CC_Churn_Predict_CODE

January 16, 2021

1 This project is to predict credit card churning using several Machine Learning Methods and Deep Learning

The purpose of this Project is to classify the churn of Credit Card customer given several parameters as shown in Column Name. The data was obtained from Kaggle, the link below. As a company who issue credit card, it is important that we know whether customer will be able to continue their transactions (not defaulted) and keep using our service. The pre-registered features such as Gender, Income Category, Total Relationship Count, can be an indicator whether a customer will be able to sustain our service. However, here, i will use all the parameters which are available for the customer that has already registered to see what type of customer will sustain, therefore once we spot a customer with such characteristic, we can reach to them, send them offer to keep using our services.

The data was obtained from Kaggle: https://www.kaggle.com/sakshigoyal7/credit-card-customers

Column Name: CLIENTNUM = Client number. Unique identifier for the customer holding the account

Attrition_Flag = Internal event (customer activity) variable - if the account is closed then 1 else 0 (THIS IS THE TARGET PARAMETER)

Customer_Age

Gender = Demographic variable - M=Male, F=Female

Dependent_count = Demographic variable - Number of dependents

Education_Level

Marital_Status

Income_Category = Demographic variable - Annual Income Category of the account holder (< $40K, $40K - 60K, $60K-$80K, $80K-$120K, > $120K, Unknown)

Card_Category = Product Variable - Type of Card (Blue, Silver, Gold, Platinum)

Months_on_book = Period of relationship with bank

Total_Relationship_Count = Total no. of products held by the customer

Months_Inactive_12_mon = No. of months inactive in the last 12 months

Contacts_Count_12_mon = No. of Contacts in the last 12 months

Credit_Limit

Total_Revolving_Bal = Total Revolving Balance on the Credit Card

Avg_Open_To_Buy = Open to Buy Credit Line (Average of last 12 months)

Total_Amt_Chng_Q4_Q1 = Change in Transaction Amount (Q4 over Q1)

Total_Trans_Amt = Total Transaction Amount (Last 12 months)

Total_Trans_Ct = Total Transaction Count (Last 12 months)

Total_Ct_Chng_Q4_Q1 = Change in Transaction Count (Q4 over Q1)

Avg_Utilization_Ratio = Average Card Utilization Ratio

[2]:
```
import pandas as pd
import numpy as np
import plotly.express as px
import copy
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from sklearn.model_selection import train_test_split
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score
from sklearn import metrics, svm
import plotly
import os
```

1.1 Lets Load the data

The data has been downloaded and resaved in the ETL, and here it is loaded from the Churn-Data.csv

[3]:
```
df = pd.read_csv('ChurnData.csv')
```

[4]:
```
df.drop(columns=df.columns[-2:], inplace=True)
df.drop('CLIENTNUM',axis=1,inplace=True)
df.head(3)
```

[4]:
```
      Attrition_Flag  Customer_Age Gender  Dependent_count Education_Level  \
0  Existing Customer            45      M                3     High School
1  Existing Customer            49      F                5        Graduate
2  Existing Customer            51      M                3        Graduate

  Marital_Status Income_Category Card_Category  Months_on_book  \
0        Married     $60K - $80K          Blue              39
```

```
1         Single  Less than $40K          Blue                44
2        Married    $80K - $120K          Blue                36

   Total_Relationship_Count  Months_Inactive_12_mon  Contacts_Count_12_mon  \
0                         5                       1                      3
1                         6                       1                      2
2                         4                       1                      0

   Credit_Limit  Total_Revolving_Bal  Avg_Open_To_Buy  Total_Amt_Chng_Q4_Q1  \
0       12691.0                  777          11914.0                 1.335
1        8256.0                  864           7392.0                 1.541
2        3418.0                    0           3418.0                 2.594

   Total_Trans_Amt  Total_Trans_Ct  Total_Ct_Chng_Q4_Q1  Avg_Utilization_Ratio
0             1144              42                1.625                  0.061
1             1291              33                3.714                  0.105
2             1887              20                2.333                  0.000
```

[5]:
```
df.shape
```

[5]: (10127, 20)

2 A. Exploratory Data Analysis

Exploratory data analysis is important to get the initial idea about what we might find in the data, before actually performing some Machine Learning or Deep Learning modeling.

2.1 Select The categorical data

Here, I would like to explore the categorical data (using pie chart) to see the proportion of each category. The pie_draw function is defined below, to make things easier.

[6]:
```
categorical=df.select_dtypes(exclude=['int64','float64']).columns
```

[11]:
```
print(*categorical)
```

```
Attrition_Flag Gender Education_Level Marital_Status Income_Category
Card_Category
```

The code below is to draw Pie_chart

[187]:
```
def pie_draw(sizes, Title, labels, explode, filename):
    fig1, ax1 = plt.subplots()
    ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
            shadow=True, startangle=90, textprops={'fontsize': 13})
    # Equal aspect ratio ensures that pie is drawn as a circle
    ax1.axis('equal')
    ax1.set_title(Title, fontsize = 14)
```

```
    plt.tight_layout()
    plt.savefig(filename)
    plt.show()
```

2.1.1 1. The Attrition Flag

[14]:
```
df['Attrition_Flag'].value_counts()
```

[14]:
```
Existing Customer    8500
Attrited Customer    1627
Name: Attrition_Flag, dtype: int64
```

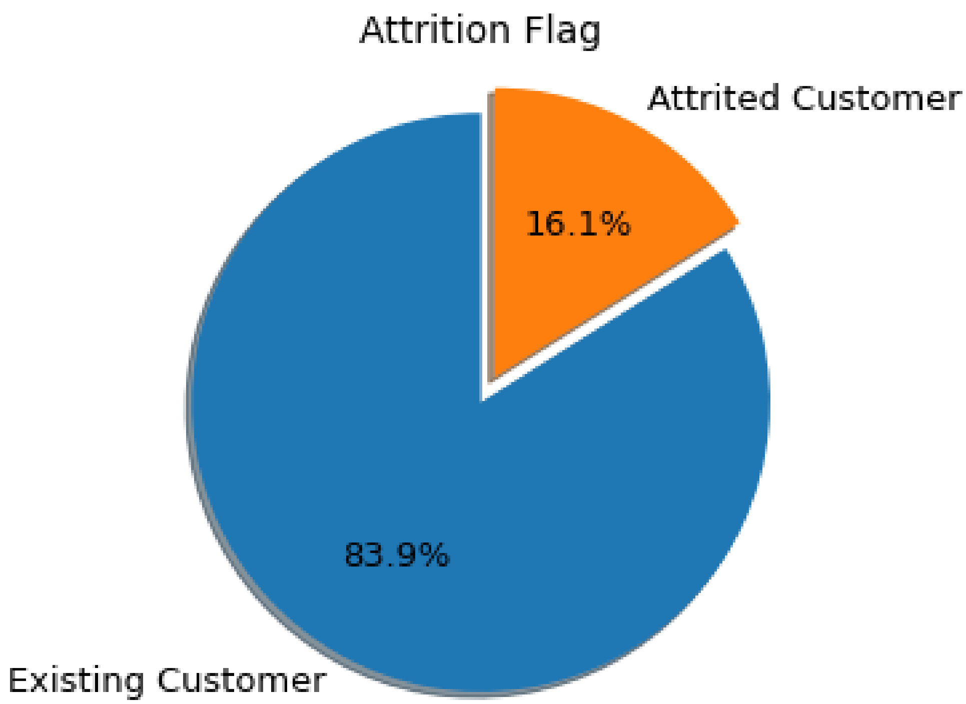
[18]:
```
print("ratio between Existing/Attrited= ", round(df['Attrition_Flag'].
 ↪value_counts()[0]/df['Attrition_Flag'].value_counts()[1],2))
```

```
ratio between Existing/Attrited=  5.22
```

We can see that the Existing Customer data are about 5.2 more than the Attrited

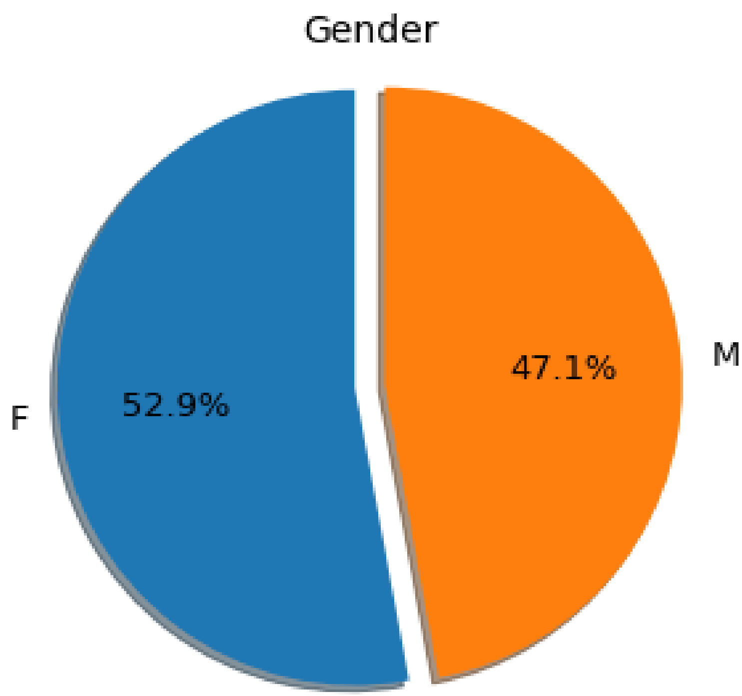
The Pie Chart

[242]:
```
# Pie chart
labels = [i for i in df['Attrition_Flag'].unique()]
sizes = [i for i in df['Attrition_Flag'].value_counts()]

explode = (0, 0.1)

pie_draw(sizes, "Attrition Flag", labels, explode,"figures/PC_Attrition.jpg")
```



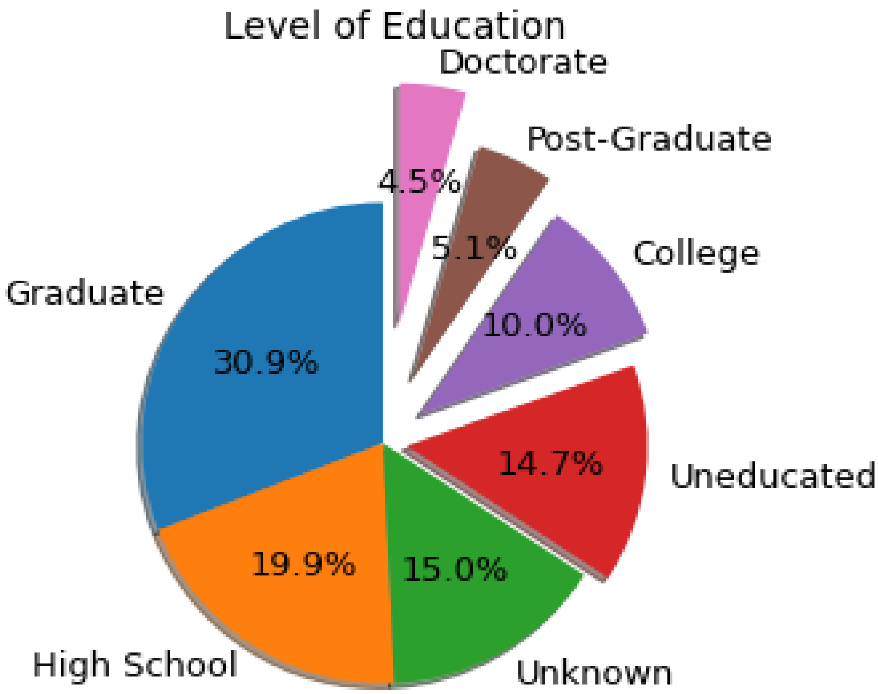
2.1.2 2. Gender

[243]:
```
# Pie chart
labels = [i for i in df['Gender'].value_counts().index]
sizes = [i for i in df['Gender'].value_counts()]

explode = (0, 0.1)

pie_draw(sizes, "Gender", labels, explode, "figures/PC_Gender.jpg")
```



2.1.3 3. Education Level

[244]:
```
# Pie chart
labels = [i for i in df['Education_Level'].value_counts().index]
sizes = [i for i in df['Education_Level'].value_counts()]

explode = (0, 0, 0, 0.1, 0.2, 0.3, 0.5)

pie_draw(sizes, "Level of Education", labels, explode, "figures/PC_Education.
↪jpg")
```

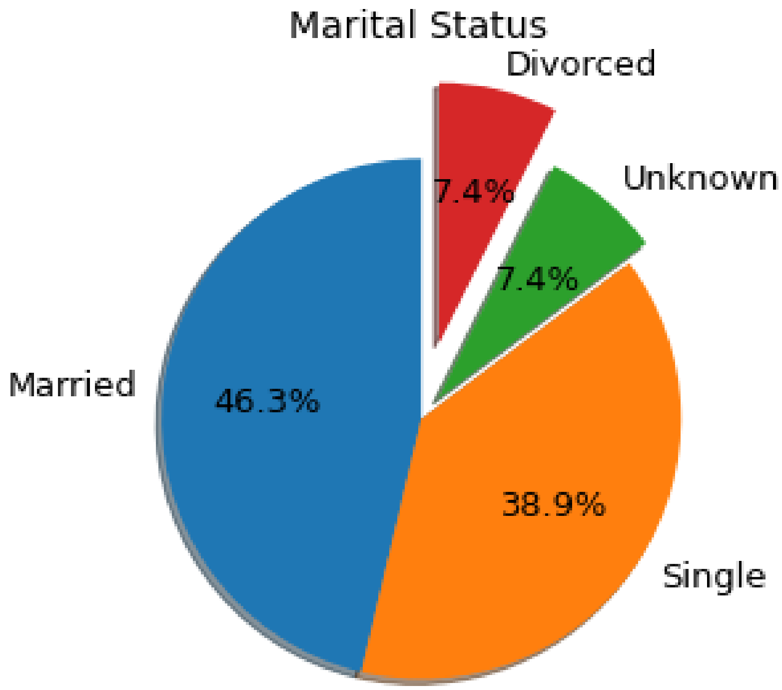


2.1.4 4. Marital Status

[30]:
```
df['Marital_Status'].value_counts()
```

[30]:
```
Married     4687
Single      3943
Unknown      749
Divorced     748
Name: Marital_Status, dtype: int64
```

[245]:
```
# Pie chart
labels = [i for i in df['Marital_Status'].value_counts().index]
sizes = [i for i in df['Marital_Status'].value_counts()]

explode = (0, 0,0.1,0.3)

pie_draw(sizes, "Marital Status", labels, explode, "figures/PC_Marital.jpg")
```

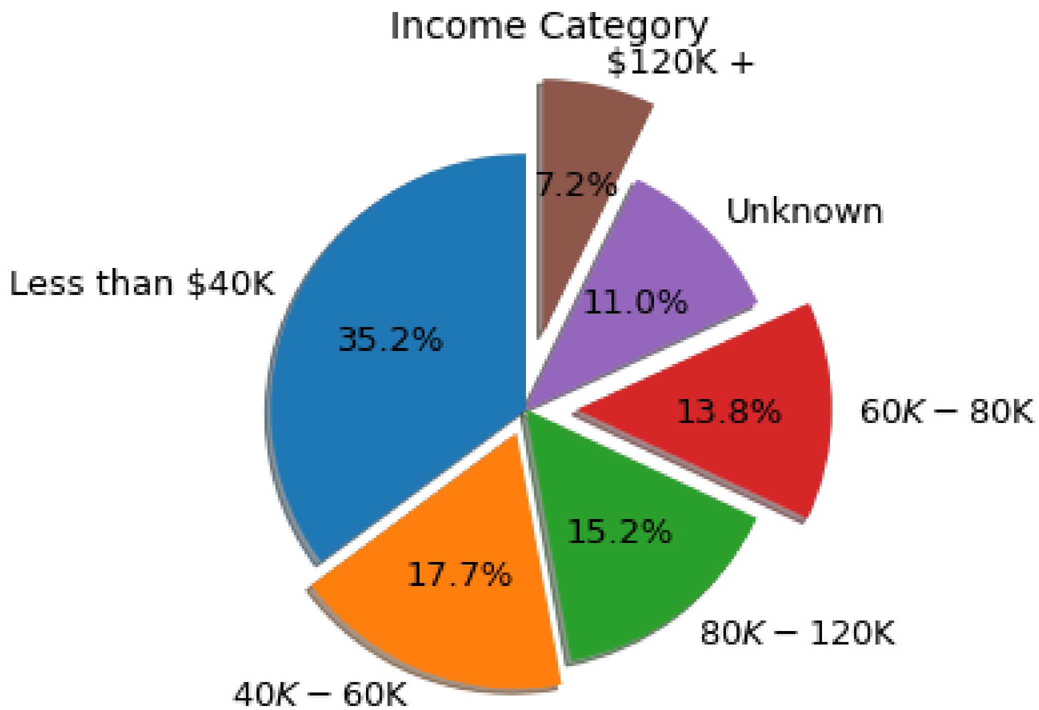


2.1.5 5. Income Category

[33]:
```
df['Income_Category'].value_counts()
```

[33]:
```
Less than $40K    3561
$40K - $60K       1790
$80K - $120K      1535
$60K - $80K       1402
Unknown           1112
$120K +            727
Name: Income_Category, dtype: int64
```

[246]:
```
# Pie chart
labels = [i for i in df['Income_Category'].value_counts().index]
sizes = [i for i in df['Income_Category'].value_counts()]

explode = (0, 0.1,0,0.2,0,0.3)

pie_draw(sizes, "Income Category", labels, explode, "figures/PC_Income_cat.jpg")
```

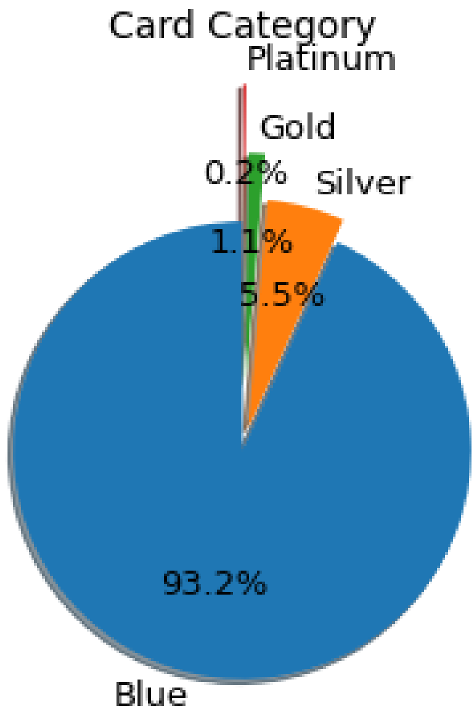


2.1.6 6. Card Category

[35]:
```
df['Card_Category'].value_counts()
```

[35]:
```
Blue        9436
Silver       555
Gold         116
Platinum      20
Name: Card_Category, dtype: int64
```

[247]:
```
# Pie chart
labels = [i for i in df['Card_Category'].value_counts().index]
sizes = [i for i in df['Card_Category'].value_counts()]

explode = (0, 0.1,0.3,0.6)

pie_draw(sizes, "Card Category", labels, explode, "figures/PC_Card_cat.jpg")
```



2.1.7 The Numerical

[38]:
```
integers = df.select_dtypes('int64').columns
integers
```

[38]:
```
Index(['Customer_Age', 'Dependent_count', 'Months_on_book',
       'Total_Relationship_Count', 'Months_Inactive_12_mon',
       'Contacts_Count_12_mon', 'Total_Revolving_Bal', 'Total_Trans_Amt',
       'Total_Trans_Ct'],
      dtype='object')
```

2.1.8 The Numbers

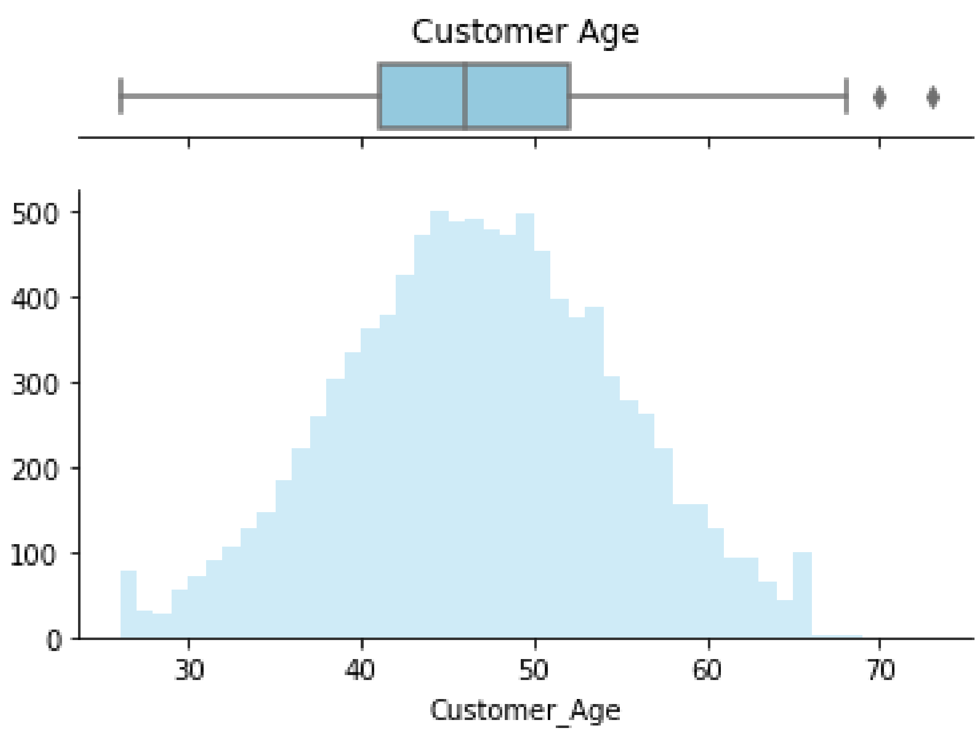
2.1.9 Code below is to draw histogram

[197]:
```
def histo(xcolumn, title, color, filename):
#     sns.boxplot(data=df, x=xcolumn).set_title(title)
#     plt.show()
    f, (ax_box, ax_hist) = plt.subplots(2, sharex=True,
                                        gridspec_kw={"height_ratios": (.15, .85)})

    sns.boxplot(data=df, x=xcolumn, color = color, ax=ax_box)
    sns.distplot(df[xcolumn], color = color, ax=ax_hist, kde=False)
```

```
    ax_box.set(yticks=[])
    ax_box.set_title(title)
    ax_box.set(xlabel=None)
    sns.despine(ax=ax_hist)
    sns.despine(ax=ax_box, left=True)
    plt.savefig(filename)
    plt.show()
```

[248]:
```
histo("Customer_Age", "Customer Age", "skyblue", "figures/HG_Cust_age.jpg")
```

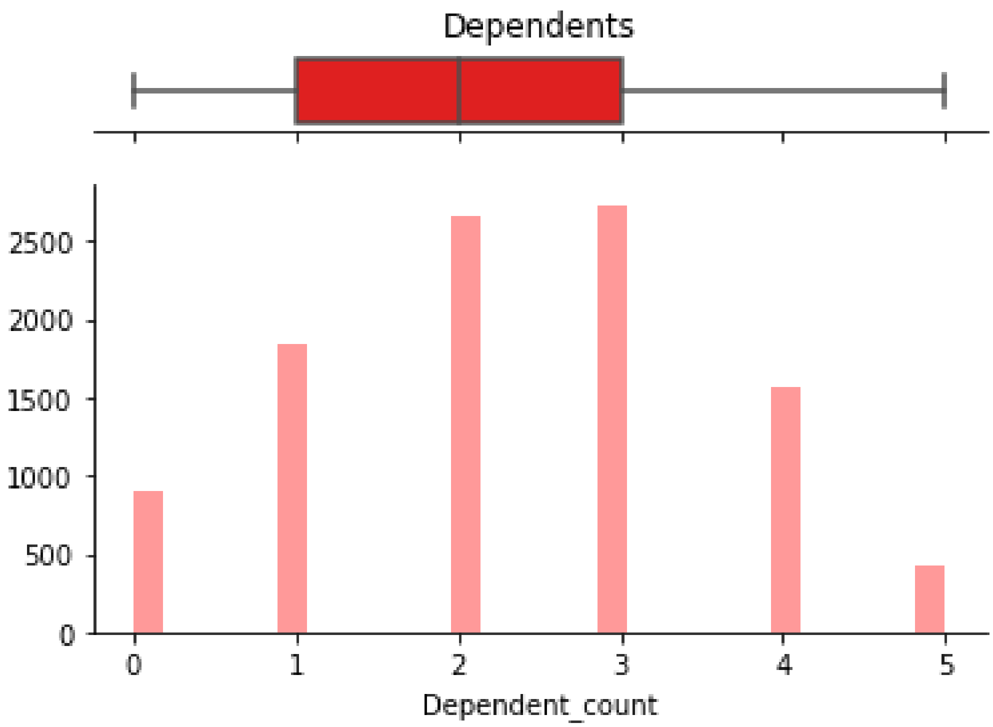
```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



[249]:
```
histo("Dependent_count", "Dependents", "red","figures/HG_Dependent_count.jpg")
```

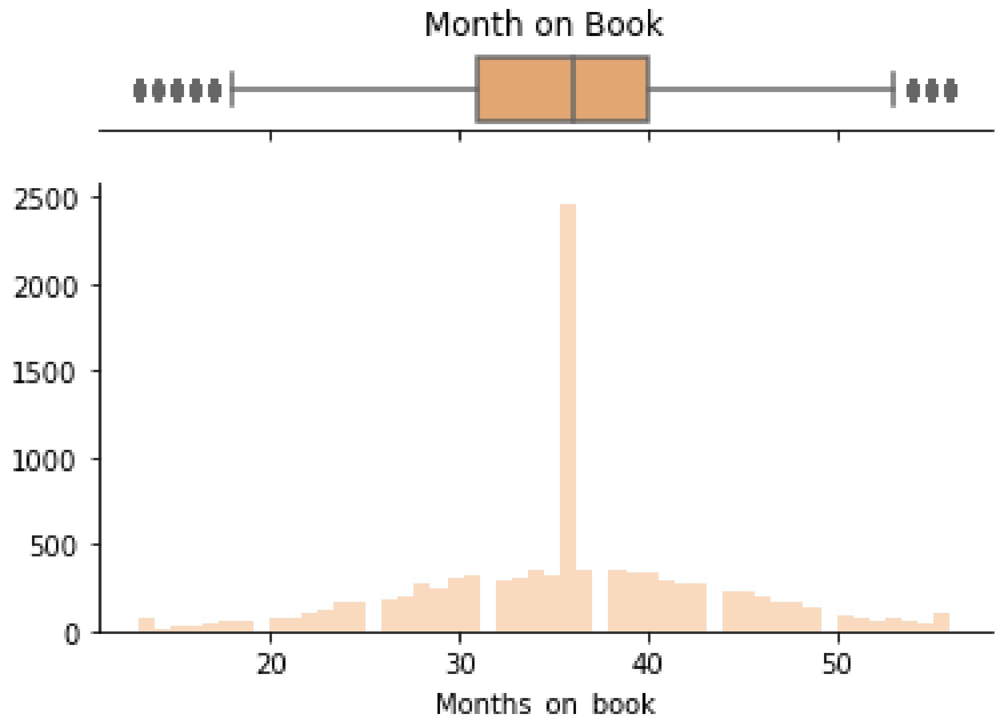
```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:
```

```
`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



[250]:
```
histo('Months_on_book', 'Month on Book', "sandybrown", "figures/
↪HG_Month_on_book.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



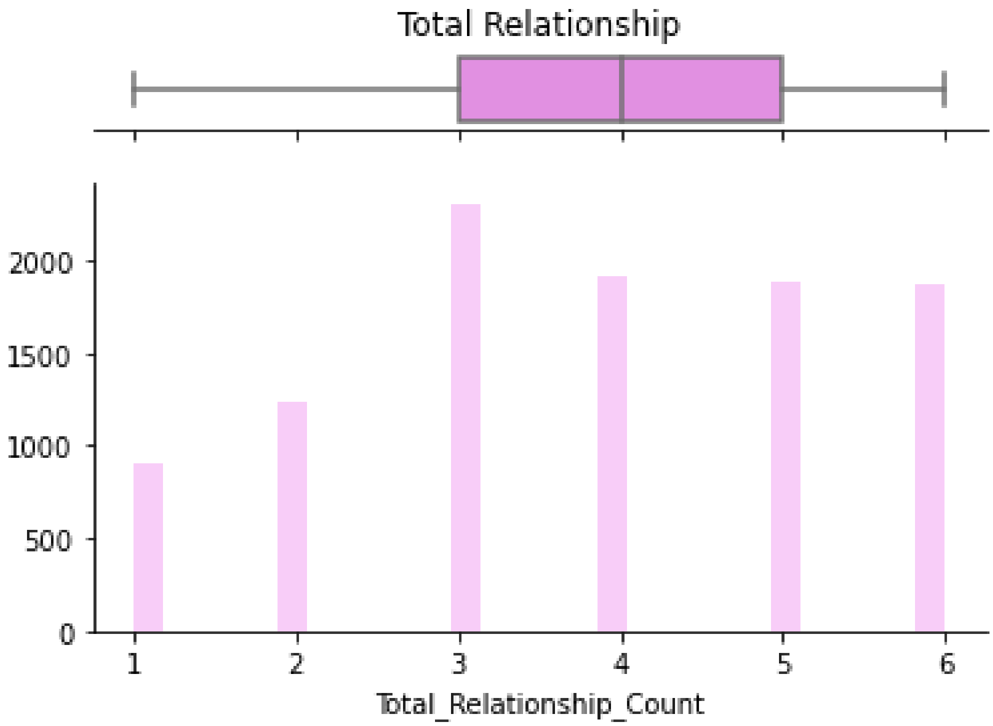

```
[44]: df['Months_on_book'].value_counts().head(3)
```

```
[44]: 36    2463
      37     358
      34     353
      Name: Months_on_book, dtype: int64
```

It can be seen that there is a significant number of people in their 36'th month. It may happened that there was an action 36 months ago: 1. whether the system automatically delete someone on their 36th month? if they are not confirm to continue service? 2. Was it due to promotion or cashback?

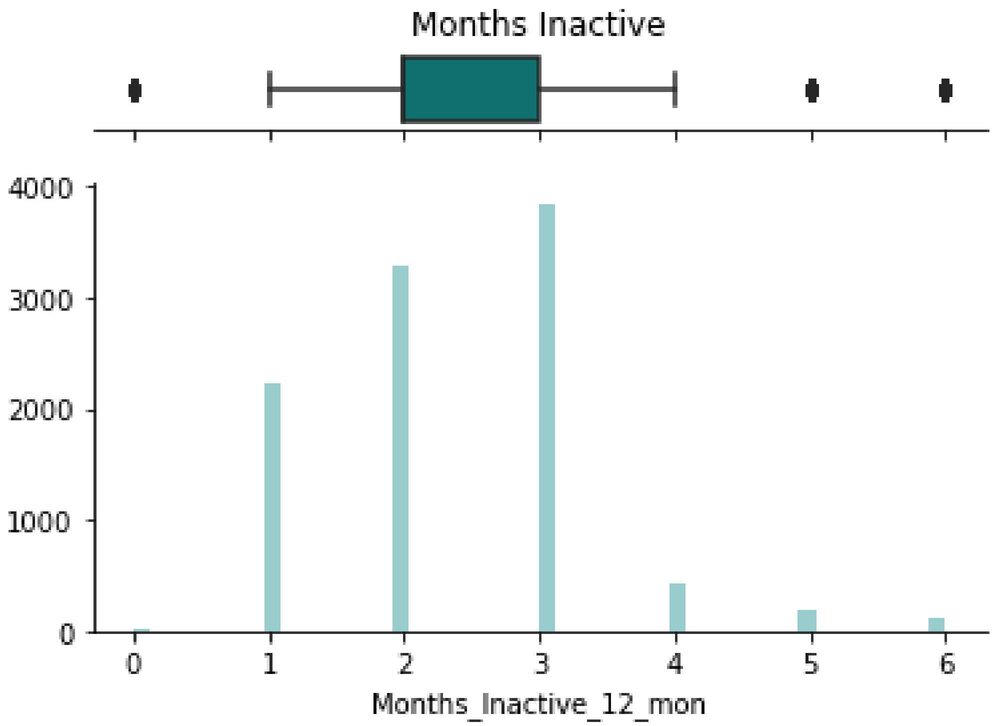
```
[251]: histo('Total_Relationship_Count', 'Total Relationship','violet', "figures/
        ↪HG_Total_relationship.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



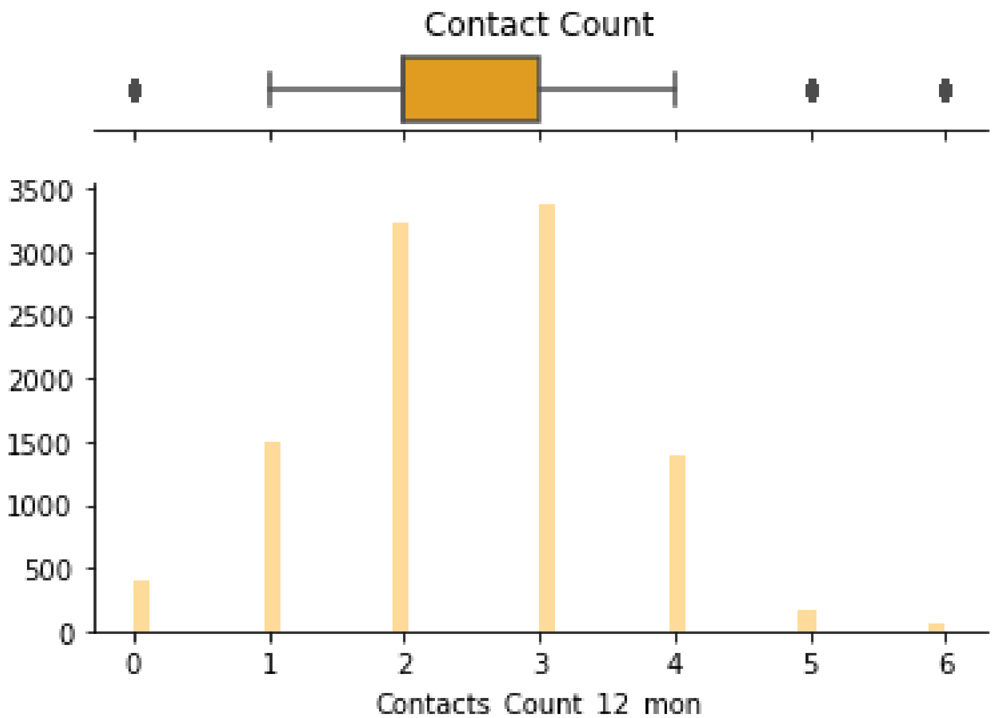
[252]:
```
histo('Months_Inactive_12_mon', 'Months Inactive', "teal", "figures/
↪HG_Month_inactive.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



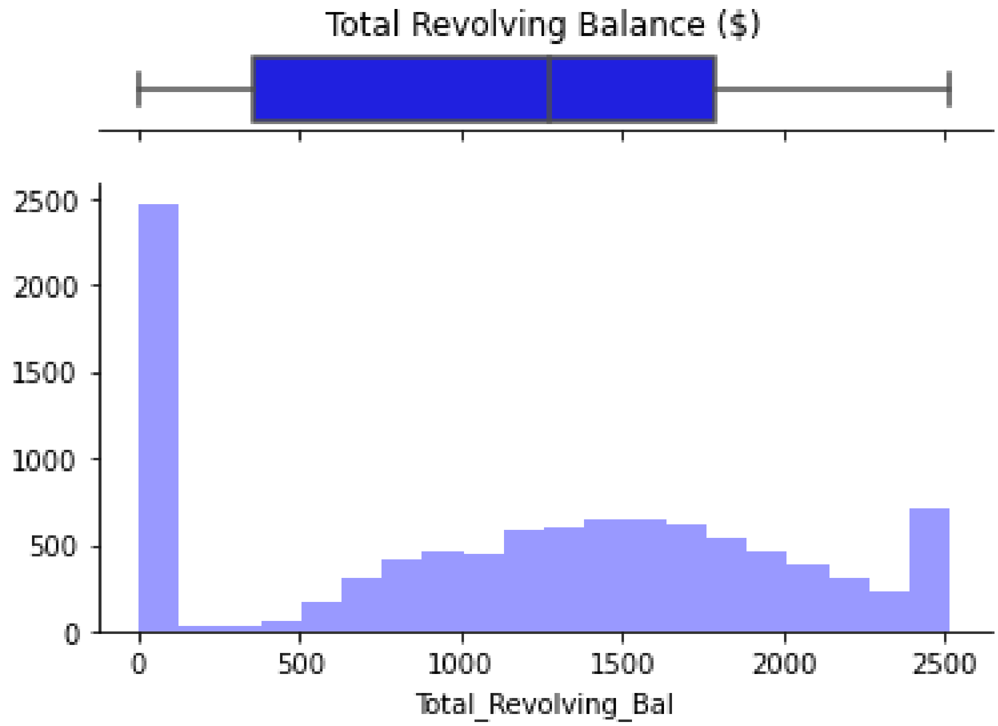
[253]:
```
histo('Contacts_Count_12_mon', 'Contact Count', "orange", "figures/
↪HG_Contacts_count.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



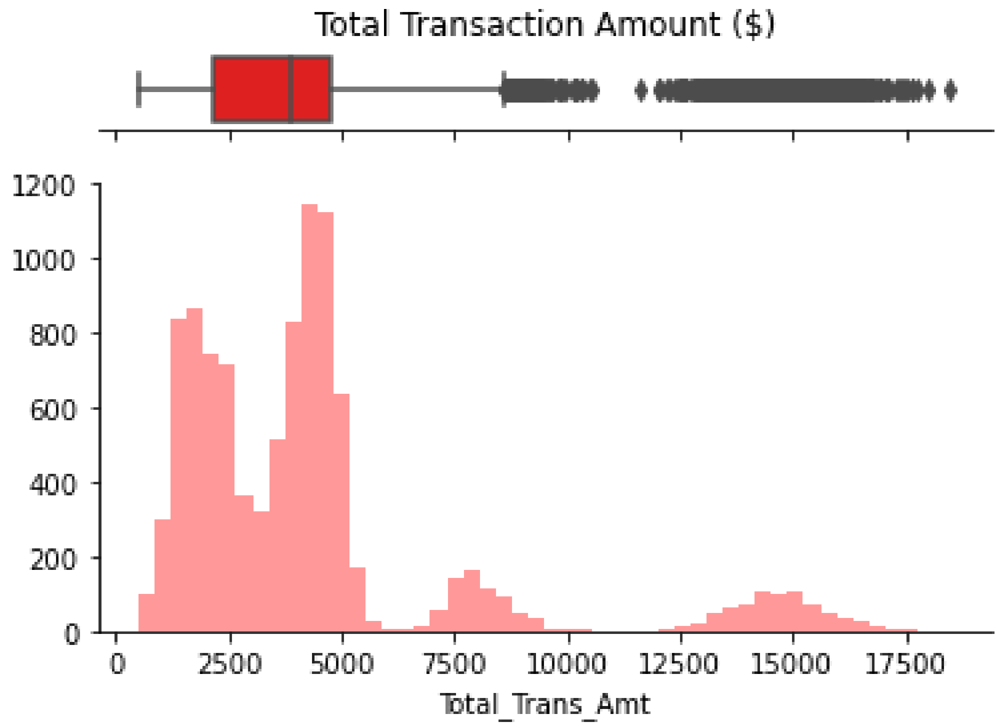
[254]:
```
histo('Total_Revolving_Bal','Total Revolving Balance ($)', "blue", "figures/
↪HG_Total_rev_balance.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



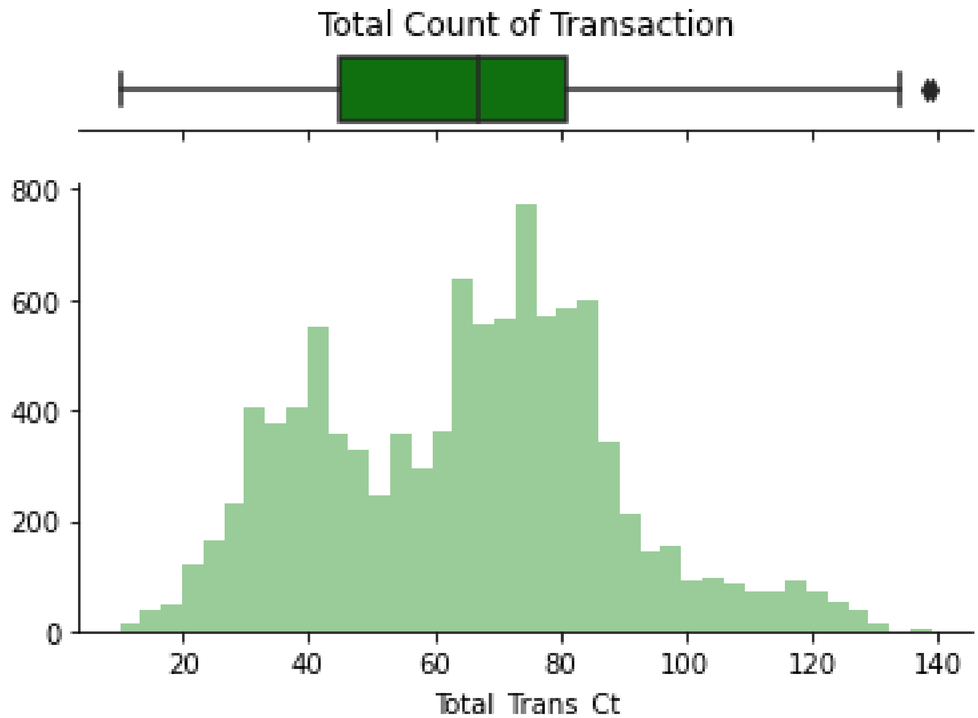
[255]:
```
histo('Total_Trans_Amt','Total Transaction Amount ($)', "red", "figures/
↪HG_Total_transaction.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



[256]:
```
histo('Total_Trans_Ct', 'Total Count of Transaction', "green", "figures/
↪HG_Transaction_counts.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```

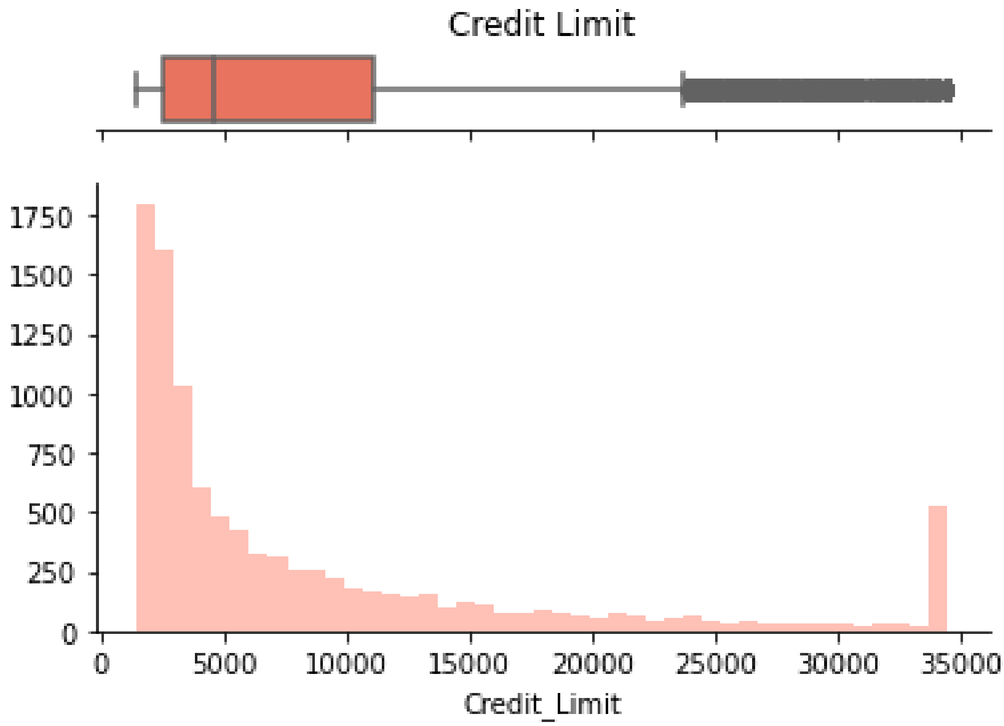


[52]:
```
floats = df.select_dtypes('float64').columns
floats
```

[52]:
```
Index(['Credit_Limit', 'Avg_Open_To_Buy', 'Total_Amt_Chng_Q4_Q1',
       'Total_Ct_Chng_Q4_Q1', 'Avg_Utilization_Ratio'],
      dtype='object')
```

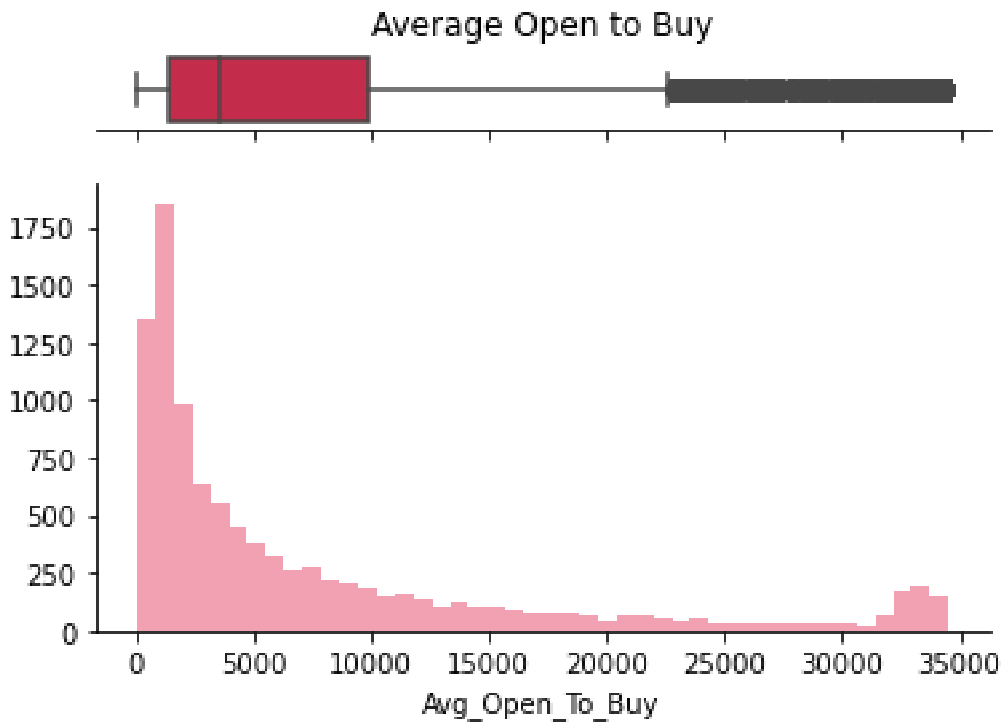
[257]:
```
histo('Credit_Limit', 'Credit Limit', "tomato", "figures/HG_Credit Limit.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



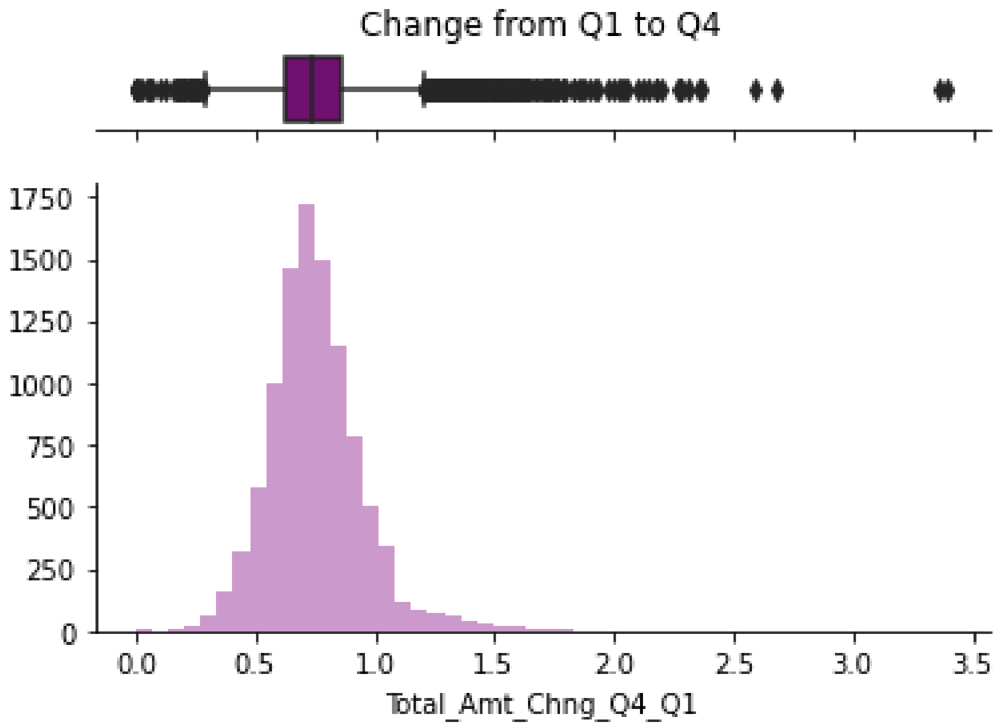
[258]:
```
histo('Avg_Open_To_Buy', 'Average Open to Buy', "crimson", "figures/
↪HG_Avg_opento_buy.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



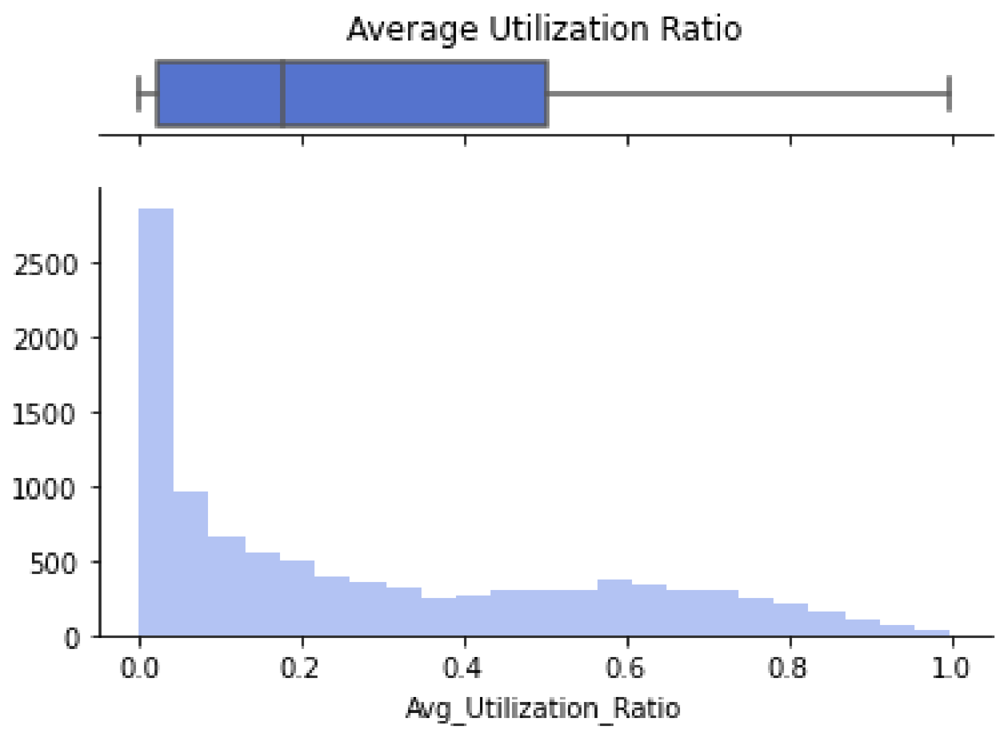
[259]:
```
histo('Total_Amt_Chng_Q4_Q1', 'Change from Q1 to Q4', "purple", "figures/
↪HG_TotalChangeQ1-Q4.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



[260]:
```
histo('Avg_Utilization_Ratio', 'Average Utilization Ratio', "royalblue",␣
↪"figures/HG_Util_ratio.jpg")
```

```
/opt/conda/lib/python3.7/site-packages/seaborn/distributions.py:2551:
FutureWarning:

`distplot` is a deprecated function and will be removed in a future version.
Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).
```



2.2 Now that we have all the individual plot, we can see their corellation

But first we have to factorize all the categorical values, like what we do next in the ETL.

It should be noted that factorizing might not be the best option compared to the one-hot-encoder as it introduced serial/sequential relation between seemingly unrelated categories. But, this will greatly reduced the number of features/dimension

[58]:
```
df_churn = df.copy()
for i in categorical:
    df_churn[i]=pd.factorize(df_churn[i])[0]
df_churn.head(4)
```

[58]:

	Attrition_Flag	Customer_Age	Gender	Dependent_count	Education_Level	\
0	0	45	0	3	0	
1	0	49	1	5	1	
2	0	51	0	3	1	
3	0	40	1	4	0	

	Marital_Status	Income_Category	Card_Category	Months_on_book	\
0	0	0	0	39	
1	1	1	0	44	
2	0	2	0	36	
3	2	1	0	34	

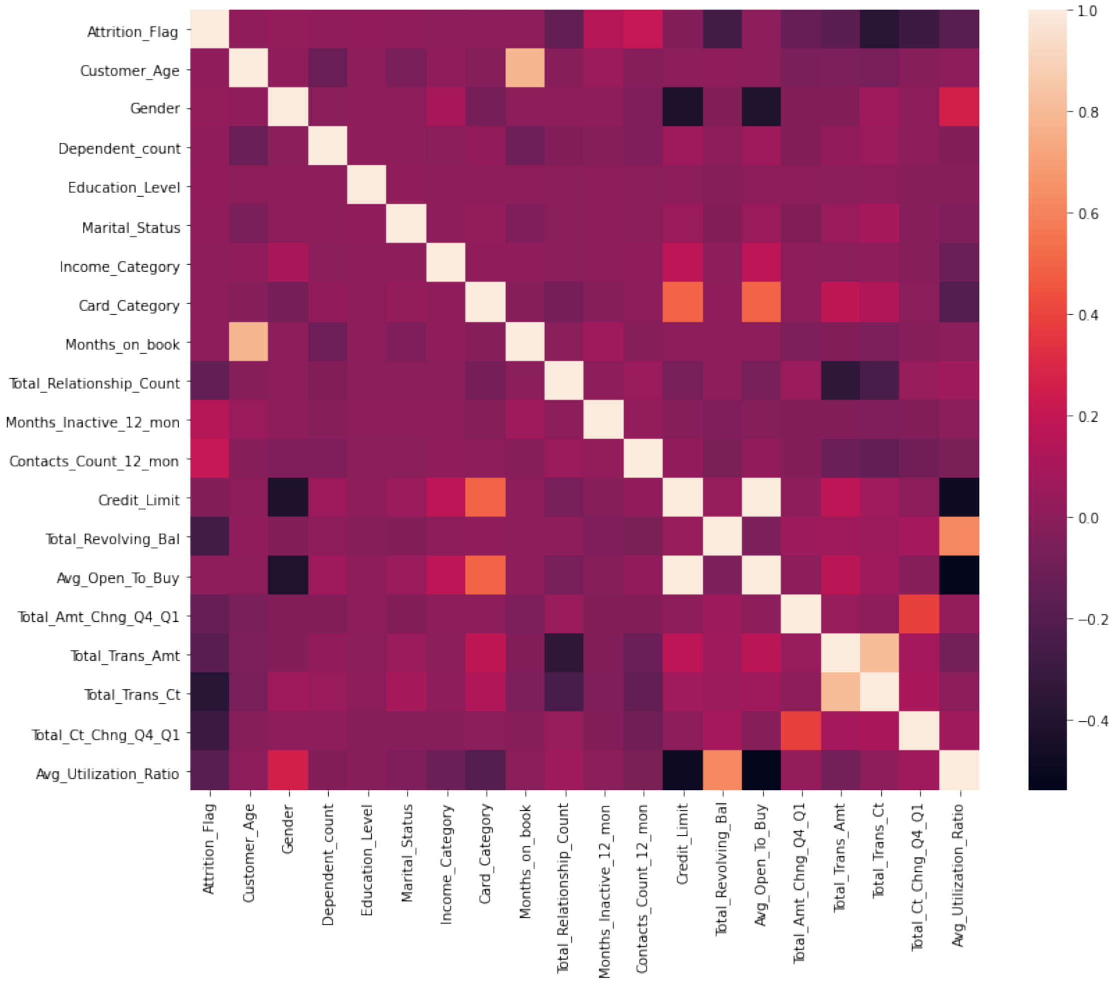
```
   Total_Relationship_Count  Months_Inactive_12_mon  Contacts_Count_12_mon  \
0                         5                       1                      3
1                         6                       1                      2
2                         4                       1                      0
3                         3                       4                      1

   Credit_Limit  Total_Revolving_Bal  Avg_Open_To_Buy  Total_Amt_Chng_Q4_Q1  \
0       12691.0                  777          11914.0                 1.335
1        8256.0                  864           7392.0                 1.541
2        3418.0                    0           3418.0                 2.594
3        3313.0                 2517            796.0                 1.405

   Total_Trans_Amt  Total_Trans_Ct  Total_Ct_Chng_Q4_Q1  Avg_Utilization_Ratio
0             1144              42                1.625                  0.061
1             1291              33                3.714                  0.105
2             1887              20                2.333                  0.000
3             1171              20                2.333                  0.760
```

[236]:

```
import seaborn as sns
from matplotlib import pyplot
```

[261]:

```
x=list(df_churn.corr().columns)
y=list(df_churn.corr().index)
corr = df_churn.corr()

fig, ax = pyplot.subplots(figsize=(12,10))
sns.heatmap(corr,
            xticklabels=corr.columns.values,
            yticklabels=corr.columns.values, ax=ax)
plt.tight_layout()
plt.savefig('figures/CM_correlation_matrix.png')
plt.show()
# values=np.array(df_churn.corr().values)
# fig = go.Figure(data=go.Heatmap(
#     z=values,
#     x=x,
#     y=y,
#     hoverongaps = False))
# # fig.write_image("figures/correlation_matrix.jpg")
# fig.show()
```



3 B. ET & L

Extract, transform, load

3.0.1 Import Library and Packages

[62]:
```
import pandas as pd
import numpy as np
```

3.0.2 About the Data

The data was obtained from Kaggle: https://www.kaggle.com/sakshigoyal7/credit-card-customers

Column Name: CLIENTNUM = Client number. Unique identifier for the customer holding the account

Attrition_Flag = Internal event (customer activity) variable - if the account is closed then 1 else 0 (THIS IS THE TARGET PARAMETER)

Customer_Age

Gender = Demographic variable - M=Male, F=Female

Dependent_count = Demographic variable - Number of dependents

Education_Level

Marital_Status

Income_Category = Demographic variable - Annual Income Category of the account holder (< $40K, $40K - 60K, $60K-$80K, $80K-$120K, > $120K, Unknown)

Card_Category = Product Variable - Type of Card (Blue, Silver, Gold, Platinum)

Months_on_book = Period of relationship with bank

Total_Relationship_Count = Total no. of products held by the customer

Months_Inactive_12_mon = No. of months inactive in the last 12 months

Contacts_Count_12_mon = No. of Contacts in the last 12 months

Credit_Limit

Total_Revolving_Bal = Total Revolving Balance on the Credit Card

Avg_Open_To_Buy = Open to Buy Credit Line (Average of last 12 months)

Total_Amt_Chng_Q4_Q1 = Change in Transaction Amount (Q4 over Q1)

Total_Trans_Amt = Total Transaction Amount (Last 12 months)

Total_Trans_Ct = Total Transaction Count (Last 12 months)

Total_Ct_Chng_Q4_Q1 = Change in Transaction Count (Q4 over Q1)

Avg_Utilization_Ratio = Average Card Utilization Ratio

3.0.3 The data was obtained from Kaggle

[63]:
```
# df = pd.read_csv('https://s3-api.us-geo.objectstorage.softlayer.net/
↪advancedmachinelearning-donotdelete-pr-gwymm5mokoi4ul/BankChurners.csv?
↪response-content-disposition=attachment%3B%20filename%3D%22BankChurners.
↪csv%22&response-content-type=text%2Fcsv&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=2021010
# df.head(3)
```

[64]:
```
# df.to_csv('ChurnData.csv',index=False)

df = pd.read_csv('ChurnData.csv')
```

[65]:
```
# df.info()
```

It can be seen from the df.info that the data are consisted of 22 columns, where all parameters have all type that supposed to. In this case, the CLIENTNUM and the last two columns are not important, therefore they are deleted.

[66]:
```
df.drop(columns=df.columns[-2:], inplace=True)
df.drop('CLIENTNUM',axis=1,inplace=True)
```

Checking the columns info :

[68]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10127 entries, 0 to 10126
Data columns (total 20 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Attrition_Flag            10127 non-null  object
 1   Customer_Age              10127 non-null  int64
 2   Gender                    10127 non-null  object
 3   Dependent_count           10127 non-null  int64
 4   Education_Level           10127 non-null  object
 5   Marital_Status            10127 non-null  object
 6   Income_Category           10127 non-null  object
 7   Card_Category             10127 non-null  object
 8   Months_on_book            10127 non-null  int64
 9   Total_Relationship_Count  10127 non-null  int64
 10  Months_Inactive_12_mon    10127 non-null  int64
 11  Contacts_Count_12_mon     10127 non-null  int64
 12  Credit_Limit              10127 non-null  float64
 13  Total_Revolving_Bal       10127 non-null  int64
 14  Avg_Open_To_Buy           10127 non-null  float64
 15  Total_Amt_Chng_Q4_Q1      10127 non-null  float64
 16  Total_Trans_Amt           10127 non-null  int64
 17  Total_Trans_Ct            10127 non-null  int64
 18  Total_Ct_Chng_Q4_Q1       10127 non-null  float64
 19  Avg_Utilization_Ratio     10127 non-null  float64
dtypes: float64(5), int64(9), object(6)
memory usage: 1.5+ MB
```

3.0.4 Lets collect all the Categorical data, which means excluding the integer and float

[70]:
```
categorical=df.select_dtypes(exclude=['int64','float64']).columns
print(*categorical)
```

```
Attrition_Flag Gender Education_Level Marital_Status Income_Category
Card_Category
```

3.0.5 It would be good to check whether this categorical data has no error in value, check it with following code:

[71]:
```
for category in categorical:
    print(df[category].value_counts(),'\n')
```

```
Existing Customer    8500
Attrited Customer    1627
Name: Attrition_Flag, dtype: int64

F    5358
M    4769
Name: Gender, dtype: int64

Graduate         3128
High School      2013
Unknown          1519
Uneducated       1487
College          1013
Post-Graduate     516
Doctorate         451
Name: Education_Level, dtype: int64

Married     4687
Single      3943
Unknown      749
Divorced     748
Name: Marital_Status, dtype: int64

Less than $40K    3561
$40K - $60K       1790
$80K - $120K      1535
$60K - $80K       1402
Unknown           1112
$120K +            727
Name: Income_Category, dtype: int64

Blue        9436
Silver       555
Gold         116
Platinum      20
Name: Card_Category, dtype: int64
```

It can be seen that there is no mistake in the values in the columns. Next lets see whether there is no null value

[72]:
```
df[df.notna().any(axis=1)].count()
```

[72]:

```
Attrition_Flag              10127
Customer_Age                10127
Gender                      10127
Dependent_count             10127
Education_Level             10127
Marital_Status              10127
Income_Category             10127
Card_Category               10127
Months_on_book              10127
Total_Relationship_Count    10127
Months_Inactive_12_mon      10127
Contacts_Count_12_mon       10127
Credit_Limit                10127
Total_Revolving_Bal         10127
Avg_Open_To_Buy             10127
Total_Amt_Chng_Q4_Q1        10127
Total_Trans_Amt             10127
Total_Trans_Ct              10127
Total_Ct_Chng_Q4_Q1         10127
Avg_Utilization_Ratio       10127
dtype: int64
```

It can also be seen that theres is no columns or row with NaN value, which is awesome, this data is basically ready to go.

3.0.6 Create new dataframe for from df

[73]:

```
df_churn = df.copy()
df_churn.head(2)
```

[73]:

```
      Attrition_Flag  Customer_Age Gender  Dependent_count Education_Level  \
0  Existing Customer            45      M                3     High School
1  Existing Customer            49      F                5        Graduate

  Marital_Status Income_Category Card_Category  Months_on_book  \
0        Married     $60K - $80K          Blue              39
1         Single  Less than $40K          Blue              44

   Total_Relationship_Count  Months_Inactive_12_mon  Contacts_Count_12_mon  \
0                         5                       1                      3
1                         6                       1                      2

   Credit_Limit  Total_Revolving_Bal  Avg_Open_To_Buy  Total_Amt_Chng_Q4_Q1  \
0       12691.0                  777          11914.0                 1.335
1        8256.0                  864           7392.0                 1.541

   Total_Trans_Amt  Total_Trans_Ct  Total_Ct_Chng_Q4_Q1  Avg_Utilization_Ratio
```

```
0          1144        42           1.625                 0.061
1          1291        33           3.714                 0.105
```

Lets factorize all categorical data:

[74]:
```
for i in categorical:
    df_churn[i]=pd.factorize(df_churn[i])[0]
df_churn.head(4)
```

[74]:
```
   Attrition_Flag  Customer_Age  Gender  Dependent_count  Education_Level  \
0               0            45       0                3                0
1               0            49       1                5                1
2               0            51       0                3                1
3               0            40       1                4                0

   Marital_Status  Income_Category  Card_Category  Months_on_book  \
0               0                0              0              39
1               1                1              0              44
2               0                2              0              36
3               2                1              0              34

   Total_Relationship_Count  Months_Inactive_12_mon  Contacts_Count_12_mon  \
0                         5                       1                      3
1                         6                       1                      2
2                         4                       1                      0
3                         3                       4                      1

   Credit_Limit  Total_Revolving_Bal  Avg_Open_To_Buy  Total_Amt_Chng_Q4_Q1  \
0       12691.0                  777          11914.0                 1.335
1        8256.0                  864           7392.0                 1.541
2        3418.0                    0           3418.0                 2.594
3        3313.0                 2517            796.0                 1.405

   Total_Trans_Amt  Total_Trans_Ct  Total_Ct_Chng_Q4_Q1  Avg_Utilization_Ratio
0             1144              42                1.625                  0.061
1             1291              33                3.714                  0.105
2             1887              20                2.333                  0.000
3             1171              20                2.333                  0.760
```

[75]:
```
df_churn.to_csv('df_churn.csv', index=False)
```

Its better to keep in track the factor number and its values

[76]:
```
catal = {}
for category in categorical:
    catal[category] = pd.DataFrame(list(zip(df_churn[category].value_counts().
 ↪index, df[category].value_counts().index)))
```

[77]:
```
# The catal is to inform us about the variable that has been factorized
catal
```

[77]:
```
{'Attrition_Flag':    0                  1
 0  0  Existing Customer
 1  1  Attrited Customer,
 'Gender':    0  1
 0  1  F
 1  0  M,
 'Education_Level':    0              1
 0  1       Graduate
 1  0    High School
 2  3        Unknown
 3  2     Uneducated
 4  4        College
 5  5  Post-Graduate
 6  6      Doctorate,
 'Marital_Status':    0         1
 0  0   Married
 1  1    Single
 2  2   Unknown
 3  3  Divorced,
 'Income_Category':    0               1
 0  1  Less than $40K
 1  3     $40K - $60K
 2  2    $80K - $120K
 3  0     $60K - $80K
 4  5         Unknown
 5  4         $120K +,
 'Card_Category':    0         1
 0  0      Blue
 1  2    Silver
 2  1      Gold
 3  3  Platinum}
```

3.1 Create training and test datasets from df_churn

3.1.1 X is all the 19 columns except Attrition_Flag, where the y is the Attrition_Flag

[78]:
```
X = df_churn.drop('Attrition_Flag', axis = 1).values

y = df_churn['Attrition_Flag'].values
```

3.1.2 Save the data

[80]:
```
from numpy import asarray
from numpy import save

# save the X and y arrays

save('X.npy', X)
save('y.npy', y)
```

3.1.3 Train Test Split

[82]:
```
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

Take 80% for Trains data and 20 for Test and

Then the X__trains and y__trains will be divided into train and validation

[83]:
```
print(int(0.8*len(df_churn)), "rows will be used as training")
```

```
8101 rows will be used as training
```

3.1.4 Unscaled

Split the data to Train and Test

[84]:
```
X_trains, X_test, y_trains, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=20, shuffle=True)
```

Split the Train data to Train and Validation

[85]:
```
X_train, X_val, y_train, y_val = train_test_split(X_trains, y_trains,␣
↪test_size=0.2, random_state=20, shuffle=True)
```

3.1.5 Scaled

It can be seen from the Exploratory data analysis that the data, overall, are not normally distributed. Therefore, I think the best way to scale it is by using MinMaxScaler, rather than StandardScaler

[86]:
```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

Split the scaled data to Train and Test

[87]:
```
X_trains_scaled, X_test_scaled, y_trains_scaled, y_test_scaled =␣
↪train_test_split(X_scaled, y, test_size=0.2, random_state=20, shuffle=True)
```

Split the Train data to another Train and Validation

[88]:
```
X_train_scaled, X_val_scaled, y_train_scaled, y_val_scaled =␣
 ↪train_test_split(X_trains_scaled, y_trains_scaled, test_size=0.2,␣
 ↪random_state=20, shuffle=True)
```

Note that the y%scaled are not actually scaled, its just for labelling to make it matched with the corresponding X

Show the shape

[91]:
```
print("Training", X_train.shape, 'and', y_train.shape)
print("Validation", X_val.shape, 'and', y_val.shape)
```

```
Training (6480, 19) and (6480,)
Validation (1621, 19) and (1621,)
```

3.1.6 Get Scaled train and test from X

3.2 Save data to npy format

[92]:
```
# # save numpy array as npy file
# from numpy import asarray
# from numpy import save
```

3.2.1 I save them in numpy format, so it can be loaded for later

[93]:
```
#Unscaled for ML
save('X_train.npy', X_train)
save('X_test.npy', X_test)
save('y_train.npy', y_train)
save('y_test.npy', y_test)
save('X_val.npy', X_val)
save('y_val.npy', y_val)


#Scaled for the Neural Network
save('X_train_scaled.npy', X_train_scaled)
save('X_test_scaled.npy', X_test_scaled)
save('y_train_scaled.npy', y_train_scaled)
save('y_test_scaled.npy', y_test_scaled)
save('X_val_scaled.npy', X_val_scaled)
save('y_val_scaled.npy', y_val_scaled)
```

4 C. This Section is to Find the best model For the Credit Card Churn

4.0.1 Load packages and library

[95]:
```
import pandas as pd
import numpy as np
from numpy import load
import plotly.express as px
import copy
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from sklearn.model_selection import train_test_split, cross_val_score
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score
from sklearn import metrics, svm
import plotly
import os
```

4.0.2 load array from the previous ETL

[96]:
```
# load array from the previous ETL
X_train = load('X_train.npy')
X_test = load('X_test.npy')
y_train = load('y_train.npy')
y_test = load('y_test.npy')
X_train_scaled = load('X_train_scaled.npy')
X_test_scaled = load('X_test_scaled.npy')
```

Now that the train and test data were loaded, model can be tested

4.1 Load functions to get confusion matrix and classifier report

[240]:
```
def conf_matrix(classifier, Title, ydata, xdata, filename):
    cf = metrics.confusion_matrix(ydata, classifier.predict(xdata),␣
 ↪labels=[1,0])
    x_axis_labels=['Attrited','Existing']
    y_axis_labels=['Attrited','Existing']

    ax = plt.axes()
    sns.heatmap(cf, ax = ax,annot=True, xticklabels=x_axis_labels,␣
 ↪yticklabels=y_axis_labels)
    ax.set_title(Title)
    ax.set_xlabel('Predicted')
```

```
    ax.set_ylabel('Actual')
    plt.savefig(filename)
    plt.show()

def class_report(classifier, ydata, xdata):
    print(metrics.classification_report(ydata,classifier.predict(xdata),␣
 ↪labels=[1,0]))
    print('Accuracy_Score:',accuracy_score(ydata,classifier.
 ↪predict(xdata))*100,'%')
    print('Recall:',metrics.recall_score(ydata, classifier.
 ↪predict(xdata))*100,'%')
```

4.2 Lets try The Machine Learning

4.2.1 In this section, 4 Machine learning algorithm and 2 Neural Network model will be used

1. Random Forest Classifier
2. Logistic Regression
3. XGBoost Classifier
4. LGBM Classifier
5. a. Long Neural Network
6. b. Wide Neural Network

5 1. Random Forest Classifier

[98]:
```
rf=RandomForestClassifier()
```

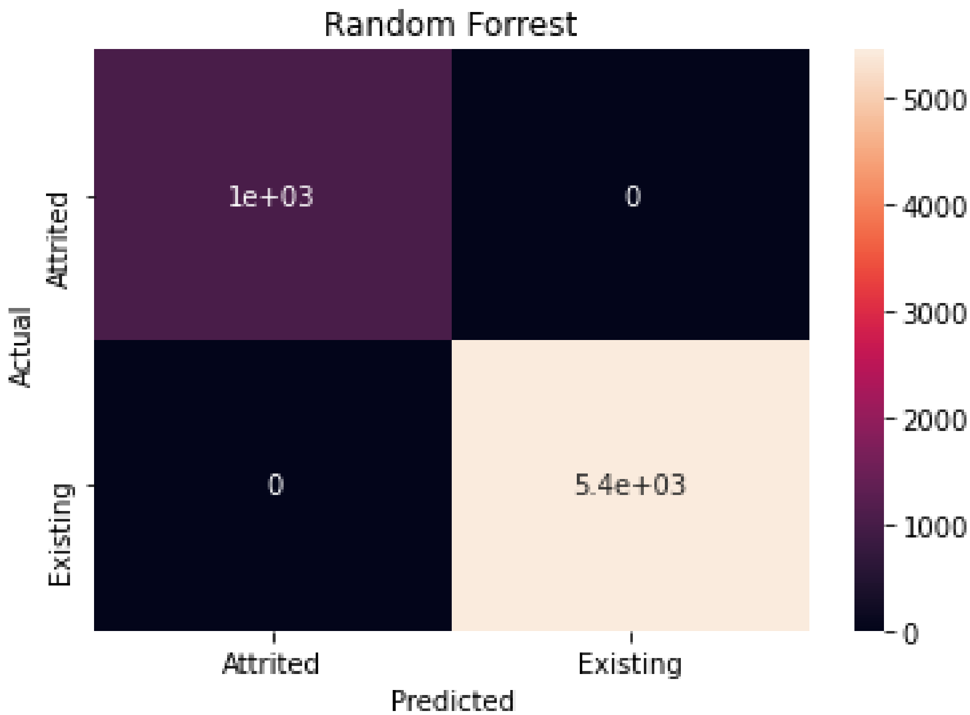
[99]:
```
rf.fit(X_train, y_train)
```

[99]: RandomForestClassifier()

[109]:
```
scores1 = cross_val_score(rf, X_train, y_train, cv=5)
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores1.
 ↪mean()*100, scores1.std()))
```

```
95.73 accuracy with a standard deviation of 0.00
```

[262]:
```
conf_matrix(rf, 'Random Forrest', y_train, X_train, "figures/ML_RandomForest.
 ↪jpg")
```



It is great to see that the model fit perfectly for the train data

5.0.1 Try it on the test data

[103]:
```
class_report(rf, y_test, X_test)
```

```
              precision    recall  f1-score   support

           1       0.92      0.82      0.87       301
           0       0.97      0.99      0.98      1725

    accuracy                           0.96      2026
   macro avg       0.95      0.90      0.92      2026
weighted avg       0.96      0.96      0.96      2026

Accuracy_Score: 96.29812438302073 %
Recall: 82.05980066445183 %
```

[104]:
```
metrics.f1_score(y_test,rf.predict(X_test))
```

[104]:
```
0.8681898066783832
```

6 2. Logistic Regression

6.0.1 Load all packages and library

[110]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline, make_pipeline
```

[111]:
```
pipe = make_pipeline(MinMaxScaler(),LogisticRegression())
```

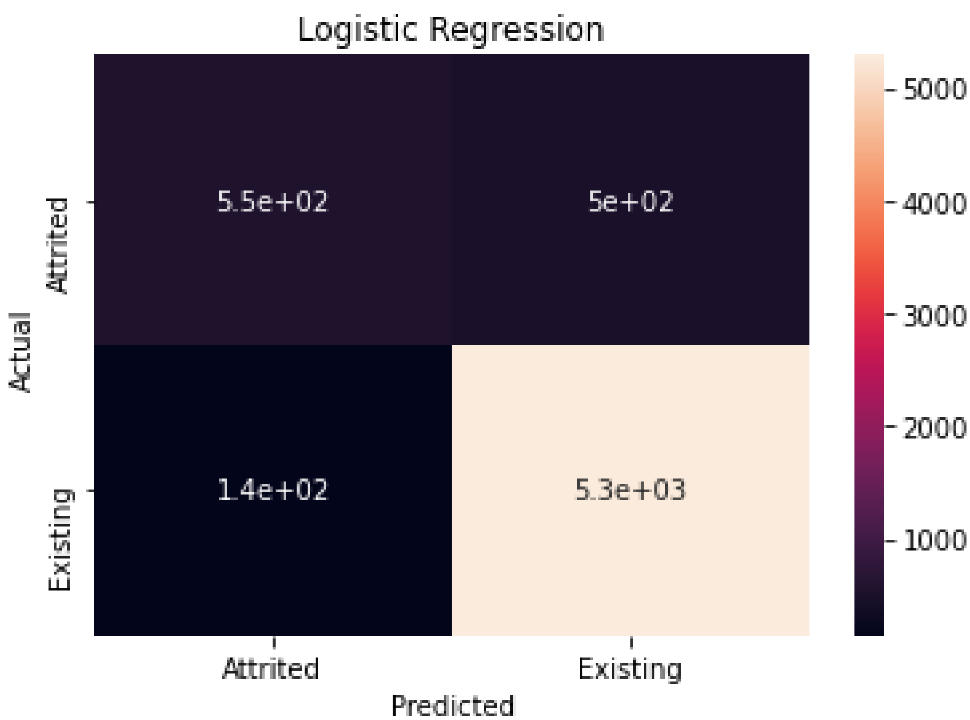
[112]:
```
lr = pipe.fit(X_train, y_train)
lr
```

[112]:
```
Pipeline(steps=[('minmaxscaler', MinMaxScaler()),
                ('logisticregression', LogisticRegression())])
```

[116]:
```
scores2 = cross_val_score(lr, X_train, y_train, cv=5)
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores2.
 ↪mean()*100, scores2.std()))
```

```
89.83 accuracy with a standard deviation of 0.01
```

[264]:
```
conf_matrix(lr, "Logistic Regression", y_train, X_train, "figures/ML_LogReg.
 ↪jpg")
```



6.0.2 Try it on the test data

[118]:
```
class_report(lr, y_test, X_test)
```

```
              precision    recall  f1-score   support

           1       0.80      0.54      0.64       301
           0       0.92      0.98      0.95      1725

    accuracy                           0.91      2026
   macro avg       0.86      0.76      0.80      2026
weighted avg       0.91      0.91      0.90      2026

Accuracy_Score: 91.11549851924975 %
Recall: 54.15282392026578 %
```

7 3. XGBoost Classifier

[119]:
```
xgb = XGBClassifier()
xgb.fit(X_train, y_train)
```

```
/opt/conda/lib/python3.7/site-packages/xgboost/sklearn.py:892: UserWarning:

The use of label encoder in XGBClassifier is deprecated and will be removed in a
future release. To remove this warning, do the following: 1) Pass option
use_label_encoder=False when constructing XGBClassifier object; and 2) Encode
your labels (y) as integers starting with 0, i.e. 0, 1, 2, …, [num_class - 1].


[04:32:30] WARNING: ../src/learner.cc:1061: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
```

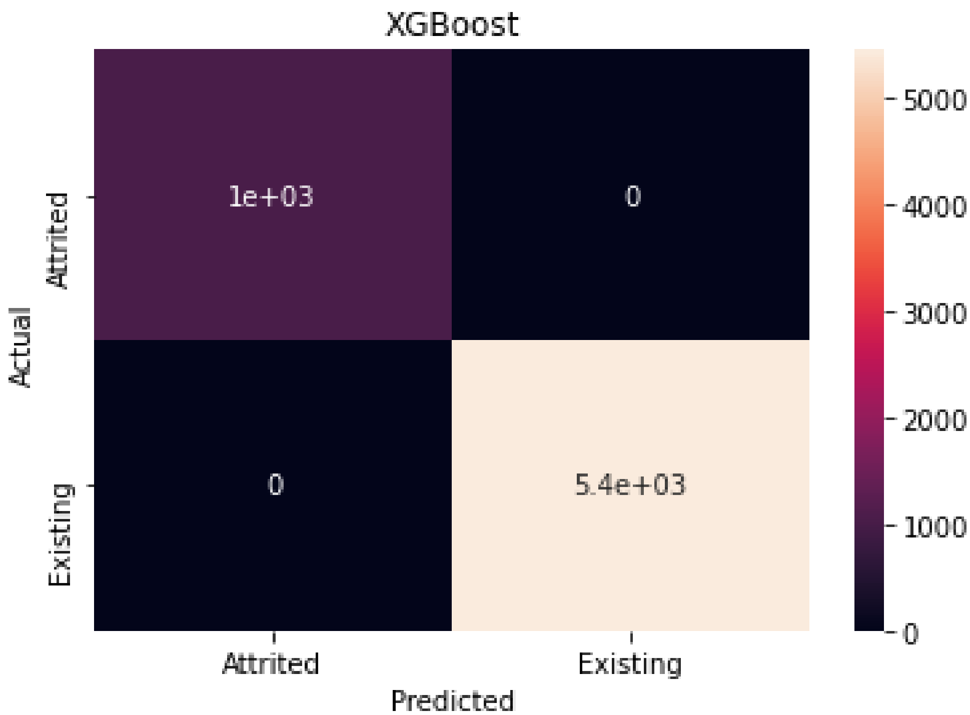
[119]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=100, n_jobs=8, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

[120]:
```
scores3 = cross_val_score(xgb, X_train, y_train, cv=5)
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores3.
 ↪mean()*100, scores3.std()))
```

```
/opt/conda/lib/python3.7/site-packages/xgboost/sklearn.py:892: UserWarning:
```

```
The use of label encoder in XGBClassifier is deprecated and will be removed in a
future release. To remove this warning, do the following: 1) Pass option
use_label_encoder=False when constructing XGBClassifier object; and 2) Encode
your labels (y) as integers starting with 0, i.e. 0, 1, 2, …, [num_class - 1].


[04:32:54] WARNING: ../src/learner.cc:1061: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
[04:32:54] WARNING: ../src/learner.cc:1061: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
[04:32:55] WARNING: ../src/learner.cc:1061: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
[04:32:55] WARNING: ../src/learner.cc:1061: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
[04:32:55] WARNING: ../src/learner.cc:1061: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
96.93 accuracy with a standard deviation of 0.00
```

[265]:
```
conf_matrix(xgb, 'XGBoost', y_train, X_train, "figures/ML_XGBoost.jpg")
```



7.0.1 Try it on the test data

[122]:
```
class_report(xgb, y_test, X_test)
```

```
              precision    recall  f1-score   support

           1       0.91      0.89      0.90       301
           0       0.98      0.98      0.98      1725

    accuracy                           0.97      2026
   macro avg       0.94      0.94      0.94      2026
weighted avg       0.97      0.97      0.97      2026

Accuracy_Score: 96.98914116485686 %
Recall: 88.70431893687709 %
```

8 4. LGBM Classifier

Light Gradient Boosting Machine

[123]:
```
lgbm=LGBMClassifier()
```

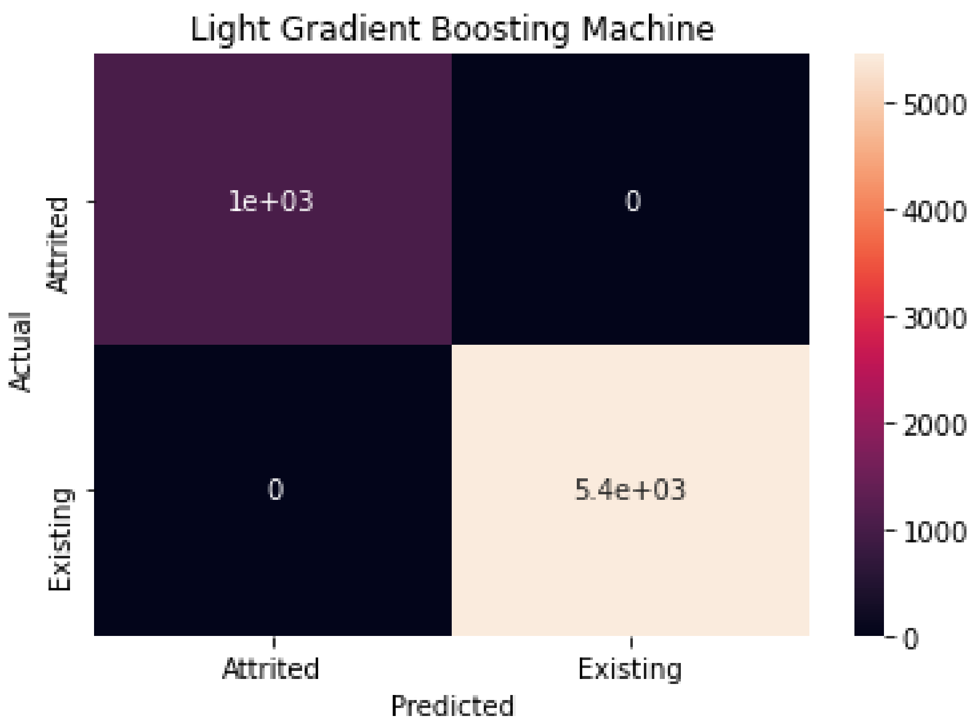
[124]:
```
lgbm.fit(X_train, y_train)
```

[124]: LGBMClassifier()

[125]:
```
scores4 = cross_val_score(lgbm, X_train, y_train, cv=5)
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores4.
 ↪mean()*100, scores4.std()))
```

```
96.96 accuracy with a standard deviation of 0.00
```

[267]:
```
conf_matrix(lgbm, "Light Gradient Boosting Machine", y_train, X_train, "figures/
 ↪ML_LGBM.jpg")
```



8.0.1 Try it on test data

[127]:
```
class_report(lgbm, y_test, X_test)
```

```
              precision    recall  f1-score   support

           1       0.90      0.89      0.89       301
           0       0.98      0.98      0.98      1725

    accuracy                           0.97      2026
   macro avg       0.94      0.94      0.94      2026
weighted avg       0.97      0.97      0.97      2026
```

```
Accuracy_Score: 96.89042448173741 %
Recall: 89.03654485049833 %
```

9 5. Try Neural Network

9.0.1 Load the necessary Packages

[128]:
```python
import numpy as np
import keras
import tensorflow as tf
from keras.models import Sequential
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.utils import to_categorical
from tensorflow.keras import regularizers
from sklearn.preprocessing import MinMaxScaler
```

```
Using TensorFlow backend.
```

9.0.2 Scale the data using MinMaxScaler

The scaled data has actually been created in the ETL section, just in case the data has not been scaled, uncomment and execute the code below

[132]:
```python
# scaler = MinMaxScaler()
# X_train_scaled = scaler.fit_transform(X_train)
# X_test_scaled = scaler.fit_transform(X_test)
```

Just to check the shape of the scaled data

[133]:
```python
print(X_train_scaled.shape, 'and', X_test_scaled.shape)
```

```
(6480, 19) and (2026, 19)
```

9.0.3 Create a model

[134]:
```python
model1 = Sequential()
model1.add(Dense(256, kernel_regularizer=regularizers.l2(0.001), input_dim=19,␣
 ↪activation='relu'))
model1.add(Dropout(rate=0.2))
model1.add(Dense(128, kernel_regularizer=regularizers.l2(0.001),␣
 ↪activation='relu'))
model1.add(Dropout(rate=0.2))
model1.add(Dense(64, kernel_regularizer=regularizers.l2(0.001),␣
 ↪activation='relu'))
model1.add(Dropout(rate=0.2))
model1.add(Dense(32, kernel_regularizer=regularizers.l2(0.001),␣
 ↪activation='relu'))
```

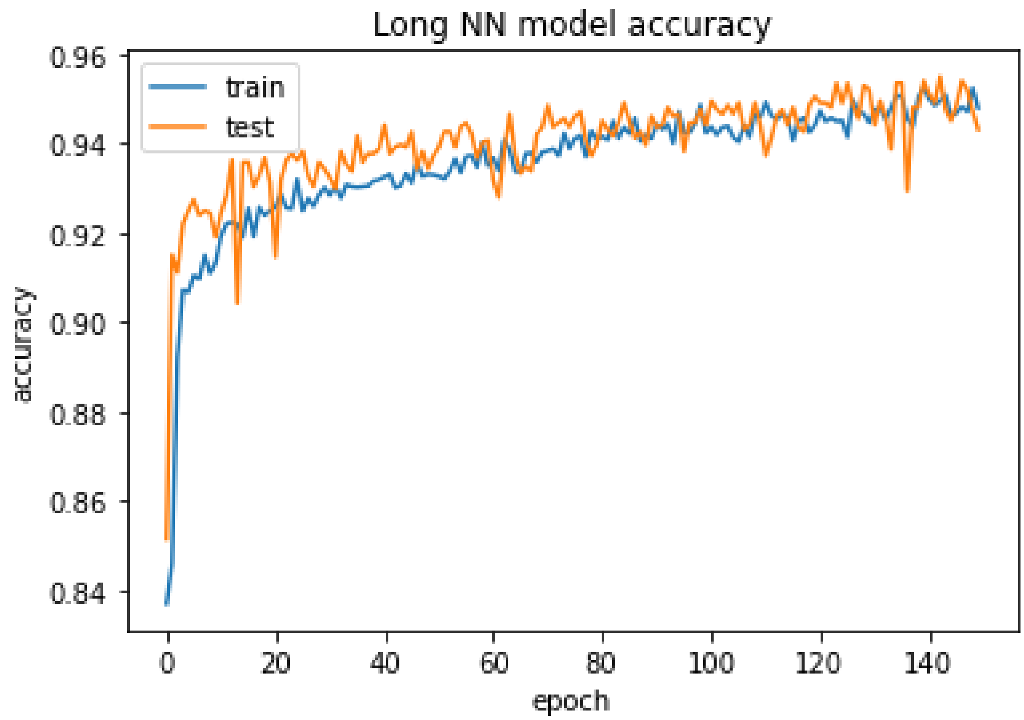
```
model1.add(Dropout(rate=0.2))
model1.add(Dense(16, kernel_regularizer=regularizers.l2(0.001),␣
 ↪activation='relu'))
model1.add(Dropout(rate=0.2))
model1.add(Dense(8, kernel_regularizer=regularizers.l2(0.001),␣
 ↪activation='relu'))
model1.add(Dropout(rate=0.1))
model1.add(Dense(1, activation='sigmoid'))
```

9.0.4 Compile the model and fit the model using the scaled data

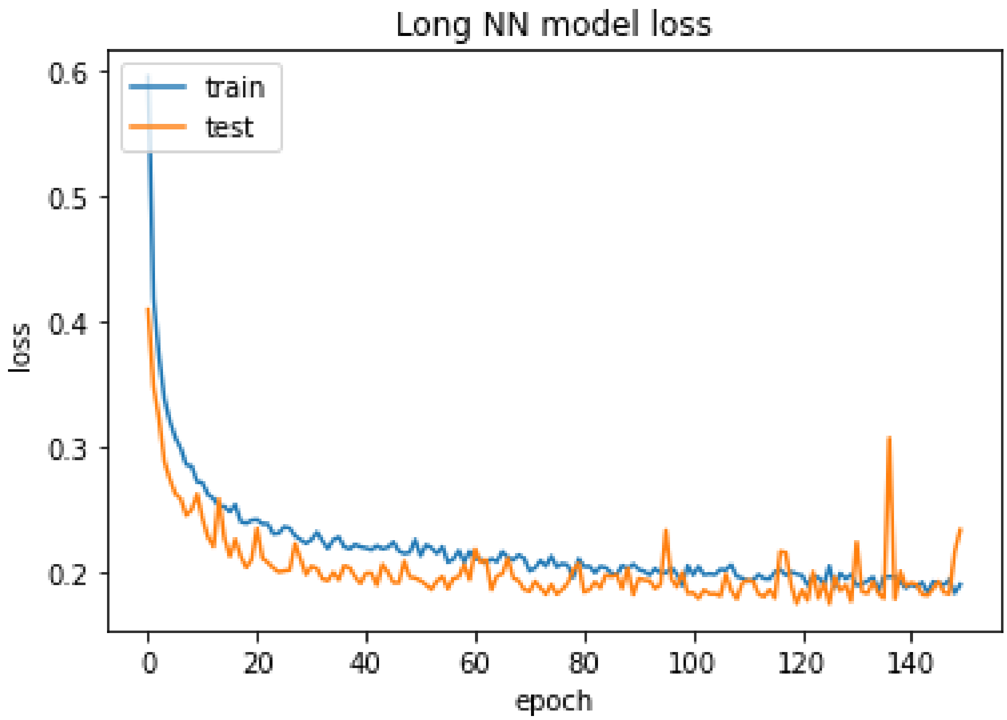
[135]:
```
model1.compile(loss = "binary_crossentropy",
                  optimizer = 'adam',
                  metrics=['accuracy'])



history = model1.fit(X_train_scaled, y_train, validation_data=(X_test_scaled,␣
 ↪y_test), epochs=150, batch_size=32, verbose=0)
history
score = model1.evaluate(X_test_scaled, y_test, verbose=0)
```

9.0.5 Plot the test vs train accuracy

[283]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Long NN model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig("figures/LongNN_acc.jpg")
plt.show()
```



[284]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Long NN model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig("figures/LongNN_loss.jpg")
plt.show()
```



9.0.6 Save the model1 to json and hdf5 file

[138]:
```
from keras.models import model_from_json
```

[139]:
```
# serialize model to JSON
model_json = model1.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)

# serialize weights to HDF5
model1.save_weights("model.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

9.0.7 Load The model

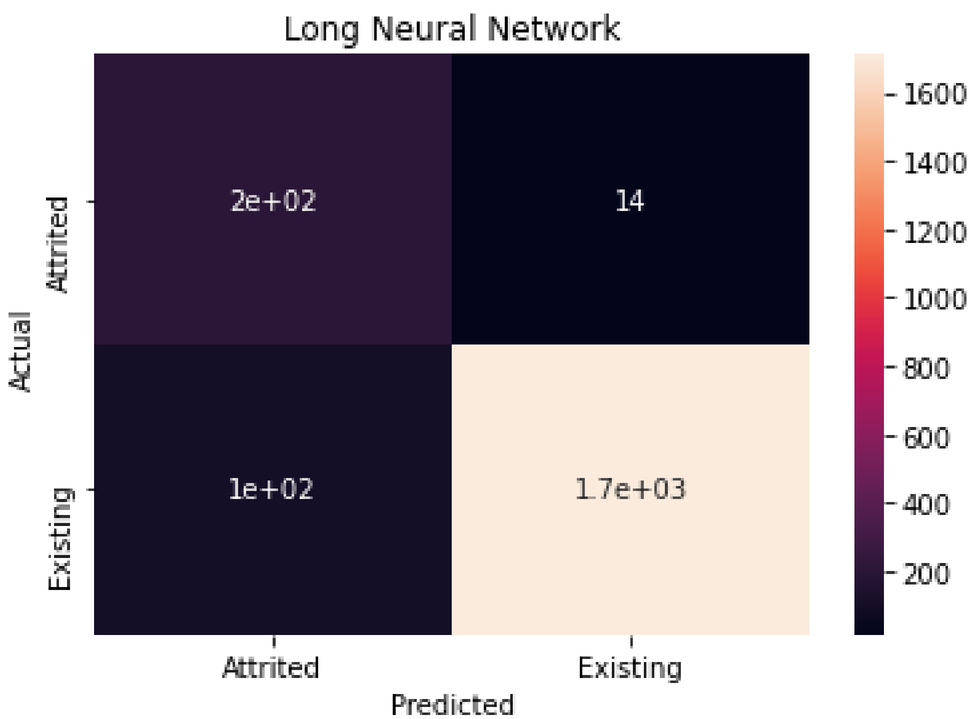
[140]:
```
# load json and create model
json_file = open('model.json', 'r')
nn_model_json = json_file.read()
json_file.close()
nn_model = model_from_json(nn_model_json)
# load weights into new model
```

```
nn_model.load_weights("model.h5")
print("Loaded model from disk")
```

```
Loaded model from disk
```

9.0.8 Evaluate using Loaded Model

[285]:
```
# Evaluation:
yprednn=nn_model.predict(X_test_scaled)
yprednn=yprednn.round()


cf = metrics.confusion_matrix(yprednn, y_test, labels=[1,0])
x_axis_labels=['Attrited','Existing']
y_axis_labels=['Attrited','Existing']

ax = plt.axes()
sns.heatmap(cf, ax = ax,annot=True, xticklabels=x_axis_labels,␣
 ↪yticklabels=y_axis_labels)
ax.set_title('Long Neural Network')
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
plt.savefig("figures/NN_LongNN_conf_matrix.jpg")
plt.show()
```



[142]:
```
print('Neural Network:\n {}\n'.format(
    metrics.classification_report(yprednn, y_test)))

nn_conf_matrix=metrics.confusion_matrix(yprednn,y_test)
conf_mat_nn = pd.DataFrame(nn_conf_matrix,
    columns=["Predicted Existing", "Predicted Attrited"],
    index=["Actual Existing", "Actual Attrited"])
print(conf_mat_nn)
```

```
Neural Network:
               precision    recall  f1-score   support

         0.0       0.99      0.94      0.97      1812
         1.0       0.66      0.93      0.78       214

    accuracy                           0.94      2026
   macro avg       0.83      0.94      0.87      2026
weighted avg       0.96      0.94      0.95      2026


                 Predicted Existing  Predicted Attrited
Actual Existing                1711                 101
Actual Attrited                  14                 200
```

9.1 6. Try a wider Neural Network model

[143]:
```
model2 = Sequential()
model2.add(Dense(256, kernel_regularizer=regularizers.l2(0.001), input_dim=19,␣
 ↪activation='relu'))
model2.add(Dropout(rate=0.2))
model2.add(Dense(512, kernel_regularizer=regularizers.l2(0.001),␣
 ↪activation='relu'))
model2.add(Dropout(rate=0.2))
model2.add(Dense(64, kernel_regularizer=regularizers.l2(0.001),␣
 ↪activation='relu'))
model2.add(Dropout(rate=0.1))
model2.add(Dense(1, activation='sigmoid'))
```

[144]:
```
model2.compile(loss = "binary_crossentropy",
                  optimizer = 'adam',
                  metrics=['accuracy'])


history2 = model2.fit(X_train_scaled, y_train, validation_data=(X_test_scaled,␣
 ↪y_test), epochs=150, batch_size=32, verbose=1)
```

```
history2
score2 = model2.evaluate(X_test_scaled, y_test, verbose=0)
```

```
Epoch 1/150
203/203 [==============================] - 3s 10ms/step - loss: 0.6705 -
accuracy: 0.8360 - val_loss: 0.3385 - val_accuracy: 0.9087
Epoch 2/150
203/203 [==============================] - 2s 9ms/step - loss: 0.3622 -
accuracy: 0.8922 - val_loss: 0.2836 - val_accuracy: 0.9141
Epoch 3/150
203/203 [==============================] - 2s 9ms/step - loss: 0.3197 -
accuracy: 0.8947 - val_loss: 0.2579 - val_accuracy: 0.9186
Epoch 4/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2862 -
accuracy: 0.9064 - val_loss: 0.2471 - val_accuracy: 0.9235
Epoch 5/150
203/203 [==============================] - 2s 10ms/step - loss: 0.2729 -
accuracy: 0.9094 - val_loss: 0.2465 - val_accuracy: 0.9215
Epoch 6/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2724 -
accuracy: 0.9121 - val_loss: 0.2604 - val_accuracy: 0.9176
Epoch 7/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2621 -
accuracy: 0.9128 - val_loss: 0.2284 - val_accuracy: 0.9225
Epoch 8/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2439 -
accuracy: 0.9185 - val_loss: 0.2279 - val_accuracy: 0.9250
Epoch 9/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2494 -
accuracy: 0.9125 - val_loss: 0.2252 - val_accuracy: 0.9279
Epoch 10/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2449 -
accuracy: 0.9160 - val_loss: 0.2144 - val_accuracy: 0.9314
Epoch 11/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2419 -
accuracy: 0.9205 - val_loss: 0.2180 - val_accuracy: 0.9225
Epoch 12/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2323 -
accuracy: 0.9235 - val_loss: 0.2047 - val_accuracy: 0.9319
Epoch 13/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2305 -
accuracy: 0.9235 - val_loss: 0.2110 - val_accuracy: 0.9304
Epoch 14/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2371 -
accuracy: 0.9182 - val_loss: 0.2094 - val_accuracy: 0.9344
Epoch 15/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2315 -
```

```
accuracy: 0.9227 - val_loss: 0.2096 - val_accuracy: 0.9329
Epoch 16/150
203/203 [==============================] - 2s 10ms/step - loss: 0.2045 -
accuracy: 0.9344 - val_loss: 0.2063 - val_accuracy: 0.9344
Epoch 17/150
203/203 [==============================] - 2s 10ms/step - loss: 0.2152 -
accuracy: 0.9328 - val_loss: 0.1949 - val_accuracy: 0.9358
Epoch 18/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2130 -
accuracy: 0.9313 - val_loss: 0.1979 - val_accuracy: 0.9339
Epoch 19/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2004 -
accuracy: 0.9339 - val_loss: 0.2124 - val_accuracy: 0.9309
Epoch 20/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2137 -
accuracy: 0.9267 - val_loss: 0.2275 - val_accuracy: 0.9260
Epoch 21/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2130 -
accuracy: 0.9301 - val_loss: 0.1929 - val_accuracy: 0.9373
Epoch 22/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2159 -
accuracy: 0.9291 - val_loss: 0.2015 - val_accuracy: 0.9353
Epoch 23/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2120 -
accuracy: 0.9284 - val_loss: 0.2017 - val_accuracy: 0.9319
Epoch 24/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2192 -
accuracy: 0.9285 - val_loss: 0.1866 - val_accuracy: 0.9378
Epoch 25/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2075 -
accuracy: 0.9302 - val_loss: 0.1991 - val_accuracy: 0.9358
Epoch 26/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2113 -
accuracy: 0.9261 - val_loss: 0.2065 - val_accuracy: 0.9299
Epoch 27/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2091 -
accuracy: 0.9294 - val_loss: 0.1939 - val_accuracy: 0.9368
Epoch 28/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2120 -
accuracy: 0.9282 - val_loss: 0.1906 - val_accuracy: 0.9378
Epoch 29/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2090 -
accuracy: 0.9318 - val_loss: 0.1892 - val_accuracy: 0.9373
Epoch 30/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2041 -
accuracy: 0.9301 - val_loss: 0.1874 - val_accuracy: 0.9358
Epoch 31/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2158 -
```

```
accuracy: 0.9286 - val_loss: 0.1812 - val_accuracy: 0.9393
Epoch 32/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1975 -
accuracy: 0.9420 - val_loss: 0.1820 - val_accuracy: 0.9432
Epoch 33/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1968 -
accuracy: 0.9356 - val_loss: 0.1882 - val_accuracy: 0.9373
Epoch 34/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2010 -
accuracy: 0.9350 - val_loss: 0.2075 - val_accuracy: 0.9383
Epoch 35/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1935 -
accuracy: 0.9395 - val_loss: 0.1920 - val_accuracy: 0.9314
Epoch 36/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1950 -
accuracy: 0.9370 - val_loss: 0.1863 - val_accuracy: 0.9413
Epoch 37/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1878 -
accuracy: 0.9380 - val_loss: 0.1819 - val_accuracy: 0.9373
Epoch 38/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1996 -
accuracy: 0.9374 - val_loss: 0.2177 - val_accuracy: 0.9294
Epoch 39/150
203/203 [==============================] - 2s 10ms/step - loss: 0.1898 -
accuracy: 0.9391 - val_loss: 0.1895 - val_accuracy: 0.9358
Epoch 40/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1983 -
accuracy: 0.9344 - val_loss: 0.1958 - val_accuracy: 0.9353
Epoch 41/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2006 -
accuracy: 0.9322 - val_loss: 0.1876 - val_accuracy: 0.9339
Epoch 42/150
203/203 [==============================] - 2s 10ms/step - loss: 0.1946 -
accuracy: 0.9350 - val_loss: 0.1775 - val_accuracy: 0.9447
Epoch 43/150
203/203 [==============================] - 2s 11ms/step - loss: 0.1951 -
accuracy: 0.9373 - val_loss: 0.1780 - val_accuracy: 0.9408
Epoch 44/150
203/203 [==============================] - 2s 11ms/step - loss: 0.1857 -
accuracy: 0.9427 - val_loss: 0.1919 - val_accuracy: 0.9334
Epoch 45/150
203/203 [==============================] - 2s 10ms/step - loss: 0.1971 -
accuracy: 0.9385 - val_loss: 0.1904 - val_accuracy: 0.9363
Epoch 46/150
203/203 [==============================] - 2s 11ms/step - loss: 0.1903 -
accuracy: 0.9378 - val_loss: 0.1984 - val_accuracy: 0.9339
Epoch 47/150
203/203 [==============================] - 2s 9ms/step - loss: 0.2086 -
```

```
accuracy: 0.9367 - val_loss: 0.1795 - val_accuracy: 0.9408
Epoch 48/150
203/203 [==============================] - 2s 11ms/step - loss: 0.1881 -
accuracy: 0.9398 - val_loss: 0.1968 - val_accuracy: 0.9398
Epoch 49/150
203/203 [==============================] - 2s 12ms/step - loss: 0.1839 -
accuracy: 0.9440 - val_loss: 0.1797 - val_accuracy: 0.9432
Epoch 50/150
203/203 [==============================] - 2s 10ms/step - loss: 0.1975 -
accuracy: 0.9378 - val_loss: 0.1897 - val_accuracy: 0.9378
Epoch 51/150
203/203 [==============================] - 2s 10ms/step - loss: 0.1851 -
accuracy: 0.9416 - val_loss: 0.1830 - val_accuracy: 0.9383
Epoch 52/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1901 -
accuracy: 0.9379 - val_loss: 0.2007 - val_accuracy: 0.9284
Epoch 53/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1889 -
accuracy: 0.9435 - val_loss: 0.1830 - val_accuracy: 0.9383
Epoch 54/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1853 -
accuracy: 0.9422 - val_loss: 0.1890 - val_accuracy: 0.9388
Epoch 55/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2015 -
accuracy: 0.9376 - val_loss: 0.1997 - val_accuracy: 0.9368
Epoch 56/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1894 -
accuracy: 0.9400 - val_loss: 0.1895 - val_accuracy: 0.9378
Epoch 57/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1877 -
accuracy: 0.9432 - val_loss: 0.1777 - val_accuracy: 0.9442
Epoch 58/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1940 -
accuracy: 0.9373 - val_loss: 0.2018 - val_accuracy: 0.9373
Epoch 59/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1901 -
accuracy: 0.9389 - val_loss: 0.1901 - val_accuracy: 0.9388
Epoch 60/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1867 -
accuracy: 0.9425 - val_loss: 0.1777 - val_accuracy: 0.9437
Epoch 61/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1857 -
accuracy: 0.9439 - val_loss: 0.1923 - val_accuracy: 0.9408
Epoch 62/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1933 -
accuracy: 0.9365 - val_loss: 0.1782 - val_accuracy: 0.9398
Epoch 63/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1903 -
```

```
accuracy: 0.9425 - val_loss: 0.1744 - val_accuracy: 0.9462
Epoch 64/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1913 -
accuracy: 0.9367 - val_loss: 0.1726 - val_accuracy: 0.9423
Epoch 65/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1780 -
accuracy: 0.9457 - val_loss: 0.1777 - val_accuracy: 0.9408
Epoch 66/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1825 -
accuracy: 0.9442 - val_loss: 0.1832 - val_accuracy: 0.9393
Epoch 67/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1882 -
accuracy: 0.9407 - val_loss: 0.1893 - val_accuracy: 0.9408
Epoch 68/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1936 -
accuracy: 0.9379 - val_loss: 0.2054 - val_accuracy: 0.9329
Epoch 69/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1828 -
accuracy: 0.9439 - val_loss: 0.1758 - val_accuracy: 0.9447
Epoch 70/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1826 -
accuracy: 0.9435 - val_loss: 0.1802 - val_accuracy: 0.9413
Epoch 71/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1757 -
accuracy: 0.9459 - val_loss: 0.1868 - val_accuracy: 0.9383
Epoch 72/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1808 -
accuracy: 0.9459 - val_loss: 0.1761 - val_accuracy: 0.9437
Epoch 73/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1845 -
accuracy: 0.9417 - val_loss: 0.1780 - val_accuracy: 0.9432
Epoch 74/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1743 -
accuracy: 0.9447 - val_loss: 0.2087 - val_accuracy: 0.9344
Epoch 75/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1815 -
accuracy: 0.9407 - val_loss: 0.1792 - val_accuracy: 0.9437
Epoch 76/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1808 -
accuracy: 0.9451 - val_loss: 0.1887 - val_accuracy: 0.9418
Epoch 77/150
203/203 [==============================] - 2s 8ms/step - loss: 0.2024 -
accuracy: 0.9331 - val_loss: 0.1814 - val_accuracy: 0.9462
Epoch 78/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1755 -
accuracy: 0.9452 - val_loss: 0.1796 - val_accuracy: 0.9437
Epoch 79/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1815 -
```

```
accuracy: 0.9458 - val_loss: 0.1739 - val_accuracy: 0.9447
Epoch 80/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1755 -
accuracy: 0.9474 - val_loss: 0.1792 - val_accuracy: 0.9447
Epoch 81/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1728 -
accuracy: 0.9436 - val_loss: 0.1805 - val_accuracy: 0.9418
Epoch 82/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1847 -
accuracy: 0.9412 - val_loss: 0.1735 - val_accuracy: 0.9462
Epoch 83/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1701 -
accuracy: 0.9455 - val_loss: 0.1809 - val_accuracy: 0.9452
Epoch 84/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1811 -
accuracy: 0.9439 - val_loss: 0.1706 - val_accuracy: 0.9487
Epoch 85/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1808 -
accuracy: 0.9471 - val_loss: 0.1708 - val_accuracy: 0.9492
Epoch 86/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1834 -
accuracy: 0.9448 - val_loss: 0.1777 - val_accuracy: 0.9427
Epoch 87/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1764 -
accuracy: 0.9470 - val_loss: 0.1688 - val_accuracy: 0.9506
Epoch 88/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1742 -
accuracy: 0.9467 - val_loss: 0.2097 - val_accuracy: 0.9363
Epoch 89/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1807 -
accuracy: 0.9443 - val_loss: 0.1837 - val_accuracy: 0.9427
Epoch 90/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1741 -
accuracy: 0.9470 - val_loss: 0.1678 - val_accuracy: 0.9477
Epoch 91/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1742 -
accuracy: 0.9452 - val_loss: 0.1814 - val_accuracy: 0.9442
Epoch 92/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1720 -
accuracy: 0.9503 - val_loss: 0.1697 - val_accuracy: 0.9467
Epoch 93/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1714 -
accuracy: 0.9510 - val_loss: 0.1814 - val_accuracy: 0.9398
Epoch 94/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1697 -
accuracy: 0.9511 - val_loss: 0.1907 - val_accuracy: 0.9383
Epoch 95/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1654 -
```

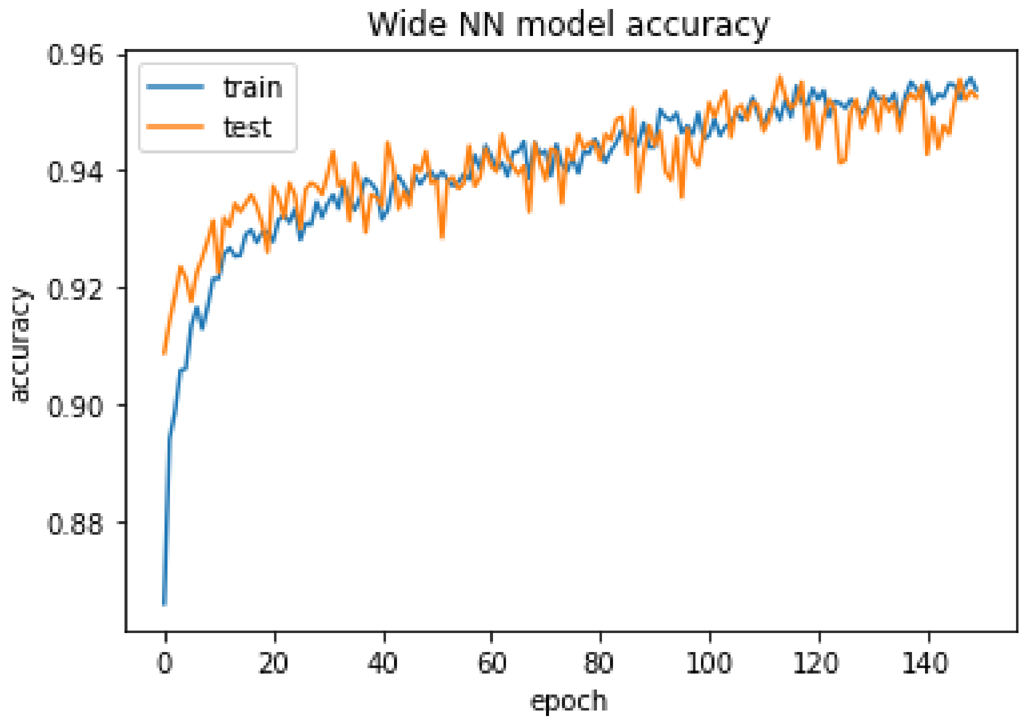
```
accuracy: 0.9523 - val_loss: 0.1793 - val_accuracy: 0.9457
Epoch 96/150
203/203 [==============================] - 2s 12ms/step - loss: 0.1814 -
accuracy: 0.9434 - val_loss: 0.2074 - val_accuracy: 0.9353
Epoch 97/150
203/203 [==============================] - 2s 12ms/step - loss: 0.1740 -
accuracy: 0.9458 - val_loss: 0.1762 - val_accuracy: 0.9472
Epoch 98/150
203/203 [==============================] - 2s 10ms/step - loss: 0.1733 -
accuracy: 0.9444 - val_loss: 0.1775 - val_accuracy: 0.9423
Epoch 99/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1719 -
accuracy: 0.9491 - val_loss: 0.1837 - val_accuracy: 0.9408
Epoch 100/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1802 -
accuracy: 0.9435 - val_loss: 0.1742 - val_accuracy: 0.9462
Epoch 101/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1712 -
accuracy: 0.9468 - val_loss: 0.1721 - val_accuracy: 0.9516
Epoch 102/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1705 -
accuracy: 0.9509 - val_loss: 0.1672 - val_accuracy: 0.9492
Epoch 103/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1674 -
accuracy: 0.9510 - val_loss: 0.1685 - val_accuracy: 0.9516
Epoch 104/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1747 -
accuracy: 0.9466 - val_loss: 0.1658 - val_accuracy: 0.9536
Epoch 105/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1676 -
accuracy: 0.9526 - val_loss: 0.1803 - val_accuracy: 0.9457
Epoch 106/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1741 -
accuracy: 0.9504 - val_loss: 0.1707 - val_accuracy: 0.9506
Epoch 107/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1592 -
accuracy: 0.9564 - val_loss: 0.1702 - val_accuracy: 0.9511
Epoch 108/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1745 -
accuracy: 0.9458 - val_loss: 0.1707 - val_accuracy: 0.9487
Epoch 109/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1648 -
accuracy: 0.9563 - val_loss: 0.1690 - val_accuracy: 0.9516
Epoch 110/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1651 -
accuracy: 0.9515 - val_loss: 0.1674 - val_accuracy: 0.9511
Epoch 111/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1679 -
```

```
accuracy: 0.9503 - val_loss: 0.1742 - val_accuracy: 0.9467
Epoch 112/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1735 -
accuracy: 0.9475 - val_loss: 0.1737 - val_accuracy: 0.9487
Epoch 113/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1731 -
accuracy: 0.9523 - val_loss: 0.1663 - val_accuracy: 0.9521
Epoch 114/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1755 -
accuracy: 0.9446 - val_loss: 0.1630 - val_accuracy: 0.9561
Epoch 115/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1635 -
accuracy: 0.9517 - val_loss: 0.1718 - val_accuracy: 0.9526
Epoch 116/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1776 -
accuracy: 0.9468 - val_loss: 0.1652 - val_accuracy: 0.9506
Epoch 117/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1518 -
accuracy: 0.9629 - val_loss: 0.1697 - val_accuracy: 0.9516
Epoch 118/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1679 -
accuracy: 0.9522 - val_loss: 0.1685 - val_accuracy: 0.9551
Epoch 119/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1650 -
accuracy: 0.9525 - val_loss: 0.1804 - val_accuracy: 0.9452
Epoch 120/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1630 -
accuracy: 0.9519 - val_loss: 0.1750 - val_accuracy: 0.9516
Epoch 121/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1653 -
accuracy: 0.9547 - val_loss: 0.2010 - val_accuracy: 0.9437
Epoch 122/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1669 -
accuracy: 0.9517 - val_loss: 0.1688 - val_accuracy: 0.9492
Epoch 123/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1739 -
accuracy: 0.9487 - val_loss: 0.1661 - val_accuracy: 0.9521
Epoch 124/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1666 -
accuracy: 0.9525 - val_loss: 0.1707 - val_accuracy: 0.9506
Epoch 125/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1687 -
accuracy: 0.9508 - val_loss: 0.1861 - val_accuracy: 0.9413
Epoch 126/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1789 -
accuracy: 0.9484 - val_loss: 0.1835 - val_accuracy: 0.9418
Epoch 127/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1674 -
```

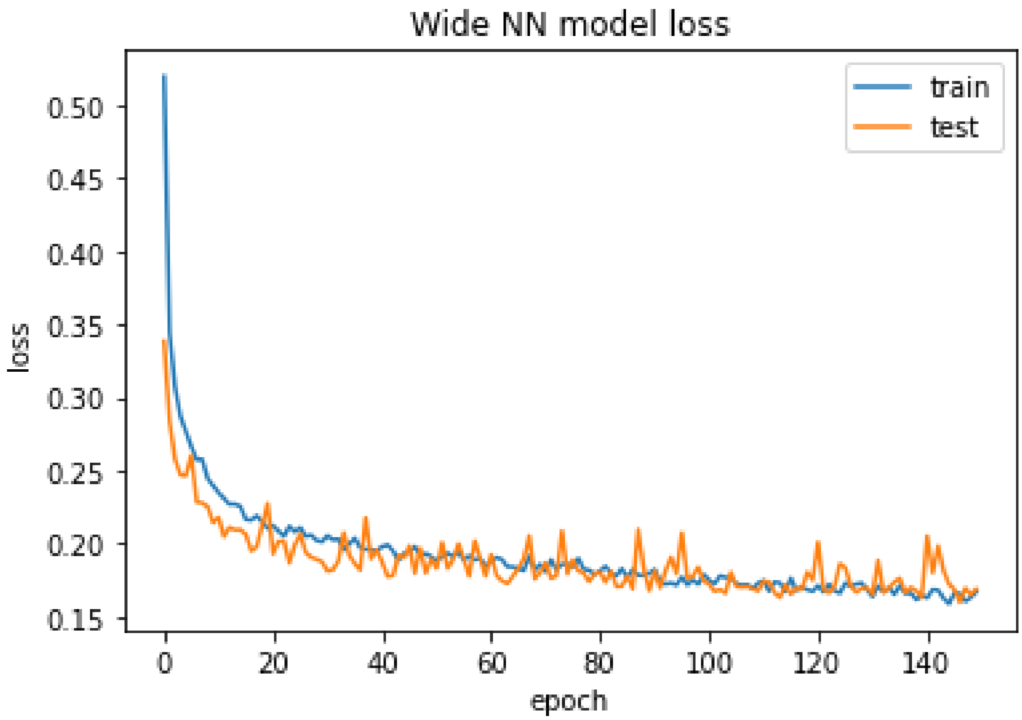
```
accuracy: 0.9528 - val_loss: 0.1707 - val_accuracy: 0.9501
Epoch 128/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1728 -
accuracy: 0.9498 - val_loss: 0.1666 - val_accuracy: 0.9521
Epoch 129/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1665 -
accuracy: 0.9544 - val_loss: 0.1693 - val_accuracy: 0.9472
Epoch 130/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1666 -
accuracy: 0.9524 - val_loss: 0.1697 - val_accuracy: 0.9501
Epoch 131/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1606 -
accuracy: 0.9539 - val_loss: 0.1652 - val_accuracy: 0.9521
Epoch 132/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1687 -
accuracy: 0.9542 - val_loss: 0.1885 - val_accuracy: 0.9467
Epoch 133/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1585 -
accuracy: 0.9561 - val_loss: 0.1662 - val_accuracy: 0.9526
Epoch 134/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1662 -
accuracy: 0.9536 - val_loss: 0.1697 - val_accuracy: 0.9501
Epoch 135/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1629 -
accuracy: 0.9548 - val_loss: 0.1735 - val_accuracy: 0.9516
Epoch 136/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1694 -
accuracy: 0.9470 - val_loss: 0.1762 - val_accuracy: 0.9467
Epoch 137/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1706 -
accuracy: 0.9510 - val_loss: 0.1666 - val_accuracy: 0.9516
Epoch 138/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1720 -
accuracy: 0.9492 - val_loss: 0.1698 - val_accuracy: 0.9531
Epoch 139/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1632 -
accuracy: 0.9532 - val_loss: 0.1677 - val_accuracy: 0.9521
Epoch 140/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1663 -
accuracy: 0.9507 - val_loss: 0.1630 - val_accuracy: 0.9546
Epoch 141/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1635 -
accuracy: 0.9549 - val_loss: 0.2051 - val_accuracy: 0.9427
Epoch 142/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1662 -
accuracy: 0.9537 - val_loss: 0.1801 - val_accuracy: 0.9492
Epoch 143/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1760 -
```

```
accuracy: 0.9488 - val_loss: 0.1984 - val_accuracy: 0.9437
Epoch 144/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1578 -
accuracy: 0.9555 - val_loss: 0.1815 - val_accuracy: 0.9477
Epoch 145/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1522 -
accuracy: 0.9578 - val_loss: 0.1721 - val_accuracy: 0.9462
Epoch 146/150
203/203 [==============================] - 2s 9ms/step - loss: 0.1715 -
accuracy: 0.9499 - val_loss: 0.1685 - val_accuracy: 0.9516
Epoch 147/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1681 -
accuracy: 0.9505 - val_loss: 0.1598 - val_accuracy: 0.9556
Epoch 148/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1660 -
accuracy: 0.9532 - val_loss: 0.1695 - val_accuracy: 0.9521
Epoch 149/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1578 -
accuracy: 0.9562 - val_loss: 0.1647 - val_accuracy: 0.9536
Epoch 150/150
203/203 [==============================] - 2s 8ms/step - loss: 0.1682 -
accuracy: 0.9542 - val_loss: 0.1689 - val_accuracy: 0.9526
```

[286]:
```
plt.plot(history2.history['accuracy'])
plt.plot(history2.history['val_accuracy'])
plt.title('Wide NN model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig("figures/NN_WideNN_acc.jpg")
plt.show()
```



[287]:
```
plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('Wide NN model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.savefig("figures/NN_WideNN_loss.jpg")
plt.show()
```



9.1.1 Save model 2 to a file

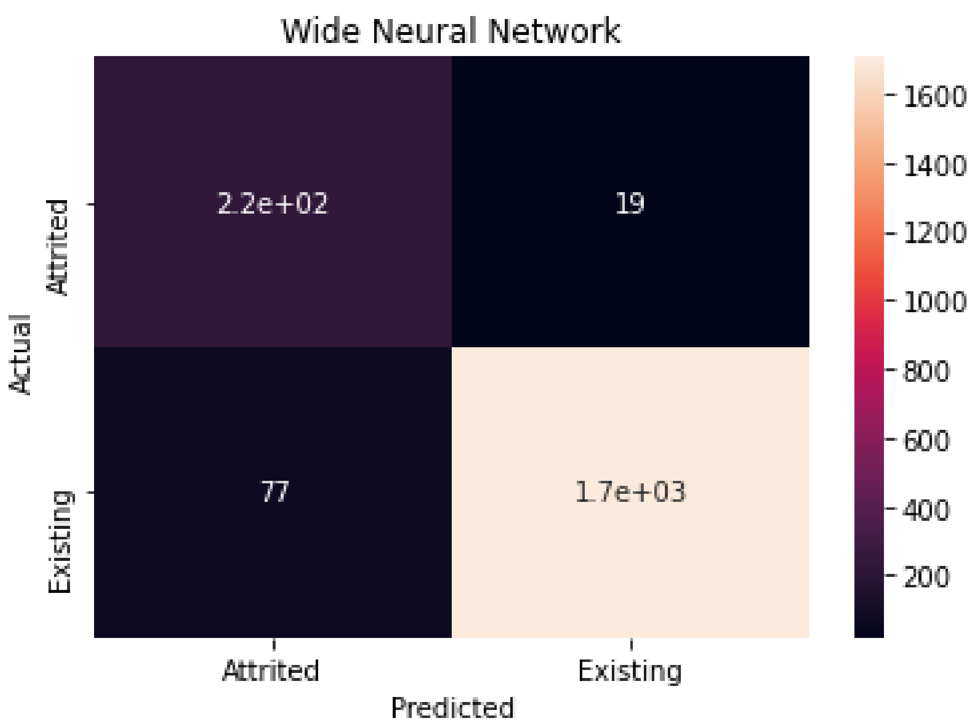
[288]:
```
# serialize model to JSON
model2_json = model2.to_json()
with open("model2.json", "w") as json_file:
    json_file.write(model2_json)

# serialize weights to HDF5
model2.save_weights("model2.h5")
print("Saved model2 to disk")
```

```
Saved model2 to disk
```

[289]:
```
# load json and create model
json_file2 = open('model2.json', 'r')
nn_model2_json = json_file2.read()
json_file2.close()
nn_model2 = model_from_json(nn_model2_json)
# load weights into new model
nn_model2.load_weights("model2.h5")
print("Loaded model from disk")
```

```
Loaded model from disk
```

[290]:
```
# Evaluation:
ypreddnn2=nn_model2.predict(X_test_scaled)
ypreddnn2=ypreddnn2.round()


cf2 = metrics.confusion_matrix(ypreddnn2, y_test, labels=[1,0])
x_axis_labels=['Attrited','Existing']
y_axis_labels=['Attrited','Existing']

ax = plt.axes()
sns.heatmap(cf2, ax = ax,annot=True, xticklabels=x_axis_labels,␣
↪yticklabels=y_axis_labels)
ax.set_title('Wide Neural Network')
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
plt.savefig("figures/NN_WideNN_conf_matrix.jpg")
plt.show()
```



[291]:
```
print('Neural Network:\n {}\n'.format(
    metrics.classification_report(ypreddnn2, y_test)))

nn_conf_matrix2=metrics.confusion_matrix(ypreddnn2,y_test)
conf_mat_nn2 = pd.DataFrame(nn_conf_matrix2,
```

```
    columns=["Predicted Existing", "Predicted Attrited"],
    index=["Actual Existing", "Actual Attrited"])
print(conf_mat_nn2)
```

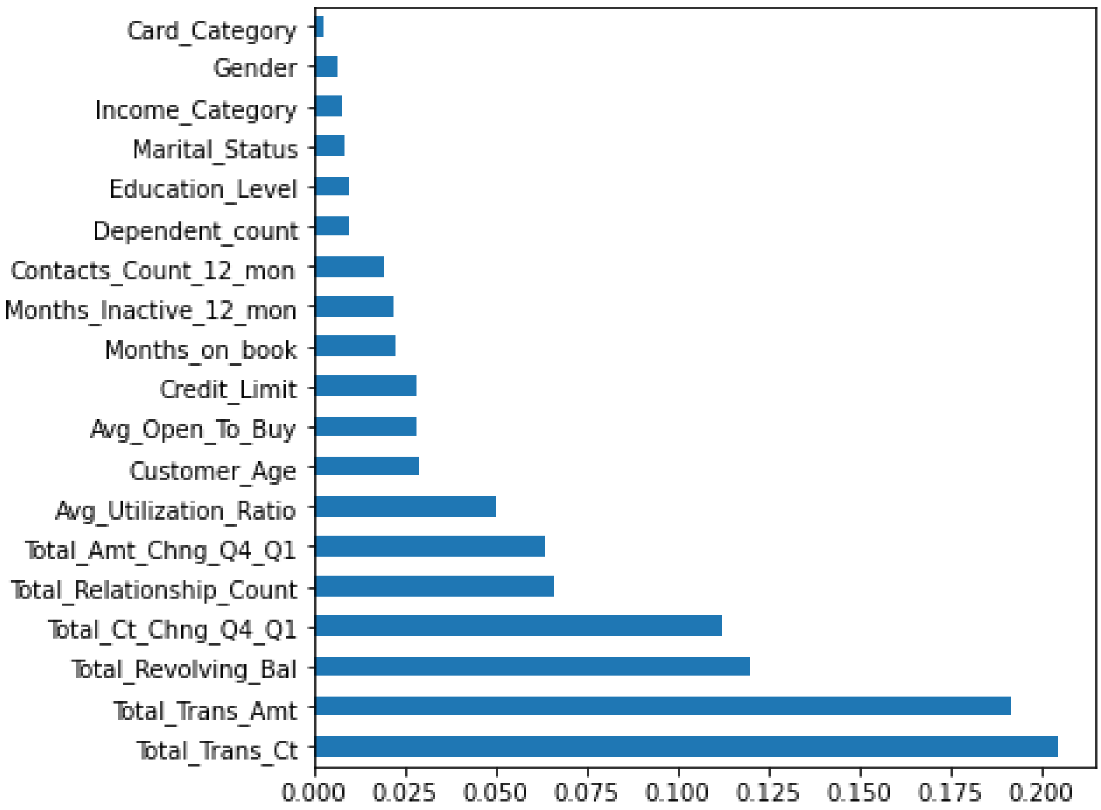
```
Neural Network:
               precision    recall  f1-score   support

         0.0       0.99      0.96      0.97      1783
         1.0       0.74      0.92      0.82       243

    accuracy                           0.95      2026
   macro avg       0.87      0.94      0.90      2026
weighted avg       0.96      0.95      0.95      2026



                 Predicted Existing  Predicted Attrited
Actual Existing                1706                  77
Actual Attrited                  19                 224
```

9.1.2 Find the most important Feature

[292]:
```
from sklearn import ensemble
```

[293]:
```
df_churn = pd.read_csv('df_churn.csv')
```

[294]:
```
rf =  ensemble.RandomForestClassifier(n_estimators=130,max_features=6,
 ↪n_jobs=-1)
rf.fit(X_train, y_train)
feature_importance = rf.feature_importances_
feat_importances = pd.Series(rf.feature_importances_, index=df_churn.iloc[:,1:].
 ↪columns)
feat_importances = feat_importances.nlargest(19)
feat_importances.plot(kind='barh' , figsize=(6,6))
plt.show()
```



10 SUMMARY

Create a dataframe containing All Algorithm with teir accuracy, f1 score and Recall

[295]:
```
from sklearn.metrics import f1_score, precision_score, recall_score,␣
↪confusion_matrix
```

[296]:
```
model_name = ['Random Forest','Logistic Regression', 'XGBoost', 'LGBM', 'Long␣
↪NN', 'Wide NN']
accuracy = []
f1 = []
recall = []

model_code = [rf, lr, xgb, lgbm, nn_model, nn_model2]

for i in model_code:
    if i == nn_model:
        accuracy.append(round(history.history['val_accuracy'][149]*100,2))
        f1.append(round(metrics.f1_score(y_test, yprednn)*100,2))
        recall.append(round(metrics.recall_score(y_test, yprednn)*100,2))
```

```python
    elif i == nn_model2:
        accuracy.append(round(history2.history['val_accuracy'][149]*100,2))
        f1.append(round(metrics.f1_score(y_test, yprednn2)*100,2))
        recall.append(round(metrics.recall_score(y_test, yprednn2)*100,2))
    else:
        accuracy.append(round(accuracy_score(y_test, i.predict(X_test))*100,2))
        f1.append(round(metrics.f1_score(y_test, i.predict(X_test))*100,2))
        recall.append(round(metrics.recall_score(y_test, i.
↪predict(X_test))*100,2))
```

[297]:
```python
summary = pd.DataFrame({'Model': model_name, 'Accuracy': accuracy, 'f1 Score':␣
↪f1, 'Recall': recall})
```

[298]:
```python
summary.sort_values(by='Accuracy', ascending=False)
```

[298]:

	Model	Accuracy	f1 Score	Recall
2	XGBoost	96.99	89.75	88.70
3	LGBM	96.89	89.48	89.04
0	Random Forest	96.25	86.90	83.72
5	Wide NN	95.26	82.35	74.42
4	Long NN	94.32	77.67	66.45
1	Logistic Regression	91.12	64.43	54.15

It can be seen that the best accuracy is XGBoost.

Surprisingly, wide NN, with more neuron per step performing better than the longer one

[ ]:
```python

```