Importing necessary Libraries and loading dataset

```
import pandas as pd
import numpy as np
```

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("barelydedicated/bank-customer-churn-modeling")

print("Path to dataset files:", path)
```

```
Downloading from https://www.kaggle.com/api/v1/datasets/download/barelydedicated/bank-customer-churn-modeling?dataset_version_number=1...
100%|██████████| 262k/262k [00:00<00:00, 59.0MB/s]Extracting files...
Path to dataset files: /root/.cache/kagglehub/datasets/barelydedicated/bank-customer-churn-modeling/versions/1
```

```
import os
print(os.listdir(path))
```

```
['Churn_Modelling.csv']
```

```
df = pd.read_csv(os.path.join(path, "Churn_Modelling.csv"))
```

```
df.head(2)
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1	101348.88	1
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0

Next steps: Generate code with df New interactive sheet

Data Preprocessing

```
df.drop(columns=["RowNumber", "CustomerId", "Surname"], inplace=True)
```

```
df.head()
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	619	France	Female	42	2	0.00	1	1	1	101348.88	1
1	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
2	502	France	Female	42	8	159660.80	3	1	0	113931.57	1
3	699	France	Female	39	1	0.00	2	0	0	93826.63	0
4	850	Spain	Female	43	2	125510.82	1	1	1	79084.10	0

Next steps: Generate code with df | New interactive sheet

```
df.shape
```

```
(10000, 11)
```

```
df.groupby('Geography').count()
```

Geography	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
France	5014	5014	5014	5014	5014	5014	5014	5014	5014	5014
Germany	2509	2509	2509	2509	2509	2509	2509	2509	2509	2509
Spain	2477	2477	2477	2477	2477	2477	2477	2477	2477	2477

```
df.describe()
```

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
count	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.00000	10000.000000	10000.000000	10000.000000
mean	650.528800	38.921800	5.012800	76485.889288	1.530200	0.70550	0.515100	100090.239881	0.203700
std	96.653299	10.487806	2.892174	62397.405202	0.581654	0.45584	0.499797	57510.492818	0.402769
min	350.000000	18.000000	0.000000	0.000000	1.000000	0.00000	0.000000	11.580000	0.000000
25%	584.000000	32.000000	3.000000	0.000000	1.000000	0.00000	0.000000	51002.110000	0.000000
50%	652.000000	37.000000	5.000000	97198.540000	1.000000	1.00000	1.000000	100193.915000	0.000000
75%	718.000000	44.000000	7.000000	127644.240000	2.000000	1.00000	1.000000	149388.247500	0.000000
max	850.000000	92.000000	10.000000	250898.090000	4.000000	1.00000	1.000000	199992.480000	1.000000

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   CreditScore      10000 non-null  int64  
 1   Geography        10000 non-null  object 
 2   Gender           10000 non-null  object 
 3   Age              10000 non-null  int64  
 4   Tenure           10000 non-null  int64  
 5   Balance          10000 non-null  float64
 6   NumOfProducts    10000 non-null  int64  
 7   HasCrCard        10000 non-null  int64  
 8   IsActiveMember   10000 non-null  int64  
 9   EstimatedSalary  10000 non-null  float64
 10  Exited           10000 non-null  int64  
dtypes: float64(2), int64(7), object(2)
memory usage: 859.5+ KB
```

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df["Geography"] = le.fit_transform(df["Geography"])
df["Gender"] = le.fit_transform(df["Gender"])
```

```
df.head(2)
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	619	0	0	42	2	0.00	1	1	1	101348.88	1
1	608	2	0	41	1	83807.86	1	0	1	112542.58	0

Next steps: Generate code with df New interactive sheet

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```
df['Balance'] = scaler.fit_transform(df[['Balance']])
df['EstimatedSalary'] = scaler.fit_transform(df[['EstimatedSalary']])
```

```
df.head(2)
```

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	619	0	0	42	2	0.000000	1	1	1	0.506735	1
1	608	2	0	41	1	0.334031	1	0	1	0.562709	0

Next steps: Generate code with df New interactive sheet

Splitting of data

```
X = df.drop(columns=["Exited"])
y = df["Exited"]
```

```
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, random_state=42)
```

Training ANN

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

```
model = Sequential()
```

```
model.add(Dense(units=16, activation='relu', input_shape=(X_train.shape[1],)))
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential model
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
model.add(Dense(units=8, activation='relu'))
```

```
model.add(Dense(units=1, activation='sigmoid'))
```

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

```
model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2)
```

```
Epoch 1/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.7662 - loss: 2.7050 - val_accuracy: 0.7987 - val_loss: 0.5234
Epoch 2/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7973 - loss: 0.5172 - val_accuracy: 0.7981 - val_loss: 0.4906
Epoch 3/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7898 - loss: 0.5081 - val_accuracy: 0.7981 - val_loss: 0.4852
Epoch 4/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7911 - loss: 0.5008 - val_accuracy: 0.7981 - val_loss: 0.4693
Epoch 5/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7898 - loss: 0.4859 - val_accuracy: 0.7981 - val_loss: 0.4677
Epoch 6/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7938 - loss: 0.4867 - val_accuracy: 0.8012 - val_loss: 0.4607
Epoch 7/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7996 - loss: 0.4745 - val_accuracy: 0.8062 - val_loss: 0.4701
Epoch 8/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7981 - loss: 0.4706 - val_accuracy: 0.7969 - val_loss: 0.4549
Epoch 9/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7873 - loss: 0.4885 - val_accuracy: 0.7912 - val_loss: 0.4581
Epoch 10/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7902 - loss: 0.4787 - val_accuracy: 0.7987 - val_loss: 0.4574
Epoch 11/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7975 - loss: 0.4729 - val_accuracy: 0.7987 - val_loss: 0.4633
Epoch 12/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7973 - loss: 0.4677 - val_accuracy: 0.8037 - val_loss: 0.4441
Epoch 13/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7910 - loss: 0.4673 - val_accuracy: 0.7987 - val_loss: 0.4684
Epoch 14/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8016 - loss: 0.4628 - val_accuracy: 0.8125 - val_loss: 0.4412
Epoch 15/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7978 - loss: 0.4622 - val_accuracy: 0.8112 - val_loss: 0.4457
Epoch 16/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8120 - loss: 0.4518 - val_accuracy: 0.7987 - val_loss: 0.4520
Epoch 17/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7966 - loss: 0.4714 - val_accuracy: 0.8094 - val_loss: 0.4387
Epoch 18/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8015 - loss: 0.4628 - val_accuracy: 0.7981 - val_loss: 0.4547
Epoch 19/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7950 - loss: 0.4698 - val_accuracy: 0.8112 - val_loss: 0.4364
Epoch 20/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8063 - loss: 0.4592 - val_accuracy: 0.8156 - val_loss: 0.4325
Epoch 21/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8106 - loss: 0.4477 - val_accuracy: 0.8131 - val_loss: 0.4346
Epoch 22/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7894 - loss: 0.4708 - val_accuracy: 0.8112 - val_loss: 0.4403
```

```
Epoch 23/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8055 - loss: 0.4538 - val_accuracy: 0.8156 - val_loss: 0.4305
Epoch 24/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8028 - loss: 0.4535 - val_accuracy: 0.8081 - val_loss: 0.4405
Epoch 25/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8077 - loss: 0.4550 - val_accuracy: 0.8188 - val_loss: 0.4396
Epoch 26/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7995 - loss: 0.4656 - val_accuracy: 0.8188 - val_loss: 0.4318
Epoch 27/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8022 - loss: 0.4567 - val_accuracy: 0.8138 - val_loss: 0.4381
Epoch 28/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8043 - loss: 0.4563 - val_accuracy: 0.7919 - val_loss: 0.4689
Epoch 29/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7986 - loss: 0.4639 - val_accuracy: 0.8169 - val_loss: 0.4271
```

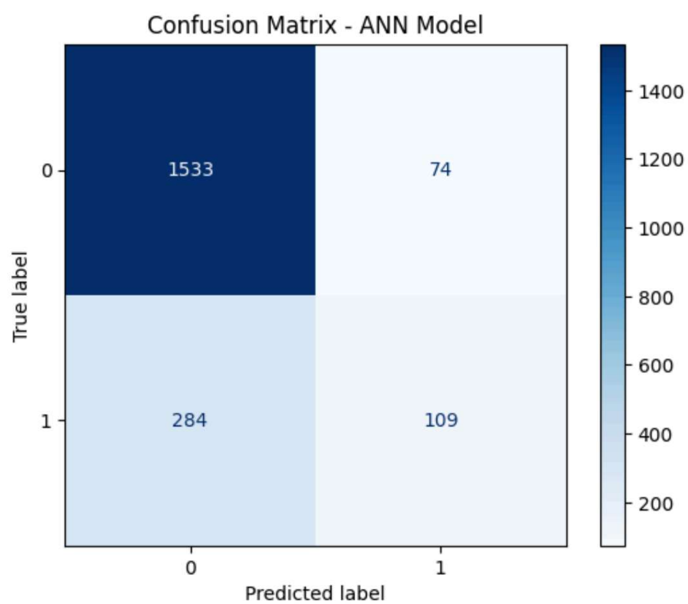
```
y_pred = (model.predict(X_test) > 0.5)
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```

Classification Report and Accuracy of Model

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix - ANN Model")
plt.show()
```



```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.821
```

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.84      0.95      0.90      1607
           1       0.60      0.28      0.38       393

    accuracy                           0.82      2000
   macro avg       0.72      0.62      0.64      2000
weighted avg       0.79      0.82      0.79      2000
```

Start coding or generate with AI.