Data mining for HeartBit dataset

Steps

1. Exploratory Data Analysis 1.1. Understanding the problem 1.2. Getting to know the data 1.3. Prepare the data for training (training intermediate model)
2. Training and improving the model and evaluating the model

1. Exploratory Data Analysis

1.1. Understanding the problem

The goal of this project is to predict the NYHA class of a patient based on the patient's data. The NYHA (New York Heart Association) class is the functional classification system for the patient with heart failure in stage C or D. The classification is related to the patient symptoms and physical activity.

The NYHA class is divided into four classes:

- Class I: No limitation of physical activity. Ordinary physical activity does not cause undue fatigue, palpitation or shortness of breath.
- Class II: Slight limitation of physical activity. Comfortable at rest. Ordinary physical activity results in fatigue, palpitation or shortness of breath.
- Class III: Marked limitation of physical activity. Comfortable at rest. Less than ordinary activity causes fatigue, palpitation or shortness of breath.
- Class IV: Unable to carry out any physical activity without discomfort. Symptoms of heart failure at rest. If any physical activity is undertaken, discomfort is increased.

1.2. Getting to know the data

For this project, I will be using the HeartBit dataset. The dataset contains values of 63 variables recorded for 469 patients. The variables can be grouped into following groups:

- Clinical
- Technical
- Demographic
- Anthropometry (pomiary antropometryczne - np. wzrost, waga)
- Comorbidities (choroby współistniejące)
- Treatment
- Biochemistry
- Fitness Level

The variables

Additional variable number	Variable name	Variable description	Character of data	Category of variable
VAR1	DEATH?	information if the patient is death (1) or alive (0)	binary	Clinical
VAR2	DEATHDATE	date of death (if death=1) or date of the confirmation that the patient is still alive	date	Technical
VAR3	TIMEFU	number of days between examination and date death or date of the confirmation that the patient is still alive	number	Technical
VAR4	QOL	result of the survey measuring the quality of life (QoL, total score range 0–105, from best to worst)	number	Clinical
VAR5	OQLsub1	scores for a QoL subscale - physical dimension (8 items, range 0–40 from best to worst)	number	Clinical
VAR6	OQLsub2	scores for a QoL subscale - emotional dimension (5 items, range 0–25 from best to worst)	number	Clinical
VAR7	DOB	date of birth	date	Technical
VAR8	DOE	date of the examination	date	Technical
VAR9	AGE	age at examination	number	Demographic
VAR10	HEIGHT.CM	body height in cm	number	Anthropometry
VAR11	WEIGHT.KG	body mass in kg	number	Anthropometry
VAR12	BMI	body mass index (mass per squared height); <18.5=underweight range, 18.5 to <25=normal, 25-<30=overweight, 30.0 or higher=obese	number	Anthropometry
VAR13	LVEF.0	left ventricular ejection fraction - information from the heart ultrasound reflecting the efficiency of pumping	number	Clinical
VAR14	NYHA	NYHA Classification - The Stages of Heart Failure	category	Clinical

Additional variable number	Variable name	Variable description	Character of data	Category of variable
VAR15	PM	information about artificial pacemaker (0=no pacemaker)	binary	Clinical
VAR16	AETH.HF	information about the clinical cause of heart failure (1=ischemic disease or 2=other)	binary	Clinical
VAR17	MI	information about previous myocardial infarction (1=yes)	binary	Comorbidities
VAR18	AF	information about atrial fibrillation (1=yes)	binary	Comorbidities
VAR19	DM	information about diabetes (1=yes)	binary	Comorbidities
VAR20	HT	information about hypertension (1=yes)	binary	Comorbidities
VAR21	COPD	information about lung disease (1=yes)	binary	Comorbidities
VAR22	STROKE	information about previous stroke (1=yes)	binary	Comorbidities
VAR23	KIDNEY.DIS	information about kidney disease (1=yes)	binary	Comorbidities
VAR24	ACEI.ARB	information about treatment using ace-inhibitors or ARB (similar drugs, 1=yes)	binary	Treatment
VAR25	BB	information about treatment using beta blockers (1=yes)	binary	Treatment
VAR26	MRA	information about treatment using aldosterone antagonists (1=yes)	binary	Treatment
VAR27	DIUR	information about treatment using oral diuretics (1=yes)	binary	Treatment
VAR28	ANTIPLAT	information about treatment using antiplatelet drugs (1=yes)	binary	Treatment
VAR29	STATIN	information about treatment using statin (1=yes)	binary	Treatment
VAR30	DIGOX	information about treatment using digoxin (1=yes)	binary	Treatment
VAR31	HB	level of blood hemoglobin	number	Biochemistry

Additional variable number	Variable name	Variable description	Character of data	Category of variable
VAR32	NA	level of blood sodium	number	Biochemistry
VAR33	K	level of blood potassium	number	Biochemistry
VAR34	BNP	level of blood peptide: BNP (elevated level is characteristic for heart failure)	number	Biochemistry
VAR35	CRP	level of blood protein CRP (characteristic for inflammation)	number	Biochemistry
VAR36	LVEDD	parameter from heart ultrasound: left ventricular end diastolic diameter (increased in heart failure)	number	Clinical
VAR37	MR	Mitral regurgitation (valvular heart disease), bigger number = worse	categories	Clinical
VAR38	REST.SBP	systolic blood pressure at rest	number	Clinical
VAR39	REST.DBP	diastolic blood pressure at rest	number	Clinical
VAR40	REST.HR	heart rate at rest	number	Clinical
VAR41	EXERCISE1	EXERCISE 1: number of seconds needed to complete the task, higher number = worse)	number	Fitness level
VAR42	EXERCISE2	EXERCISE 2: number of repeated movements during the task (higher number = better result)	number	Fitness level
VAR43	EXERCISE3	EXERCISE 3: number of repeated movements during the task (higher number = better result)	number	Fitness level
VAR44	6MWT.DIST	number of meters (distance) covered by the patient during the 6 minute walking test (bigger number = better)	number	Fitness level
VAR45	6MWT.FATIGUE	the level of fatigue assessed by the patient after the walking test (using a scale 0-10)	number OR categories	Fitness level

Additional variable number	Variable name	Variable description	Character of data	Category of variable
VAR46	6MWT.DYSPN	the level of shortness of breath during walking test assessed by the patient (using a scale 0-10)	number OR categories	Fitness level
VAR47	6MWT.SBP1	systolic blood pressure before the walking test	number	Fitness level
VAR48	6MWT.DBP1	diastolic blood pressure before the walking test	number	Fitness level
VAR49	6MWT.HR1	heart rate before the walking test	number	Fitness level
VAR50	6MWT.SBP2	systolic blood pressure after the walking test	number	Fitness level
VAR51	6MWT.DBP2	diastolic blood pressure after the walking test	number	Fitness level
VAR52	6MWT.HR2	heart rate after the walking test	number	Fitness level
VAR53	EXERCISE4	EXERCISE4: the patient is asked to touch his/her feet with the fingers of the palms. The number represents centimeters	number	Fitness level
VAR54	EXERCISE5	EXERCISE5: the patient is asked to touch left palm with the right one but it has to be done at his back. The number represents centimeters	number	Fitness level
VAR55	CPX.TIME	time of exercise on a treadmill	number	Fitness level
VAR56	CPX.PEAKVO2	peak oxygen consumption during exercise testing on a treadmill	number	Fitness level
VAR57	CPX.PEAKVO2FORBM	peak oxygen consumption during exercise testing on a treadmill per body mass	number	Fitness level
VAR58	RER	respiratory exchange ratio (index of metabolism during an exercise on a treadmill)	number	Fitness level
VAR59	SLOPE	slope between oxygen and carbon dioxide during exercise testing on a treadmill	number	Fitness level

Additional variable number	Variable name	Variable description	Character of data	Category of variable
VAR60	METS	number of metabolic equivalents (level of work performed by the patient during exercise using a treadmill)	number	Fitness level
VAR61	WEBER	The Weber classification: stratification of patients based on peak VO2 and anaerobic threshold to define functional physical capacity. Higher class is worse	categories	Fitness level
VAR62	PEAK>18	Dividing patients based on a cutoff value of peak oxygen consumption (used for Weber)	binary	Fitness level
VAR63	SLOPE>35	Dividing patients based on a cutoff value of slope (used for Weber)	binary	Fitness level

NOTES:

- Sprawdzić dominujące wartości w kolumnach
- Zapoznanie się z działaniem drzewa decyzyjnego z wykorzystaniem biblioteki scikit-learn (https://scikit-learn.org/stable/modules/tree.html)
- Zamiana atrybutów nominalnych na numeryczne (https://scikit-learn.org/stable/modules/preprocessing.html#encoding-categorical-features)
- Redukcja wielowymiarowości (https://scikit-learn.org/stable/modules/decomposition.html#decompositions)
- Wyważanie klas ze względu na niezbalansowane dane, aby zapobiec tendencyjności modelu w kierunku klasy dominującej - ustawienie class_weight='balanced' (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)
- przycinanie drzewa decyzyjnego
- scikit-learn wykorzystuje zoptymalizowaną wersję algorytmu CART (Classification and Regression Trees) i nie obsługuje obecnie zmiennych kategorycznych

1.3. Prepare data for analysis

During the data preparation phase, we will perform the following steps:

- Load data

- Remove unnecessary columns
- Normalize NYHA target values to 0-4 range
- Divide attributes into groups
- Create the model for each group of attributes
- Analysis of each group of attributes to find the most important ones

---

In [9]:
```
# Imports
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score, KFold
from sklearn.tree import DecisionTreeClassifier
```

In [10]:
```
# Load data variables names and additional variable numbers
data_headers = pd.read_excel("data/heartbit.xlsx", nrows=1)
# Drop unnecessary CODES column
data_headers = data_headers.drop(columns=['CODES'])
data_headers.head()
data_headers["VAR32"][0] = "NA"
data_headers.head()
```

```
C:\Users\Krystian\AppData\Local\Temp\ipykernel_24696\2532535120.py:6: SettingWithCop
yWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/u
ser_guide/indexing.html#returning-a-view-versus-a-copy
  data_headers["VAR32"][0] = "NA"
```

Out[10]:

	VAR1	VAR2	VAR3	VAR4	VAR5	VAR6	VAR7	VAR8	VAR9	VAR
0	DEATH?	DEATHDATE	TIMEFU	QOL	OQLsub1	OQLsub2	DOB	DOE	AGE	HEIGHT.C

1 rows × 63 columns

In [11]:
```
# Load data
data = pd.read_excel("data/heartbit.xlsx", header=1)

# Remove unnecessary CODES column
data = data.drop(columns=['ID CODES '])

# First normalize NYHA target values to 0-4 range. Replace 1.5, 2.5, 3.5 with 1, 2,
data['NYHA'] = data['NYHA'].replace([1.5, 2.5, 3.5], [1, 2, 4])
print(f"NYHA missing values: {data['NYHA'].isnull().sum()}")
data = data.dropna(subset=['NYHA'])
size_of_data = data.shape
print(f"Data size: {size_of_data}")
NYHA = data['NYHA']
# Calculate size of datas per target value
print(NYHA.value_counts())
```

```
data.head()
```

```
NYHA missing values: 9
Data size: (460, 63)
NYHA
2.0    238
3.0    125
1.0     85
4.0     12
Name: count, dtype: int64
```

Out[11]:

	DEATH?	DEATHDATE	TIMEFU	QOL	OQLsub1	OQLsub2	DOB	DOE	AGE	HE
0	0.0	2012-02-03	2076	27.0	23.0	4.0	1959-06-14	2006-05-29	46.989041	
1	0.0	2012-08-12	2316	42.0	37.0	5.0	1958-12-22	2006-04-10	47.331507	
2	0.0	2012-02-21	2349	0.0	0.0	0.0	1945-11-25	2005-09-16	59.849315	
3	0.0	2013-02-03	2459	10.0	7.0	3.0	1945-03-20	2006-05-12	61.186301	
4	0.0	2013-02-03	2629	4.0	2.0	2.0	1982-03-07	2005-11-23	23.731507	

5 rows × 63 columns

In [12]:
```
# Check missing values
data.isnull().sum()
# Print all columns with missing values in descending order
data.isnull().sum().sort_values(ascending=False)
```

Out[12]:
```
OQLsub1        232
OQLsub2        232
CPX.PEAKVO2    213
QOL            209
METS           199
              ...
NYHA             0
WEIGHT.KG        0
HEIGHT.CM        0
BMI              0
DOE              0
Length: 63, dtype: int64
```

In [13]:
```
groups = {
    'clinical': ['VAR1', 'VAR4', 'VAR5', 'VAR6', 'VAR13', 'VAR14', 'VAR15', 'VAR16'
                 'VAR40'],
    'technical': ['VAR2', 'VAR3', 'VAR7', 'VAR8'],
    'demographic': ['VAR9'],
    'anthropometry': ['VAR10', 'VAR11', 'VAR12'],
    'comorbidities': ['VAR17', 'VAR18', 'VAR19', 'VAR20', 'VAR21', 'VAR22', 'VAR23'
    'treatment': ['VAR24', 'VAR25', 'VAR26', 'VAR27', 'VAR28', 'VAR29', 'VAR30'],
```

```
    'biochemistry': ['VAR31', 'VAR32', 'VAR33', 'VAR34', 'VAR35'],
    'fitness_level': ['VAR41', 'VAR42', 'VAR43', 'VAR44', 'VAR45', 'VAR46', 'VAR47'
                      'VAR52', 'VAR53', 'VAR54', 'VAR55', 'VAR56', 'VAR57', 'VAR58'
                      'VAR63']
}

# Replace variable numbers with variable names in groups
for group_name, list_of_variables in groups.items():
    for variable in list_of_variables:
        groups[group_name][list_of_variables.index(variable)] = data_headers[variab
```

Grouping variables

Before I start analyzing data I want to group variables into categories. I will use this grouping later to analyze correlation between variables and to select variables for training the model. I will also use this grouping to analyze correlation between variables and target variable.

Helper functions

In [14]:
```
param_grid = {
    'class_weight': ['balanced', None],
    'criterion': ['gini', 'entropy'],
    'max_depth': [2, 3, 4, 5, 6, 7, 8, 9, 10],
    'min_samples_split': [2, 3, 4, 5, 6, 7, 8, 9, 10],
    'min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8, 9],
}


def plot_corr_matrix(data, title, size):
    corr_matrix = data.corr()

    # Plot correlation chart
    plt.figure(figsize=size)
    sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
    plt.title(f'Correlation matrix for {title}')
    plt.show()


def plot_boxplot(data, title, size, x='NYHA', y=None):
    plt.figure(figsize=size)
    sns.boxplot(x=x, y=y, data=data)
    plt.title(f'Boxplot for {title}')
    plt.show()


def prepare_group_checking(group):
    corr_matrix = group.corr()
    group_checking = {}
    for column in group.columns:
        if column == 'NYHA':
            continue
```

```
        group_checking[column] = [abs(corr_matrix[column]['NYHA']),
                                  group[column].isnull().sum()]

    return group_checking


def add_importance_to_group_checking(group_checking, feature_importances, X):
    for column in group_checking.keys():
        if column == 'NYHA':
            continue

        group_checking[column].append(feature_importances[X.columns.get_loc(column)

    return group_checking


def convert_group_checking_to_dataframe(group_checking, by='correlation', ascending
                                        columns=['correlation', 'missing_values', '
    group_checking_df = pd.DataFrame.from_dict(group_checking, orient='index', colu
    group_checking_df = group_checking_df.sort_values(by=[by], ascending=ascending)

    return group_checking_df


def print_feature_importances(feature_importances, X):
    feature_importances_df = pd.DataFrame({'feature': X.columns, 'importance': feat
    feature_importances_df = feature_importances_df.sort_values(by=['importance'],
    print(feature_importances_df)


def create_grid_search(model, X, y, parameters=param_grid, cv=10):
    # cross validation score before grid search
    before_grid_search = cross_val_score(model, X, y, cv=KFold(n_splits=cv, shuffle
    print(f"Cross validation score before grid search: {before_grid_search}")

    grid_search = GridSearchCV(model, parameters, cv=cv, scoring='accuracy', return
    grid_search.fit(X, y)
    # The best parameters
    print(f"Best parameters: {grid_search.best_params_}")
    # The best score
    print(f"Best score: {grid_search.best_score_}")

    return grid_search, before_grid_search


def test_with_cross_validation(model, X, y, cv=10, before_grid_search=0):
    # cross validation score after grid search
    after_grid_search = cross_val_score(model, X, y, cv=KFold(n_splits=cv, shuffle=
    print(f"Cross validation score after grid search: {after_grid_search}")
    print(f"Mean cross validation score after optimize: {after_grid_search.mean()}"
    print(f"Growth of cross validation score: {after_grid_search.mean() - before_gr
    return after_grid_search
```

Clinical group

Clinical group description

The clinical group contains variables that provide information related to the clinical aspects of patients' health conditions.

Clinical group variables

- **DEATH?** [VAR1] - binary - information if the patient is death (1) or alive (0)
- **QOL** [VAR4] - number - result of the survey measuring the quality of life (QoL, total score range 0–105, from best to worst)
- **OQLsub1** [VAR5] - number - scores for a QoL subscale - physical dimension (8 items, range 0–40 from best to worst)
- **OQLsub2** [VAR6] - number - scores for a QoL subscale - emotional dimension (5 items, range 0–25 from best to worst)
- **LVEF.0** [VAR13] - number - left ventricular ejection fraction - information from the heart ultrasound reflecting the efficiency of pumping. According to the definition value = 45 or lower is characteristic for systolic heart failure
- **NYHA** [VAR14] - category - NYHA Classification
- **PM** [VAR15] - binary - information about artificial pacemaker (0=no pacemaker)
- **AETH.HF** [VAR16] - binary - information about the clinical cause of heart failure (1=ischemic disease or 2 = other)
- **LVEDD** [VAR36] - number - parameter from heart ultrasound: left ventricular end diastolic diameter (increased in heart failure)
- **MR** [VAR37] - categories - Mitral regurgitation (valvular heart disease), bigger number = worse
- **REST.SBP** [VAR38] - number - systolic blood pressure at rest
- **REST.DBP** [VAR39] - number - diastolic blood pressure at rest
- **REST.HR** [VAR40] - number - heart rate at rest

Data preparation

In [15]:
```
# Create clinical group
clinical_group = data[groups['clinical']].copy()
print(f"Clinical group columns: {clinical_group.columns}")
print(f"Missing values in clinical group: {clinical_group.isnull().sum()}")

# Validation of each column

# Check DEATH? column (binary)
print(f"DEATH? column unique values: {clinical_group['DEATH?'].unique()}")
# Set all missing values to 0
```

```
clinical_group['DEATH?'] = clinical_group['DEATH?'].fillna(0)

# Check QOL column (number 0-105)
print(f"QOL column unique values: {clinical_group['QOL'].unique()}")
print(f"QOL max value: {clinical_group['QOL'].max()}")
print(f"QOL min value: {clinical_group['QOL'].min()}")
# Set all values bigger than 105 to 105
clinical_group.loc[clinical_group['QOL'] > 105, 'QOL'] = 105
# Set all missing values with backward fill or forward fill
clinical_group['QOL'] = clinical_group['QOL'].fillna(method='bfill')
clinical_group['QOL'] = clinical_group['QOL'].fillna(method='ffill')

# Check OQLsub1 column (number 0-40)
print(f"OQLsub1 column unique values: {clinical_group['OQLsub1'].unique()}")
print(f"OQLsub1 max value: {clinical_group['OQLsub1'].max()}")
print(f"OQLsub1 min value: {clinical_group['OQLsub1'].min()}")
# Change all values bigger than 40 to 40
clinical_group.loc[clinical_group['OQLsub1'] > 40, 'OQLsub1'] = 40
# Set all missing values with backward fill or forward fill
clinical_group['OQLsub1'] = clinical_group['OQLsub1'].fillna(method='bfill')
clinical_group['OQLsub1'] = clinical_group['OQLsub1'].fillna(method='ffill')

# Check OQLsub2 column (number 0-25)
print(f"OQLsub2 column unique values: {clinical_group['OQLsub2'].unique()}")
print(f"OQLsub2 max value: {clinical_group['OQLsub2'].max()}")
print(f"OQLsub2 min value: {clinical_group['OQLsub2'].min()}")
# Change all values bigger than 25 to 25
clinical_group.loc[clinical_group['OQLsub2'] > 25, 'OQLsub2'] = 25
# Set all missing values with backward fill or forward fill
clinical_group['OQLsub2'] = clinical_group['OQLsub2'].fillna(method='bfill')
clinical_group['OQLsub2'] = clinical_group['OQLsub2'].fillna(method='ffill')

# Check LVEF.0 column (number)
print(f"LVEF.0 column unique values: {clinical_group['LVEF.0'].unique()}")
print(f"LVEF.0 max value: {clinical_group['LVEF.0'].max()}")
print(f"LVEF.0 min value: {clinical_group['LVEF.0'].min()}")
# Set all missing values with backward fill
clinical_group['LVEF.0'] = clinical_group['LVEF.0'].fillna(method='bfill')

# Check NYHA column (categories)
print(f"NYHA column unique values: {clinical_group['NYHA'].unique()}")

# Check PM column (binary)
print(f"PM column unique values: {clinical_group['PM'].unique()}")
# Set all missing values to 0
clinical_group['PM'] = clinical_group['PM'].fillna(0)

# Check AETH.HF column (binary)
print(f"AETH.HF column unique values: {clinical_group['AETH.HF'].unique()}")
# Decrease all values by 1
clinical_group['AETH.HF'] = clinical_group['AETH.HF'] - 1
# Set all missing values to 0
clinical_group['AETH.HF'] = clinical_group['AETH.HF'].fillna(0)

# Check LVEDD column (number)
print(f"LVEDD column unique values: {clinical_group['LVEDD'].unique()}")
```

```
print(f"LVEDD max value: {clinical_group['LVEDD'].max()}")
print(f"LVEDD min value: {clinical_group['LVEDD'].min()}")
# Set all missing values with backward fill
clinical_group['LVEDD'] = clinical_group['LVEDD'].fillna(method='bfill')

# Check MR column (categories)
print(f"MR column unique values: {clinical_group['MR'].unique()}")
# Set all missing values to the most frequent value
clinical_group['MR'] = clinical_group['MR'].fillna(clinical_group['MR'].mode()[0])

# Check REST.SBP column (number)
print(f"REST.SBP column unique values: {clinical_group['REST.SBP'].unique()}")
print(f"REST.SBP max value: {clinical_group['REST.SBP'].max()}")
print(f"REST.SBP min value: {clinical_group['REST.SBP'].min()}")
# Set all missing values with backward fill
clinical_group['REST.SBP'] = clinical_group['REST.SBP'].fillna(method='bfill')

# Check REST.DBP column (number)
print(f"REST.DBP column unique values: {clinical_group['REST.DBP'].unique()}")
print(f"REST.DBP max value: {clinical_group['REST.DBP'].max()}")
print(f"REST.DBP min value: {clinical_group['REST.DBP'].min()}")
# Set all missing values with backward fill
clinical_group['REST.DBP'] = clinical_group['REST.DBP'].fillna(method='bfill')

# Check REST.HR column (number)
print(f"REST.HR column unique values: {clinical_group['REST.HR'].unique()}")
print(f"REST.HR max value: {clinical_group['REST.HR'].max()}")
print(f"REST.HR min value: {clinical_group['REST.HR'].min()}")
# Set all missing values with backward fill
clinical_group['REST.HR'] = clinical_group['REST.HR'].fillna(method='bfill')

# Number of missing values after cleaning
print(f"Number of missing values after cleaning: {clinical_group.isnull().sum()}")
```

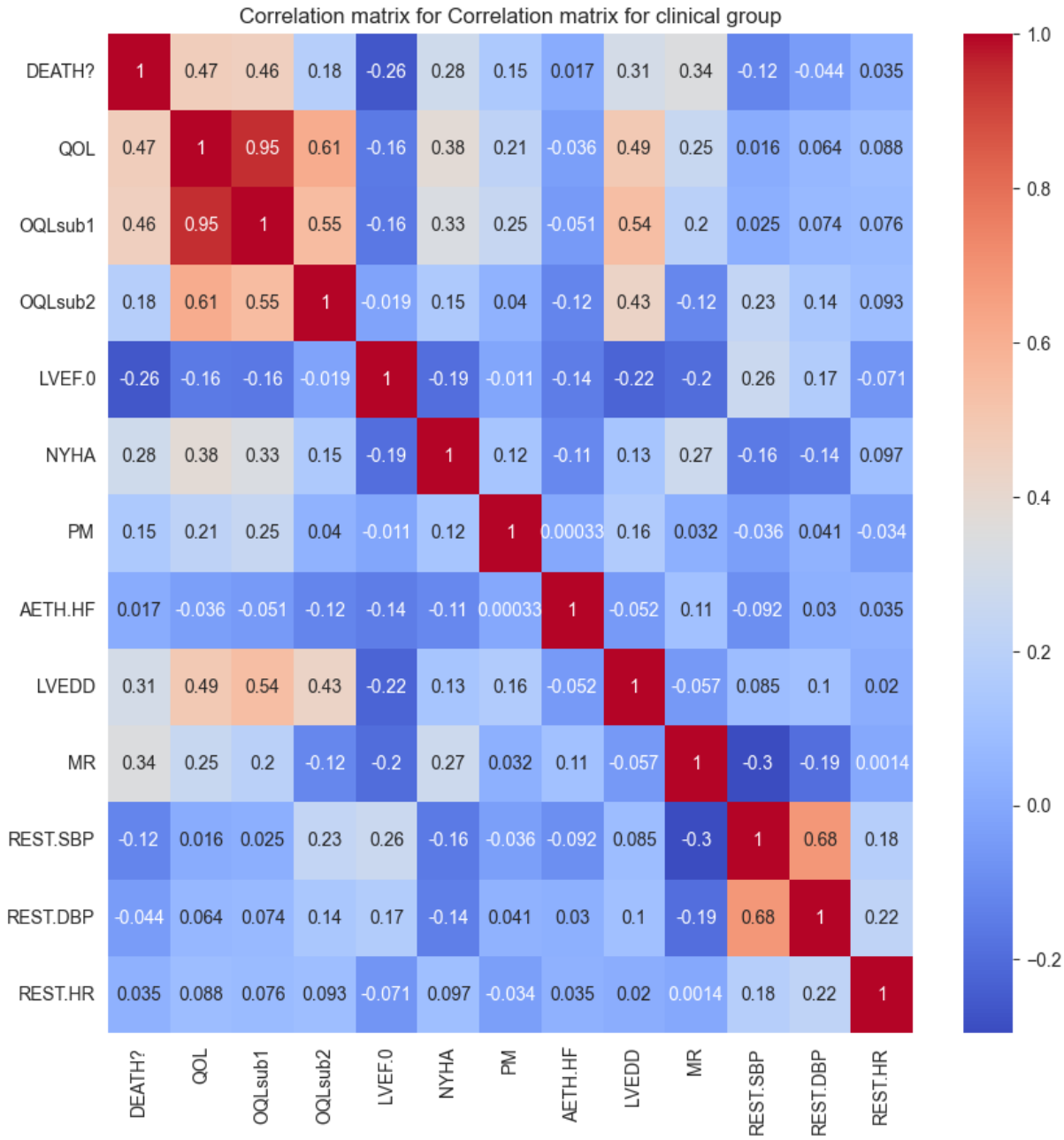
```
Clinical group columns: Index(['DEATH?', 'QOL', 'OQLsub1', 'OQLsub2', 'LVEF.0', 'NYH
A', 'PM',
       'AETH.HF', 'LVEDD', 'MR', 'REST.SBP', 'REST.DBP', 'REST.HR'],
      dtype='object')
Missing values in clinical group: DEATH?       80
QOL         209
OQLsub1     232
OQLsub2     232
LVEF.0        4
NYHA          0
PM           69
AETH.HF       4
LVEDD       158
MR          164
REST.SBP    105
REST.DBP    105
REST.HR     107
dtype: int64
DEATH? column unique values: [ 0.  1. nan]
QOL column unique values: [27. 42.  0. 10.  4. 50. 30. 23.  6. 38.  5. 63. 60. 68. 2
4. 15. 19. 55.
 45. 44. 13. 18. 28. 76. 46. 56. 31. 54. 52. 73. 32. 65.  8. nan 17. 70.
 29. 64. 62. 77. 91. 49. 61. 66.  2. 98. 86. 69. 34. 25. 37. 12. 14.  9.
 58. 22. 11. 16. 40. 36. 51. 74. 67. 48. 26. 47. 21. 81. 35. 78. 79. 57.
 87. 95. 43. 53. 41.  1.  3. 33.]
QOL max value: 98.0
QOL min value: 0.0
OQLsub1 column unique values: [23. 37.  0.  7.  2. 46. 29. 20.  6.  5. 49. 48. 55. 2
2. 42. 15. 16. 39.
 31. 12. 24.  4. 35. 58. 56. 27. 47. 57. 32. 38. 63. nan 17. 28. 21. 45.
 40. 60. 50. 67. 66. 59. 80. 54.  9. 44. 14. 13. 11. 19. 33.  8. 36. 10.
  3. 52. 53. 64. 18. 25. 41. 26. 51. 34. 30. 62. 68. 61. 76. 74. 43.  1.]
OQLsub1 max value: 80.0
OQLsub1 min value: 0.0
OQLsub2 column unique values: [ 4.  5.  0.  3.  2.  1.  9. 14. 12. 13.  8.  6. 10. 1
8.  7. 16. nan 15.
 25. 19. 20. 11. 29. 26. 24. 22. 21. 17.]
OQLsub2 max value: 29.0
OQLsub2 min value: 0.0
LVEF.0 column unique values: [38. 25. 30. 45. 32. 17. 20. 40. 15. 10. 31. 35. 39. 1
8. 28. 23. 24. 26.
 34. 29. 44. 19. nan 37. 27. 22. 42. 43. 36. 33. 16. 55. 60.]
LVEF.0 max value: 60.0
LVEF.0 min value: 10.0
NYHA column unique values: [2. 1. 3. 4.]
PM column unique values: [ 0.  1. nan]
AETH.HF column unique values: [ 1.  2. nan]
LVEDD column unique values: [ 64.   76.   71.   65.   59.   60.  100.   81.   61.
68.   63.   73.
  70.   72.   77.   66.   58.   84.   79.   51.   78.   62.   83.   75.
  86.   99.   91.   74.   80.   67.   88.    nan  55.   57.   52.   54.
  69.   82.   50.   87.   53.   47.   97.   56.   89.   44.   90.   46.
  95.   85.    6.7]
LVEDD max value: 100.0
LVEDD min value: 6.7
MR column unique values: [0.5 1.  2.  0.  3.  2.5 3.5 nan 1.5 4. ]
```

```
REST.SBP column unique values: [145. 115. 140. 120. 130. 105. 100. 160.  90. 110. 15
0. 125. 155.  95.
  80. 135.  nan 180.  70.  85. 165.]
REST.SBP max value: 180.0
REST.SBP min value: 70.0
REST.DBP column unique values: [100.  70.  80.  75.  60.  90.  85.  95.  65.  50.  n
an 105.  40. 110.
   5.]
REST.DBP max value: 110.0
REST.DBP min value: 5.0
REST.HR column unique values: [ 72.  70.  64.  60.  68.  78.  80.  92.  56.  76.  8
4.  85.  52.  88.
  nan 100.  96.  48.  75.  62.  94.  45.  95.  74.]
REST.HR max value: 100.0
REST.HR min value: 45.0
Number of missing values after cleaning: DEATH?      0
QOL          0
OQLsub1      0
OQLsub2      0
LVEF.0       0
NYHA         0
PM           0
AETH.HF      0
LVEDD        0
MR           0
REST.SBP     0
REST.DBP     0
REST.HR      0
dtype: int64
```

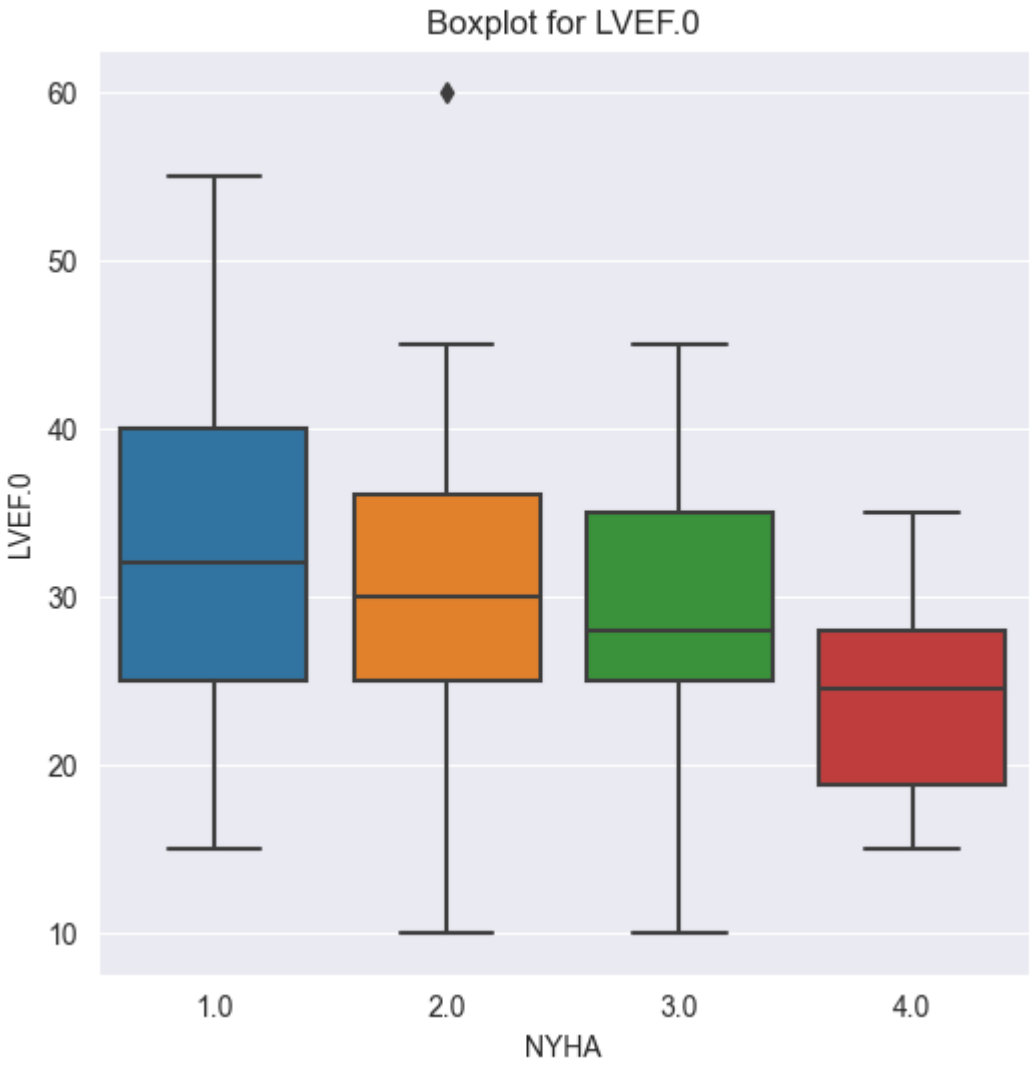
In [ ]:

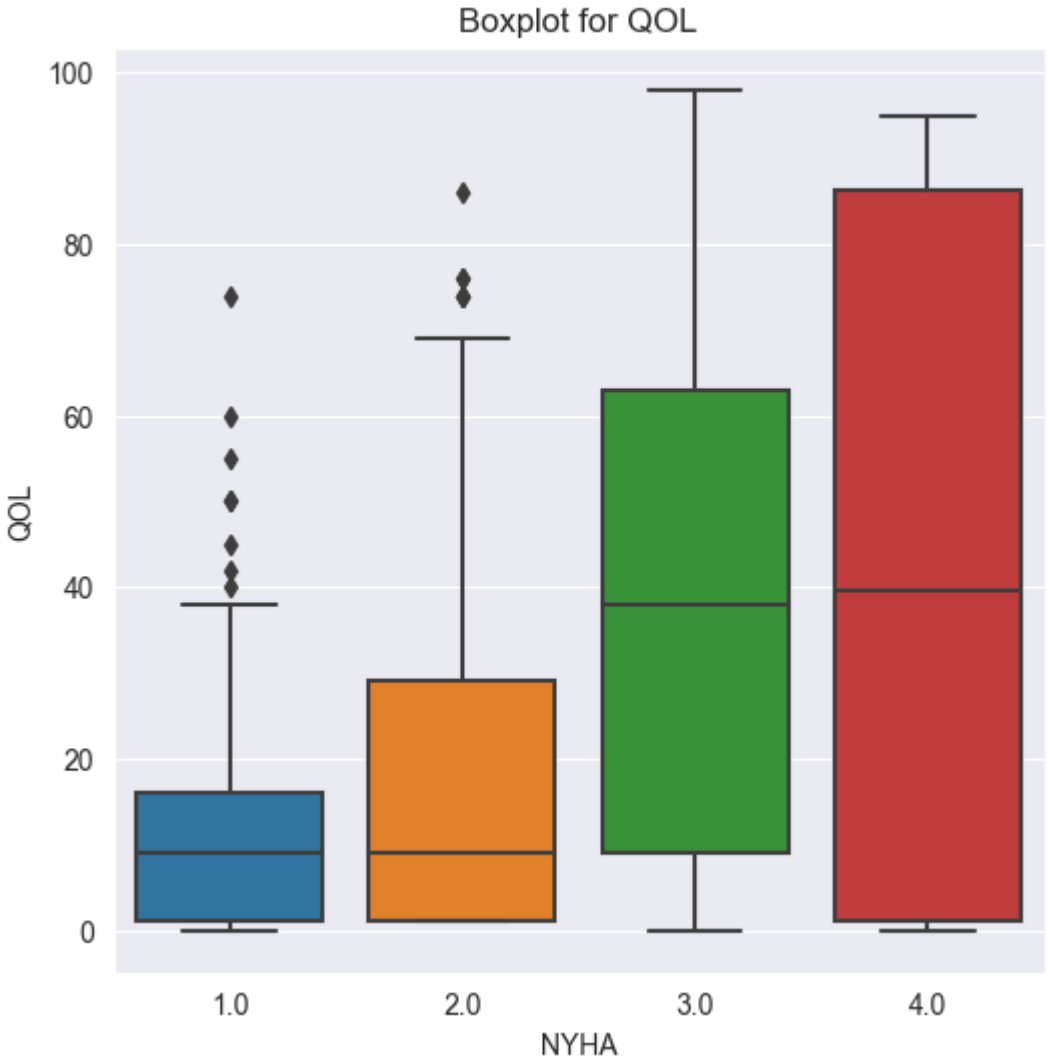
```
corr_matrix = clinical_group.corr()
# Plot correlation chart
plot_corr_matrix(clinical_group, 'Correlation matrix for clinical group', (10, 10))

# Prepare data with correlation, missing values and importance
group_checking = prepare_group_checking(clinical_group)

# Prepare data for Decision Tree Classifier
X = clinical_group.drop(columns=['NYHA'])
y = clinical_group['NYHA']

model = DecisionTreeClassifier()
model.fit(X, y)
feature_importances = model.feature_importances_

group_checking = add_importance_to_group_checking(group_checking, feature_importanc
group_checking = convert_group_checking_to_dataframe(group_checking)
print(group_checking)

# Print boxplots for variables 'LVEF.0','QOL', 'OQLsub1'
box_plots = ['LVEF.0', 'QOL', 'OQLsub1']
for column in box_plots:
    plot_boxplot(clinical_group, column, (6, 6), y=column)

# Remove column with correlation lower than 0.15
columns_to_remove = group_checking[group_checking['correlation'] < 0.18].index
```

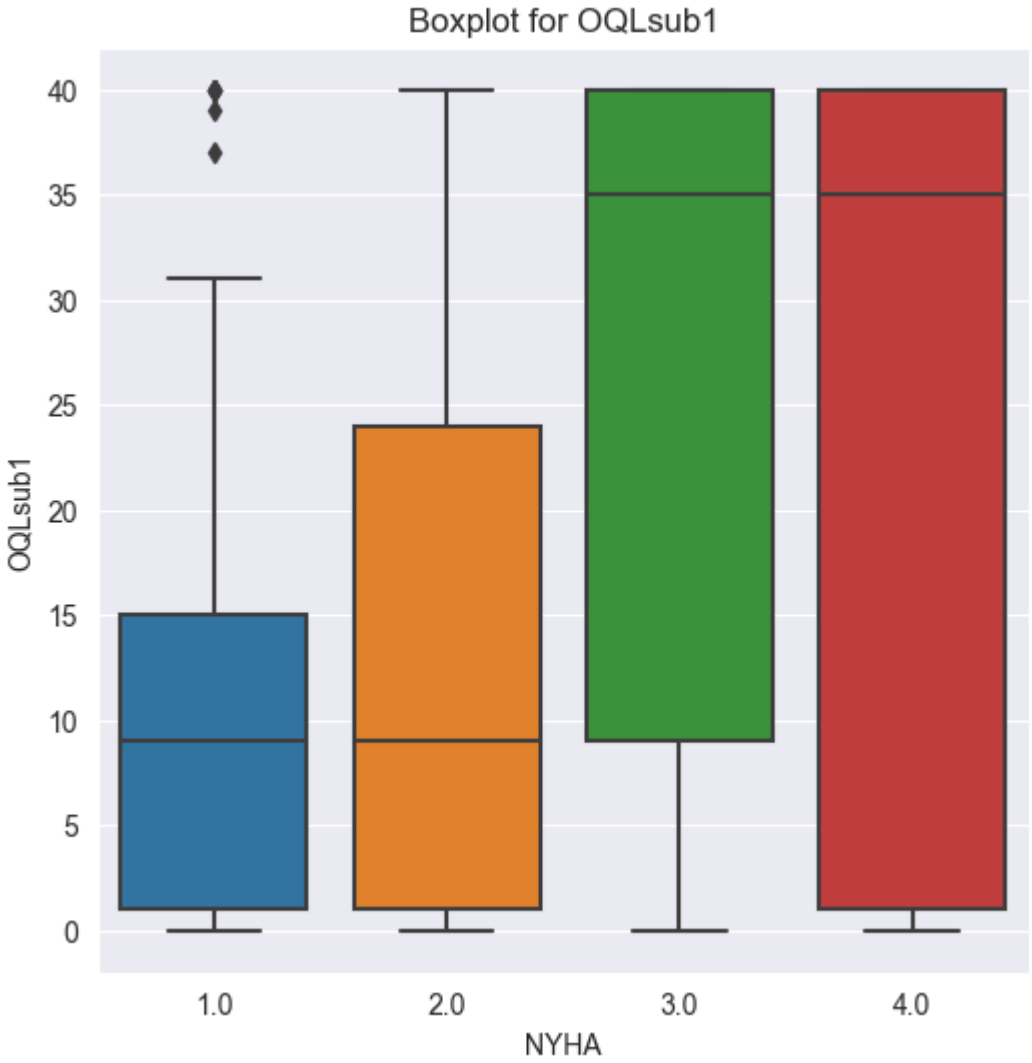
```
clinical_group = clinical_group.drop(columns=columns_to_remove)

plot_corr_matrix(clinical_group,
                 'Correlation matrix for clinical group after removing columns with
                 (6, 6))

# Remove QOL and OQLsub2 columns - they are highly correlated with OQLsub1
columns_to_remove = ['QOL']
clinical_group = clinical_group.drop(columns=columns_to_remove)

# Additionally remove LVEDD and MR columns
# columns_to_remove = ['LVEDD', 'MR']
# clinical_group = clinical_group.drop(columns=columns_to_remove)

plot_corr_matrix(clinical_group, 'Correlation matrix for clinical group after remov

X = clinical_group.drop(columns=['NYHA'])
y = clinical_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)

grid_search, before_grid_search = create_grid_search(model, X, y)
clinical_model = grid_search.best_estimator_

test_with_cross_validation(clinical_model, X, y, before_grid_search=before_grid_sea
print_feature_importances(clinical_model.feature_importances_, X)
```



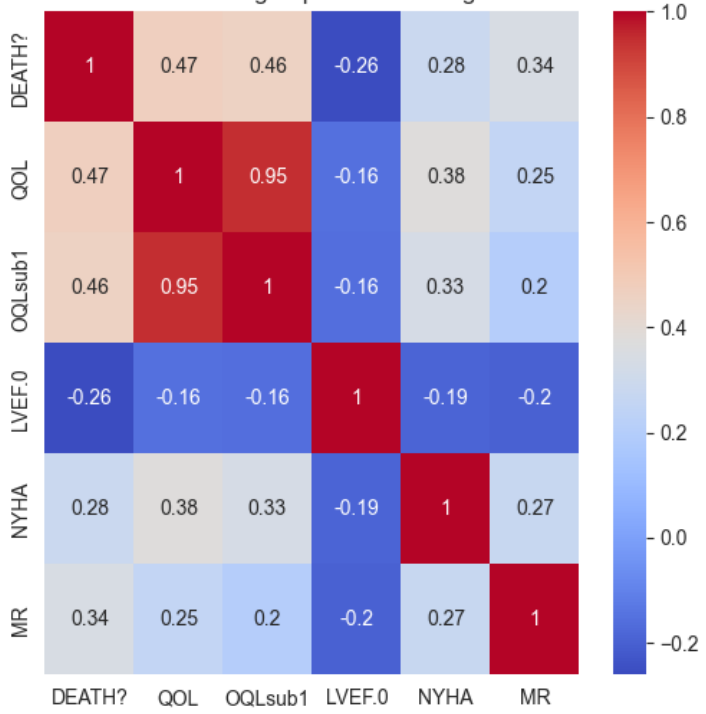
	correlation	missing_values	importance
QOL	0.379008	0	0.148958
OQLsub1	0.333027	0	0.053248
DEATH?	0.283967	0	0.037340
MR	0.272725	0	0.077251
LVEF.0	0.189337	0	0.150666
REST.SBP	0.156368	0	0.077282
OQLsub2	0.153981	0	0.025174
REST.DBP	0.138919	0	0.080358
LVEDD	0.127758	0	0.131470
PM	0.124848	0	0.016977
AETH.HF	0.109894	0	0.063212
REST.HR	0.097181	0	0.138064

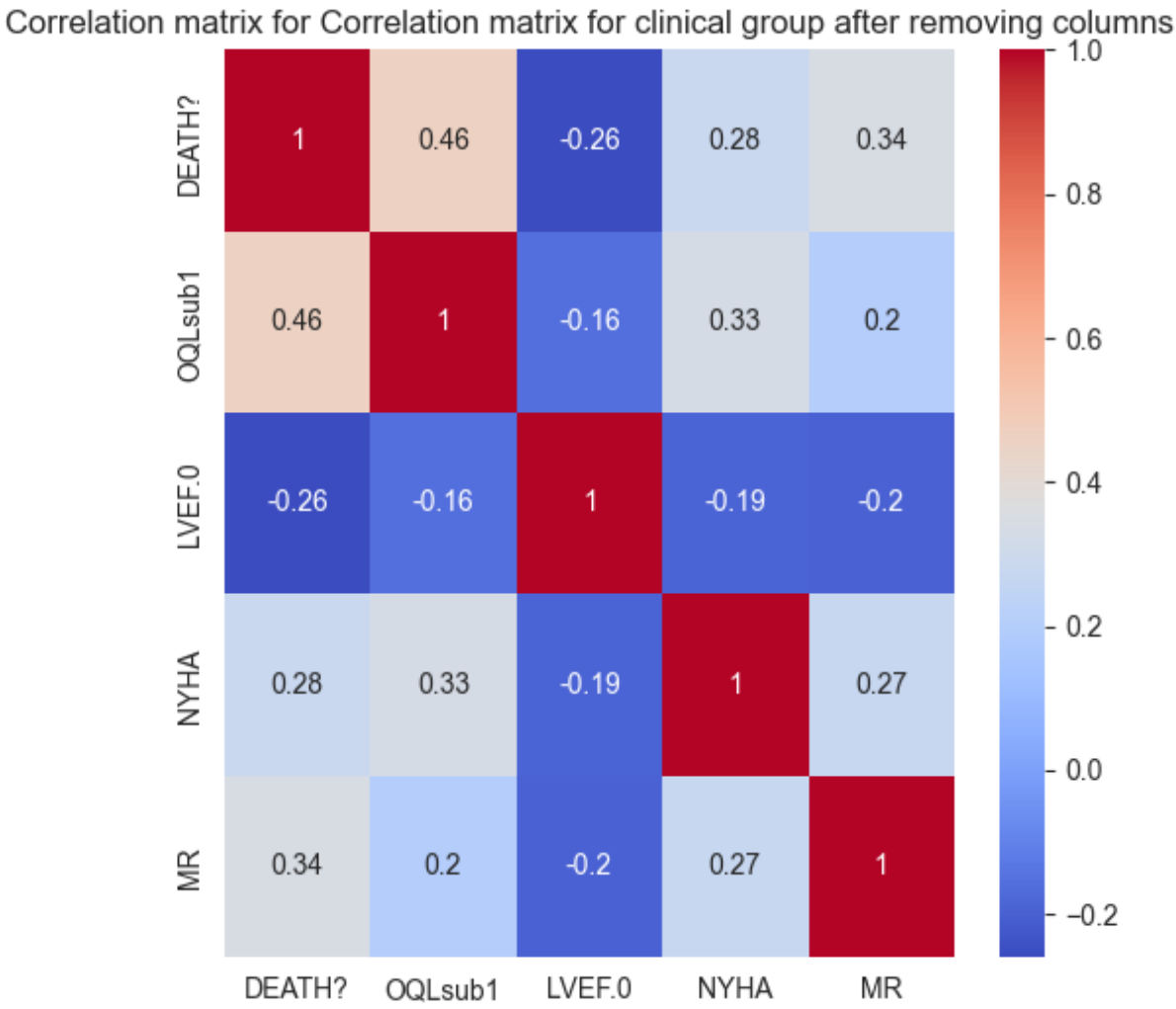

Boxplot for LVEF.0
60
50
40
30
20
10
LVEF.0
1.0
2.0
3.0
4.0
NYHA


Boxplot for QOL
100
80
60
40
20
0
QOL
1.0
2.0
3.0
4.0
NYHA



Correlation matrix for Correlation matrix for clinical group after removing columns with correlation lower than 0.15






```
Cross validation score before grid search: 0.4695652173913043
```

Conclusion from clinical group:

From the above table we can see that the most important variables are:

- OQLsub1 - left ventricular ejection fraction - information from the heart ultrasound reflecting the efficiency of pumping. According to the definition value = 45 or lower is characteristic for systolic heart failure.
- LVEF.0 - left ventricular ejection fraction
- MR - mitral regurgitation - information from the heart ultrasound reflecting the efficiency of pumping. According to the definition value = 45 or lower is characteristic for systolic heart failure.
- DEATH? - information about the patient's death From this group we choose QOL and LVEF.0 variables as discriminative variables. And replace missing values with mean value.

Technical group

Technical group description

The technical group contains variables that are not directly related to the patient's health. These are variables such as the date of the examination, the date of death, the date of birth, etc.

Technical group variables

DEATHDATE [VAR2] - date - date of death (if death=1) or date of the confirmation that the patient is still alive **TIMEFU** [VAR3] - number - number of days between examination and date death or date of the confirmation that the patient is still alive **DOB** [VAR7] - date - date of birth **DOE** [VAR8] - date - date of the examination

---

Data preparation

In [ ]:
```
technical_group = data[groups['technical']].copy()

# Print columns
print(f"Technical group columns: {technical_group.columns}")
print(f"Missing values: {technical_group.isnull().sum()}")

# Validation of each column

# Check DEATHDATE column (date)
print(f"Type of DEATHDATE column: {technical_group['DEATHDATE'].dtypes}")
print(f"DEATHDATE column unique values: {technical_group['DEATHDATE'].unique()}")
# Set all missing values with backward fill
technical_group['DEATHDATE'] = technical_group['DEATHDATE'].fillna(method='bfill')

# Check TIMEFU column (number)
print(f"Type of column TIMEFU: {technical_group['TIMEFU'].dtypes}")
print(f"TIMEFU column unique values: {technical_group['TIMEFU'].unique()}")
# Set incorrect string values to NaN
technical_group["TIMEFU"] = technical_group["TIMEFU"].apply(lambda x: np.nan if typ
# Set all missing values with backward fill
technical_group["TIMEFU"] = technical_group["TIMEFU"].fillna(method='bfill')

# Check DOE column (date)
print(f"Type of DOE column: {technical_group['DOE'].dtypes}")
print(f"DOE column unique values: {technical_group['DOE'].unique()}")
# Set all missing values with backward fill
technical_group['DOE'] = technical_group['DOE'].fillna(method='bfill')

# Check DOB column (date)
print(f"Type of DOB column: {technical_group['DOB'].dtypes}")
print(f"DOB column unique values: {technical_group['DOB'].unique()}")
# Set all missing values with backward fill
technical_group['DOB'] = technical_group['DOB'].fillna(method='bfill')
```

```
# Add NYHA target variable
technical_group['NYHA'] = NYHA
```

In [ ]:
```
plot_corr_matrix(technical_group, 'Correlation matrix for technical group', (6, 6))

# Print together corelation values for NYHA target variable with number of missing
checking_data = prepare_group_checking(technical_group)
checking_data = convert_group_checking_to_dataframe(checking_data, columns=['correl
print(checking_data)
```

Conclusion from technical group:

From the above table we don't see any variable with high correlation with NYHA target variable. So we will not use any variable from this group.

Demographic group

Demographic group description

The group contains only one variable - age.

Demographic group variables

AGE [VAR9] - number - age at examination

Data preparation

In [ ]:
```
demographic_group = data[groups['demographic']].copy()
print(f"Demographic group columns: {demographic_group.columns}")
print(f"Missing values: {demographic_group.isnull().sum()}")

# Validation of each column

# Check AGE column (number)
print(f"Type of AGE column: {demographic_group['AGE'].dtypes}")
print(f"AGE column unique values: {demographic_group['AGE'].unique()}")
# Round age to integer
demographic_group['AGE'] = demographic_group['AGE'].round()
# Set all missing values with backward fill
demographic_group['AGE'] = demographic_group['AGE'].fillna(method='bfill')
# Change type of AGE column to int
demographic_group['AGE'] = demographic_group['AGE'].astype(int)
```

```
# Add NYHA target variable
demographic_group['NYHA'] = NYHA
```

In [ ]:
```
# Print correlation chart
plot_corr_matrix(demographic_group, 'Correlation matrix for demographic group', (3,

# Print boxplot for NYHA and AGE
plot_boxplot(demographic_group, 'NYHA', (3, 3), x='NYHA', y='AGE')
```

Conclusion from demographic group:

We choose AGE variable as discriminative variable. And replace missing values with mean value.

Antrhopometric group

Antrhopometric group description

The group contains variables related to the patient's body mass and height.

Antrhopometric group variables

HEIGHT.CM [VAR10] - number - body height in cm **WEIGHT.KG** [VAR11] - number - body mass in kg **BMI** [VAR12] - number - body mass index (mass per squared height); <18.5=underweight range, 18.5 to <25 = normal, 25 - <30 = overweight, 30.0 or higher = obese.

Data preparation

In [ ]:
```
# Create anthropometric group
anthropometric_group = data[groups['anthropometry']].copy()
print(f"Anthropometry group columns: {anthropometric_group.columns}")
print(f"Mising values: {anthropometric_group.isnull().sum()}")

# Validation of each column

# Check HEIGHT.CM column (number)
print(f"Type of HEIGHT.CM column: {anthropometric_group['HEIGHT.CM'].dtypes}")
# print(f"HEIGHT.CM column unique values: {anthropometric_group['HEIGHT.CM'].unique
print(f"Max value of HEIGHT.CM column: {anthropometric_group['HEIGHT.CM'].max()}")
print(f"Min value of HEIGHT.CM column: {anthropometric_group['HEIGHT.CM'].min()}")
```

```
# Check WEIGHT.KG column (number)
print(f"Type of WEIGHT.KG column: {anthropometric_group['WEIGHT.KG'].dtypes}")
# print(f"WEIGHT.KG column unique values: {anthropometric_group['WEIGHT.KG'].unique
print(f"Max value of WEIGHT.KG column: {anthropometric_group['WEIGHT.KG'].max()}")
print(f"Min value of WEIGHT.KG column: {anthropometric_group['WEIGHT.KG'].min()}")

# Check BMI column (number)
print(f"Type of BMI column: {anthropometric_group['BMI'].dtypes}")
# print(f"BMI column unique values: {anthropometric_group['BMI'].unique()}")
print(f"Max value of BMI column: {anthropometric_group['BMI'].max()}")
print(f"Min value of BMI column: {anthropometric_group['BMI'].min()}")

# Add NYHA target variable
anthropometric_group['NYHA'] = NYHA
```

In [ ]:
```
plot_corr_matrix(anthropometric_group, 'Correlation matrix for anthropometric group

# Print together corelation values for NYHA target variable with number of missing
group_checking = prepare_group_checking(anthropometric_group)
group_checking = convert_group_checking_to_dataframe(group_checking, columns=['corr
print(group_checking)
```

Conclusion from anthropometric group:

From the above table we don't see any variable with high correlation with NYHA target variable. So we will not use any variable from this group.

Comorbidities group

Comorbidities group description

The group contains variables related to the patient's comorbidities.

Comorbidities group variables

MI [VAR17] - binary - information about previous myocardial infarction (1=yes) **AF** [VAR18] - binary - information about atrial fibrillation (1=yes) **DM** [VAR19] - binary - information about diabetes (1=yes) **HT** [VAR20] - binary - information about hypertension (1=yes) **COPD** [VAR21] - binary - information about lung disease (1=yes) **STROKE** [VAR22] - binary - information about previous stroke (1=yes) **KIDNEY.DIS** [VAR23] - binary - information about kidney disease (1=yes)

Data preparation

In [ ]:
```
# Create comorbidities group
comorbidities_group = data[groups['comorbidities']].copy()
print(f"Comorbidities group columns: {comorbidities_group.columns}")
print(f"Mising values: {comorbidities_group.isnull().sum()}")

# Validation of each column

# Check MI column (binary)
print(f"Type of MI column: {comorbidities_group['MI'].dtypes}")
print(f"MI column unique values: {comorbidities_group['MI'].unique()}")
# Set all missing values with 0 (no comorbidity)
comorbidities_group['MI'] = comorbidities_group['MI'].fillna(0)

# Check AF column (binary)
print(f"Type of AF column: {comorbidities_group['AF'].dtypes}")
print(f"AF column unique values: {comorbidities_group['AF'].unique()}")
# Set all missing values with 0 (no comorbidity)
comorbidities_group['AF'] = comorbidities_group['AF'].fillna(0)

# Check DM column (binary)
print(f"Type of DM column: {comorbidities_group['DM'].dtypes}")
print(f"DM column unique values: {comorbidities_group['DM'].unique()}")
# Set all missing values with 0 (no comorbidity)
comorbidities_group['DM'] = comorbidities_group['DM'].fillna(0)

# Check HT column (binary)
print(f"Type of HT column: {comorbidities_group['HT'].dtypes}")
print(f"HT column unique values: {comorbidities_group['HT'].unique()}")
# Set all missing values with 0 (no comorbidity)
comorbidities_group['HT'] = comorbidities_group['HT'].fillna(0)

# Check COPD column (binary)
print(f"Type of COPD column: {comorbidities_group['COPD'].dtypes}")
print(f"COPD column unique values: {comorbidities_group['COPD'].unique()}")
# Set all missing values with 0 (no comorbidity)
comorbidities_group['COPD'] = comorbidities_group['COPD'].fillna(0)

# Check STROKE column (binary)
print(f"Type of STROKE column: {comorbidities_group['STROKE'].dtypes}")
print(f"STROKE column unique values: {comorbidities_group['STROKE'].unique()}")
# Set all missing values with 0 (no comorbidity)
comorbidities_group['STROKE'] = comorbidities_group['STROKE'].fillna(0)

# Check KIDNEY.DIS column (binary)
print(f"Type of KIDNEY.DIS column: {comorbidities_group['KIDNEY.DIS'].dtypes}")
print(f"KIDNEY.DIS column unique values: {comorbidities_group['KIDNEY.DIS'].unique(
# Set all missing values with 0 (no comorbidity)
comorbidities_group['KIDNEY.DIS'] = comorbidities_group['KIDNEY.DIS'].fillna(0)

# Add NYHA target variable
comorbidities_group['NYHA'] = NYHA
```

In [24]:
```
# Print correlation chart
plot_corr_matrix(comorbidities_group, 'Correlation matrix for comorbidities group',

group_checking = prepare_group_checking(comorbidities_group)

# Print together corelation values for NYHA target variable with number of missing
X = comorbidities_group.drop(columns=['NYHA'])
y = comorbidities_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)

group_checking = add_importance_to_group_checking(group_checking, model.feature_imp
group_checking = convert_group_checking_to_dataframe(group_checking)
print(group_checking)

# Remove columns with correlation less than 0.15
columns_to_remove = group_checking[group_checking['correlation'] < 0.15].index
print(f"Columns to remove: {columns_to_remove}")
print(f"Columns before removing: {comorbidities_group.columns}")
comorbidities_group = comorbidities_group.drop(columns=columns_to_remove)

plot_corr_matrix(comorbidities_group, 'Correlation matrix for clinical group after

X = comorbidities_group.drop(columns=['NYHA'])
y = comorbidities_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)

grid_search, before_grid_search = create_grid_search(model, X, y)
comorbidities_model = grid_search.best_estimator_

test_with_cross_validation(comorbidities_model, X, y, before_grid_search=before_gri
print_feature_importances(comorbidities_model.feature_importances_, X)
```

```
Best parameters: {'class_weight': None, 'criterion': 'gini', 'max_depth': 2, 'min_sa
mples_leaf': 1, 'min_samples_split': 2}
Best score: 0.5260869565217391
Cross validation score after grid search: 0.5043478260869565
Mean cross validation score after optimize: 0.5043478260869565
Growth of cross validation score: 0.008695652173913049
      feature  importance
1          DM    0.581672
2  KIDNEY.DIS    0.418328
0          AF    0.000000
```

Conclusion from comorbidities group:

From the above table we can see that the most important variables are:

- AF - atrial fibrillation (migotanie przedsionków)
- KIDNEY.DIS - kidney disease (choroba nerek)
- DM - diabetes mellitus (cukrzyca)

Treatment group

Treatment group description

The treatment group contains information about treatment of patients.

Treatment group variables

- **ACEI.ARB** [VAR24] - binary - information about treatment using ace-inhibitors or ARB (similar drugs, 1=yes)
- **BB** [VAR25] - binary - information about treatment using beta blockers (1=yes)
- **MRA** [VAR26] - binary - information about treatment using aldosterone antagonists (1=yes)
- **DIUR** [VAR27] - binary - information about treatment using oral diuretics (1=yes)
- **ANTIPLAT** [VAR28] - binary - information about treatment using antiplatelet drugs (1=yes)
- **STATIN** [VAR29] - binary - information about treatment using statin (1=yes)
- **DIGOX** [VAR30] - binary - information about treatment using digoxin (1=yes)

Data preparation

In [25]:
```python
# Create treatment group
treatment_group = data[groups['treatment']].copy()
print(f"Treatment group columns: {treatment_group.columns}")
print(f"Missing values: {treatment_group.isnull().sum()}")

# Validation of each column

# Check ACEI.ARB column (binary)
print(f"Type of ACEI.ARB column: {treatment_group['ACEI.ARB'].dtypes}")
print(f"ACEI.ARB column unique values: {treatment_group['ACEI.ARB'].unique()}")
# Set all missing values with 0 (no treatment)
treatment_group['ACEI.ARB'] = treatment_group['ACEI.ARB'].fillna(0)

# Check BB column (binary)
print(f"Type of BB column: {treatment_group['BB'].dtypes}")
print(f"BB column unique values: {treatment_group['BB'].unique()}")
# Set all missing values with 0 (no treatment)
treatment_group['BB'] = treatment_group['BB'].fillna(0)

# Check MRA column (binary)
print(f"Type of MRA column: {treatment_group['MRA'].dtypes}")
print(f"MRA column unique values: {treatment_group['MRA'].unique()}")
# Set all missing values with 0 (no treatment)
treatment_group['MRA'] = treatment_group['MRA'].fillna(0)
```

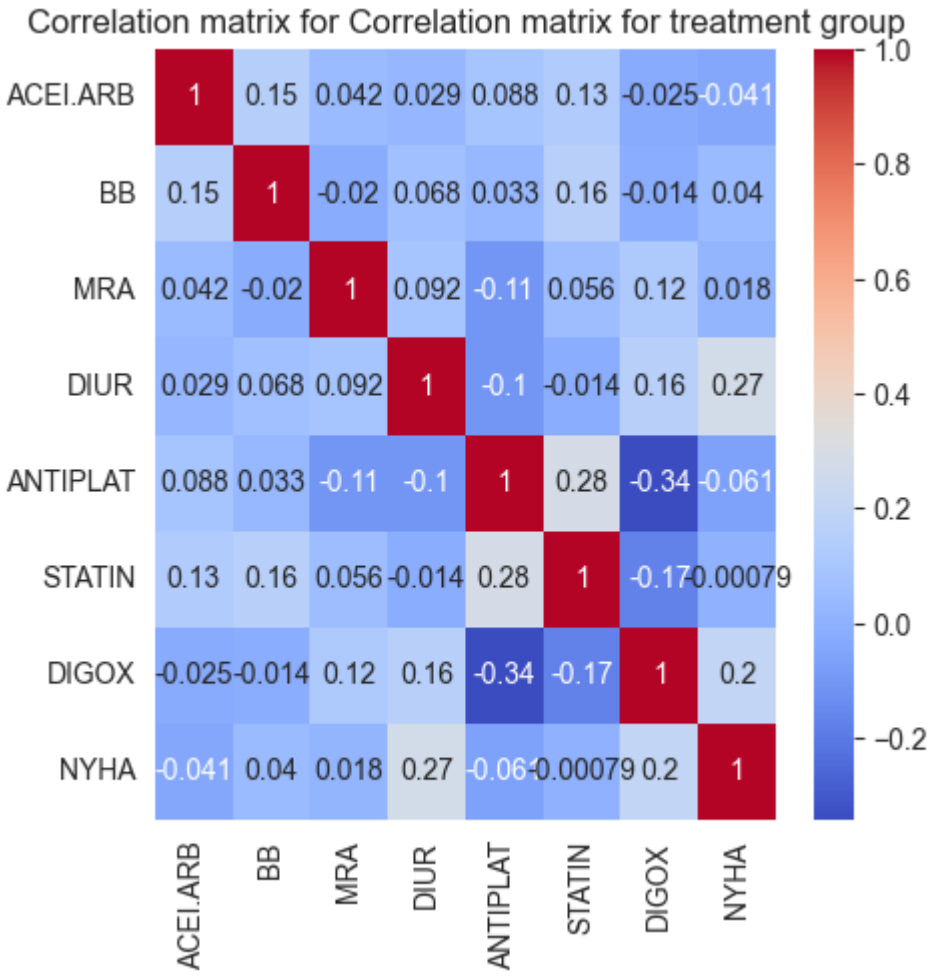
```
# Check DIUR column (binary)
print(f"Type of DIUR column: {treatment_group['DIUR'].dtypes}")
print(f"DIUR column unique values: {treatment_group['DIUR'].unique()}")
# Set all missing values with 0 (no treatment)
treatment_group['DIUR'] = treatment_group['DIUR'].fillna(0)

# Check ANTIPLAT column (binary)
print(f"Type of ANTIPLAT column: {treatment_group['ANTIPLAT'].dtypes}")
print(f"ANTIPLAT column unique values: {treatment_group['ANTIPLAT'].unique()}")
# Set all missing values with 0 (no treatment)
treatment_group['ANTIPLAT'] = treatment_group['ANTIPLAT'].fillna(0)

# Check STATIN column (binary)
print(f"Type of STATIN column: {treatment_group['STATIN'].dtypes}")
print(f"STATIN column unique values: {treatment_group['STATIN'].unique()}")
# Set all missing values with 0 (no treatment)
treatment_group['STATIN'] = treatment_group['STATIN'].fillna(0)

# Check DIGOX column (binary)
print(f"Type of DIGOX column: {treatment_group['DIGOX'].dtypes}")
print(f"DIGOX column unique values: {treatment_group['DIGOX'].unique()}")
# In DIGOX we have additional value 9.0 - unknown so we replace it with 0
treatment_group['DIGOX'] = treatment_group['DIGOX'].replace(9.0, 0)
print(f"DIGOX column unique values: {treatment_group['DIGOX'].unique()}")
# Set all missing values with 0 (no treatment)
treatment_group['DIGOX'] = treatment_group['DIGOX'].fillna(0)

# Check if there are any missing values
print(f"Missing values: {treatment_group.isnull().sum()}")

# Add NYHA target variable
treatment_group['NYHA'] = NYHA
```

```
Treatment group columns: Index(['ACEI.ARB', 'BB', 'MRA', 'DIUR', 'ANTIPLAT', 'STATI
N', 'DIGOX'], dtype='object')
Missing values: ACEI.ARB     1
BB           1
MRA         13
DIUR         5
ANTIPLAT     1
STATIN       2
DIGOX        2
dtype: int64
Type of ACEI.ARB column: float64
ACEI.ARB column unique values: [ 1.  0. nan]
Type of BB column: float64
BB column unique values: [ 1.  0. nan]
Type of MRA column: float64
MRA column unique values: [ 1.  0. nan]
Type of DIUR column: float64
DIUR column unique values: [ 1.  0. nan]
Type of ANTIPLAT column: float64
ANTIPLAT column unique values: [ 1.  0. nan]
Type of STATIN column: float64
STATIN column unique values: [ 1.  0. nan]
Type of DIGOX column: float64
DIGOX column unique values: [ 0.  1.  9. nan]
DIGOX column unique values: [ 0.  1. nan]
Missing values: ACEI.ARB    0
BB          0
MRA         0
DIUR        0
ANTIPLAT    0
STATIN      0
DIGOX       0
dtype: int64
```

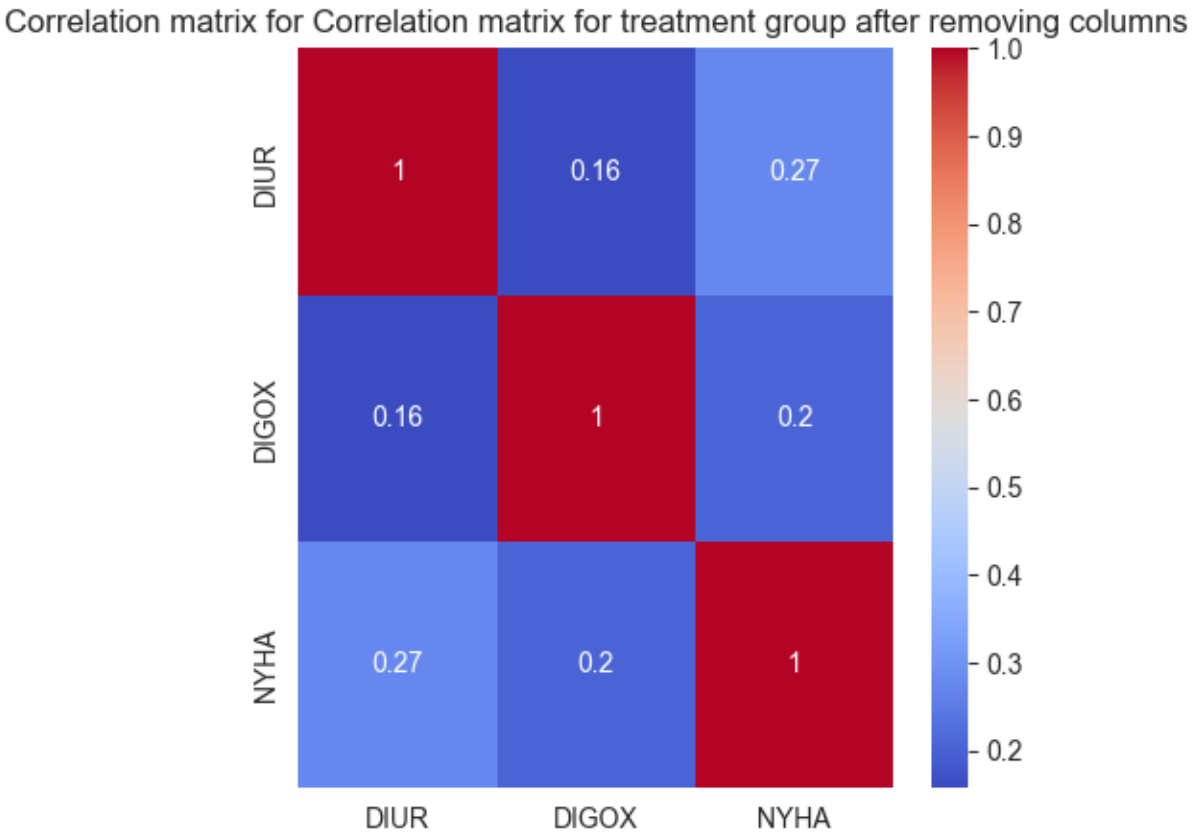
In [26]:

```python
# Correlation matrix
plot_corr_matrix(treatment_group, 'Correlation matrix for treatment group', (5, 5))

group_checking = prepare_group_checking(treatment_group)

# Find important variables and replace all missing values with 0 (no treatment)
treatment_group = treatment_group.fillna(0)

# Prepare data for Decision Tree Classifier
X = treatment_group.drop(columns=['NYHA'])
y = treatment_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)
group_checking = add_importance_to_group_checking(group_checking, model.feature_imp
group_checking = convert_group_checking_to_dataframe(group_checking)
print(group_checking)

# Remove columns with correlation less than 0.05
columns_to_remove = group_checking[group_checking['correlation'] < 0.20].index
treatment_group = treatment_group.drop(columns=columns_to_remove)

plot_corr_matrix(treatment_group, 'Correlation matrix for treatment group after rem
```

```
X = treatment_group.drop(columns=['NYHA'])
y = treatment_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)

grid_search, before_grid_search = create_grid_search(model, X, y)
treatment_model = grid_search.best_estimator_

test_with_cross_validation(treatment_model, X, y, before_grid_search=before_grid_se
print_feature_importances(treatment_model.feature_importances_, X)
```



	correlation	missing_values	importance
DIUR	0.273308	0	0.203359
DIGOX	0.203265	0	0.066039
ANTIPLAT	0.060562	0	0.120560
ACEI.ARB	0.041197	0	0.115715
BB	0.040309	0	0.113801
MRA	0.018168	0	0.228760
STATIN	0.000790	0	0.151766




```
Cross validation score before grid search: 0.5152173913043478
Best parameters: {'class_weight': None, 'criterion': 'gini', 'max_depth': 2, 'min_sa
mples_leaf': 1, 'min_samples_split': 2}
Best score: 0.5217391304347826
Cross validation score after grid search: 0.5282608695652175
Mean cross validation score after optimize: 0.5282608695652175
Growth of cross validation score: 0.013043478260869601
  feature  importance
0    DIUR    0.754865
1   DIGOX    0.245135
```

Conclusion from treatment group:

From the above table we can see that the most important variables are:

- DIUR - diuretics (leki moczopędne)
- DIGOX - digoxin (leki glikozydowe)

Biochemistry group

Biochemistry group description

This group contains variables from biochemistry tests.

Biochemistry group variables

- **HB** [VAR31] - number - level of blood hemoglobin
- **NA** [VAR32] - number - level of blood sodium
- **K** [VAR33] - number - level of blood potassium
- **BNP** [VAR34] - number - level of blood peptide: BNP (elevated level is characteristic for heart failure)
- **CRP** [VAR35] - number - level of blood protein CRP (characteristic for inflammation)

Data preparation

In [27]:
```
# Create biochemistry group
biochemistry_group = data[groups['biochemistry']].copy()
print(f"Biochemistry group columns: {biochemistry_group.columns}")
print(f"Missing values: {biochemistry_group.isnull().sum()}")

# Validation of each column

# Check HB column (number)
print(f"Type of HB column: {biochemistry_group['HB'].dtypes}")
print(f"HB column unique values: {biochemistry_group['HB'].unique()}")
# Set all missing values with bfill (use next valid observation to fill gap)
biochemistry_group['HB'] = biochemistry_group['HB'].fillna(method='bfill')

# Check NA column (number)
print(f"Type of NA column: {biochemistry_group['NA'].dtypes}")
print(f"NA column unique values: {biochemistry_group['NA'].unique()}")
# Set all missing values with bfill (use next valid observation to fill gap)
biochemistry_group['NA'] = biochemistry_group['NA'].fillna(method='bfill')

# Check K column (number)
print(f"Type of K column: {biochemistry_group['K'].dtypes}")
print(f"K column unique values: {biochemistry_group['K'].unique()}")
# Set all missing values with bfill (use next valid observation to fill gap)
biochemistry_group['K'] = biochemistry_group['K'].fillna(method='bfill')

# Check BNP column (number)
print(f"Type of BNP column: {biochemistry_group['BNP'].dtypes}")
print(f"BNP column unique values: {biochemistry_group['BNP'].unique()}")
# In BNP we have additional value 9999.0 - unknown so we replace it with NaN
is_string = biochemistry_group['BNP'].apply(lambda x: isinstance(x, str))
biochemistry_group['BNP'] = biochemistry_group['BNP'].where(~is_string, np.nan)
# Set all missing values with bfill (use next valid observation to fill gap)
biochemistry_group['BNP'] = biochemistry_group['BNP'].fillna(method='bfill')

# Check CRP column (number)
print(f"Type of CRP column: {biochemistry_group['CRP'].dtypes}")
print(f"CRP column unique values: {biochemistry_group['CRP'].unique()}")
# Set all missing values with bfill (use next valid observation to fill gap)
biochemistry_group['CRP'] = biochemistry_group['CRP'].fillna(method='bfill')
```

```
# Add NYHA target variable
biochemistry_group['NYHA'] = NYHA
```

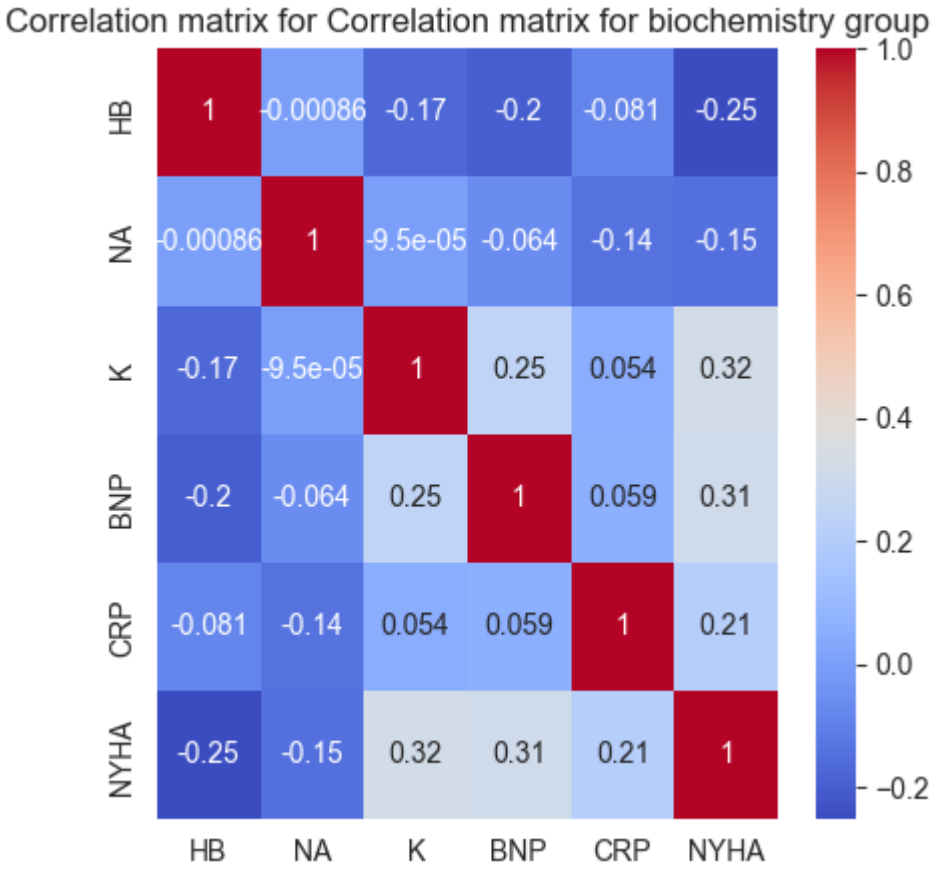
```
Biochemistry group columns: Index(['HB', 'NA', 'K', 'BNP', 'CRP'], dtype='object')
Missing values: HB      15
NA      16
K       16
BNP     44
CRP    150
dtype: int64
Type of HB column: float64
HB column unique values: [17.8 14.5 15.1 14.6 13.7 15.7 15.6 14.7 16.7 12.3 15.5 16.
1 13.1 12.8
 15.2 13.8 15.3 17.6 14.1 17.  12.7 10.7 11.1 13.4 13.9 16.  14.8 16.9
 11.4 16.6 15.4 14.3 12.9 13.6  9.3 12.5 14.4 12.2 14.  14.9 15.8 10.2
 13.2 18.3 13.5 12.6  nan 15.  11.6 17.7 16.3 17.1 14.2 13.  12.  11.8
 16.8 16.5 12.1 12.4 11.9 10.3 17.3 11.3  9.4 10.  10.4 10.5 13.3 17.5
 11.2 16.4 11.5 15.9 11.7 11.  16.2  9.6]
Type of NA column: float64
NA column unique values: [140. 141. 142. 143. 138. 144. 135. 146. 147. 139. 137. 13
4. 132. 136.
 130. 145. 129. 131.  nan 148. 152. 150. 121. 127. 128.  14.  43.]
Type of K column: float64
K column unique values: [0.94 1.3  1.02 1.11 0.8  0.77 1.01 1.1  0.98 1.12 1.03 0.81
0.99 1.15
 1.07 0.9  1.05 1.19 0.93 1.45 0.97 1.31 1.2  0.89 1.04 1.08 1.46 0.86
 1.24 0.88 1.   1.4  1.26 1.18 0.74 1.22 1.06 0.95 2.16 1.42 1.33 1.14
 3.2  1.34 1.7  1.29 1.53 1.28 1.89 0.87 0.67 1.13 1.21 0.79 0.65 1.49
 1.25 0.83  nan 1.09 0.58 0.96 0.91 1.36 0.78 2.05 0.85 1.16 2.55 0.82
 0.76 0.72 1.52 2.17 1.23 0.92 1.41 1.27 1.35 1.38 1.54 1.93 1.81 0.6
 1.37 2.41 1.56 1.86 1.43 1.51 1.39 1.64 2.76 1.63 1.5  1.17 1.98 1.88
 1.65 0.73 1.69 1.32 0.59 0.68 0.69 0.84 0.71 4.76 1.47 1.48 0.64 4.23
 0.55 2.14 1.67 0.7  1.8 ]
Type of BNP column: object
BNP column unique values: [361 389.2 2515 514 32 345.4 1540 1128 2034.9 nan 342 560
2486 1644 13180
 8742 3445.7 896 5711 1824.5 1379.2 4615 3119.9 1690.6 745.3 148.5 771
 4433.1 10499 481 325 5424.5 1647 718 1470 4517.7 681 699.3 1975 174.8
 2011 3850.6 866.6 12753 3848.7 5662.3 9852.7 9895.8 5210.3 4478 2921 3132
 3823.6 7854.5 1956.8 22148 531 29811 294.8 19128 13302 4995.1 705 2879.7
 4543.7 473.8 2610 1563 3010 2976 1801 3575 321 863 171 1445 5013 1190 931
 966 1592 3211 1412 8759 3135 1689 5650 4562 227 3964 143 670 204 322 1120
 155 14 816 91 167 1766 920 228 174 511 1031 436 747 821 1962 1432 236 254
 110 404 666 4453 2278 624 1897 282 181 388 349 1183 2133 760 753 389 940
 316 1121 3003 257 1129 238 3291 323 721 2910 1272 5155 2745 625 2268 109
 190 346 1703 1680 4931 23954 1286 3796 341.7 3268 294.7 293.9 24.27 206
 333.3 124.8 1283 84 1421 184.7 57.2 357 943.3 339.9 400.4 173 3954 4335
 383.7 7200 843.8 1673.1 575 1197 310 6222 140.4 941.1 520 111.6 253 1832
 1645 1162 28.6 378 2295 2002 303.2 614.3 413.6 1169 279.5 6808 4717 208
 9685 3830 26134 482.4 1340 13582 1450 211.1 1251 494.5 1297 3200 3487
 8206 8763 731.6 297.3 2907.4 454.9 3833 656 2082 1612 431.2 1005 9542
 421.4 389.8 3216 2582 4055 6421 3430 10319 468.8 31140 4429 878.8 8682
 4329 48.4 3049 1378 7201 397.5 5137 386.2 36800 607 18812 191.8 711 405
 829.9 289.6 1141 7322 420 2655 4226 1347 1710.1 1942 189 197 2362 9907.2
 419 4863 3139 12080 362 1149 430 557 2051 396 1740 25 45 126 258 577 1940
 343 74 3598 408 5236 5768 6520 245 835 1929 4653 976 730 551 606 1509 294
 320 1862 188 195 3768 2383 88 6550 3772 7061 4757 7492 5546 1364 671 92
 1246 1072 754 317 'od Andrzeja' 57 1079 6851 971 1081 2401 1218 641 395
 353 1998 305 324 125 504 621 2287 1332 2314 833 3113 1047 198 352 233 'x'
```

```
 15 151 764 54 146 603 69 1391 230 880 548 1497 4736 895 955 675 1548 1430
 528 284 646 515 725 287 453 5067 1533 9.99 1622.8 10022 873.2 241 3499
 3116.8 10593 417 2108 9651 19154 218 1929.1 1988]
Type of CRP column: float64
CRP column unique values: [10.5   3.17  4.02  1.41  1.02  1.24  4.29  0.52  0.99  1.
12  4.93  0.7
  2.02  3.5  12.2   7.2   1.16  0.42  7.07  1.39  3.45 10.8   6.27  1.44
  1.95  1.47  1.4   1.2   7.65  2.9   7.3   4.26 14.1   3.    7.78  7.86
  8.76  1.85 34.6   1.7   1.35 10.    6.76  1.26  3.7   4.88  5.85  5.46
  0.87 22.5   1.87 30.3   7.31  4.95 12.8   6.8   6.83  1.11  0.83  1.06
  9.78  9.7  29.5  19.7   1.31  0.38  1.98  1.45  3.08  0.49   nan 11.2
  0.75  4.23  7.58  7.5   2.41  5.5   6.41  1.91  0.57  0.51  1.18  5.21
  2.66  0.64  7.46 10.2   1.84  0.68  1.    7.82  0.61  0.76  9.75  0.17
  1.92  1.6   0.25  4.66  0.23  2.71  0.41  5.74  7.22  3.2   5.6   0.71
  1.8   4.71  0.88  0.86  1.88  2.25  1.28  6.53  5.38  2.62  0.35  2.35
  0.62  6.29  1.53  6.18  3.03  0.3   6.47  3.8   7.32  9.85  2.04  2.7
  1.38  1.37  3.25  2.8   4.56  1.15  1.56  3.31  1.86  0.55  5.39  1.46
 14.8   5.1   6.58  3.93 12.7  13.1   2.08 28.6   1.48  1.3  13.3   4.18
  7.37  1.19  7.91  1.1  11.7   1.49  1.52  5.02 12.1   1.61  2.3   4.34
  9.52  5.04  3.49  0.98  5.63  0.94  3.68  3.77  4.04  1.67 19.5   3.81
  7.85  5.4  16.5   7.23  6.23 14.7   6.19  6.59  1.5   3.71  5.34 13.5
  4.5   6.28 15.    2.15  8.24  1.58  5.47  1.77 12.4   8.57  7.84 20.7
  1.13  5.    4.1   1.27  4.31  1.54  6.34  3.98  3.69  8.93  0.16  7.92
  9.51  2.64  1.97  3.43  5.12  4.2   1.75  1.01 16.2   1.22  0.4   2.05
  4.33  4.21  9.44  3.44  1.55 23.    4.4   6.14]
```

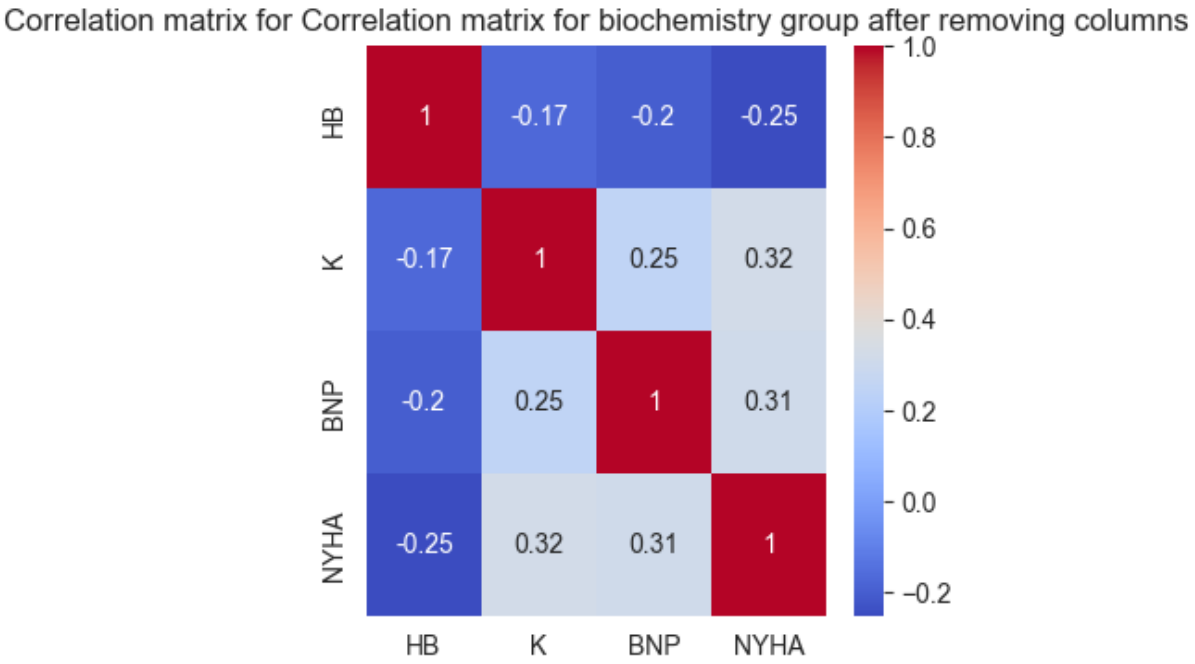
In [28]:

```
# Correlation matrix
plot_corr_matrix(biochemistry_group, 'Correlation matrix for biochemistry group', (

# Check correlation with NYHA
group_checking = prepare_group_checking(biochemistry_group)

# Find important variables and replace all missing values with bifill (use next val
biochemistry_group = biochemistry_group.fillna(method='bfill')

# Prepare data for Decision Tree Classifier
X = biochemistry_group.drop(columns=['NYHA'])
y = biochemistry_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)

group_checking = add_importance_to_group_checking(group_checking, model.feature_imp
group_checking = convert_group_checking_to_dataframe(group_checking)
print(group_checking)

# Remove columns with correlation less than 0.15
columns_to_remove = group_checking[group_checking['correlation'] < 0.25].index
biochemistry_group = biochemistry_group.drop(columns=columns_to_remove)

plot_corr_matrix(biochemistry_group, 'Correlation matrix for biochemistry group aft

X = biochemistry_group.drop(columns=['NYHA'])
y = biochemistry_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)
```

```
grid_search, before_grid_search = create_grid_search(model, X, y)
biochemistry_model = grid_search.best_estimator_

test_with_cross_validation(biochemistry_model, X, y, before_grid_search=before_grid
print_feature_importances(biochemistry_model.feature_importances_, X)
```



	correlation	missing_values	importance
K	0.322426	0	0.210941
BNP	0.312063	0	0.268203
HB	0.252262	0	0.187667
CRP	0.211861	0	0.202588
NA	0.147550	0	0.130601




```
Cross validation score before grid search: 0.46521739130434775
Best parameters: {'class_weight': None, 'criterion': 'gini', 'max_depth': 2, 'min_sa
mples_leaf': 1, 'min_samples_split': 2}
Best score: 0.5695652173913044
Cross validation score after grid search: 0.5630434782608695
Mean cross validation score after optimize: 0.5630434782608695
Growth of cross validation score: 0.09782608695652178
  feature  importance
2     BNP    0.562549
1       K    0.437451
0      HB    0.000000
```

Conclusion from biochemistry group:

From the above table we can see that the most important variables are:

- BNP - level of blood peptide (poziom peptydu BNP) (podwyższony poziom BNP wskazuje na niewydolność serca)
- K - level of potassium (poziom potasu)
- HB - level of hemoglobin (poziom hemoglobiny)

Fitness level group

Fitness level group description

The fitness level group contains variables that describe the patient's fitness level.

Fitness level group variables

- **EXERCISE1** [VAR41] - number - number of seconds needed to complete the task, higher number = worse
- **EXERCISE2** [VAR42] - number - number of repeated movements during the task (higher number = better result)
- **EXERCISE3** [VAR43] - number - number of repeated movements during the task (higher number = better result)
- **6MWT.DIST** [VAR44] - number - number of meters (distance) covered by the patient during the 6 minute walking test (bigger number = better)
- **6MWT.FATIGUE** [VAR45] - number OR categories(?) - the level of fatigue assessed by the patient after walking test (using a scale 0-10)
- **6MWT.DYSPN** [VAR46] - number OR categories(?) - the level of shortness of breath during walking test assessed by the patient (using a scale 0-10)
- **6MWT.SBP1** [VAR47] - number - systolic blood pressure measured before the walking test
- **6MWT.DBP1** [VAR48] - number - diastolic blood pressure measured before the walking test
- **6MWT.HR1** [VAR49] - number - heart rate measured before the walking test
- **6MWT.SBP2** [VAR50] - number - systolic blood pressure measured after the walking test
- **6MWT.DBP2** [VAR51] - number - diastolic blood pressure measured after the walking test
- **6MWT.HR2** [VAR52] - number - heart rate measured after the walking test
- **EXERCISE4** [VAR53] - number - the patient is asked to touch his/her feet with the fingers of the palms. The number represents centimeters. Negative number means: number of centimeters between the patients palm and leg in case of failing to touch the feet (higher number of negative centimeters = worse result). Positive number means that the patient could touch his feet without problems (higher number - better result)
- **EXERCISE5** [VAR54] - number - the patient is asked to touch left palm with the right one but it has to be done at his back (one palm is directed up, the second is directed down). The number represents centimeters. Negative number means: number of centimeters between the patients palms in case of failing to touch both palms (higher number of negative centimeters = worse result). Positive number means that the patient could touch his palms without problems (higher number - better result)
- **CPX.TIME** [VAR55] - number - time of exercise on a treadmill
- **CPX.PEAKVO2** [VAR56] - number - peak oxygen consumption during exercise testing on a treadmill
- **CPX.PEAKVO2FORBM** [VAR57] - number - peak oxygen consumption during exercise testing on a treadmill per body mass
- **RER** [VAR58] - number - respiratory exchange ratio(index of metabolism during an exercise on a treadmill)

- **SLOPE** [VAR59] - number - slope between oxygen and carbon dioxide during exercise testing on a treadmill
- **METS** [VAR60] - number - number of metabolic equivalents (level of work performed by the patient during exercise using a treadmill)
- **WEBER** [VAR61] - categories - The Weber classification: stratification of patients based on peak VO2 and anaerobic threshold to define functional physical capacity. Higher class is worse
- **PEAK>18** [VAR62] - binary - Dividing patients based on a cutoff value of peak oxygen consumption (used for Weber)
- **SLOPE>35** [VAR63] - binary - Dividing patients based on a cutoff value of slope (used for Weber)

Prepare fitness level group

In [29]:
```
# Create fitness_level group
fitness_level_group = data[groups['fitness_level']].copy()
print(f"Fitness level group columns: {fitness_level_group.columns}")
print(f"Missing values: {fitness_level_group.isna().sum()}")
potential_remove_columns = []
# Validation of each column

# EXERCISE1
print(f"EXERCISE1 missing values: {fitness_level_group['EXERCISE1'].isna().sum()}")
print(f"Max value: {fitness_level_group['EXERCISE1'].max()}")
print(f"Min value: {fitness_level_group['EXERCISE1'].min()}")
# Replace missing values with the mean
fitness_level_group['EXERCISE1'] = fitness_level_group['EXERCISE1'].fillna(fitness_

# EXERCISE2
print(f"EXERCISE2 missing values: {fitness_level_group['EXERCISE2'].isna().sum()}")
print(f"Max value: {fitness_level_group['EXERCISE2'].max()}")
print(f"Min value: {fitness_level_group['EXERCISE2'].min()}")
# Replace missing values with the mean
fitness_level_group['EXERCISE2'] = fitness_level_group['EXERCISE2'].fillna(fitness_

# EXERCISE3
print(f"EXERCISE3 missing values: {fitness_level_group['EXERCISE3'].isna().sum()}")
print(f"Max value: {fitness_level_group['EXERCISE3'].max()}")
print(f"Min value: {fitness_level_group['EXERCISE3'].min()}")
# Replace missing values with the mean
fitness_level_group['EXERCISE3'] = fitness_level_group['EXERCISE3'].fillna(fitness_

# 6MWT.DIST
print(f"6MWT.DIST missing values: {fitness_level_group['6MWT.DIST'].isna().sum()}")
print(f"Max value: {fitness_level_group['6MWT.DIST'].max()}")
print(f"Min value: {fitness_level_group['6MWT.DIST'].min()}")
# Replace missing values with the mean
fitness_level_group['6MWT.DIST'] = fitness_level_group['6MWT.DIST'].fillna(fitness_
```

```
# 6MWT.FATIGUE
print(f"6MWT.FATIGUE missing values: {fitness_level_group['6MWT.FATIGUE'].isna().su
print(f"Max value: {fitness_level_group['6MWT.FATIGUE'].max()}")
print(f"Min value: {fitness_level_group['6MWT.FATIGUE'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.FATIGUE')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.FATIGUE'] = fitness_level_group['6MWT.FATIGUE'].fillna(me

# 6MWT.DYSPN
print(f"6MWT.DYSPN missing values: {fitness_level_group['6MWT.DYSPN'].isna().sum()}
print(f"Max value: {fitness_level_group['6MWT.DYSPN'].max()}")
print(f"Min value: {fitness_level_group['6MWT.DYSPN'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.DYSPN')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.DYSPN'] = fitness_level_group['6MWT.DYSPN'].fillna(method

# 6MWT.SBP1
print(f"6MWT.SBP1 missing values: {fitness_level_group['6MWT.SBP1'].isna().sum()}")
print(f"Max value: {fitness_level_group['6MWT.SBP1'].max()}")
print(f"Min value: {fitness_level_group['6MWT.SBP1'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.SBP1')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.SBP1'] = fitness_level_group['6MWT.SBP1'].fillna(method='

# 6MWT.DBP1
print(f"6MWT.DBP1 missing values: {fitness_level_group['6MWT.DBP1'].isna().sum()}")
print(f"Max value: {fitness_level_group['6MWT.DBP1'].max()}")
print(f"Min value: {fitness_level_group['6MWT.DBP1'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.DBP1')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.DBP1'] = fitness_level_group['6MWT.DBP1'].fillna(method='

# 6MWT.HR1
print(f"6MWT.HR1 missing values: {fitness_level_group['6MWT.HR1'].isna().sum()}")
print(f"Max value: {fitness_level_group['6MWT.HR1'].max()}")
print(f"Min value: {fitness_level_group['6MWT.HR1'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.HR1')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.HR1'] = fitness_level_group['6MWT.HR1'].fillna(method='bf

# 6MWT.SBP2
print(f"6MWT.SBP2 missing values: {fitness_level_group['6MWT.SBP2'].isna().sum()}")
print(f"Max value: {fitness_level_group['6MWT.SBP2'].max()}")
print(f"Min value: {fitness_level_group['6MWT.SBP2'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.SBP2')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.SBP2'] = fitness_level_group['6MWT.SBP2'].fillna(method='

# 6MWT.DBP2
print(f"6MWT.DBP2 missing values: {fitness_level_group['6MWT.DBP2'].isna().sum()}")
```

```
print(f"Max value: {fitness_level_group['6MWT.DBP2'].max()}")
print(f"Min value: {fitness_level_group['6MWT.DBP2'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.DBP2')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.DBP2'] = fitness_level_group['6MWT.DBP2'].fillna(method='

# 6MWT.HR2
print(f"6MWT.HR2 missing values: {fitness_level_group['6MWT.HR2'].isna().sum()}")
print(f"Max value: {fitness_level_group['6MWT.HR2'].max()}")
print(f"Min value: {fitness_level_group['6MWT.HR2'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('6MWT.HR2')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['6MWT.HR2'] = fitness_level_group['6MWT.HR2'].fillna(method='bf

# EXERCISE4
print(f"EXERCISE4 missing values: {fitness_level_group['EXERCISE4'].isna().sum()}")
print(f"Max value: {fitness_level_group['EXERCISE4'].max()}")
print(f"Min value: {fitness_level_group['EXERCISE4'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('EXERCISE4')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['EXERCISE4'] = fitness_level_group['EXERCISE4'].fillna(method='

# EXERCISE5
print(f"EXERCISE5 missing values: {fitness_level_group['EXERCISE5'].isna().sum()}")
print(f"Max value: {fitness_level_group['EXERCISE5'].max()}")
print(f"Min value: {fitness_level_group['EXERCISE5'].min()}")
#  Add to potential remove columns because of high number of missing values
potential_remove_columns.append('EXERCISE5')
# Replace missing values with bfill (use next valid observation to fill gap)
fitness_level_group['EXERCISE5'] = fitness_level_group['EXERCISE5'].fillna(method='

# CPX.TIME
print(f"CPX.TIME missing values: {fitness_level_group['CPX.TIME'].isna().sum()}")
print(f"Max value: {fitness_level_group['CPX.TIME'].max()}")
print(f"Min value: {fitness_level_group['CPX.TIME'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('CPX.TIME')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['CPX.TIME'] = fitness_level_group['CPX.TIME'].fillna(method='bf
fitness_level_group['CPX.TIME'] = fitness_level_group['CPX.TIME'].fillna(method='ff

# CPX.PEAKVO2
print(f"CPX.PEAKVO2 missing values: {fitness_level_group['CPX.PEAKVO2'].isna().sum(
print(f"Max value: {fitness_level_group['CPX.PEAKVO2'].max()}")
print(f"Min value: {fitness_level_group['CPX.PEAKVO2'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('CPX.PEAKVO2')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['CPX.PEAKVO2'] = fitness_level_group['CPX.PEAKVO2'].fillna(meth
fitness_level_group['CPX.PEAKVO2'] = fitness_level_group['CPX.PEAKVO2'].fillna(meth

# CPX.PEAKVO2FORBM
print(f"CPX.PEAKVO2FORBM missing values: {fitness_level_group['CPX.PEAKVO2FORBM'].i
```

```
print(f"Max value: {fitness_level_group['CPX.PEAKVO2FORBM'].max()}")
print(f"Min value: {fitness_level_group['CPX.PEAKVO2FORBM'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('CPX.PEAKVO2FORBM')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['CPX.PEAKVO2FORBM'] = fitness_level_group['CPX.PEAKVO2FORBM'].f
fitness_level_group['CPX.PEAKVO2FORBM'] = fitness_level_group['CPX.PEAKVO2FORBM'].f

# RER
print(f"RER missing values: {fitness_level_group['RER'].isna().sum()}")
print(f"Max value: {fitness_level_group['RER'].max()}")
print(f"Min value: {fitness_level_group['RER'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('RER')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['RER'] = fitness_level_group['RER'].fillna(method='bfill')
fitness_level_group['RER'] = fitness_level_group['RER'].fillna(method='ffill')

# SLOPE
print(f"SLOPE missing values: {fitness_level_group['SLOPE'].isna().sum()}")
print(f"Max value: {fitness_level_group['SLOPE'].max()}")
print(f"Min value: {fitness_level_group['SLOPE'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('SLOPE')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['SLOPE'] = fitness_level_group['SLOPE'].fillna(method='bfill')
fitness_level_group['SLOPE'] = fitness_level_group['SLOPE'].fillna(method='ffill')

# METS
print(f"METS missing values: {fitness_level_group['METS'].isna().sum()}")
print(f"Max value: {fitness_level_group['METS'].max()}")
print(f"Min value: {fitness_level_group['METS'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('METS')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['METS'] = fitness_level_group['METS'].fillna(method='bfill')
fitness_level_group['METS'] = fitness_level_group['METS'].fillna(method='ffill')

# WEBER
print(f"WEBER missing values: {fitness_level_group['WEBER'].isna().sum()}")
print(f"Max value: {fitness_level_group['WEBER'].max()}")
print(f"Min value: {fitness_level_group['WEBER'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('WEBER')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['WEBER'] = fitness_level_group['WEBER'].fillna(method='bfill')
fitness_level_group['WEBER'] = fitness_level_group['WEBER'].fillna(method='ffill')

# PEAK>18
print(f"PEAK>18 missing values: {fitness_level_group['PEAK>18'].isna().sum()}")
print(f"Max value: {fitness_level_group['PEAK>18'].max()}")
print(f"Min value: {fitness_level_group['PEAK>18'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('PEAK>18')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['PEAK>18'] = fitness_level_group['PEAK>18'].fillna(method='bfil
```

```
fitness_level_group['PEAK>18'] = fitness_level_group['PEAK>18'].fillna(method='ffil

# SLOPE>35
print(f"SLOPE>35 missing values: {fitness_level_group['SLOPE>35'].isna().sum()}")
print(f"Max value: {fitness_level_group['SLOPE>35'].max()}")
print(f"Min value: {fitness_level_group['SLOPE>35'].min()}")
# Add to potential remove columns because of high number of missing values
potential_remove_columns.append('SLOPE>35')
# Replace missing values with bfill (use next valid observation to fill gap) and ff
fitness_level_group['SLOPE>35'] = fitness_level_group['SLOPE>35'].fillna(method='bf
fitness_level_group['SLOPE>35'] = fitness_level_group['SLOPE>35'].fillna(method='ff

# Add NYHA target variable
fitness_level_group['NYHA'] = NYHA

# Print missing values
print(f"Missing values: {fitness_level_group.isna().sum().sum()}")
```

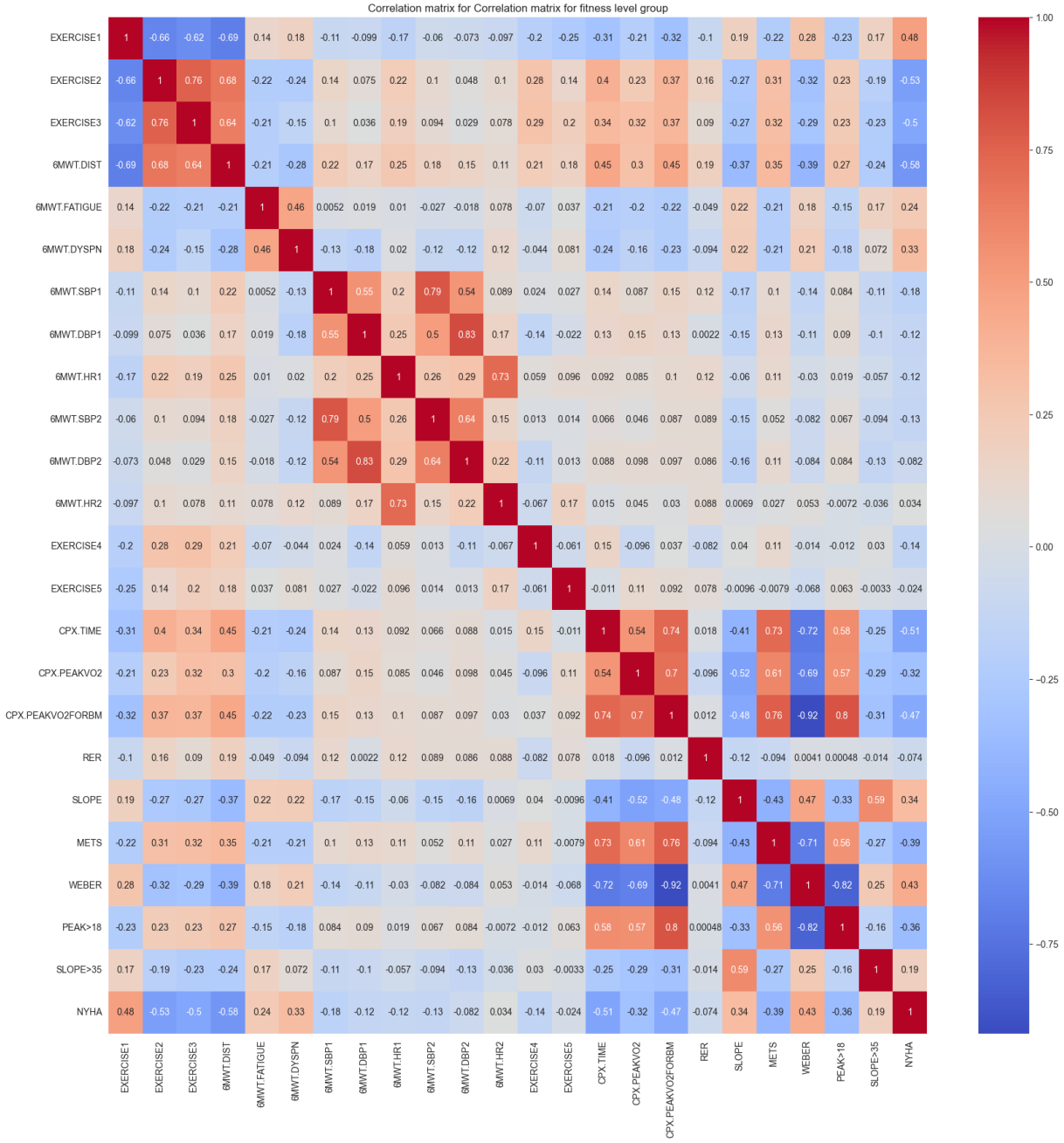
```
Fitness level group columns: Index(['EXERCISE1', 'EXERCISE2', 'EXERCISE3', '6MWT.DIS
T', '6MWT.FATIGUE',
       '6MWT.DYSPN', '6MWT.SBP1', '6MWT.DBP1', '6MWT.HR1', '6MWT.SBP2',
       '6MWT.DBP2', '6MWT.HR2', 'EXERCISE4', 'EXERCISE5', 'CPX.TIME',
       'CPX.PEAKVO2', 'CPX.PEAKVO2FORBM', 'RER', 'SLOPE', 'METS', 'WEBER',
       'PEAK>18', 'SLOPE>35'],
      dtype='object')
Missing values: EXERCISE1             6
EXERCISE2             6
EXERCISE3             6
6MWT.DIST             5
6MWT.FATIGUE        103
6MWT.DYSPN          103
6MWT.SBP1            47
6MWT.DBP1            48
6MWT.HR1             49
6MWT.SBP2            48
6MWT.DBP2            48
6MWT.HR2             51
EXERCISE4            69
EXERCISE5            72
CPX.TIME            114
CPX.PEAKVO2         213
CPX.PEAKVO2FORBM     51
RER                  82
SLOPE                52
METS                199
WEBER                51
PEAK>18              51
SLOPE>35             85
dtype: int64
EXERCISE1 missing values: 6
Max value: 16.1
Min value: 3.1
EXERCISE2 missing values: 6
Max value: 27.0
Min value: 2.0
EXERCISE3 missing values: 6
Max value: 29.0
Min value: 1.0
6MWT.DIST missing values: 5
Max value: 780.0
Min value: 90.0
6MWT.FATIGUE missing values: 103
Max value: 10.0
Min value: 0.0
6MWT.DYSPN missing values: 103
Max value: 10.0
Min value: 0.0
6MWT.SBP1 missing values: 47
Max value: 220.0
Min value: 15.0
6MWT.DBP1 missing values: 48
Max value: 160.0
Min value: 40.0
6MWT.HR1 missing values: 49
```

```
Max value: 136.0
Min value: 50.0
6MWT.SBP2 missing values: 48
Max value: 210.0
Min value: 70.0
6MWT.DBP2 missing values: 48
Max value: 160.0
Min value: 40.0
6MWT.HR2 missing values: 51
Max value: 126.0
Min value: 40.0
EXERCISE4 missing values: 69
Max value: 30.0
Min value: -27.0
EXERCISE5 missing values: 72
Max value: 24.0
Min value: -54.0
CPX.TIME missing values: 114
Max value: 19.8333333333333
Min value: 1.76666666666667
CPX.PEAKVO2 missing values: 213
Max value: 3210.4
Min value: 467.25
CPX.PEAKVO2FORBM missing values: 51
Max value: 35.59
Min value: 4.7125
RER missing values: 82
Max value: 1.91
Min value: 0.687777777777778
SLOPE missing values: 52
Max value: 117.157846597311
Min value: 18.8287225209598
METS missing values: 199
Max value: 10.1685714285714
Min value: 1.0
WEBER missing values: 51
Max value: 4.0
Min value: 1.0
PEAK>18 missing values: 51
Max value: 1.0
Min value: 0.0
SLOPE>35 missing values: 85
Max value: 1.0
Min value: 0.0
Missing values: 0
```

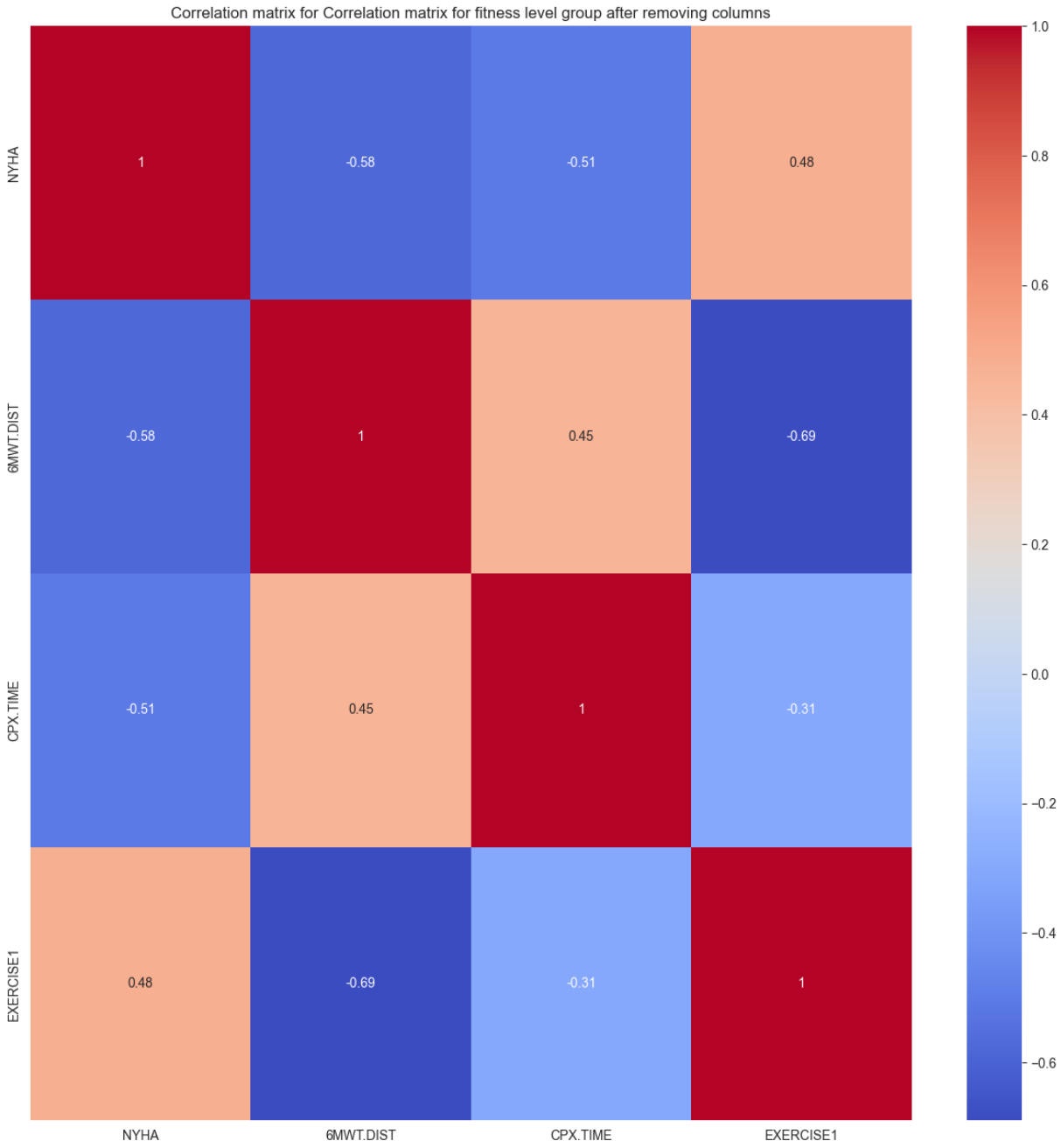
In [30]:

```python
# Correlation matrix
plot_corr_matrix(fitness_level_group, 'Correlation matrix for fitness level group',

# Check correlation with NYHA
group_checking = prepare_group_checking(fitness_level_group)

# Prepare data for Decision Tree Classifier
X = fitness_level_group.drop(columns=['NYHA'])
y = fitness_level_group['NYHA']
model = DecisionTreeClassifier()
```

```
model.fit(X, y)

group_checking = add_importance_to_group_checking(group_checking, model.feature_imp
group_checking = convert_group_checking_to_dataframe(group_checking)
print(group_checking)

# Remove columns with correlation less than 0.15
columns_to_remove = group_checking[group_checking['correlation'] < 0.35].index
fitness_level_group = fitness_level_group.drop(columns=columns_to_remove)

# Use only NYHA, 6MWT.DIST, CPX.TIME, EXERCISE1 columns
fitness_level_group = fitness_level_group[['NYHA', '6MWT.DIST', 'CPX.TIME', 'EXERCI

plot_corr_matrix(fitness_level_group, 'Correlation matrix for fitness level group a

X = fitness_level_group.drop(columns=['NYHA'])
y = fitness_level_group['NYHA']
model = DecisionTreeClassifier()
model.fit(X, y)

grid_search, before_grid_search = create_grid_search(model, X, y)
fitness_level_model = grid_search.best_estimator_

test_with_cross_validation(fitness_level_model, X, y, before_grid_search=before_gri
print_feature_importances(fitness_level_model.feature_importances_, X)
```


Correlation matrix for Correlation matrix for fitness level group
EXERCISE1 1 -0.66 -0.62 -0.69 0.14 0.18 -0.11 -0.099 -0.17 -0.06 -0.073 -0.097 -0.2 -0.25 -0.31 -0.21 -0.32 -0.1 0.19 -0.22 0.28 -0.23 0.17 0.48
EXERCISE2 -0.66 1 0.76 0.68 -0.22 -0.24 0.14 0.075 0.22 0.1 0.048 0.1 0.28 0.14 0.4 0.23 0.37 0.16 -0.27 0.31 -0.32 0.23 -0.19 -0.53
EXERCISE3 -0.62 0.76 1 0.64 -0.21 -0.15 0.1 0.036 0.19 0.094 0.029 0.078 0.29 0.2 0.34 0.32 0.37 0.09 -0.27 0.32 -0.29 0.23 -0.23 -0.5
6MWT.DIST -0.69 0.68 0.64 1 -0.21 -0.28 0.22 0.17 0.25 0.18 0.15 0.11 0.21 0.18 0.45 0.3 0.45 0.19 -0.37 0.35 -0.39 0.27 -0.24 -0.58
6MWT.FATIGUE 0.14 -0.22 -0.21 -0.21 1 0.46 0.0052 0.019 0.01 -0.027 -0.018 0.078 -0.07 0.037 -0.21 -0.2 -0.22 -0.049 0.22 -0.21 0.18 -0.15 0.17 0.24
6MWT.DYSPN 0.18 -0.24 -0.15 -0.28 0.46 1 -0.13 -0.18 0.02 -0.12 -0.12 0.12 -0.044 0.081 -0.24 -0.16 -0.23 -0.094 0.22 -0.21 0.21 -0.18 0.072 0.33
6MWT.SBP1 -0.11 0.14 0.1 0.22 0.0052 -0.13 1 0.55 0.2 0.79 0.54 0.089 0.024 0.027 0.14 0.087 0.15 0.12 -0.17 0.1 -0.14 0.084 -0.11 -0.18
6MWT.DBP1 -0.099 0.075 0.036 0.17 0.019 -0.18 0.55 1 0.25 0.5 0.83 0.17 -0.14 -0.022 0.13 0.15 0.13 0.0022 -0.15 0.13 -0.11 0.09 -0.1 -0.12
6MWT.HR1 -0.17 0.22 0.19 0.25 0.01 0.02 0.2 0.25 1 0.26 0.29 0.73 0.059 0.096 0.092 0.085 0.1 0.12 -0.06 0.11 -0.03 0.019 -0.057 -0.12
6MWT.SBP2 -0.06 0.1 0.094 0.18 -0.027 -0.12 0.79 0.5 0.26 1 0.64 0.15 0.013 0.014 0.066 0.046 0.087 0.089 -0.15 0.052 -0.082 0.067 -0.094 -0.13
6MWT.DBP2 -0.073 0.048 0.029 0.15 -0.018 -0.12 0.54 0.83 0.29 0.64 1 0.22 -0.11 0.013 0.088 0.098 0.097 0.086 -0.16 0.11 -0.084 0.084 -0.13 -0.082
6MWT.HR2 -0.097 0.1 0.078 0.11 0.078 0.12 0.089 0.17 0.73 0.15 0.22 1 -0.067 0.17 0.015 0.045 0.03 0.088 0.0069 0.027 0.053 -0.0072 -0.036 0.034
EXERCISE4 -0.2 0.28 0.29 0.21 -0.07 -0.044 0.024 -0.14 0.059 0.013 -0.11 -0.067 1 -0.061 0.15 -0.096 0.037 -0.082 0.04 0.11 -0.014 -0.012 0.03 -0.14
EXERCISE5 -0.25 0.14 0.2 0.18 0.037 0.081 0.027 -0.022 0.096 0.014 0.013 0.17 -0.061 1 -0.011 0.11 0.092 0.078 -0.0096 -0.0079 -0.068 0.063 -0.0033 -0.024
CPX.TIME -0.31 0.4 0.34 0.45 -0.21 -0.24 0.14 0.13 0.092 0.066 0.088 0.015 0.15 -0.011 1 0.54 0.74 0.018 -0.41 0.73 -0.72 0.58 -0.25 -0.51
CPX.PEAKVO2 -0.21 0.23 0.32 0.3 -0.2 -0.16 0.087 0.15 0.085 0.046 0.098 0.045 -0.096 0.11 0.54 1 0.7 -0.096 -0.52 0.61 -0.69 0.57 -0.29 -0.32
CPX.PEAKVO2FORBM -0.32 0.37 0.37 0.45 -0.22 -0.23 0.15 0.13 0.1 0.087 0.097 0.03 0.037 0.092 0.74 0.7 1 0.012 -0.48 0.76 -0.92 0.8 -0.31 -0.47
RER -0.1 0.16 0.09 0.19 -0.049 -0.094 0.12 0.0022 0.12 0.089 0.086 0.088 -0.082 0.078 0.018 -0.096 0.012 1 -0.12 -0.094 0.0041 0.00048 -0.014 -0.074
SLOPE 0.19 -0.27 -0.27 -0.37 0.22 0.22 -0.17 -0.15 -0.06 -0.15 -0.16 0.0069 0.04 -0.0096 -0.41 -0.52 -0.48 -0.12 1 -0.43 0.47 -0.33 0.59 0.34
METS -0.22 0.31 0.32 0.35 -0.21 -0.21 0.1 0.13 0.11 0.052 0.11 0.027 0.11 -0.0079 0.73 0.61 0.76 -0.094 -0.43 1 -0.71 0.56 -0.27 -0.39
WEBER 0.28 -0.32 -0.29 -0.39 0.18 0.21 -0.14 -0.11 -0.03 -0.082 -0.084 0.053 -0.014 -0.068 -0.72 -0.69 -0.92 0.0041 0.47 -0.71 1 -0.82 0.25 0.43
PEAK>18 -0.23 0.23 0.23 0.27 -0.15 -0.18 0.084 0.09 0.019 0.067 0.084 -0.0072 -0.012 0.063 0.58 0.57 0.8 0.00048 -0.33 0.56 -0.82 1 -0.16 -0.36
SLOPE>35 0.17 -0.19 -0.23 -0.24 0.17 0.072 -0.11 -0.1 -0.057 -0.094 -0.13 -0.036 0.03 -0.0033 -0.25 -0.29 -0.31 -0.014 0.59 -0.27 0.25 -0.16 1 0.19
NYHA 0.48 -0.53 -0.5 -0.58 0.24 0.33 -0.18 -0.12 -0.12 -0.13 -0.082 0.034 -0.14 -0.024 -0.51 -0.32 -0.47 -0.074 0.34 -0.39 0.43 -0.36 0.19 1
EXERCISE1 EXERCISE2 EXERCISE3 6MWT.DIST 6MWT.FATIGUE 6MWT.DYSPN 6MWT.SBP1 6MWT.DBP1 6MWT.HR1 6MWT.SBP2 6MWT.DBP2 6MWT.HR2 EXERCISE4 EXERCISE5 CPX.TIME CPX.PEAKVO2 CPX.PEAKVO2FORBM RER SLOPE METS WEBER PEAK>18 SLOPE>35 NYHA
1.00 0.75 0.50 0.25 0.00 −0.25 −0.50 −0.75

	correlation	missing_values	importance
6MWT.DIST	0.579540	0	0.171144
EXERCISE2	0.534651	0	0.022418
CPX.TIME	0.505976	0	0.101883
EXERCISE3	0.498229	0	0.028769
EXERCISE1	0.483570	0	0.070102
CPX.PEAKVO2FORBM	0.469179	0	0.040339
WEBER	0.428791	0	0.009330
METS	0.390270	0	0.054597
PEAK>18	0.358438	0	0.000000
SLOPE	0.339074	0	0.067794
6MWT.DYSPN	0.331550	0	0.045053
CPX.PEAKVO2	0.323248	0	0.055580
6MWT.FATIGUE	0.238910	0	0.036899
SLOPE>35	0.193296	0	0.000000
6MWT.SBP1	0.176010	0	0.005353
EXERCISE4	0.139605	0	0.051576
6MWT.SBP2	0.128836	0	0.027528
6MWT.DBP1	0.124792	0	0.011426
6MWT.HR1	0.120863	0	0.085219
6MWT.DBP2	0.081639	0	0.019203
RER	0.074310	0	0.029109
6MWT.HR2	0.034155	0	0.039359
EXERCISE5	0.024325	0	0.027319




```
Cross validation score before grid search: 0.5260869565217392
Best parameters: {'class_weight': None, 'criterion': 'gini', 'max_depth': 3, 'min_sa
mples_leaf': 5, 'min_samples_split': 2}
Best score: 0.6130434782608696
Cross validation score after grid search: 0.5978260869565217
Mean cross validation score after optimize: 0.5978260869565217
Growth of cross validation score: 0.07173913043478253
     feature  importance
0  6MWT.DIST    0.488246
1   CPX.TIME    0.381552
2  EXERCISE1    0.130201
```

Conclusion from fitness level group:

From the above table we can see that the most important variables are:

- 6MWT.DIST - distance in 6 minute walk test (dystans w teście 6 minutowego marszu)
- CPX.TIME - time of exercise on a treadmill (czas ćwiczeń na bieżni)
- EXERCISE1 - number of seconds needed to complete the task, higher number = worse) (liczba sekund potrzebnych do wykonania zadania, im wyższa liczba tym gorzej)

2. Training and improving the model

Some approach to create the model.

1. Create the model per each group of variables and then use this models together to create the ensemble model.
2. Create the model using only variables that are correlated with target variable which are representative for the model.

All of this approaches will be tested and compared to each other.

Help functions

In [31]:
```
import os
from sklearn.tree import plot_tree
from sklearn.metrics import confusion_matrix
from sklearn.tree import _tree


# https://mljar.com/blog/extract-rules-decision-tree/
def get_rules(tree, feature_names, class_names):
    tree_ = tree.tree_
    feature_name = [
        feature_names[i] if i != _tree.TREE_UNDEFINED else "undefined!"
        for i in tree_.feature
    ]

    paths = []
    path = []

    def recurse(node, path, paths):

        if tree_.feature[node] != _tree.TREE_UNDEFINED:
            name = feature_name[node]
            threshold = tree_.threshold[node]
            p1, p2 = list(path), list(path)
            p1 += [f"({name} <= {np.round(threshold, 3)})"]
            recurse(tree_.children_left[node], p1, paths)
            p2 += [f"({name} > {np.round(threshold, 3)})"]
            recurse(tree_.children_right[node], p2, paths)
        else:
            path += [(tree_.value[node], tree_.n_node_samples[node])]
            paths += [path]
```

```
    recurse(0, path, paths)

    # sort by samples count
    samples_count = [p[-1][1] for p in paths]
    ii = list(np.argsort(samples_count))
    paths = [paths[i] for i in reversed(ii)]

    rules = []
    for path in paths:
        rule = "if "

        for p in path[:-1]:
            if rule != "if ":
                rule += " AND "
            rule += str(p)
        rule += " then "
        if class_names is None:
            rule += "response: " + str(np.round(path[-1][0][0][0], 3))
        else:
            classes = path[-1][0][0]
            l = np.argmax(classes)
            rule += f"(class={class_names[l]}) (proba: {np.round(100.0 * classes[l]
        rule += f" | based on {path[-1][1]:,} samples"
        rules += [rule]

    return rules


def plot_decision_tree(model, X, figsize=(20, 10), dpi=1000, filename=None):
    plt.figure(figsize=figsize, dpi=dpi)

    filename = filename if filename else f"max_depth_{model.max_depth}_min_samples_
    plot_tree(model, feature_names=X.columns, filled=True, rounded=True,
              class_names=["NYHA I", "NYHA II", "NYHA III", "NYHA IV"]);
    plt.title(f"Decision tree for {filename}")
    plt.savefig(f"plots/{filename}.png", dpi=dpi)


def plot_confusion_matrix(model, X, y, figsize=(10, 6), dpi=1000, filename=None):
    preds = model.predict(X)
    plt.figure(figsize=figsize, dpi=dpi)
    sns.heatmap(confusion_matrix(y, preds), annot=True, cmap="viridis", fmt="g")
    plt.xlabel("Predicted label")
    plt.ylabel("True label")
    plt.title("Confusion matrix")
    filename = filename if filename else f"max_depth_{model.max_depth}_min_samples_
    plt.savefig(f"plots/confusion_matrix_{filename}.png", dpi=dpi)


def save_rules_to_file(filename, rules: str):
    # create a directory for rules
    if not os.path.exists("rules"):
        os.makedirs("rules")

    # save rules to file
```

```
    with open(f"rules/{filename}.txt", "w") as f:
        f.write(rules)


def plot_confusion_matrix_for_predictions(y_true, y_pred, figsize=(20, 10), dpi=100
    plt.figure(figsize=figsize, dpi=dpi)
    sns.heatmap(confusion_matrix(y_true, y_pred), annot=True, cmap="viridis", fmt="
    plt.xlabel("Predicted label")
    plt.ylabel("True label")
    plt.title("Confusion matrix")
    filename = filename if filename else f"max_depth_{model.max_depth}_min_samples_
    plt.savefig(f"plots/confusion_matrix_{filename}.png", dpi=dpi)
```

2.1. Prepare data

In [32]:
```
# Create new dataframe with chosen variables
df = pd.DataFrame()
df['NYHA'] = NYHA

columns_to_add = clinical_group.columns
for column in columns_to_add:
    df[column] = clinical_group[column]

df['AGE'] = demographic_group['AGE']
df['AGE'] = df['AGE'].fillna(df['AGE'].mean())

columns_to_add = comorbidities_group.columns
for column in columns_to_add:
    df[column] = comorbidities_group[column]

columns_to_add = treatment_group.columns
for column in columns_to_add:
    df[column] = treatment_group[column]

columns_to_add = fitness_level_group.columns
for column in columns_to_add:
    df[column] = fitness_level_group[column]
```

2.2. Create ensemble model from all groups

In [41]:
```
from sklearn.ensemble import VotingClassifier

# Remove all date columns
print(data.columns)
X = df.drop(columns=['NYHA'])
y = df['NYHA']

estimators = [('treatment', treatment_model), ('biochemistry', biochemistry_model),
              ('comorbidities_model', comorbidities_model), ('fitness_level', fitne
voting_model = VotingClassifier(
    estimators=estimators)
voting_model.fit(X, y)
```

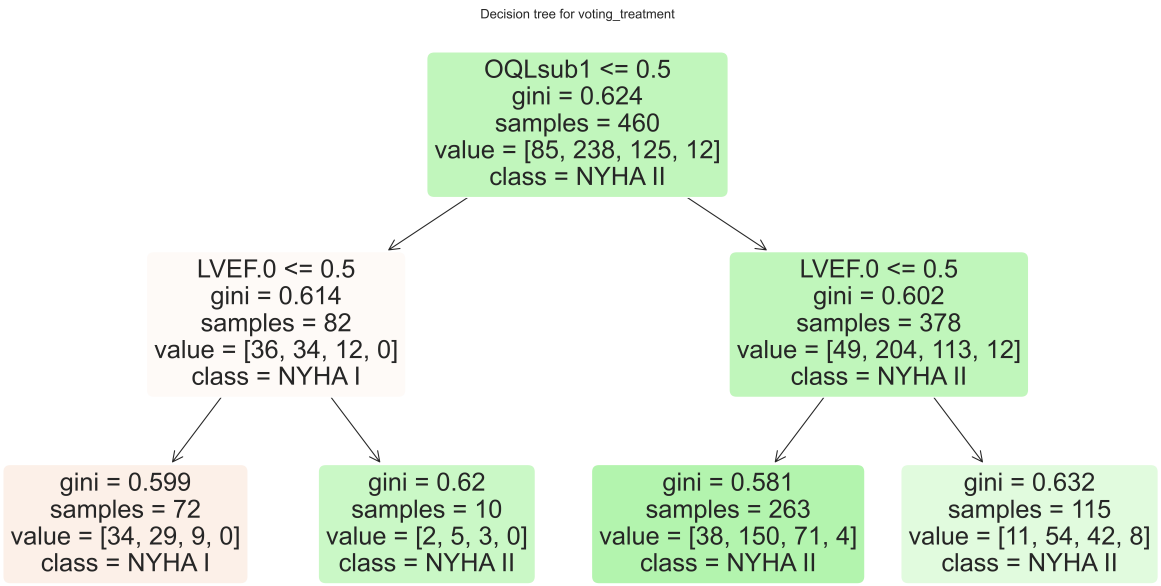
```python
# Find the best parameters for voting model
param_grid = {
    'voting': ['hard', 'soft'],
    'flatten_transform': [True, False],
}

grid_search = GridSearchCV(estimator=voting_model, param_grid=param_grid, cv=100)
grid_search.fit(X, y)
voting_model = grid_search.best_estimator_

print(f"Best voting model parameters: {grid_search.best_params_}")
print(f"Best voting model score: {grid_search.best_score_}")

# Save rules and plot decision tree for each estimator
for estimator in estimators:
    print(f"==================== {estimator[0]} ====================")
    rules = get_rules(estimator[1], list(X.columns), ['NYHA 1', 'NYHA 2', 'NYHA 3',
    rules_txt = ""
    for r in rules:
        rules_txt += r + "\n"
    save_rules_to_file(f"voting_{estimator[0]}", rules_txt)
    plot_decision_tree(estimator[1], X, filename=f"voting_{estimator[0]}")
    print(rules_txt)
    print(f"=========================================================")
    # plot_confusion_matrix(estimator[1], X, y, filename=f"voting_{estimator[0]}")
```

```
Index(['DEATH?', 'DEATHDATE', 'TIMEFU', 'QOL', 'OQLsub1', 'OQLsub2', 'DOB',
       'DOE', 'AGE', 'HEIGHT.CM', 'WEIGHT.KG', 'BMI', 'LVEF.0', 'NYHA', 'PM',
       'AETH.HF', 'MI', 'AF', 'DM', 'HT', 'COPD', 'STROKE', 'KIDNEY.DIS',
       'ACEI.ARB', 'BB', 'MRA', 'DIUR', 'ANTIPLAT', 'STATIN', 'DIGOX', 'HB',
       'NA', 'K', 'BNP', 'CRP', 'LVEDD', 'MR', 'REST.SBP', 'REST.DBP',
       'REST.HR', 'EXERCISE1', 'EXERCISE2', 'EXERCISE3', '6MWT.DIST',
       '6MWT.FATIGUE', '6MWT.DYSPN', '6MWT.SBP1', '6MWT.DBP1', '6MWT.HR1',
       '6MWT.SBP2', '6MWT.DBP2', '6MWT.HR2', 'EXERCISE4', 'EXERCISE5',
       'CPX.TIME', 'CPX.PEAKVO2', 'CPX.PEAKVO2FORBM', 'RER', 'SLOPE', 'METS',
       'WEBER', 'PEAK>18', 'SLOPE>35'],
      dtype='object')
```

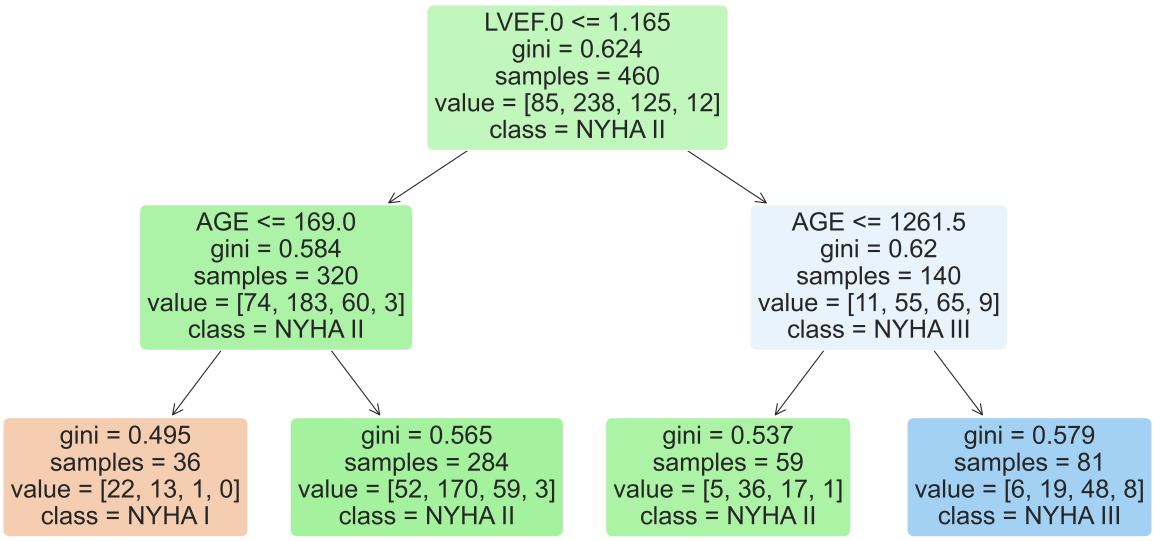
```
C:\Users\Krystian\work\task\HeartBit\venv\lib\site-packages\sklearn\model_selection
\_split.py:700: UserWarning: The least populated class in y has only 12 members, whi
ch is less than n_splits=100.
  warnings.warn(
```

```
Best voting model parameters: {'flatten_transform': True, 'voting': 'soft'}
Best voting model score: 0.618
==================== treatment ====================
if (OQLsub1 > 0.5) AND (LVEF.0 <= 0.5) then (class=NYHA 2) (proba: 57.03%) | based o
n 263 samples
if (OQLsub1 > 0.5) AND (LVEF.0 > 0.5) then (class=NYHA 2) (proba: 46.96%) | based on
115 samples
if (OQLsub1 <= 0.5) AND (LVEF.0 <= 0.5) then (class=NYHA 1) (proba: 47.22%) | based
on 72 samples
if (OQLsub1 <= 0.5) AND (LVEF.0 > 0.5) then (class=NYHA 2) (proba: 50.0%) | based on
10 samples

======================================================
==================== biochemistry ====================
if (LVEF.0 <= 1.165) AND (AGE > 169.0) then (class=NYHA 2) (proba: 59.86%) | based o
n 284 samples
if (LVEF.0 > 1.165) AND (AGE > 1261.5) then (class=NYHA 3) (proba: 59.26%) | based o
n 81 samples
if (LVEF.0 > 1.165) AND (AGE <= 1261.5) then (class=NYHA 2) (proba: 61.02%) | based
on 59 samples
if (LVEF.0 <= 1.165) AND (AGE <= 169.0) then (class=NYHA 1) (proba: 61.11%) | based
on 36 samples

======================================================
==================== clinical ====================
if (LVEF.0 <= 29.5) AND (AGE > 29.5) AND (AF > 1.25) AND (AGE <= 44.5) then (class=N
YHA 2) (proba: 66.94%) | based on 124 samples
if (LVEF.0 <= 29.5) AND (AGE <= 29.5) AND (LVEF.0 > 2.5) AND (LVEF.0 <= 26.0) then
(class=NYHA 2) (proba: 46.88%) | based on 64 samples
if (LVEF.0 <= 29.5) AND (AGE <= 29.5) AND (LVEF.0 <= 2.5) AND (AGE <= 27.5) then (cl
ass=NYHA 2) (proba: 61.9%) | based on 42 samples
if (LVEF.0 > 29.5) AND (OQLsub1 > 0.5) AND (AF <= 2.25) AND (LVEF.0 > 35.5) then (cl
ass=NYHA 3) (proba: 52.63%) | based on 38 samples
if (LVEF.0 > 29.5) AND (OQLsub1 > 0.5) AND (AF > 2.25) AND (AGE <= 34.5) then (class
=NYHA 3) (proba: 75.68%) | based on 37 samples
if (LVEF.0 <= 29.5) AND (AGE > 29.5) AND (AF <= 1.25) AND (AGE <= 35.5) then (class=
NYHA 2) (proba: 57.14%) | based on 35 samples
if (LVEF.0 <= 29.5) AND (AGE > 29.5) AND (AF <= 1.25) AND (AGE > 35.5) then (class=N
YHA 1) (proba: 50.0%) | based on 30 samples
if (LVEF.0 > 29.5) AND (OQLsub1 <= 0.5) AND (AF <= 1.75) AND (LVEF.0 > 33.5) then (c
lass=NYHA 3) (proba: 50.0%) | based on 20 samples
if (LVEF.0 > 29.5) AND (OQLsub1 <= 0.5) AND (AF > 1.75) AND (LVEF.0 > 39.5) then (cl
ass=NYHA 2) (proba: 64.71%) | based on 17 samples
if (LVEF.0 <= 29.5) AND (AGE > 29.5) AND (AF > 1.25) AND (AGE > 44.5) then (class=NY
HA 1) (proba: 46.15%) | based on 13 samples
if (LVEF.0 > 29.5) AND (OQLsub1 <= 0.5) AND (AF > 1.75) AND (LVEF.0 <= 39.5) then (c
lass=NYHA 2) (proba: 88.89%) | based on 9 samples
if (LVEF.0 <= 29.5) AND (AGE <= 29.5) AND (LVEF.0 <= 2.5) AND (AGE > 27.5) then (cla
ss=NYHA 2) (proba: 44.44%) | based on 9 samples
if (LVEF.0 <= 29.5) AND (AGE <= 29.5) AND (LVEF.0 > 2.5) AND (LVEF.0 > 26.0) then (c
lass=NYHA 2) (proba: 75.0%) | based on 8 samples
if (LVEF.0 > 29.5) AND (OQLsub1 > 0.5) AND (AF > 2.25) AND (AGE > 34.5) then (class=
NYHA 2) (proba: 42.86%) | based on 7 samples
if (LVEF.0 > 29.5) AND (OQLsub1 > 0.5) AND (AF <= 2.25) AND (LVEF.0 <= 35.5) then (c
lass=NYHA 3) (proba: 100.0%) | based on 4 samples
if (LVEF.0 > 29.5) AND (OQLsub1 <= 0.5) AND (AF <= 1.75) AND (LVEF.0 <= 33.5) then
```

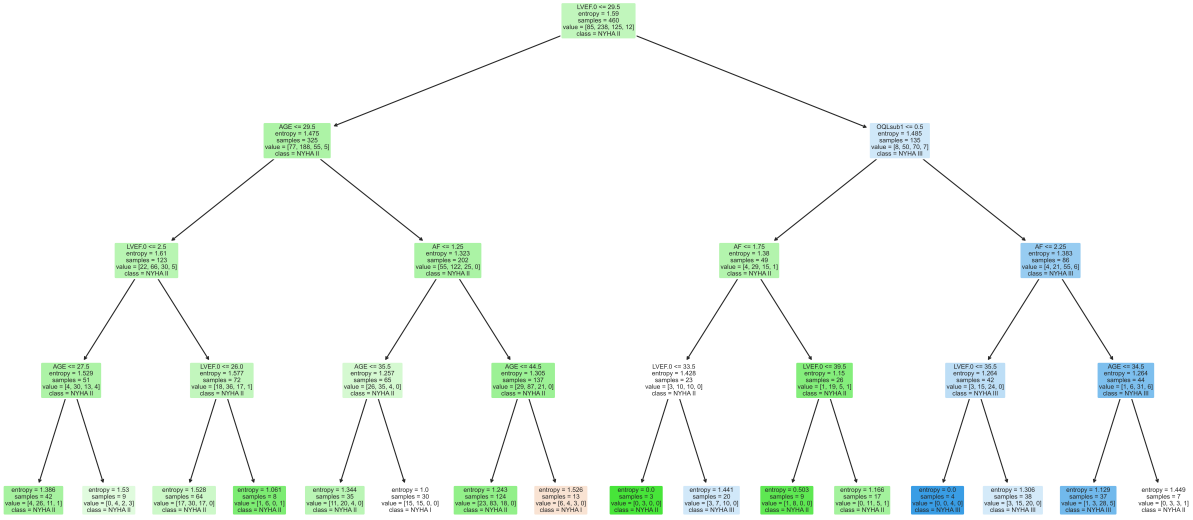
```
(class=NYHA 2) (proba: 100.0%) | based on 3 samples

=====================================================
==================== comorbidities_model ====================
if (LVEF.0 <= 0.5) AND (AGE <= 0.5) then (class=NYHA 2) (proba: 58.5%) | based on 30
6 samples
if (LVEF.0 > 0.5) AND (AGE <= 0.5) then (class=NYHA 2) (proba: 43.52%) | based on 10
8 samples
if (LVEF.0 <= 0.5) AND (AGE > 0.5) then (class=NYHA 3) (proba: 44.83%) | based on 29
samples
if (LVEF.0 > 0.5) AND (AGE > 0.5) then (class=NYHA 3) (proba: 58.82%) | based on 17
samples

=====================================================
==================== fitness_level ====================
if (OQLsub1 > 402.5) AND (LVEF.0 <= 12.858) AND (LVEF.0 > 7.858) then (class=NYHA 2)
(proba: 75.17%) | based on 149 samples
if (OQLsub1 > 402.5) AND (LVEF.0 > 12.858) AND (AGE > 4.81) then (class=NYHA 2) (pro
ba: 60.22%) | based on 93 samples
if (OQLsub1 > 402.5) AND (LVEF.0 > 12.858) AND (AGE <= 4.81) then (class=NYHA 1) (pr
oba: 64.52%) | based on 62 samples
if (OQLsub1 > 402.5) AND (LVEF.0 <= 12.858) AND (LVEF.0 <= 7.858) then (class=NYHA
3) (proba: 50.0%) | based on 52 samples
if (OQLsub1 <= 402.5) AND (OQLsub1 <= 329.5) AND (AGE <= 9.35) then (class=NYHA 3)
(proba: 79.49%) | based on 39 samples
if (OQLsub1 <= 402.5) AND (OQLsub1 > 329.5) AND (LVEF.0 > 7.767) then (class=NYHA 2)
(proba: 52.94%) | based on 34 samples
if (OQLsub1 <= 402.5) AND (OQLsub1 > 329.5) AND (LVEF.0 <= 7.767) then (class=NYHA
3) (proba: 76.47%) | based on 17 samples
if (OQLsub1 <= 402.5) AND (OQLsub1 <= 329.5) AND (AGE > 9.35) then (class=NYHA 3) (p
roba: 57.14%) | based on 14 samples

=====================================================
```



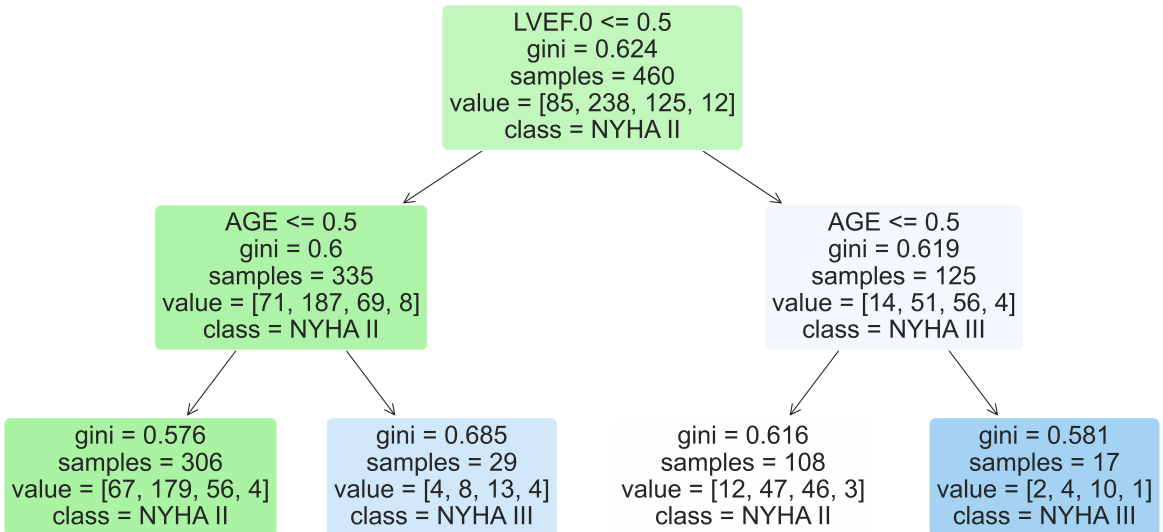
Decision tree for voting_biochemistry

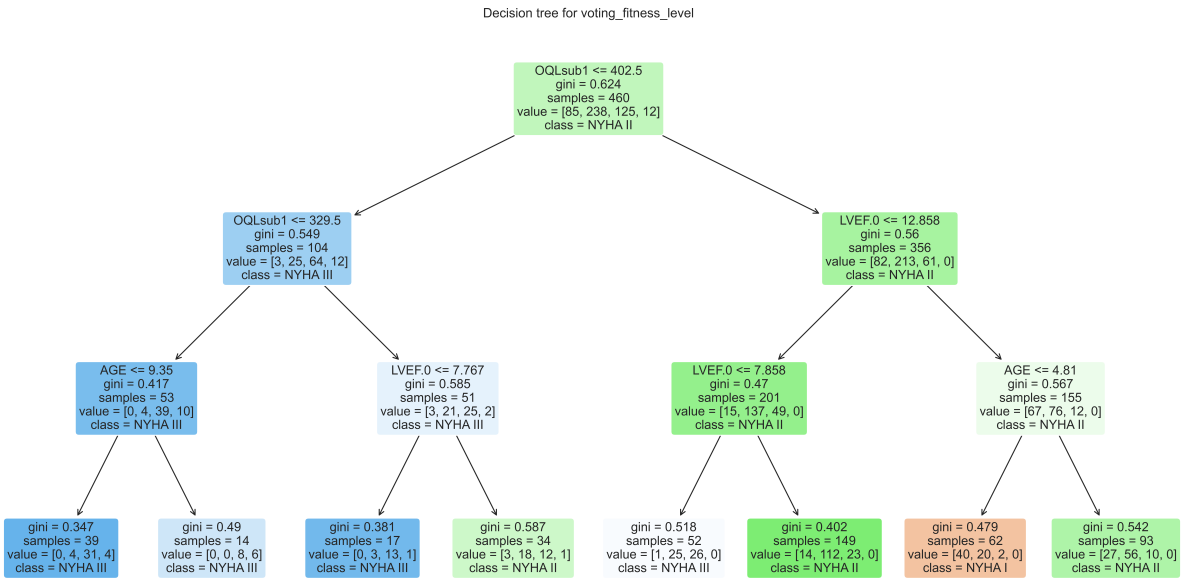


Decision tree for voting_clinical



Decision tree for voting_comorbidities_model





2.2.1. Evaluate the model

In [34]:
```
# Evaluate with cross validation
X = df.drop(columns=['NYHA'])
y = df['NYHA']

kfold = KFold(n_splits=100, shuffle=True)
voting_results = cross_val_score(voting_model, X, y, cv=kfold)
print(f"Accuracy: {voting_results.mean() * 100.0}%")
```

```
Accuracy: 61.5%
```

2.3. Create the model using only variables that are correlated with target variable

In [35]:
```
from sklearn.tree import DecisionTreeClassifier

# Remove OQLsub1, AGE, KIDNEY.DIS, EXERCISE1, AF, DEATH?, MR, DM, DIUR, DIGOX from
df = df.drop(['MR', 'DM', 'DIUR', 'DEATH?'], axis=1)

# Create a decision tree classifier
X = df.drop(columns=["NYHA"])
y = df["NYHA"]

model = DecisionTreeClassifier()

# Find the best parameters for the model
grid_search, _ = create_grid_search(model, X, y, cv=10)
tree_model = grid_search.best_estimator_
print(f"Best parameters: {grid_search.best_params_}")
print(f"Best score: {grid_search.best_score_}")
print(f"Feature importances: {tree_model.feature_importances_}")
print_feature_importances(tree_model.feature_importances_, X)
```

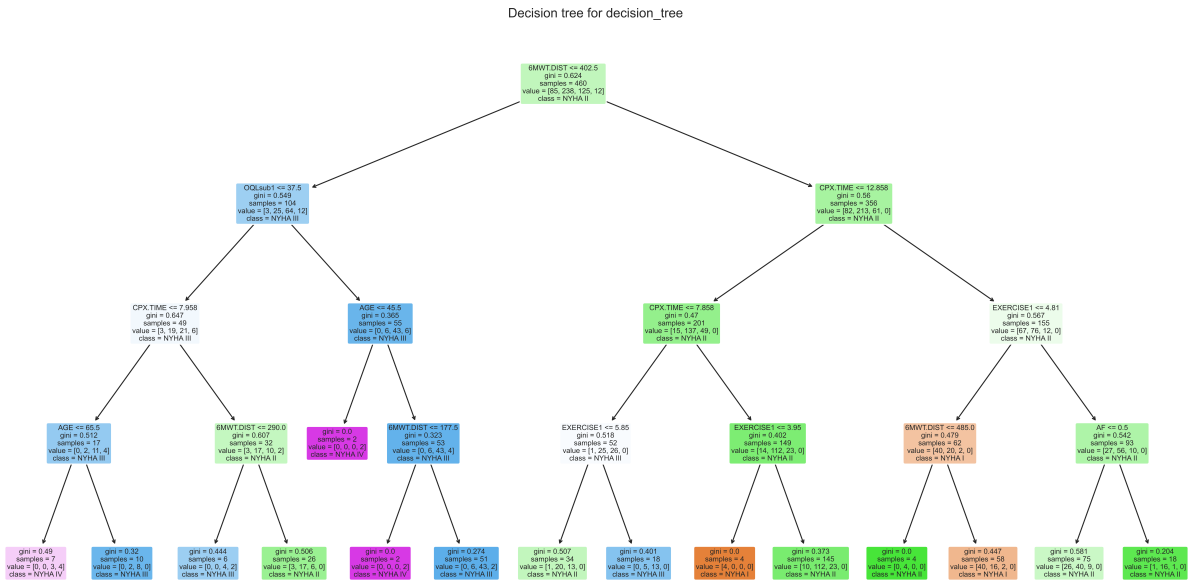
```
Cross validation score before grid search: 0.5369565217391306
Best parameters: {'class_weight': None, 'criterion': 'gini', 'max_depth': 4, 'min_sa
mples_leaf': 1, 'min_samples_split': 2}
Best score: 0.6217391304347826
Best parameters: {'class_weight': None, 'criterion': 'gini', 'max_depth': 4, 'min_sa
mples_leaf': 1, 'min_samples_split': 2}
Best score: 0.6217391304347826
Feature importances: [0.05418928 0.         0.05084892 0.0316728  0.         0.
 0.41602444 0.2841095  0.16315507]
      feature  importance
6   6MWT.DIST    0.416024
7    CPX.TIME    0.284109
8   EXERCISE1    0.163155
0     OQLsub1    0.054189
2         AGE    0.050849
3          AF    0.031673
1      LVEF.0    0.000000
4  KIDNEY.DIS    0.000000
5       DIGOX    0.000000
```

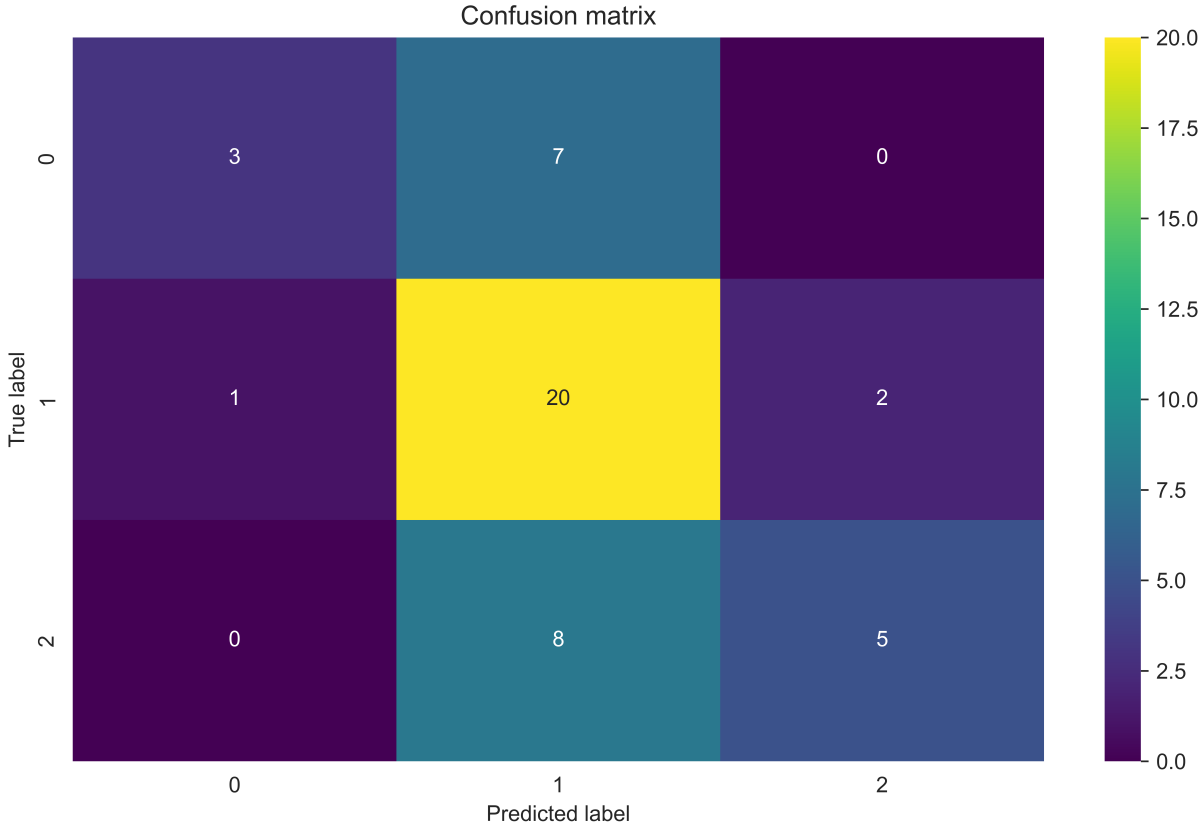
2.3.1. Evaluate the model

In [36]:

```python
from sklearn.model_selection import train_test_split

# Evaluate with cross validation
X = df.drop(columns=['NYHA'])
y = df['NYHA']

kfold = KFold(n_splits=100, shuffle=True)
tree_results = cross_val_score(tree_model, X, y, cv=kfold)
print(f"Accuracy: {tree_results.mean() * 100.0}%")

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
plot_decision_tree(tree_model, X, filename="decision_tree")
plot_confusion_matrix(tree_model, X_test, y_test, filename="decision_tree")
```

```
Accuracy: 62.9%
```





Summary of the model

The best model is the Decission Tree learner with the following parameters:

- class_weight: None
- criterion: gini

- max_depth: 4
- min_samples_leaf: 1
- min_samples_split: 2

The accuracy of the model is 62.17%.

Decision rules

In [37]:
```
from sklearn.tree import export_text

r = export_text(tree_model, feature_names=list(X.columns))
print(r)
```

```
|--- 6MWT.DIST <= 402.50
|   |--- OQLsub1 <= 37.50
|   |   |--- CPX.TIME <= 7.96
|   |   |   |--- AGE <= 65.50
|   |   |   |   |--- class: 4.0
|   |   |   |--- AGE >  65.50
|   |   |   |   |--- class: 3.0
|   |   |--- CPX.TIME >  7.96
|   |   |   |--- 6MWT.DIST <= 290.00
|   |   |   |   |--- class: 3.0
|   |   |   |--- 6MWT.DIST >  290.00
|   |   |   |   |--- class: 2.0
|   |--- OQLsub1 >  37.50
|   |   |--- AGE <= 45.50
|   |   |   |--- class: 4.0
|   |   |--- AGE >  45.50
|   |   |   |--- 6MWT.DIST <= 177.50
|   |   |   |   |--- class: 4.0
|   |   |   |--- 6MWT.DIST >  177.50
|   |   |   |   |--- class: 3.0
|--- 6MWT.DIST >  402.50
|   |--- CPX.TIME <= 12.86
|   |   |--- CPX.TIME <= 7.86
|   |   |   |--- EXERCISE1 <= 5.85
|   |   |   |   |--- class: 2.0
|   |   |   |--- EXERCISE1 >  5.85
|   |   |   |   |--- class: 3.0
|   |   |--- CPX.TIME >  7.86
|   |   |   |--- EXERCISE1 <= 3.95
|   |   |   |   |--- class: 1.0
|   |   |   |--- EXERCISE1 >  3.95
|   |   |   |   |--- class: 2.0
|   |--- CPX.TIME >  12.86
|   |   |--- EXERCISE1 <= 4.81
|   |   |   |--- 6MWT.DIST <= 485.00
|   |   |   |   |--- class: 2.0
|   |   |   |--- 6MWT.DIST >  485.00
|   |   |   |   |--- class: 1.0
|   |   |--- EXERCISE1 >  4.81
|   |   |   |--- AF <= 0.50
|   |   |   |   |--- class: 2.0
|   |   |   |--- AF >  0.50
|   |   |   |   |--- class: 2.0
```

In [39]:

```
rules = get_rules(tree_model, list(X.columns), ['NYHA 1', 'NYHA 2', 'NYHA 3', 'NYHA
rules_txt = ""
for r in rules:
    rules_txt += r + "\n"
save_rules_to_file("decision_tree_", rules_txt)
print(rules_txt)
```

```
if (6MWT.DIST > 402.5) AND (CPX.TIME <= 12.858) AND (CPX.TIME > 7.858) AND (EXERCISE
1 > 3.95) then (class=NYHA 2) (proba: 77.24%) | based on 145 samples
if (6MWT.DIST > 402.5) AND (CPX.TIME > 12.858) AND (EXERCISE1 > 4.81) AND (AF <= 0.
5) then (class=NYHA 2) (proba: 53.33%) | based on 75 samples
if (6MWT.DIST > 402.5) AND (CPX.TIME > 12.858) AND (EXERCISE1 <= 4.81) AND (6MWT.DIS
T > 485.0) then (class=NYHA 1) (proba: 68.97%) | based on 58 samples
if (6MWT.DIST <= 402.5) AND (OQLsub1 > 37.5) AND (AGE > 45.5) AND (6MWT.DIST > 177.
5) then (class=NYHA 3) (proba: 84.31%) | based on 51 samples
if (6MWT.DIST > 402.5) AND (CPX.TIME <= 12.858) AND (CPX.TIME <= 7.858) AND (EXERCIS
E1 <= 5.85) then (class=NYHA 2) (proba: 58.82%) | based on 34 samples
if (6MWT.DIST <= 402.5) AND (OQLsub1 <= 37.5) AND (CPX.TIME > 7.958) AND (6MWT.DIST
> 290.0) then (class=NYHA 2) (proba: 65.38%) | based on 26 samples
if (6MWT.DIST > 402.5) AND (CPX.TIME > 12.858) AND (EXERCISE1 > 4.81) AND (AF > 0.5)
then (class=NYHA 2) (proba: 88.89%) | based on 18 samples
if (6MWT.DIST > 402.5) AND (CPX.TIME <= 12.858) AND (CPX.TIME <= 7.858) AND (EXERCIS
E1 > 5.85) then (class=NYHA 3) (proba: 72.22%) | based on 18 samples
if (6MWT.DIST <= 402.5) AND (OQLsub1 <= 37.5) AND (CPX.TIME <= 7.958) AND (AGE > 65.
5) then (class=NYHA 3) (proba: 80.0%) | based on 10 samples
if (6MWT.DIST <= 402.5) AND (OQLsub1 <= 37.5) AND (CPX.TIME <= 7.958) AND (AGE <= 6
5.5) then (class=NYHA 4) (proba: 57.14%) | based on 7 samples
if (6MWT.DIST <= 402.5) AND (OQLsub1 <= 37.5) AND (CPX.TIME > 7.958) AND (6MWT.DIST
<= 290.0) then (class=NYHA 3) (proba: 66.67%) | based on 6 samples
if (6MWT.DIST > 402.5) AND (CPX.TIME > 12.858) AND (EXERCISE1 <= 4.81) AND (6MWT.DIS
T <= 485.0) then (class=NYHA 2) (proba: 100.0%) | based on 4 samples
if (6MWT.DIST > 402.5) AND (CPX.TIME <= 12.858) AND (CPX.TIME > 7.858) AND (EXERCISE
1 <= 3.95) then (class=NYHA 1) (proba: 100.0%) | based on 4 samples
if (6MWT.DIST <= 402.5) AND (OQLsub1 > 37.5) AND (AGE > 45.5) AND (6MWT.DIST <= 177.
5) then (class=NYHA 4) (proba: 100.0%) | based on 2 samples
if (6MWT.DIST <= 402.5) AND (OQLsub1 > 37.5) AND (AGE <= 45.5) then (class=NYHA 4)
(proba: 100.0%) | based on 2 samples
```

In [ ]: