),
                                                          len([i for i in␣
 ↪data
                                                               if i > data.
 ↪quantile(.95)])))
    print('  --3sd: {:.1f} / {} values exceed that'.format(mean + 3*(std),␣
 ↪len(outliers)))
    print('  --99p: {:.1f} / {} values exceed that'.format(data.quantile(.99),
                                                          len([i for i in␣
 ↪data
                                                               if i > data.
 ↪quantile(.99)])))
```

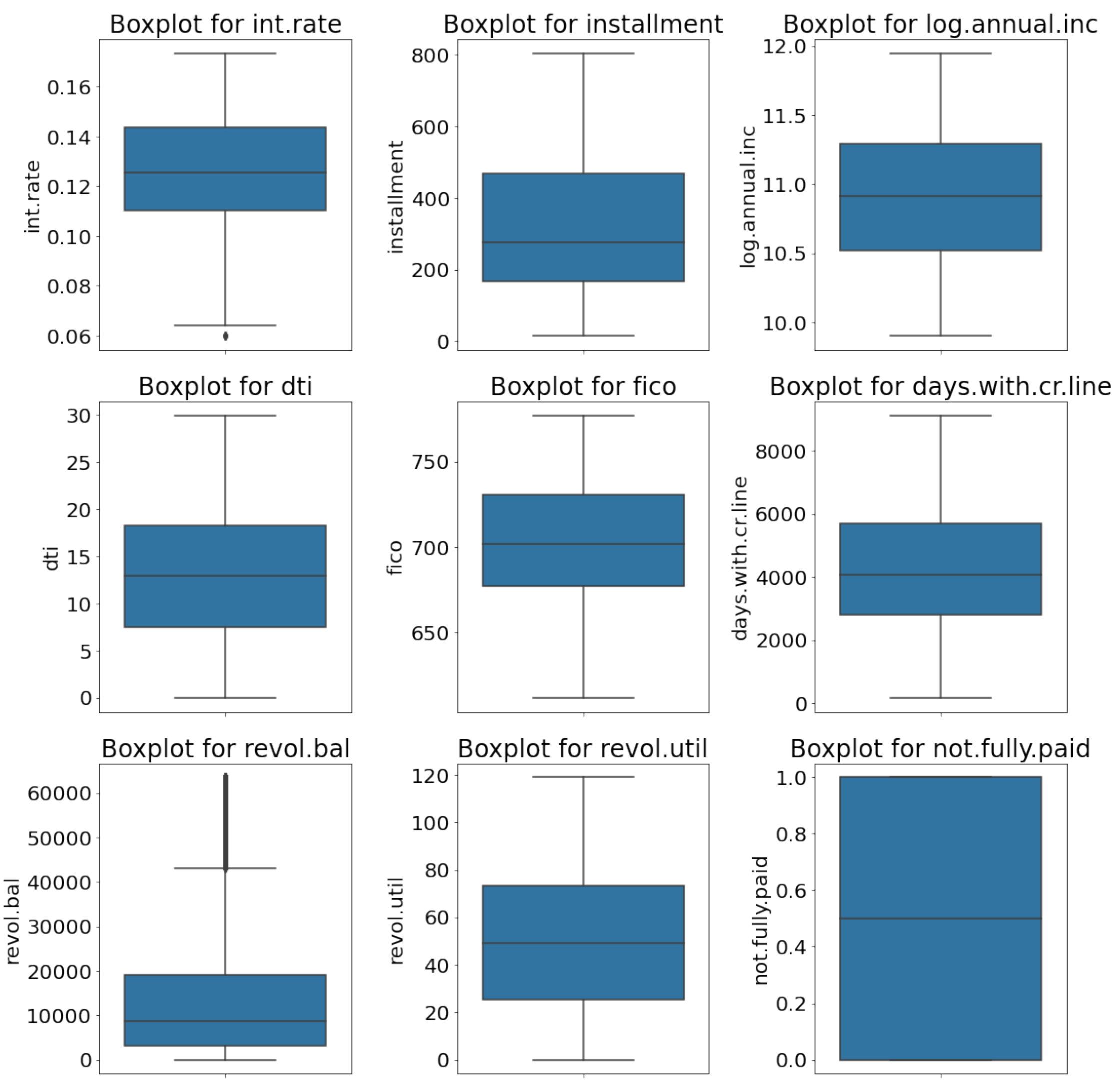
[43]:
```
# Determining what the upperbound should be for continuous features in␣
 ↪dataframe.
for feat in num_data:
    detect_outlier(feat)
```

```
Outlier caps for int.rate
  --95p: 0.2 / 783 values exceed that
  --3sd: 0.2 / 43 values exceed that
  --99p: 0.2 / 129 values exceed that

Outlier caps for installment
  --95p: 804.2 / 801 values exceed that
  --3sd: 975.7 / 0 values exceed that
  --99p: 882.4 / 157 values exceed that
```

```
Outlier caps for log.annual.inc
  --95p: 11.9 / 805 values exceed that
  --3sd: 12.8 / 163 values exceed that
  --99p: 12.6 / 157 values exceed that

Outlier caps for dti
  --95p: 23.9 / 803 values exceed that
  --3sd: 33.6 / 0 values exceed that
  --99p: 26.9 / 153 values exceed that

Outlier caps for fico
  --95p: 777.0 / 645 values exceed that
  --3sd: 816.2 / 18 values exceed that
  --99p: 802.0 / 100 values exceed that

Outlier caps for days.with.cr.line
  --95p: 9120.0 / 803 values exceed that
  --3sd: 11906.6 / 246 values exceed that
  --99p: 12823.2 / 161 values exceed that

Outlier caps for revol.bal
  --95p: 63556.2 / 805 values exceed that
  --3sd: 142103.6 / 280 values exceed that
  --99p: 190575.9 / 161 values exceed that

Outlier caps for revol.util
  --95p: 94.5 / 796 values exceed that
  --3sd: 136.3 / 0 values exceed that
  --99p: 99.2 / 160 values exceed that

Outlier caps for not.fully.paid
  --95p: 1.0 / 0 values exceed that
  --3sd: 2.0 / 0 values exceed that
  --99p: 1.0 / 0 values exceed that
```

[44]:
```
# Capping features in df to remover outliers in numerical features

# Upper bounded outliers
for var in ['int.rate' ,'installment', 'log.annual.inc', 'fico', 'days.with.cr.
↪line', 'revol.bal', 'not.fully.paid']:
    df[var].clip(upper=df[var].quantile(.95), inplace=True)

# Lower and Upper bounded outliers
for var in ['log.annual.inc']:
    df[var].clip(lower = df[var].quantile(.05), upper = df[var].quantile(0.95),␣
↪inplace=True)
```

[45]:
```python
# Checking for the presence of outliers in the numerical data of the dataframe␣
↪again

# Defining subplot grid
numerical_df = df[num_data_features]
plt.figure(figsize = (15, 15))

def num_plot(df, a, var):

    ax = plt.subplot(3, 3, a+1)
    sns.boxplot(y = df[var])
    ax.set_title(f"Boxplot for {var}")
    plt.tight_layout()

for i, col in enumerate(numerical_df):
    num_plot(numerical_df, i, col)
```



[46]:
```
#After capping off outliers, we can see that outliers are now elimited in the
↪numerical variables except revol.bal because it standard deviation is
↪extremely high.
```

[47]:
```
#Checking the skewness in the numerical data of the dataframe
```

[48]:
```
# Checking for skewness in the numerical features
vars_skewed = df[num_data_features].apply(lambda x: skew(x)).
↪sort_values(ascending = False)
vars_skewed
```

[48]:
```
revol.bal            1.689392
installment          0.782681
days.with.cr.line    0.486275
fico                 0.376468
log.annual.inc       0.022196
dti                  0.012952
not.fully.paid       0.000000
revol.util          -0.034845
int.rate            -0.089395
dtype: float64
```

[49]:
```
#Most of the numerical data is positively skewed. Skewness however, has to be
↪correted in features with skewness higher than 0.3.
```

[50]:
```
# Getting numerical features with skewness higher than 0.3.
high_skew = vars_skewed[abs(vars_skewed) > 0.3]
high_skew
```

[50]:
```
revol.bal            1.689392
installment          0.782681
days.with.cr.line    0.486275
fico                 0.376468
dtype: float64
```

[51]:
```
# Correcting the skeness in the numerical features
for feat in high_skew.index:
    df[feat] = np.log1p(df[feat])
```

[52]:
```
# Checking for skewness in the numerical data again for the entire data set
vars_skewed = df[num_data_features].apply(lambda x: skew(x)).
↪sort_values(ascending = False)
vars_skewed
```

[52]:
```
fico                 0.288314
log.annual.inc       0.022196
dti                  0.012952
not.fully.paid       0.000000
revol.util          -0.034845
int.rate            -0.089395
installment         -0.581665
days.with.cr.line   -1.145711
revol.bal           -2.298323
dtype: float64
```

[53]:
```
#The skewness in some features reduces while others have remained the same and␣
↪others have switched to being negatively skewed.
```

[54]:
```
#Handing outliers and skewness in categorical features in our dataframe
```

[55]:
```
# Detecting outliers in categorical data
for feat in cat_data:
    detect_outlier(feat)
```

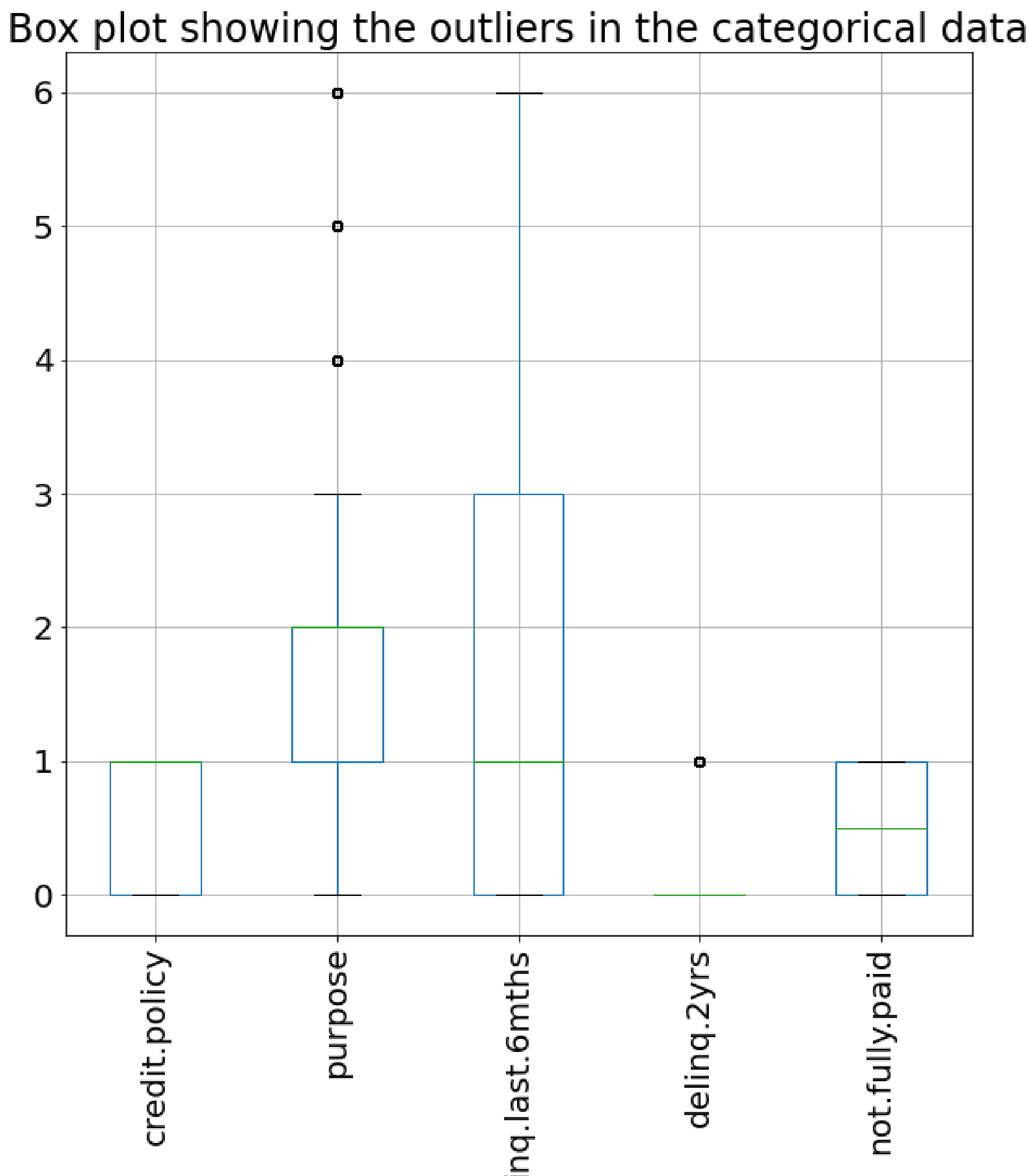
```
Outlier caps for credit.policy
  --95p: 1.0 / 0 values exceed that
  --3sd: 2.1 / 0 values exceed that
  --99p: 1.0 / 0 values exceed that

Outlier caps for purpose
  --95p: 6.0 / 0 values exceed that
  --3sd: 7.3 / 0 values exceed that
  --99p: 6.0 / 0 values exceed that

Outlier caps for inq.last.6mths
  --95p: 6.0 / 795 values exceed that
  --3sd: 9.5 / 243 values exceed that
  --99p: 12.0 / 125 values exceed that

Outlier caps for delinq.2yrs
  --95p: 1.0 / 483 values exceed that
  --3sd: 1.7 / 483 values exceed that
  --99p: 2.0 / 155 values exceed that

Outlier caps for not.fully.paid
  --95p: 1.0 / 0 values exceed that
  --3sd: 2.0 / 0 values exceed that
  --99p: 1.0 / 0 values exceed that
```

[56]:

```
# Capping features in combined_df to remove outliers in categorical features

# Upper bounded outliers
for cat in ['credit.policy', 'purpose', 'inq.last.6mths', 'delinq.2yrs']:
    df[cat].clip(upper=df[cat].quantile(.95), inplace=True)
```

[57]:

```
# Checking for the presence of outliers in the numerical data of the dataframe␣
↪again

# Define subplot grid
categorical_df = df[cat_data_features]

plt.figure(figsize = (10, 10))
categorical_df.boxplot()
plt.xticks(rotation = 90)
plt.title("Box plot showing the outliers in the categorical data")
plt.show()
```



[58]:
```
#We can see that the number of outliers in categorical data reduce␣
↪significantly as compared to the previous case.
```

[59]:
```
#Handling skewness in the categorical data of the dataframe
```

[60]:
```
# Identifying the skewness in the categorical data
for cat in cat_data:
    cat_skewed = df[cat].skew()
```

```
    print(f"{cat}", cat_skewed)
```

```
credit.policy -1.1340861722541922
purpose 0.8668927531072763
inq.last.6mths 1.039386679439289
delinq.2yrs 2.354257610954032
not.fully.paid 0.0
```

[61]:
```
#It can be seen that most of the data is positively skewed.
```

[62]:
```
# Correcting the skewness in categorical features of the dataframe if skewness␣
↪is greater than 0.3.
for cat in cat_data:
    cat_skewed = df[cat].skew()
    if (cat_skewed) > 0.3:
        df[cat] = np.log1p(df[cat])
```

[63]:
```
# Confirming the correction  of the skewness in the categorical data again
for cat in cat_data:
    cat_skewed = df[cat].skew()
    print(f"{cat}", cat_skewed)
```

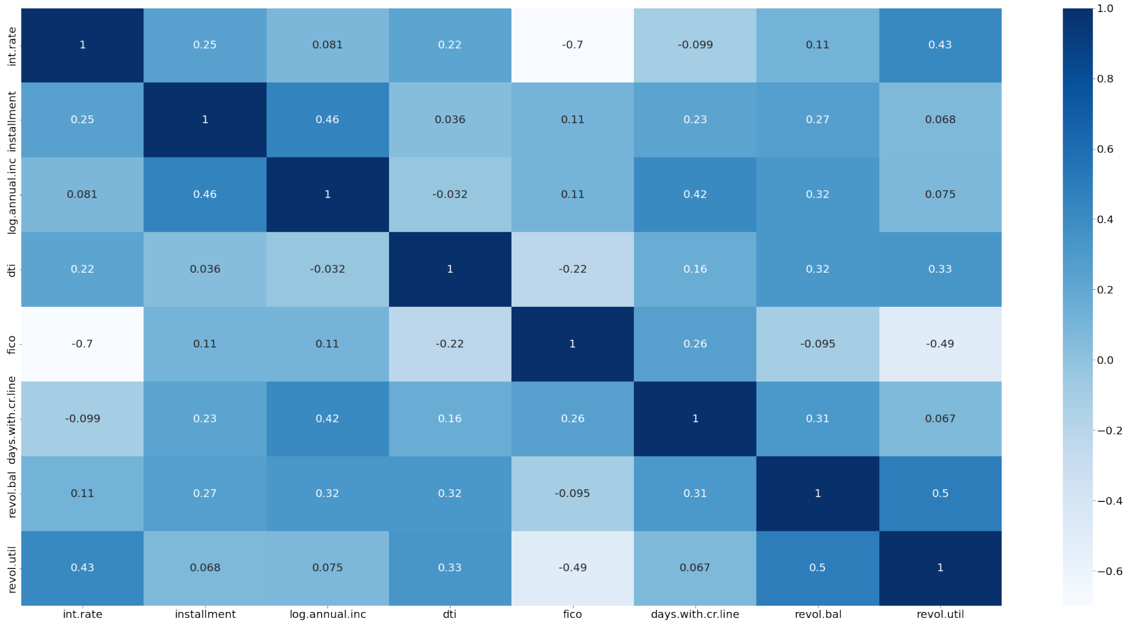
```
credit.policy -1.1340861722541922
purpose -0.2206682619344331
inq.last.6mths 0.23779485819093604
delinq.2yrs 2.3542576109540314
not.fully.paid 0.0
```

[64]:
```
#The skewness in features such as purpose and inq.last.6mths has been reduced␣
↪below 0.3 but other features reduced insignificantly.
```

[65]:
```
#FEATURE ENGINEERING
```

[66]:
```
# Identifying the correlations in the numerical data

# Independent variables
X_num = df[num_data_features]
X_num = X_num.drop(['not.fully.paid'], axis = 1)

# Dependent variable
Y = df[['not.fully.paid']]
```

[67]:
```
# Generating a correlation
matrix = X_num.corr()
plt.figure(figsize = [40, 20])
sns.heatmap(matrix, annot = True, cmap = "Blues");
```




```
[68]: #Strong correlations among features is not highly encouragable because it␣
      ↪results into a noisy signal in the prediction model which cannot give us␣
      ↪clear information about the features that are contributing more to the␣
      ↪predictions. As such, features with strong correlations among themselves␣
      ↪will be eliminated.
```

```
[69]: # Selecting strong correlations among features
      cor_pairs = matrix.unstack()
      sorted_pairs = cor_pairs.sort_values(kind = 'quicksort')
      strong_pairs = sorted_pairs[abs(sorted_pairs) > 0.7]

      print(strong_pairs)
```

```
int.rate           int.rate           1.0
days.with.cr.line  days.with.cr.line  1.0
fico               fico               1.0
dti                dti                1.0
log.annual.inc     log.annual.inc     1.0
installment        installment        1.0
revol.bal          revol.bal          1.0
revol.util         revol.util         1.0
dtype: float64
```

```
[70]: def get_redundant_pairs(df):
          '''Get diagonal and lower triangular pairs of correlation matrix'''
          pairs_to_drop = set()
          cols = df.columns
```

```
    for i in range(0, df.shape[1]):
        for j in range(0, i+1):
            pairs_to_drop.add((cols[i], cols[j]))
    return pairs_to_drop

# Get top pairs
def get_top_abs_correlations(df, n=10):
    corr_list = df.abs().unstack()
    labels_to_drop = get_redundant_pairs(df)
    corr_list = corr_list.drop(labels=labels_to_drop).
↪sort_values(ascending=False)
    return corr_list[0:n]
```

[71]:
```
# Getting top 10 correlation pairs
print('Top 10 correlation pairs:')
get_top_abs_correlations(matrix, 5)
```

```
Top 10 correlation pairs:
```

[71]:
```
int.rate     fico              0.699302
revol.bal    revol.util        0.496650
fico         revol.util        0.494853
installment  log.annual.inc    0.458943
int.rate     revol.util        0.427362
dtype: float64
```

[72]:
```
# Feature Selection
Y = le.fit_transform(Y)
from sklearn.datasets import make_friedman1
from sklearn.svm import SVR
X_num, Y = make_friedman1(n_samples=9578, n_features=8, random_state=42)
estimator = SVR(kernel="linear")
rfe = RFE(estimator, n_features_to_select=5, step=1)
rfe = rfe.fit(X_num, Y.ravel())
```

```
/usr/local/lib/python3.7/site-packages/sklearn/preprocessing/_label.py:115:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

[73]:
```
num_cols = df[num_data_features].drop(['not.fully.paid'], axis = 1)
```

[74]:
```
num_cols = num_cols.columns
num_cols
```

```
[74]: Index(['int.rate', 'installment', 'log.annual.inc', 'dti', 'fico',
       'days.with.cr.line', 'revol.bal', 'revol.util'],
      dtype='object')
```

```
[75]: # Checking the RFE ranking
X_num = pd.DataFrame(X_num, columns = [num_cols])
list(zip(X_num.columns, rfe.support_, rfe.ranking_))
```

```
[75]: [(('int.rate',), True, 1),
 (('installment',), True, 1),
 (('log.annual.inc',), True, 1),
 (('dti',), True, 1),
 (('fico',), True, 1),
 (('days.with.cr.line',), False, 2),
 (('revol.bal',), False, 3),
 (('revol.util',), False, 4)]
```

```
[76]: # Columns selected by RFE
cols = X_num.columns[rfe.support_]
cols
```

```
[76]: MultiIndex([(      'int.rate',),
            (   'installment',),
            ('log.annual.inc',),
            (           'dti',),
            (          'fico',)],
           )
```

```
[77]: # columns not selected by RFE
X_num.columns[~rfe.support_]
```

```
[77]: MultiIndex([('days.with.cr.line',),
            (        'revol.bal',),
            (       'revol.util',)],
           )
```

```
[78]: #The factors that contribute most to whether someone will be default or not are␣
 ↪such as int.rate, installment, log.annual.inc, dti and fico.
```

```
[79]: # Showing the selected numerical features
num_vals = df[['int.rate', 'installment', 'log.annual.inc', 'dti', 'fico']]
num_vals.head()
```

[79]:

	int.rate	installment	log.annual.inc	dti	fico
4640	0.1670	6.566715	11.264464	24.60	6.519147
4694	0.1734	6.600442	11.050890	8.86	6.519147
3013	0.1379	4.223910	11.332602	13.42	6.519147

```
1125    0.0863    4.965150    11.277152   7.53  6.583409
5611    0.0788    5.748788    10.858999   9.09  6.637258
```

[80]:
```python
#Selecting the best features in categorical data
```

[81]:
```python
# Collecting the categorical data
cat_vars = df[cat_data_features].drop(['not.fully.paid'], axis = 1)
cat_vars
```

[81]:

	credit.policy	purpose	inq.last.6mths	delinq.2yrs
4640	1	1.609438	0.693147	0.000000
4694	1	1.098612	0.000000	0.000000
3013	1	0.000000	1.098612	0.693147
1125	1	1.098612	1.098612	0.000000
5611	1	0.000000	0.000000	0.000000
…	…	…	…	…
8271	0	1.098612	0.000000	0.000000
5519	1	0.693147	0.693147	0.000000
4914	1	0.693147	0.693147	0.000000
878	1	0.000000	1.386294	0.000000
6533	1	0.000000	0.000000	0.000000

[16090 rows x 4 columns]

[82]:
```python
# Performing the chi test and determine the f score and the p value
f_p_values = chi2(cat_vars, df['not.fully.paid'])
f_p_values
```

[82]:
```
(array([158.29859319,   7.90392614, 272.83520787,   1.57750738]),
 array([2.66316023e-36, 4.93276160e-03, 2.73526732e-61, 2.09120101e-01]))
```

[83]:
```python
#Chi-square test is used to find F-score and p-values for categorical features.
#So in this case the first array is for F score and the second array is for␣
 ↪p-values.
#The higher the value of the F score is the more important the feature and the␣
 ↪smaller the value of the p-value the more important will be the feature.
#A p-value less 0.05 indicates that the feature is important.
```

[84]:
```python
# Representing the p values in list form
p_values = pd.Series(f_p_values[1])
p_values.index = cat_vars.columns
p_values
```

[84]:
```
credit.policy     2.663160e-36
purpose           4.932762e-03
inq.last.6mths    2.735267e-61
delinq.2yrs       2.091201e-01
```

```
dtype: float64
```

[85]:
```
# Sorting the p values in ascending order
p_values.sort_values(ascending = True)
```

[85]:
```
inq.last.6mths    2.735267e-61
credit.policy     2.663160e-36
purpose           4.932762e-03
delinq.2yrs       2.091201e-01
dtype: float64
```

[86]:
```
#Categorical features such as inq.last.6mths, credit.policy, and purpose have a␣
↪p-value that is less than 0.05.
#These will therefore be taken into consideration for model training as they␣
↪are seen to mostly influence wether a client will default or not.
```

[87]:
```
#DATA TRAINING
```

[88]:
```
# Dividing the data into features and target variables
X = df[['int.rate', 'installment', 'log.annual.inc', 'dti', 'fico', 'inq.last.
↪6mths',
        'credit.policy', 'purpose']]
y = df['not.fully.paid']
```

[89]:
```
# Spliting the data into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 42)
```

[90]:
```
# Scale the data
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

[91]:
```
model  = keras.Sequential(
    [
        keras.layers.Dense(
        256, activation="relu", input_shape=[8]),
        keras.layers.Dense(256, activation="relu"),
        keras.layers.Dropout(0.3),
        keras.layers.Dense(256, activation="relu"),
        keras.layers.Dropout(0.3),
        keras.layers.Dense(1, activation="sigmoid"),
    ]
)
model.summary()
```

```
Model: "sequential"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               2304

 dense_1 (Dense)             (None, 256)               65792

 dropout (Dropout)           (None, 256)               0

 dense_2 (Dense)             (None, 256)               65792

 dropout_1 (Dropout)         (None, 256)               0

 dense_3 (Dense)             (None, 1)                 257

=================================================================
Total params: 134,145
Trainable params: 134,145
Non-trainable params: 0
_________________________________________________________________
```

[92]:
```
model.compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics =␣
 ↪['binary_accuracy'])
```

[93]:
```
early_stopping = keras.callbacks.EarlyStopping(patience=10, min_delta=0.001,␣
 ↪restore_best_weights=True)
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    batch_size=256,
    epochs=1000,
    callbacks=[early_stopping],
    verbose=1,
)
```

```
Epoch 1/1000
51/51 [==============================] - 5s 10ms/step - loss: 0.6521 -
binary_accuracy: 0.6108 - val_loss: 0.6455 - val_binary_accuracy: 0.6206
Epoch 2/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.6419 -
binary_accuracy: 0.6255 - val_loss: 0.6367 - val_binary_accuracy: 0.6342
Epoch 3/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.6373 -
binary_accuracy: 0.6325 - val_loss: 0.6361 - val_binary_accuracy: 0.6314
Epoch 4/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.6340 -
binary_accuracy: 0.6349 - val_loss: 0.6323 - val_binary_accuracy: 0.6392
Epoch 5/1000
```

```
51/51 [==============================] - 0s 7ms/step - loss: 0.6277 -
binary_accuracy: 0.6408 - val_loss: 0.6309 - val_binary_accuracy: 0.6383
Epoch 6/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.6241 -
binary_accuracy: 0.6461 - val_loss: 0.6270 - val_binary_accuracy: 0.6401
Epoch 7/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.6196 -
binary_accuracy: 0.6468 - val_loss: 0.6216 - val_binary_accuracy: 0.6457
Epoch 8/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.6134 -
binary_accuracy: 0.6546 - val_loss: 0.6205 - val_binary_accuracy: 0.6482
Epoch 9/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.6074 -
binary_accuracy: 0.6614 - val_loss: 0.6150 - val_binary_accuracy: 0.6523
Epoch 10/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.6001 -
binary_accuracy: 0.6697 - val_loss: 0.6058 - val_binary_accuracy: 0.6644
Epoch 11/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.5956 -
binary_accuracy: 0.6704 - val_loss: 0.6089 - val_binary_accuracy: 0.6513
Epoch 12/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.5905 -
binary_accuracy: 0.6748 - val_loss: 0.6020 - val_binary_accuracy: 0.6591
Epoch 13/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.5801 -
binary_accuracy: 0.6881 - val_loss: 0.6029 - val_binary_accuracy: 0.6532
Epoch 14/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.5771 -
binary_accuracy: 0.6835 - val_loss: 0.5922 - val_binary_accuracy: 0.6712
Epoch 15/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.5695 -
binary_accuracy: 0.6923 - val_loss: 0.5861 - val_binary_accuracy: 0.6837
Epoch 16/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.5640 -
binary_accuracy: 0.6992 - val_loss: 0.5887 - val_binary_accuracy: 0.6759
Epoch 17/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.5566 -
binary_accuracy: 0.7043 - val_loss: 0.5771 - val_binary_accuracy: 0.6905
Epoch 18/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.5541 -
binary_accuracy: 0.7047 - val_loss: 0.5804 - val_binary_accuracy: 0.6833
Epoch 19/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.5475 -
binary_accuracy: 0.7157 - val_loss: 0.5698 - val_binary_accuracy: 0.6865
Epoch 20/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.5389 -
binary_accuracy: 0.7185 - val_loss: 0.5620 - val_binary_accuracy: 0.6942
Epoch 21/1000
```

```
51/51 [==============================] - 0s 8ms/step - loss: 0.5345 -
binary_accuracy: 0.7186 - val_loss: 0.5597 - val_binary_accuracy: 0.7054
Epoch 22/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.5258 -
binary_accuracy: 0.7337 - val_loss: 0.5561 - val_binary_accuracy: 0.7020
Epoch 23/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.5160 -
binary_accuracy: 0.7415 - val_loss: 0.5485 - val_binary_accuracy: 0.7185
Epoch 24/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.5121 -
binary_accuracy: 0.7448 - val_loss: 0.5404 - val_binary_accuracy: 0.7172
Epoch 25/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.5058 -
binary_accuracy: 0.7451 - val_loss: 0.5382 - val_binary_accuracy: 0.7213
Epoch 26/1000
51/51 [==============================] - 0s 9ms/step - loss: 0.5005 -
binary_accuracy: 0.7449 - val_loss: 0.5338 - val_binary_accuracy: 0.7225
Epoch 27/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4927 -
binary_accuracy: 0.7571 - val_loss: 0.5298 - val_binary_accuracy: 0.7306
Epoch 28/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.4847 -
binary_accuracy: 0.7613 - val_loss: 0.5189 - val_binary_accuracy: 0.7340
Epoch 29/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4797 -
binary_accuracy: 0.7631 - val_loss: 0.5158 - val_binary_accuracy: 0.7359
Epoch 30/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4746 -
binary_accuracy: 0.7671 - val_loss: 0.5258 - val_binary_accuracy: 0.7293
Epoch 31/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.4662 -
binary_accuracy: 0.7725 - val_loss: 0.5052 - val_binary_accuracy: 0.7439
Epoch 32/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4608 -
binary_accuracy: 0.7725 - val_loss: 0.5182 - val_binary_accuracy: 0.7464
Epoch 33/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4608 -
binary_accuracy: 0.7735 - val_loss: 0.5165 - val_binary_accuracy: 0.7455
Epoch 34/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.4498 -
binary_accuracy: 0.7839 - val_loss: 0.4930 - val_binary_accuracy: 0.7669
Epoch 35/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.4433 -
binary_accuracy: 0.7857 - val_loss: 0.4876 - val_binary_accuracy: 0.7585
Epoch 36/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4382 -
binary_accuracy: 0.7904 - val_loss: 0.4856 - val_binary_accuracy: 0.7598
Epoch 37/1000
```

```
51/51 [==============================] - 0s 7ms/step - loss: 0.4284 -
binary_accuracy: 0.7967 - val_loss: 0.4868 - val_binary_accuracy: 0.7676
Epoch 38/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.4212 -
binary_accuracy: 0.7999 - val_loss: 0.4789 - val_binary_accuracy: 0.7676
Epoch 39/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.4193 -
binary_accuracy: 0.7995 - val_loss: 0.4764 - val_binary_accuracy: 0.7707
Epoch 40/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4086 -
binary_accuracy: 0.8063 - val_loss: 0.4644 - val_binary_accuracy: 0.7766
Epoch 41/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.4097 -
binary_accuracy: 0.8078 - val_loss: 0.4658 - val_binary_accuracy: 0.7722
Epoch 42/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.3999 -
binary_accuracy: 0.8145 - val_loss: 0.4554 - val_binary_accuracy: 0.7781
Epoch 43/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3959 -
binary_accuracy: 0.8142 - val_loss: 0.4558 - val_binary_accuracy: 0.7906
Epoch 44/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3923 -
binary_accuracy: 0.8147 - val_loss: 0.4564 - val_binary_accuracy: 0.7865
Epoch 45/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3873 -
binary_accuracy: 0.8198 - val_loss: 0.4501 - val_binary_accuracy: 0.7884
Epoch 46/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3826 -
binary_accuracy: 0.8252 - val_loss: 0.4402 - val_binary_accuracy: 0.7955
Epoch 47/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3732 -
binary_accuracy: 0.8326 - val_loss: 0.4340 - val_binary_accuracy: 0.8024
Epoch 48/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3670 -
binary_accuracy: 0.8315 - val_loss: 0.4425 - val_binary_accuracy: 0.7955
Epoch 49/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.3596 -
binary_accuracy: 0.8383 - val_loss: 0.4398 - val_binary_accuracy: 0.8005
Epoch 50/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3621 -
binary_accuracy: 0.8329 - val_loss: 0.4234 - val_binary_accuracy: 0.8052
Epoch 51/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3552 -
binary_accuracy: 0.8400 - val_loss: 0.4343 - val_binary_accuracy: 0.8055
Epoch 52/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3450 -
binary_accuracy: 0.8442 - val_loss: 0.4118 - val_binary_accuracy: 0.8160
Epoch 53/1000
```

```
51/51 [==============================] - 0s 8ms/step - loss: 0.3409 -
binary_accuracy: 0.8492 - val_loss: 0.4192 - val_binary_accuracy: 0.8160
Epoch 54/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.3296 -
binary_accuracy: 0.8581 - val_loss: 0.4148 - val_binary_accuracy: 0.8176
Epoch 55/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3322 -
binary_accuracy: 0.8549 - val_loss: 0.4063 - val_binary_accuracy: 0.8195
Epoch 56/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3317 -
binary_accuracy: 0.8523 - val_loss: 0.4200 - val_binary_accuracy: 0.8210
Epoch 57/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.3208 -
binary_accuracy: 0.8563 - val_loss: 0.4088 - val_binary_accuracy: 0.8195
Epoch 58/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3155 -
binary_accuracy: 0.8592 - val_loss: 0.4056 - val_binary_accuracy: 0.8226
Epoch 59/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3172 -
binary_accuracy: 0.8623 - val_loss: 0.4070 - val_binary_accuracy: 0.8266
Epoch 60/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.3149 -
binary_accuracy: 0.8619 - val_loss: 0.4062 - val_binary_accuracy: 0.8337
Epoch 61/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.3072 -
binary_accuracy: 0.8633 - val_loss: 0.3884 - val_binary_accuracy: 0.8337
Epoch 62/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.3057 -
binary_accuracy: 0.8637 - val_loss: 0.3902 - val_binary_accuracy: 0.8437
Epoch 63/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2986 -
binary_accuracy: 0.8703 - val_loss: 0.4004 - val_binary_accuracy: 0.8325
Epoch 64/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2977 -
binary_accuracy: 0.8699 - val_loss: 0.3986 - val_binary_accuracy: 0.8319
Epoch 65/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2867 -
binary_accuracy: 0.8769 - val_loss: 0.3852 - val_binary_accuracy: 0.8359
Epoch 66/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2947 -
binary_accuracy: 0.8695 - val_loss: 0.3802 - val_binary_accuracy: 0.8443
Epoch 67/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2840 -
binary_accuracy: 0.8786 - val_loss: 0.3944 - val_binary_accuracy: 0.8393
Epoch 68/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2838 -
binary_accuracy: 0.8783 - val_loss: 0.3965 - val_binary_accuracy: 0.8378
Epoch 69/1000
```

```
51/51 [==============================] - 0s 8ms/step - loss: 0.2746 -
binary_accuracy: 0.8815 - val_loss: 0.3722 - val_binary_accuracy: 0.8431
Epoch 70/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2704 -
binary_accuracy: 0.8846 - val_loss: 0.3771 - val_binary_accuracy: 0.8527
Epoch 71/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2732 -
binary_accuracy: 0.8814 - val_loss: 0.3880 - val_binary_accuracy: 0.8409
Epoch 72/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2677 -
binary_accuracy: 0.8864 - val_loss: 0.3601 - val_binary_accuracy: 0.8577
Epoch 73/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2721 -
binary_accuracy: 0.8823 - val_loss: 0.3927 - val_binary_accuracy: 0.8384
Epoch 74/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2580 -
binary_accuracy: 0.8891 - val_loss: 0.3856 - val_binary_accuracy: 0.8440
Epoch 75/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2599 -
binary_accuracy: 0.8905 - val_loss: 0.3580 - val_binary_accuracy: 0.8602
Epoch 76/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2551 -
binary_accuracy: 0.8930 - val_loss: 0.3763 - val_binary_accuracy: 0.8484
Epoch 77/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2577 -
binary_accuracy: 0.8895 - val_loss: 0.3585 - val_binary_accuracy: 0.8549
Epoch 78/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2520 -
binary_accuracy: 0.8946 - val_loss: 0.3726 - val_binary_accuracy: 0.8502
Epoch 79/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2422 -
binary_accuracy: 0.8978 - val_loss: 0.3521 - val_binary_accuracy: 0.8586
Epoch 80/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2469 -
binary_accuracy: 0.8979 - val_loss: 0.3756 - val_binary_accuracy: 0.8583
Epoch 81/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2422 -
binary_accuracy: 0.8983 - val_loss: 0.3730 - val_binary_accuracy: 0.8630
Epoch 82/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2378 -
binary_accuracy: 0.9027 - val_loss: 0.3573 - val_binary_accuracy: 0.8567
Epoch 83/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2331 -
binary_accuracy: 0.9023 - val_loss: 0.3584 - val_binary_accuracy: 0.8633
Epoch 84/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2334 -
binary_accuracy: 0.9054 - val_loss: 0.3728 - val_binary_accuracy: 0.8567
Epoch 85/1000
```

```
51/51 [==============================] - 0s 8ms/step - loss: 0.2338 -
binary_accuracy: 0.9040 - val_loss: 0.3669 - val_binary_accuracy: 0.8617
Epoch 86/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2280 -
binary_accuracy: 0.9072 - val_loss: 0.3678 - val_binary_accuracy: 0.8608
Epoch 87/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2280 -
binary_accuracy: 0.9026 - val_loss: 0.3490 - val_binary_accuracy: 0.8713
Epoch 88/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2214 -
binary_accuracy: 0.9105 - val_loss: 0.3565 - val_binary_accuracy: 0.8673
Epoch 89/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2252 -
binary_accuracy: 0.9066 - val_loss: 0.3667 - val_binary_accuracy: 0.8599
Epoch 90/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2237 -
binary_accuracy: 0.9058 - val_loss: 0.3495 - val_binary_accuracy: 0.8695
Epoch 91/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2185 -
binary_accuracy: 0.9110 - val_loss: 0.3593 - val_binary_accuracy: 0.8617
Epoch 92/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2217 -
binary_accuracy: 0.9080 - val_loss: 0.3374 - val_binary_accuracy: 0.8686
Epoch 93/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2136 -
binary_accuracy: 0.9140 - val_loss: 0.3544 - val_binary_accuracy: 0.8682
Epoch 94/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2134 -
binary_accuracy: 0.9117 - val_loss: 0.3526 - val_binary_accuracy: 0.8661
Epoch 95/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2035 -
binary_accuracy: 0.9163 - val_loss: 0.3580 - val_binary_accuracy: 0.8713
Epoch 96/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2090 -
binary_accuracy: 0.9152 - val_loss: 0.3515 - val_binary_accuracy: 0.8748
Epoch 97/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2050 -
binary_accuracy: 0.9170 - val_loss: 0.3586 - val_binary_accuracy: 0.8707
Epoch 98/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2116 -
binary_accuracy: 0.9145 - val_loss: 0.3354 - val_binary_accuracy: 0.8791
Epoch 99/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.2067 -
binary_accuracy: 0.9117 - val_loss: 0.3531 - val_binary_accuracy: 0.8686
Epoch 100/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.2009 -
binary_accuracy: 0.9173 - val_loss: 0.3393 - val_binary_accuracy: 0.8757
Epoch 101/1000
```

```
51/51 [==============================] - 0s 7ms/step - loss: 0.2011 -
binary_accuracy: 0.9171 - val_loss: 0.3504 - val_binary_accuracy: 0.8723
Epoch 102/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1948 -
binary_accuracy: 0.9213 - val_loss: 0.3590 - val_binary_accuracy: 0.8735
Epoch 103/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1912 -
binary_accuracy: 0.9229 - val_loss: 0.3359 - val_binary_accuracy: 0.8816
Epoch 104/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1869 -
binary_accuracy: 0.9252 - val_loss: 0.3408 - val_binary_accuracy: 0.8773
Epoch 105/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1866 -
binary_accuracy: 0.9243 - val_loss: 0.3284 - val_binary_accuracy: 0.8782
Epoch 106/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1879 -
binary_accuracy: 0.9224 - val_loss: 0.3551 - val_binary_accuracy: 0.8692
Epoch 107/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1951 -
binary_accuracy: 0.9201 - val_loss: 0.3676 - val_binary_accuracy: 0.8745
Epoch 108/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1868 -
binary_accuracy: 0.9246 - val_loss: 0.3536 - val_binary_accuracy: 0.8766
Epoch 109/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1848 -
binary_accuracy: 0.9281 - val_loss: 0.3314 - val_binary_accuracy: 0.8847
Epoch 110/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1843 -
binary_accuracy: 0.9258 - val_loss: 0.3214 - val_binary_accuracy: 0.8847
Epoch 111/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1863 -
binary_accuracy: 0.9231 - val_loss: 0.3569 - val_binary_accuracy: 0.8738
Epoch 112/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1843 -
binary_accuracy: 0.9284 - val_loss: 0.3283 - val_binary_accuracy: 0.8838
Epoch 113/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1749 -
binary_accuracy: 0.9311 - val_loss: 0.3446 - val_binary_accuracy: 0.8832
Epoch 114/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1867 -
binary_accuracy: 0.9260 - val_loss: 0.3424 - val_binary_accuracy: 0.8769
Epoch 115/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1734 -
binary_accuracy: 0.9339 - val_loss: 0.3450 - val_binary_accuracy: 0.8810
Epoch 116/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1729 -
binary_accuracy: 0.9322 - val_loss: 0.3446 - val_binary_accuracy: 0.8866
Epoch 117/1000
```

```
51/51 [==============================] - 0s 7ms/step - loss: 0.1659 -
binary_accuracy: 0.9319 - val_loss: 0.3196 - val_binary_accuracy: 0.8934
Epoch 118/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1605 -
binary_accuracy: 0.9368 - val_loss: 0.3378 - val_binary_accuracy: 0.8915
Epoch 119/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1658 -
binary_accuracy: 0.9351 - val_loss: 0.3616 - val_binary_accuracy: 0.8832
Epoch 120/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1647 -
binary_accuracy: 0.9340 - val_loss: 0.3750 - val_binary_accuracy: 0.8738
Epoch 121/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1710 -
binary_accuracy: 0.9331 - val_loss: 0.3691 - val_binary_accuracy: 0.8745
Epoch 122/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1679 -
binary_accuracy: 0.9333 - val_loss: 0.3532 - val_binary_accuracy: 0.8847
Epoch 123/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1643 -
binary_accuracy: 0.9316 - val_loss: 0.3225 - val_binary_accuracy: 0.8894
Epoch 124/1000
51/51 [==============================] - 0s 8ms/step - loss: 0.1649 -
binary_accuracy: 0.9330 - val_loss: 0.3472 - val_binary_accuracy: 0.8828
Epoch 125/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1560 -
binary_accuracy: 0.9387 - val_loss: 0.3278 - val_binary_accuracy: 0.8891
Epoch 126/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1544 -
binary_accuracy: 0.9380 - val_loss: 0.3212 - val_binary_accuracy: 0.8922
Epoch 127/1000
51/51 [==============================] - 0s 7ms/step - loss: 0.1602 -
binary_accuracy: 0.9373 - val_loss: 0.3439 - val_binary_accuracy: 0.8838
```

[94]:
```python
predictions =(model.predict(X_test)>0.5).astype("int32")

predictions
```

[94]:
```
array([[1],
       [0],
       [1],
       …,
       [1],
       [0],
       [0]], dtype=int32)
```

[95]:
```python
from sklearn.metrics import classification_report, confusion_matrix,␣
↪accuracy_score
```

```
accuracy_score(y_test, predictions)
```

[95]: 0.8934120571783717

[96]:
```
print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.94      0.84      0.89      1601
           1       0.85      0.95      0.90      1617

    accuracy                           0.89      3218
   macro avg       0.90      0.89      0.89      3218
weighted avg       0.90      0.89      0.89      3218
```

[ ]:
```
#It can be seen that our model has an accuracy of 89.34 which is quite good
```