```
# Step 1: Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import confusion_matrix, accuracy_score
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
import matplotlib.pyplot as plt
```

```
# Step 2: Load the dataset
dataset = pd.read_csv('Churn_Modelling.csv')

# Check the first few rows of the dataset
dataset.head()
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1
2	3	15619304	Onio	502	France	Female	42	8	159660.80	3	1	0
3	4	15701354	Boni	699	France	Female	39	1	0.00	2	0	0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	1	1	1

Next steps: Generate code with dataset | View recommended plots | New interactive sheet

```
# Step 3: Distinguish the feature and target set
# We are using all features except CustomerId, Surname, and RowNumber as they are irrelevant
X = dataset.iloc[:, 3:-1]  # Exclude CustomerId, Surname, and RowNumber
y = dataset.iloc[:, -1].values  # Target column

# Step 4: Encode categorical data (Geography, Gender)
# Label encode the "Gender" column (binary)
labelencoder_gender = LabelEncoder()
X['Gender'] = labelencoder_gender.fit_transform(X['Gender'])

# One-hot encode the "Geography" column (multi-class)
X = pd.get_dummies(X, columns=['Geography'], drop_first=True)  # Convert Geography to one-hot encoding

# Check the transformed features
X.head()
```

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

Next steps: Generate code with X | View recommended plots | New interactive sheet

```
# Step 5: Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Check the shape of the training and test data
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((8000, 11), (2000, 11), (8000,), (2000,))
```

```
# Step 6: Normalize the training and test data
sc = StandardScaler()
X_train = sc.fit_transform(X_train)  # Normalize training data
X_test = sc.transform(X_test)  # Normalize test data
```

```
# Step 7: Build the Neural Network model
model = Sequential()

# Input layer and first hidden layer with Dropout to prevent overfitting
model.add(Dense(units=16, activation='relu', input_dim=X_train.shape[1]))
model.add(Dropout(0.3))

# Second hidden layer
model.add(Dense(units=16, activation='relu'))
model.add(Dropout(0.3))

# Output layer with sigmoid activation for binary classification
model.add(Dense(units=1, activation='sigmoid'))

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

# Print model summary
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` arg
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 16)	192
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 16)	272
dropout_1 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 1)	17

Total params: 481 (1.88 KB)
Trainable params: 481 (1.88 KB)

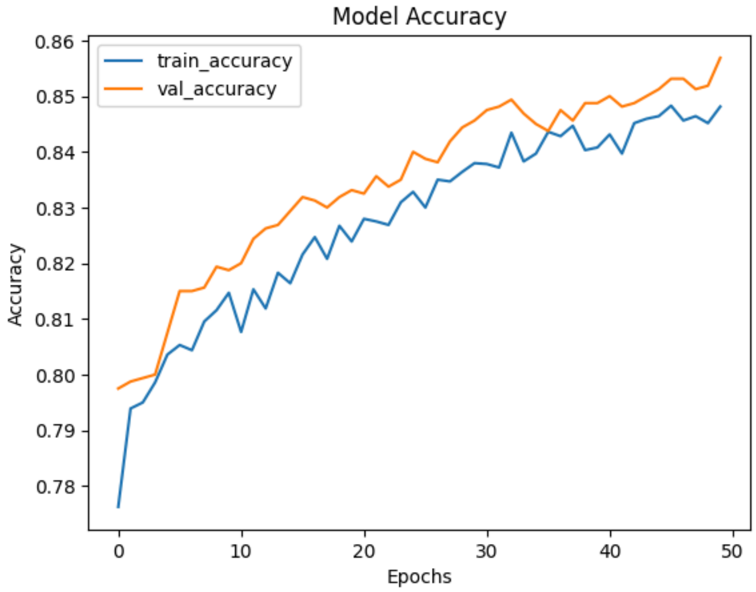
```
# Step 8: Train the model
history = model.fit(X_train, y_train, batch_size=32, epochs=50, validation_split=0.2)
```

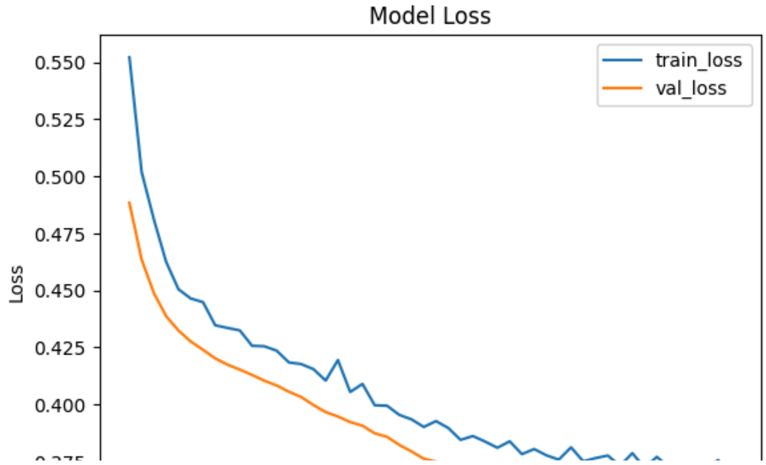
```
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8397 - loss: 0.3824 - val_accuracy: 0.8481 - val_loss: 0.3599
Epoch 43/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8432 - loss: 0.3771 - val_accuracy: 0.8487 - val_loss: 0.3587
Epoch 44/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8478 - loss: 0.3701 - val_accuracy: 0.8500 - val_loss: 0.3596
Epoch 45/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8457 - loss: 0.3781 - val_accuracy: 0.8512 - val_loss: 0.3582
Epoch 46/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8497 - loss: 0.3621 - val_accuracy: 0.8531 - val_loss: 0.3558
Epoch 47/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8491 - loss: 0.3745 - val_accuracy: 0.8531 - val_loss: 0.3570
Epoch 48/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8430 - loss: 0.3752 - val_accuracy: 0.8512 - val_loss: 0.3582
Epoch 49/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8414 - loss: 0.3803 - val_accuracy: 0.8519 - val_loss: 0.3577
Epoch 50/50
200/200 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8446 - loss: 0.3694 - val_accuracy: 0.8569 - val_loss: 0.3559
```

```python
# Step 9: Evaluate the model and print accuracy score
y_pred = (model.predict(X_test) > 0.5).astype(int)

# Confusion matrix and accuracy
cm = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy * 100:.2f}%')
print('Confusion Matrix:')
print(cm)
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Accuracy: 86.40%
Confusion Matrix:
[[1543   52]
 [ 220  185]]
```

```python
# Plot the model's accuracy and loss curves
plt.plot(history.history['accuracy'], label='train_accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```


Model Accuracy
train_accuracy
val_accuracy
Accuracy
Epochs
0.86
0.85
0.84
0.83
0.82
0.81
0.80
0.79
0.78
0
10
20
30
40
50


Model Loss
train_loss
val_loss
Loss
0.550
0.525
0.500
0.475
0.450
0.425
0.400