```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow import keras
```

```
df=pd.read_csv(r"C:\Users\arjun\Downloads\
heart_failure_clinical_records_dataset.csv")
df.head()
```

```
    age  anaemia  creatinine_phosphokinase  diabetes
ejection_fraction  \
0  75.0        0                       582         0
20
1  55.0        0                      7861         0
38
2  65.0        0                       146         0
20
3  50.0        1                       111         0
20
4  65.0        1                       160         1
20

   high_blood_pressure  platelets  serum_creatinine  serum_sodium  sex
\
0                    1  265000.00               1.9           130    1

1                    0  263358.03               1.1           136    1

2                    0  162000.00               1.3           129    1

3                    0  210000.00               1.9           137    1

4                    0  327000.00               2.7           116    0


   smoking  time  DEATH_EVENT
0        0     4            1
1        0     6            1
2        1     7            1
3        0     7            1
4        0     8            1
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 299 entries, 0 to 298
Data columns (total 13 columns):
```

```
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   age                       299 non-null    float64
 1   anaemia                   299 non-null    int64  
 2   creatinine_phosphokinase  299 non-null    int64  
 3   diabetes                  299 non-null    int64  
 4   ejection_fraction         299 non-null    int64  
 5   high_blood_pressure       299 non-null    int64  
 6   platelets                 299 non-null    float64
 7   serum_creatinine          299 non-null    float64
 8   serum_sodium              299 non-null    int64  
 9   sex                       299 non-null    int64  
 10  smoking                   299 non-null    int64  
 11  time                      299 non-null    int64  
 12  DEATH_EVENT               299 non-null    int64  
dtypes: float64(3), int64(10)
memory usage: 30.5 KB
```

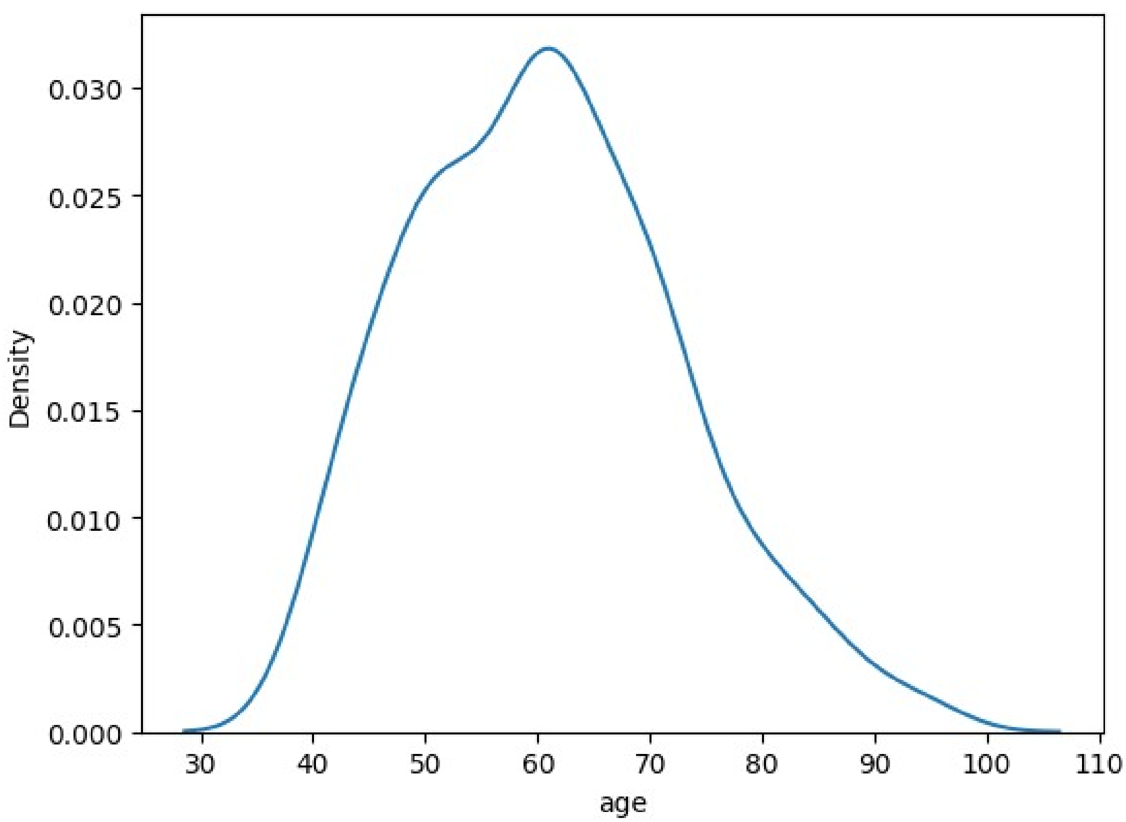
```
df.describe()
```

```
              age     anaemia  creatinine_phosphokinase    diabetes  \
count  299.000000  299.000000                299.000000  299.000000   
mean    60.833893    0.431438                581.839465    0.418060   
std     11.894809    0.496107                970.287881    0.494067   
min     40.000000    0.000000                 23.000000    0.000000   
25%     51.000000    0.000000                116.500000    0.000000   
50%     60.000000    0.000000                250.000000    0.000000   
75%     70.000000    1.000000                582.000000    1.000000   
max     95.000000    1.000000               7861.000000    1.000000   

       ejection_fraction  high_blood_pressure      platelets  \
count         299.000000           299.000000     299.000000   
mean           38.083612             0.351171  263358.029264   
std            11.834841             0.478136   97804.236869   
min            14.000000             0.000000   25100.000000   
25%            30.000000             0.000000  212500.000000   
50%            38.000000             0.000000  262000.000000   
75%            45.000000             1.000000  303500.000000   
max            80.000000             1.000000  850000.000000   

       serum_creatinine  serum_sodium         sex    smoking
time  \
count         299.00000    299.000000  299.000000  299.00000
299.000000   
mean            1.39388    136.625418    0.648829    0.32107
130.260870   
std             1.03451      4.412477    0.478136    0.46767
77.614208   
min             0.50000    113.000000    0.000000    0.00000
4.000000   
```

```
25%              0.90000   134.000000   0.000000   0.00000
73.000000
50%              1.10000   137.000000   1.000000   0.00000
115.000000
75%              1.40000   140.000000   1.000000   1.00000
203.000000
max              9.40000   148.000000   1.000000   1.00000
285.000000

       DEATH_EVENT
count    299.00000
mean       0.32107
std        0.46767
min        0.00000
25%        0.00000
50%        0.00000
75%        1.00000
max        1.00000
```

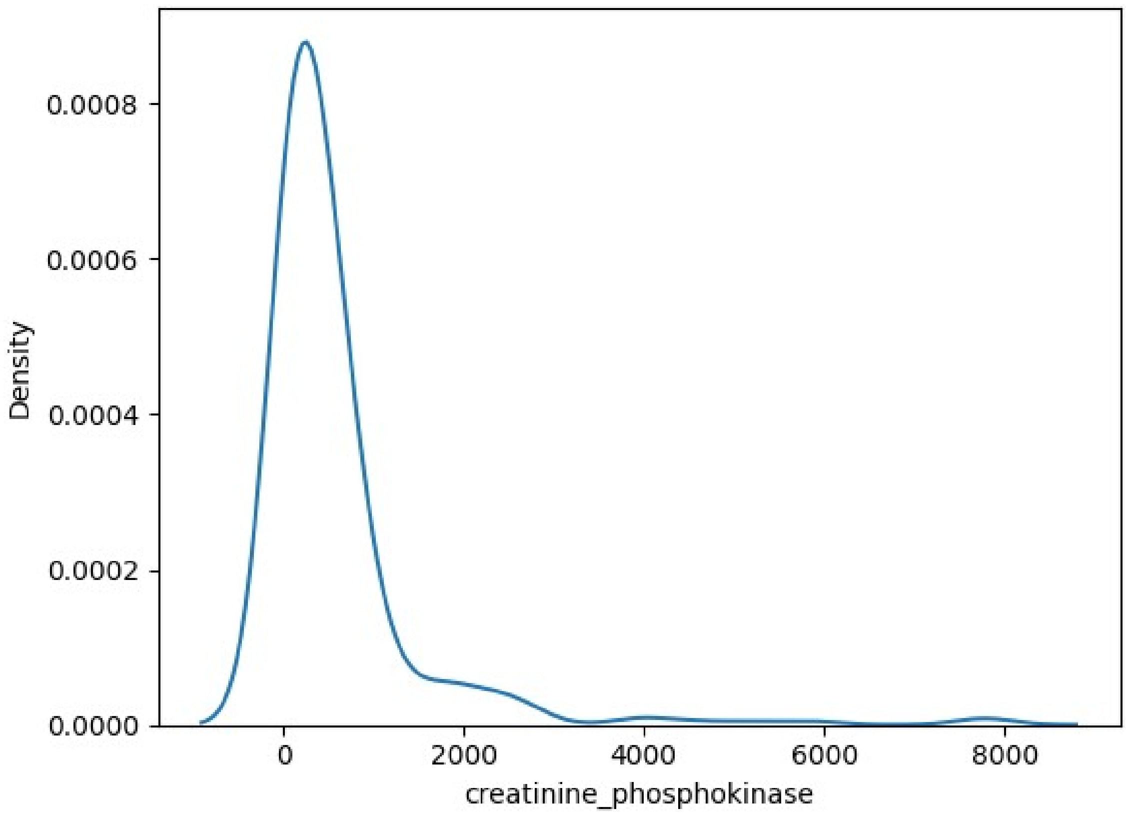
```python
sns.kdeplot(df['age'])
```

```
<Axes: xlabel='age', ylabel='Density'>
```



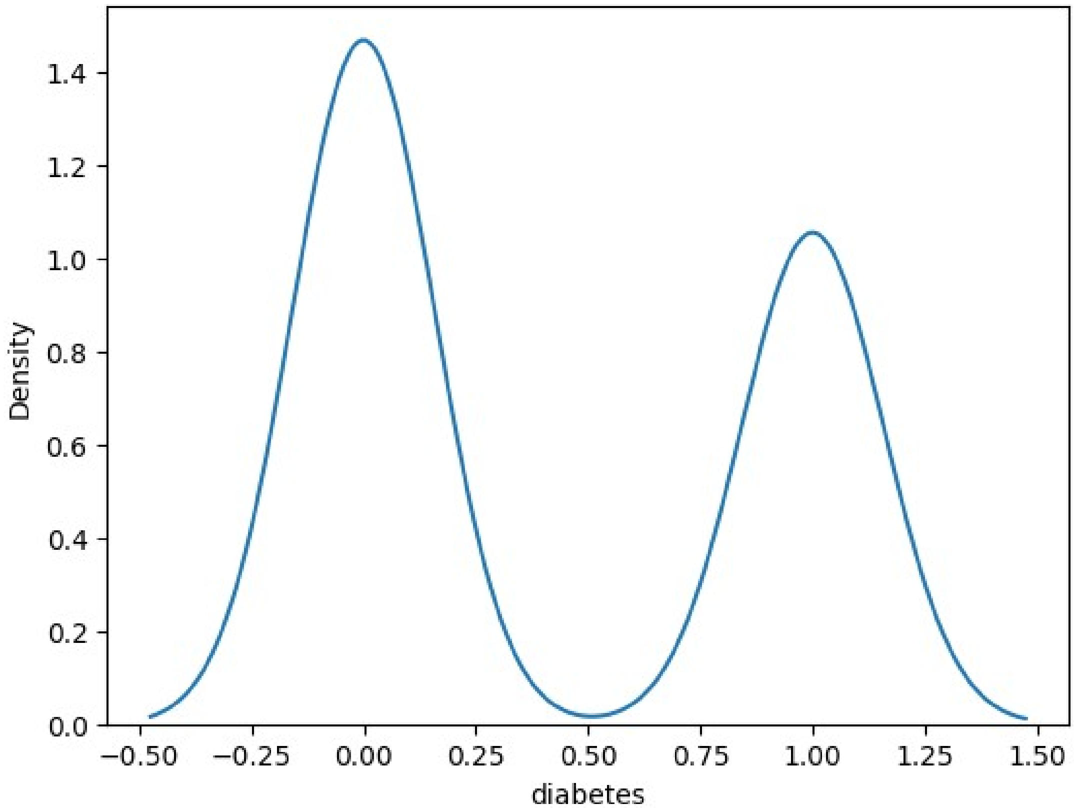

```python
sns.kdeplot(df['creatinine_phosphokinase'])
```

```
<Axes: xlabel='creatinine_phosphokinase', ylabel='Density'>
```



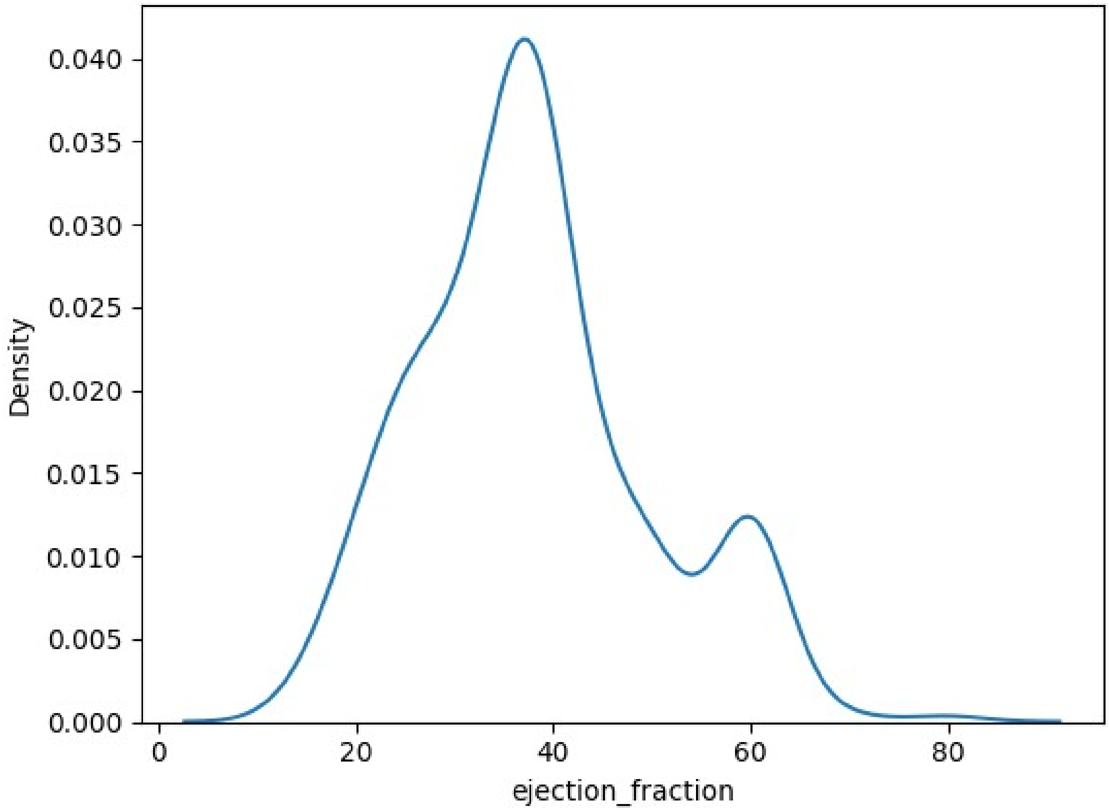

```
sns.kdeplot(df['diabetes'])
```

```
<Axes: xlabel='diabetes', ylabel='Density'>
```



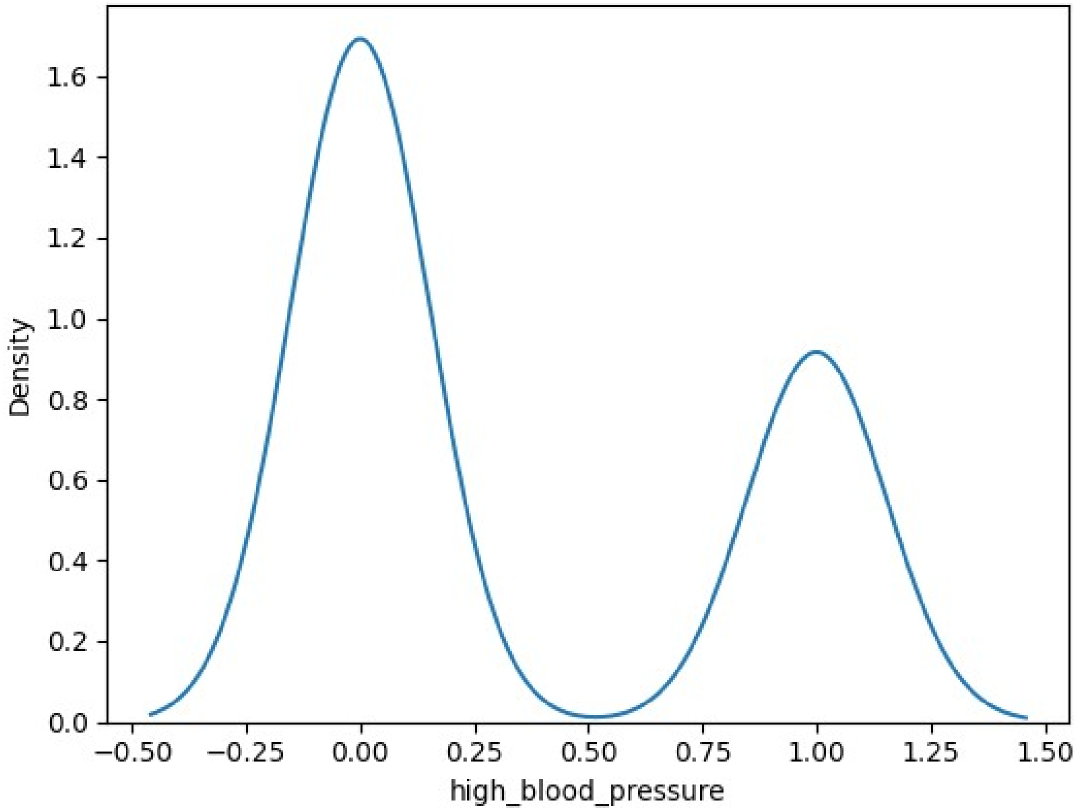

```
sns.kdeplot(df['ejection_fraction'])
```

```
<Axes: xlabel='ejection_fraction', ylabel='Density'>
```



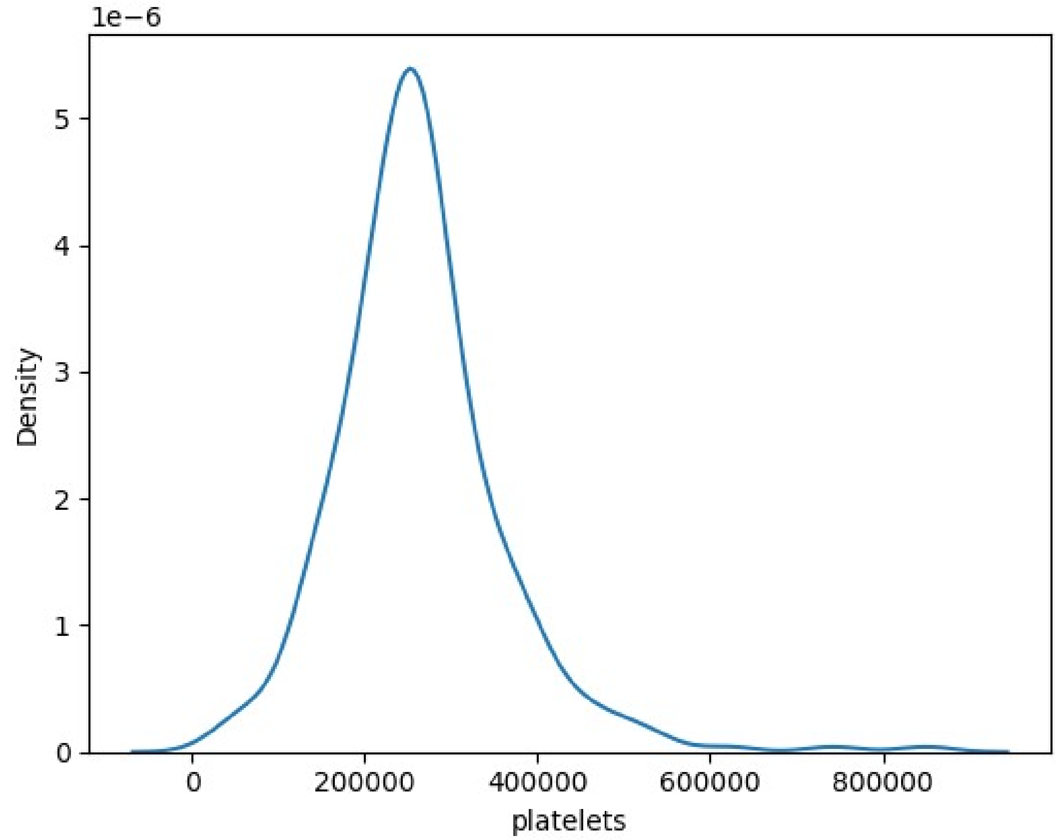

```
sns.kdeplot(df['high_blood_pressure'])
```

```
<Axes: xlabel='high_blood_pressure', ylabel='Density'>
```



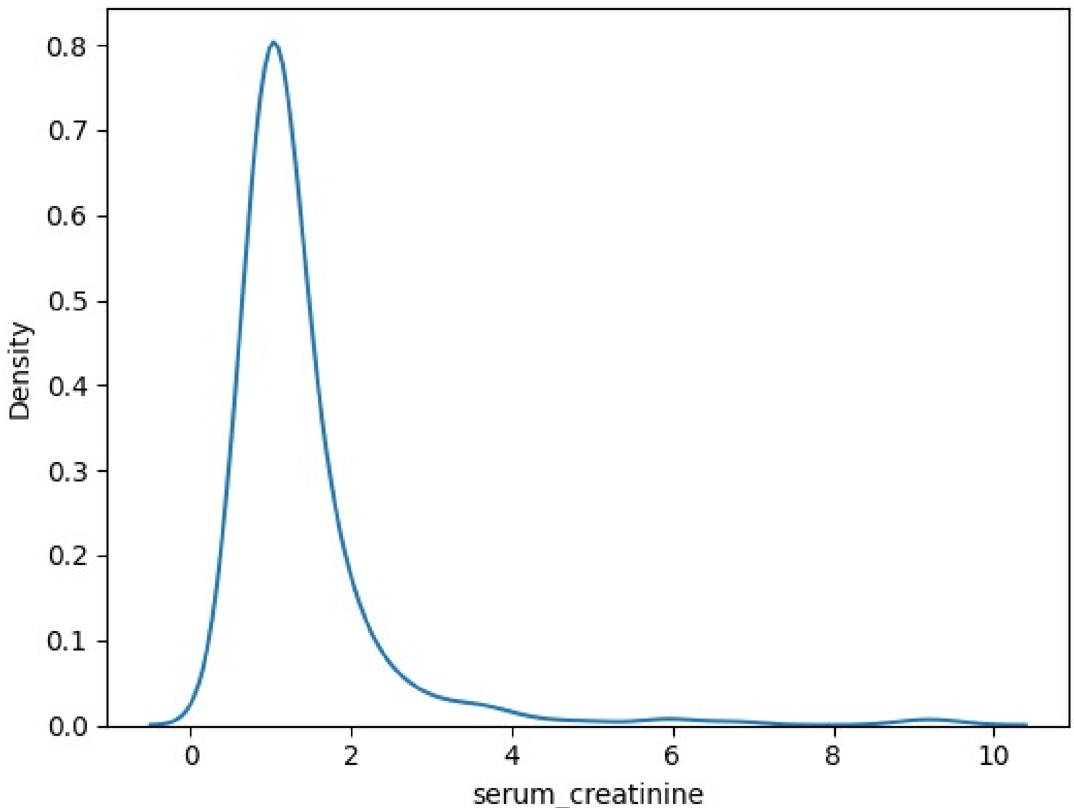

```
sns.kdeplot(df['platelets'])
```

```
<Axes: xlabel='platelets', ylabel='Density'>
```



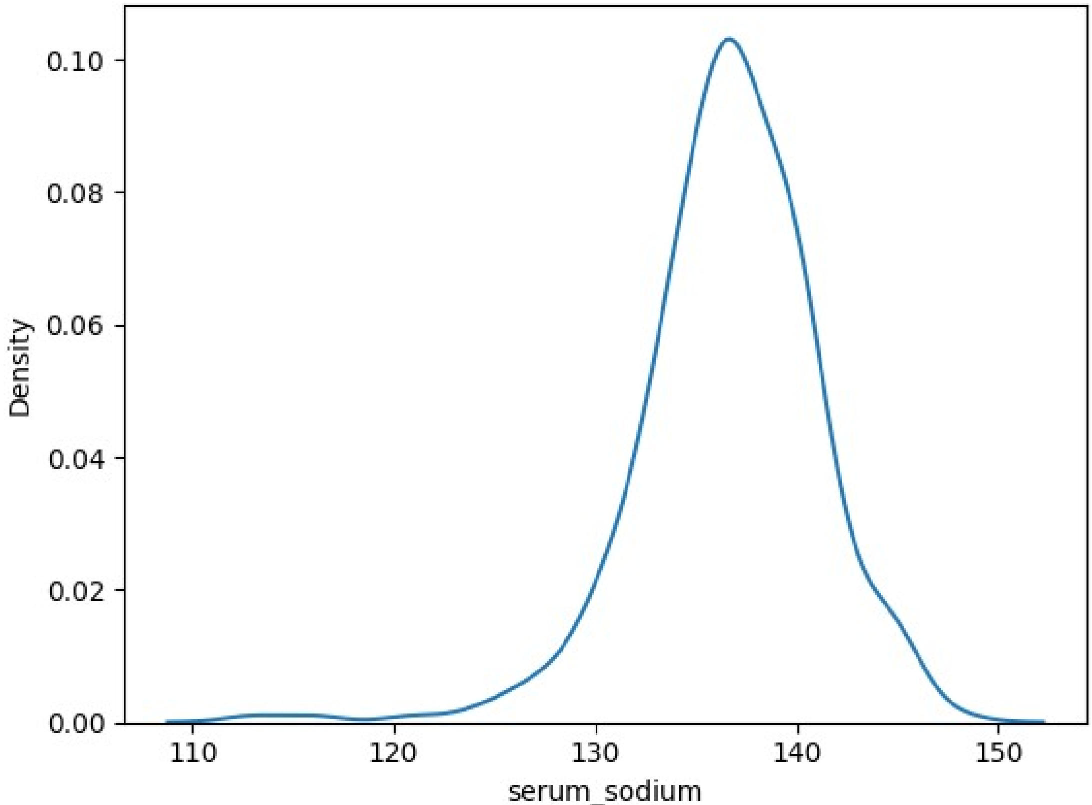

```
sns.kdeplot(df['serum_creatinine'])
```

```
<Axes: xlabel='serum_creatinine', ylabel='Density'>
```



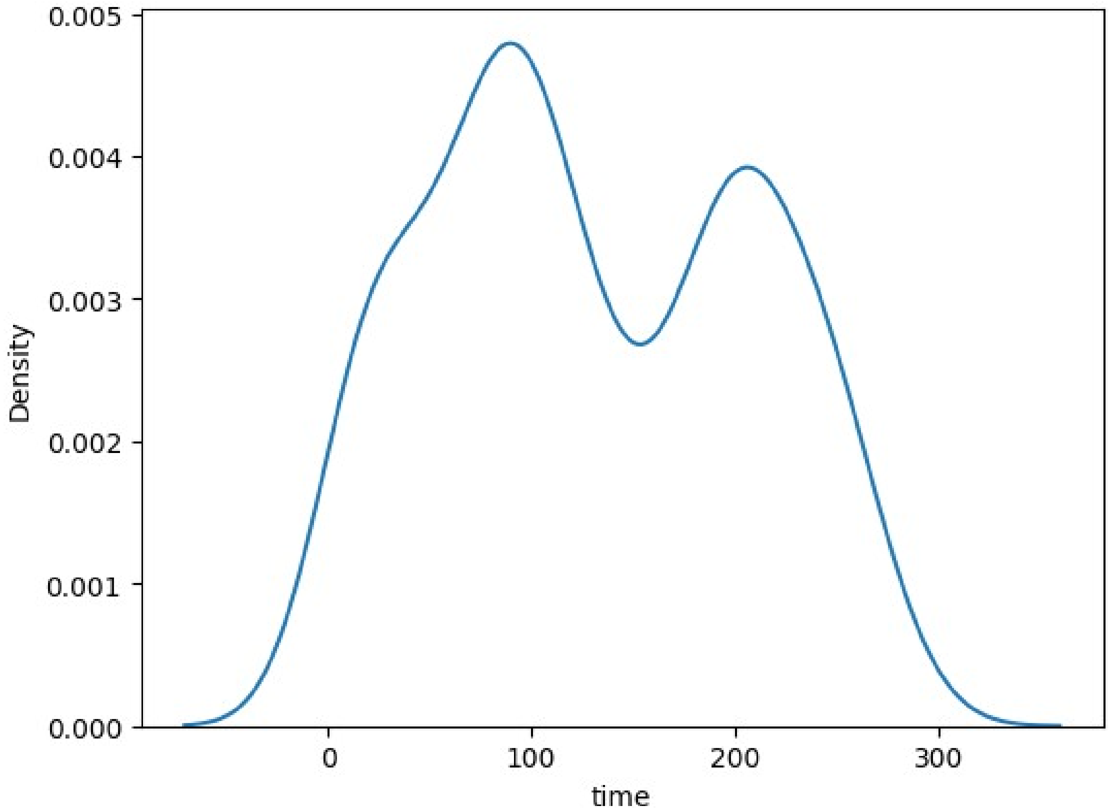

```
sns.kdeplot(df['serum_sodium'])
```

```
<Axes: xlabel='serum_sodium', ylabel='Density'>
```




```
sns.kdeplot(df['time'])
```

```
<Axes: xlabel='time', ylabel='Density'>
```



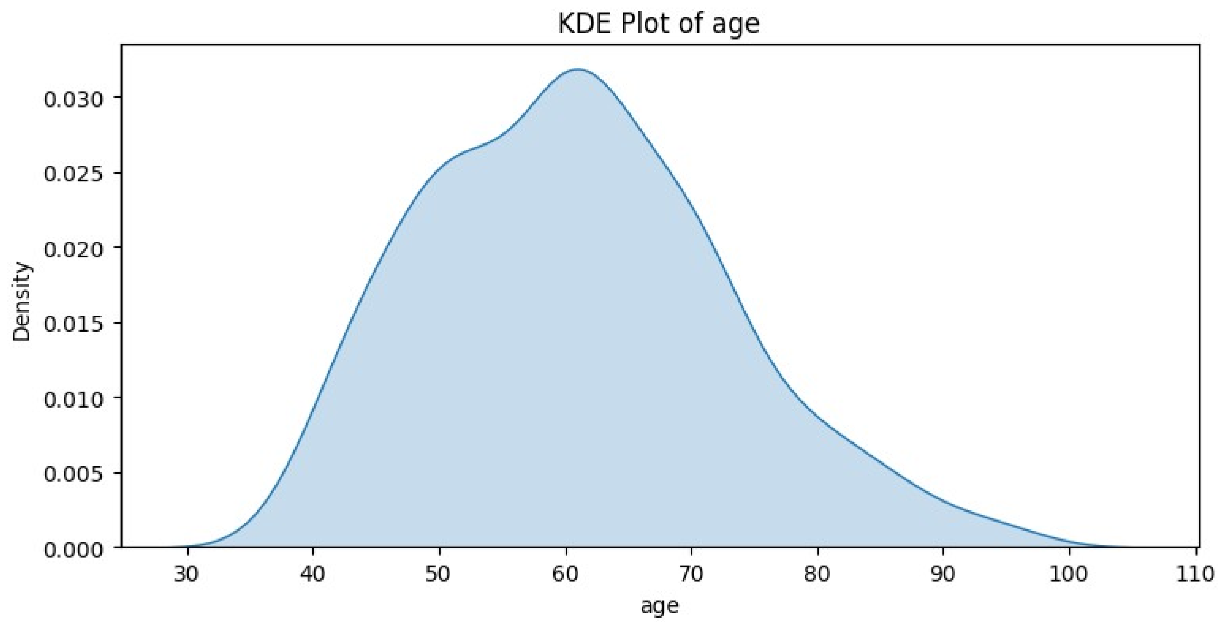

```
df['DEATH_EVENT'].value_counts()
```

```
DEATH_EVENT
0    203
1     96
Name: count, dtype: int64
```

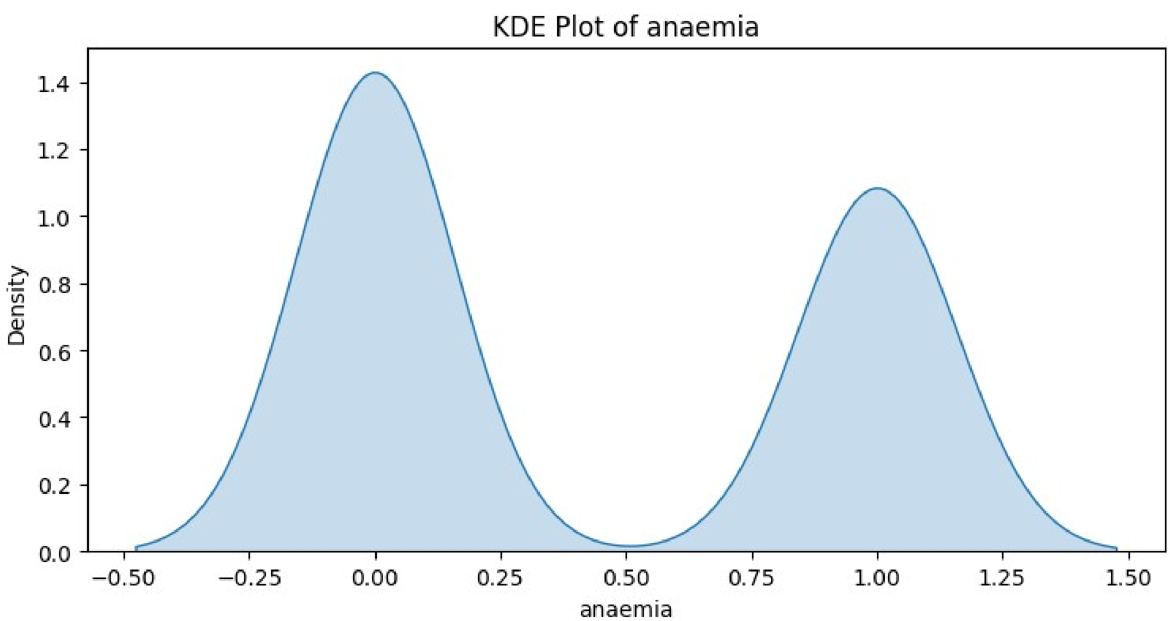
```
"""from sklearn.preprocessing import RobustScaler,MinMaxScaler

# Initialize RobustScaler
scaler = MinMaxScaler()

# Apply scaling (convert Series to 2D before transforming)
df['serum_sodium'] = scaler.fit_transform(df[['serum_sodium']])
df['serum_creatinine'] =
scaler.fit_transform(df[['serum_creatinine']])
df['platelets'] = scaler.fit_transform(df[['platelets']])
df['creatinine_phosphokinase'] =
scaler.fit_transform(df[['creatinine_phosphokinase']])
df['time'] = scaler.fit_transform(df[['time']])
df['age'] = scaler.fit_transform(df[['age']])
df['ejection_fraction'] =
scaler.fit_transform(df[['ejection_fraction']])"""
```

```
"from sklearn.preprocessing import RobustScaler,MinMaxScaler\n\n#
Initialize RobustScaler\nscaler = MinMaxScaler()\n\n# Apply scaling
(convert Series to 2D before transforming)\ndf['serum_sodium'] =
scaler.fit_transform(df[['serum_sodium']])\ndf['serum_creatinine'] =
scaler.fit_transform(df[['serum_creatinine']])\ndf['platelets'] =
scaler.fit_transform(df[['platelets']])\
ndf['creatinine_phosphokinase'] =
scaler.fit_transform(df[['creatinine_phosphokinase']])\ndf['time'] =
scaler.fit_transform(df[['time']])\ndf['age'] =
scaler.fit_transform(df[['age']])\ndf['ejection_fraction'] =
scaler.fit_transform(df[['ejection_fraction']])"

"""import pandas as pd

def remove_outliers_all_columns(df):

    Removes outliers from all numerical columns in a DataFrame using
the IQR method.

    Parameters:
    df (pd.DataFrame): The input DataFrame.

    Returns:
    pd.DataFrame: DataFrame with outliers removed from all numerical
columns.

    df_filtered = df.copy()  # Make a copy to avoid modifying the
original DataFrame

    for column in
df_filtered.select_dtypes(include=['number']).columns:
        Q1 = df_filtered[column].quantile(0.25)
        Q3 = df_filtered[column].quantile(0.75)
        IQR = Q3 - Q1

        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        df_filtered = df_filtered[(df_filtered[column] >= lower_bound)
& (df_filtered[column] <= upper_bound)]

    return df_filtered


df = remove_outliers_all_columns(df) """
```

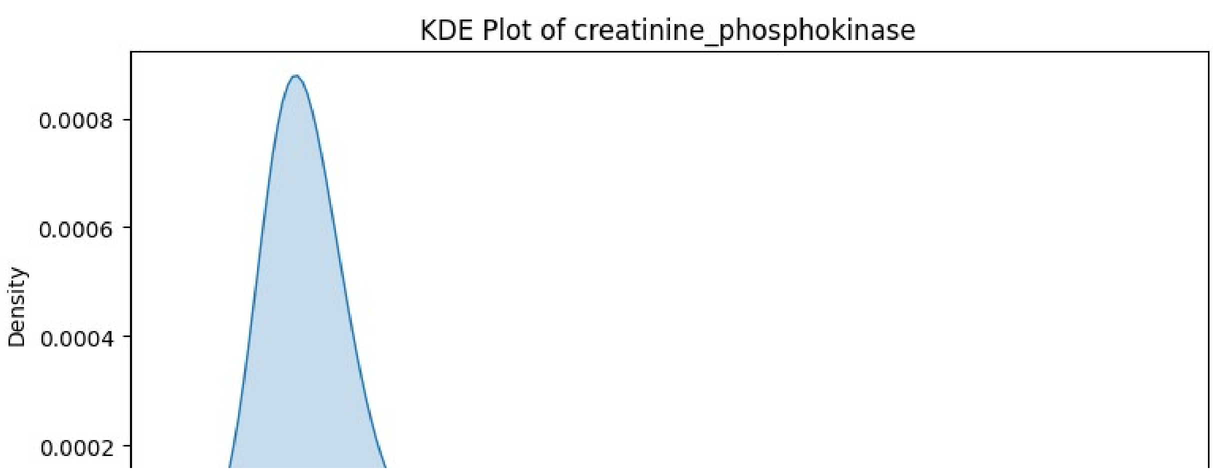
```
"import pandas as pd\n\ndef remove_outliers_all_columns(df):\n    \n
Removes outliers from all numerical columns in a DataFrame using the
IQR method.\n\n    Parameters:\n    df (pd.DataFrame): The input
DataFrame.\n\n    Returns:\n    pd.DataFrame: DataFrame with outliers
removed from all numerical columns.\n    \n    df_filtered = df.copy()
# Make a copy to avoid modifying the original DataFrame\n    \n    for
column in df_filtered.select_dtypes(include=['number']).columns:\n
Q1 = df_filtered[column].quantile(0.25)\n        Q3 =
df_filtered[column].quantile(0.75)\n        IQR = Q3 - Q1\n\n
lower_bound = Q1 - 1.5 * IQR\n        upper_bound = Q3 + 1.5 * IQR\n\n
df_filtered = df_filtered[(df_filtered[column] >= lower_bound) &
(df_filtered[column] <= upper_bound)]\n    \n    return df_filtered\n\
n\ndf = remove_outliers_all_columns(df) "

import seaborn as sns
import matplotlib.pyplot as plt

# Define number of columns to plot
num_columns = df.shape[1]  # Total columns in DataFrame

# Set up subplots
fig, axes = plt.subplots(nrows=num_columns, ncols=1, figsize=(8, 4 *
num_columns))

# Loop through each column and plot KDE
for i, column in enumerate(df.columns):
    sns.kdeplot(df[column], ax=axes[i], fill=True)
    axes[i].set_title(f'KDE Plot of {column}')

# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```



KDE Plot of age
Density
0.030
0.025
0.020
0.015
0.010
0.005
0.000
30
40
50
60
70
80
90
100
110
age



KDE Plot of anaemia
Density
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50
anaemia



KDE Plot of creatinine_phosphokinase
Density
0.0008
0.0006
0.0004
0.0002

ANN MODEL

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from imblearn.over_sampling import SMOTE
from collections import Counter


# 2 Define Columns to Scale
columns_to_scale = ['age', 'creatinine_phosphokinase',
'ejection_fraction',
                    'platelets', 'serum_creatinine',
'serum_sodium','time']  # Example columns to scale

# 3 Separate Features and Target
X = df.drop(columns=['DEATH_EVENT'])  # Drop target column
y = df['DEATH_EVENT']  # Target variable

# 4 Train-Test Split (Stratified for Balance)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42, stratify=y)

# 5 Apply SMOTE for Balancing the Dataset (Before Scaling)
smote = SMOTE(sampling_strategy=0.5,random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train,
y_train)

# 6 Apply MinMaxScaler to Selected Columns
scaler = MinMaxScaler()
X_train_resampled[columns_to_scale] =
scaler.fit_transform(X_train_resampled[columns_to_scale])
X_test[columns_to_scale] = scaler.transform(X_test[columns_to_scale])

# 7 Define ANN Model
model = Sequential([
    Dense(16, activation='relu',
input_shape=(X_train_resampled.shape[1],)),  # Input layer
    Dense(8, activation='relu'),  # Hidden layer
    Dense(1, activation='sigmoid')  # Output layer for binary
classification
])

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

```
# 9 Train Model
history = model.fit(X_train_resampled, y_train_resampled, epochs=50,
batch_size=32, validation_data=(X_test, y_test))

# □ Evaluate Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 1/50
8/8 [==============================] - 3s 85ms/step - loss: 0.6668 -
accuracy: 0.6667 - val_loss: 0.6389 - val_accuracy: 0.6833
Epoch 2/50
8/8 [==============================] - 0s 25ms/step - loss: 0.6619 -
accuracy: 0.6667 - val_loss: 0.6341 - val_accuracy: 0.6833
Epoch 3/50
8/8 [==============================] - 0s 24ms/step - loss: 0.6573 -
accuracy: 0.6667 - val_loss: 0.6300 - val_accuracy: 0.6833
Epoch 4/50
8/8 [==============================] - 0s 24ms/step - loss: 0.6532 -
accuracy: 0.6667 - val_loss: 0.6260 - val_accuracy: 0.6833
Epoch 5/50
8/8 [==============================] - 0s 24ms/step - loss: 0.6491 -
accuracy: 0.6667 - val_loss: 0.6230 - val_accuracy: 0.6833
Epoch 6/50
8/8 [==============================] - 0s 24ms/step - loss: 0.6450 -
accuracy: 0.6667 - val_loss: 0.6197 - val_accuracy: 0.6833
Epoch 7/50
8/8 [==============================] - 0s 22ms/step - loss: 0.6413 -
accuracy: 0.6667 - val_loss: 0.6170 - val_accuracy: 0.6833
Epoch 8/50
8/8 [==============================] - 0s 24ms/step - loss: 0.6373 -
accuracy: 0.6667 - val_loss: 0.6135 - val_accuracy: 0.6833
Epoch 9/50
8/8 [==============================] - 0s 22ms/step - loss: 0.6339 -
accuracy: 0.6667 - val_loss: 0.6108 - val_accuracy: 0.6833
Epoch 10/50
8/8 [==============================] - 0s 23ms/step - loss: 0.6305 -
accuracy: 0.6667 - val_loss: 0.6080 - val_accuracy: 0.6833
Epoch 11/50
8/8 [==============================] - 0s 25ms/step - loss: 0.6271 -
accuracy: 0.6667 - val_loss: 0.6052 - val_accuracy: 0.6833
Epoch 12/50
8/8 [==============================] - 0s 25ms/step - loss: 0.6241 -
accuracy: 0.6667 - val_loss: 0.6025 - val_accuracy: 0.6833
Epoch 13/50
8/8 [==============================] - 0s 23ms/step - loss: 0.6208 -
accuracy: 0.6667 - val_loss: 0.6001 - val_accuracy: 0.6833
Epoch 14/50
8/8 [==============================] - 0s 23ms/step - loss: 0.6178 -
```

```
accuracy: 0.6667 - val_loss: 0.5973 - val_accuracy: 0.6833
Epoch 15/50
8/8 [==============================] - 0s 25ms/step - loss: 0.6143 -
accuracy: 0.6667 - val_loss: 0.5948 - val_accuracy: 0.6833
Epoch 16/50
8/8 [==============================] - 0s 24ms/step - loss: 0.6109 -
accuracy: 0.6667 - val_loss: 0.5932 - val_accuracy: 0.6833
Epoch 17/50
8/8 [==============================] - 0s 23ms/step - loss: 0.6074 -
accuracy: 0.6667 - val_loss: 0.5904 - val_accuracy: 0.6833
Epoch 18/50
8/8 [==============================] - 0s 25ms/step - loss: 0.6037 -
accuracy: 0.6667 - val_loss: 0.5877 - val_accuracy: 0.6833
Epoch 19/50
8/8 [==============================] - 0s 25ms/step - loss: 0.6006 -
accuracy: 0.6667 - val_loss: 0.5849 - val_accuracy: 0.6833
Epoch 20/50
8/8 [==============================] - 0s 23ms/step - loss: 0.5969 -
accuracy: 0.6667 - val_loss: 0.5818 - val_accuracy: 0.6833
Epoch 21/50
8/8 [==============================] - 0s 23ms/step - loss: 0.5933 -
accuracy: 0.6667 - val_loss: 0.5789 - val_accuracy: 0.6833
Epoch 22/50
8/8 [==============================] - 0s 24ms/step - loss: 0.5896 -
accuracy: 0.6667 - val_loss: 0.5765 - val_accuracy: 0.6833
Epoch 23/50
8/8 [==============================] - 0s 27ms/step - loss: 0.5855 -
accuracy: 0.6667 - val_loss: 0.5737 - val_accuracy: 0.6833
Epoch 24/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5820 -
accuracy: 0.6667 - val_loss: 0.5707 - val_accuracy: 0.6833
Epoch 25/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5775 -
accuracy: 0.6667 - val_loss: 0.5677 - val_accuracy: 0.6833
Epoch 26/50
8/8 [==============================] - 0s 26ms/step - loss: 0.5733 -
accuracy: 0.6667 - val_loss: 0.5647 - val_accuracy: 0.6833
Epoch 27/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5693 -
accuracy: 0.6667 - val_loss: 0.5611 - val_accuracy: 0.6833
Epoch 28/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5647 -
accuracy: 0.6667 - val_loss: 0.5583 - val_accuracy: 0.6833
Epoch 29/50
8/8 [==============================] - 0s 23ms/step - loss: 0.5603 -
accuracy: 0.6667 - val_loss: 0.5553 - val_accuracy: 0.6833
Epoch 30/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5558 -
accuracy: 0.6667 - val_loss: 0.5526 - val_accuracy: 0.6833
```

```
Epoch 31/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5516 -
accuracy: 0.6667 - val_loss: 0.5495 - val_accuracy: 0.6833
Epoch 32/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5464 -
accuracy: 0.6667 - val_loss: 0.5463 - val_accuracy: 0.6833
Epoch 33/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5419 -
accuracy: 0.6667 - val_loss: 0.5429 - val_accuracy: 0.6833
Epoch 34/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5368 -
accuracy: 0.6790 - val_loss: 0.5392 - val_accuracy: 0.6833
Epoch 35/50
8/8 [==============================] - 0s 26ms/step - loss: 0.5328 -
accuracy: 0.6790 - val_loss: 0.5355 - val_accuracy: 0.7000
Epoch 36/50
8/8 [==============================] - 0s 24ms/step - loss: 0.5258 -
accuracy: 0.7037 - val_loss: 0.5317 - val_accuracy: 0.7167
Epoch 37/50
8/8 [==============================] - 0s 24ms/step - loss: 0.5201 -
accuracy: 0.7160 - val_loss: 0.5282 - val_accuracy: 0.7167
Epoch 38/50
8/8 [==============================] - 0s 25ms/step - loss: 0.5144 -
accuracy: 0.7160 - val_loss: 0.5252 - val_accuracy: 0.7167
Epoch 39/50
8/8 [==============================] - 0s 24ms/step - loss: 0.5086 -
accuracy: 0.7202 - val_loss: 0.5219 - val_accuracy: 0.7000
Epoch 40/50
8/8 [==============================] - 0s 26ms/step - loss: 0.5017 -
accuracy: 0.7407 - val_loss: 0.5188 - val_accuracy: 0.7000
Epoch 41/50
8/8 [==============================] - 0s 24ms/step - loss: 0.4954 -
accuracy: 0.7613 - val_loss: 0.5161 - val_accuracy: 0.7667
Epoch 42/50
8/8 [==============================] - 0s 24ms/step - loss: 0.4901 -
accuracy: 0.7860 - val_loss: 0.5132 - val_accuracy: 0.7667
Epoch 43/50
8/8 [==============================] - 0s 24ms/step - loss: 0.4844 -
accuracy: 0.7901 - val_loss: 0.5103 - val_accuracy: 0.7667
Epoch 44/50
8/8 [==============================] - 0s 25ms/step - loss: 0.4798 -
accuracy: 0.7819 - val_loss: 0.5086 - val_accuracy: 0.7667
Epoch 45/50
8/8 [==============================] - 0s 26ms/step - loss: 0.4739 -
accuracy: 0.7819 - val_loss: 0.5056 - val_accuracy: 0.7667
Epoch 46/50
8/8 [==============================] - 0s 24ms/step - loss: 0.4676 -
accuracy: 0.7860 - val_loss: 0.5037 - val_accuracy: 0.7667
Epoch 47/50
```

```
8/8 [==============================] - 0s 24ms/step - loss: 0.4625 -
accuracy: 0.7984 - val_loss: 0.5018 - val_accuracy: 0.7833
Epoch 48/50
8/8 [==============================] - 0s 26ms/step - loss: 0.4580 -
accuracy: 0.8107 - val_loss: 0.5001 - val_accuracy: 0.7833
Epoch 49/50
8/8 [==============================] - 0s 24ms/step - loss: 0.4535 -
accuracy: 0.8148 - val_loss: 0.4978 - val_accuracy: 0.7833
Epoch 50/50
8/8 [==============================] - 0s 24ms/step - loss: 0.4496 -
accuracy: 0.8148 - val_loss: 0.4960 - val_accuracy: 0.7833
2/2 [==============================] - 0s 6ms/step - loss: 0.4960 -
accuracy: 0.7833
Test Accuracy: 0.7833
```

```python
import matplotlib.pyplot as plt

# Assuming history is from model.fit()
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title("Training vs Validation Loss")
plt.show()
```


```
train_loss, train_acc = model.evaluate(X_train_resampled,
y_train_resampled)
test_loss, test_acc = model.evaluate(X_test, y_test)

print(f"Training Accuracy: {train_acc:.4f}")
print(f"Testing Accuracy: {test_acc:.4f}")

8/8 [==============================] - 0s 11ms/step - loss: 0.4462 -
accuracy: 0.8148
2/2 [==============================] - 0s 13ms/step - loss: 0.4960 -
accuracy: 0.7833
Training Accuracy: 0.8148
Testing Accuracy: 0.7833

# Dummy Input (Without Target Column)
dummy_input = np.array([[75, 0, 582, 0, 20, 1, 265000.00, 1.9, 130, 1,
0, 4]])  # 12 Features

# Convert to DataFrame with Proper Column Names
dummy_input_df = pd.DataFrame(dummy_input, columns=X.columns)  #
X.columns ensures correct order

# Scale Only Selected Columns
```

```
dummy_input_df[columns_to_scale] =
scaler.transform(dummy_input_df[columns_to_scale])

# Convert to NumPy for Prediction
dummy_input_scaled = dummy_input_df.to_numpy()

# Make a Prediction
prediction = model.predict(dummy_input_scaled)

# Convert Probability to Class (0 or 1)
predicted_class = (prediction > 0.5).astype(int)

print(f"Predicted Probability: {prediction[0][0]:.4f}")
print(f"Predicted Death Event: {predicted_class[0][0]}")
```

```
WARNING:tensorflow:5 out of the last 7 calls to <function
Model.make_predict_function.<locals>.predict_function at
0x0000021382204040> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.
1/1 [==============================] - 0s 179ms/step
Predicted Probability: 0.6417
Predicted Death Event: 1
```

```
import numpy as np
import pandas as pd

# Define column names
columns = ["age", "anaemia", "creatinine_phosphokinase", "diabetes",
"ejection_fraction",
           "high_blood_pressure", "platelets", "serum_creatinine",
"serum_sodium",
           "sex", "smoking", "time"]

# Case 1: High Risk of Death (Expected DEATH_EVENT = 1)
high_risk_patient = np.array([[85, 1, 1000, 1, 15, 1, 150000, 2.5,
125, 1, 1, 3]])

# Case 2: Low Risk of Death (Expected DEATH_EVENT = 0)
low_risk_patient = np.array([[40, 0, 250, 0, 50, 0, 300000, 1.0, 140,
0, 0, 30]])
```

```
# Convert to DataFrame for scaling
high_risk_df = pd.DataFrame(high_risk_patient, columns=columns)
low_risk_df = pd.DataFrame(low_risk_patient, columns=columns)

# Scale specific columns (same scaler used for training)
columns_to_scale = ["age", "creatinine_phosphokinase",
"ejection_fraction",
                    "platelets", "serum_creatinine", "serum_sodium",
"time"]

high_risk_df[columns_to_scale] =
scaler.transform(high_risk_df[columns_to_scale])
low_risk_df[columns_to_scale] =
scaler.transform(low_risk_df[columns_to_scale])

# Make predictions
high_risk_pred = model.predict(high_risk_df)
low_risk_pred = model.predict(low_risk_df)

print(f"High Risk Patient - Predicted Probability: {high_risk_pred[0]
[0]:.4f}, Predicted Death Event: {int(high_risk_pred[0][0] > 0.5)}")
print(f"Low Risk Patient - Predicted Probability: {low_risk_pred[0]
[0]:.4f}, Predicted Death Event: {int(low_risk_pred[0][0] > 0.5)}")
```

```
WARNING:tensorflow:6 out of the last 8 calls to <function
Model.make_predict_function.<locals>.predict_function at
0x0000021382204040> triggered tf.function retracing. Tracing is
expensive and the excessive number of tracings could be due to (1)
creating @tf.function repeatedly in a loop, (2) passing tensors with
different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more
details.
1/1 [==============================] - 0s 176ms/step
1/1 [==============================] - 0s 58ms/step
High Risk Patient - Predicted Probability: 0.7465, Predicted Death
Event: 1
Low Risk Patient - Predicted Probability: 0.2658, Predicted Death
Event: 0
```

```
import numpy as np
import pandas as pd
# Define column names
columns = ["age", "anaemia", "creatinine_phosphokinase", "diabetes",
"ejection_fraction",
           "high_blood_pressure", "platelets", "serum_creatinine",
"serum_sodium",
```

```
           "sex", "smoking", "time"]
# Case 1: High Risk of Death (Expected DEATH_EVENT = 1)
high_risk_patient = np.array([[85, 1, 1000, 1, 15, 1, 150000, 2.5,
125, 1, 1, 3]])
# Case 2: Low Risk of Death (Expected DEATH_EVENT = 0)
low_risk_patient = np.array([[40, 0, 250, 0, 50, 0, 300000, 1.0, 140,
0, 0, 30]])
# Case 3: Moderate-High Risk of Death (Expected DEATH_EVENT = 1)
moderate_high_risk = np.array([[72, 1, 850, 1, 25, 1, 200000, 1.9,
130, 1, 0, 6]])
# Case 4: Moderate-Low Risk of Death (Expected DEATH_EVENT = 0)
moderate_low_risk = np.array([[55, 0, 350, 0, 45, 0, 280000, 1.2, 138,
0, 1, 25]])
# Convert to DataFrame for scaling
high_risk_df = pd.DataFrame(high_risk_patient, columns=columns)
low_risk_df = pd.DataFrame(low_risk_patient, columns=columns)
moderate_high_df = pd.DataFrame(moderate_high_risk, columns=columns)
moderate_low_df = pd.DataFrame(moderate_low_risk, columns=columns)
# Scale specific columns (same scaler used for training)
columns_to_scale = ["age", "creatinine_phosphokinase",
"ejection_fraction",
                    "platelets", "serum_creatinine", "serum_sodium",
"time"]
high_risk_df[columns_to_scale] =
scaler.transform(high_risk_df[columns_to_scale])
low_risk_df[columns_to_scale] =
scaler.transform(low_risk_df[columns_to_scale])
moderate_high_df[columns_to_scale] =
scaler.transform(moderate_high_df[columns_to_scale])
moderate_low_df[columns_to_scale] =
scaler.transform(moderate_low_df[columns_to_scale])
# Make predictions
high_risk_pred = model.predict(high_risk_df)
low_risk_pred = model.predict(low_risk_df)
moderate_high_pred = model.predict(moderate_high_df)
moderate_low_pred = model.predict(moderate_low_df)
print(f"High Risk Patient - Predicted Probability: {high_risk_pred[0]
[0]:.4f}, Predicted Death Event: {int(high_risk_pred[0][0] > 0.5)}")
print(f"Low Risk Patient - Predicted Probability: {low_risk_pred[0]
[0]:.4f}, Predicted Death Event: {int(low_risk_pred[0][0] > 0.5)}")
print(f"Moderate-High Risk Patient - Predicted Probability:
{moderate_high_pred[0][0]:.4f}, Predicted Death Event:
{int(moderate_high_pred[0][0] > 0.5)}")
print(f"Moderate-Low Risk Patient - Predicted Probability:
{moderate_low_pred[0][0]:.4f}, Predicted Death Event:
{int(moderate_low_pred[0][0] > 0.5)}")
```

```
1/1 [==============================] - 0s 60ms/step
1/1 [==============================] - 0s 62ms/step
1/1 [==============================] - 0s 90ms/step
```

```
1/1 [==============================] - 0s 50ms/step
High Risk Patient - Predicted Probability: 0.7465, Predicted Death
Event: 1
Low Risk Patient - Predicted Probability: 0.2658, Predicted Death
Event: 0
Moderate-High Risk Patient - Predicted Probability: 0.7400, Predicted
Death Event: 1
Moderate-Low Risk Patient - Predicted Probability: 0.2834, Predicted
Death Event: 0
```