```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder, OneHotEncoder,
StandardScaler, normalize
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
import tensorflow as tf

from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

```python
path =
"/content/drive/MyDrive/customer_churn/dataset/customer_churn_large_da
taset.xlsx"

df = pd.read_excel(path)

# Converting to Pandas DataFrame
df = pd.DataFrame(df)
df.head()
```

```
   CustomerID        Name  Age  Gender     Location  \
0           1  Customer_1   63    Male  Los Angeles
1           2  Customer_2   62  Female     New York
2           3  Customer_3   24  Female  Los Angeles
3           4  Customer_4   36  Female        Miami
4           5  Customer_5   46  Female        Miami

   Subscription_Length_Months  Monthly_Bill  Total_Usage_GB  Churn
0                          17         73.36             236      0
1                           1         48.76             172      0
2                           5         85.47             460      0
3                           3         97.94             297      1
4                          19         58.14             266      0
```

```python
df.shape
```

```
(100000, 9)
```

```python
df.dtypes
```

```
CustomerID                      int64
Name                           object
Age                             int64
Gender                         object
Location                       object
Subscription_Length_Months      int64
Monthly_Bill                  float64
Total_Usage_GB                  int64
Churn                           int64
dtype: object
```

```
df.describe(include='all')
```

```
           CustomerID         Name            Age  Gender Location  \
count   100000.000000       100000  100000.000000  100000   100000
unique            NaN       100000            NaN       2        5
top               NaN   Customer_1            NaN  Female  Houston
freq              NaN            1            NaN   50216    20157
mean     50000.500000          NaN      44.027020     NaN      NaN
std      28867.657797          NaN      15.280283     NaN      NaN
min          1.000000          NaN      18.000000     NaN      NaN
25%      25000.750000          NaN      31.000000     NaN      NaN
50%      50000.500000          NaN      44.000000     NaN      NaN
75%      75000.250000          NaN      57.000000     NaN      NaN
max     100000.000000          NaN      70.000000     NaN      NaN

        Subscription_Length_Months   Monthly_Bill  Total_Usage_GB  \
count                100000.000000  100000.000000   100000.000000
unique                         NaN            NaN             NaN
top                            NaN            NaN             NaN
freq                           NaN            NaN             NaN
mean                     12.490100      65.053197      274.393650
std                       6.926461      20.230696      130.463063
min                       1.000000      30.000000       50.000000
25%                       6.000000      47.540000      161.000000
50%                      12.000000      65.010000      274.000000
75%                      19.000000      82.640000      387.000000
max                      24.000000     100.000000      500.000000

                Churn
count   100000.000000
unique            NaN
top               NaN
freq              NaN
mean         0.497790
std          0.499998
min          0.000000
25%          0.000000
50%          0.000000
```

```
75%          1.000000
max          1.000000
```

Count for each column is 100,000 which is the total number of data points

Location

- Number of locations = 5
- Most common occurring location is **Houston** with 20157 data points

Gender

- Majority of the customers are **Female**
- Total number of females = 50216

Age

- Mininum age = 18
- Average age = 44
- Maximum Age = 70

Subscription Duration

- Minimum = 1 month
- Average = 1 year
- Maximum = 2 years

=> There are no subscriptions that are for more than 2 years

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
 #   Column                      Non-Null Count   Dtype  
---  ------                      --------------   -----  
 0   CustomerID                  100000 non-null  int64  
 1   Name                        100000 non-null  object 
 2   Age                         100000 non-null  int64  
 3   Gender                      100000 non-null  object 
 4   Location                    100000 non-null  object 
 5   Subscription_Length_Months  100000 non-null  int64  
 6   Monthly_Bill                100000 non-null  float64
 7   Total_Usage_GB              100000 non-null  int64  
 8   Churn                       100000 non-null  int64  
dtypes: float64(1), int64(5), object(3)
memory usage: 6.9+ MB
```

=> There are no null values present in the dataset

##This can easily be confirmed using the isnull() method as well as visualized using a bar plot

```
df.isnull().any()
```

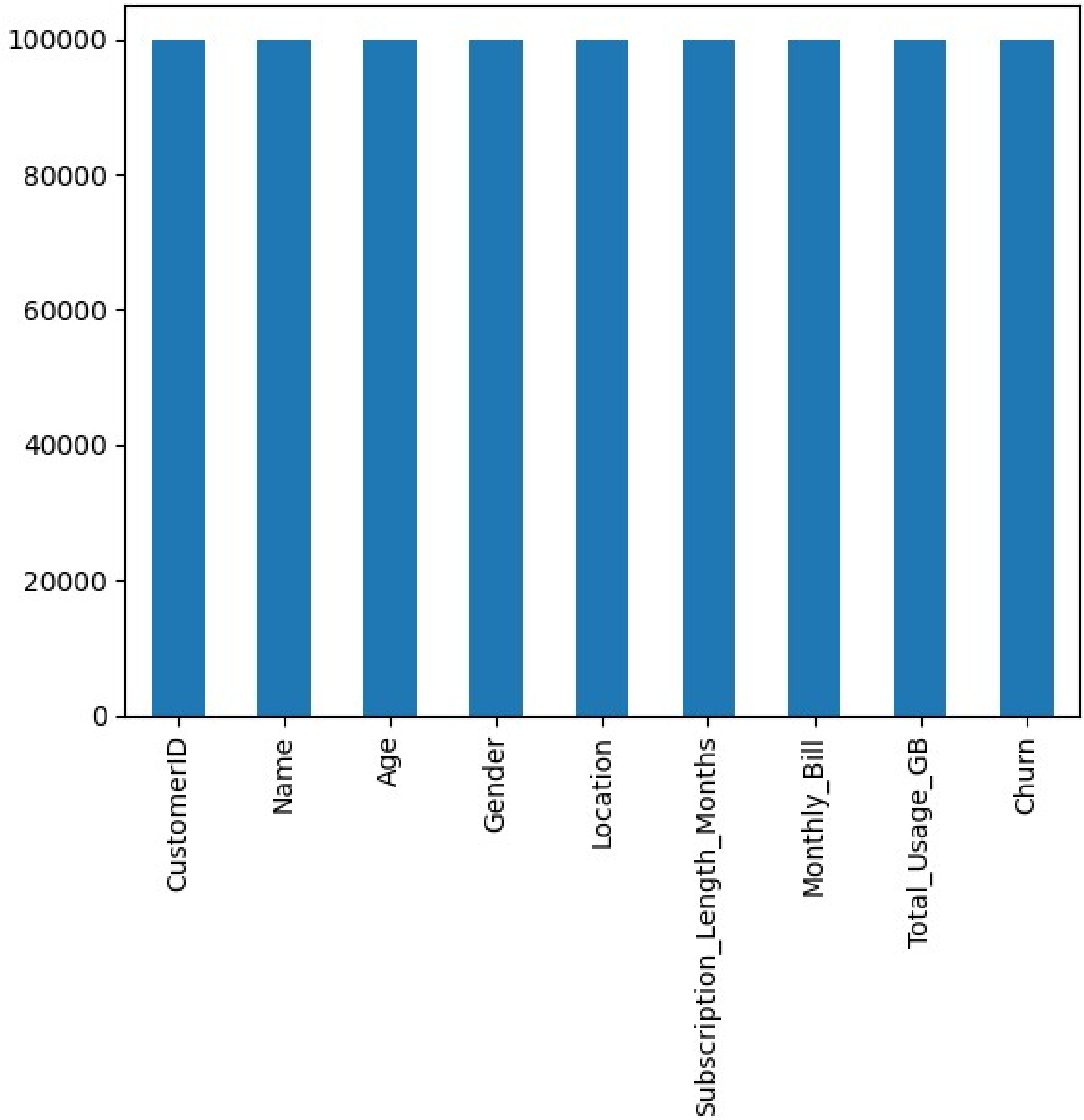
```
CustomerID                    False
Name                          False
Age                           False
Gender                        False
Location                      False
Subscription_Length_Months    False
Monthly_Bill                  False
Total_Usage_GB                False
Churn                         False
dtype: bool
```

=> This indicates that none of the features contains any null values.

Visualization for the same can be seen below:

```
df.count().plot.bar()
```

```
<Axes: >
```

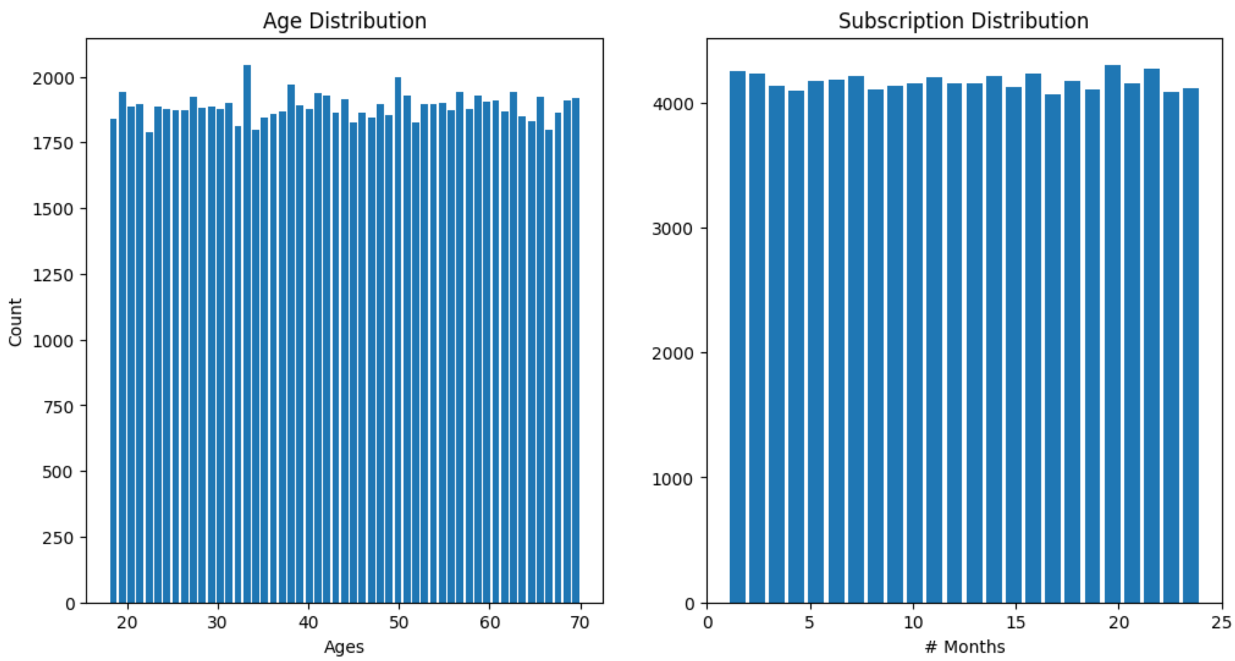


Outlier Detection

Visualization of outliers using Distribution

```
# Creating a 1x2 grid of subplots
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Histogram of Age
axs[0].hist(df.Age, bins=(df.Age.max()-df.Age.min())+1, rwidth=0.8)
axs[0].set_title("Age Distribution")

# Histogram of subscription months
axs[1].hist(df.Subscription_Length_Months,
```

```
bins=(df.Subscription_Length_Months.max()-
df.Subscription_Length_Months.min())+1,rwidth=0.8)
axs[1].set_title("Subscription Distribution")

axs[0].set_ylabel('Count')
axs[0].set_xlabel('Ages')
axs[1].set_xlabel('# Months')
plt.show()
```



Age Distribution

Age histogram exhibits uniform distribution along with the repeated peaks. Same pattern is followed throughout which rules out any possibility of outliers

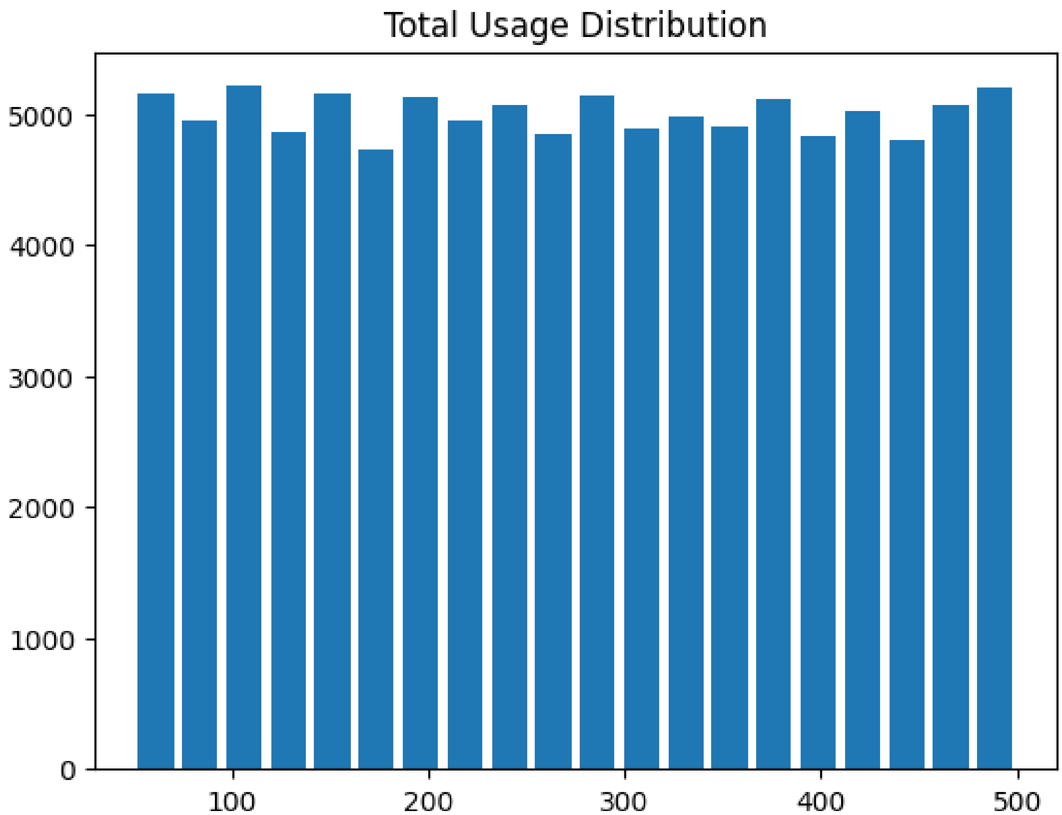
Subscription Distribution

- The even distribution of subscriptions suggests that people are subscribing uniformly across various time intervals.

- It that there are no particular periods that attract significantly more or fewer subscriptions than others

```
# Histogram of total usage
plt.hist(df.Total_Usage_GB, bins=20, rwidth=0.8)
```

```
plt.title("Total Usage Distribution")
plt.show()
```



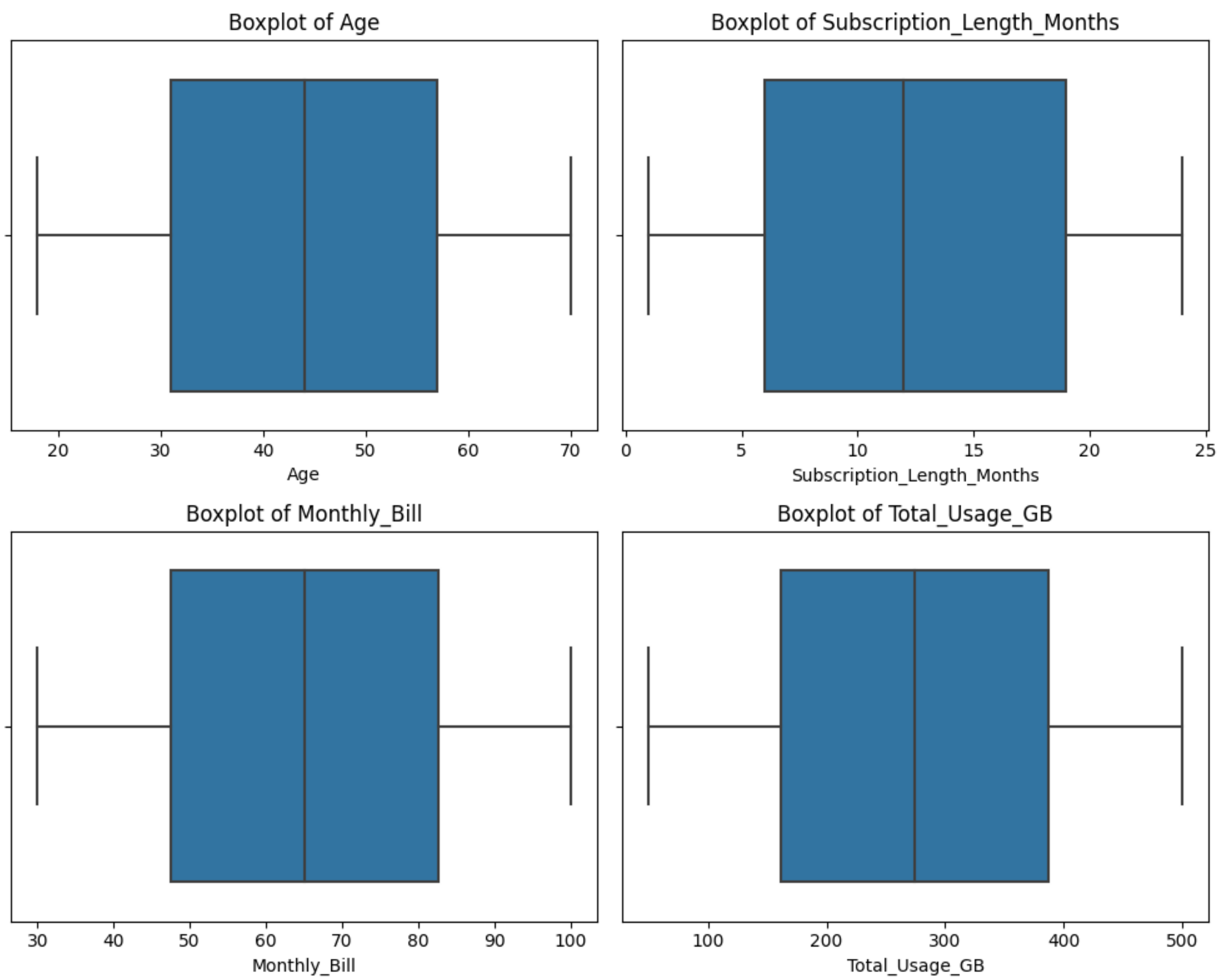
- There is a uniform distribution of usage across various data packages which indicates that each package is utilized by similar proportion of the total population

- This could imply that there is no strong preference or bias towards any particular package

- It is safe to say that there are no outliers

```
import seaborn as sns

# columns to analyze outliers
numerical_columns = ['Age', 'Subscription_Length_Months',
'Monthly_Bill', 'Total_Usage_GB']

# Plotting boxplots for the numerical columns
plt.figure(figsize=(10, 8))
for idx, col in enumerate(numerical_columns, 1):
    plt.subplot(2, 2, idx)
```

```
    sns.boxplot(x=df[col])
    plt.title(f'Boxplot of {col}')

plt.tight_layout()
plt.show()
```



Outlier Detection using Z-Score

Considering threshold value to be 3. If the z-score > 3, this implies existence of outliers

```
# Age
z_score_age = (df.Age - df.Age.mean())/df.Age.std()
z_score_age

0         1.241664
1         1.176220
```

```
2       -1.310645
3       -0.525319
4        0.129119
           ...
99995   -0.721650
99996    1.176220
99997    1.307108
99998    0.456338
99999   -1.114313
Name: Age, Length: 100000, dtype: float64
```

```
df.Age[z_score_age >=3].count()
```

```
0
```

```
# Monthly Bill
z_score_bill = (df.Monthly_Bill - 
df.Monthly_Bill.mean())/df.Monthly_Bill.std()
z_score_bill
```

```
0        0.410604
1       -0.805370
2        1.009199
3        1.625589
4       -0.341718
           ...
99995   -0.490502
99996   -0.168219
99997    1.535133
99998   -0.781149
99999    0.569274
Name: Monthly_Bill, Length: 100000, dtype: float64
```

```
df.Monthly_Bill[z_score_bill >=3].count()
```

```
0
```

```
# Total_Usage_GB
# Monthly Bill
z_score_usage = (df.Total_Usage_GB - 
df.Total_Usage_GB.mean())/df.Total_Usage_GB.std()
z_score_usage
```

```
0       -0.294288
1       -0.784848
2        1.422674
3        0.173278
4       -0.064337
           ...
99995   -0.370938
99996    0.587188
```

```
99997   -0.179312
99998    1.223383
99999   -0.777183
Name: Total_Usage_GB, Length: 100000, dtype: float64
```

```
df.Total_Usage_GB[z_score_bill >=3].count()
```

```
0
```

The Z-score test implies that there are no existence of outliers in the features - Age, Monthly Bill and Total Usage

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
 #   Column                      Non-Null Count   Dtype
---  ------                      --------------   -----
 0   CustomerID                  100000 non-null  int64
 1   Name                        100000 non-null  object
 2   Age                         100000 non-null  int64
 3   Gender                      100000 non-null  object
 4   Location                    100000 non-null  object
 5   Subscription_Length_Months  100000 non-null  int64
 6   Monthly_Bill                100000 non-null  float64
 7   Total_Usage_GB              100000 non-null  int64
 8   Churn                       100000 non-null  int64
dtypes: float64(1), int64(5), object(3)
memory usage: 6.9+ MB
```

Encoding Categorical Data

```
df.head()
```

```
   CustomerID        Name  Age  Gender     Location  \
0           1  Customer_1   63    Male  Los Angeles
1           2  Customer_2   62  Female     New York
2           3  Customer_3   24  Female  Los Angeles
3           4  Customer_4   36  Female        Miami
4           5  Customer_5   46  Female        Miami

   Subscription_Length_Months  Monthly_Bill  Total_Usage_GB  Churn
0                          17         73.36             236      0
1                           1         48.76             172      0
2                           5         85.47             460      0
3                           3         97.94             297      1
4                          19         58.14             266      0
```

There are two categorial features that need to be encoded, namely, Gender and Location

Extracting unique values from each feature

```
df['Gender'].unique()
```

```
array(['Male', 'Female'], dtype=object)
```

=> Two unique values:

- Male
- Female

```
df['Location'].unique()
```

```
array(['Los Angeles', 'New York', 'Miami', 'Chicago', 'Houston'],
      dtype=object)
```

=> Five unique values:

- Los Angeles
- New York
- Miami
- Chicago
- Houston

```
df
```

```
       CustomerID             Name  Age  Gender     Location  \
0               1       Customer_1   63    Male  Los Angeles
1               2       Customer_2   62  Female     New York
2               3       Customer_3   24  Female  Los Angeles
3               4       Customer_4   36  Female        Miami
4               5       Customer_5   46  Female        Miami
...           ...              ...  ...     ...          ...
99995       99996   Customer_99996   33    Male      Houston
99996       99997   Customer_99997   62  Female     New York
99997       99998   Customer_99998   64    Male      Chicago
99998       99999   Customer_99999   51  Female     New York
99999      100000  Customer_100000   27  Female  Los Angeles

       Subscription_Length_Months  Monthly_Bill  Total_Usage_GB  Churn

0                              17         73.36             236      0

1                               1         48.76             172      0

2                               5         85.47             460      0

3                               3         97.94             297      1
```

```
4                               19         58.14             266      0
...                            ...           ...             ...    ...
99995                           23         55.13             226      1
99996                           19         61.65             351      0
99997                           17         96.11             251      1
99998                           20         49.25             434      1
99999                           19         76.57             173      1

[100000 rows x 9 columns]
```

Applying Label-Encoding on Gender Feature

using sklearn

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
 #   Column                      Non-Null Count   Dtype
---  ------                      --------------   -----
 0   CustomerID                  100000 non-null  int64
 1   Name                        100000 non-null  object
 2   Age                         100000 non-null  int64
 3   Gender                      100000 non-null  object
 4   Location                    100000 non-null  object
 5   Subscription_Length_Months  100000 non-null  int64
 6   Monthly_Bill                100000 non-null  float64
 7   Total_Usage_GB              100000 non-null  int64
 8   Churn                       100000 non-null  int64
dtypes: float64(1), int64(5), object(3)
memory usage: 6.9+ MB

# Label Encoding
le = LabelEncoder()
df['Gender'] = le.fit_transform(df['Gender'])

df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
 #   Column                      Non-Null Count   Dtype
```

```
---  ------                     --------------   -----
 0   CustomerID                 100000 non-null  int64
 1   Name                       100000 non-null  object
 2   Age                        100000 non-null  int64
 3   Gender                     100000 non-null  int64
 4   Location                   100000 non-null  object
 5   Subscription_Length_Months  100000 non-null  int64
 6   Monthly_Bill               100000 non-null  float64
 7   Total_Usage_GB             100000 non-null  int64
 8   Churn                      100000 non-null  int64
dtypes: float64(1), int64(6), object(2)
memory usage: 6.9+ MB
```

Applying One Hot Encoding on Locations Feature

```python
# Saving column names
dummy_cols = pd.get_dummies(df['Location']).columns
dummy_cols = list(dummy_cols)
print(dummy_cols)
cols =  dummy_cols + ['CustomerId','Name','Age','Gender'] +
['Subscription_Length_Months','Monthly_Bill','Total_Usage_GB','Churn']
cols
```

```
['Chicago', 'Houston', 'Los Angeles', 'Miami', 'New York']
```

```
['Chicago',
 'Houston',
 'Los Angeles',
 'Miami',
 'New York',
 'CustomerId',
 'Name',
 'Age',
 'Gender',
 'Subscription_Length_Months',
 'Monthly_Bill',
 'Total_Usage_GB',
 'Churn']
```

```python
df
```

```
       CustomerID              Name  Age  Gender     Location  \
0               1        Customer_1   63       1  Los Angeles
1               2        Customer_2   62       0     New York
2               3        Customer_3   24       0  Los Angeles
3               4        Customer_4   36       0        Miami
4               5        Customer_5   46       0        Miami
...           ...               ...  ...     ...          ...
99995       99996    Customer_99996   33       1      Houston
99996       99997    Customer_99997   62       0     New York
```

```
99997       99998   Customer_99998   64       1      Chicago
99998       99999   Customer_99999   51       0     New York
99999      100000  Customer_100000   27       0  Los Angeles

       Subscription_Length_Months  Monthly_Bill  Total_Usage_GB  Churn

0                              17         73.36             236      0

1                               1         48.76             172      0

2                               5         85.47             460      0

3                               3         97.94             297      1

4                              19         58.14             266      0

...                           ...           ...             ...    ...

99995                          23         55.13             226      1

99996                          19         61.65             351      0

99997                          17         96.11             251      1

99998                          20         49.25             434      1

99999                          19         76.57             173      1


[100000 rows x 9 columns]
```

```
ct = ColumnTransformer(transformers = [('encoder', OneHotEncoder(),
[4])], remainder='passthrough')
df = pd.DataFrame(ct.fit_transform(df))

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 13 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   0       100000 non-null  object
 1   1       100000 non-null  object
 2   2       100000 non-null  object
 3   3       100000 non-null  object
 4   4       100000 non-null  object
 5   5       100000 non-null  object
 6   6       100000 non-null  object
 7   7       100000 non-null  object
 8   8       100000 non-null  object
```

```
 9   9       100000 non-null  object
 10  10      100000 non-null  object
 11  11      100000 non-null  object
 12  12      100000 non-null  object
dtypes: object(13)
memory usage: 9.9+ MB
```

```
df.columns = cols
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 13 columns):
 #   Column                      Non-Null Count   Dtype 
---  ------                      --------------   ----- 
 0   Chicago                     100000 non-null  object
 1   Houston                     100000 non-null  object
 2   Los Angeles                 100000 non-null  object
 3   Miami                       100000 non-null  object
 4   New York                    100000 non-null  object
 5   CustomerId                  100000 non-null  object
 6   Name                        100000 non-null  object
 7   Age                         100000 non-null  object
 8   Gender                      100000 non-null  object
 9   Subscription_Length_Months  100000 non-null  object
 10  Monthly_Bill                100000 non-null  object
 11  Total_Usage_GB              100000 non-null  object
 12  Churn                       100000 non-null  object
dtypes: object(13)
memory usage: 9.9+ MB
```

```
df[cols] = df[cols].apply(pd.to_numeric, errors='coerce')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 13 columns):
 #   Column                      Non-Null Count   Dtype  
---  ------                      --------------   -----  
 0   Chicago                     100000 non-null  float64
 1   Houston                     100000 non-null  float64
 2   Los Angeles                 100000 non-null  float64
 3   Miami                       100000 non-null  float64
 4   New York                    100000 non-null  float64
 5   CustomerId                  100000 non-null  int64  
 6   Name                        0 non-null       float64
 7   Age                         100000 non-null  int64  
 8   Gender                      100000 non-null  int64  
 9   Subscription_Length_Months  100000 non-null  int64  
 10  Monthly_Bill                100000 non-null  float64
```

```
 11  Total_Usage_GB                 100000 non-null  int64
 12  Churn                          100000 non-null  int64
dtypes: float64(7), int64(6)
memory usage: 9.9 MB
```

Train Test Splitting

```
y = np.array(df['Churn'])
y
df = df.drop(columns=['Churn'], axis = 1)
df
```

```
       Chicago  Houston  Los Angeles  Miami  New York  CustomerId
Name  Age  \
0          0.0      0.0          1.0    0.0       0.0           1
NaN   63
1          0.0      0.0          0.0    0.0       1.0           2
NaN   62
2          0.0      0.0          1.0    0.0       0.0           3
NaN   24
3          0.0      0.0          0.0    1.0       0.0           4
NaN   36
4          0.0      0.0          0.0    1.0       0.0           5
NaN   46
...        ...      ...          ...    ...       ...         ...   ..
.  ...
99995      0.0      1.0          0.0    0.0       0.0       99996
NaN   33
99996      0.0      0.0          0.0    0.0       1.0       99997
NaN   62
99997      1.0      0.0          0.0    0.0       0.0       99998
NaN   64
99998      0.0      0.0          0.0    0.0       1.0       99999
NaN   51
99999      0.0      0.0          1.0    0.0       0.0      100000
NaN   27

       Gender  Subscription_Length_Months  Monthly_Bill
Total_Usage_GB
0           1                          17         73.36
236
1           0                           1         48.76
172
2           0                           5         85.47
460
3           0                           3         97.94
297
```

```
4           0                          19         58.14
266
...       ...                         ...           ...              ..
.
99995       1                          23         55.13
226
99996       0                          19         61.65
351
99997       1                          17         96.11
251
99998       0                          20         49.25
434
99999       0                          19         76.57
173

[100000 rows x 12 columns]

# drop CustomerId column
df = df.drop(columns=['CustomerId', 'Name'],axis=1)
X = df
X

       Chicago  Houston  Los Angeles  Miami  New York  Age  Gender  \
0          0.0      0.0          1.0    0.0       0.0   63       1
1          0.0      0.0          0.0    0.0       1.0   62       0
2          0.0      0.0          1.0    0.0       0.0   24       0
3          0.0      0.0          0.0    1.0       0.0   36       0
4          0.0      0.0          0.0    1.0       0.0   46       0
...        ...      ...          ...    ...       ...  ...     ...
99995      0.0      1.0          0.0    0.0       0.0   33       1
99996      0.0      0.0          0.0    0.0       1.0   62       0
99997      1.0      0.0          0.0    0.0       0.0   64       1
99998      0.0      0.0          0.0    0.0       1.0   51       0
99999      0.0      0.0          1.0    0.0       0.0   27       0

       Subscription_Length_Months  Monthly_Bill  Total_Usage_GB
0                              17         73.36             236
1                               1         48.76             172
2                               5         85.47             460
3                               3         97.94             297
4                              19         58.14             266
...                           ...           ...             ...
99995                          23         55.13             226
99996                          19         61.65             351
99997                          17         96.11             251
99998                          20         49.25             434
99999                          19         76.57             173

[100000 rows x 10 columns]
```

```python
# splitting the dataset
X_train,X_test,y_train,y_test =
train_test_split(X,y,test_size=0.2,random_state=42)

X_train
```

```
       Chicago  Houston  Los Angeles  Miami  New York  Age  Gender  \
75220      0.0      0.0          0.0    0.0       1.0   54       0
48955      0.0      0.0          0.0    0.0       1.0   28       1
44966      1.0      0.0          0.0    0.0       0.0   57       1
13568      0.0      1.0          0.0    0.0       0.0   19       1
92727      0.0      0.0          0.0    1.0       0.0   56       0
...        ...      ...          ...    ...       ...  ...     ...
6265       0.0      0.0          0.0    1.0       0.0   35       1
54886      1.0      0.0          0.0    0.0       0.0   56       1
76820      0.0      1.0          0.0    0.0       0.0   69       1
860        1.0      0.0          0.0    0.0       0.0   55       1
15795      0.0      0.0          1.0    0.0       0.0   26       0

       Subscription_Length_Months  Monthly_Bill  Total_Usage_GB
75220                           5         84.50             205
48955                          24         82.06             239
44966                          12         52.29              62
13568                          19         32.57             173
92727                           8         33.52             314
...                           ...           ...             ...
6265                           21         67.33             235
54886                          13         85.40             347
76820                           2         76.24             321
860                            12         89.19             315
15795                          17         70.41             335

[80000 rows x 10 columns]
```

Feature Scaling

Standardization (z-score) - Normalization

```python
X_train.describe()
```

```
            Chicago       Houston   Los Angeles         Miami      New
York  \
count  80000.000000  80000.000000  80000.000000  80000.000000
80000.000000
mean       0.198313      0.201387      0.200175      0.201513
0.198612
std        0.398731      0.401039      0.400134      0.401132
0.398958
```

```
min        0.000000      0.000000      0.000000      0.000000
0.000000
25%        0.000000      0.000000      0.000000      0.000000
0.000000
50%        0.000000      0.000000      0.000000      0.000000
0.000000
75%        0.000000      0.000000      0.000000      0.000000
0.000000
max        1.000000      1.000000      1.000000      1.000000
1.000000

                Age        Gender  Subscription_Length_Months
Monthly_Bill  \
count  80000.000000  80000.000000                 80000.00000
80000.000000
mean      44.016225      0.497762                    12.48990
65.076958
std       15.278733      0.499998                     6.91766
20.227098
min       18.000000      0.000000                     1.00000
30.000000
25%       31.000000      0.000000                     6.00000
47.600000
50%       44.000000      0.000000                    12.00000
65.030000
75%       57.000000      1.000000                    18.00000
82.700000
max       70.000000      1.000000                    24.00000
100.000000

       Total_Usage_GB
count    80000.000000
mean       274.662787
std        130.510754
min         50.000000
25%        161.000000
50%        274.000000
75%        388.000000
max        500.000000

# Feature scaling using maximum values
X_train['Age'] = X_train['Age']/70
X_train['Subscription_Length_Months'] =
X_train['Subscription_Length_Months']/24
X_train['Monthly_Bill'] = X_train['Monthly_Bill']/100
X_train['Total_Usage_GB'] = X_train['Total_Usage_GB']/500
X_train

       Chicago  Houston  Los Angeles  Miami  New York       Age
Gender  \
```

```
75220       0.0      0.0          0.0    0.0       1.0  0.771429
0
48955       0.0      0.0          0.0    0.0       1.0  0.400000
1
44966       1.0      0.0          0.0    0.0       0.0  0.814286
1
13568       0.0      1.0          0.0    0.0       0.0  0.271429
1
92727       0.0      0.0          0.0    1.0       0.0  0.800000
0
...         ...      ...          ...    ...       ...       ...     ..
.
6265        0.0      0.0          0.0    1.0       0.0  0.500000
1
54886       1.0      0.0          0.0    0.0       0.0  0.800000
1
76820       0.0      1.0          0.0    0.0       0.0  0.985714
1
860         1.0      0.0          0.0    0.0       0.0  0.785714
1
15795       0.0      0.0          1.0    0.0       0.0  0.371429
0

       Subscription_Length_Months  Monthly_Bill  Total_Usage_GB
75220                    0.208333        0.8450           0.410
48955                    1.000000        0.8206           0.478
44966                    0.500000        0.5229           0.124
13568                    0.791667        0.3257           0.346
92727                    0.333333        0.3352           0.628
...                           ...           ...             ...
6265                     0.875000        0.6733           0.470
54886                    0.541667        0.8540           0.694
76820                    0.083333        0.7624           0.642
860                      0.500000        0.8919           0.630
15795                    0.708333        0.7041           0.670

[80000 rows x 10 columns]
```

Model Building

Logistic Regression

```
print(X_train)
print(X_train.shape)

       Chicago  Houston  Los Angeles  Miami  New York       Age
Gender  \
```

```
75220      0.0      0.0          0.0    0.0       1.0  0.771429
0
48955      0.0      0.0          0.0    0.0       1.0  0.400000
1
44966      1.0      0.0          0.0    0.0       0.0  0.814286
1
13568      0.0      1.0          0.0    0.0       0.0  0.271429
1
92727      0.0      0.0          0.0    1.0       0.0  0.800000
0
...        ...      ...          ...    ...       ...       ...    ..
.
6265       0.0      0.0          0.0    1.0       0.0  0.500000
1
54886      1.0      0.0          0.0    0.0       0.0  0.800000
1
76820      0.0      1.0          0.0    0.0       0.0  0.985714
1
860        1.0      0.0          0.0    0.0       0.0  0.785714
1
15795      0.0      0.0          1.0    0.0       0.0  0.371429
0

       Subscription_Length_Months  Monthly_Bill  Total_Usage_GB
75220                    0.208333        0.8450           0.410
48955                    1.000000        0.8206           0.478
44966                    0.500000        0.5229           0.124
13568                    0.791667        0.3257           0.346
92727                    0.333333        0.3352           0.628
...                           ...           ...             ...
6265                     0.875000        0.6733           0.470
54886                    0.541667        0.8540           0.694
76820                    0.083333        0.7624           0.642
860                      0.500000        0.8919           0.630
15795                    0.708333        0.7041           0.670

[80000 rows x 10 columns]
(80000, 10)
```

```
y_train.shape
```

```
(80000,)
```

```
lg_model = LogisticRegression()
lg_model = lg_model.fit(X_train,y_train)
```

```
y_pred_lg_train = lg_model.predict(X_train)
print(classification_report(y_train,y_pred_lg_train))
print(y_train)
print(y_pred_lg_train)
```

```
              precision    recall  f1-score   support

           0       0.50      0.62      0.56     40142
           1       0.50      0.38      0.44     39858

    accuracy                           0.50     80000
   macro avg       0.50      0.50      0.50     80000
weighted avg       0.50      0.50      0.50     80000

[1 1 1 ... 1 1 0]
[1 1 1 ... 0 0 0]
```

Total overall accuracy is 50%

Precision for True Negatives is 51% and that of True Positives is 51%

The performance is average

Support Vector Machines

```
svm_model = SVC()
svm_model = svm_model.fit(X_train,y_train)

y_pred_svm = svm_model.predict(X_test)
print(classification_report(y_test,y_pred_svm))
```

The performance of SVM is quite similar to Logistic Regression

However in some cases it performes better, like in case of recall and f1-score

Random Forest Classifier

```
rand_forest_model = RandomForestClassifier()
rand_forest_model.fit(X_train,y_train)

y_pred_forest = rand_forest_model.predict(X_test)
print(classification_report(y_test,y_pred_forest))

rand_forest_model.score(X_test,y_test)
```

Overall the Random Forest is performing worse than both Logistic Regression and SVM with an accuracy of less than 50%

Fine Tuning Random Forest

Checking the performance by increasing the number of Trees

```
rand_forest_model = RandomForestClassifier(n_estimators=20)
rand_forest_model.fit(X_train,y_train)

rand_forest_model.score(X_test,y_test)
```

For 20 trees the model does not improve that much

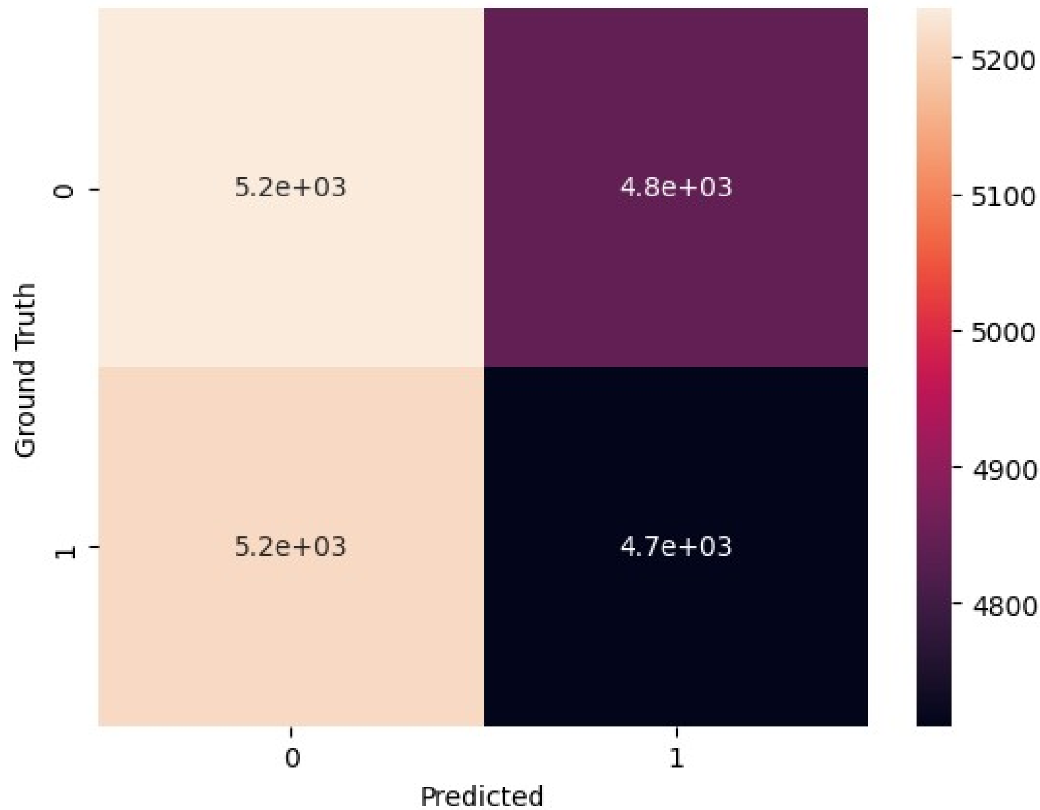
Further increasing the number of Trees

```
rand_forest_model = RandomForestClassifier(n_estimators=40)
rand_forest_model.fit(X_train,y_train)

rand_forest_model.score(X_test,y_test)

rand_forest_pred = rand_forest_model.predict(X_test)
```

Best performance occurs with n_estimators = 40

Further increasing the value leads to diminishing returns

```
# Plotting Confusion Matrix for Random Forest
cm = confusion_matrix(y_test,rand_forest_pred)

cm

# Visualizing the confusion matrix
%matplotlib inline
import seaborn as sns
sns.heatmap(cm,annot=True)
plt.xlabel('Predicted')
plt.ylabel('Ground Truth')
plt.show()
```



- 5200 True Negatives were correctly predicted

- 4700 True Positives were correctly predicted

- A similar number of predicted values were wrongly predicted

-- This implies the accuracy of the model is around 50% as seen previously

Neural Network

```
def plot_loss(history):
  plt.plot(history.history['loss'],label='loss')
  plt.plot(history.history['val_loss'],label='val_loss')
  plt.xlabel('Epoch')
  plt.ylabel('Binary Crossentropy')
  plt.legend()
  plt.grid(True)
  plt.show()

def plot_accuracy(history):
  plt.plot(history.history['accuracy'],label='accuracy')
  plt.plot(history.history['val_accuracy'],label='val_accuracy')
```

```python
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.legend()
  plt.grid(True)
  plt.show()


nn_model = tf.keras.Sequential([
    tf.keras.layers.Dense(64,activation='relu'),
    tf.keras.layers.Dense(32,activation='relu'),
    tf.keras.layers.Dense(1,activation='sigmoid'),
])

nn_model.compile(optimizer=tf.keras.optimizers.Adam(0.001),loss='binar
y_crossentropy',metrics=['accuracy'])

history = nn_model.fit(
    X_train, y_train,epochs=100,batch_size=32,validation_split=0.2
)
```

```
Epoch 1/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6945 - accuracy: 0.5004 - val_loss: 0.6940 - val_accuracy: 0.5013
Epoch 2/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6934 - accuracy: 0.5054 - val_loss: 0.6933 - val_accuracy: 0.5003
Epoch 3/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6932 - accuracy: 0.5054 - val_loss: 0.6933 - val_accuracy: 0.4963
Epoch 4/100
2000/2000 [==============================] - 7s 4ms/step - loss:
0.6931 - accuracy: 0.5064 - val_loss: 0.6938 - val_accuracy: 0.4970
Epoch 5/100
2000/2000 [==============================] - 14s 7ms/step - loss:
0.6931 - accuracy: 0.5089 - val_loss: 0.6933 - val_accuracy: 0.5019
Epoch 6/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6930 - accuracy: 0.5090 - val_loss: 0.6934 - val_accuracy: 0.5002
Epoch 7/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6930 - accuracy: 0.5102 - val_loss: 0.6935 - val_accuracy: 0.5017
Epoch 8/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6929 - accuracy: 0.5088 - val_loss: 0.6938 - val_accuracy: 0.4980
Epoch 9/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6928 - accuracy: 0.5119 - val_loss: 0.6938 - val_accuracy: 0.4986
Epoch 10/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6928 - accuracy: 0.5132 - val_loss: 0.6934 - val_accuracy: 0.5038
```

```
Epoch 11/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6927 - accuracy: 0.5126 - val_loss: 0.6933 - val_accuracy: 0.5006
Epoch 12/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6926 - accuracy: 0.5121 - val_loss: 0.6935 - val_accuracy: 0.4992
Epoch 13/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6925 - accuracy: 0.5164 - val_loss: 0.6933 - val_accuracy: 0.5008
Epoch 14/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6925 - accuracy: 0.5138 - val_loss: 0.6940 - val_accuracy: 0.5011
Epoch 15/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6925 - accuracy: 0.5160 - val_loss: 0.6939 - val_accuracy: 0.4972
Epoch 16/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6924 - accuracy: 0.5158 - val_loss: 0.6946 - val_accuracy: 0.5008
Epoch 17/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6923 - accuracy: 0.5186 - val_loss: 0.6944 - val_accuracy: 0.4979
Epoch 18/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6922 - accuracy: 0.5168 - val_loss: 0.6937 - val_accuracy: 0.5001
Epoch 19/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6922 - accuracy: 0.5178 - val_loss: 0.6952 - val_accuracy: 0.5012
Epoch 20/100
2000/2000 [==============================] - 12s 6ms/step - loss:
0.6921 - accuracy: 0.5194 - val_loss: 0.6942 - val_accuracy: 0.4980
Epoch 21/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6921 - accuracy: 0.5193 - val_loss: 0.6946 - val_accuracy: 0.4947
Epoch 22/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6920 - accuracy: 0.5195 - val_loss: 0.6949 - val_accuracy: 0.4964
Epoch 23/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6919 - accuracy: 0.5204 - val_loss: 0.6952 - val_accuracy: 0.4964
Epoch 24/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6919 - accuracy: 0.5203 - val_loss: 0.6942 - val_accuracy: 0.5058
Epoch 25/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6919 - accuracy: 0.5213 - val_loss: 0.6946 - val_accuracy: 0.4999
Epoch 26/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6918 - accuracy: 0.5200 - val_loss: 0.6944 - val_accuracy: 0.5020
Epoch 27/100
```

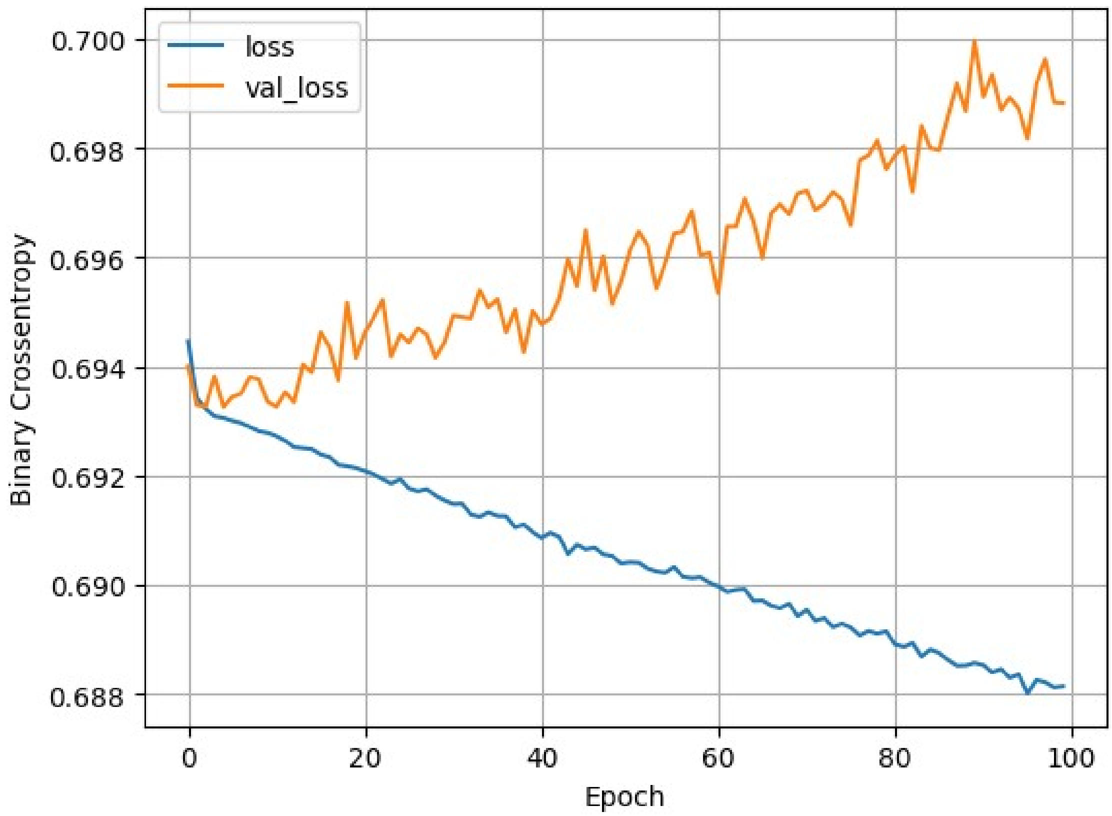
```
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6917 - accuracy: 0.5214 - val_loss: 0.6947 - val_accuracy: 0.4955
Epoch 28/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6918 - accuracy: 0.5209 - val_loss: 0.6946 - val_accuracy: 0.5010
Epoch 29/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6916 - accuracy: 0.5235 - val_loss: 0.6942 - val_accuracy: 0.5050
Epoch 30/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6915 - accuracy: 0.5211 - val_loss: 0.6944 - val_accuracy: 0.5026
Epoch 31/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6915 - accuracy: 0.5241 - val_loss: 0.6949 - val_accuracy: 0.5011
Epoch 32/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6915 - accuracy: 0.5228 - val_loss: 0.6949 - val_accuracy: 0.4990
Epoch 33/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6913 - accuracy: 0.5247 - val_loss: 0.6949 - val_accuracy: 0.5014
Epoch 34/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6912 - accuracy: 0.5262 - val_loss: 0.6954 - val_accuracy: 0.5024
Epoch 35/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6913 - accuracy: 0.5248 - val_loss: 0.6951 - val_accuracy: 0.5016
Epoch 36/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6913 - accuracy: 0.5249 - val_loss: 0.6952 - val_accuracy: 0.5029
Epoch 37/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6913 - accuracy: 0.5258 - val_loss: 0.6946 - val_accuracy: 0.5073
Epoch 38/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6911 - accuracy: 0.5249 - val_loss: 0.6950 - val_accuracy: 0.5039
Epoch 39/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6911 - accuracy: 0.5239 - val_loss: 0.6943 - val_accuracy: 0.5076
Epoch 40/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6910 - accuracy: 0.5266 - val_loss: 0.6950 - val_accuracy: 0.5056
Epoch 41/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6909 - accuracy: 0.5260 - val_loss: 0.6948 - val_accuracy: 0.5027
Epoch 42/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6910 - accuracy: 0.5267 - val_loss: 0.6949 - val_accuracy: 0.5063
Epoch 43/100
2000/2000 [==============================] - 5s 2ms/step - loss:
```

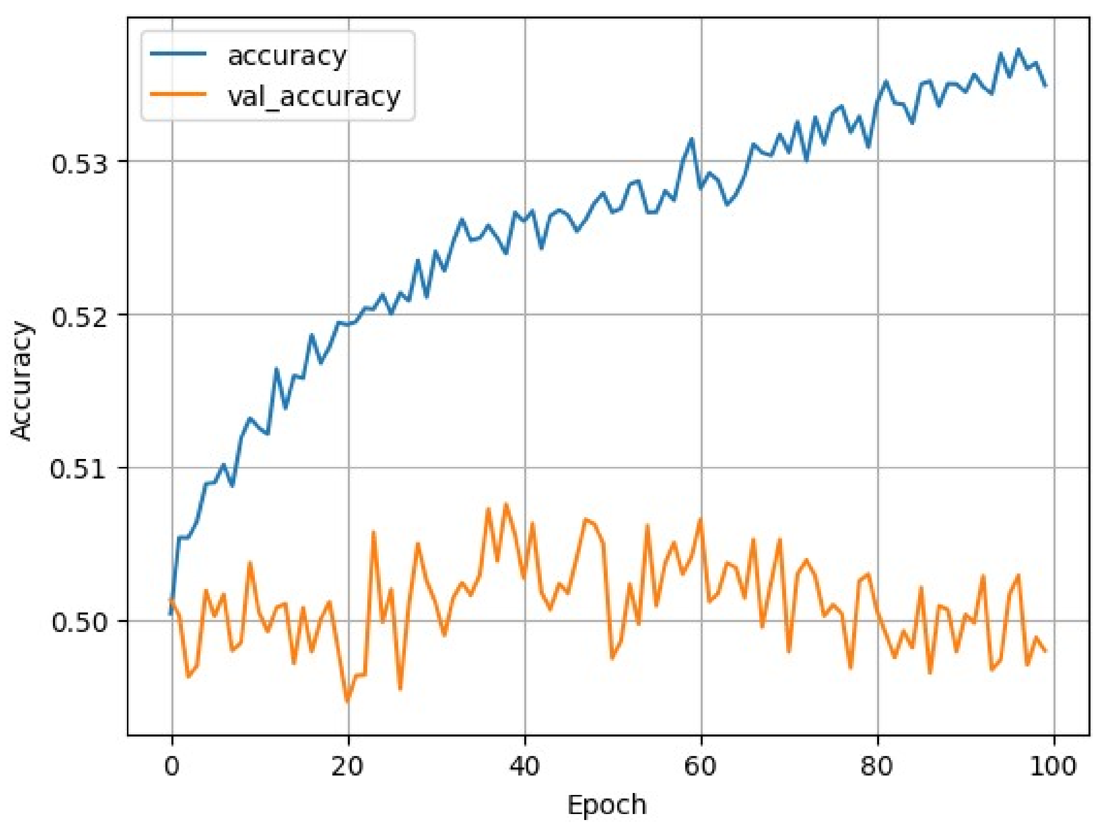
```
0.6909 - accuracy: 0.5243 - val_loss: 0.6953 - val_accuracy: 0.5018
Epoch 44/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6906 - accuracy: 0.5264 - val_loss: 0.6960 - val_accuracy: 0.5007
Epoch 45/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6907 - accuracy: 0.5268 - val_loss: 0.6955 - val_accuracy: 0.5024
Epoch 46/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6906 - accuracy: 0.5265 - val_loss: 0.6965 - val_accuracy: 0.5017
Epoch 47/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6907 - accuracy: 0.5254 - val_loss: 0.6954 - val_accuracy: 0.5041
Epoch 48/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6906 - accuracy: 0.5261 - val_loss: 0.6960 - val_accuracy: 0.5066
Epoch 49/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6905 - accuracy: 0.5272 - val_loss: 0.6951 - val_accuracy: 0.5063
Epoch 50/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6904 - accuracy: 0.5279 - val_loss: 0.6956 - val_accuracy: 0.5050
Epoch 51/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6904 - accuracy: 0.5266 - val_loss: 0.6961 - val_accuracy: 0.4975
Epoch 52/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6904 - accuracy: 0.5269 - val_loss: 0.6965 - val_accuracy: 0.4986
Epoch 53/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6903 - accuracy: 0.5285 - val_loss: 0.6962 - val_accuracy: 0.5024
Epoch 54/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6902 - accuracy: 0.5287 - val_loss: 0.6954 - val_accuracy: 0.4997
Epoch 55/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6902 - accuracy: 0.5266 - val_loss: 0.6959 - val_accuracy: 0.5062
Epoch 56/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6903 - accuracy: 0.5266 - val_loss: 0.6964 - val_accuracy: 0.5009
Epoch 57/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6901 - accuracy: 0.5280 - val_loss: 0.6965 - val_accuracy: 0.5037
Epoch 58/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6901 - accuracy: 0.5274 - val_loss: 0.6969 - val_accuracy: 0.5051
Epoch 59/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6901 - accuracy: 0.5299 - val_loss: 0.6960 - val_accuracy: 0.5030
```

```
Epoch 60/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6900 - accuracy: 0.5314 - val_loss: 0.6961 - val_accuracy: 0.5042
Epoch 61/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6900 - accuracy: 0.5282 - val_loss: 0.6953 - val_accuracy: 0.5066
Epoch 62/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6899 - accuracy: 0.5292 - val_loss: 0.6966 - val_accuracy: 0.5012
Epoch 63/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6899 - accuracy: 0.5287 - val_loss: 0.6966 - val_accuracy: 0.5017
Epoch 64/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6899 - accuracy: 0.5271 - val_loss: 0.6971 - val_accuracy: 0.5038
Epoch 65/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6897 - accuracy: 0.5278 - val_loss: 0.6967 - val_accuracy: 0.5034
Epoch 66/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6897 - accuracy: 0.5290 - val_loss: 0.6960 - val_accuracy: 0.5014
Epoch 67/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6896 - accuracy: 0.5311 - val_loss: 0.6968 - val_accuracy: 0.5052
Epoch 68/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6896 - accuracy: 0.5305 - val_loss: 0.6970 - val_accuracy: 0.4996
Epoch 69/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6897 - accuracy: 0.5303 - val_loss: 0.6968 - val_accuracy: 0.5025
Epoch 70/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6894 - accuracy: 0.5317 - val_loss: 0.6972 - val_accuracy: 0.5052
Epoch 71/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6895 - accuracy: 0.5305 - val_loss: 0.6972 - val_accuracy: 0.4979
Epoch 72/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6893 - accuracy: 0.5325 - val_loss: 0.6969 - val_accuracy: 0.5030
Epoch 73/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6894 - accuracy: 0.5300 - val_loss: 0.6970 - val_accuracy: 0.5039
Epoch 74/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6892 - accuracy: 0.5328 - val_loss: 0.6972 - val_accuracy: 0.5029
Epoch 75/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6893 - accuracy: 0.5311 - val_loss: 0.6971 - val_accuracy: 0.5002
Epoch 76/100
```

```
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6892 - accuracy: 0.5331 - val_loss: 0.6966 - val_accuracy: 0.5010
Epoch 77/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6891 - accuracy: 0.5336 - val_loss: 0.6978 - val_accuracy: 0.5004
Epoch 78/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6892 - accuracy: 0.5319 - val_loss: 0.6979 - val_accuracy: 0.4969
Epoch 79/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6891 - accuracy: 0.5329 - val_loss: 0.6981 - val_accuracy: 0.5026
Epoch 80/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6891 - accuracy: 0.5309 - val_loss: 0.6976 - val_accuracy: 0.5030
Epoch 81/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6889 - accuracy: 0.5338 - val_loss: 0.6979 - val_accuracy: 0.5006
Epoch 82/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6889 - accuracy: 0.5352 - val_loss: 0.6980 - val_accuracy: 0.4991
Epoch 83/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6889 - accuracy: 0.5338 - val_loss: 0.6972 - val_accuracy: 0.4976
Epoch 84/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6887 - accuracy: 0.5337 - val_loss: 0.6984 - val_accuracy: 0.4993
Epoch 85/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6888 - accuracy: 0.5324 - val_loss: 0.6980 - val_accuracy: 0.4982
Epoch 86/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6888 - accuracy: 0.5350 - val_loss: 0.6980 - val_accuracy: 0.5021
Epoch 87/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6886 - accuracy: 0.5352 - val_loss: 0.6986 - val_accuracy: 0.4966
Epoch 88/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6885 - accuracy: 0.5335 - val_loss: 0.6992 - val_accuracy: 0.5009
Epoch 89/100
2000/2000 [==============================] - 7s 4ms/step - loss:
0.6885 - accuracy: 0.5350 - val_loss: 0.6987 - val_accuracy: 0.5007
Epoch 90/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6886 - accuracy: 0.5350 - val_loss: 0.7000 - val_accuracy: 0.4979
Epoch 91/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6885 - accuracy: 0.5345 - val_loss: 0.6989 - val_accuracy: 0.5004
Epoch 92/100
2000/2000 [==============================] - 5s 3ms/step - loss:
```

```
0.6884 - accuracy: 0.5356 - val_loss: 0.6994 - val_accuracy: 0.4998
Epoch 93/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6885 - accuracy: 0.5348 - val_loss: 0.6987 - val_accuracy: 0.5029
Epoch 94/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6883 - accuracy: 0.5344 - val_loss: 0.6989 - val_accuracy: 0.4967
Epoch 95/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6884 - accuracy: 0.5370 - val_loss: 0.6987 - val_accuracy: 0.4974
Epoch 96/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6880 - accuracy: 0.5355 - val_loss: 0.6982 - val_accuracy: 0.5017
Epoch 97/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6883 - accuracy: 0.5373 - val_loss: 0.6992 - val_accuracy: 0.5029
Epoch 98/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6882 - accuracy: 0.5360 - val_loss: 0.6996 - val_accuracy: 0.4971
Epoch 99/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6881 - accuracy: 0.5364 - val_loss: 0.6988 - val_accuracy: 0.4989
Epoch 100/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6881 - accuracy: 0.5349 - val_loss: 0.6988 - val_accuracy: 0.4980

plot_loss(history)
plot_accuracy(history)
```


loss
val_loss
Binary Crossentropy
Epoch
0.700
0.698
0.696
0.694
0.692
0.690
0.688
0
20
40
60
80
100


accuracy
val_accuracy
Accuracy
Epoch
0.53
0.52
0.51
0.50
0
20
40
60
80
100

The Neural Networks are performing well as the loss is clearly decreasing and the accuracy is increasing with each iteration.

The accuracy however is 53% which is better than the other algorithms

Fine Tuning model parameters

testing on learning rate = 0.005 and 64 neurons in the 2nd layer

```
nn_model = tf.keras.Sequential([
    tf.keras.layers.Dense(64,activation='relu'),
    tf.keras.layers.Dense(64,activation='relu'),
    tf.keras.layers.Dense(1,activation='sigmoid'),
])

nn_model.compile(optimizer=tf.keras.optimizers.Adam(0.001),loss='binar
y_crossentropy',metrics=['accuracy'])

history = nn_model.fit(
    X_train, y_train,epochs=100,batch_size=32,validation_split=0.2
)
```

```
Epoch 1/100
2000/2000 [==============================] - 9s 3ms/step - loss:
0.6943 - accuracy: 0.4991 - val_loss: 0.6945 - val_accuracy: 0.4971
Epoch 2/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6935 - accuracy: 0.5016 - val_loss: 0.6935 - val_accuracy: 0.4930
Epoch 3/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6932 - accuracy: 0.5040 - val_loss: 0.6932 - val_accuracy: 0.5009
Epoch 4/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6932 - accuracy: 0.5063 - val_loss: 0.6933 - val_accuracy: 0.4960
Epoch 5/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6930 - accuracy: 0.5095 - val_loss: 0.6933 - val_accuracy: 0.4999
Epoch 6/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6929 - accuracy: 0.5086 - val_loss: 0.6935 - val_accuracy: 0.4974
Epoch 7/100
2000/2000 [==============================] - 5s 3ms/step - loss:
```

```
0.6929 - accuracy: 0.5092 - val_loss: 0.6935 - val_accuracy: 0.5039
Epoch 8/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6928 - accuracy: 0.5127 - val_loss: 0.6940 - val_accuracy: 0.5004
Epoch 9/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6928 - accuracy: 0.5110 - val_loss: 0.6935 - val_accuracy: 0.4998
Epoch 10/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6927 - accuracy: 0.5121 - val_loss: 0.6937 - val_accuracy: 0.4978
Epoch 11/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6926 - accuracy: 0.5139 - val_loss: 0.6940 - val_accuracy: 0.4959
Epoch 12/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6927 - accuracy: 0.5147 - val_loss: 0.6940 - val_accuracy: 0.4956
Epoch 13/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6925 - accuracy: 0.5145 - val_loss: 0.6942 - val_accuracy: 0.4996
Epoch 14/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6925 - accuracy: 0.5160 - val_loss: 0.6943 - val_accuracy: 0.5024
Epoch 15/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6924 - accuracy: 0.5142 - val_loss: 0.6941 - val_accuracy: 0.5026
Epoch 16/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6923 - accuracy: 0.5175 - val_loss: 0.6941 - val_accuracy: 0.4992
Epoch 17/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6923 - accuracy: 0.5194 - val_loss: 0.6938 - val_accuracy: 0.5008
Epoch 18/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6923 - accuracy: 0.5155 - val_loss: 0.6941 - val_accuracy: 0.5031
Epoch 19/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6922 - accuracy: 0.5163 - val_loss: 0.6944 - val_accuracy: 0.4983
Epoch 20/100
2000/2000 [==============================] - 4s 2ms/step - loss:
0.6920 - accuracy: 0.5194 - val_loss: 0.6945 - val_accuracy: 0.4989
Epoch 21/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6920 - accuracy: 0.5197 - val_loss: 0.6946 - val_accuracy: 0.5006
Epoch 22/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6920 - accuracy: 0.5200 - val_loss: 0.6943 - val_accuracy: 0.5029
Epoch 23/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6919 - accuracy: 0.5200 - val_loss: 0.6947 - val_accuracy: 0.4979
```

```
Epoch 24/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6918 - accuracy: 0.5190 - val_loss: 0.6942 - val_accuracy: 0.5044
Epoch 25/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6918 - accuracy: 0.5214 - val_loss: 0.6943 - val_accuracy: 0.5033
Epoch 26/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6919 - accuracy: 0.5200 - val_loss: 0.6946 - val_accuracy: 0.5023
Epoch 27/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6918 - accuracy: 0.5213 - val_loss: 0.6948 - val_accuracy: 0.4994
Epoch 28/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6916 - accuracy: 0.5231 - val_loss: 0.6946 - val_accuracy: 0.4991
Epoch 29/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6916 - accuracy: 0.5228 - val_loss: 0.6949 - val_accuracy: 0.5017
Epoch 30/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6915 - accuracy: 0.5220 - val_loss: 0.6948 - val_accuracy: 0.5042
Epoch 31/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6914 - accuracy: 0.5218 - val_loss: 0.6946 - val_accuracy: 0.5031
Epoch 32/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6912 - accuracy: 0.5248 - val_loss: 0.6950 - val_accuracy: 0.5019
Epoch 33/100
2000/2000 [==============================] - 7s 4ms/step - loss:
0.6913 - accuracy: 0.5228 - val_loss: 0.6949 - val_accuracy: 0.4988
Epoch 34/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6912 - accuracy: 0.5228 - val_loss: 0.6948 - val_accuracy: 0.5034
Epoch 35/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6911 - accuracy: 0.5256 - val_loss: 0.6953 - val_accuracy: 0.5042
Epoch 36/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6910 - accuracy: 0.5268 - val_loss: 0.6956 - val_accuracy: 0.5005
Epoch 37/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6909 - accuracy: 0.5254 - val_loss: 0.6955 - val_accuracy: 0.5017
Epoch 38/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6910 - accuracy: 0.5242 - val_loss: 0.6953 - val_accuracy: 0.4998
Epoch 39/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6910 - accuracy: 0.5256 - val_loss: 0.6954 - val_accuracy: 0.4982
Epoch 40/100
```

```
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6908 - accuracy: 0.5276 - val_loss: 0.6949 - val_accuracy: 0.5038
Epoch 41/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6908 - accuracy: 0.5265 - val_loss: 0.6955 - val_accuracy: 0.5021
Epoch 42/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6906 - accuracy: 0.5276 - val_loss: 0.6953 - val_accuracy: 0.5021
Epoch 43/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6907 - accuracy: 0.5263 - val_loss: 0.6957 - val_accuracy: 0.5026
Epoch 44/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6905 - accuracy: 0.5281 - val_loss: 0.6960 - val_accuracy: 0.5027
Epoch 45/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6906 - accuracy: 0.5262 - val_loss: 0.6954 - val_accuracy: 0.5029
Epoch 46/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6905 - accuracy: 0.5260 - val_loss: 0.6964 - val_accuracy: 0.5014
Epoch 47/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6904 - accuracy: 0.5277 - val_loss: 0.6957 - val_accuracy: 0.5038
Epoch 48/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6903 - accuracy: 0.5282 - val_loss: 0.6966 - val_accuracy: 0.5045
Epoch 49/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6902 - accuracy: 0.5294 - val_loss: 0.6959 - val_accuracy: 0.4991
Epoch 50/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6901 - accuracy: 0.5310 - val_loss: 0.6959 - val_accuracy: 0.5013
Epoch 51/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6902 - accuracy: 0.5293 - val_loss: 0.6956 - val_accuracy: 0.5034
Epoch 52/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6901 - accuracy: 0.5285 - val_loss: 0.6959 - val_accuracy: 0.4988
Epoch 53/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6900 - accuracy: 0.5301 - val_loss: 0.6972 - val_accuracy: 0.5008
Epoch 54/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6900 - accuracy: 0.5315 - val_loss: 0.6961 - val_accuracy: 0.5019
Epoch 55/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6899 - accuracy: 0.5302 - val_loss: 0.6962 - val_accuracy: 0.5059
Epoch 56/100
2000/2000 [==============================] - 5s 3ms/step - loss:
```

```
0.6898 - accuracy: 0.5313 - val_loss: 0.6976 - val_accuracy: 0.4976
Epoch 57/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6897 - accuracy: 0.5291 - val_loss: 0.6968 - val_accuracy: 0.4998
Epoch 58/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6898 - accuracy: 0.5316 - val_loss: 0.6962 - val_accuracy: 0.5038
Epoch 59/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6898 - accuracy: 0.5317 - val_loss: 0.6970 - val_accuracy: 0.5018
Epoch 60/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6897 - accuracy: 0.5314 - val_loss: 0.6963 - val_accuracy: 0.5092
Epoch 61/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6897 - accuracy: 0.5298 - val_loss: 0.6960 - val_accuracy: 0.5077
Epoch 62/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6897 - accuracy: 0.5318 - val_loss: 0.6966 - val_accuracy: 0.5060
Epoch 63/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6896 - accuracy: 0.5298 - val_loss: 0.6966 - val_accuracy: 0.5039
Epoch 64/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6894 - accuracy: 0.5328 - val_loss: 0.6967 - val_accuracy: 0.5016
Epoch 65/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6895 - accuracy: 0.5319 - val_loss: 0.6969 - val_accuracy: 0.5030
Epoch 66/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6894 - accuracy: 0.5307 - val_loss: 0.6967 - val_accuracy: 0.5077
Epoch 67/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6892 - accuracy: 0.5322 - val_loss: 0.6969 - val_accuracy: 0.5069
Epoch 68/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6891 - accuracy: 0.5351 - val_loss: 0.6978 - val_accuracy: 0.4979
Epoch 69/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6892 - accuracy: 0.5332 - val_loss: 0.6970 - val_accuracy: 0.5065
Epoch 70/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6892 - accuracy: 0.5332 - val_loss: 0.6969 - val_accuracy: 0.5017
Epoch 71/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6890 - accuracy: 0.5342 - val_loss: 0.6974 - val_accuracy: 0.5001
Epoch 72/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6891 - accuracy: 0.5349 - val_loss: 0.6969 - val_accuracy: 0.5054
```

```
Epoch 73/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6889 - accuracy: 0.5341 - val_loss: 0.6973 - val_accuracy: 0.5059
Epoch 74/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6889 - accuracy: 0.5346 - val_loss: 0.6985 - val_accuracy: 0.5049
Epoch 75/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6888 - accuracy: 0.5364 - val_loss: 0.6975 - val_accuracy: 0.5027
Epoch 76/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6889 - accuracy: 0.5352 - val_loss: 0.6980 - val_accuracy: 0.5021
Epoch 77/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6888 - accuracy: 0.5357 - val_loss: 0.6967 - val_accuracy: 0.5069
Epoch 78/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6887 - accuracy: 0.5360 - val_loss: 0.6977 - val_accuracy: 0.5030
Epoch 79/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6888 - accuracy: 0.5348 - val_loss: 0.6973 - val_accuracy: 0.5042
Epoch 80/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6886 - accuracy: 0.5342 - val_loss: 0.6969 - val_accuracy: 0.5044
Epoch 81/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6885 - accuracy: 0.5353 - val_loss: 0.6970 - val_accuracy: 0.5052
Epoch 82/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6885 - accuracy: 0.5357 - val_loss: 0.6972 - val_accuracy: 0.4997
Epoch 83/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6885 - accuracy: 0.5357 - val_loss: 0.6984 - val_accuracy: 0.4984
Epoch 84/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6885 - accuracy: 0.5379 - val_loss: 0.6974 - val_accuracy: 0.5038
Epoch 85/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6883 - accuracy: 0.5365 - val_loss: 0.6983 - val_accuracy: 0.5039
Epoch 86/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6884 - accuracy: 0.5351 - val_loss: 0.6971 - val_accuracy: 0.5055
Epoch 87/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6883 - accuracy: 0.5364 - val_loss: 0.6979 - val_accuracy: 0.5043
Epoch 88/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6883 - accuracy: 0.5376 - val_loss: 0.6984 - val_accuracy: 0.5032
Epoch 89/100
```

```
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6882 - accuracy: 0.5357 - val_loss: 0.6984 - val_accuracy: 0.5043
Epoch 90/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6882 - accuracy: 0.5364 - val_loss: 0.6974 - val_accuracy: 0.5026
Epoch 91/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6881 - accuracy: 0.5393 - val_loss: 0.6975 - val_accuracy: 0.5063
Epoch 92/100
2000/2000 [==============================] - 7s 4ms/step - loss:
0.6882 - accuracy: 0.5373 - val_loss: 0.6976 - val_accuracy: 0.5081
Epoch 93/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6879 - accuracy: 0.5377 - val_loss: 0.6981 - val_accuracy: 0.5036
Epoch 94/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6880 - accuracy: 0.5372 - val_loss: 0.6977 - val_accuracy: 0.5050
Epoch 95/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6880 - accuracy: 0.5378 - val_loss: 0.6974 - val_accuracy: 0.5080
Epoch 96/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6881 - accuracy: 0.5363 - val_loss: 0.6982 - val_accuracy: 0.5042
Epoch 97/100
2000/2000 [==============================] - 7s 3ms/step - loss:
0.6879 - accuracy: 0.5391 - val_loss: 0.6982 - val_accuracy: 0.5005
Epoch 98/100
2000/2000 [==============================] - 5s 3ms/step - loss:
0.6879 - accuracy: 0.5382 - val_loss: 0.6978 - val_accuracy: 0.5023
Epoch 99/100
2000/2000 [==============================] - 6s 3ms/step - loss:
0.6877 - accuracy: 0.5392 - val_loss: 0.6985 - val_accuracy: 0.5024
Epoch 100/100
2000/2000 [==============================] - 5s 2ms/step - loss:
0.6878 - accuracy: 0.5392 - val_loss: 0.6987 - val_accuracy: 0.5039
```

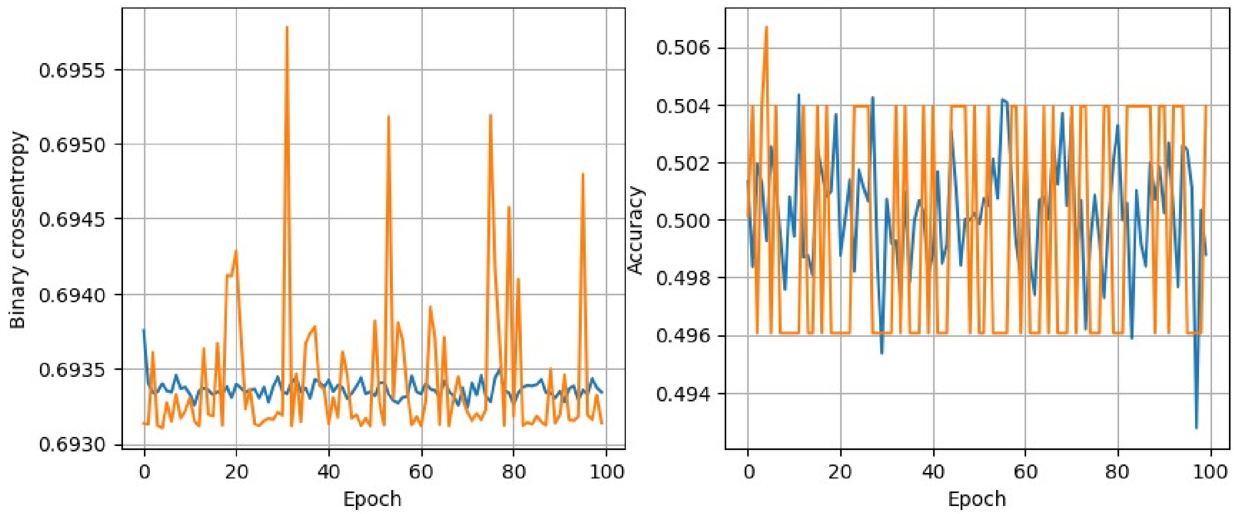
Accuracy is approx 53% which is better than the rest of the algorithms

Script to find the best possible parameters for neural network

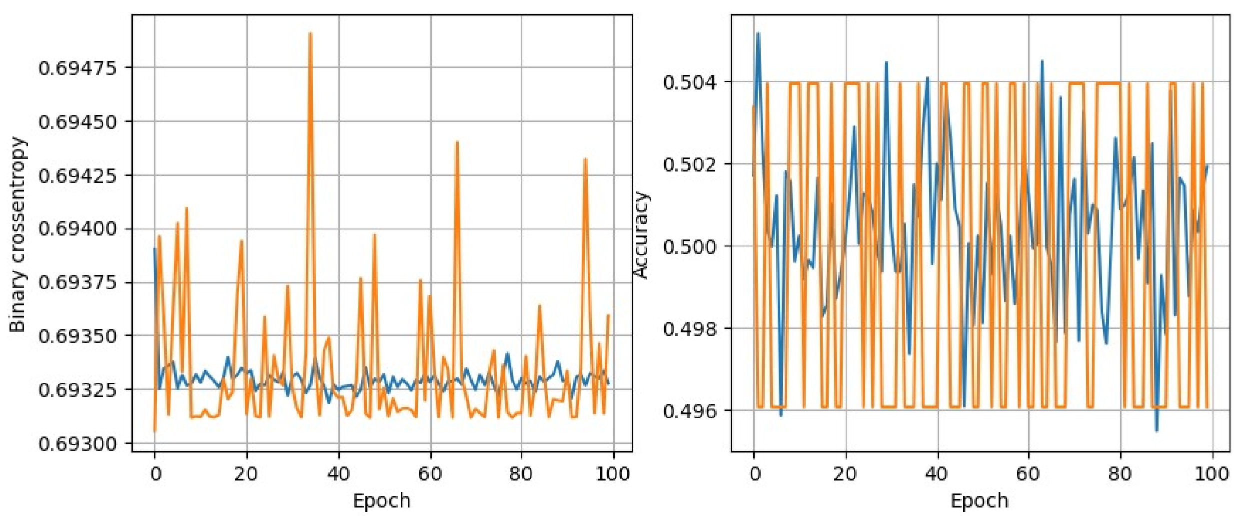
```
def plot_history(history):
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
  ax1.plot(history.history['loss'], label='loss')
  ax1.plot(history.history['val_loss'], label='val_loss')
  ax1.set_xlabel('Epoch')
  ax1.set_ylabel('Binary crossentropy')
  ax1.grid(True)
```

```
  ax2.plot(history.history['accuracy'], label='accuracy')
  ax2.plot(history.history['val_accuracy'], label='val_accuracy')
  ax2.set_xlabel('Epoch')
  ax2.set_ylabel('Accuracy')
  ax2.grid(True)

  plt.show()

def train_model(X_train, y_train, num_nodes, dropout_prob, lr,
batch_size, epochs):
  nn_model = tf.keras.Sequential([
      tf.keras.layers.Dense(num_nodes, activation='relu'),
      tf.keras.layers.Dropout(dropout_prob),
      tf.keras.layers.Dense(num_nodes, activation='relu'),
      tf.keras.layers.Dropout(dropout_prob),
      tf.keras.layers.Dense(1, activation='sigmoid')
  ])

  nn_model.compile(optimizer=tf.keras.optimizers.Adam(lr),
loss='binary_crossentropy',
                  metrics=['accuracy'])
  history = nn_model.fit(
    X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_split=0.2, verbose=0
  )

  return nn_model, history

least_val_loss = float('inf')
least_loss_model = None
epochs=100

"""
  UNCOMMENT THE CODE BELOW TO RUN
"""

# for num_nodes in [16, 32, 64]:
#   for dropout_prob in[0, 0.2]:
#     for lr in [0.01, 0.005, 0.001]:
#       for batch_size in [32, 64]:
#         print(f"{num_nodes} nodes, dropout {dropout_prob}, lr {lr},
batch size {batch_size}")
#         model, history = train_model(X_train, y_train, num_nodes,
dropout_prob, lr, batch_size, epochs)
#         plot_history(history)
#         val_loss = model.evaluate(X_test, y_test)[0]
#         if val_loss < least_val_loss:
#           least_val_loss = val_loss
#           least_loss_model = model
```

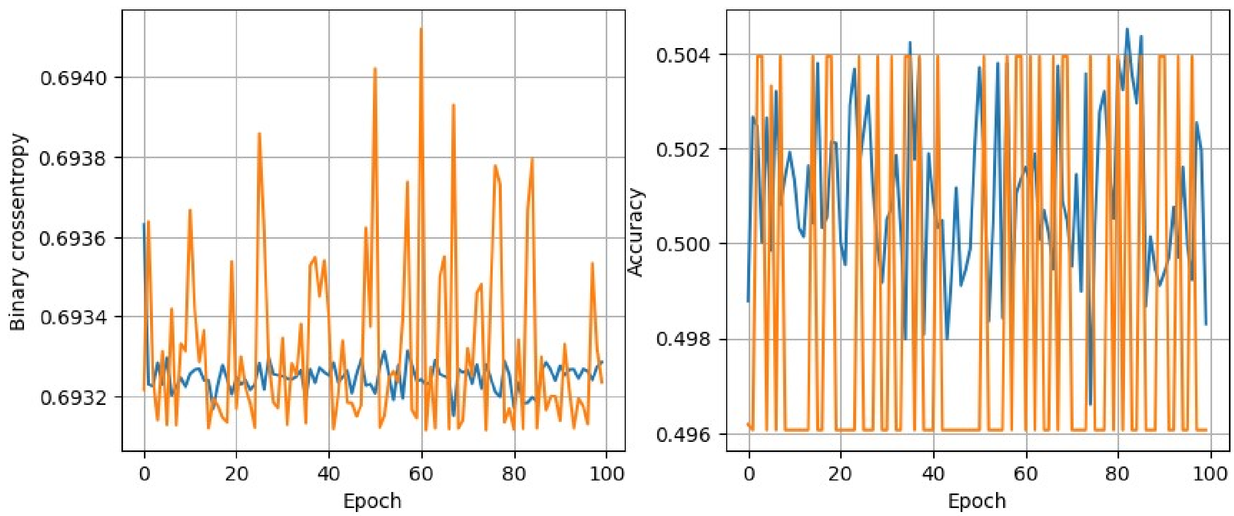
```
16 nodes, dropout 0, lr 0.01, batch size 32
```



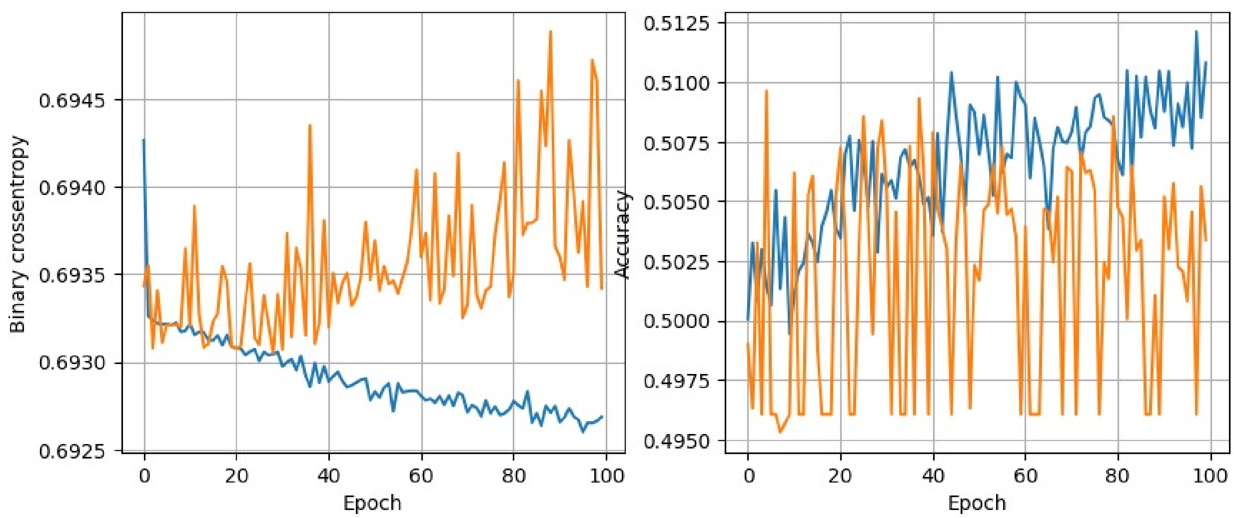

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6932
- accuracy: 0.4961
16 nodes, dropout 0, lr 0.01, batch size 64
```



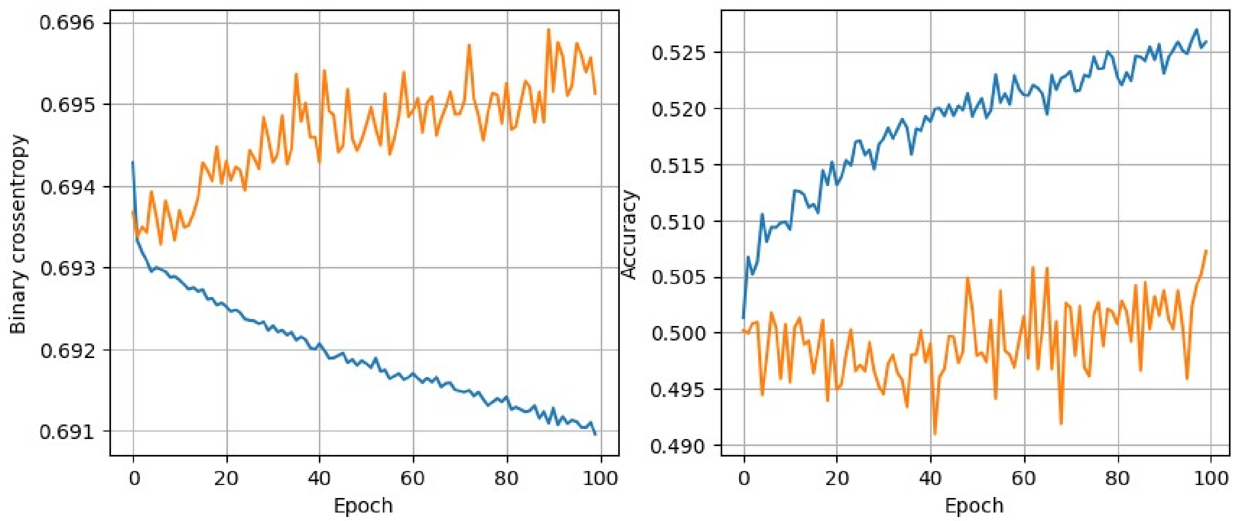

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6932
- accuracy: 0.5039
16 nodes, dropout 0, lr 0.005, batch size 32
```



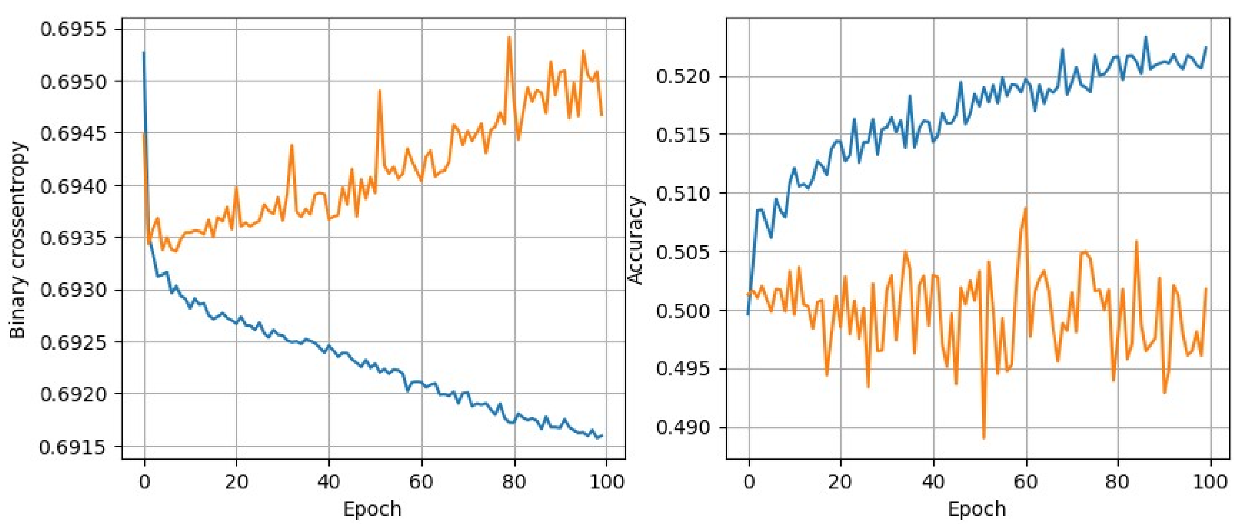

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6931
- accuracy: 0.5039
16 nodes, dropout 0, lr 0.005, batch size 64
```



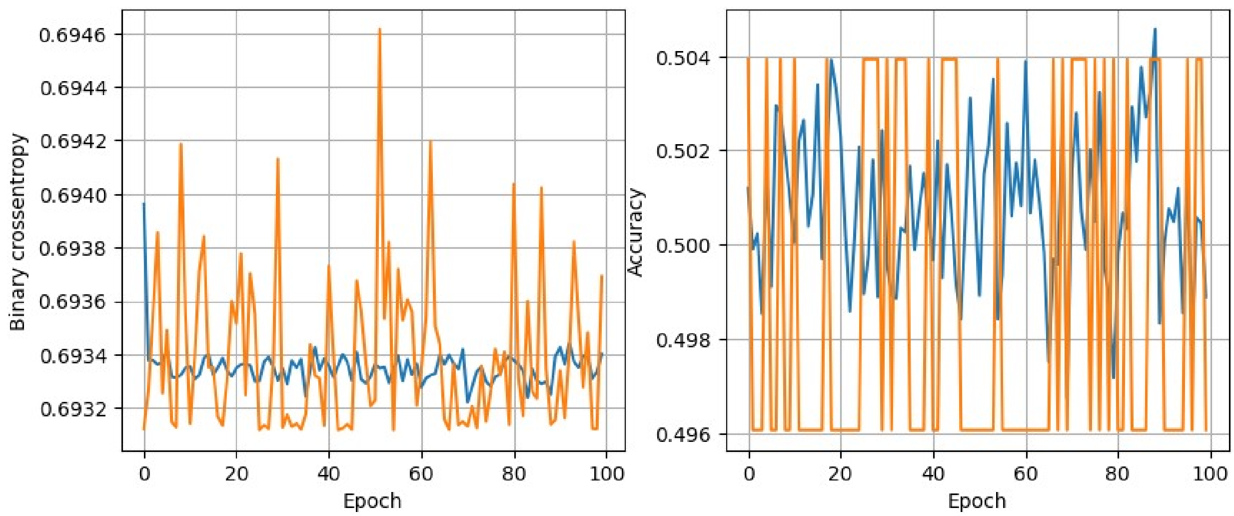

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6937
- accuracy: 0.4975
16 nodes, dropout 0, lr 0.001, batch size 32
```



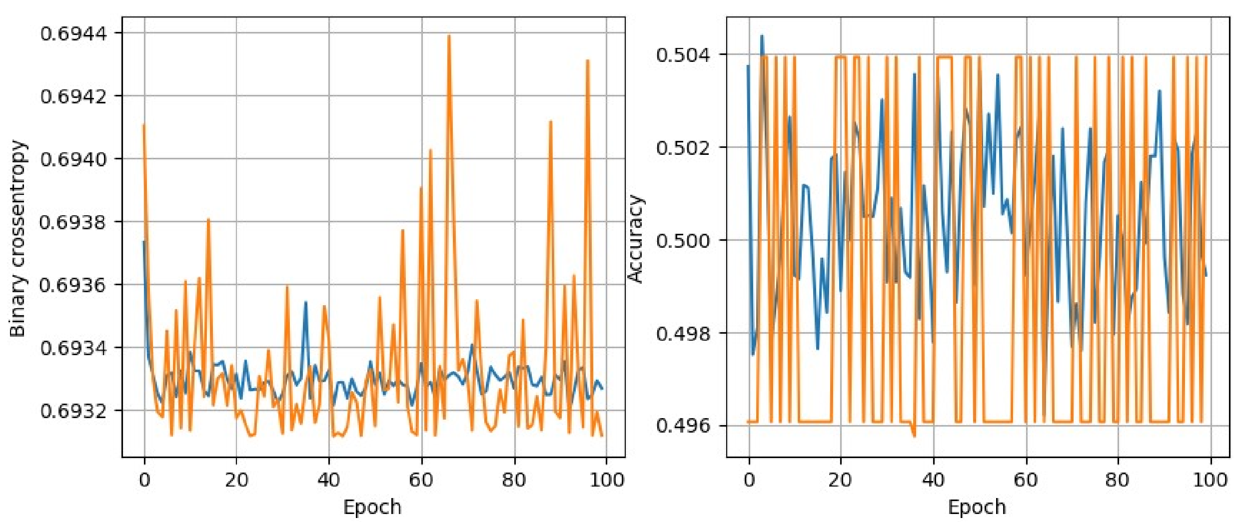

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6958
- accuracy: 0.4988
16 nodes, dropout 0, lr 0.001, batch size 64
```



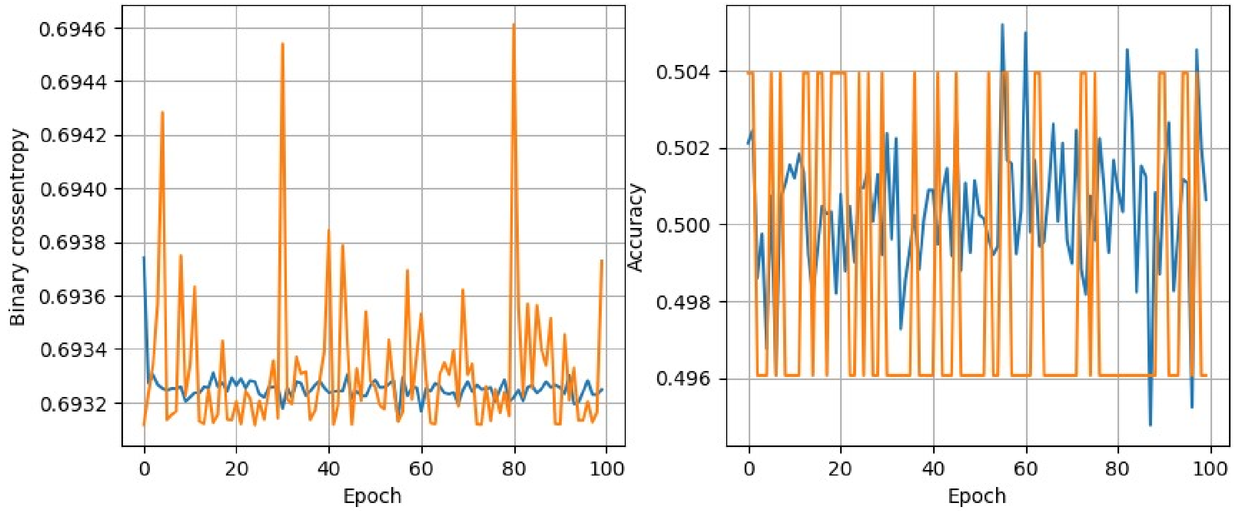

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6947
- accuracy: 0.4988
16 nodes, dropout 0.2, lr 0.01, batch size 32
```



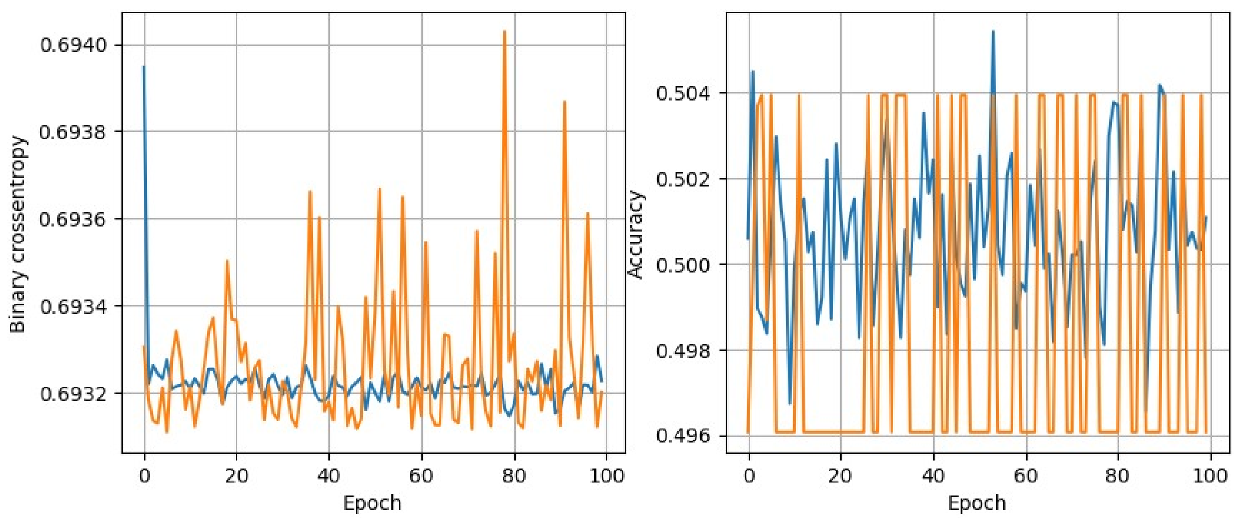

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6933
- accuracy: 0.5039
16 nodes, dropout 0.2, lr 0.01, batch size 64
```



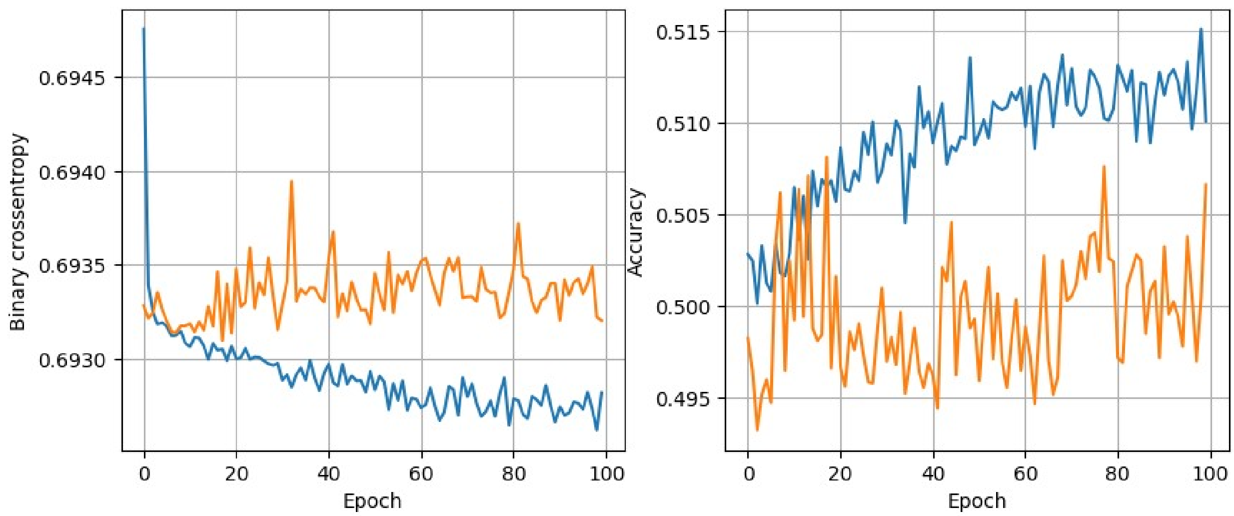

```
625/625 [==============================] - 3s 2ms/step - loss: 0.6933
- accuracy: 0.4961
16 nodes, dropout 0.2, lr 0.005, batch size 32
```



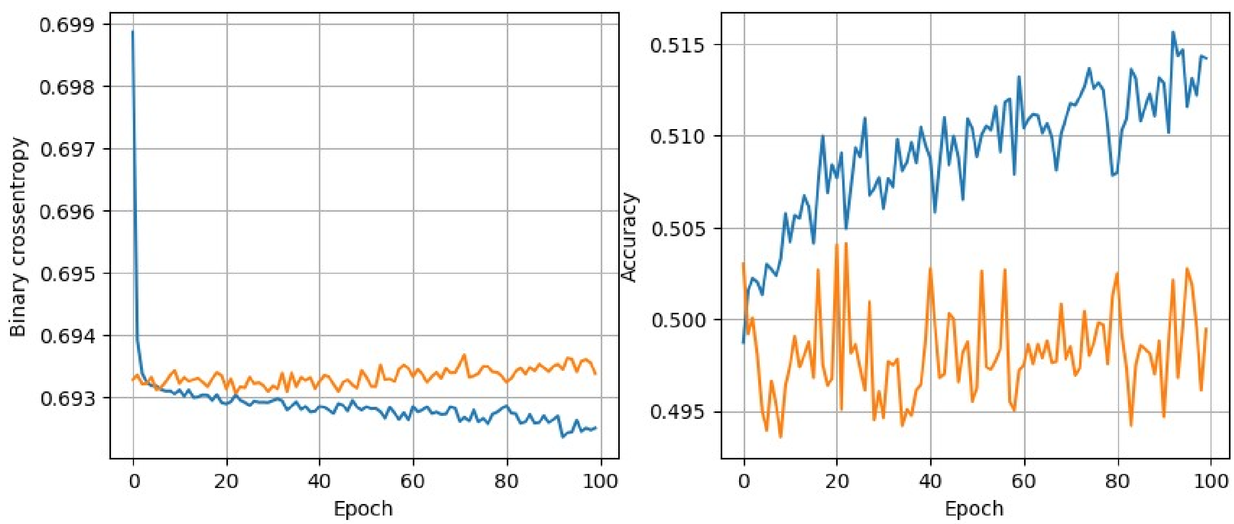

```
625/625 [==============================] - 1s 2ms/step - loss: 0.6933
- accuracy: 0.5039
16 nodes, dropout 0.2, lr 0.005, batch size 64
```




```
625/625 [==============================] - 2s 3ms/step - loss: 0.6931
- accuracy: 0.5039
16 nodes, dropout 0.2, lr 0.001, batch size 32
```




```
625/625 [==============================] - 1s 2ms/step - loss: 0.6933
- accuracy: 0.5016
16 nodes, dropout 0.2, lr 0.001, batch size 64
```




```
625/625 [==============================] - 1s 2ms/step - loss: 0.6937
- accuracy: 0.4982
32 nodes, dropout 0, lr 0.01, batch size 32

----------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
<ipython-input-96-c7802061c1b1> in <cell line: 4>()
      7       for batch_size in [32, 64]:
      8         print(f"{num_nodes} nodes, dropout {dropout_prob}, lr
{lr}, batch size {batch_size}")
```

```
----> 9         model, history = train_model(X_train, y_train,
num_nodes, dropout_prob, lr, batch_size, epochs)
     10         plot_history(history)
     11         val_loss = model.evaluate(X_test, y_test)[0]

<ipython-input-94-97d214b36a92> in train_model(X_train, y_train,
num_nodes, dropout_prob, lr, batch_size, epochs)
     10   nn_model.compile(optimizer=tf.keras.optimizers.Adam(lr),
loss='binary_crossentropy',
     11                   metrics=['accuracy'])
---> 12   history = nn_model.fit(
     13     X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_split=0.2, verbose=0
     14   )

/usr/local/lib/python3.10/dist-packages/keras/utils/traceback_utils.py
in error_handler(*args, **kwargs)
     63         filtered_tb = None
     64         try:
---> 65             return fn(*args, **kwargs)
     66         except Exception as e:
     67             filtered_tb =
_process_traceback_frames(e.__traceback__)

/usr/local/lib/python3.10/dist-packages/keras/engine/training.py in
fit(self, x, y, batch_size, epochs, verbose, callbacks,
validation_split, validation_data, shuffle, class_weight,
sample_weight, initial_epoch, steps_per_epoch, validation_steps,
validation_batch_size, validation_freq, max_queue_size, workers,
use_multiprocessing)
   1683                         ):
   1684
callbacks.on_train_batch_begin(step)
-> 1685                             tmp_logs =
self.train_function(iterator)
   1686                             if data_handler.should_sync:
   1687                                 context.async_wait()

/usr/local/lib/python3.10/dist-packages/tensorflow/python/util/traceba
ck_utils.py in error_handler(*args, **kwargs)
    148     filtered_tb = None
    149     try:
--> 150       return fn(*args, **kwargs)
    151     except Exception as e:
    152       filtered_tb = _process_traceback_frames(e.__traceback__)

/usr/local/lib/python3.10/dist-packages/tensorflow/python/eager/polymo
rphic_function/polymorphic_function.py in __call__(self, *args,
**kwds)
    892
```

```
    893       with OptionalXlaContext(self._jit_compile):
--> 894         result = self._call(*args, **kwds)
    895 
    896       new_tracing_count =
self.experimental_get_tracing_count()

/usr/local/lib/python3.10/dist-packages/tensorflow/python/eager/polymo
rphic_function/polymorphic_function.py in _call(self, *args, **kwds)
    924       # In this case we have created variables on the first
call, so we run the
    925       # defunned version which is guaranteed to never create
variables.
--> 926       return self._no_variable_creation_fn(*args, **kwds)  #
pylint: disable=not-callable
    927     elif self._variable_creation_fn is not None:
    928       # Release the lock early so that multiple threads can
perform the call

/usr/local/lib/python3.10/dist-packages/tensorflow/python/eager/polymo
rphic_function/tracing_compiler.py in __call__(self, *args, **kwargs)
    141       (concrete_function,
    142        filtered_flat_args) = self._maybe_define_function(args,
kwargs)
--> 143     return concrete_function._call_flat(
    144         filtered_flat_args,
captured_inputs=concrete_function.captured_inputs)  # pylint:
disable=protected-access
    145 

/usr/local/lib/python3.10/dist-packages/tensorflow/python/eager/polymo
rphic_function/monomorphic_function.py in _call_flat(self, args,
captured_inputs, cancellation_manager)
   1755         and executing_eagerly):
   1756       # No tape is watching; skip to running the function.
-> 1757       return
self._build_call_outputs(self._inference_function.call(
   1758           ctx, args,
cancellation_manager=cancellation_manager))
   1759     forward_backward =
self._select_forward_and_backward_functions(

/usr/local/lib/python3.10/dist-packages/tensorflow/python/eager/polymo
rphic_function/monomorphic_function.py in call(self, ctx, args,
cancellation_manager)
    379       with _InterpolateFunctionError(self):
    380         if cancellation_manager is None:
--> 381           outputs = execute.execute(
    382               str(self.signature.name),
    383               num_outputs=self._num_outputs,
```

```
/usr/local/lib/python3.10/dist-packages/tensorflow/python/eager/execut
e.py in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     50   try:
     51     ctx.ensure_initialized()
---> 52     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle,
device_name, op_name,
     53                                         inputs, attrs,
num_outputs)
     54   except core._NotOkStatusException as e:

KeyboardInterrupt:
```

```python
import pickle
pickle.dump(lg_model,open('lg_model.pkl','wb'))

loaded_model = pickle.load(open('lg_model.pkl','rb'))
```