CODE 1

```
import pandas as pd

# Load the dataset
df = pd.read_csv('path_to_your_downloaded/bank-customer-churn-modeling.csv')

# Display the first few rows of the dataset
df.head()

from sklearn.model_selection import train_test_split

# Features: Drop the target column and any other unnecessary columns like 'CustomerId',
'Surname'
X = df.drop(['Exited', 'CustomerId', 'Surname'], axis=1)

# Target: 'Exited' column represents customer churn (1 = will leave, 0 = won't leave)
y = df['Exited']

# Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Normalize the training data
X_train_scaled = scaler.fit_transform(X_train)

# Normalize the testing data (using the same scaler from training data)
X_test_scaled = scaler.transform(X_test)

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

# Initialize the model
model = Sequential()

# Add the input layer (input_dim=number of features)
model.add(Dense(units=64, activation='relu', input_dim=X_train_scaled.shape[1]))

# Add one hidden layer
```

```
model.add(Dense(units=64, activation='relu'))

# Add the output layer (binary classification: sigmoid activation)
model.add(Dense(units=1, activation='sigmoid'))

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

# Print model summary
model.summary()

# Train the model
history = model.fit(X_train_scaled, y_train, epochs=10, batch_size=32,
validation_data=(X_test_scaled, y_test))

# Evaluate the model on the test data
loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Test accuracy: {accuracy}")

from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Predict using the trained model
y_pred = (model.predict(X_test_scaled) > 0.5)  # Threshold at 0.5 for binary classification

# Accuracy score
from sklearn.metrics import accuracy_score
print(f"Accuracy Score: {accuracy_score(y_test, y_pred)}")

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

# Plotting confusion matrix
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Not Churned", "Churned"],
yticklabels=["Not Churned", "Churned"])
plt.title("Confusion Matrix")
plt.show()
```

CODE 2

```
#!/usr/bin/env python
# coding: utf-8
```

```
import heapq
# Node structure for Huffman Tree
class HuffmanNode:
def __init__(self, char, freq):
self.char = char
self.freq = freq
self.left = None
self.right = None
# Defining less than operator for priority queue comparison
def __lt__(self, other):
return self.freq < other.freq
# Function to generate Huffman codes
def generate_codes(root, current_code, codes):
if root is None:
return
if root.char is not None:
codes[root.char] = current_code
generate_codes(root.left, current_code + "0", codes)
generate_codes(root.right, current_code + "1", codes)
# Function to build Huffman Tree
def build_huffman_tree(frequency):
heap = []
# Insert all characters with their frequencies into the heap
for char, freq in frequency.items():
heapq.heappush(heap, HuffmanNode(char, freq))
# Merge nodes until we have one tree
while len(heap) > 1:
node1 = heapq.heappop(heap)
node2 = heapq.heappop(heap)
# Create a new internal node with the combined frequency
merged = HuffmanNode(None, node1.freq + node2.freq)
merged.left = node1
merged.right = node2
heapq.heappush(heap, merged)
# The root of the Huffman Tree
return heapq.heappop(heap)
# Function to calculate frequency of characters
def calculate_frequency(data):
frequency = {}
for char in data:
if char not in frequency:
frequency[char] = 0
frequency[char] += 1
return frequency
```

```
# Huffman Encoding process
def huffman_encoding(data):
frequency = calculate_frequency(data)
huffman_tree_root = build_huffman_tree(frequency)
codes = {}
generate_codes(huffman_tree_root, "", codes)
# Encode the input data
encoded_data = "".join([codes[char] for char in data])
return encoded_data, huffman_tree_root
# Huffman Decoding process
def huffman_decoding(encoded_data, huffman_tree_root):
decoded_data = ""
current_node = huffman_tree_root
for bit in encoded_data:
if bit == '0':
current_node = current_node.left
else:
current_node = current_node.right
if current_node.left is None and current_node.right is None:
decoded_data += current_node.char
current_node = huffman_tree_root
return decoded_data
# Driver code
if __name__ == "__main__":
data = input("Enter the string to be encoded using Huffman Encoding: ")
encoded_data, huffman_tree_root = huffman_encoding(data)
print(f"\nEncoded Data: {encoded_data}")
decoded_data = huffman_decoding(encoded_data, huffman_tree_root)
print(f"Decoded Data: {decoded_data}")
```