ChurnAnalysis-ANN

January 2, 2023

0.0.1 Problem statement

Pinnacle Development Bank (PDB) is a leading retail bank in the country. The bank is facing several challenges in recent time such as Customer Churn, increment in NPA (Non-performing assets), low customer acquisition and high grievances, etc.

The cost of acquiring new customers is five times higher than the cost of retaining existing customers. Thus it is imperative to prevent customer churn. Mr. Anupam who is the head of sales decided to take help from the data science team to build a Machine learning model which will be capable to predict customer churn in advance.

****To prevent customer churn it is necessary to know which customer is going to churn. Unfortunately, domain experts can help to a very little extent in this case. We do not have the capability to predict anything in the future. With the development of computer science and statistical approaches, we can find the pattern, which would be helpful to predict it in advance.***

0.0.2 *Feature Details*

`CustomerID` - Customer identification Unique Id for each customer.

`DateOfBirth` - Date of birth of the customer

`Gender` - Gender of the customer

`City` - City where customer lives in

`AccountBalance` - Current account balance available in the account

`HavingFD` - If customer has FD, 0- No DF, 1- Yes FD(s) as there.

`HavingCC` - If customer has Credit Card, 0- No DF, 1- Yes FD(s) as there.

`CIBIL_Score` - Lies between 300-900 signifies customer's creditworthiness.

`HavingLoan` - If customer has Loan, 0- No DF, 1- Yes Loan as there.

`RV` - Total quanitative relationship values with the bank. Including all (AccountBalance, FD, Credit Card, Loan)

`BranchType` - Type of branch-Rural, Semi-urban and Urban based on the geograpihcal and Population)

`Churn` - value is 0 and 1. 1 means customer has left the bank 0 mean did not churn)

[2]:
```python
#importing required libraries-
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```python
#Setting the float format(don't want in scientific notation)
pd.set_option('display.float_format', lambda x: '%.3f' % x)
```

[4]:
```python
#Visualization format
sns.set_theme(context='notebook', style='darkgrid', palette='deep',␣
 ↪font='sans-serif',
                  font_scale=1, color_codes=True, rc=None)
```

[5]:
```python
#Don't want to print warning error messages
import warnings
warnings.filterwarnings('ignore')
```

[6]:
```python
#Importing the data
df = pd.read_csv('/kaggle/input/churn/churn.csv')
```

[7]:
```python
#Checking first 5 records
df.head()
```

[7]:
```
  CustomerID DateOfBirth Gender        City  AccountBalance  HavingFD  \
0   C7975142  01-02-1930      M  SALODARIYA             469         0
1   C7496772  15-07-1930      F     JHAJJAR           28544         1
2   C7541191  21-11-1930      M      KALYAN          107470         1
3   C6531683  09-04-1932      M      MARGAO           25829         0
4   C4715364  05-05-1932      M   CURCHOREM             216         0

   HavingCC  CIBIL_Score  HavingLoan      RV BranchType  Churn
0         0          833           0  163076      Rural      0
1         0          798           0  131878      Rural      0
2         0          602           1  161612      Rural      1
3         0          594           1   57874      Rural      0
4         0          808           1  184943      Rural      0
```

[8]:
```python
#Shape
df.shape
```

[8]: (332008, 12)

[9]:
```python
#Checking basic info
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 332008 entries, 0 to 332007
```

```
Data columns (total 12 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   CustomerID      332008 non-null  object
 1   DateOfBirth     331955 non-null  object
 2   Gender          332008 non-null  object
 3   City            332008 non-null  object
 4   AccountBalance  332008 non-null  int64
 5   HavingFD        332008 non-null  int64
 6   HavingCC        332008 non-null  int64
 7   CIBIL_Score     332008 non-null  int64
 8   HavingLoan      332008 non-null  int64
 9   RV              332008 non-null  int64
 10  BranchType      331962 non-null  object
 11  Churn           332008 non-null  int64
dtypes: int64(7), object(5)
memory usage: 30.4+ MB
```

[10]:
```
#Checking data-types
df.dtypes
```

[10]:
```
CustomerID        object
DateOfBirth       object
Gender            object
City              object
AccountBalance     int64
HavingFD           int64
HavingCC           int64
CIBIL_Score        int64
HavingLoan         int64
RV                 int64
BranchType        object
Churn              int64
dtype: object
```

We have 5 Object(Non Numerical) and 7 Numerical Datatype

[11]:
```
#Descibring the data
df.describe()
```

[11]:
```
       AccountBalance   HavingFD   HavingCC  CIBIL_Score  HavingLoan  \
count      332008.000 332008.000 332008.000   332008.000  332008.000
mean       103598.116      0.364      0.309      599.667       0.448
std        685848.257      0.481      0.462      173.392       0.497
min             0.000      0.000      0.000      300.000       0.000
25%          4617.750      0.000      0.000      449.000       0.000
50%         16188.000      0.000      0.000      600.000       0.000
75%         54066.000      1.000      1.000      750.000       1.000
```

```
max     164489264.000       1.000       1.000       899.000        1.000

                  RV      Churn
count    332008.000 332008.000
mean     198624.251      0.219
std      688073.388      0.413
min           0.000      0.000
25%       69274.500      0.000
50%      133492.500      0.000
75%      194690.250      0.000
max   164608203.000      1.000
```

0.0.3 *Observations:-*

- *We have 25% of customers whose account balance is more than 54066.00*
- *Average CIBIL Score is 600 which shows we have a good set of customers who is financially aware and pay all the due EMI/Credit card on time.*
- *25% of the customers have 750 or above CIBIL Score, It is very important to signify one's financial health.*
- *The standard deviation of RV (relationship value) is 688073.00 which is very high and it shows that few customers are very rich and others are poor. There is a significant difference in wealth distribution.*

[12]:
```python
#Checking null/missing values
df.isnull().sum()
```

[12]:
```
CustomerID         0
DateOfBirth       53
Gender             0
City               0
AccountBalance     0
HavingFD           0
HavingCC           0
CIBIL_Score        0
HavingLoan         0
RV                 0
BranchType        46
Churn              0
dtype: int64
```

There are some missing values in `DateOfBirth` and `BranchType` column which I'll impute during EDA

[13]:
```python
df.dtypes
```

[13]:
```
CustomerID        object
DateOfBirth       object
Gender            object
City              object
AccountBalance     int64
HavingFD           int64
HavingCC           int64
CIBIL_Score        int64
HavingLoan         int64
RV                 int64
BranchType        object
Churn              int64
dtype: object
```

0.0.4 *Feature Engineering & Exploratory Data Analysis(EDA)*

Feature Engineering is a process to use domain knowledge in order to derive a new feature or modify existing feature to extract maximum information from it.

In Feature Engineering I will perform below operations:-

1. *I will fetch `Age` using DateofBirth Column.*
2. *I will categories customers based on their `Age`.*

Exploratory data analysis is an approach of analyzing data sets to summarize their main characteristics, often using statistical graphics and other data visualization methods

I will perform below steps in EDA Section:-

1. *Duplicate data finding*
2. *Analyzing missing values(Missing value imputation).*
3. *Exploring all the features(Categorical & Numerical separately).*
4. *Finding Outliers.*
5. *Finding relationship(Correlation)*

0.0.5 *A. Feature Engineering*

[14]:
```python
df.columns
```

[14]:
```
Index(['CustomerID', 'DateOfBirth', 'Gender', 'City', 'AccountBalance',
       'HavingFD', 'HavingCC', 'CIBIL_Score', 'HavingLoan', 'RV', 'BranchType',
       'Churn'],
      dtype='object')
```

We can easily derive Age from the given DateOfBirth

[15]:
```python
#We need datetime package for that
from datetime import datetime
```

```
today = datetime.now()
```

[16]:
```
df['DateOfBirth'] = pd.to_datetime(df['DateOfBirth']) #Convering object to␣
↪Datetime

df['Age'] = (today - df['DateOfBirth']).apply(lambda x: np.round(x.days /␣
↪365,0)) #Getting age column
```

Now based on the age I will categorize as `minor` *< 18,* `Young` *, 18- 35* `Young-Adult`*, 36-60* `Adult`*, 60>*`Senior Citizen`

[17]:
```
age_categories = ['Minor', 'Young-Adult', 'Adult', 'Senior Citizen']
age_bins = [0, 18, 35, 60, 100]
```

[18]:
```
df['Category'] = pd.cut(df['Age'], age_bins, labels=age_categories)
```

[19]:
```
df.head()
```

[19]:
```
  CustomerID DateOfBirth Gender        City  AccountBalance  HavingFD  \
0   C7975142  1930-01-02      M  SALODARIYA             469         0
1   C7496772  1930-07-15      F     JHAJJAR           28544         1
2   C7541191  1930-11-21      M      KALYAN          107470         1
3   C6531683  1932-09-04      M      MARGAO           25829         0
4   C4715364  1932-05-05      M   CURCHOREM             216         0

   HavingCC  CIBIL_Score  HavingLoan      RV BranchType  Churn    Age  \
0         0          833           0  163076      Rural      0 93.000
1         0          798           0  131878      Rural      0 93.000
2         0          602           1  161612      Rural      1 92.000
3         0          594           1   57874      Rural      0 90.000
4         0          808           1  184943      Rural      0 91.000

         Category
0  Senior Citizen
1  Senior Citizen
2  Senior Citizen
3  Senior Citizen
4  Senior Citizen
```

Now since we get the age, I will drop `DateOfBirth`

[20]:
```
#before that I will create a copy to be in safer side.
df_new = df.copy()
```

[21]:
```
df_new.drop( 'DateOfBirth', axis = 1, inplace = True)
```

[22]:
```
df_new.head()
```

[22]:
```
  CustomerID Gender        City  AccountBalance  HavingFD  HavingCC  \
0   C7975142      M  SALODARIYA             469         0         0
1   C7496772      F     JHAJJAR           28544         1         0
2   C7541191      M      KALYAN          107470         1         0
3   C6531683      M      MARGAO           25829         0         0
4   C4715364      M   CURCHOREM             216         0         0

   CIBIL_Score  HavingLoan      RV BranchType  Churn     Age         Category
0          833           0  163076      Rural      0  93.000  Senior Citizen
1          798           0  131878      Rural      0  93.000  Senior Citizen
2          602           1  161612      Rural      1  92.000  Senior Citizen
3          594           1   57874      Rural      0  90.000  Senior Citizen
4          808           1  184943      Rural      0  91.000  Senior Citizen
```

0.0.6 *B. Exploratory Data Analysis(EDA)*

1. Finding duplicate data

[23]:
```python
df_new.duplicated().sum()
```

[23]: 2422

[24]:
```python
df_new.duplicated(subset= 'CustomerID').sum()
```

[24]: 47301

I will drop these duplicate customers.

[25]:
```python
df_new.drop_duplicates(subset= 'CustomerID', inplace= True)
```

I will also drop `CustomerID` and `City` because this is a static feature and doesn't have any significance in our analysis

[26]:
```python
df_new.drop( ['CustomerID','City'], axis = 1, inplace = True)
```

2. Analyzing missing values

[27]:
```python
df_new.isnull().sum()
```

[27]:
```
Gender             0
AccountBalance     0
HavingFD           0
HavingCC           0
CIBIL_Score        0
HavingLoan         0
RV                 0
BranchType        33
Churn              0
Age               48
Category          48
```

```
dtype: int64
```

*For **Age** I will impute mean value(as it is numerical feature) and for **BranchType** and use mode imputation*

[28]:
```
df_new['Age'] = df_new['Age'].fillna(value= df_new['Age'].mean())
df_new['BranchType'] = df_new['BranchType'].fillna(value= df_new['BranchType'].
 ↪mode()[0])
```

[29]:
```
#Defining the category after missing value imputation
df_new['Category'] = pd.cut(df_new['Age'], age_bins, labels=age_categories)
```

[30]:
```
df_new.isnull().sum()
```

[30]:
```
Gender            0
AccountBalance    0
HavingFD          0
HavingCC          0
CIBIL_Score       0
HavingLoan        0
RV                0
BranchType        0
Churn             0
Age               0
Category          0
dtype: int64
```

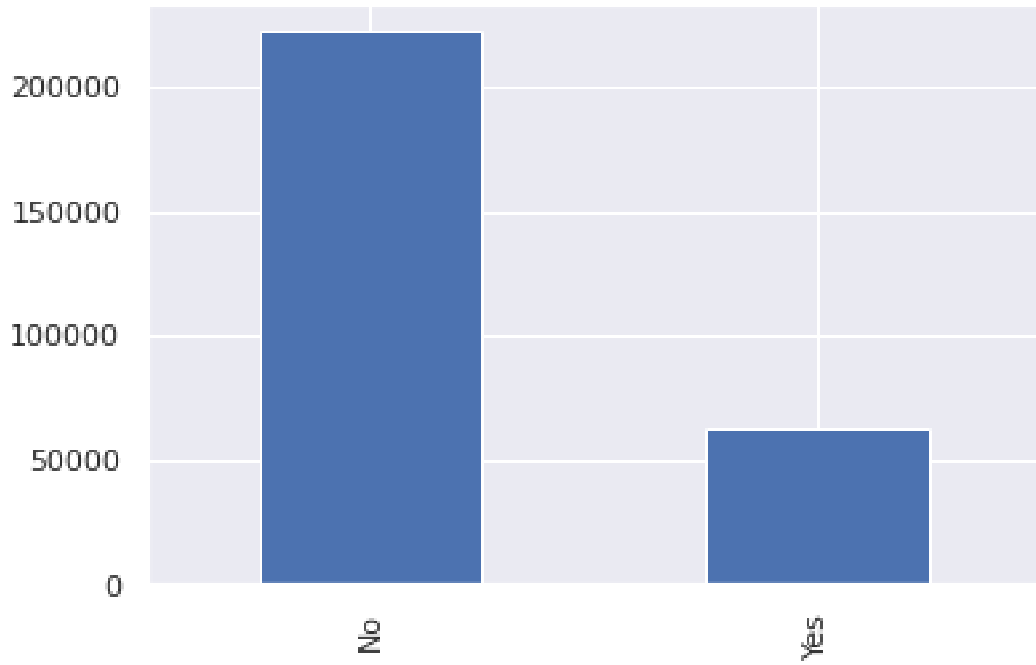
3. Feature Exploration.

[97]:
```
df_new2 = df_new.copy()
df_new2['Churned'] = np.where(df_new2['Churn'] == 1,'Yes', 'No')
```

[98]:
```
df_new2.drop('Churn', axis = 1, inplace = True)
df_new2.rename(columns = {'Churned' : 'Churn'}, inplace = True)
```

[99]:
```
df_new2['Churn'].value_counts().plot(kind = 'bar')
df_new2['Churn'].value_counts(normalize= True)
```

[99]:
```
No     0.781
Yes    0.219
Name: Churn, dtype: float64
```



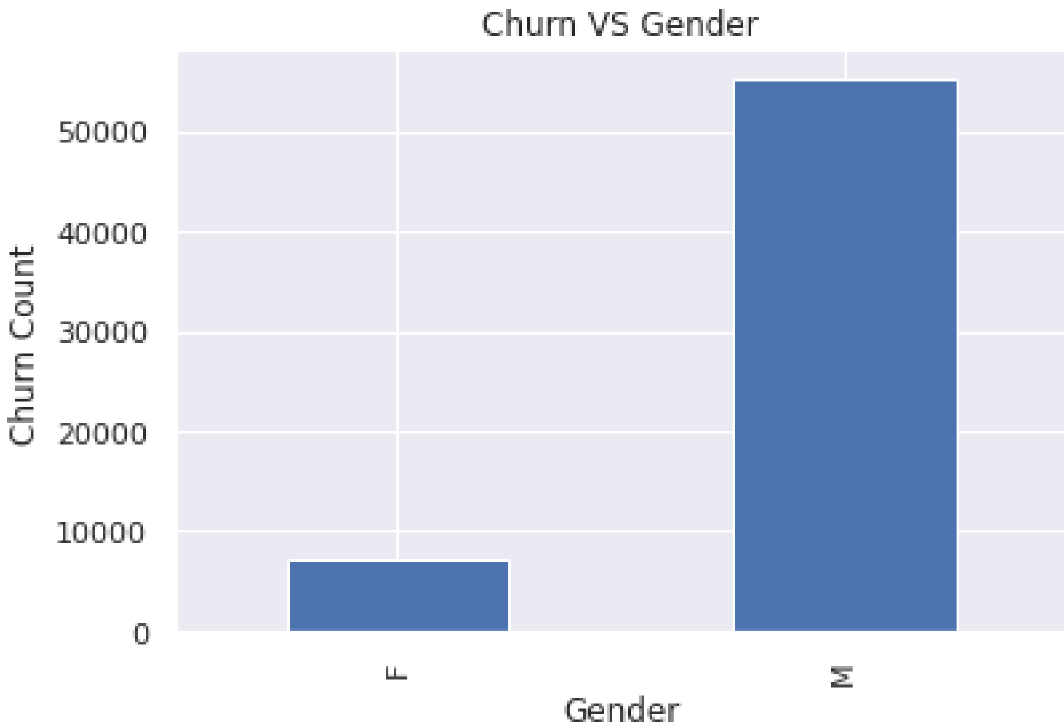
It is a highly imbalaced dataset, 78% is not churned and 21% churned

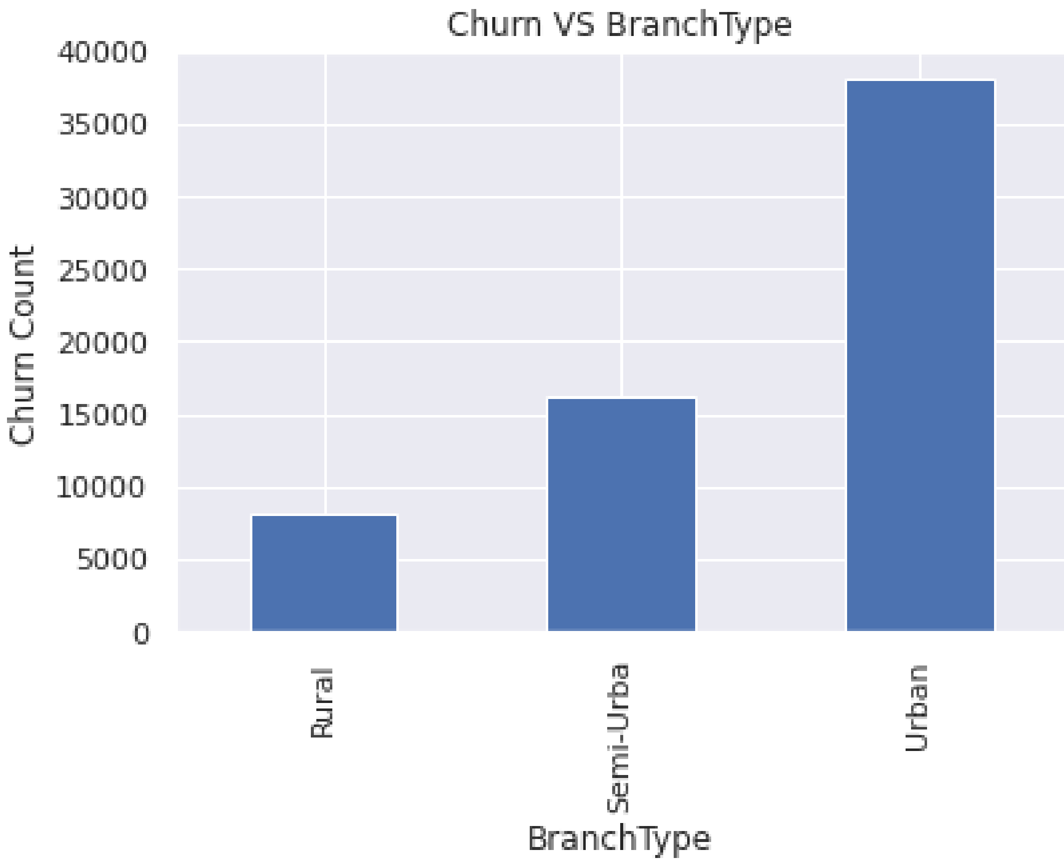
Categorical features

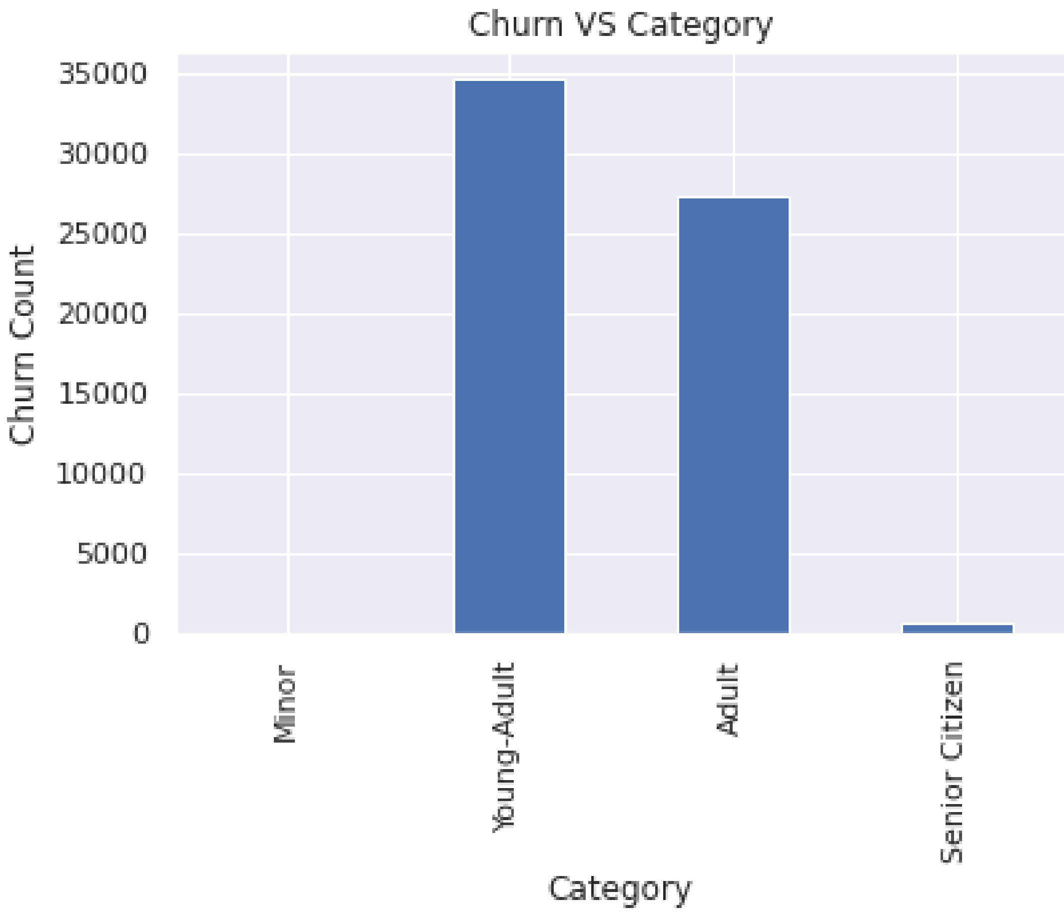
[100]:
```
categorical_features = [feature for feature in df_new.columns if␣
↪df_new[feature].dtypes == 'O' or df_new[feature].dtypes == 'category']
```

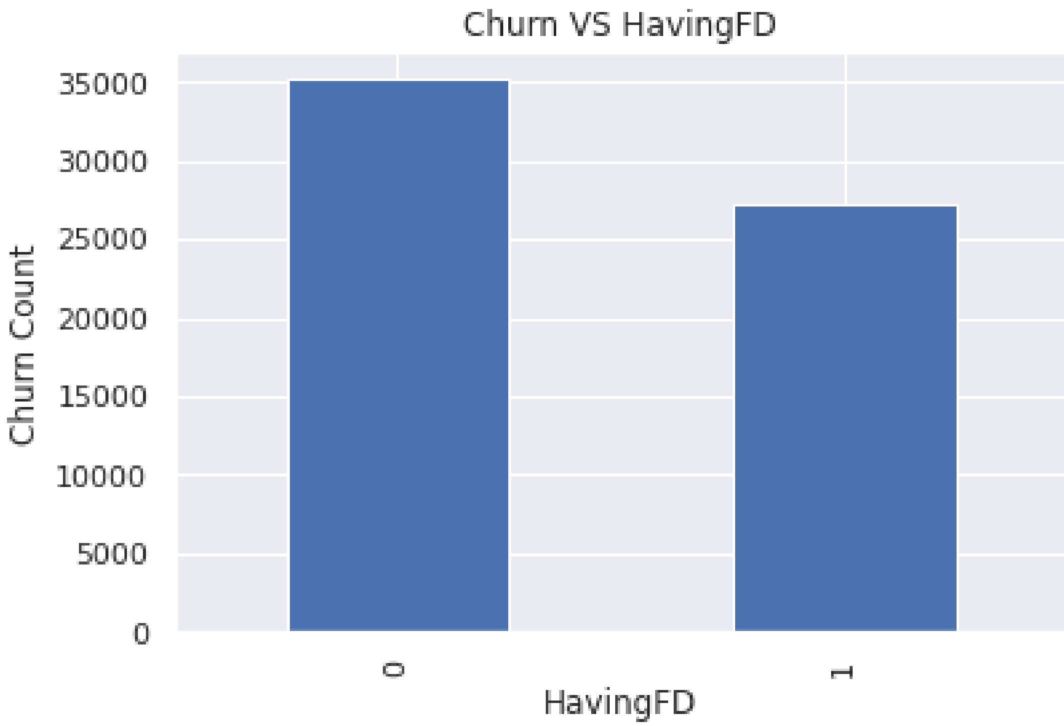
[101]:
```
categorical_features.extend(['HavingFD', 'HavingCC','HavingLoan'])
```

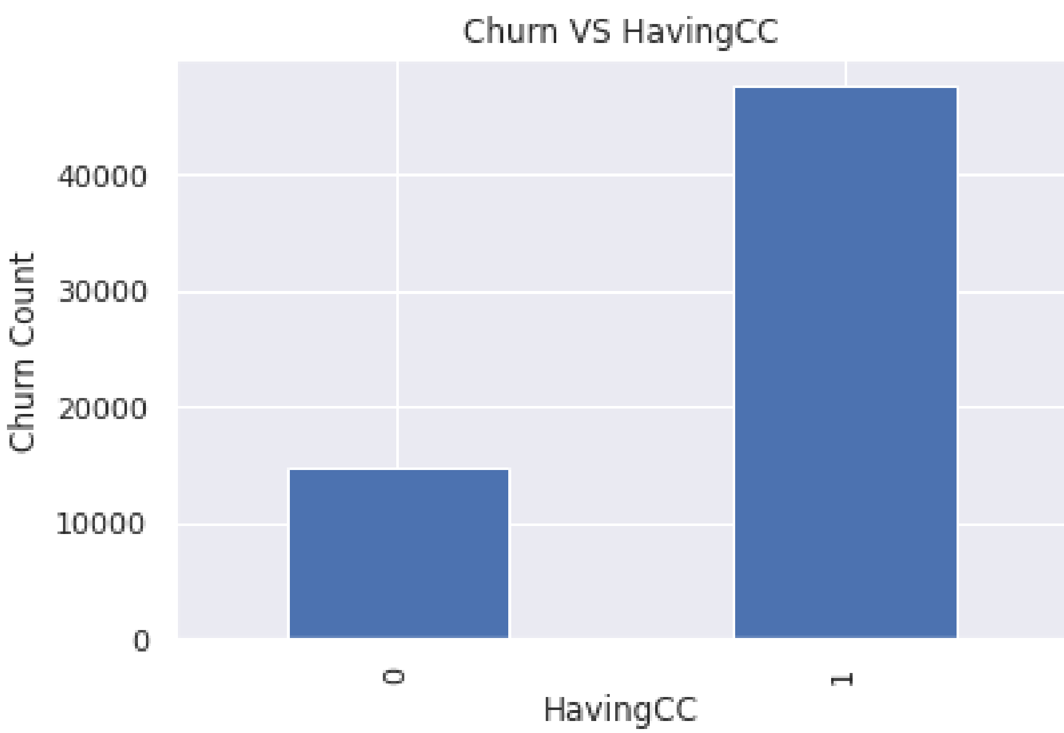
[107]:
```
for i in categorical_features:
    df_new2[(df_new2.Churn == 'Yes')].groupby(i)['Churn'].count().plot.bar()
    plt.xlabel(i)
    plt.ylabel('Churn Count')
    plt.title('Churn VS {}'.format(i))
    plt.show()
```

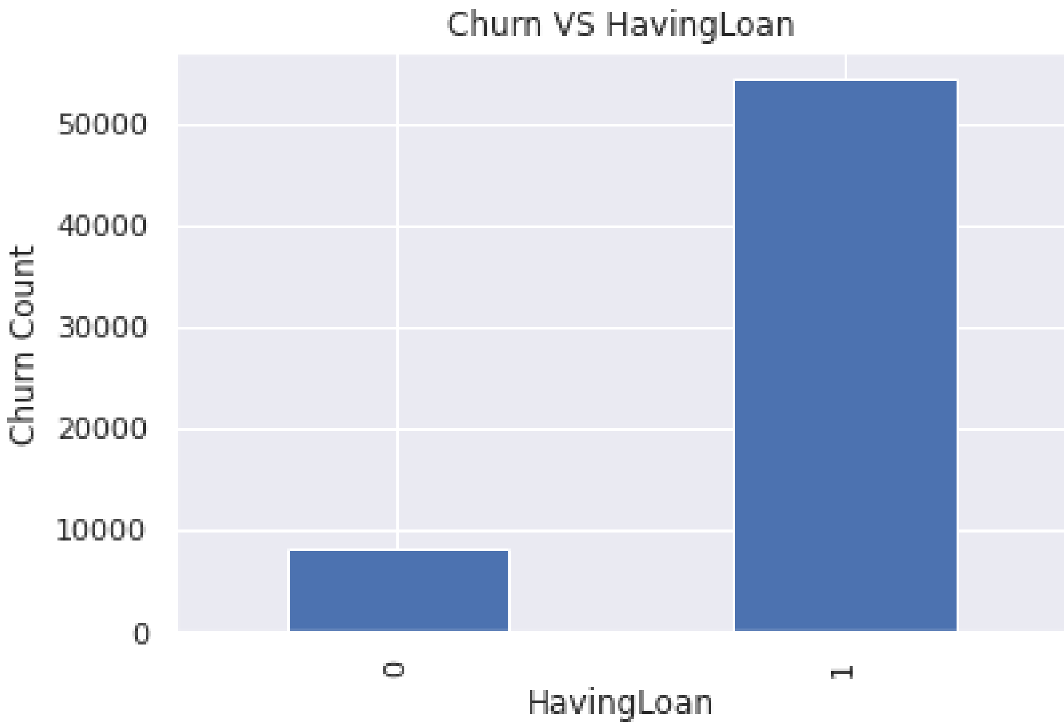

Churn VS Gender
Churn Count
50000
40000
30000
20000
10000
0
F
M
Gender


Churn VS BranchType
Churn Count
0
5000
10000
15000
20000
25000
30000
35000
40000
Rural
Semi-Urba
Urban
BranchType


Churn VS Category
Churn Count
35000
30000
25000
20000
15000
10000
5000
0
Minor
Young-Adult
Adult
Senior Citizen
Category


Churn VS HavingFD
Churn Count
35000
30000
25000
20000
15000
10000
5000
0
0
1
HavingFD


Churn VS HavingCC
Churn Count
40000
30000
20000
10000
0
0
1
HavingCC

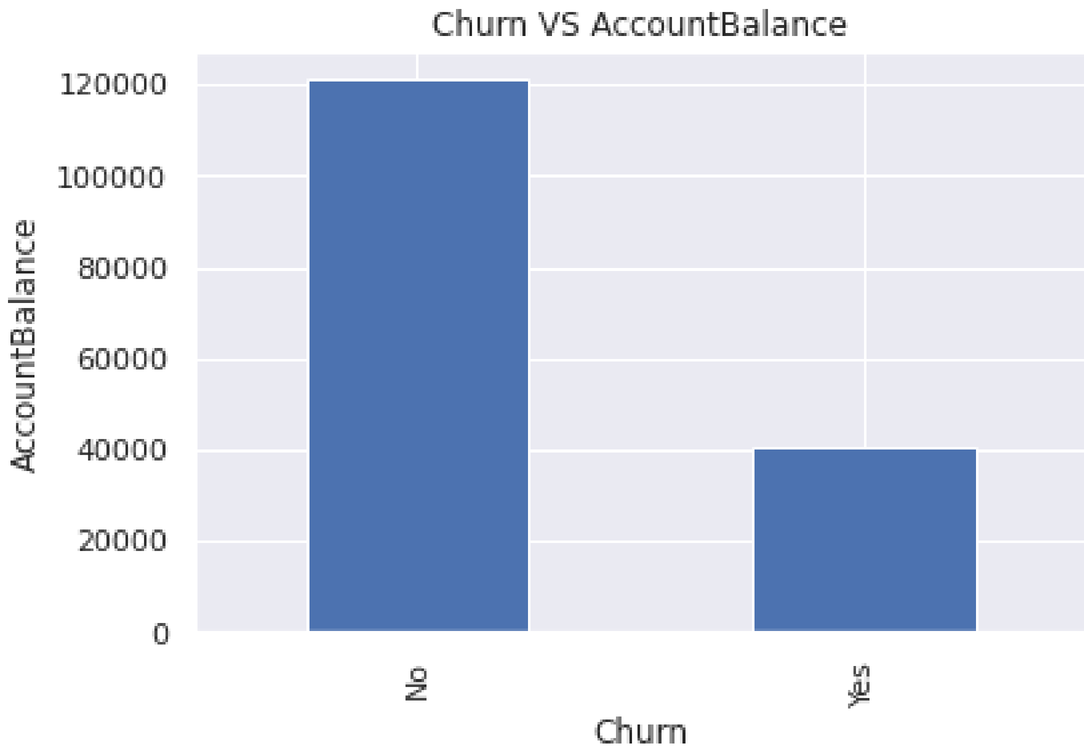


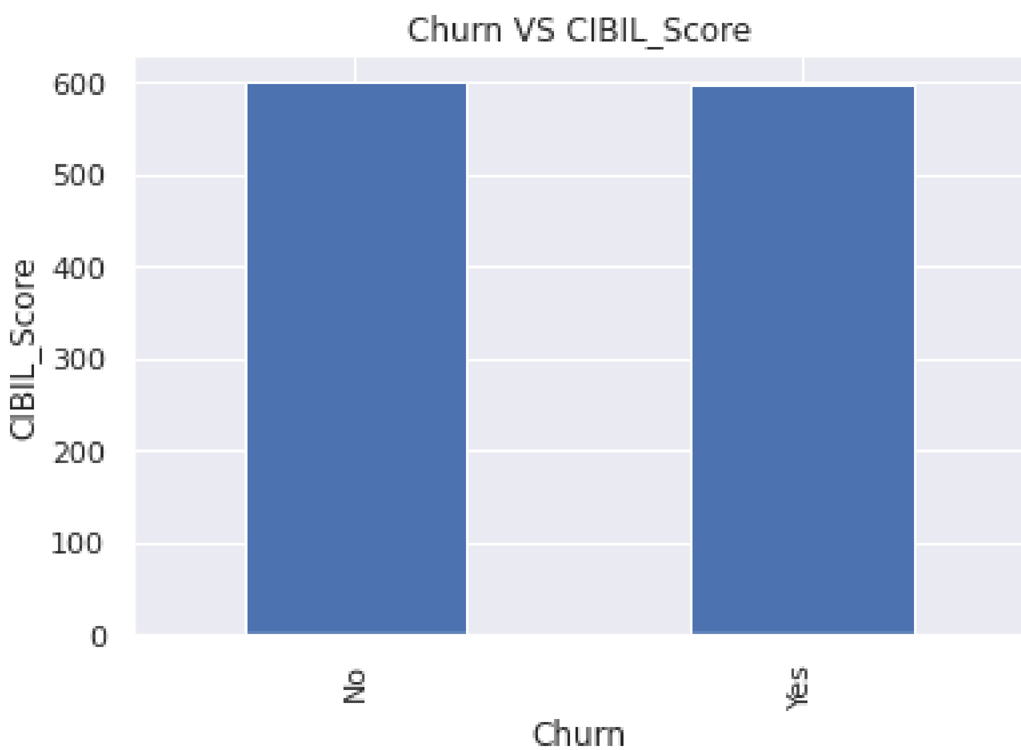
Observations:-

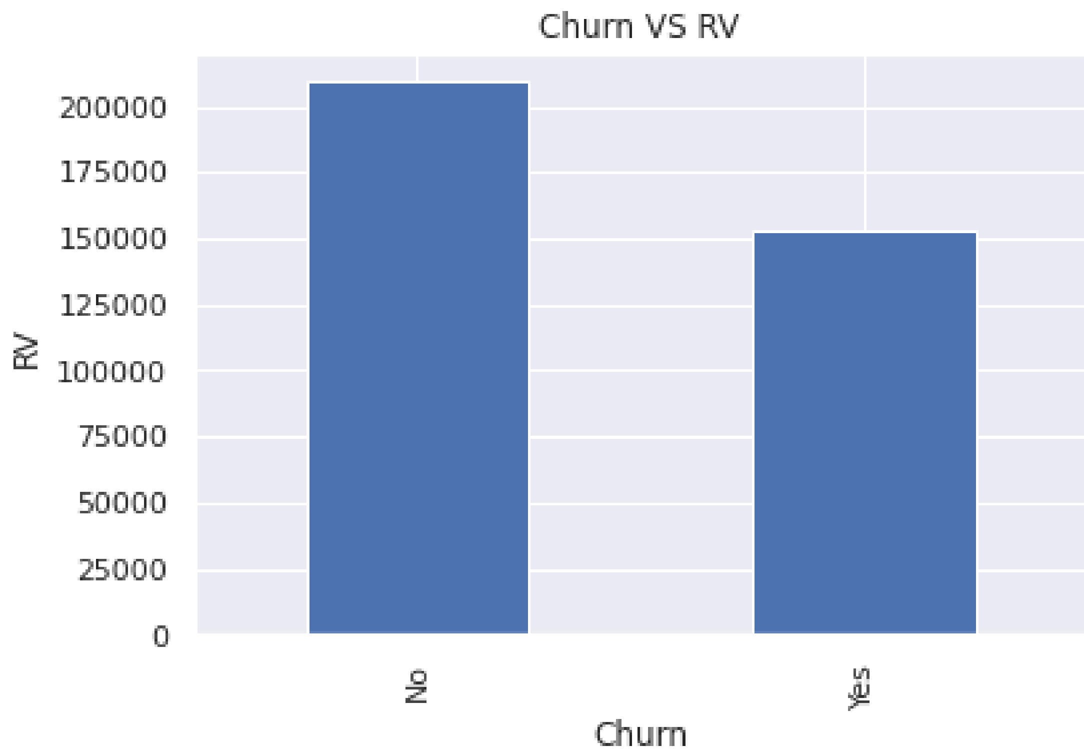
- *`Male` customer has churned more than `female`*
- *Customer residing in `Urban` Area has churned more than `Rural` or `Semi-urban`. This could be because the Urban customer may have more banks to open accounts.*
- *`Young-Adult` and `Adult` has more tendency to churn compared to Other categories. This is because they are mostly salary holders and many other banks approach them via mail/sms/calls.*
- *Customer who has FD, Credit Card or Loan has less tends to churn. Bank should try to sell them these product to prevent Churn.*

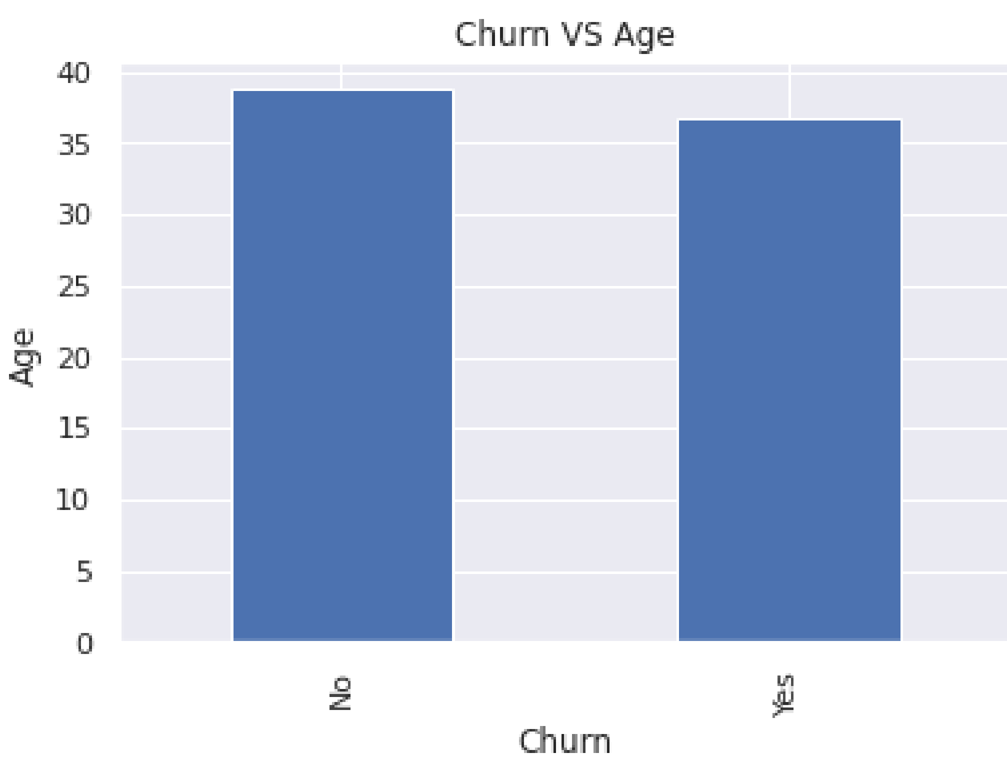
Numerical features

[55]:
```
for i in df_new2:
    if i not in categorical_features and i != 'Churn':
        df_new2.groupby('Churn')[i].mean().plot.bar()
        plt.xlabel('Churn')
        plt.ylabel(i)
        plt.title('Churn VS {}'.format(i))
        plt.show()
```


Churn VS AccountBalance
AccountBalance
120000
100000
80000
60000
40000
20000
0
No
Yes
Churn


Churn VS CIBIL_Score
CIBIL_Score
600
500
400
300
200
100
0
No
Yes
Churn


Churn VS RV
RV
200000
175000
150000
125000
100000
75000
50000
25000
0
No
Yes
Churn

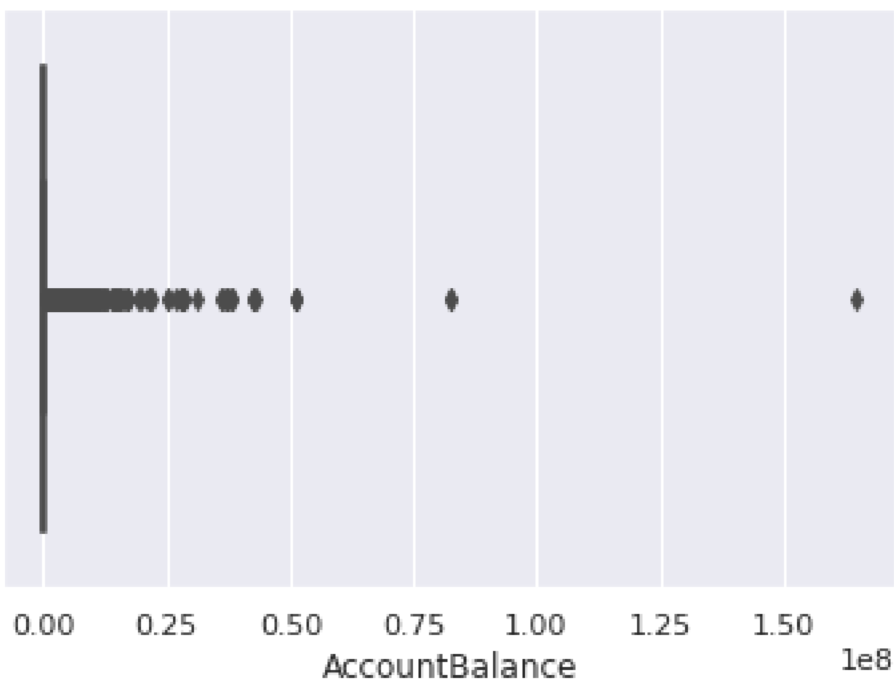

Churn VS Age
Age
40
35
30
25
20
15
10
5
0
No
Yes
Churn

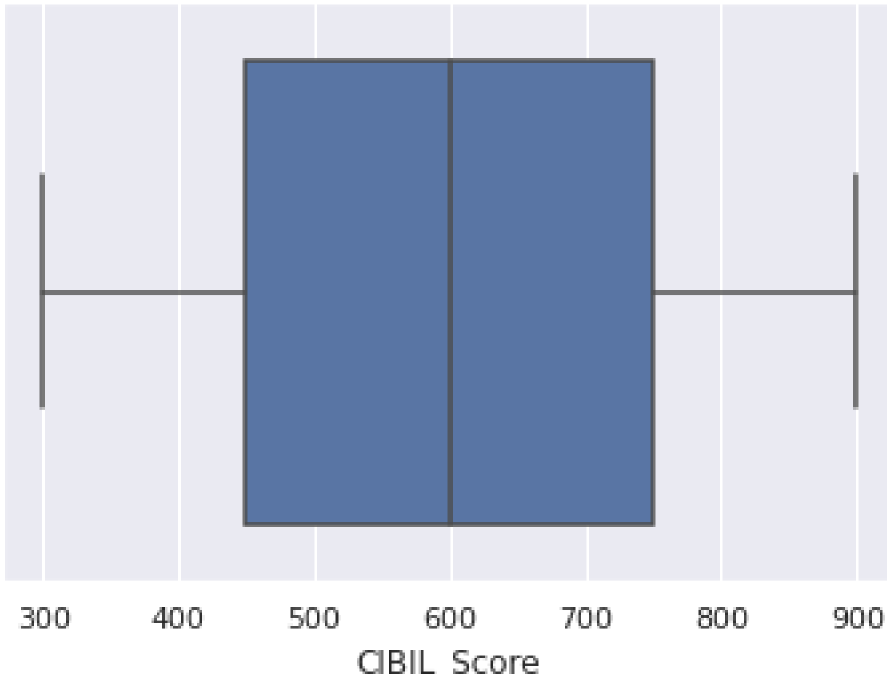
Observations:-

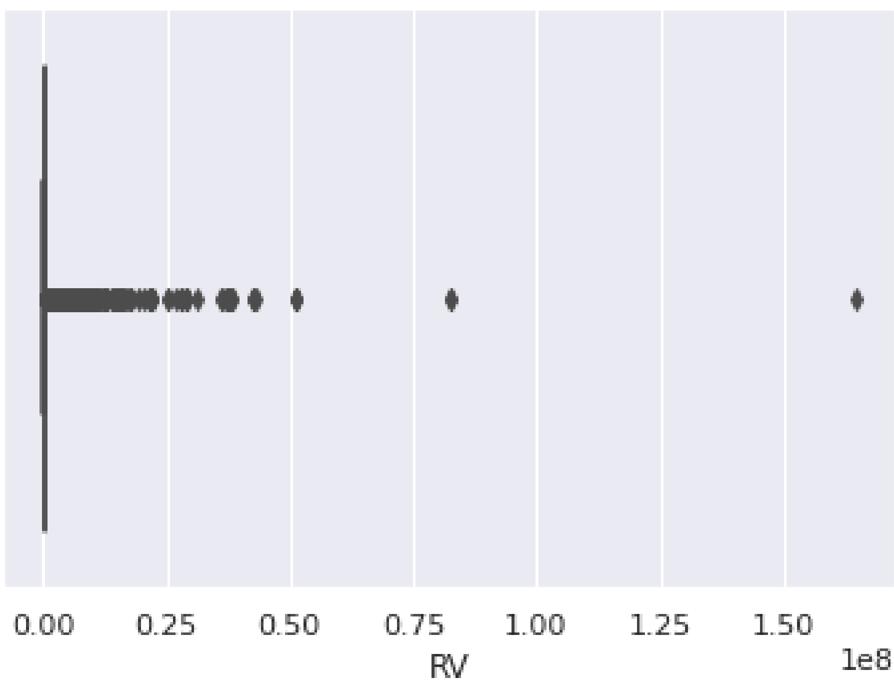
- *Customer who has churned had a very low average account balance/RV.*
- *There is no significant impact of CIBIL Score in Churning.*
- *Average age of the customer who churned is around 32-33 years(As shown in the category-Adult and Young-Adult)*

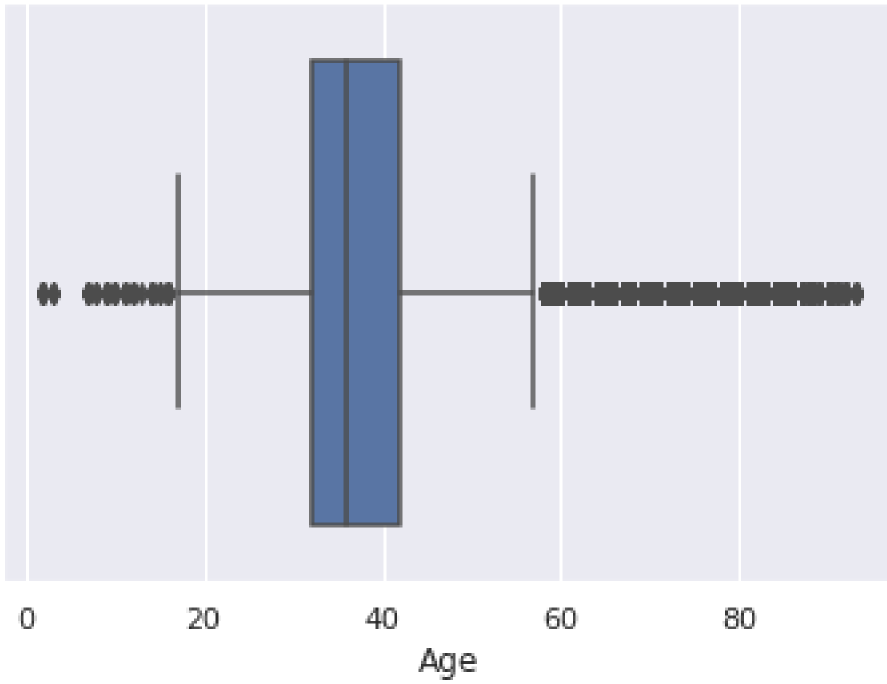
4. Outliers Check.

[56]:
```
#df_new2 = df_new.copy()
for i in df_new2.columns:
    if i not in categorical_features and i != 'Churn':
        sns.boxplot(df_new2[i])
        plt.xlabel(i)
        plt.show()
```




300
400
500
600
700
800
900
CIBIL_Score


0.00
0.25
0.50
0.75
1.00
1.25
1.50
1e8
RV



Observations:-

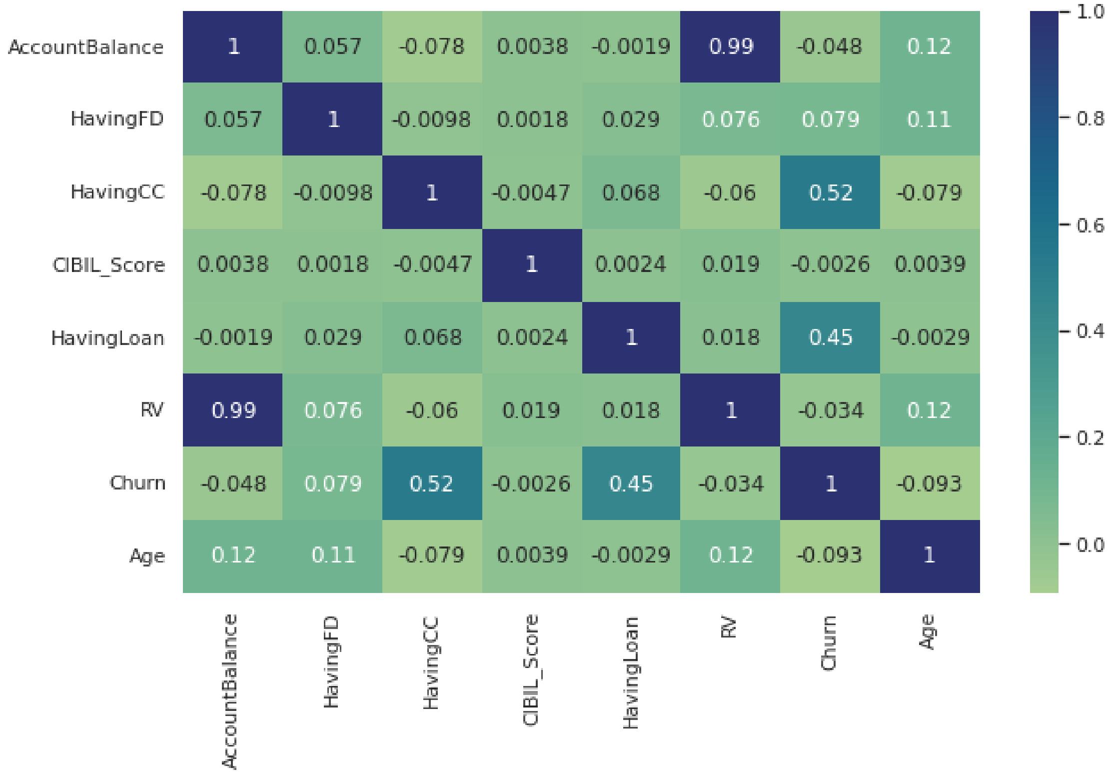
- *Account Balance and RV(Relationship Value) has many outliers. This is because many customers have a very high account balances.*
- *CIBIL Score has no outliers because it falls between 300-900.*
- *Age has slight outliers because of Senior Citizen Customers and few Minor Customers.*

Note:- Despite so many outliers I will not remove these outliers because after removing it will not be a true representative of the actual data. Since we are dealing with bank data, an imbalance dataset is a common phenomenon.

5. Correlation.

[57]:
```
plt.figure(figsize=(10,6))
sns.heatmap(df_new.corr(), annot= True,cmap="crest")
```

[57]: <AxesSubplot:>



[38]:
```python
#Checking absolute Correlation
df_new.corr()['Churn'].abs().sort_values(ascending = False)
```

[38]:
```
Churn            1.000
HavingCC         0.523
HavingLoan       0.449
Age              0.093
HavingFD         0.079
AccountBalance   0.048
RV               0.034
CIBIL_Score      0.003
Name: Churn, dtype: float64
```

Observations:-

- *Credit Card and Loan Customer has a positive Correlation. This means if someone has a Credit card or Loan has tend to churn more.*
- *Account balance and Age have very slight correlations.*

0.0.7 *Model Building*

Now turn to building an efficient and accurate(as much as possible) model which will help to predict if a customer is going to churn or not.

There are several steps during model building:-

1. *Creating Dummy variable(for categorical features) & Definig X and y variable.*
2. *Train and Test splitting.*
3. *Selecting a proper model(Out of few model tested).*
4. *Getting proper model parameters using Cross Validation Technique.*
5. *Measuring the Accuracy.*
6. *Deploying the model into production*

1. Creating Dummy variable & Definig X and y variable

[39]:
```
df2 = pd.get_dummies(df_new, columns= categorical_features, drop_first= True)
```

[40]:
```
X = df2.drop('Churn', axis = 1)
y = df2['Churn']
```

2. Train and Test splitting

Since we have highly imbalance data, I will use stratified spliting to get actual ratio of Churn.

[41]:
```
from sklearn.model_selection import train_test_split

x_train,x_test,y_train,y_test = train_test_split(X,y, test_size= 0.3,␣
↪random_state= 100, stratify= y)
```

3. Train and Test splitting

I will build ANN Model using Tensorflow

[42]:
```
#ANN architecture
import tensorflow
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

[60]:
```
model = Sequential()
model.add(Dense(11,activation='relu',input_dim=13))
model.add(Dense(7,activation='relu'))
model.add(Dense(1,activation='sigmoid')) ##For Binary activation function =␣
↪sigmoid
```

[61]:
```
model.summary()
```

```
Model: "sequential_2"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 11)                154

_________________________________________________________________
dense_7 (Dense)              (None, 7)                 84
```

```
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 8
=================================================================
Total params: 246
Trainable params: 246
Non-trainable params: 0
_________________________________________________________________
```

[62]:
```python
model.compile(optimizer='Adam',loss='binary_crossentropy',metrics=['accuracy'])
```

[82]:
```python
history = model.
 ↪fit(x_train,y_train,batch_size=50,epochs=50,verbose=1,validation_split=0.2)
```

```
Epoch 1/50
3189/3189 [==============================] - 6s 2ms/step - loss: 14.7528 -
accuracy: 0.7128 - val_loss: 29.0681 - val_accuracy: 0.7831
Epoch 2/50
3189/3189 [==============================] - 6s 2ms/step - loss: 11.2415 -
accuracy: 0.7150 - val_loss: 22.5441 - val_accuracy: 0.7830
Epoch 3/50
3189/3189 [==============================] - 6s 2ms/step - loss: 8.3382 -
accuracy: 0.7119 - val_loss: 5.5955 - val_accuracy: 0.5365
Epoch 4/50
3189/3189 [==============================] - 7s 2ms/step - loss: 4.1849 -
accuracy: 0.7099 - val_loss: 2.2093 - val_accuracy: 0.7626
Epoch 5/50
3189/3189 [==============================] - 7s 2ms/step - loss: 3.4722 -
accuracy: 0.7122 - val_loss: 0.7640 - val_accuracy: 0.7849
Epoch 6/50
3189/3189 [==============================] - 7s 2ms/step - loss: 1.2846 -
accuracy: 0.7216 - val_loss: 0.8388 - val_accuracy: 0.7829
Epoch 7/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.6844 -
accuracy: 0.7441 - val_loss: 0.4376 - val_accuracy: 0.7907
Epoch 8/50
3189/3189 [==============================] - 6s 2ms/step - loss: 0.5609 -
accuracy: 0.7739 - val_loss: 0.4933 - val_accuracy: 0.7828
Epoch 9/50
3189/3189 [==============================] - 6s 2ms/step - loss: 0.4963 -
accuracy: 0.7804 - val_loss: 0.4924 - val_accuracy: 0.7828
Epoch 10/50
3189/3189 [==============================] - 6s 2ms/step - loss: 0.4960 -
accuracy: 0.7804 - val_loss: 0.4915 - val_accuracy: 0.7828
Epoch 11/50
3189/3189 [==============================] - 6s 2ms/step - loss: 0.4947 -
accuracy: 0.7804 - val_loss: 0.4901 - val_accuracy: 0.7827
Epoch 12/50
3189/3189 [==============================] - 6s 2ms/step - loss: 0.5309 -
```

```
accuracy: 0.7798 - val_loss: 0.4939 - val_accuracy: 0.7828
Epoch 13/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.5063 -
accuracy: 0.7798 - val_loss: 0.4912 - val_accuracy: 0.7828
Epoch 14/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.5133 -
accuracy: 0.7801 - val_loss: 0.5118 - val_accuracy: 0.7828
Epoch 15/50
3189/3189 [==============================] - 8s 2ms/step - loss: 0.5147 -
accuracy: 0.7804 - val_loss: 0.5114 - val_accuracy: 0.7828
Epoch 16/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.5021 -
accuracy: 0.7804 - val_loss: 0.4911 - val_accuracy: 0.7828
Epoch 17/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4938 -
accuracy: 0.7804 - val_loss: 0.4904 - val_accuracy: 0.7828
Epoch 18/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4946 -
accuracy: 0.7804 - val_loss: 0.4912 - val_accuracy: 0.7828
Epoch 19/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4944 -
accuracy: 0.7804 - val_loss: 0.4903 - val_accuracy: 0.7828
Epoch 20/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4939 -
accuracy: 0.7804 - val_loss: 0.4899 - val_accuracy: 0.7828
Epoch 21/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4958 -
accuracy: 0.7803 - val_loss: 0.4908 - val_accuracy: 0.7828
Epoch 22/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4935 -
accuracy: 0.7804 - val_loss: 0.4937 - val_accuracy: 0.7828
Epoch 23/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4988 -
accuracy: 0.7804 - val_loss: 0.4941 - val_accuracy: 0.7828
Epoch 24/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4933 -
accuracy: 0.7804 - val_loss: 0.4907 - val_accuracy: 0.7828
Epoch 25/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.5017 -
accuracy: 0.7804 - val_loss: 0.4933 - val_accuracy: 0.7828
Epoch 26/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4938 -
accuracy: 0.7804 - val_loss: 0.4897 - val_accuracy: 0.7828
Epoch 27/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4975 -
accuracy: 0.7803 - val_loss: 0.4901 - val_accuracy: 0.7828
Epoch 28/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4930 -
```

```
accuracy: 0.7804 - val_loss: 0.4902 - val_accuracy: 0.7828
Epoch 29/50
3189/3189 [==============================] - 8s 2ms/step - loss: 0.4931 -
accuracy: 0.7804 - val_loss: 0.4901 - val_accuracy: 0.7828
Epoch 30/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4941 -
accuracy: 0.7804 - val_loss: 0.4908 - val_accuracy: 0.7828
Epoch 31/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.5031 -
accuracy: 0.7802 - val_loss: 0.4887 - val_accuracy: 0.7828
Epoch 32/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.5029 -
accuracy: 0.7803 - val_loss: 0.4904 - val_accuracy: 0.7828
Epoch 33/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4935 -
accuracy: 0.7804 - val_loss: 0.4900 - val_accuracy: 0.7828
Epoch 34/50
3189/3189 [==============================] - 8s 2ms/step - loss: 0.4932 -
accuracy: 0.7804 - val_loss: 0.4923 - val_accuracy: 0.7828
Epoch 35/50
3189/3189 [==============================] - 8s 2ms/step - loss: 0.5011 -
accuracy: 0.7803 - val_loss: 0.4907 - val_accuracy: 0.7828
Epoch 36/50
3189/3189 [==============================] - 8s 2ms/step - loss: 0.4939 -
accuracy: 0.7804 - val_loss: 0.4900 - val_accuracy: 0.7828
Epoch 37/50
3189/3189 [==============================] - 7s 2ms/step - loss: 0.4957 -
accuracy: 0.7804 - val_loss: 0.4966 - val_accuracy: 0.7826
Epoch 38/50
3189/3189 [==============================] - 8s 2ms/step - loss: 0.4958 -
accuracy: 0.7804 - val_loss: 0.4902 - val_accuracy: 0.7828
Epoch 39/50
3189/3189 [==============================] - 8s 3ms/step - loss: 0.4934 -
accuracy: 0.7804 - val_loss: 0.4902 - val_accuracy: 0.7828
Epoch 40/50
3189/3189 [==============================] - 9s 3ms/step - loss: 0.4938 -
accuracy: 0.7804 - val_loss: 0.4887 - val_accuracy: 0.7828
Epoch 41/50
3189/3189 [==============================] - 9s 3ms/step - loss: 0.5019 -
accuracy: 0.7802 - val_loss: 0.4895 - val_accuracy: 0.7828
Epoch 42/50
3189/3189 [==============================] - 9s 3ms/step - loss: 0.4921 -
accuracy: 0.7804 - val_loss: 0.4890 - val_accuracy: 0.7828
Epoch 43/50
3189/3189 [==============================] - 9s 3ms/step - loss: 0.4938 -
accuracy: 0.7804 - val_loss: 0.4885 - val_accuracy: 0.7828
Epoch 44/50
3189/3189 [==============================] - 9s 3ms/step - loss: 0.4906 -
```

```
accuracy: 0.7804 - val_loss: 0.4874 - val_accuracy: 0.7828
Epoch 45/50
3189/3189 [==============================] - 9s 3ms/step - loss: 0.4916 -
accuracy: 0.7804 - val_loss: 0.4888 - val_accuracy: 0.7828
Epoch 46/50
3189/3189 [==============================] - 9s 3ms/step - loss: 0.4920 -
accuracy: 0.7804 - val_loss: 0.4868 - val_accuracy: 0.7828
Epoch 47/50
3189/3189 [==============================] - 10s 3ms/step - loss: 0.4916 -
accuracy: 0.7804 - val_loss: 0.4881 - val_accuracy: 0.7828
Epoch 48/50
3189/3189 [==============================] - 10s 3ms/step - loss: 0.4899 -
accuracy: 0.7804 - val_loss: 0.4918 - val_accuracy: 0.7828
Epoch 49/50
3189/3189 [==============================] - 10s 3ms/step - loss: 0.4901 -
accuracy: 0.7804 - val_loss: 0.4876 - val_accuracy: 0.7828
Epoch 50/50
3189/3189 [==============================] - 10s 3ms/step - loss: 0.5070 -
accuracy: 0.7804 - val_loss: 0.5063 - val_accuracy: 0.7828
```

[83]:
```
y_pred = model.predict(x_test)
```

[84]:
```
y_pred_f = np.where(y_pred < 0.5 , 0,1)
```

5. Accuracy check with F1 score.

[85]:
```
from sklearn.metrics import f1_score,accuracy_score
```

[86]:
```
f1_score(y_test,y_pred_f)
```

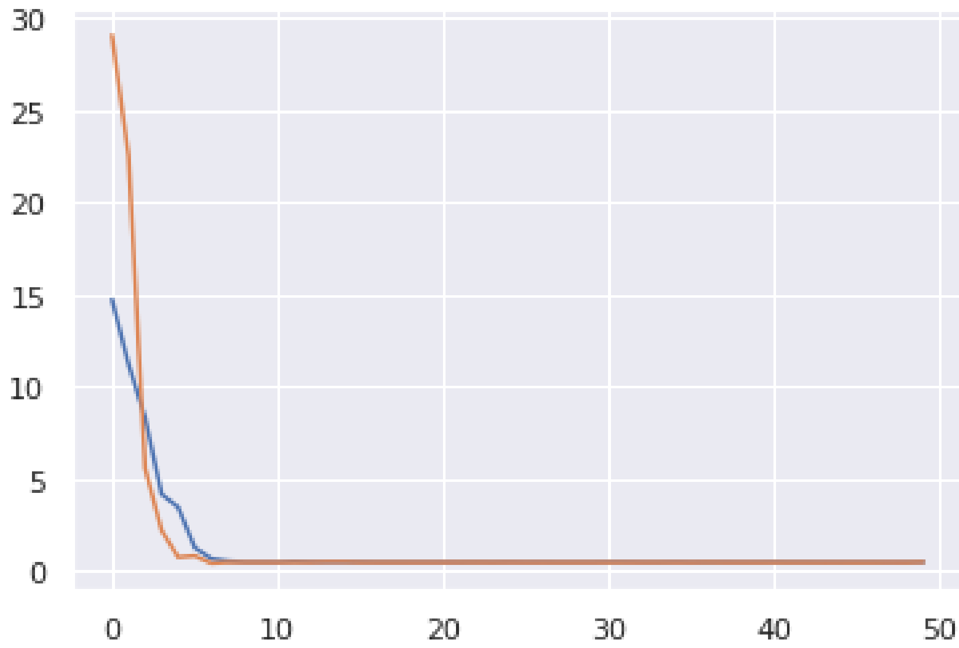
[86]:
```
0.0
```

[87]:
```
accuracy_score(y_test,y_pred_f)
```

[87]:
```
0.7808998630185101
```

[88]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
```

[88]:
```
[<matplotlib.lines.Line2D at 0x7ffa2ffdf7d0>]
```



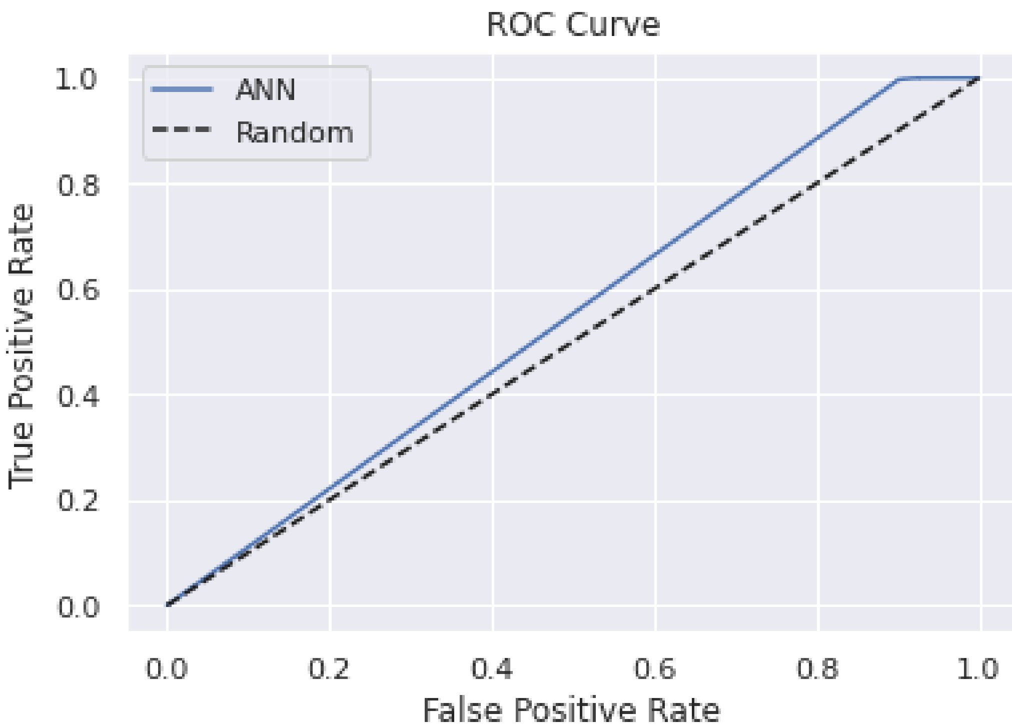
Ploting AUC-ROC curve:-

[89]:
```
from sklearn.metrics import roc_curve
```

[90]:
```
y_score = model.predict(x_test)
```

[91]:
```
fpr, tpr, thresholds = roc_curve(y_test, y_score)
```

[92]:
```
# Plot the ROC curve
plt.plot(fpr, tpr, 'b-', label='ANN')
plt.plot([0, 1], [0, 1], 'k--', label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='best')
plt.show()
```



Now, this model can be deployed in any local machine or cloud server as a full production grade application This can be integrated into any web application that will help bank staff to check if a customer is going to churn and take preventive measures to stop it..

0.0.8 Copyright © Gopi Nath Pandit