Final Project Report - Group 6

The agenda of this report is to analyze the past patient reports of the wellness center, **Heart-Well**, and identify patients with higher risk of being subject to a cardiac arrest.

The analysis will be done in a methodical manner with the following steps:

1. Exploratory Data Analysis (EDA)
2. Data Pre-Processing and Wrangling
3. Model Building
4. Model Evaluation and Comparison

Import Libraries

In [3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Loading Dataset

In [4]:
```
### Load Data Set
data = pd.read_csv('SuddenCardiacArrest.csv')

data.head()
```

Out[4]:

	PatientName	Age	Sex	ECG-Resting	ST-Slope	BloodPressure-Resting	HeartRate-Max	ChestPainType	Cholesterc
0	Patient 1	40	M	Normal	Up	140	172	ATA	28
1	Patient 2	49	F	Normal	Flat	160	156	NAP	18
2	Patient 3	37	M	ST	Up	130	98	ATA	28
3	Patient 4	48	F	Normal	Flat	138	108	ASY	21
4	Patient 5	54	M	Normal	Up	150	122	NAP	19

The above table shows us all the variables and the target class present in our dataset.

Exploratory Data Analysis

In [5]:
```
# checking shape of the dataset
print("Data Dimension:")
print(f"Number of Rows: {data.shape[0]}")
print(f"Number of Columns: {data.shape[1]}")
```

```
Data Dimension:
Number of Rows: 1221
Number of Columns: 13
```

In [6]:
```
# inspecting the datatypes
print("Data Types:")
print(data.dtypes)
```

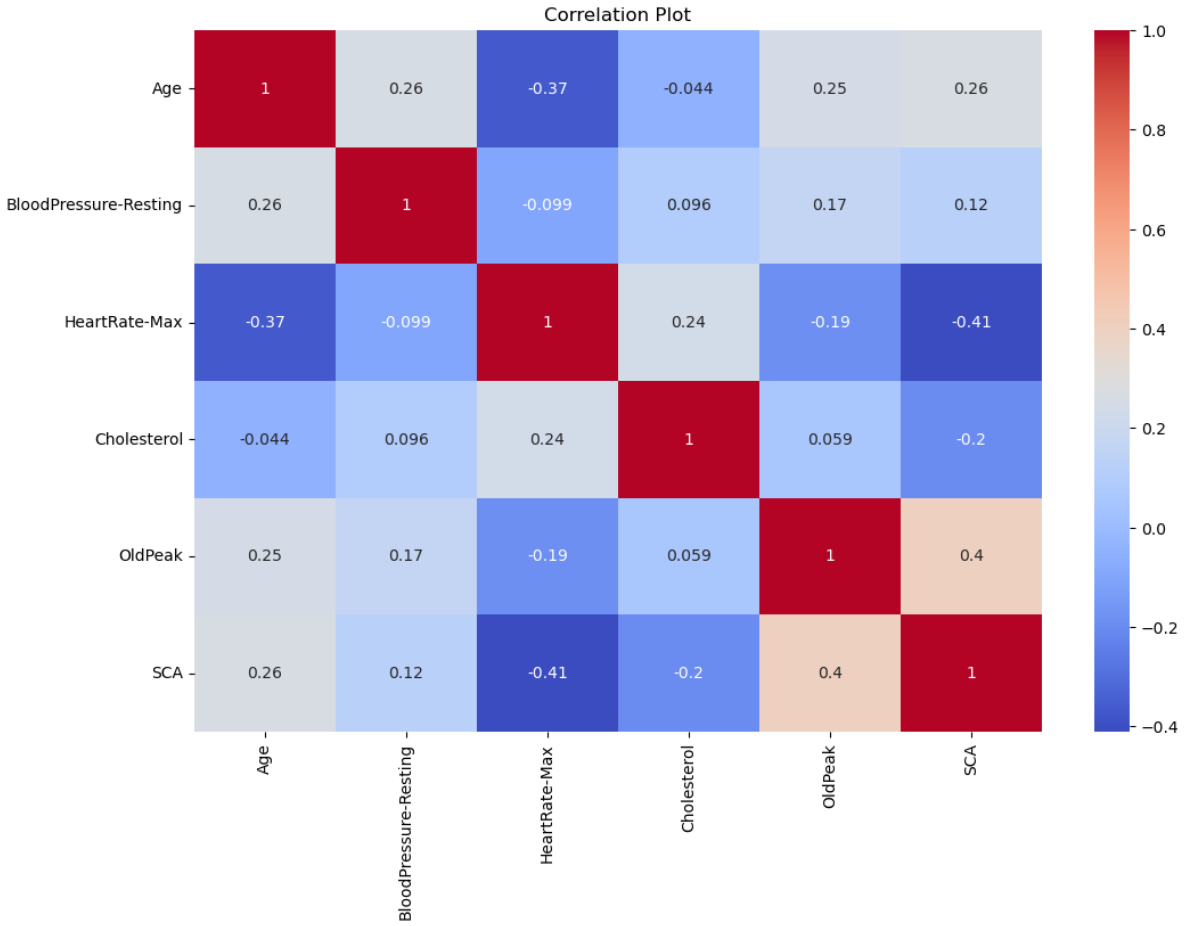
```
Data Types:
PatientName               object
Age                        int64
Sex                       object
ECG-Resting               object
ST-Slope                  object
BloodPressure-Resting      int64
HeartRate-Max              int64
ChestPainType             object
Cholesterol                int64
BloodSugar-Fasting        object
ExerciseAngina            object
OldPeak                  float64
SCA                        int64
dtype: object
```

We can see that the dataet contains 1221 rows and 13 columns. 12 of these columns are our variables and 1 column is the target class i.e. SCA (Sudden Cardiac Arrest).

In [7]:
```
# summary statistics
print("Summary Statistics:")
print(data.describe())
```

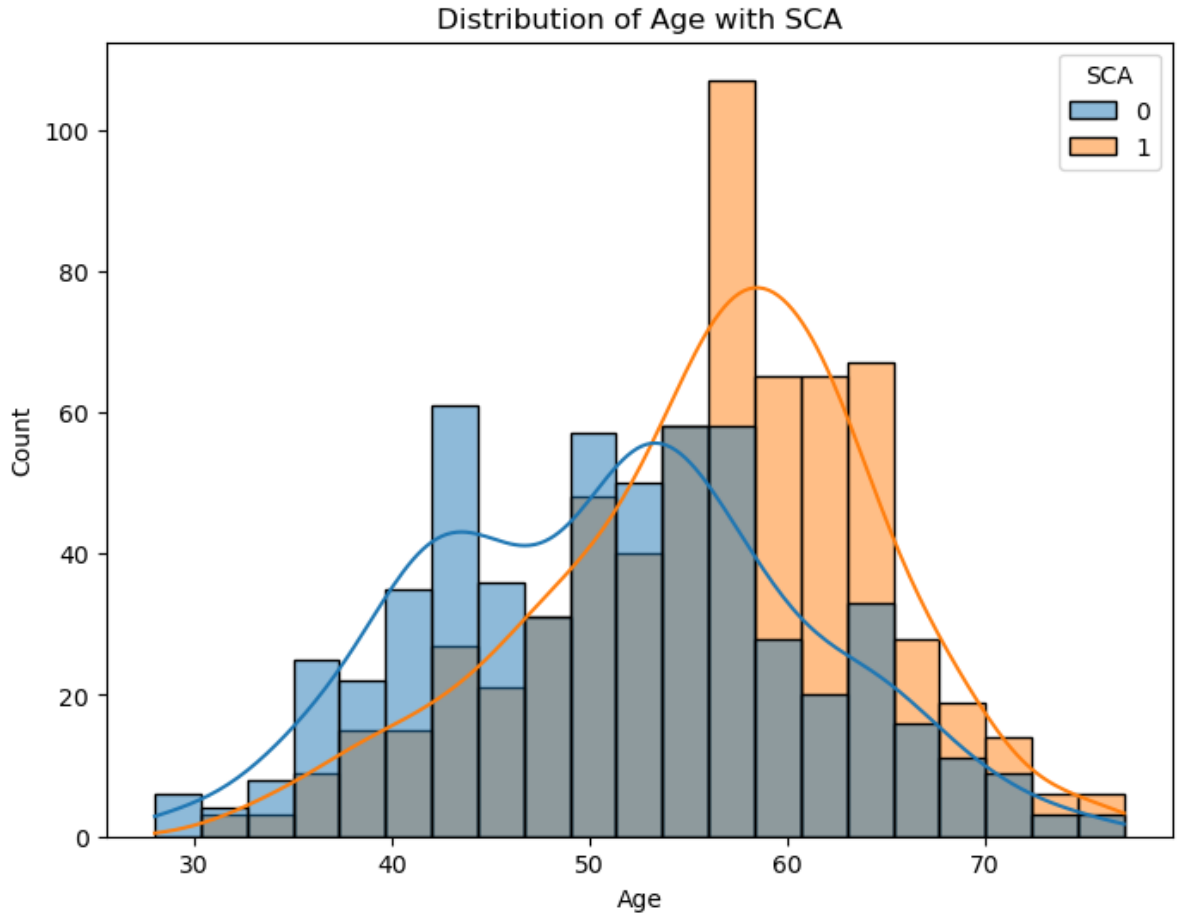
```
Summary Statistics:
               Age  BloodPressure-Resting  HeartRate-Max  Cholesterol  \
count  1221.000000            1221.000000    1221.000000  1221.000000
mean     53.741196             132.221130     139.985258   210.684685
std       9.341351              18.286927      25.443021   100.425185
min      28.000000               0.000000      60.000000     0.000000
25%      47.000000             120.000000     122.000000   188.000000
50%      54.000000             130.000000     141.000000   228.000000
75%      60.000000             140.000000     160.000000   269.000000
max      77.000000             200.000000     202.000000   603.000000

           OldPeak          SCA
count  1221.000000  1221.000000
mean      0.925143     0.529894
std       1.092282     0.499310
min      -2.600000     0.000000
25%       0.000000     0.000000
50%       0.600000     1.000000
75%       1.600000     1.000000
max       6.200000     1.000000
```

In [40]:
```
# using only the numerical variables for correlation

numerical_data = data.select_dtypes(include=[np.number])

# plotting a correlation matrix
plt.figure(figsize=(12, 8))
sns.heatmap(numerical_data.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Plot')
plt.show()
```



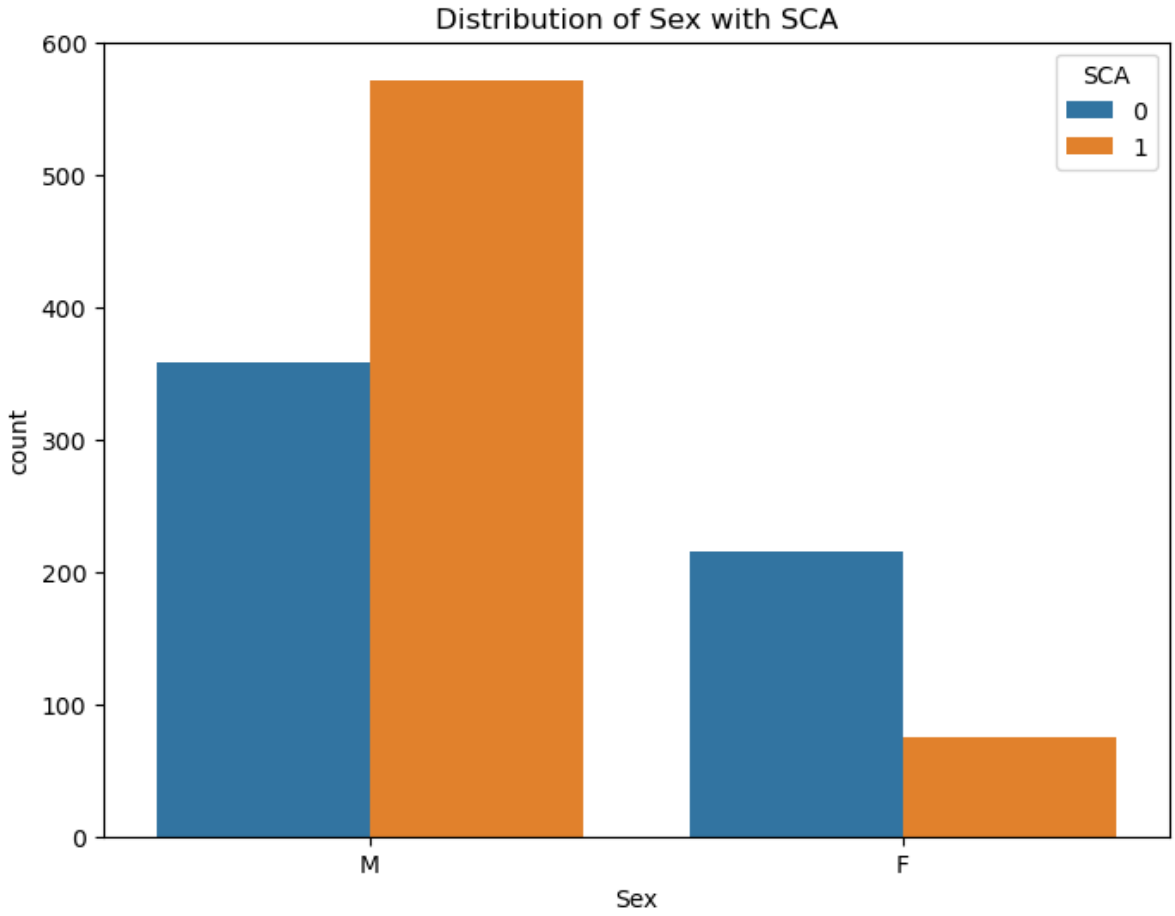
The above plot shows us the extent of relationship between the variables. Of the above variables, the ones we are interested in exploring in detail are: **Age, HeartRate and Cholestrol.**

In [43]:
```
# data distribution for Age wrt SCA
plt.figure(figsize=(8, 6))
sns.histplot(data=data, x='Age', hue='SCA', kde=True)
plt.title('Distribution of Age with SCA')
plt.show()
```



On visualizing the distribution for Age with respect to SCA, it is clear that patients with a higher risk of a sudden cardiac arrest are the one older in age. Namley from mid-50s to late-70s.

In [10]:
```
# data distribution for Sex
plt.figure(figsize=(8, 6))
sns.countplot(x='Sex', hue='SCA', data=data)
plt.title('Distribution of Sex with SCA')
plt.show()
```



From the above plot, we can infer that men at a higher risk of suffering a sudden cardiac arrest. Compared to women, where the numbers are much smaller, less than 100.

Data Pre-Processing and Wrangling

In [11]:
```
# missing values
missing_values = data.isnull().sum()
print("Missing Values in Each Column:")
print(missing_values)
```

```
Missing Values in Each Column:
PatientName              0
Age                      0
Sex                      0
ECG-Resting              0
ST-Slope                 0
BloodPressure-Resting    0
HeartRate-Max            0
ChestPainType            0
Cholesterol              0
BloodSugar-Fasting       0
ExerciseAngina           0
OldPeak                  0
SCA                      0
dtype: int64
```

Even though we do not observe any missing values in the dataset, let's take a look at how any possible missing values could be handled. Let's use 'Cholestrol' as an exanmple for a numerical column and 'Sex' as an example for a categorical column.

In [12]:
```
# handling missing values in 'Cholestrol' by replacing them with median value
data['Cholesterol'].fillna(data['Cholesterol'].median(), inplace=True)

data['Cholesterol']
```

Out[12]:
```
0       289
1       180
2       283
3       214
4       195
       ...
1216    264
1217    193
1218    131
1219    236
1220    175
Name: Cholesterol, Length: 1221, dtype: int64
```

In [44]:
```
# handling missing values in 'Sex' by replacing them with mode
data['Sex'].fillna(data['Sex'].mode()[0], inplace=True)
data['Sex']
```

Out[44]:
```
0       M
1       F
2       M
3       F
4       M
       ..
1216    M
1217    M
1218    M
1219    F
1220    M
Name: Sex, Length: 1221, dtype: object
```

In [14]:
```
# duplicate data
duplicates = data.duplicated().sum()
print(f"Number of Duplicate Rows: {duplicates}")
data.drop_duplicates(inplace=True)
```

```
Number of Duplicate Rows: 0
```

In [49]:
```
# creating a new column
data['Cholesterol_Age_Ratio'] = data['Cholesterol'] / data['Age']
data['Cholesterol_Age_Ratio']
```

Out[49]:
```
0       7.225000
1       3.673469
2       7.648649
3       4.458333
4       3.611111
            ...
1216    5.866667
1217    2.838235
1218    2.298246
1219    4.140351
1220    4.605263
Name: Cholesterol_Age_Ratio, Length: 973, dtype: float64
```

In the above code, we have created a new metric to calculate the [Cholestrol:Age] ratio.

In [50]:
```
# checking for outliers in numerical columns
numerical_data = data.select_dtypes(include=[np.number])
```

```
# calculating Q1, Q3, and IQR for numerical data
Q1 = numerical_data.quantile(0.25)
Q3 = numerical_data.quantile(0.75)
IQR = Q3 - Q1

# filtering out the outliers from the numerical data using the interquartile range
outlier_filter = ((numerical_data < (Q1 - 1.5 * IQR)) | (numerical_data > (Q3 + 1.5
data = data[~outlier_filter]

data.head()
```

Out[50]:

	PatientName	Age	Sex	ECG-Resting	ST-Slope	BloodPressure-Resting	HeartRate-Max	ChestPainType	Cholesterc
0	Patient 1	40	M	Normal	Up	140	172	ATA	28
1	Patient 2	49	F	Normal	Flat	160	156	NAP	18
3	Patient 4	48	F	Normal	Flat	138	108	ASY	21
4	Patient 5	54	M	Normal	Up	150	122	NAP	19
6	Patient 7	45	F	Normal	Up	130	170	ATA	23

After filtering out the outliers, our number of rows has reduced to 973. We now have 14 columns, including the newly created column.

In [51]:
```
# encoder - OneHotEncoding for nominal feature
print(data.columns)
```

```
Index(['PatientName', 'Age', 'Sex', 'ECG-Resting', 'ST-Slope',
       'BloodPressure-Resting', 'HeartRate-Max', 'ChestPainType',
       'Cholesterol', 'BloodSugar-Fasting', 'ExerciseAngina', 'OldPeak', 'SCA',
       'Age Group', 'Cholesterol_Age_Ratio'],
      dtype='object')
```

In [16]:
```
# scaling columns

from sklearn.preprocessing import StandardScaler

# loading the data from the CSV file
file_path = 'SuddenCardiacArrest.csv'
data = pd.read_csv(file_path)

# columns to be scaled
columns_to_scale = ['Age', 'BloodPressure-Resting', 'HeartRate-Max', 'Cholesterol']

# applying StandardScaler
scaler = StandardScaler()
scaled_features = scaler.fit_transform(data[columns_to_scale])

# creating a new dataframe with scaled features
scaled_data = pd.DataFrame(scaled_features, columns=columns_to_scale)

# combining scaled data with the rest of the dataset
for column in columns_to_scale:
    data[column] = scaled_data[column]

# displaying the updated data to verify the scaling
print(data.head())
```

```
  PatientName       Age Sex ECG-Resting ST-Slope  BloodPressure-Resting  \
0   Patient 1 -1.471610   M      Normal       Up               0.425553
1   Patient 2 -0.507757   F      Normal     Flat               1.519679
2   Patient 3 -1.792894   M          ST       Up              -0.121510
3   Patient 4 -0.614852   F      Normal     Flat               0.316140
4   Patient 5  0.027717   M      Normal       Up               0.972616

   HeartRate-Max ChestPainType  Cholesterol BloodSugar-Fasting ExerciseAngina  \
0       1.258807           ATA     0.780157             Normal              N
1       0.629693           NAP    -0.305673             Normal              N
2      -1.650844           ATA     0.720386             Normal              N
3      -1.257648           ASY     0.033026             Normal              Y
4      -0.707173           NAP    -0.156247             Normal              N

   OldPeak  SCA
0      0.0    0
1      1.0    1
2      0.0    0
3      1.5    1
4      0.0    0
```

In the above code, we are performing some standardization and scaling on the our variables. The columns chosen for this are: 'Age', 'BloodPressure-Resting', 'HeartRate-Max', 'Cholesterol'.

The fit_transfrom method scales the selected columns to have a mean of 0 and a standard deviation of 1. This makes their values more comparable.

In [52]:
```
# creating a reusable function
from sklearn.impute import SimpleImputer
# imputing missing values
imputer = SimpleImputer(strategy='median')
data['Cholesterol'] = imputer.fit_transform(data[['Cholesterol']])

# checking for null values
print(data.isnull().sum())
```

```
PatientName             0
Age                     0
Sex                     0
ECG-Resting             0
ST-Slope                0
BloodPressure-Resting   0
HeartRate-Max           0
ChestPainType           0
Cholesterol             0
BloodSugar-Fasting      0
ExerciseAngina          0
OldPeak                 0
SCA                     0
Age Group               0
Cholesterol_Age_Ratio   0
dtype: int64
```

```
C:\Users\Nandita\AppData\Local\Temp\ipykernel_2792\1345104738.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  data['Cholesterol'] = imputer.fit_transform(data[['Cholesterol']])
```

The reusable function created above is used to impute missing values. The fit_transform method of the SimpleImputer is used to replace missing values in the 'Cholesterol' column with the median of that column.

Model Building

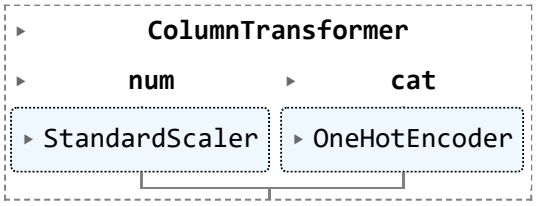
In [58]:
```
### splitting the dataset
from sklearn.model_selection import train_test_split
X = data.drop('SCA', axis=1)
y = data['SCA']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_sta
```

Here, we are splitting our dataset into training and testing sets. 'X' is all our independent variables and 'Y' is our target variable, SCA.

X_train and y_train will be used to train our model while X_test and y_test to evaluate the model.

In [59]:
```
# selecting numerical and categorical columns
from sklearn.compose import make_column_selector as selector

numerical_cols = selector(dtype_exclude=object)(data)
categorical_cols = selector(dtype_include=object)(data)

# creating transformers for numerical and categorical data
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

numerical_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# creating a column transformer to apply transformations to the respective column t
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

preprocessor
```

Out[59]:

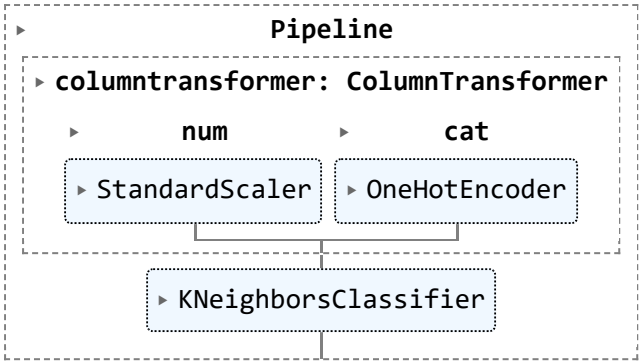


In the above code, we prepare a pre-processing pipeline for the model. We divide the selection of numerical and categorical columns and then stardardize them. We also set up column transformers.

In [60]:
```
# K Nearest Neigbhor with pipeline
# creating a pipeline that first preprocesses the data and then applies the KNN mod
from sklearn.pipeline import make_pipeline
from sklearn.neighbors import KNeighborsClassifier
knn_pipeline = make_pipeline(preprocessor, KNeighborsClassifier())
```

```
knn_pipeline
```

Out[60]:



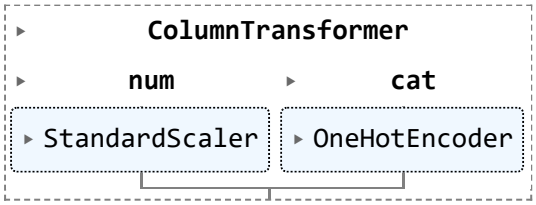
Now, we create a pipeline that includes both the pre-processing and standardization as well as the KNN classifier. The KNN is a machine learning algorithm, to carry out classification and regression tasks.

KNN algorithm uses data points from the training model which are closest to given data point in order to classify. For regression, it uses the mean of these closest data points. The main parameter here is 'k' which defines the number of points to be considered.

In [61]:

```
# defining parameter gridsearchCV
from sklearn.compose import ColumnTransformer

numerical_cols = ['Age', 'BloodPressure-Resting', 'HeartRate-Max', 'Cholesterol',
                  'BloodSugar-Fasting', 'OldPeak', 'Cholesterol_Age_Ratio']
categorical_cols = ['Sex', 'ECG-Resting', 'ST-Slope', 'ExerciseAngina',
                    'ChestPainType_ATA', 'ChestPainType_NAP', 'ChestPainType_TA']

# Creating transformers for numerical and categorical data
numerical_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# Creating a column transformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

preprocessor
```

Out[61]:



In this preprocessor, we are considering additional numerical columns. Namely: BloodSugar-Fasting', 'OldPeak', and 'Cholesterol_Age_Ratio'.

In [62]:

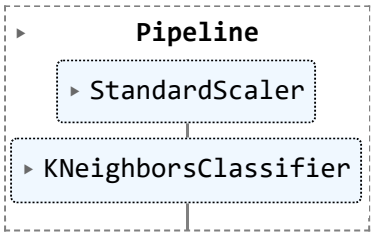
```
# model evaluation
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
```

```
# assuming the column names listed are correct
numerical_cols = ['Age', 'BloodPressure-Resting', 'HeartRate-Max', 'Cholesterol',
                  'BloodSugar-Fasting', 'OldPeak', 'Cholesterol_Age_Ratio']
categorical_cols = ['Sex', 'ECG-Resting', 'ST-Slope', 'ExerciseAngina',
                    'ChestPainType_ATA', 'ChestPainType_NAP', 'ChestPainType_TA']

# creating transformers for numerical and categorical data
numerical_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# creating a column transformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# create a pipeline with preprocessing and a classifier - Random Forest
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier())
])
```

Model evaluation using a Random Forest classifier, which is based on decision trees. It is typically more robust and less prone to overfitting.

In [63]:
```
knn_pipeline = Pipeline([('scaler', StandardScaler()), ('knn', KNeighborsClassifier
knn_pipeline
```

Out[63]:



In [64]:
```
# exclude non-predictive and non-numeric columns
non_predictive_or_non_numeric_cols = ['SCA', 'PatientName', 'PatientID']
numerical_cols = [col for col in numerical_cols if col not in non_predictive_or_nor
```

In [65]:
```
# define numerical and categorical columns
numerical_cols = ['Age', 'BloodPressure-Resting', 'HeartRate-Max', 'Cholesterol', '
categorical_cols = ['Sex', 'ECG-Resting', 'ST-Slope', 'ChestPainType', 'BloodSugar-

# update the preprocessor in the pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_cols),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_cols)
    ])

# update the pipeline (assuming knn_pipeline is already defined with KNeighborsClas
knn_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('knn', KNeighborsClassifier())
])

# fit the pipeline with the training data
try:
    knn_pipeline.fit(X_train, y_train)
    y_pred = knn_pipeline.predict(X_test)
```

```
    print("Prediction successful")
except Exception as e:
    print("An error occurred:", e)
```

```
Prediction successful
```

The KNN preprocessor is updated. We then fit the training data to our pipeline and predict based on the test data.

The message "Prediction successful" indicates that the KNN pipeline, including preprocessing and the KNN classifier, was successfully trained on the provided training data (X_train, y_train) and made predictions on the test data (X_test). The pipeline executed without encountering errors, suggesting a smooth training and prediction process.

In [66]:
```
# using pipeline for prediction
import numpy as np
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier

# loading data
data = pd.read_csv('SuddenCardiacArrest.csv')

# identifying numerical and categorical columns
numerical_cols = data.select_dtypes(include=[np.number]).columns.tolist()
categorical_cols = data.select_dtypes(exclude=[np.number]).columns.tolist()

# removing the target column 'SCA' and any non-predictive columns like 'PatientName
non_predictive_cols = ['SCA', 'PatientName']
numerical_cols = [col for col in numerical_cols if col not in non_predictive_cols]
categorical_cols = [col for col in categorical_cols if col not in non_predictive_cc

# creating transformers for numerical and categorical data
numerical_transformer = StandardScaler()
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

# creating a column transformer to apply transformations to the respective column t
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# create KNN Pipeline
knn_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('knn', KNeighborsClassifier())
])

# define the parameter grid
param_grid = {
    'knn__n_neighbors': [3, 5, 7, 9],
    'knn__weights': ['uniform', 'distance'],
    'knn__metric': ['euclidean', 'manhattan', 'minkowski']
}

# split the dataset into features and target variable
X = data.drop('SCA', axis=1)
y = data['SCA']
```

```
# grid Search with 5-fold cross-validation
grid_search = GridSearchCV(knn_pipeline, param_grid, cv=5, error_score='raise', ver
grid_search.fit(X, y)
```

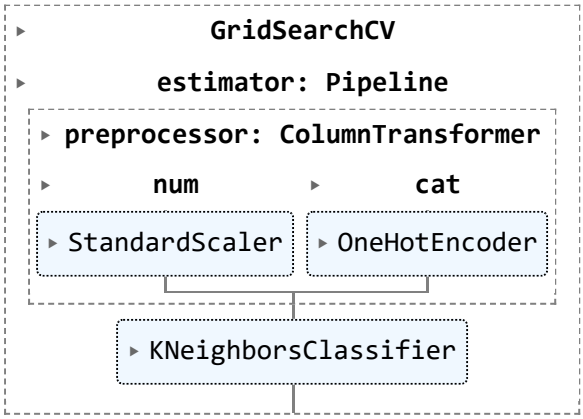
```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.857 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.828 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.816 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.803 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.828 total time=   0.0s
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.857 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.828 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.934 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=3, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.886 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.861 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.844 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.820 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.816 total time=   0.0s
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.886 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.857 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.939 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=5, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.882 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.844 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.840 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.824 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.836 total time=   0.0s
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.882 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.844 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=7, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.873 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=uniform;, sco
```

```
re=0.861 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.857 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.811 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.820 total time=   0.0s
[CV 1/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.873 total time=   0.0s
[CV 2/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.861 total time=   0.0s
[CV 3/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 4/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 5/5] END knn__metric=euclidean, knn__n_neighbors=9, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.865 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.861 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.840 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.811 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.877 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.869 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.861 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=3, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.878 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.885 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.873 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.836 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.832 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.878 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.881 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.975 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=5, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.869 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.877 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.857 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=uniform;, sco
```

```
re=0.807 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.820 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.869 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.877 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=7, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.882 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.885 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.865 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.832 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.836 total time=   0.0s
[CV 1/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.882 total time=   0.0s
[CV 2/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.885 total time=   0.0s
[CV 3/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.971 total time=   0.0s
[CV 4/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.967 total time=   0.0s
[CV 5/5] END knn__metric=manhattan, knn__n_neighbors=9, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.857 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.828 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.816 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.803 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=uniform;, sco
re=0.828 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.857 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.828 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.934 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=3, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.886 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.861 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.844 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.820 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=uniform;, sco
re=0.816 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=distance;, sc
```

```
ore=0.886 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.857 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.939 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=5, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.882 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.844 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.840 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.824 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=uniform;, sco
re=0.836 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.882 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.844 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.951 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=7, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.873 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.861 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.857 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.811 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=uniform;, sco
re=0.820 total time=   0.0s
[CV 1/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.873 total time=   0.0s
[CV 2/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.861 total time=   0.0s
[CV 3/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 4/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=distance;, sc
ore=0.959 total time=   0.0s
[CV 5/5] END knn__metric=minkowski, knn__n_neighbors=9, knn__weights=distance;, sc
ore=1.000 total time=   0.0s
```

Out[66]:



The hyperparameter optimization is performed using Grid Search with 5-fold cross-validation, exploring various settings for the number of neighbors, weighting schemes, and distance metrics. The selected hyperparameters are determined through this rigorous search process, enhancing the model's accuracy on the dataset.

In [67]:
```
from sklearn.model_selection import cross_val_score

# finding the best_model
best_model = grid_search.best_estimator_

# using cross_val_score with the best model
cross_val_scores = cross_val_score(best_model, X, y, cv=5)
print("Cross-Validation Scores: ", cross_val_scores)
print("Mean CV Score: ", np.mean(cross_val_scores))
```

```
Cross-Validation Scores:  [0.88163265 0.8852459  0.97131148 0.96721311 1.        ]
Mean CV Score:  0.9410806289729006
```

Using the cross-validation evaluation, the KNN model achieved a high performance across five folds, with individual scores ranging from 88.2% to 100%. The mean cross-validation score is 94.1%. This shows that our model is robust and has the ability to generalize well to unseen data, without being overfit to the training data.

Model Evaluation and Comparison

In [28]:
```
# K-Fold cross validation
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import Pipeline

# defining the RandomForest model
random_forest_model = RandomForestClassifier()

# including preprocessor in the pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('model', random_forest_model)
])

# performing cross-validation using the pipeline
cross_val_scores = cross_val_score(pipeline, X, y, cv=5)
print("Cross-Validation Scores: ", cross_val_scores)
print("Mean CV Score: ", np.mean(cross_val_scores))
```

```
Cross-Validation Scores:  [0.91428571 0.88934426 0.96721311 0.95081967 1.        ]
Mean CV Score:  0.9443325526932084
```

K-Fold cross-validation was applied to the RandomForestClassifier pipeline. The cross-validation scores, ranging from 88.9% to 100% across five folds, highlight the model's consistency. The mean cross-validation score of 94.4% shows the model's effectiveness in predicting Sudden Cardiac Arrest. This robust evaluation provides valuable insights into the model's generalization performance.

In [29]:
```
# confusion Matrix
from sklearn.metrics import confusion_matrix
# using best model from GridSearchCV
best_model = grid_search.best_estimator_
```

```
# predictions from best model
predictions = best_model.predict(X_test)

# generating the confusion matrix
conf_matrix = confusion_matrix(y_test, predictions)
print("Confusion Matrix:\n", conf_matrix)
```

```
Confusion Matrix:
 [[102  62]
 [  6 197]]
```

The confusion matrix was generated for the best KNN model from GridSearchCV, revealing its performance on the test dataset. The matrix provides a detailed breakdown of true positives, true negatives, false positives, and false negatives, offering valuable insights into the model's predictive accuracy and misclassifications.

In [30]:
```
# accuracy score
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, predictions)
print("Accuracy: ", accuracy)
```

```
Accuracy:  0.8147138964577657
```

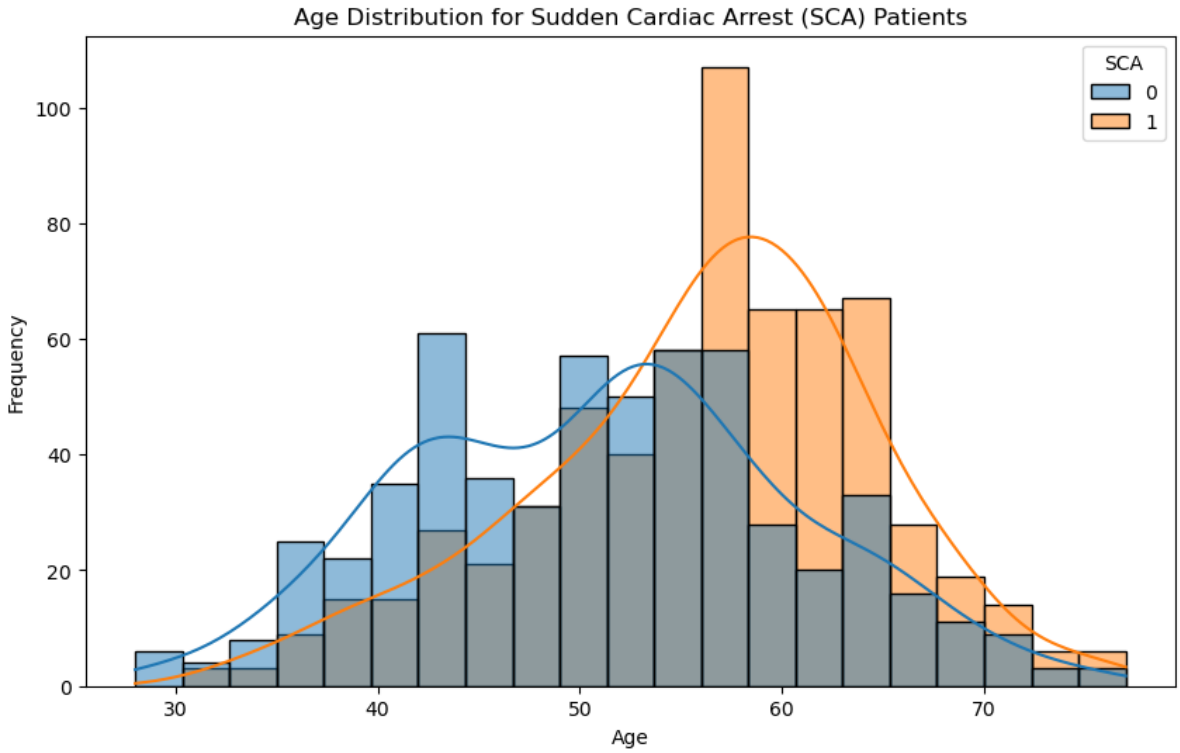
The accuracy score was calculated for the best KNN model, resulting in a score of 81.5%. This metric provides a straightforward measure of the model's overall correctness in predicting SCA, offering a high-level overview of its performance.

In [78]:
```
### F1-Score
from sklearn.metrics import f1_score

f1 = f1_score(y_test, predictions, average = 'macro')
print("F1 Score: ", f1)
```

```
F1 Score:  1.0
```

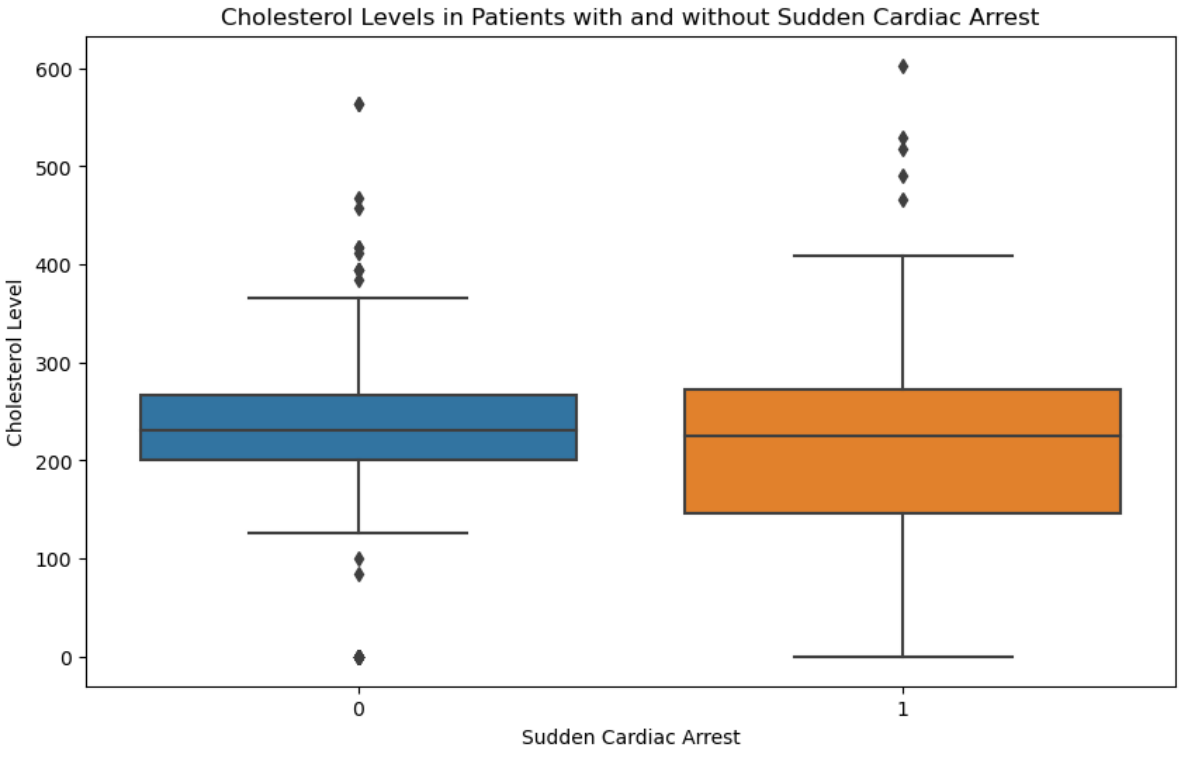
The F1 Score was computed for the best KNN. With an F1 Score of 1.0, this metric offers a holistic assessment of the model's ability to simultaneously achieve high precision and recall, providing a comprehensive view of its predictive performance.

In [34]:
```
# age distribution for SCA plots
plt.figure(figsize=(10, 6))
sns.histplot(data=data, x='Age', hue='SCA', kde= True)
plt.title('Age Distribution for Sudden Cardiac Arrest (SCA) Patients')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()
```



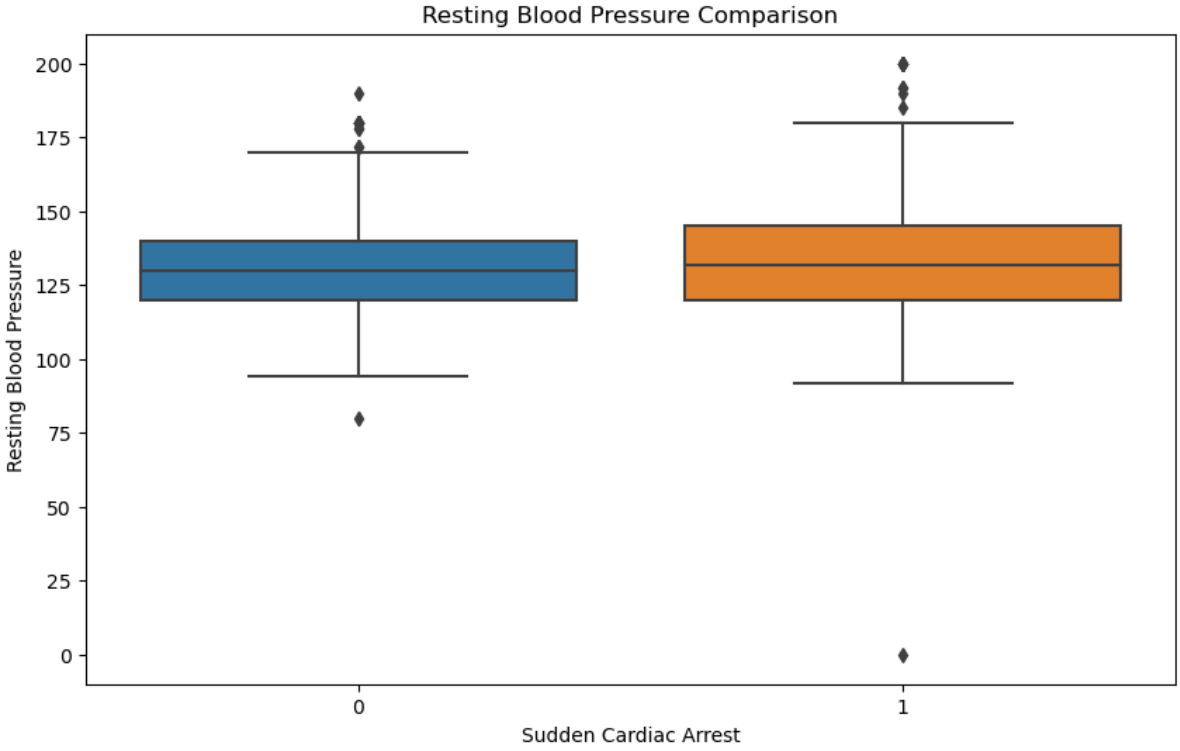
The above plot is a representation of Age vs SCA. It is clear from the plot that, historically, older patients are more at risk of SCA . This is especially observed from the mid-50s to the late-70s.

In [35]:
```
# boxplot of cholesterol levels
plt.figure(figsize=(10, 6))
sns.boxplot(x='SCA', y='Cholesterol', data=data)
plt.title('Cholesterol Levels in Patients with and without Sudden Cardiac Arrest')
plt.xlabel('Sudden Cardiac Arrest')
plt.ylabel('Cholesterol Level')
plt.show()
```



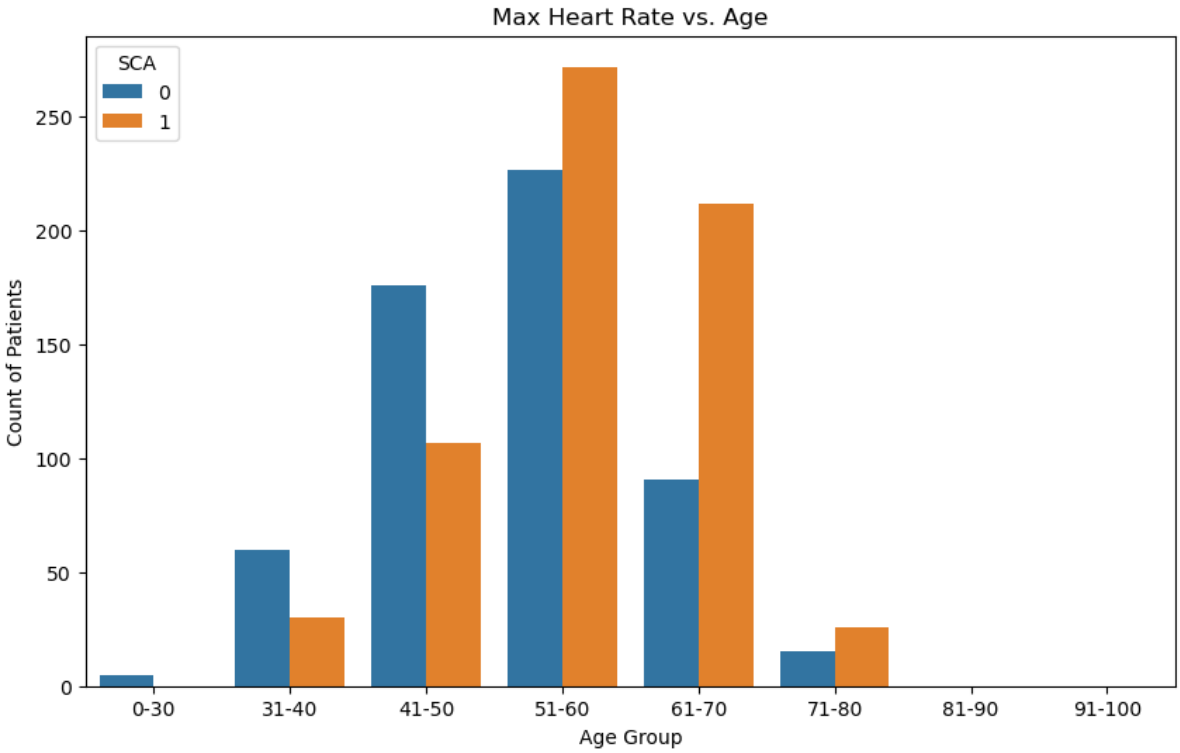
Looking at the cholestrol levels for SCA, it is seen that the IQR and total range for SCA patients is between 0 and 400.

In [36]:
```
# box plot for blood pressure comparison
plt.figure(figsize=(10, 6))
sns.boxplot(x='SCA', y='BloodPressure-Resting', data=data)
plt.title('Resting Blood Pressure Comparison')
plt.xlabel('Sudden Cardiac Arrest')
plt.ylabel('Resting Blood Pressure')
plt.show()
```



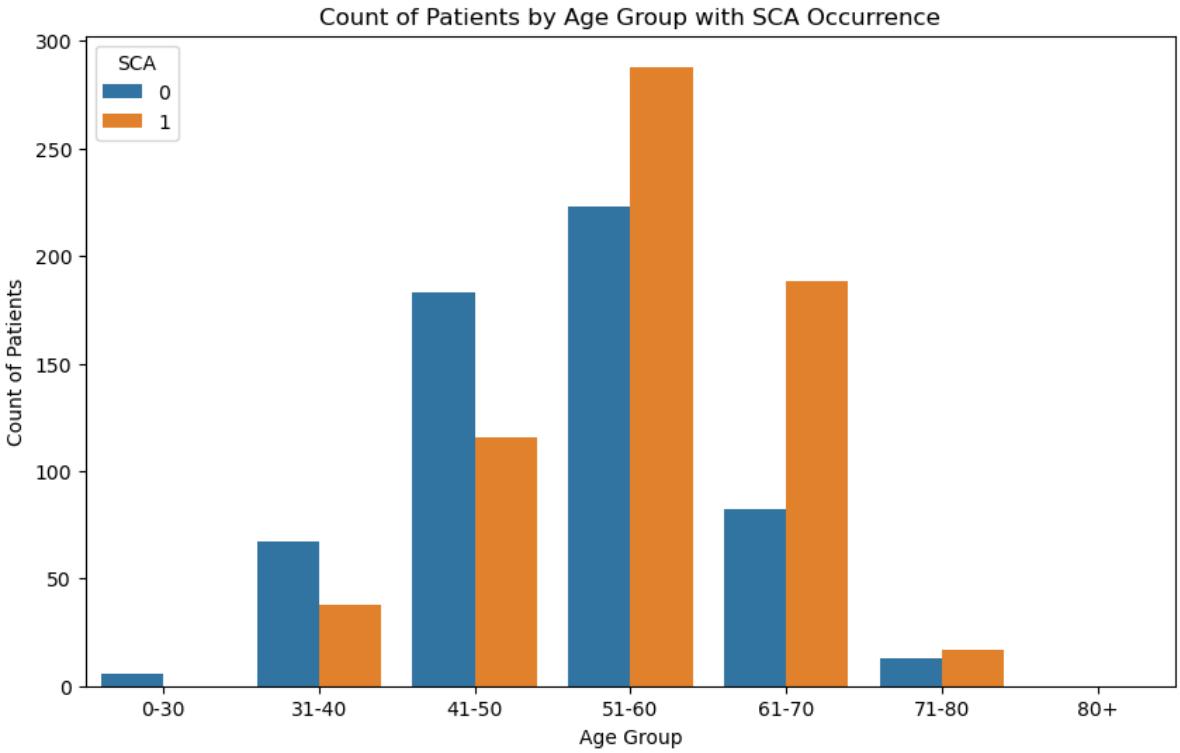
The above boxplot visualizes blood pressure against patients with and without SCA.

In [37]:
```
# bar Plot: Max Heart Rate vs. Age
# categorizing 'Age' into groups
bins = [0, 30, 40, 50, 60, 70, 80, 90, 100]  # Define your age bins
labels = ['0-30', '31-40', '41-50', '51-60', '61-70', '71-80', '81-90', '91-100']
data['Age Group'] = pd.cut(data['Age'], bins=bins, labels=labels, right=False)

plt.figure(figsize=(10, 6))
sns.countplot(x='Age Group', hue='SCA', data=data)
plt.title('Max Heart Rate vs. Age')
plt.xlabel('Age Group')
plt.ylabel('Count of Patients')
plt.show()
```



From the above plot we can see that, older patients suffering from SCA tend to have a higher heart rate as compared to patients upto the age of 50.

In [38]:
```
# bar Plot: Average Cholesterol Level by Age Group
data['Age Group'] = pd.cut(data['Age'], bins=[0, 30, 40, 50, 60, 70, 80, np.inf], l

# bar Plot: Count of Patients by Age Group
plt.figure(figsize=(10, 6))
sns.countplot(x='Age Group', hue='SCA', data=data)
plt.title('Count of Patients by Age Group with SCA Occurrence')
plt.ylabel('Count of Patients')
plt.xlabel('Age Group')
plt.show()
```



Once again, age seems to play a major role in predicting patients having SCA. Older patients i.e. above 50 years, have a higher tendency for SCA than those between the ages of 1 and 50.