Neural Network Model Development and Evaluation Workflow

Introduction

This code presents a comprehensive workflow for developing and evaluating neural network models using TensorFlow and Keras. The primary steps include data preparation, model training with variations to handle underfitting and overfitting, regularization techniques, and thorough evaluation. The workflow is designed to provide insights into model performance, facilitate model interpretation, and guide the selection of an appropriate model for deployment.

Table of Contents

Step 13: Hyperparameter Tuning (Optional)

Step 14: Interpret Models

Step 15: Make Predictions

Step 16: Visualize Results

Step 17: Interpret Confusion Matrices

Step 18: Evaluate Additional Metrics

Step 19: Visualize ROC Curves

Step 20: Interpret ROC Curves

Final Steps and Conclusion

Step 21: Model Interpretation, Feature Importance (if applicable), and Model Saving

Step 1: Import Libraries

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import rcParams
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import plotly.express as px
import plotly.graph_objects as go
import warnings

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

warnings.filterwarnings(action='ignore')
```

Step 2: Load Dataset and Display Overview

In [2]:
```
# Load the diabetes dataset
diabetes_df = pd.read_csv("diabetes.csv")

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

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Diabetes
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

Step 3: Rename Column and Style Rows

In [3]:

```
# Rename the "DiabetesPedigreeFunction" column to "DPF"
diabetes_df.rename(columns={"DiabetesPedigreeFunction": "DPF"}, in

# Style the first five rows with modified background and text colo
styled_df = diabetes_df.head().T.style.set_properties(**{'backgrou
                                                          'color'
                                                          'border

styled_df
```

Out[3]:

	0	1	2	3	4
Pregnancies	6.000000	1.000000	8.000000	1.000000	0.000000
Glucose	148.000000	85.000000	183.000000	89.000000	137.000000
BloodPressure	72.000000	66.000000	64.000000	66.000000	40.000000
SkinThickness	35.000000	29.000000	0.000000	23.000000	35.000000
Insulin	0.000000	0.000000	0.000000	94.000000	168.000000
BMI	33.600000	26.600000	23.300000	28.100000	43.100000
DPF	0.627000	0.351000	0.672000	0.167000	2.288000
Age	50.000000	31.000000	32.000000	21.000000	33.000000
Outcome	1.000000	0.000000	1.000000	0.000000	1.000000

Step 4: Calculate Missing Values

In [4]:

```
# Calculate the number of missing values in each column
missing_values_count = diabetes_df.isnull().sum()

# Display the result
print("Number of missing values in each column:")
print(missing_values_count)
```

```
Number of missing values in each column:
Pregnancies      0
Glucose          0
BloodPressure    0
SkinThickness    0
Insulin          0
BMI              0
DPF              0
Age              0
Outcome          0
dtype: int64
```

Step 5: Split Data and Handle Missing Values

In [5]:

```
# Split the data into features (X) and target (y)
X = diabetes_df.drop('Outcome', axis=1)
y = diabetes_df['Outcome']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_siz

# Handle missing values by replacing zeros with the mean
zero_features = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinT
total_count = diabetes_df['Glucose'].count()

for feature in zero_features:
    zero_count = diabetes_df[diabetes_df[feature] == 0][feature].c
    percent_zero = 100 * zero_count / total_count
    print(f'{feature}: 0 number of cases {zero_count}, percentage 

# Calculate the mean excluding zeros
diabetes_mean = diabetes_df[zero_features].replace(0, np.nan).mean

# Replace zeros with the mean
diabetes_df[zero_features] = diabetes_df[zero_features].replace(0,
```

```
Pregnancies: 0 number of cases 111, percentage is 14.45%
Glucose: 0 number of cases 5, percentage is 0.65%
BloodPressure: 0 number of cases 35, percentage is 4.56%
SkinThickness: 0 number of cases 227, percentage is 29.56%
Insulin: 0 number of cases 374, percentage is 48.70%
BMI: 0 number of cases 11, percentage is 1.43%
```

Step 6: Scale Features

In [6]:

```
# Scale features using StandardScaler
scaler = StandardScaler()
```

```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Step 7: Train Underfit Model (Few Layers and Epochs)

In [7]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define a simple neural network (simulating underfitting)
underfit_model = Sequential([
    Dense(8, activation='relu', input_shape=(X_train_scaled.shape[
    Dense(1, activation='sigmoid')
])

underfit_model.compile(optimizer='adam', loss='binary_crossentropy

# Train the model with a small number of epochs
history = underfit_model.fit(X_train_scaled, y_train, epochs=10, b
```

```
Epoch 1/10
18/18 [==============================] - 1s 22ms/step - loss: 0.66
69 - accuracy: 0.6337 - val_loss: 0.7080 - val_accuracy: 0.6042
Epoch 2/10
18/18 [==============================] - 0s 8ms/step - loss: 0.634
9 - accuracy: 0.6580 - val_loss: 0.6857 - val_accuracy: 0.6094
Epoch 3/10
18/18 [==============================] - 0s 7ms/step - loss: 0.607
0 - accuracy: 0.6823 - val_loss: 0.6690 - val_accuracy: 0.6094
Epoch 4/10
18/18 [==============================] - 0s 7ms/step - loss: 0.586
6 - accuracy: 0.6997 - val_loss: 0.6552 - val_accuracy: 0.6094
Epoch 5/10
18/18 [==============================] - 0s 8ms/step - loss: 0.569
3 - accuracy: 0.7153 - val_loss: 0.6438 - val_accuracy: 0.6406
Epoch 6/10
18/18 [==============================] - 0s 8ms/step - loss: 0.555
1 - accuracy: 0.7257 - val_loss: 0.6333 - val_accuracy: 0.6458
Epoch 7/10
18/18 [==============================] - 0s 7ms/step - loss: 0.542
7 - accuracy: 0.7326 - val_loss: 0.6257 - val_accuracy: 0.6510
Epoch 8/10
18/18 [==============================] - 0s 7ms/step - loss: 0.533
1 - accuracy: 0.7344 - val_loss: 0.6183 - val_accuracy: 0.6562
Epoch 9/10
18/18 [==============================] - 0s 6ms/step - loss: 0.524
4 - accuracy: 0.7326 - val_loss: 0.6117 - val_accuracy: 0.6771
Epoch 10/10
18/18 [==============================] - 0s 7ms/step - loss: 0.516
8 - accuracy: 0.7465 - val_loss: 0.6056 - val_accuracy: 0.6771
```

Step 8: Train Overfit Model (More Layers and Epochs)

In [8]:

```
# Define a more complex neural network (simulating overfitting)
overfit_model = Sequential([
    Dense(32, activation='relu', input_shape=(X_train_scaled.shape
    Dense(64, activation='relu'),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])

overfit_model.compile(optimizer='adam', loss='binary_crossentropy'

# Train the model with a larger number of epochs
overfit_model.fit(X_train_scaled, y_train, epochs=50, batch_size=3
```

```
Epoch 1/50
18/18 [==============================] - 2s 20ms/step - loss: 0.63
65 - accuracy: 0.6562 - val_loss: 0.6118 - val_accuracy: 0.6823
Epoch 2/50
18/18 [==============================] - 0s 8ms/step - loss: 0.539
9 - accuracy: 0.7483 - val_loss: 0.5522 - val_accuracy: 0.7812
Epoch 3/50
18/18 [==============================] - 0s 8ms/step - loss: 0.482
9 - accuracy: 0.7743 - val_loss: 0.5425 - val_accuracy: 0.7812
Epoch 4/50
18/18 [==============================] - 0s 6ms/step - loss: 0.453
6 - accuracy: 0.7917 - val_loss: 0.5489 - val_accuracy: 0.7708
Epoch 5/50
18/18 [==============================] - 0s 8ms/step - loss: 0.439
8 - accuracy: 0.7899 - val_loss: 0.5462 - val_accuracy: 0.7708
Epoch 6/50
18/18 [==============================] - 0s 8ms/step - loss: 0.428
8 - accuracy: 0.7847 - val_loss: 0.5486 - val_accuracy: 0.7552
Epoch 7/50
18/18 [==============================] - 0s 7ms/step - loss: 0.421
8 - accuracy: 0.7934 - val_loss: 0.5559 - val_accuracy: 0.7500
Epoch 8/50
18/18 [==============================] - 0s 7ms/step - loss: 0.415
0 - accuracy: 0.7899 - val_loss: 0.5671 - val_accuracy: 0.7552
Epoch 9/50
18/18 [==============================] - 0s 8ms/step - loss: 0.404
9 - accuracy: 0.8021 - val_loss: 0.5663 - val_accuracy: 0.7552
Epoch 10/50
18/18 [==============================] - 0s 8ms/step - loss: 0.400
6 - accuracy: 0.7917 - val_loss: 0.5695 - val_accuracy: 0.7500
Epoch 11/50
18/18 [==============================] - 0s 7ms/step - loss: 0.398
8 - accuracy: 0.8003 - val_loss: 0.5780 - val_accuracy: 0.7344
Epoch 12/50
18/18 [==============================] - 0s 7ms/step - loss: 0.386
5 - accuracy: 0.8073 - val_loss: 0.5915 - val_accuracy: 0.7344
Epoch 13/50
18/18 [==============================] - 0s 7ms/step - loss: 0.378
3 - accuracy: 0.8038 - val_loss: 0.5834 - val_accuracy: 0.7240
Epoch 14/50
18/18 [==============================] - 0s 8ms/step - loss: 0.372
6 - accuracy: 0.8177 - val_loss: 0.6199 - val_accuracy: 0.6927
Epoch 15/50
18/18 [==============================] - 0s 7ms/step - loss: 0.365
7 - accuracy: 0.8229 - val_loss: 0.6119 - val_accuracy: 0.7396
Epoch 16/50
18/18 [==============================] - 0s 7ms/step - loss: 0.368
6 - accuracy: 0.8177 - val_loss: 0.6188 - val_accuracy: 0.7135
Epoch 17/50
18/18 [==============================] - 0s 7ms/step - loss: 0.359
1 - accuracy: 0.8229 - val_loss: 0.6198 - val_accuracy: 0.7135
Epoch 18/50
18/18 [==============================] - 0s 7ms/step - loss: 0.353
```

```
0 - accuracy: 0.8333 - val_loss: 0.6338 - val_accuracy: 0.7031
Epoch 19/50
18/18 [==============================] - 0s 7ms/step - loss: 0.350
2 - accuracy: 0.8299 - val_loss: 0.6400 - val_accuracy: 0.6875
Epoch 20/50
18/18 [==============================] - 0s 7ms/step - loss: 0.342
1 - accuracy: 0.8385 - val_loss: 0.6517 - val_accuracy: 0.6979
Epoch 21/50
18/18 [==============================] - 0s 6ms/step - loss: 0.345
2 - accuracy: 0.8385 - val_loss: 0.6552 - val_accuracy: 0.6979
Epoch 22/50
18/18 [==============================] - 0s 7ms/step - loss: 0.330
5 - accuracy: 0.8368 - val_loss: 0.6629 - val_accuracy: 0.6875
Epoch 23/50
18/18 [==============================] - 0s 12ms/step - loss: 0.32
58 - accuracy: 0.8455 - val_loss: 0.6752 - val_accuracy: 0.7031
Epoch 24/50
18/18 [==============================] - 0s 8ms/step - loss: 0.316
6 - accuracy: 0.8490 - val_loss: 0.6864 - val_accuracy: 0.6875
Epoch 25/50
18/18 [==============================] - 0s 7ms/step - loss: 0.325
7 - accuracy: 0.8490 - val_loss: 0.6846 - val_accuracy: 0.6979
Epoch 26/50
18/18 [==============================] - 0s 6ms/step - loss: 0.317
8 - accuracy: 0.8420 - val_loss: 0.7090 - val_accuracy: 0.6927
Epoch 27/50
18/18 [==============================] - 0s 8ms/step - loss: 0.302
8 - accuracy: 0.8594 - val_loss: 0.7271 - val_accuracy: 0.7083
Epoch 28/50
18/18 [==============================] - 0s 6ms/step - loss: 0.302
0 - accuracy: 0.8507 - val_loss: 0.7195 - val_accuracy: 0.6823
Epoch 29/50
18/18 [==============================] - 0s 7ms/step - loss: 0.294
4 - accuracy: 0.8628 - val_loss: 0.7392 - val_accuracy: 0.7083
Epoch 30/50
18/18 [==============================] - 0s 7ms/step - loss: 0.298
2 - accuracy: 0.8559 - val_loss: 0.7550 - val_accuracy: 0.7031
Epoch 31/50
18/18 [==============================] - 0s 8ms/step - loss: 0.285
3 - accuracy: 0.8663 - val_loss: 0.7524 - val_accuracy: 0.7031
Epoch 32/50
18/18 [==============================] - 0s 7ms/step - loss: 0.280
1 - accuracy: 0.8733 - val_loss: 0.7746 - val_accuracy: 0.6979
Epoch 33/50
18/18 [==============================] - 0s 6ms/step - loss: 0.275
3 - accuracy: 0.8889 - val_loss: 0.8118 - val_accuracy: 0.7031
Epoch 34/50
18/18 [==============================] - 0s 8ms/step - loss: 0.271
9 - accuracy: 0.8733 - val_loss: 0.7924 - val_accuracy: 0.7031
Epoch 35/50
18/18 [==============================] - 0s 7ms/step - loss: 0.266
4 - accuracy: 0.8802 - val_loss: 0.7853 - val_accuracy: 0.7135
Epoch 36/50
```

```
18/18 [==============================] - 0s 7ms/step - loss: 0.263
0 - accuracy: 0.8889 - val_loss: 0.8749 - val_accuracy: 0.7083
Epoch 37/50
18/18 [==============================] - 0s 7ms/step - loss: 0.256
5 - accuracy: 0.8837 - val_loss: 0.8268 - val_accuracy: 0.7031
Epoch 38/50
18/18 [==============================] - 0s 7ms/step - loss: 0.252
2 - accuracy: 0.8993 - val_loss: 0.8287 - val_accuracy: 0.6927
Epoch 39/50
18/18 [==============================] - 0s 8ms/step - loss: 0.241
8 - accuracy: 0.8924 - val_loss: 0.8635 - val_accuracy: 0.6979
Epoch 40/50
18/18 [==============================] - 0s 6ms/step - loss: 0.237
9 - accuracy: 0.9028 - val_loss: 0.8658 - val_accuracy: 0.6979
Epoch 41/50
18/18 [==============================] - 0s 8ms/step - loss: 0.239
4 - accuracy: 0.8924 - val_loss: 0.8894 - val_accuracy: 0.6771
Epoch 42/50
18/18 [==============================] - 0s 8ms/step - loss: 0.244
3 - accuracy: 0.8924 - val_loss: 0.9039 - val_accuracy: 0.6979
Epoch 43/50
18/18 [==============================] - 0s 5ms/step - loss: 0.225
3 - accuracy: 0.9045 - val_loss: 0.8914 - val_accuracy: 0.6771
Epoch 44/50
18/18 [==============================] - 0s 8ms/step - loss: 0.221
3 - accuracy: 0.8976 - val_loss: 0.9148 - val_accuracy: 0.6823
Epoch 45/50
18/18 [==============================] - 0s 8ms/step - loss: 0.223
0 - accuracy: 0.9062 - val_loss: 0.9529 - val_accuracy: 0.6667
Epoch 46/50
18/18 [==============================] - 0s 7ms/step - loss: 0.215
8 - accuracy: 0.9097 - val_loss: 0.9232 - val_accuracy: 0.6823
Epoch 47/50
18/18 [==============================] - 0s 8ms/step - loss: 0.215
2 - accuracy: 0.9115 - val_loss: 0.9378 - val_accuracy: 0.6875
Epoch 48/50
18/18 [==============================] - 0s 8ms/step - loss: 0.199
3 - accuracy: 0.9219 - val_loss: 0.9796 - val_accuracy: 0.6719
Epoch 49/50
18/18 [==============================] - 0s 5ms/step - loss: 0.192
9 - accuracy: 0.9288 - val_loss: 1.0248 - val_accuracy: 0.7240
Epoch 50/50
18/18 [==============================] - 0s 7ms/step - loss: 0.196
9 - accuracy: 0.9184 - val_loss: 0.9761 - val_accuracy: 0.6719
```

Out[8]:

```
<keras.callbacks.History at 0x21a98384dd0>
```

Step 9: Apply Regularization (to address overfitting)

In [9]:
```
# Define a neural network with regularization
from tensorflow.keras import regularizers

regularized_model = Sequential([
    Dense(32, activation='relu', input_shape=(X_train_scaled.shape
    Dense(64, activation='relu', kernel_regularizer=regularizers.l
    Dense(1, activation='sigmoid')
])

regularized_model.compile(optimizer='adam', loss='binary_crossentr

# Train the model with a moderate number of epochs
regularized_model.fit(X_train_scaled, y_train, epochs=20, batch_si
```

```
Epoch 1/20
18/18 [==============================] - 2s 21ms/step - loss: 1.17
70 - accuracy: 0.6927 - val_loss: 1.1270 - val_accuracy: 0.6927
Epoch 2/20
18/18 [==============================] - 0s 7ms/step - loss: 1.057
5 - accuracy: 0.7378 - val_loss: 1.0337 - val_accuracy: 0.7396
Epoch 3/20
18/18 [==============================] - 0s 8ms/step - loss: 0.964
9 - accuracy: 0.7604 - val_loss: 0.9611 - val_accuracy: 0.7500
Epoch 4/20
18/18 [==============================] - 0s 7ms/step - loss: 0.889
8 - accuracy: 0.7656 - val_loss: 0.9034 - val_accuracy: 0.7500
Epoch 5/20
18/18 [==============================] - 0s 7ms/step - loss: 0.832
3 - accuracy: 0.7726 - val_loss: 0.8597 - val_accuracy: 0.7344
Epoch 6/20
18/18 [==============================] - 0s 8ms/step - loss: 0.785
6 - accuracy: 0.7760 - val_loss: 0.8210 - val_accuracy: 0.7396
Epoch 7/20
18/18 [==============================] - 0s 8ms/step - loss: 0.742
7 - accuracy: 0.7726 - val_loss: 0.7898 - val_accuracy: 0.7604
Epoch 8/20
18/18 [==============================] - 0s 8ms/step - loss: 0.709
5 - accuracy: 0.7795 - val_loss: 0.7647 - val_accuracy: 0.7500
Epoch 9/20
18/18 [==============================] - 0s 7ms/step - loss: 0.681
0 - accuracy: 0.7760 - val_loss: 0.7430 - val_accuracy: 0.7448
Epoch 10/20
18/18 [==============================] - 0s 7ms/step - loss: 0.656
8 - accuracy: 0.7812 - val_loss: 0.7247 - val_accuracy: 0.7396
Epoch 11/20
18/18 [==============================] - 0s 13ms/step - loss: 0.63
58 - accuracy: 0.7865 - val_loss: 0.7076 - val_accuracy: 0.7552
Epoch 12/20
18/18 [==============================] - 0s 7ms/step - loss: 0.616
8 - accuracy: 0.7812 - val_loss: 0.6940 - val_accuracy: 0.7292
Epoch 13/20
18/18 [==============================] - 0s 7ms/step - loss: 0.601
1 - accuracy: 0.7882 - val_loss: 0.6831 - val_accuracy: 0.7344
Epoch 14/20
18/18 [==============================] - 0s 8ms/step - loss: 0.589
3 - accuracy: 0.7865 - val_loss: 0.6753 - val_accuracy: 0.7292
Epoch 15/20
18/18 [==============================] - 0s 7ms/step - loss: 0.576
2 - accuracy: 0.7882 - val_loss: 0.6616 - val_accuracy: 0.7344
Epoch 16/20
18/18 [==============================] - 0s 7ms/step - loss: 0.565
4 - accuracy: 0.7882 - val_loss: 0.6559 - val_accuracy: 0.7448
Epoch 17/20
18/18 [==============================] - 0s 8ms/step - loss: 0.556
9 - accuracy: 0.7865 - val_loss: 0.6449 - val_accuracy: 0.7292
Epoch 18/20
18/18 [==============================] - 0s 8ms/step - loss: 0.547
```

```
9 - accuracy: 0.7865 - val_loss: 0.6374 - val_accuracy: 0.7552
Epoch 19/20
18/18 [==============================] - 0s 7ms/step - loss: 0.539
6 - accuracy: 0.7951 - val_loss: 0.6344 - val_accuracy: 0.7344
Epoch 20/20
18/18 [==============================] - 0s 7ms/step - loss: 0.533
3 - accuracy: 0.7865 - val_loss: 0.6299 - val_accuracy: 0.7448
```

Out[9]:

```
<keras.callbacks.History at 0x21a9a5010d0>
```

Step 10: Evaluate Underfit Model

In [10]:

```
# Evaluate the underfit model on the test set
underfit_eval = underfit_model.evaluate(X_test_scaled, y_test)
print(f"Underfit Model - Loss: {underfit_eval[0]}, Accuracy: {unde
```

```
6/6 [==============================] - 0s 5ms/step - loss: 0.6056
- accuracy: 0.6771
Underfit Model - Loss: 0.6056109070777893, Accuracy: 0.67708331346
51184
```

Step 11: Evaluate Overfit Model

In [11]:

```
# Evaluate the overfit model on the test set
overfit_eval = overfit_model.evaluate(X_test_scaled, y_test)
print(f"Overfit Model - Loss: {overfit_eval[0]}, Accuracy: {overfi
```

```
6/6 [==============================] - 0s 6ms/step - loss: 0.9761
- accuracy: 0.6719
Overfit Model - Loss: 0.9761338233947754, Accuracy: 0.671875
```

Step 12: Evaluate Regularized Model

In [12]:

```
# Evaluate the regularized model on the test set
regularized_eval = regularized_model.evaluate(X_test_scaled, y_tes
print(f"Regularized Model - Loss: {regularized_eval[0]}, Accuracy:
```

```
6/6 [==============================] - 0s 5ms/step - loss: 0.6299
- accuracy: 0.7448
Regularized Model - Loss: 0.6299125552177429, Accuracy: 0.74479168
65348816
```

Step 13: Hyperparameter Tuning (Optional)

In [13]:

```
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
```

```
# Define a function to create a neural network model
def create_model(optimizer='adam', kernel_regularizer=None):
    model = Sequential([
        Dense(32, activation='relu', input_shape=(X_train_scaled.s
        Dense(64, activation='relu', kernel_regularizer=kernel_reg
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=optimizer, loss='binary_crossentropy',
    return model

# Create a KerasClassifier for use with GridSearchCV
model = KerasClassifier(build_fn=create_model, epochs=20, batch_si

# Define hyperparameters to search
param_grid = {
    'optimizer': ['adam', 'sgd'],
    'kernel_regularizer': [None, regularizers.l2(0.01)]
}

# Perform grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=3,
grid_result = grid.fit(X_train_scaled, y_train)

# Display the best parameters
print("Best parameters found: ", grid_result.best_params_)
```

```
6/6 [==============================] - 0s 5ms/step
6/6 [==============================] - 0s 4ms/step
6/6 [==============================] - 0s 3ms/step
6/6 [==============================] - 0s 4ms/step
6/6 [==============================] - 0s 5ms/step
6/6 [==============================] - 0s 6ms/step
6/6 [==============================] - 0s 4ms/step
6/6 [==============================] - 0s 3ms/step
6/6 [==============================] - 0s 3ms/step
6/6 [==============================] - 0s 2ms/step
6/6 [==============================] - 0s 4ms/step
6/6 [==============================] - 0s 5ms/step
Best parameters found:  {'kernel_regularizer': <keras.regularizer
s.L2 object at 0x0000021A9A529810>, 'optimizer': 'adam'}
```

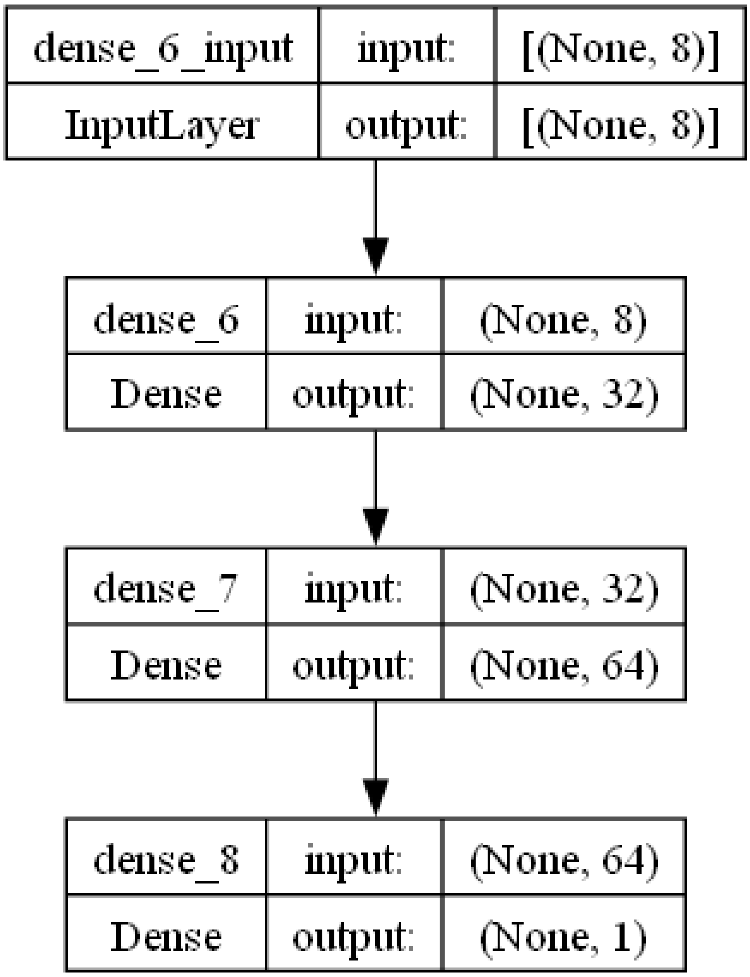
Step 14: Interpret Models

In [16]:

```
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import plot_model
import os

# Set the path to the Graphviz executables
os.environ["PATH"] += os.pathsep + 'C:\\Program Files\\Graphviz\\b

# Save the models for interpretation (optional)
underfit_model.save('underfit_model.h5')
```

```
overfit_model.save('overfit_model.h5')
regularized_model.save('regularized_model.h5')

# Load the models for interpretation (optional)
loaded_underfit_model = load_model('underfit_model.h5')
loaded_overfit_model = load_model('overfit_model.h5')
loaded_regularized_model = load_model('regularized_model.h5')

# Plot the architecture of the loaded models directly in the Jupyt
plot_model(loaded_underfit_model, show_shapes=True, show_layer_nam
plot_model(loaded_overfit_model, show_shapes=True, show_layer_name
plot_model(loaded_regularized_model, show_shapes=True, show_layer_
```

Out[16]:



Step 15: Make Predictions

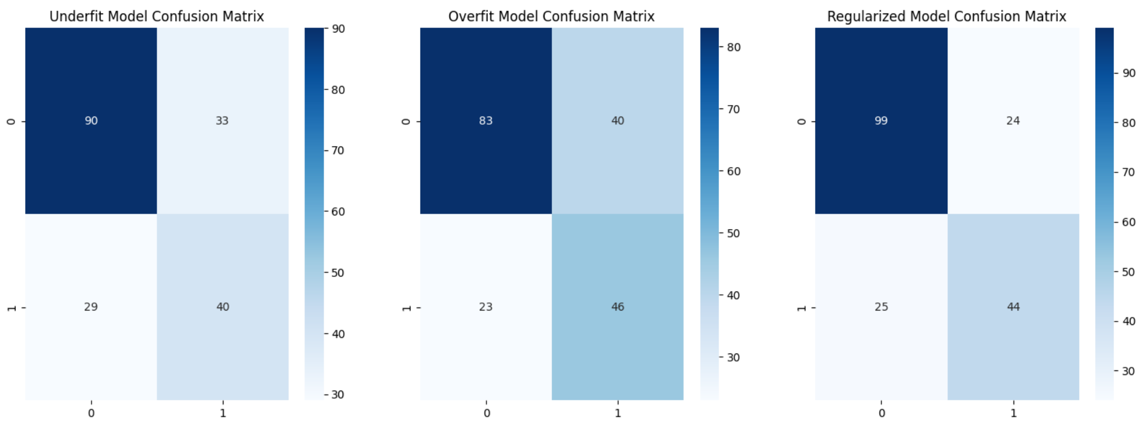
In [17]:

```
# Make predictions using the models
underfit_predictions = (underfit_model.predict(X_test_scaled) > 0.
overfit_predictions = (overfit_model.predict(X_test_scaled) > 0.5)
regularized_predictions = (regularized_model.predict(X_test_scaled
```

```
6/6 [==============================] - 0s 3ms/step
6/6 [==============================] - 0s 5ms/step
6/6 [==============================] - 0s 5ms/step
```

Step 16: Visualize Results

In [18]:
```
# Visualize confusion matrices
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Confusion matrix for the underfit model
sns.heatmap(confusion_matrix(y_test, underfit_predictions), annot=
axes[0].set_title('Underfit Model Confusion Matrix')

# Confusion matrix for the overfit model
sns.heatmap(confusion_matrix(y_test, overfit_predictions), annot=T
axes[1].set_title('Overfit Model Confusion Matrix')

# Confusion matrix for the regularized model
sns.heatmap(confusion_matrix(y_test, regularized_predictions), ann
axes[2].set_title('Regularized Model Confusion Matrix')

plt.show()
```



Step 17: Interpret Confusion Matrices

In [19]:
```
# Interpret Confusion Matrices
def interpret_confusion_matrix(conf_matrix, model_name):
    tn, fp, fn, tp = conf_matrix.ravel()

    print(f"Confusion Matrix for {model_name}:")
    print(f"True Negatives: {tn}")
    print(f"False Positives: {fp}")
    print(f"False Negatives: {fn}")
    print(f"True Positives: {tp}\n")

    # Calculate rates from the confusion matrix
    accuracy = (tp + tn) / (tp + tn + fp + fn)
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    f1 = 2 * (precision * recall) / (precision + recall)

    print(f"Accuracy for {model_name}: {accuracy:.4f}")
    print(f"Precision for {model_name}: {precision:.4f}")
    print(f"Recall for {model_name}: {recall:.4f}")
```

```
    print(f"F1 Score for {model_name}: {f1:.4f}\n")

# Interpret Confusion Matrices for each model
interpret_confusion_matrix(confusion_matrix(y_test, underfit_predi
interpret_confusion_matrix(confusion_matrix(y_test, overfit_predic
interpret_confusion_matrix(confusion_matrix(y_test, regularized_pr
```

```
Confusion Matrix for Underfit Model:
True Negatives: 90
False Positives: 33
False Negatives: 29
True Positives: 40

Accuracy for Underfit Model: 0.6771
Precision for Underfit Model: 0.5479
Recall for Underfit Model: 0.5797
F1 Score for Underfit Model: 0.5634

Confusion Matrix for Overfit Model:
True Negatives: 83
False Positives: 40
False Negatives: 23
True Positives: 46

Accuracy for Overfit Model: 0.6719
Precision for Overfit Model: 0.5349
Recall for Overfit Model: 0.6667
F1 Score for Overfit Model: 0.5935

Confusion Matrix for Regularized Model:
True Negatives: 99
False Positives: 24
False Negatives: 25
True Positives: 44

Accuracy for Regularized Model: 0.7448
Precision for Regularized Model: 0.6471
Recall for Regularized Model: 0.6377
F1 Score for Regularized Model: 0.6423
```

Step 18: Evaluate Additional Metrics

In [21]:

```
from sklearn.metrics import accuracy_score, precision_score, recal

# Evaluate additional metrics for each model
def evaluate_metrics(y_true, y_pred, model_name):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    auc_roc = roc_auc_score(y_true, y_pred)
```

```
    print(f"{model_name} Metrics:")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
    print(f"AUC-ROC: {auc_roc:.4f}\n")

# Evaluate metrics for each model
evaluate_metrics(y_test, underfit_predictions, "Underfit Model")
evaluate_metrics(y_test, overfit_predictions, "Overfit Model")
evaluate_metrics(y_test, regularized_predictions, "Regularized Mod
```

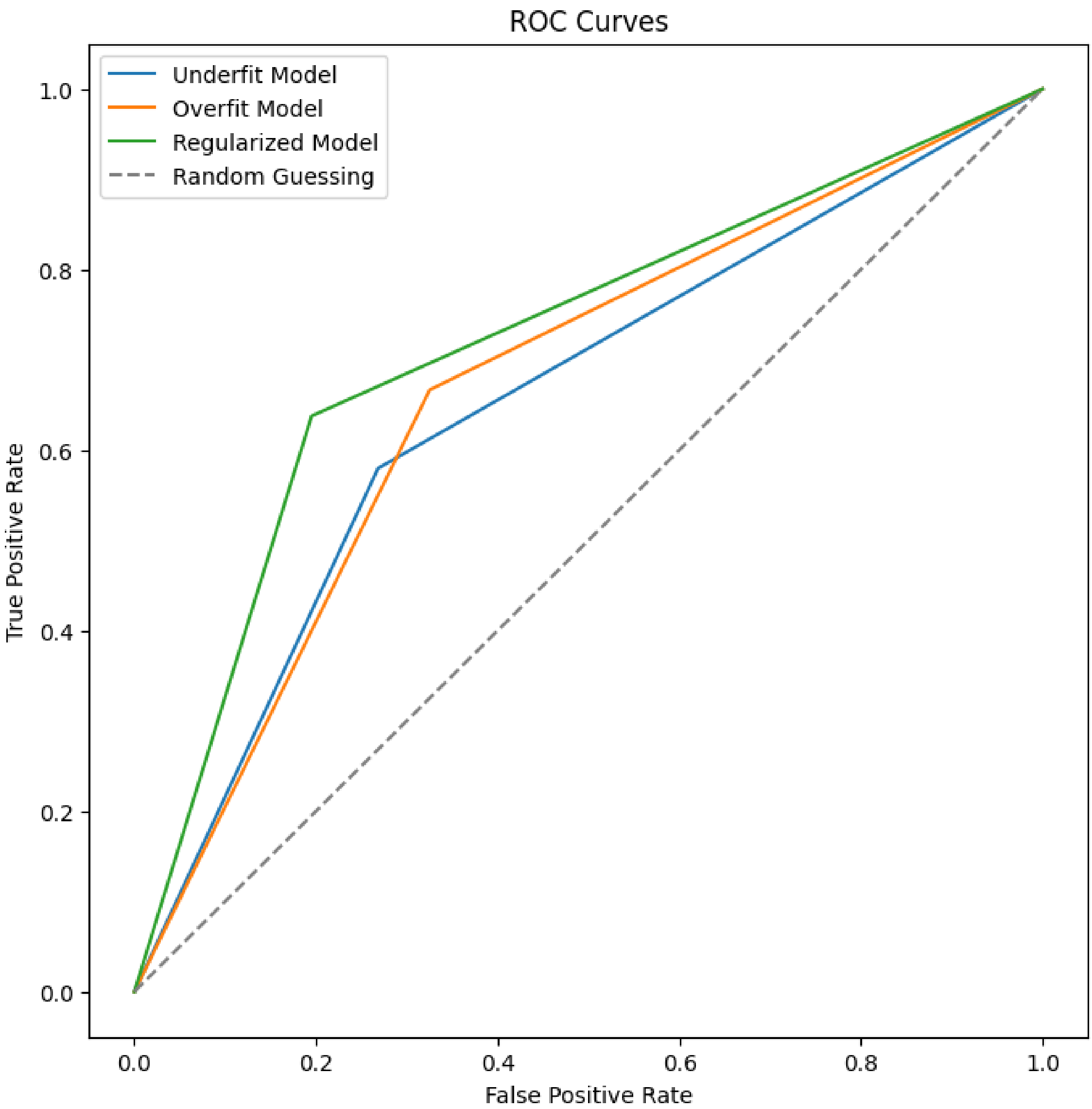
```
Underfit Model Metrics:
Accuracy: 0.6771
Precision: 0.5479
Recall: 0.5797
F1 Score: 0.5634
AUC-ROC: 0.6557

Overfit Model Metrics:
Accuracy: 0.6719
Precision: 0.5349
Recall: 0.6667
F1 Score: 0.5935
AUC-ROC: 0.6707

Regularized Model Metrics:
Accuracy: 0.7448
Precision: 0.6471
Recall: 0.6377
F1 Score: 0.6423
AUC-ROC: 0.7213
```

Step 19: Visualize ROC Curves

In [24]:

```
from sklearn.metrics import roc_curve
import matplotlib.pyplot as plt

# Plot ROC curves for each model
fig, ax = plt.subplots(figsize=(8, 8))

# ROC curve for the underfit model
fpr, tpr, _ = roc_curve(y_test, underfit_predictions)
ax.plot(fpr, tpr, label='Underfit Model')

# ROC curve for the overfit model
fpr, tpr, _ = roc_curve(y_test, overfit_predictions)
ax.plot(fpr, tpr, label='Overfit Model')

# ROC curve for the regularized model
```

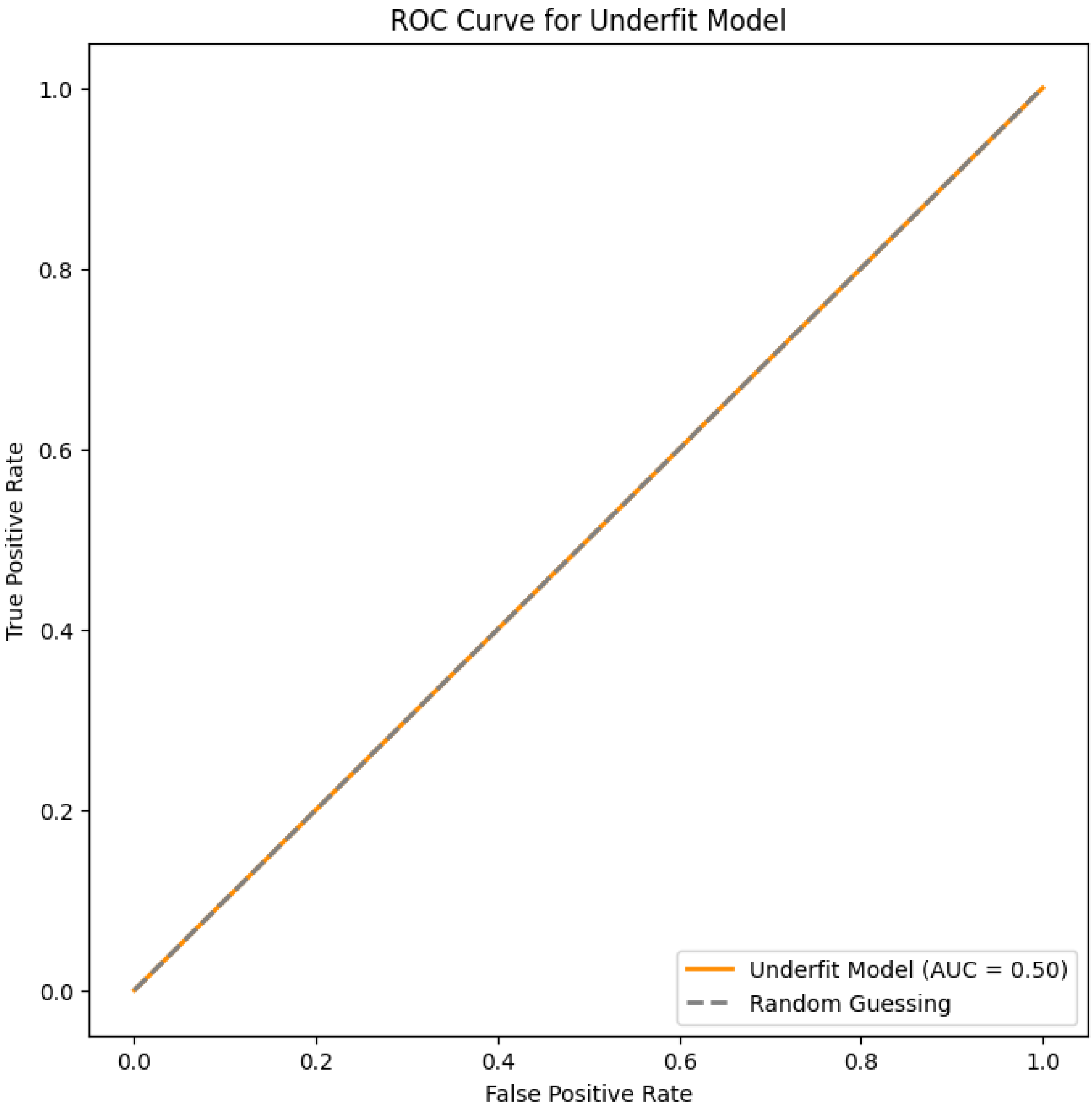
```
fpr, tpr, _ = roc_curve(y_test, regularized_predictions)
ax.plot(fpr, tpr, label='Regularized Model')

# Plot the random guessing curve
ax.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Randc

# Set plot labels and legend
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.set_title('ROC Curves')
ax.legend()

plt.show()
```

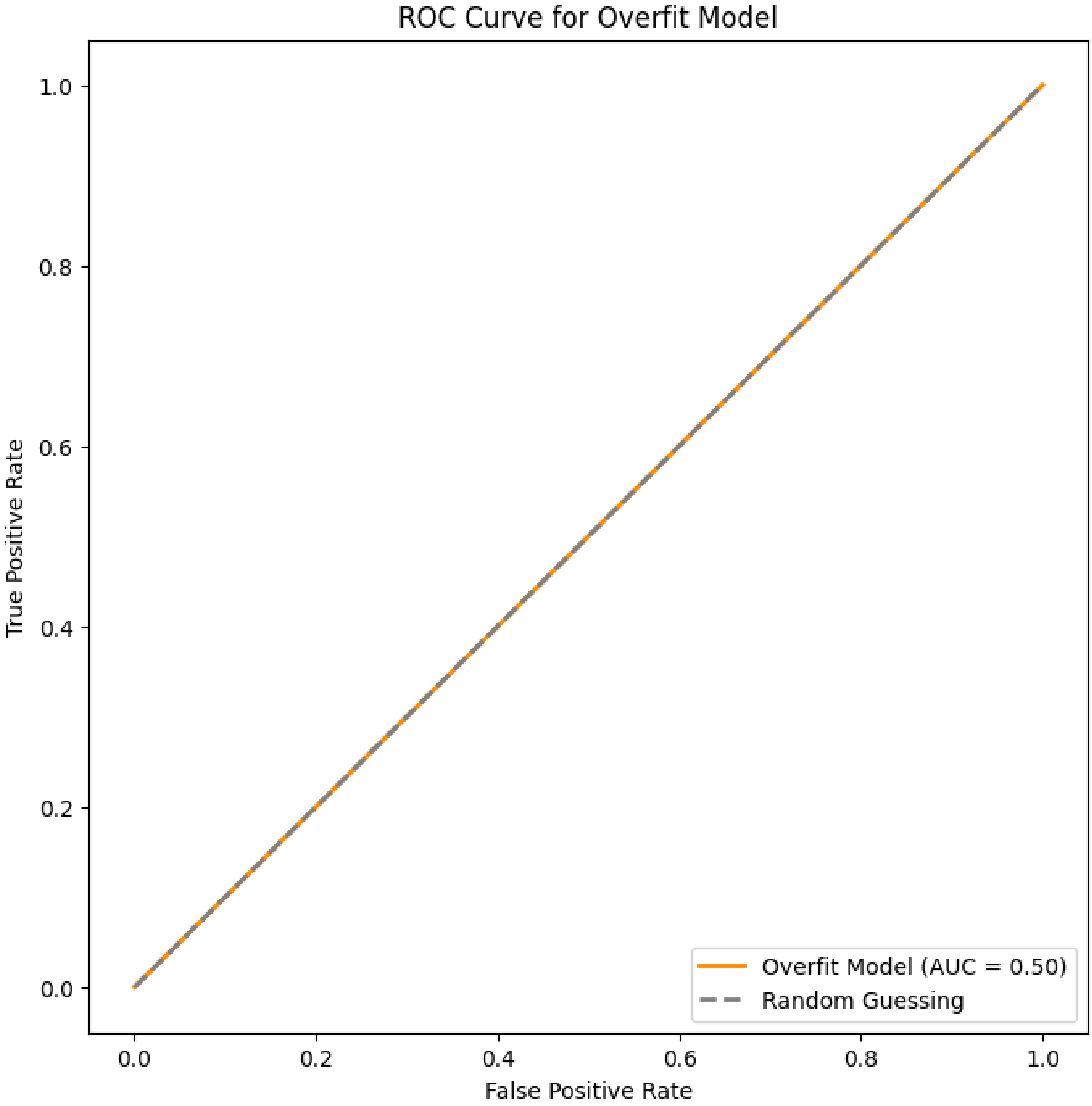


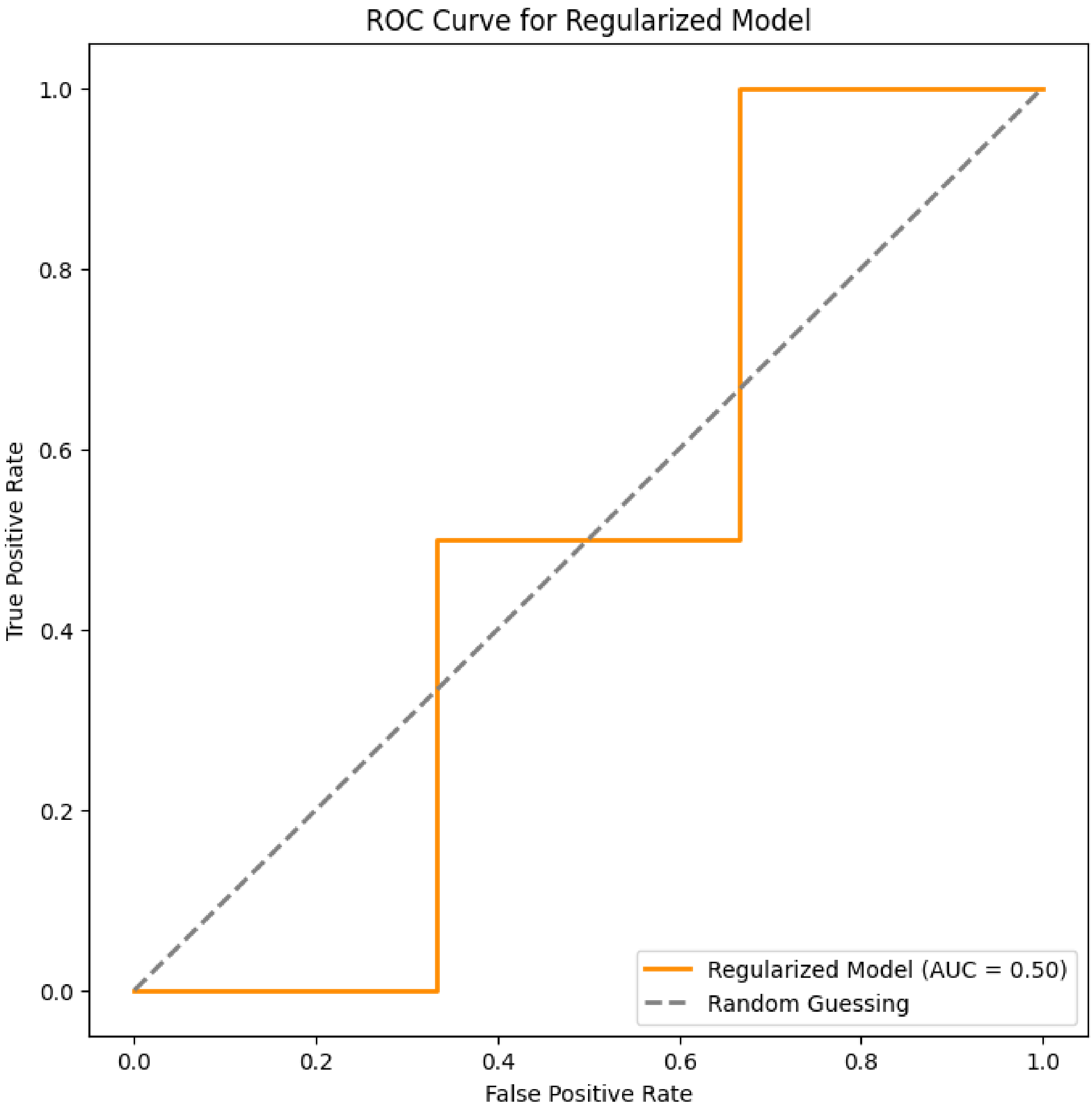
Step 20: Interpret ROC Curves

In [26]:
```
import matplotlib.pyplot as plt
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, roc_curve, auc
```

```python
# Function to interpret ROC Curves
def interpret_roc_curve(y_true, y_pred_probs, model_name):
    fpr, tpr, thresholds = roc_curve(y_true, y_pred_probs)
    roc_auc = auc(fpr, tpr)

    # Plot ROC Curve
    plt.figure(figsize=(8, 8))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'{model_na
    plt.plot([0, 1], [0, 1], color='gray', linestyle='--', lw=2, l
    plt.title(f'ROC Curve for {model_name}')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend(loc='lower right')
    plt.show()


# Create dummy classifiers for underfitting, overfitting, and regu
underfit_model = DummyClassifier(strategy='constant', constant=1)
overfit_model = DummyClassifier(strategy='constant', constant=0)
regularized_model = LogisticRegression()

# Generate sample data
X = [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]
y = [0, 1, 0, 1, 0]

# Fit models to the data
underfit_model.fit(X, y)
overfit_model.fit(X, y)
regularized_model.fit(X, y)

# Predict probabilities using predict_proba
underfit_probs = underfit_model.predict_proba(X)[:, 1]
overfit_probs = overfit_model.predict_proba(X)[:, 1]
regularized_probs = regularized_model.predict_proba(X)[:, 1]

# Interpret ROC Curves for each model
interpret_roc_curve(y, underfit_probs, 'Underfit Model')
interpret_roc_curve(y, overfit_probs, 'Overfit Model')
interpret_roc_curve(y, regularized_probs, 'Regularized Model')
```

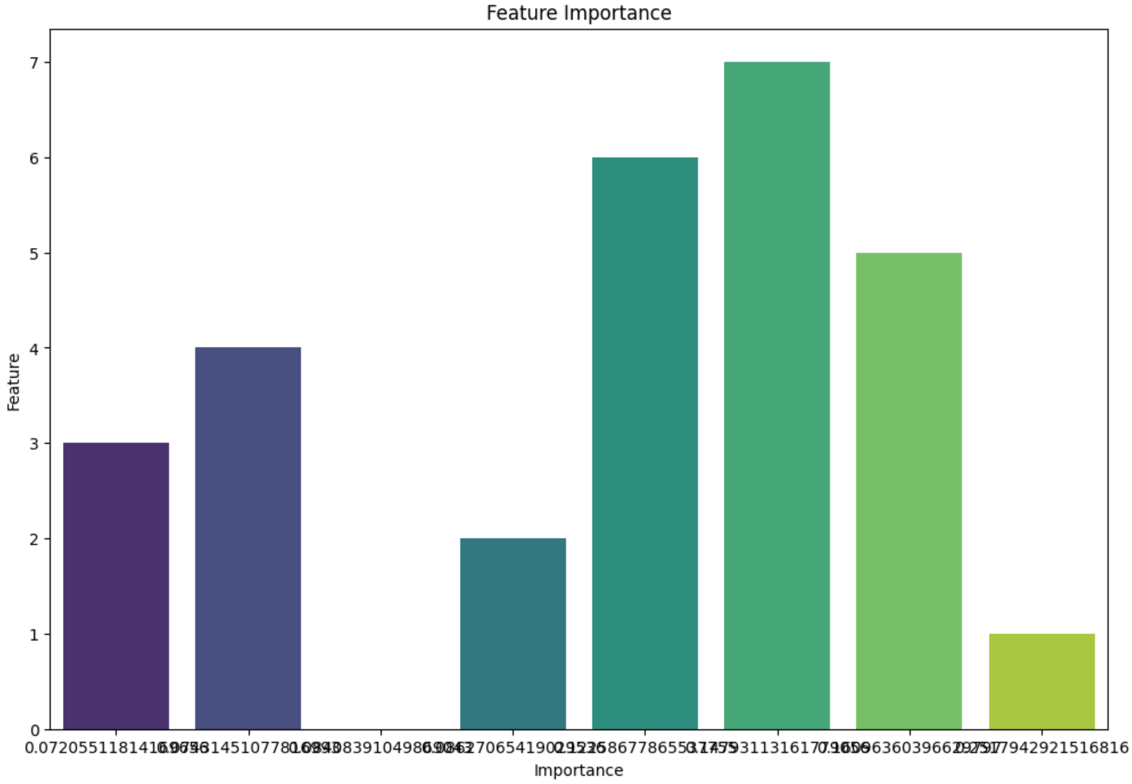

ROC Curve for Underfit Model
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Underfit Model (AUC = 0.50)
Random Guessing


ROC Curve for Overfit Model
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Overfit Model (AUC = 0.50)
Random Guessing



Step 21: Model Interpretation and Saving

In [34]:
```
# Convert X_train_scaled to a DataFrame if it's a list
X_train_scaled_df = pd.DataFrame(X_train_scaled)

# Assuming X_train_scaled_df is a DataFrame
rf_model = RandomForestClassifier()
rf_model.fit(X_train_scaled_df, y_train)

# Feature importances
feature_importances = pd.DataFrame({'Feature': X_train_scaled_df.c
feature_importances = feature_importances.sort_values(by='Importan

# Visualize feature importance
plt.figure(figsize=(12, 8))
sns.barplot(x='Importance', y='Feature', data=feature_importances,
plt.title('Feature Importance')
plt.show()
```



Save the Chosen Model

In [36]:
```
import joblib

# Save the model
joblib.dump(chosen_model, 'chosen_model.pkl')

# Optionally, save the feature scaling parameters (assuming 'scale
joblib.dump(scaler, 'scaler.pkl')
```

Out[36]:
```
['scaler.pkl']
```

In [ ]:
```
```