

# churn-prediction-using-ann

January 31, 2024

```
[2]: #imports necessary Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')
```

```
[3]: df = pd.read_csv('/kaggle/input/churndata/Churn_Modelling.csv')
df.head()
```

```
[3]:  RowNumber  CustomerId  Surname  CreditScore  Geography  Gender  Age  \
0         1    15634602  Hargrave         619    France  Female  42
1         2    15647311    Hill         608    Spain  Female  41
2         3    15619304    Onio         502    France  Female  42
3         4    15701354    Boni         699    France  Female  39
4         5    15737888  Mitchell         850    Spain  Female  43

      Tenure  Balance  NumOfProducts  HasCrCard  IsActiveMember  \
0         2     0.00             1           1              1
1         1  83807.86             1           0              1
2         8 159660.80             3           1              0
3         1     0.00             2           0              0
4         2 125510.82             1           1              1

      EstimatedSalary  Exited
0         101348.88      1
1         112542.58      0
2         113931.57      1
3          93826.63      0
4          79084.10      0
```

```
[4]: df.shape
```

```
[4]: (10000, 14)
```

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
#   Column                Non-Null Count  Dtype  
---  -
0   RowNumber             10000 non-null  int64   
1   CustomerId            10000 non-null  int64   
2   Surname               10000 non-null  object  
3   CreditScore           10000 non-null  int64   
4   Geography             10000 non-null  object  
5   Gender                10000 non-null  object  
6   Age                   10000 non-null  int64   
7   Tenure                10000 non-null  int64   
8   Balance               10000 non-null  float64  
9   NumOfProducts         10000 non-null  int64   
10  HasCrCard             10000 non-null  int64   
11  IsActiveMember        10000 non-null  int64   
12  EstimatedSalary       10000 non-null  float64  
13  Exited                10000 non-null  int64   
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

```
[6]: df.describe()
```

```
[6]:
```

	RowNumber	CustomerId	CreditScore	Age	Tenure \
count	10000.00000	1.000000e+04	10000.000000	10000.000000	10000.000000
mean	5000.50000	1.569094e+07	650.528800	38.921800	5.012800
std	2886.89568	7.193619e+04	96.653299	10.487806	2.892174
min	1.00000	1.556570e+07	350.000000	18.000000	0.000000
25%	2500.75000	1.562853e+07	584.000000	32.000000	3.000000
50%	5000.50000	1.569074e+07	652.000000	37.000000	5.000000
75%	7500.25000	1.575323e+07	718.000000	44.000000	7.000000
max	10000.00000	1.581569e+07	850.000000	92.000000	10.000000

	Balance	NumOfProducts	HasCrCard	IsActiveMember \
count	10000.000000	10000.000000	10000.000000	10000.000000
mean	76485.889288	1.530200	0.70550	0.515100
std	62397.405202	0.581654	0.45584	0.499797
min	0.000000	1.000000	0.00000	0.000000
25%	0.000000	1.000000	0.00000	0.000000
50%	97198.540000	1.000000	1.00000	1.000000
75%	127644.240000	2.000000	1.00000	1.000000
max	250898.090000	4.000000	1.00000	1.000000

	EstimatedSalary	Exited
--	-----------------	--------

count	10000.000000	10000.000000
mean	100090.239881	0.203700
std	57510.492818	0.402769
min	11.580000	0.000000
25%	51002.110000	0.000000
50%	100193.915000	0.000000
75%	149388.247500	0.000000
max	199992.480000	1.000000

```
[7]: df.isnull().sum()
```

```
[7]: RowNumber      0
     CustomerId    0
     Surname        0
     CreditScore    0
     Geography      0
     Gender         0
     Age            0
     Tenure         0
     Balance        0
     NumOfProducts  0
     HasCrCard      0
     IsActiveMember 0
     EstimatedSalary 0
     Exited         0
     dtype: int64
```

```
[8]: df.isnull().any().any()
```

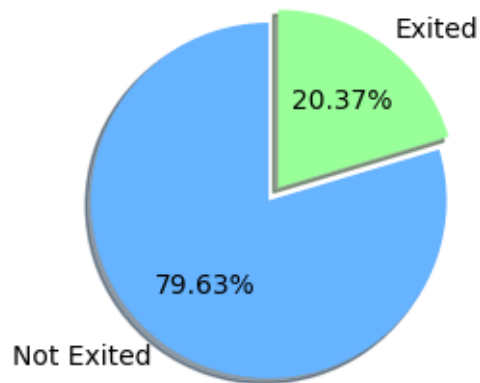
```
[8]: False
```

```
[9]: df.columns
```

```
[9]: Index(['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography',
          'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard',
          'IsActiveMember', 'EstimatedSalary', 'Exited'],
          dtype='object')
```

```
[10]: values = df['Exited'].value_counts()
      labels = ['Not Exited', 'Exited']
      colors = ['#66b3ff', '#99ff99']

      fig, ax = plt.subplots(figsize=(4, 3), dpi=100)
      ax.pie(values, labels=labels, autopct='%1.2f%%', startangle=90, explode=(0, 0.
      ↪09), colors=colors, shadow=True)
      plt.show()
```

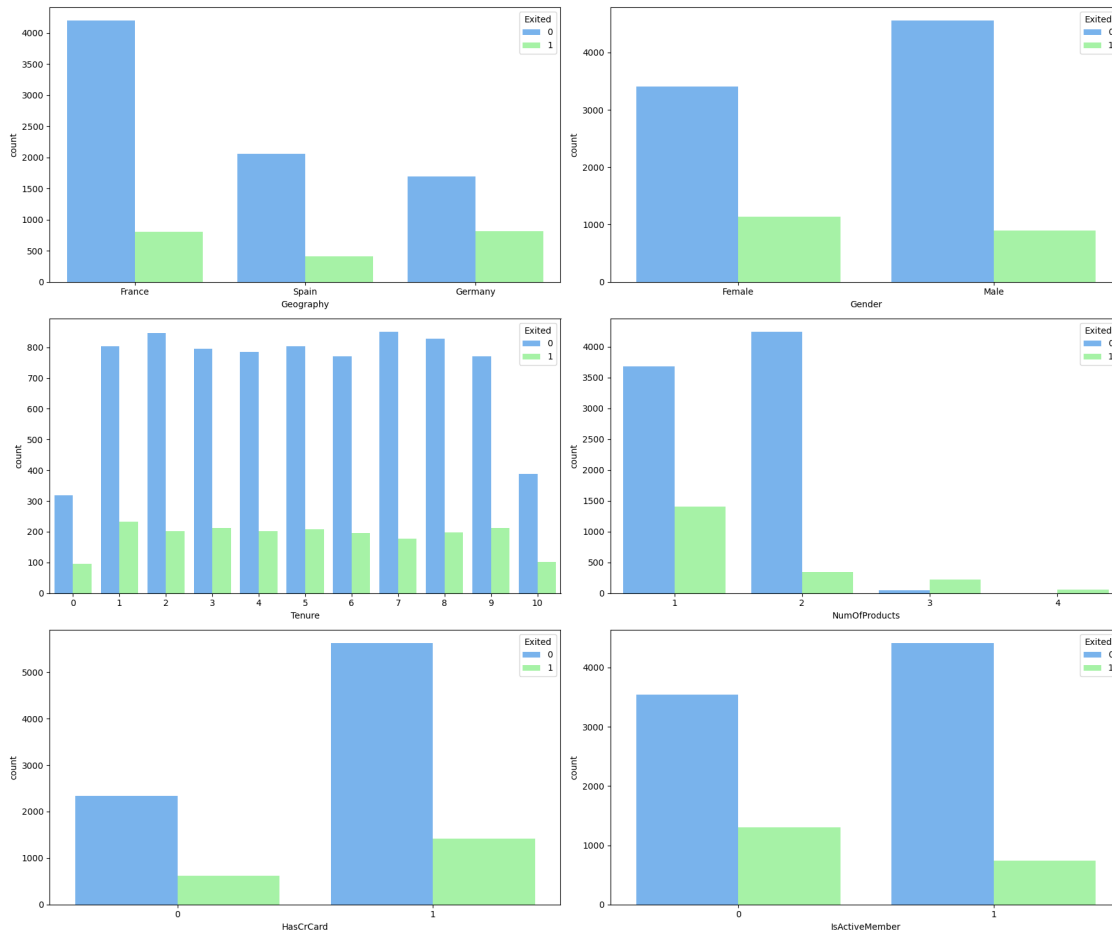


20% of the customers have churned and 80% haven't.

```
[11]: # visualizing categorical variables
fig, ax = plt.subplots(3, 2, figsize=(18, 15))
colors = ['#66b3ff', '#99ff99']

sns.countplot(x='Geography', hue='Exited', data=df, ax=ax[0][0], palette=colors)
sns.countplot(x='Gender', hue='Exited', data=df, ax=ax[0][1], palette=colors)
sns.countplot(x='Tenure', hue='Exited', data=df, ax=ax[1][0], palette=colors)
sns.countplot(x='NumOfProducts', hue='Exited', data=df, ax=ax[1][1],
    ↪palette=colors)
sns.countplot(x='HasCrCard', hue='Exited', data=df, ax=ax[2][0], palette=colors)
sns.countplot(x='IsActiveMember', hue='Exited', data=df, ax=ax[2][1],
    ↪palette=colors)

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



### 0.0.1 Based on the plots above, we can determine that:-

The bulk of the clients are from France, but the majority of those that have left are from Germany, possibly as a result of resource shortages brought on by the small client base.

In addition, a higher percentage of female customers are leaving than male customers.

The majority of clients have tenures ranging from 1 to 9, and the rate of attrition is high in between.

The majority of customers own one or two products, and the majority of those that have churned have only one product—possibly because they were dissatisfied and wanted something else.

It's interesting to note that most of the customers who churned had credit cards, though this could just be a coincidence given that most customers do.

It should come as no surprise that there is a higher turnover rate among inactive members, with a relatively high percentage overall.

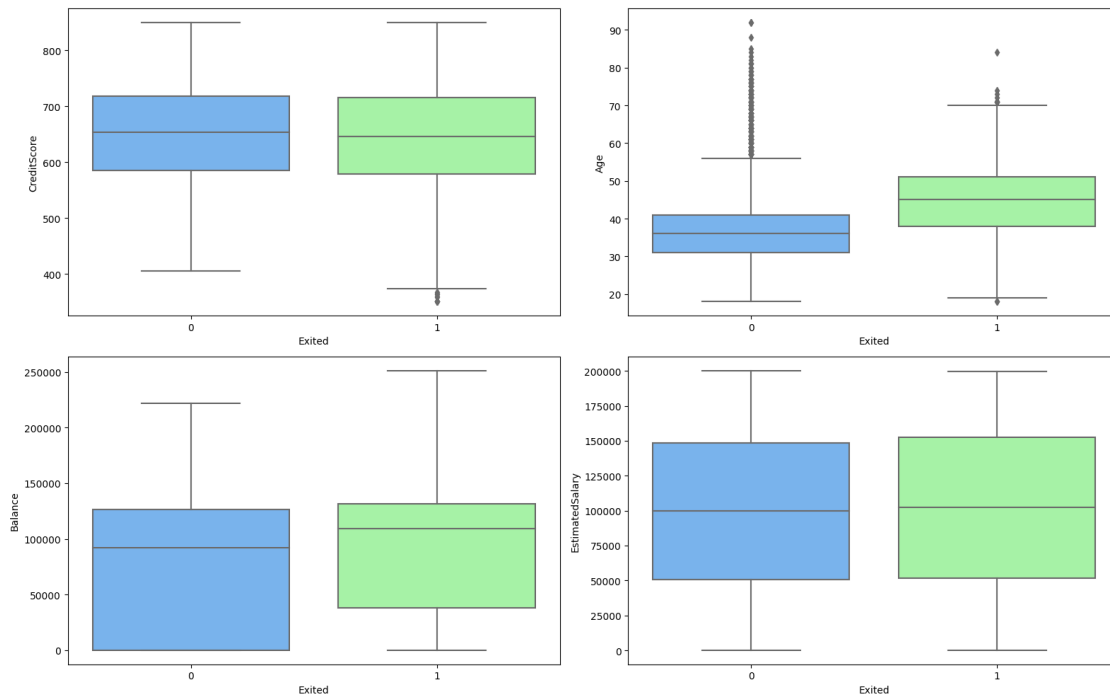
```
[12]: fig, ax = plt.subplots(2, 2, figsize=(16, 10))
      colors = ['#66b3ff', '#99ff99']
```

```

sns.boxplot(x='Exited', y='CreditScore', data=df, ax=ax[0][0], palette=colors)
sns.boxplot(x='Exited', y='Age', data=df, ax=ax[0][1], palette=colors)
sns.boxplot(x='Exited', y='Balance', data=df, ax=ax[1][0], palette=colors)
sns.boxplot(x='Exited', y='EstimatedSalary', data=df, ax=ax[1][1],
            palette=colors)

plt.tight_layout()
plt.show()

```



## 0.0.2 Based on the plots above, we can determine that:

The distribution of credit scores does not significantly differ between customers who experience churning and those who do not.

More elderly clients are leaving than new ones.

The bank is losing clients who have sizable balances.

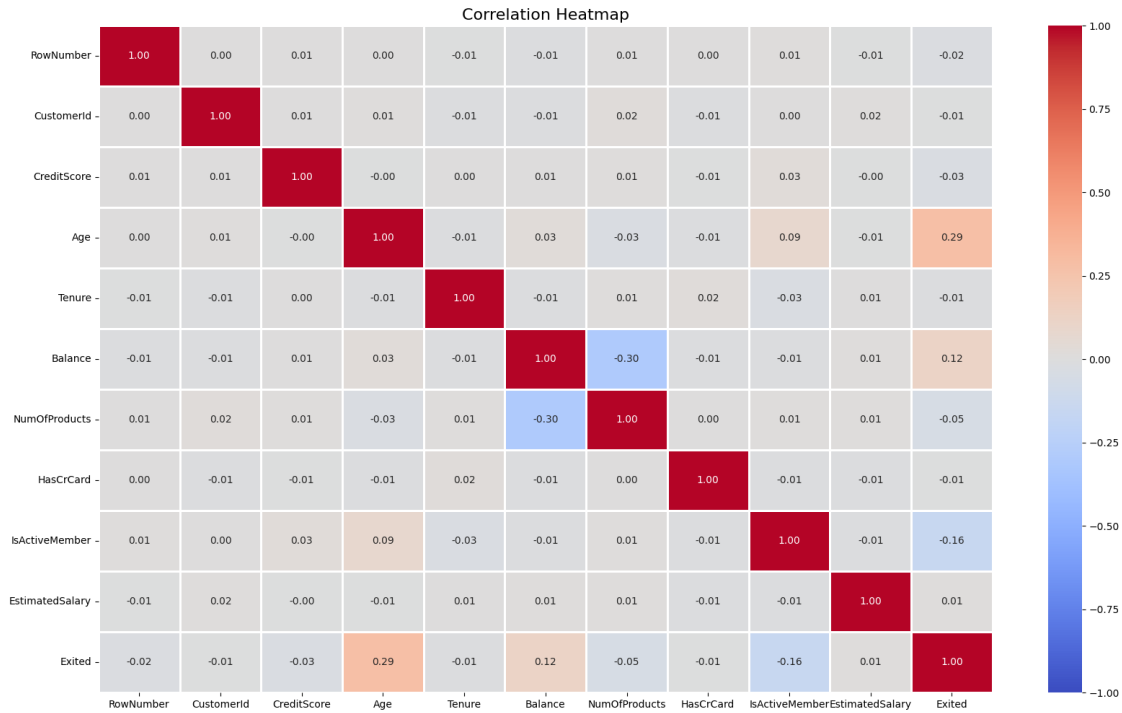
The chance of churn is not significantly impacted by estimated salary.

It's interesting to note that most of the customers who churned had credit cards, though this could just be a coincidence given that most customers do.

It should come as no surprise that there is a higher turnover rate among inactive members, with a relatively high percentage overall.

```
[13]: #Heatmap
numeric_df = df.select_dtypes(include=['float64', 'int64'])
plt.figure(figsize=(20, 12))
corr = numeric_df.corr()
sns.heatmap(corr, linewidths=1, annot=True, fmt=".2f", cmap='coolwarm',
            vmin=-1, vmax=1)

plt.title('Correlation Heatmap', fontsize=16)
plt.show()
```



```
[14]: # dropping useless columns
df.drop(columns = ['RowNumber', 'CustomerId', 'Surname'], axis = 1, inplace =
        True)
df.head()
```

```
[14]:
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	\
0	619	France	Female	42	2	0.00	1	
1	608	Spain	Female	41	1	83807.86	1	
2	502	France	Female	42	8	159660.80	3	
3	699	France	Female	39	1	0.00	2	
4	850	Spain	Female	43	2	125510.82	1	

	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	1	101348.88	1

1	0	1	112542.58	0
2	1	0	113931.57	1
3	0	0	93826.63	0
4	1	1	79084.10	0

```
[15]: df.Geography.value_counts()
```

```
[15]: Geography
France      5014
Germany     2509
Spain       2477
Name: count, dtype: int64
```

```
[16]: # Encoding categorical variables

df['Geography'] = df['Geography'].map({'France' : 0, 'Germany' : 1, 'Spain' : 2})
df['Gender'] = df['Gender'].map({'Male' : 0, 'Female' : 1})
```

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

```
[17]:
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	\
0	619	0	1	42	2	0.00	1	
1	608	2	1	41	1	83807.86	1	
2	502	0	1	42	8	159660.80	3	
3	699	0	1	39	1	0.00	2	
4	850	2	1	43	2	125510.82	1	

	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	1	101348.88	1
1	0	1	112542.58	0
2	1	0	113931.57	1
3	0	0	93826.63	0
4	1	1	79084.10	0

```
[18]: # creating features and label
from tensorflow.keras.utils import to_categorical

X = df.drop('Exited', axis = 1)
y = to_categorical(df.Exited)
```

```
2024-01-31 14:54:56.471769: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-01-31 14:54:56.471908: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
```



```
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-01-31 14:54:56.667096: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
```

```
[19]: # splitting data into training set and test set

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

```
[20]: # Scaling data

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
[21]: import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import BatchNormalization

# initializing ann
model = Sequential()

# adding the first input layer and the first hidden layer
model.add(Dense(10, kernel_initializer = 'normal', activation = 'relu',
    ↪ input_shape = (10, )))

# adding batch normalization and dropout layer
model.add(Dropout(rate = 0.1))
model.add(BatchNormalization())

# adding the third hidden layer
model.add(Dense(7, kernel_initializer = 'normal', activation = 'relu'))

# adding batch normalization and dropout layer
model.add(Dropout(rate = 0.1))
model.add(BatchNormalization())

# adding the output layer
model.add(Dense(2, kernel_initializer = 'normal', activation = 'sigmoid'))

# compiling the model
```

```

model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
↳['accuracy'])

# fitting the model to the training set

model_history = model.fit(X_train, y_train, validation_split = 0.20,
↳validation_data = (X_test, y_test), epochs = 50)

```

```

Epoch 1/50
250/250 [=====] - 3s 4ms/step - loss: 0.6020 -
accuracy: 0.7237 - val_loss: 0.5272 - val_accuracy: 0.7990
Epoch 2/50
250/250 [=====] - 1s 3ms/step - loss: 0.4278 -
accuracy: 0.8234 - val_loss: 0.3874 - val_accuracy: 0.8460
Epoch 3/50
250/250 [=====] - 1s 3ms/step - loss: 0.3868 -
accuracy: 0.8389 - val_loss: 0.3609 - val_accuracy: 0.8515
Epoch 4/50
250/250 [=====] - 1s 3ms/step - loss: 0.3805 -
accuracy: 0.8421 - val_loss: 0.3608 - val_accuracy: 0.8500
Epoch 5/50
250/250 [=====] - 1s 3ms/step - loss: 0.3764 -
accuracy: 0.8438 - val_loss: 0.3593 - val_accuracy: 0.8520
Epoch 6/50
250/250 [=====] - 1s 3ms/step - loss: 0.3804 -
accuracy: 0.8380 - val_loss: 0.3596 - val_accuracy: 0.8495
Epoch 7/50
250/250 [=====] - 1s 3ms/step - loss: 0.3713 -
accuracy: 0.8476 - val_loss: 0.3620 - val_accuracy: 0.8490
Epoch 8/50
250/250 [=====] - 1s 3ms/step - loss: 0.3726 -
accuracy: 0.8469 - val_loss: 0.3636 - val_accuracy: 0.8510
Epoch 9/50
250/250 [=====] - 1s 4ms/step - loss: 0.3722 -
accuracy: 0.8434 - val_loss: 0.3598 - val_accuracy: 0.8485
Epoch 10/50
250/250 [=====] - 1s 4ms/step - loss: 0.3744 -
accuracy: 0.8487 - val_loss: 0.3607 - val_accuracy: 0.8500
Epoch 11/50
250/250 [=====] - 1s 3ms/step - loss: 0.3724 -
accuracy: 0.8464 - val_loss: 0.3586 - val_accuracy: 0.8490
Epoch 12/50
250/250 [=====] - 1s 4ms/step - loss: 0.3667 -
accuracy: 0.8489 - val_loss: 0.3619 - val_accuracy: 0.8490
Epoch 13/50
250/250 [=====] - 1s 3ms/step - loss: 0.3683 -
accuracy: 0.8457 - val_loss: 0.3577 - val_accuracy: 0.8450

```

Epoch 14/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3678 -  
accuracy: 0.8503 - val\_loss: 0.3598 - val\_accuracy: 0.8490  
Epoch 15/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3705 -  
accuracy: 0.8455 - val\_loss: 0.3615 - val\_accuracy: 0.8495  
Epoch 16/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3725 -  
accuracy: 0.8474 - val\_loss: 0.3628 - val\_accuracy: 0.8495  
Epoch 17/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3673 -  
accuracy: 0.8496 - val\_loss: 0.3615 - val\_accuracy: 0.8435  
Epoch 18/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3679 -  
accuracy: 0.8465 - val\_loss: 0.3572 - val\_accuracy: 0.8470  
Epoch 19/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3714 -  
accuracy: 0.8465 - val\_loss: 0.3597 - val\_accuracy: 0.8480  
Epoch 20/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3689 -  
accuracy: 0.8499 - val\_loss: 0.3568 - val\_accuracy: 0.8475  
Epoch 21/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3663 -  
accuracy: 0.8511 - val\_loss: 0.3560 - val\_accuracy: 0.8475  
Epoch 22/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3693 -  
accuracy: 0.8497 - val\_loss: 0.3556 - val\_accuracy: 0.8500  
Epoch 23/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3662 -  
accuracy: 0.8504 - val\_loss: 0.3559 - val\_accuracy: 0.8515  
Epoch 24/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3680 -  
accuracy: 0.8453 - val\_loss: 0.3592 - val\_accuracy: 0.8480  
Epoch 25/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3680 -  
accuracy: 0.8505 - val\_loss: 0.3559 - val\_accuracy: 0.8470  
Epoch 26/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3642 -  
accuracy: 0.8503 - val\_loss: 0.3571 - val\_accuracy: 0.8500  
Epoch 27/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3645 -  
accuracy: 0.8480 - val\_loss: 0.3593 - val\_accuracy: 0.8470  
Epoch 28/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3673 -  
accuracy: 0.8509 - val\_loss: 0.3565 - val\_accuracy: 0.8465  
Epoch 29/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3689 -  
accuracy: 0.8454 - val\_loss: 0.3601 - val\_accuracy: 0.8495

Epoch 30/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3616 -  
accuracy: 0.8551 - val\_loss: 0.3567 - val\_accuracy: 0.8510  
Epoch 31/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3639 -  
accuracy: 0.8497 - val\_loss: 0.3592 - val\_accuracy: 0.8470  
Epoch 32/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3645 -  
accuracy: 0.8485 - val\_loss: 0.3570 - val\_accuracy: 0.8485  
Epoch 33/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3647 -  
accuracy: 0.8509 - val\_loss: 0.3594 - val\_accuracy: 0.8500  
Epoch 34/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3622 -  
accuracy: 0.8514 - val\_loss: 0.3570 - val\_accuracy: 0.8505  
Epoch 35/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3683 -  
accuracy: 0.8497 - val\_loss: 0.3559 - val\_accuracy: 0.8495  
Epoch 36/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3634 -  
accuracy: 0.8503 - val\_loss: 0.3651 - val\_accuracy: 0.8465  
Epoch 37/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3637 -  
accuracy: 0.8551 - val\_loss: 0.3661 - val\_accuracy: 0.8475  
Epoch 38/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3660 -  
accuracy: 0.8511 - val\_loss: 0.3582 - val\_accuracy: 0.8480  
Epoch 39/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3637 -  
accuracy: 0.8526 - val\_loss: 0.3566 - val\_accuracy: 0.8510  
Epoch 40/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3642 -  
accuracy: 0.8536 - val\_loss: 0.3575 - val\_accuracy: 0.8485  
Epoch 41/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3614 -  
accuracy: 0.8514 - val\_loss: 0.3594 - val\_accuracy: 0.8455  
Epoch 42/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3646 -  
accuracy: 0.8489 - val\_loss: 0.3573 - val\_accuracy: 0.8465  
Epoch 43/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3596 -  
accuracy: 0.8533 - val\_loss: 0.3546 - val\_accuracy: 0.8460  
Epoch 44/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3676 -  
accuracy: 0.8495 - val\_loss: 0.3562 - val\_accuracy: 0.8460  
Epoch 45/50  
250/250 [=====] - 1s 3ms/step - loss: 0.3626 -  
accuracy: 0.8512 - val\_loss: 0.3549 - val\_accuracy: 0.8485

```
Epoch 46/50
250/250 [=====] - 1s 3ms/step - loss: 0.3630 -
accuracy: 0.8539 - val_loss: 0.3574 - val_accuracy: 0.8475
Epoch 47/50
250/250 [=====] - 1s 3ms/step - loss: 0.3623 -
accuracy: 0.8512 - val_loss: 0.3591 - val_accuracy: 0.8470
Epoch 48/50
250/250 [=====] - 1s 3ms/step - loss: 0.3608 -
accuracy: 0.8518 - val_loss: 0.3588 - val_accuracy: 0.8455
Epoch 49/50
250/250 [=====] - 1s 3ms/step - loss: 0.3661 -
accuracy: 0.8510 - val_loss: 0.3615 - val_accuracy: 0.8450
Epoch 50/50
250/250 [=====] - 1s 3ms/step - loss: 0.3653 -
accuracy: 0.8533 - val_loss: 0.3586 - val_accuracy: 0.8480
```

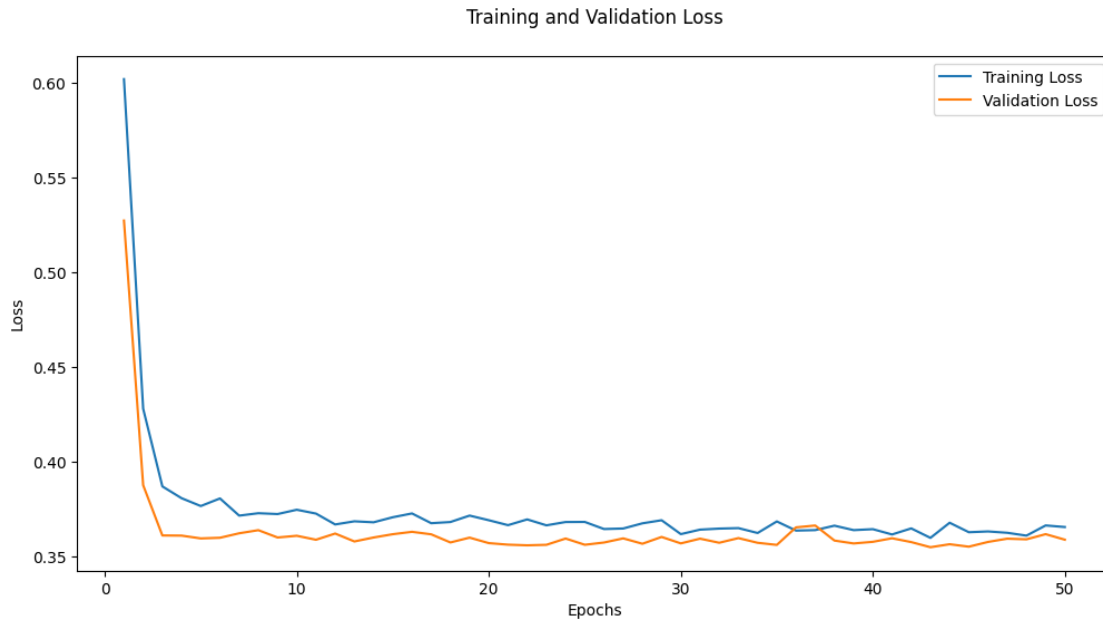
[27]: *#Visualizing Training and Validation Loss*

```
plt.figure(figsize=(12, 6))

train_loss = model_history.history['loss']
val_loss = model_history.history['val_loss']
epoch = range(1, 51)

sns.lineplot(x=epoch, y=train_loss, label='Training Loss')
sns.lineplot(x=epoch, y=val_loss, label='Validation Loss')

plt.title('Training and Validation Loss\n')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

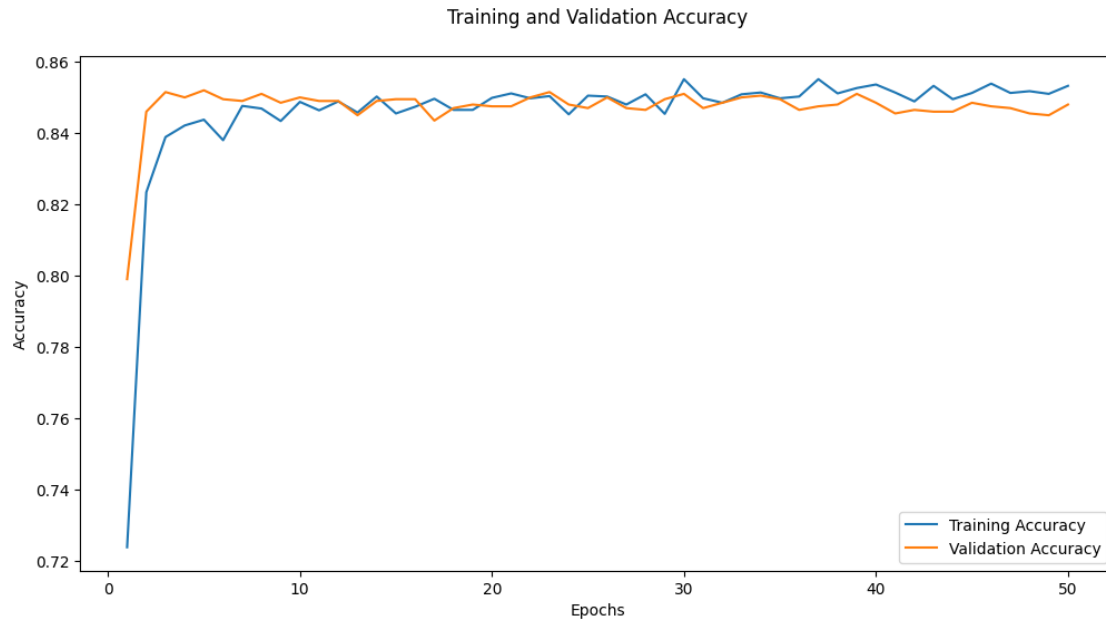


```
[30]: # Visualizing Training and Validation accuracy
plt.figure(figsize=(12, 6))

train_accuracy = model_history.history['accuracy']
val_accuracy = model_history.history['val_accuracy']
epoch = range(1, 51)

sns.lineplot(x=epoch, y=train_accuracy, label='Training Accuracy')
sns.lineplot(x=epoch, y=val_accuracy, label='Validation Accuracy')

plt.title('Training and Validation Accuracy\n')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



```
[32]: from sklearn.metrics import classification_report

y_pred = model.predict(X_test)

# Convert the one-hot encoded predictions to class labels
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Calculate and print the classification report
print(classification_report(y_test_classes, y_pred_classes))
```

```
63/63 [=====] - 0s 2ms/step
```

	precision	recall	f1-score	support
0	0.86	0.97	0.91	1598
1	0.76	0.36	0.48	402
accuracy			0.85	2000
macro avg	0.81	0.66	0.70	2000
weighted avg	0.84	0.85	0.83	2000