Import Required Libraries

In [114]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data Acquistion

In [115]:
```
df = pd.read_excel('Dataset1(7 sheets).xlsx', sheet_name=None)
```

Exploratory Data Analysis

Data Description

In [116]:
```
for key in df.keys():
    display(df[key].describe())
```

	As:1	As:2	As:3	As:4	As:5	As:6	As	Qz:1	Qz:2	Qz:3	
count	71.000000	70.000000	70.000000	70.000000	69.000000	63.000000	70.000000	70.000000	69.000000	68.000000	65
mean	41.802817	78.700000	107.142857	56.507143	77.579710	57.388889	11.432857	3.800000	2.065217	3.816176	1
std	10.514439	20.022524	27.718584	19.490334	25.939072	20.725052	2.062006	2.251248	2.284721	3.060222	2
min	3.000000	3.000000	3.000000	0.000000	0.000000	3.000000	5.290000	0.500000	0.000000	0.000000	0
25%	38.250000	72.000000	91.500000	45.750000	65.000000	41.000000	10.420000	2.125000	0.000000	1.000000	0
50%	44.000000	82.000000	119.000000	63.500000	80.000000	65.000000	11.755000	3.250000	2.000000	3.000000	0
75%	48.000000	92.750000	125.000000	70.000000	95.000000	75.000000	12.935000	5.000000	3.000000	6.000000	1
max	60.000000	100.000000	140.000000	80.000000	120.000000	80.000000	15.000000	10.000000	10.000000	10.000000	10

	As:1	As:2	As:3	As:4	As:5	As:6	As	Qz:1	Qz:2	Qz:3
count	41.000000	41.000000	40.00000	41.000000	38.000000	40.000000	40.000000	40.000000	38.000000	35.000000	3

Data Info

In [117]:
```
for key in df.keys():
    display(df[key].info())
```

```
 2   As:2        35 non-null     float64
 3   As:3        35 non-null     float64
 4   As:4        34 non-null     float64
 5   As:5        35 non-null     float64
 6   As:6        34 non-null     float64
 7   As:7        30 non-null     float64
 8   As          33 non-null     float64
 9   Qz:1        33 non-null     float64
 10  Qz:2        34 non-null     float64
 11  Qz:3        34 non-null     float64
 12  Qz:4        35 non-null     float64
 13  Qz:5        35 non-null     float64
 14  Qz          34 non-null     float64
 15  S-I         33 non-null     float64
 16  S-II        34 non-null     float64
 17  Grade       33 non-null     object
dtypes: float64(16), object(2)
memory usage: 5.2+ KB

None
```

Check null values available

In [118]:
```
for key in df.keys():
    display(df[key].isna().sum())
```

```
Unnamed: 0     0
As:1           1
As:2           2
As:3           2
As:4           2
As:5           3
As:6           9
As             2
Qz:1           2
Qz:2           3
Qz:3           4
Qz:4           7
Qz:5          11
Qz:6           7
Qz:7          12
Qz             2
S-I            1
S-II           1
Grade          3
```

Preprocessing Data

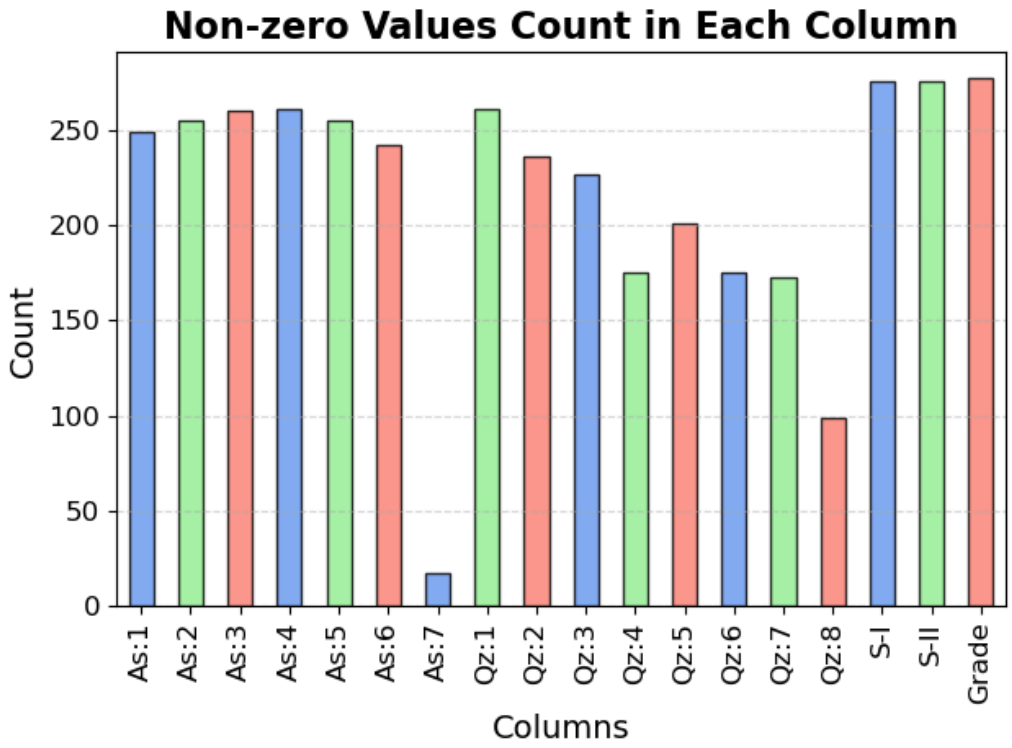
In [119]:
```
df = pd.read_excel('Dataset1(7 sheets).xlsx', sheet_name=None)
for key in df.keys():
    total = np.array(df[key].iloc[1].values[1:], dtype=float)
    weight = np.array(df[key].iloc[0].values[1:], dtype=float)
    nanInTotal = np.isnan(total)
    assignmentTotal = []
    quizTotal = []
    assignmentWeight = []
    quizWeight = []
    trueCome = 0
    index = 0
    for isNan in nanInTotal:
        if isNan == False and trueCome == 0:
            assignmentTotal.append(total[index])
            assignmentWeight.append(weight[index])
            index = index + 1
        elif isNan == False and trueCome == 1:
            quizTotal.append(total[index])
            quizWeight.append(weight[index])
            index = index + 1
        elif isNan == True:
            trueCome = trueCome + 1
            index = index + 1
    # print(assignmentTotal, quizTotal, assignmentWeight, quizWeight)
    assignmentTotal.extend(quizTotal)
    assignmentWeight.extend(quizWeight)
    totalByWeights = np.array(assignmentTotal)/np.array(assignmentWeight)
    df[key] = df[key].iloc[3:].drop(['Unnamed: 0', 'As', 'Qz'], axis=1).fillna(0) # Data Cleaning & Imp
    # Making the data unit free by converting the assignments and quizzes numbers to absolutes
    for i in range(len(totalByWeights)):
        df[key].iloc[:, i] = df[key].iloc[:, i] / totalByWeights[i]
df_final = pd.concat(df.values(), ignore_index=True).fillna(0).reindex(columns=['As:1','As:2', 'As:3',
                                                                                'Qz:1', 'Qz:2',
                                                                                'Qz:8', 'S-I','
df_final
```

Out[119]:

	As:1	As:2	As:3	As:4	As:5	As:6	As:7	Qz:1	Qz:2	Qz:3	Qz:4	Qz:5	Qz:6	Qz:7	Qz:8	S
0	1.975000	2.700000	2.571429	3.00000	2.125	2.812500	0.0	1.5	0.900000	0.9	0.0	0.2	1.0	0.000000	0.0	9.7
1	2.000000	1.860000	1.992857	1.21875	1.875	2.850000	0.0	0.3	0.000000	0.1	0.0	0.2	0.4	0.000000	0.0	3.3
2	2.125000	1.890000	2.571429	2.32500	1.625	1.875000	0.0	0.0	0.000000	0.2	0.0	0.2	0.0	2.000000	0.0	6.5
3	1.025000	1.260000	1.285714	2.62500	1.750	0.375000	0.0	0.2	0.400000	0.0	0.0	0.0	0.0	2.000000	0.0	5.0
4	2.150000	1.950000	2.678571	0.37500	2.750	0.937500	0.0	0.6	0.200000	0.0	0.0	0.0	0.0	0.000000	0.0	4.5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
272	2.330769	2.121429	0.000000	2.85000	2.200	1.533333	0.0	1.2	1.133333	0.1	0.0	0.2	0.8	0.333333	2.0	6.5
273	1.430769	0.000000	1.900000	1.80000	2.175	2.233333	0.0	0.0	1.333333	0.0	0.2	0.6	0.0	1.000000	2.0	3.3
274	1.938462	2.185714	1.483333	1.68000	2.000	1.600000	0.0	0.8	0.933333	0.2	0.0	1.8	1.2	1.166667	2.0	5.2
275	2.884615	2.571429	2.800000	2.43000	2.750	2.866667	0.0	1.8	0.800000	0.0	0.0	0.6	1.0	0.333333	2.0	8.0
276	2.238462	0.000000	1.666667	2.46000	1.800	0.000000	0.0	0.0	0.600000	0.0	0.0	0.6	0.0	0.000000	2.0	4.4

277 rows × 18 columns

In [120]:
```
df_final.astype(bool).sum(axis=0).plot(kind='bar', color=['cornflowerblue', 'lightgreen', 'salmon'], e
plt.title('Non-zero Values Count in Each Column', fontsize=16, fontweight='bold')
plt.xlabel('Columns', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.xticks(rotation=90, fontsize=12)
plt.yticks(fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```



In [121]:
```
df_final.describe()
```

Out[121]:

:4	As:5	As:6	As:7	Qz:1	Qz:2	Qz:3	Qz:4	Qz:5	Qz:6	Qz:7	
)0	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277
0	1.875503	2.003044	0.151625	0.911673	0.678670	0.556655	0.710830	0.948857	0.707220	0.808424	0
)9	0.792778	0.960023	0.613799	0.521244	0.484233	0.549414	0.768206	0.821027	0.713488	0.815358	0
)0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
)0	1.500000	1.575000	0.000000	0.600000	0.300000	0.100000	0.000000	0.000000	0.000000	0.000000	0
)0	2.000000	2.333333	0.000000	0.800000	0.666667	0.400000	0.400000	0.666667	0.600000	0.500000	0
57	2.425000	2.766667	0.000000	1.300000	1.000000	0.866667	1.400000	2.000000	1.400000	1.666667	2
)0	3.000000	3.000000	2.933333	2.000000	2.000000	2.000000	2.000000	2.000000	2.000000	2.000000	2

Phase II

Data Preprocessing

In [122]:
```
X=df_final.iloc[:,[0,1,2,3,7,8,9,10,15]].values
```

In [123]:
```
X
```

Out[123]:
```
array([[1.975     , 2.7       , 2.57142857, ..., 0.9       , 0.        ,
         9.75      ],
       [2.        , 1.86      , 1.99285714, ..., 0.1       , 0.        ,
         3.37      ],
       [2.125     , 1.89      , 2.57142857, ..., 0.2       , 0.        ,
         6.56      ],
       ...,
       [1.93846154, 2.18571429, 1.48333333, ..., 0.2       , 0.        ,
         5.25      ],
       [2.88461538, 2.57142857, 2.8       , ..., 0.        , 0.        ,
         8.06      ],
       [2.23846154, 0.        , 1.66666667, ..., 0.        , 0.        ,
         4.4       ]])
```

In [124]:
```
y=df_final.iloc[::,-1].values
y
```

Out[124]:
```
array(['Pass', 'Fail', 'Fail', 'Fail', 'Fail', 'Pass', 'Fail', 'Pass',
       'Pass', 'Pass', 'Fail', 'Pass', 'Pass', 'Fail', 'Fail', 'Pass',
       'Fail', 'Pass', 'Fail', 'Pass', 'Fail', 'Pass', 'Fail', 'Pass',
       'Fail', 'Fail', 'Fail', 'Fail', 'Pass', 'Fail', 'Fail', 'Fail',
       'Pass', 'Fail', 'Pass', 'Pass', 'Pass', 'Fail', 'Fail', 'Fail',
       'Fail', 'Fail', 'Fail', 'Fail', 'Fail', 'Fail', 'Pass', 'Pass',
       'Fail', 'Pass', 'Pass', 'Pass', 'Pass', 'Fail', 'Pass', 'Pass',
       'Pass', 'Fail', 'Pass', 'Pass', 'Pass', 'Fail', 'Fail', 'Fail',
       'Fail', 'Fail', 'Fail', 'Fail', 'Pass', 'Fail', 'Fail', 'Pass',
       'Pass', 'Pass', 'Fail', 'Pass', 'Pass', 'Pass', 'Fail', 'Pass',
       'Pass', 'Fail', 'Fail', 'Pass', 'Fail', 'Fail', 'Fail', 'Pass',
       'Fail', 'Pass', 'Fail', 'Fail', 'Pass', 'Pass', 'Fail', 'Pass',
       'Fail', 'Fail', 'Fail', 'Pass', 'Fail', 'Pass', 'Pass', 'Pass',
       'Fail', 'Fail', 'Pass', 'Pass', 'Fail', 'Pass', 'Pass', 'Pass',
       'Fail', 'Fail', 'Pass', 'Fail', 'Pass', 'Fail', 'Fail', 'Fail',
       'Pass', 'Fail', 'Pass', 'Fail', 'Fail', 'Pass', 'Pass', 'Fail',
       'Fail', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass',
       'Pass', 'Fail', 'Fail', 'Pass', 'Pass', 'Pass', 'Fail', 'Pass',
       'Fail', 'Pass', 'Fail', 'Fail', 'Pass', 'Fail', 'Pass', 'Pass',
       'Fail', 'Pass', 'Pass', 'Pass', 'Pass', 'Fail', 'Pass', 'Pass',
       'Fail', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Fail',
       'Pass', 'Fail', 'Pass', 'Pass', 'Pass', 'Fail', 'Fail', 'Fail',
       'Fail', 'Pass', 'Fail', 'Fail', 'Pass', 'Pass', 'Pass', 'Pass',
       'Pass', 'Fail', 'Pass', 'Fail', 'Fail', 'Fail', 'Fail', 'Pass',
       'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Fail', 'Pass', 'Pass',
       'Fail', 'Fail', 'Pass', 'Pass', 'Fail', 'Fail', 'Fail', 'Fail',
       'Pass', 'Pass', 'Fail', 'Pass', 'Pass', 'Fail', 'Pass', 'Pass',
       'Pass', 'Fail', 'Fail', 'Fail', 'Fail', 'Pass', 'Pass', 'Fail',
       'Fail', 'Fail', 'Fail', 'Fail', 'Pass', 'Fail', 'Fail', 'Pass',
       'Fail', 'Fail', 'Fail', 'Fail', 'Pass', 'Pass', 'Pass', 'Fail',
       'Fail', 'Pass', 'Pass', 'Pass', 'Pass', 'Fail', 'Fail', 'Pass',
       'Fail', 'Pass', 'Fail', 'Pass', 'Fail', 'Pass', 'Pass', 'Pass',
       'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass', 'Pass',
       'Pass', 'Pass', 'Fail', 'Pass', 'Fail', 'Pass', 'Fail', 'Pass',
       'Fail', 'Fail', 'Fail', 'Pass', 'Fail'], dtype=object)
```

Encoding the Dependent Variable

In [125]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
```

In [126]:
```
y
```

Out[126]:
```
array([1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,
       0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0,
       0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1,
       0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1,
       1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1,
       1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0,
       0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0,
       1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0,
       0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1,
       1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0])
```

Splitting the dataset into the Training set and Test set

In [127]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,stratify=y, random_state = 1
```

In [128]:
```
print(X_train)
X_train.shape
```

```
[[ 1.025       1.26        1.28571429 ...  0.          0.
   5.06      ]
 [ 2.425       2.1         1.62857143 ...  0.4         1.
   6.93      ]
 [ 2.37692308  0.          2.66666667 ...  2.          2.
  11.06      ]
 ...
 [ 1.5         2.82        2.22857143 ...  0.3         1.2
   2.15      ]
 [ 2.55        2.31        2.67857143 ...  1.3         1.6
   8.62      ]
 [ 2.375       2.7         2.25       ...  0.          0.
   3.93      ]]
```

Out[128]:
```
(221, 9)
```

In [129]:
```
print(X_test)
X_test.shape
```

```
  0.52        2.          6.75      ]
 [ 2.60769231  2.65714286  0.          2.82        1.4         0.33333333
   0.2         0.2         3.56      ]
 [ 1.9         2.25        2.12142857  2.52        1.2         0.6
   0.1         0.          4.87      ]
 [ 1.65        2.4         2.67857143  1.9875      0.7         0.2
   1.6         0.          2.25      ]
 [ 1.2         2.55        1.28571429  2.1         1.6         0.3
   0.2         1.8         5.06      ]
 [ 2.52        1.64        2.54285714  1.26666667  1.          0.45
   0.4         0.          6.18      ]
 [ 2.45        2.55        2.46428571  3.          1.4         0.8
   0.05        1.8         5.62      ]
 [ 2.25        1.65        2.57142857  2.4375      0.2         0.5
   0.4         0.          1.5       ]
 [ 2.05        0.          0.          0.          1.4         0.
   0.          0.          3.        ]
 [ 1.02        2.32        2.8         2.03333333  0.86666667  1.1
   1.25        1.          8.43      ]
 [ 2.3         1.83        2.89285714  2.625       0.2         0.
```

In [130]:
```
print(y_train)
y_train.shape
```

```
[0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 1 1 0 1
 1 1 1 1 1 0 0 1 0 1 0 0 1 0 0 0 0 1 1 1 0 0 1 1 0 1 0 1 1 1 1 0 1 0 0 0 1
 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 1 1 1 0 1 1 0 1 0 0 0 0
 1 1 1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 0 1 0 0 0 0 0 1
 0 1 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 1 0 1 0 0 1 1 0
 1 1 1 1 0 1 1 1 0 0 1 0 1 1 1 1 0 1 0 1 0 1 1 1 0 1 0 0 0 0 0 1 1 0 1 1]
```

Out[130]: (221,)

In [131]:
```
print(y_test)
y_test.shape
```

```
[0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 0 1 1 1 0 0 1 0 1 0 0 1 1 0 0 1 0 1 0 1 1 1
 1 1 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1 1 1]
```

Out[131]: (56,)

Feature Scaling

In [132]:
```
temp=pd.DataFrame(X_train)
# print(temp.describe())
temp
```

Out[132]:

	0	1	2	3	4	5	6	7	8
0	1.025000	1.26	1.285714	2.625000	0.2	0.4	0.000000	0.0	5.06
1	2.425000	2.10	1.628571	2.561538	0.5	0.7	0.400000	1.0	6.93
2	2.376923	0.00	2.666667	3.000000	1.6	1.2	2.000000	2.0	11.06
3	2.700000	2.49	2.957143	2.925000	0.8	0.4	1.500000	0.0	10.12
4	1.425000	2.25	2.892857	2.812500	0.6	0.0	0.500000	0.2	3.75
...	...	...	...	...	...	...	...	...	...
216	1.975000	2.70	2.571429	3.000000	1.5	0.9	0.900000	0.0	9.75
217	2.670000	3.00	2.717822	2.875000	1.1	1.4	0.933333	1.6	8.25
218	1.500000	2.82	2.228571	2.940000	1.4	1.6	0.300000	1.2	2.15
219	2.550000	2.31	2.678571	2.930769	0.6	1.8	1.300000	1.6	8.62
220	2.375000	2.70	2.250000	0.960000	1.4	0.4	0.000000	0.0	3.93

221 rows × 9 columns

In [133]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [134]:
```
temp=pd.DataFrame(X_train)
# print(temp.describe())
temp
```

Out[134]:

	0	1	2	3	4	5	6	7	8
0	-1.197229	-1.080250	-1.121746	0.653061	-1.357380	-0.632193	-1.023383	-0.972777	-0.301092
1	0.580047	-0.027812	-0.651867	0.574339	-0.773001	-0.015276	-0.297556	0.345063	0.485509
2	0.519014	-2.658908	0.770820	1.118238	1.369720	1.012920	2.605751	1.662904	2.222762
3	0.929155	0.460820	1.168911	1.025203	-0.188623	-0.632193	1.698468	-0.972777	1.827358
4	-0.689436	0.160124	1.080809	0.885650	-0.578209	-1.454750	-0.116099	-0.709209	-0.852134
...	...	...	...	...	...	...	...	...	...
216	0.008780	0.723930	0.640298	1.118238	1.174927	0.396003	0.609727	-0.972777	1.671720
217	0.891071	1.099801	0.840927	0.963179	0.395756	1.424199	0.670213	1.135768	1.040757
218	-0.594225	0.874278	0.170420	1.043810	0.980134	1.835477	-0.479013	0.608631	-1.525162
219	0.738733	0.235298	0.787135	1.032359	-0.578209	2.246755	1.335554	1.135768	1.196395
220	0.516573	0.723930	0.199787	-1.412325	0.980134	-0.632193	-1.023383	-0.972777	-0.776418

221 rows × 9 columns

In [135]:
```
# temp=pd.DataFrame(X_test)
# # print(temp.describe())
# temp
```

Result Prediction before Mid 2

Grid Search

In [136]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

def grid_search_knn(x_train, y_train, param_grid, cv=10):
    """
    Perform grid search for hyperparameter tuning of K-Nearest Neighbors (KNN) classifier.

    Args:
        x_train (numpy.ndarray): Training features.
        y_train (numpy.ndarray): Training labels.
        param_grid (dict): Dictionary with parameters names (string) as keys and lists of
                           parameter settings to try as values.
        cv (int): Number of folds for cross-validation.

    Returns:
        sklearn.model_selection.GridSearchCV: GridSearchCV object with the best parameters.
    """
    knn_model = KNeighborsClassifier()
    grid_search = GridSearchCV(knn_model, param_grid, cv=cv)
    grid_search.fit(x_train, y_train)
    return grid_search.best_params_

def grid_search_decision_tree(x_train, y_train, param_grid, cv=10):
    """
    Perform grid search for hyperparameter tuning of Decision Tree classifier.

    Args:
        x_train (numpy.ndarray): Training features.
        y_train (numpy.ndarray): Training labels.
        param_grid (dict): Dictionary with parameters names (string) as keys and lists of
                           parameter settings to try as values.
        cv (int): Number of folds for cross-validation.

    Returns:
        sklearn.model_selection.GridSearchCV: GridSearchCV object with the best parameters.
    """
    tree_model = DecisionTreeClassifier()
    grid_search = GridSearchCV(tree_model, param_grid, cv=cv)
    grid_search.fit(x_train, y_train)
    return grid_search.best_params_
```

In [137]:
```
# Example parameter grid for KNN
knn_param_grid = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']
}

# Example parameter grid for Decision Tree
tree_param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 3, 5, 10]
}

# Perform grid search for KNN
knn_grid_search = grid_search_knn(X_train, y_train, knn_param_grid)
print("knn_grid_search",knn_grid_search)
# Perform grid search for Decision Tree
tree_grid_search = grid_search_decision_tree(X_train, y_train, tree_param_grid)
print("tree_grid_search",tree_grid_search)
```

```
knn_grid_search {'algorithm': 'auto', 'n_neighbors': 7, 'weights': 'uniform'}
tree_grid_search {'criterion': 'gini', 'max_depth': None}
```

In [138]:
```
# Define the hyperparameters for KNN model
knn_params = {'algorithm': 'auto', 'n_neighbors': 7, 'weights': 'uniform'}

# Define the hyperparameters for Decision Tree model
tree_params = {'criterion': 'gini', 'max_depth': None}

# Create KNN model with best hyperparameters
knn_model = KNeighborsClassifier(**knn_params)

# Create Decision Tree model with best hyperparameters
tree_model = DecisionTreeClassifier(**tree_params)

# Assuming X_train and y_train are your training data
# Train the KNN model
knn_model.fit(X_train, y_train)

# Train the Decision Tree model
tree_model.fit(X_train, y_train)
```

Out[138]:

DecisionTreeClassifier (https://scikit-learn.org/1.4/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

```
DecisionTreeClassifier()
```

In [139]:
```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score

def evaluate_model(model, X_test, y_test):
    """
    Evaluate the performance of the specified model on the test data.

    Args:
        model: Trained classifier model.
        X_test (array-like): Test features.
        y_test (array-like): True labels for the test data.

    Returns:
        dict: Dictionary containing performance metrics.
    """
    # Make predictions
    y_pred = model.predict(X_test)

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)

    # Accuracy
    accuracy = accuracy_score(y_test, y_pred)

    # Precision
    precision = precision_score(y_test, y_pred)

    # Recall (Sensitivity)
    recall = recall_score(y_test, y_pred)

    # F1-score
    f1 = f1_score(y_test, y_pred)

    # Specificity
    tn, fp, fn, tp = cm.ravel()
    specificity = tn / (tn + fp)

    # Create a dictionary to store performance metrics
    performance_metrics = {
        'confusion_matrix': cm,
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1_score': f1,
        'specificity': specificity
    }

    return performance_metrics

# Evaluate KNN model
knn_performance = evaluate_model(knn_model, X_test, y_test)

# Evaluate Decision Tree model
tree_performance = evaluate_model(tree_model, X_test, y_test)

# Print performance metrics for KNN model
print("Performance metrics for KNN model:")
for metric, value in knn_performance.items():
    print(f"{metric}: {value}")

# Print performance metrics for Decision Tree model
print("\nPerformance metrics for Decision Tree model:")
for metric, value in tree_performance.items():
    print(f"{metric}: {value}")
```

```
Performance metrics for KNN model:
confusion_matrix: [[23  3]
 [ 8 22]]
accuracy: 0.8035714285714286
precision: 0.88
recall: 0.7333333333333333
f1_score: 0.8
specificity: 0.8846153846153846

Performance metrics for Decision Tree model:
confusion_matrix: [[19  7]
 [ 9 21]]
accuracy: 0.7142857142857143
precision: 0.75
recall: 0.7
f1_score: 0.7241379310344828
specificity: 0.7307692307692307
```

Result Prediction before Final Exams

In [140]:
```
df_final
```

Out[140]:

	As:1	As:2	As:3	As:4	As:5	As:6	As:7	Qz:1	Qz:2	Qz:3	Qz:4	Qz:5	Qz:6	Qz:7	Qz:8	S
0	1.975000	2.700000	2.571429	3.00000	2.125	2.812500	0.0	1.5	0.900000	0.9	0.0	0.2	1.0	0.000000	0.0	9.7
1	2.000000	1.860000	1.992857	1.21875	1.875	2.850000	0.0	0.3	0.000000	0.1	0.0	0.2	0.4	0.000000	0.0	3.3
2	2.125000	1.890000	2.571429	2.32500	1.625	1.875000	0.0	0.0	0.000000	0.2	0.0	0.2	0.0	2.000000	0.0	6.5
3	1.025000	1.260000	1.285714	2.62500	1.750	0.375000	0.0	0.2	0.400000	0.0	0.0	0.0	0.0	2.000000	0.0	5.0
4	2.150000	1.950000	2.678571	0.37500	2.750	0.937500	0.0	0.6	0.200000	0.0	0.0	0.0	0.0	0.000000	0.0	4.5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
272	2.330769	2.121429	0.000000	2.85000	2.200	1.533333	0.0	1.2	1.133333	0.1	0.0	0.2	0.8	0.333333	2.0	6.5
273	1.430769	0.000000	1.900000	1.80000	2.175	2.233333	0.0	0.0	1.333333	0.0	0.2	0.6	0.0	1.000000	2.0	3.3
274	1.938462	2.185714	1.483333	1.68000	2.000	1.600000	0.0	0.8	0.933333	0.2	0.0	1.8	1.2	1.166667	2.0	5.2
275	2.884615	2.571429	2.800000	2.43000	2.750	2.866667	0.0	1.8	0.800000	0.0	0.0	0.6	1.0	0.333333	2.0	8.0
276	2.238462	0.000000	1.666667	2.46000	1.800	0.000000	0.0	0.0	0.600000	0.0	0.0	0.6	0.0	0.000000	2.0	4.4

277 rows × 18 columns

In [163]:
```
temp = df_final.iloc[:, 0:7].values

# Sort each row independently
temp = np.sort(temp, axis=1)

print(temp.shape)
```

```
(277, 7)
```

In [186]:
```
tempq=df_final.iloc[:,7:15].values
tempq=np.sort(tempq, axis=1)
tempq.shape
```

Out[186]:
```
(277, 8)
```

In [192]:
```
X1 = np.concatenate((temp[:, -5:] , tempq[:, -5:]),axis=1)  # Assuming you meant to use temp instead o
X = np.concatenate((X1, df_final.iloc[:, [15, 16]].values), axis=1)
X=pd.DataFrame(X)
X
```

Out[192]:

	0	1	2	3	4	5	6	7	8	9	10	11
0	2.125000	2.571429	2.700000	2.812500	3.000000	0.200000	0.900000	0.900000	1.000000	1.5	9.75	8.62
1	1.860000	1.875000	1.992857	2.000000	2.850000	0.000000	0.100000	0.200000	0.300000	0.4	3.37	3.93
2	1.875000	1.890000	2.125000	2.325000	2.571429	0.000000	0.000000	0.200000	0.200000	2.0	6.56	0.93
3	1.025000	1.260000	1.285714	1.750000	2.625000	0.000000	0.000000	0.200000	0.400000	2.0	5.06	2.81
4	0.937500	1.950000	2.150000	2.678571	2.750000	0.000000	0.000000	0.000000	0.200000	0.6	4.50	2.25
...	...	...	...	...	...	...	...	...	...	...	...	...
272	1.533333	2.121429	2.200000	2.330769	2.850000	0.333333	0.800000	1.133333	1.200000	2.0	6.56	2.90
273	1.430769	1.800000	1.900000	2.175000	2.233333	0.200000	0.600000	1.000000	1.333333	2.0	3.37	1.59
274	1.600000	1.680000	1.938462	2.000000	2.185714	0.933333	1.166667	1.200000	1.800000	2.0	5.25	1.50
275	2.571429	2.750000	2.800000	2.866667	2.884615	0.600000	0.800000	1.000000	1.800000	2.0	8.06	4.31
276	0.000000	1.666667	1.800000	2.238462	2.460000	0.000000	0.000000	0.600000	0.600000	2.0	4.40	1.40

277 rows × 12 columns

In [193]:
```
X.describe()
```

Out[193]:

	0	1	2	3	4	5	6	7	8	9
count	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000	277.000000
mean	1.700468	2.008619	2.234676	2.460029	2.676644	0.514817	0.776017	1.040138	1.380646	1.89085
std	0.753311	0.632393	0.557434	0.467748	0.383121	0.439677	0.502911	0.506920	0.523581	0.33236
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.20000
25%	1.375000	1.740000	2.016667	2.250000	2.550000	0.200000	0.400000	0.666667	1.000000	2.00000
50%	1.860000	2.137500	2.335714	2.550000	2.760000	0.400000	0.700000	1.000000	1.500000	2.00000
75%	2.220000	2.425000	2.614286	2.807143	2.930769	0.800000	1.100000	1.400000	1.800000	2.00000
max	2.875000	3.000000	3.000000	3.000000	3.000000	2.000000	2.000000	2.000000	2.000000	2.00000

Splitting the dataset into the Training set and Test set

In [200]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,stratify=y, random_state = 1
```

In [197]:
```
print(X_train.shape,y_train.shape)
```

```
(221, 12) (221,)
```

In [198]:
```
print(X_test.shape,y_test.shape)
```

```
(56, 12) (56,)
```

Feature Scaling

In [201]:
```
temp=pd.DataFrame(X_train)
temp
```

Out[201]:

	0	1	2	3	4	5	6	7	8	9	10	11
3	1.025000	1.260000	1.285714	1.750000	2.625000	0.000000	0.0	0.2	0.4	2.0	5.06	2.81
86	1.628571	1.900000	2.100000	2.425000	2.561538	0.400000	0.5	0.7	1.0	1.8	6.93	4.68
261	2.376923	2.666667	2.750000	3.000000	3.000000	1.600000	2.0	2.0	2.0	2.0	11.06	7.21
47	2.490000	2.512500	2.700000	2.925000	2.957143	0.400000	0.6	0.8	1.5	2.0	10.12	6.93
16	2.250000	2.750000	2.812500	2.887500	2.892857	0.000000	0.2	0.5	0.6	2.0	3.75	4.12
...	...	...	...	...	...	...	...	...	...	...	...	...
0	2.125000	2.571429	2.700000	2.812500	3.000000	0.200000	0.9	0.9	1.0	1.5	9.75	8.62
159	2.670000	2.717822	2.875000	2.933333	3.000000	0.933333	1.1	1.4	1.6	2.0	8.25	8.06
230	2.175000	2.228571	2.820000	2.833333	2.940000	1.100000	1.2	1.4	1.6	2.0	2.15	3.28
101	2.550000	2.678571	2.750000	2.900000	2.930769	1.300000	1.6	1.6	1.8	2.0	8.62	9.00
214	0.960000	1.266667	2.250000	2.375000	2.700000	0.400000	1.0	1.4	1.7	2.0	3.93	2.43

221 rows × 12 columns

In [202]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [203]:
```
temp=pd.DataFrame(X_train)
temp
```

Out[203]:

	0	1	2	3	4	5	6	7	8	9	10	
0	-0.887339	-1.158167	-1.697522	-1.487167	-0.057675	-1.197376	-1.598425	-1.735126	-1.946669	0.337206	-0.301092	-0.76
1	-0.096651	-0.164754	-0.221063	-0.042399	-0.213026	-0.280202	-0.591203	-0.714925	-0.765994	-0.248817	0.485509	-0.07
2	0.883701	1.025272	0.957514	1.188329	0.860312	2.471322	2.430464	1.937597	1.201797	0.337206	2.222762	0.85
3	1.031834	0.785973	0.866854	1.027799	0.755399	-0.280202	-0.389758	-0.510885	0.217901	0.337206	1.827358	0.75
4	0.717430	1.154622	1.070838	0.947534	0.598030	-1.197376	-1.195536	-1.123006	-1.553111	0.337206	-0.852134	-0.28
...	...	...	...	...	...	...	...	...	...	...	...	
216	0.553678	0.877442	0.866854	0.787004	0.860312	-0.738789	0.214575	-0.306845	-0.765994	-1.127852	1.671720	1.38
217	1.267637	1.104675	1.184163	1.045635	0.860312	0.942698	0.617464	0.713356	0.414681	0.337206	1.040757	1.17
218	0.619179	0.345257	1.084437	0.831596	0.713434	1.324854	0.818909	0.713356	0.414681	0.337206	-1.525162	-0.59
219	1.110435	1.043750	0.957514	0.974289	0.690837	1.783441	1.624686	1.121436	0.808239	0.337206	1.196395	1.52
220	-0.972490	-1.147818	0.050916	-0.149419	0.125923	-0.280202	0.416020	0.713356	0.611460	0.337206	-0.776418	-0.90

221 rows × 12 columns

Grid Search

In [205]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

def grid_search_knn(x_train, y_train, param_grid, cv=10):
    """
    Perform grid search for hyperparameter tuning of K-Nearest Neighbors (KNN) classifier.

    Args:
        x_train (numpy.ndarray): Training features.
        y_train (numpy.ndarray): Training labels.
        param_grid (dict): Dictionary with parameters names (string) as keys and lists of
                           parameter settings to try as values.
        cv (int): Number of folds for cross-validation.

    Returns:
        sklearn.model_selection.GridSearchCV: GridSearchCV object with the best parameters.
    """
    knn_model = KNeighborsClassifier()
    grid_search = GridSearchCV(knn_model, param_grid, cv=cv)
    grid_search.fit(x_train, y_train)
    return grid_search.best_params_

def grid_search_decision_tree(x_train, y_train, param_grid, cv=10):
    """
    Perform grid search for hyperparameter tuning of Decision Tree classifier.

    Args:
        x_train (numpy.ndarray): Training features.
        y_train (numpy.ndarray): Training labels.
        param_grid (dict): Dictionary with parameters names (string) as keys and lists of
                           parameter settings to try as values.
        cv (int): Number of folds for cross-validation.

    Returns:
        sklearn.model_selection.GridSearchCV: GridSearchCV object with the best parameters.
    """
    tree_model = DecisionTreeClassifier()
    grid_search = GridSearchCV(tree_model, param_grid, cv=cv)
    grid_search.fit(x_train, y_train)
    return grid_search.best_params_
```

In [206]:
```
# Define the hyperparameters for KNN model
knn_params = {'algorithm': 'auto', 'n_neighbors': 7, 'weights': 'uniform'}

# Define the hyperparameters for Decision Tree model
tree_params = {'criterion': 'gini', 'max_depth': None}

# Create KNN model with best hyperparameters
knn_model = KNeighborsClassifier(**knn_params)

# Create Decision Tree model with best hyperparameters
tree_model = DecisionTreeClassifier(**tree_params)

# Assuming X_train and y_train are your training data
# Train the KNN model
knn_model.fit(X_train, y_train)

# Train the Decision Tree model
tree_model.fit(X_train, y_train)
```

Out[206]:
```
DecisionTreeClassifier
DecisionTreeClassifier()
```
(https://scikit-learn.org/1.4/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

In [207]:
```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score

def evaluate_model(model, X_test, y_test):
    """
    Evaluate the performance of the specified model on the test data.

    Args:
        model: Trained classifier model.
        X_test (array-like): Test features.
        y_test (array-like): True labels for the test data.

    Returns:
        dict: Dictionary containing performance metrics.
    """
    # Make predictions
    y_pred = model.predict(X_test)

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)

    # Accuracy
    accuracy = accuracy_score(y_test, y_pred)

    # Precision
    precision = precision_score(y_test, y_pred)

    # Recall (Sensitivity)
    recall = recall_score(y_test, y_pred)

    # F1-score
    f1 = f1_score(y_test, y_pred)

    # Specificity
    tn, fp, fn, tp = cm.ravel()
    specificity = tn / (tn + fp)

    # Create a dictionary to store performance metrics
    performance_metrics = {
        'confusion_matrix': cm,
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1_score': f1,
        'specificity': specificity
    }

    return performance_metrics

# Evaluate KNN model
knn_performance = evaluate_model(knn_model, X_test, y_test)

# Evaluate Decision Tree model
tree_performance = evaluate_model(tree_model, X_test, y_test)

# Print performance metrics for KNN model
print("Performance metrics for KNN model:")
for metric, value in knn_performance.items():
    print(f"{metric}: {value}")

# Print performance metrics for Decision Tree model
print("\nPerformance metrics for Decision Tree model:")
for metric, value in tree_performance.items():
    print(f"{metric}: {value}")
```

```
Performance metrics for KNN model:
confusion_matrix: [[22  4]
 [ 6 24]]
accuracy: 0.8214285714285714
precision: 0.8571428571428571
recall: 0.8
f1_score: 0.8275862068965517
specificity: 0.8461538461538461

Performance metrics for Decision Tree model:
confusion_matrix: [[20  6]
 [ 5 25]]
accuracy: 0.8035714285714286
precision: 0.8064516129032258
recall: 0.8333333333333334
f1_score: 0.819672131147541
specificity: 0.7692307692307693
```