HEART FAILURE

Cardiovascular diseases are the most common cause of deaths globally, taking an estimated 17.9 million lives each year, which accounts for 31% of all deaths worldwide. Heart failure is a common event caused by Cardiovascular diseases. It is characterized by the heart's inability to pump an adequate supply of blood to the body. Without sufficient blood flow, all major body functions are disrupted. Heart failure is a condition or a collection of symptoms that weaken the heart.

TABLE OF CONTENTS

IMPORTING LIBRARIES

In [1]:
```
import warnings
# Ignore all warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import seaborn as sns
from keras.layers import Dense, BatchNormalization, Dropout, LSTM
from keras.models import Sequential
from keras.utils import to_categorical
```

```python
from keras import callbacks
from sklearn.metrics import precision_score, recall_score, confusion_matrix, classificat
```

LOADING DATA

In [2]:
```python
#loading data
data = pd.read_csv("heart_failure_clinical_records_dataset.csv")
data.head()
```

Out[2]:

	age	anaemia	creatinine_phosphokinase	diabetes	ejection_fraction	high_blood_pressure	platelets	serum_cre
0	75.0	0	582	0	20	1	265000.00	
1	55.0	0	7861	0	38	0	263358.03	
2	65.0	0	146	0	20	0	162000.00	
3	50.0	1	111	0	20	0	210000.00	
4	65.0	1	160	1	20	0	327000.00	

In [3]:
```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 299 entries, 0 to 298
Data columns (total 13 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   age                       299 non-null    float64
 1   anaemia                   299 non-null    int64  
 2   creatinine_phosphokinase  299 non-null    int64  
 3   diabetes                  299 non-null    int64  
 4   ejection_fraction         299 non-null    int64  
 5   high_blood_pressure       299 non-null    int64  
 6   platelets                 299 non-null    float64
 7   serum_creatinine          299 non-null    float64
 8   serum_sodium              299 non-null    int64  
 9   sex                       299 non-null    int64  
 10  smoking                   299 non-null    int64  
 11  time                      299 non-null    int64  
 12  DEATH_EVENT               299 non-null    int64  
dtypes: float64(3), int64(10)
memory usage: 30.5 KB
```

About the data:

- age: Age of the patient
- anaemia: If the patient had the haemoglobin below the normal range
- creatinine_phosphokinase: The level of the creatine phosphokinase in the blood in mcg/L
- diabetes: If the patient was diabetic
- ejection_fraction: Ejection fraction is a measurement of how much blood the left ventricle pumps out with each contraction
- high_blood_pressure: If the patient had hypertension
- platelets: Platelet count of blood in kiloplatelets/mL
- serum_creatinine: The level of serum creatinine in the blood in mg/dL
- serum_sodium: The level of serum sodium in the blood in mEq/L
- sex: The sex of the patient
- smoking: If the patient smokes actively or ever did in past

- time: It is the time of the patient's follow-up visit for the disease in months
- DEATH_EVENT: If the patient deceased during the follow-up period

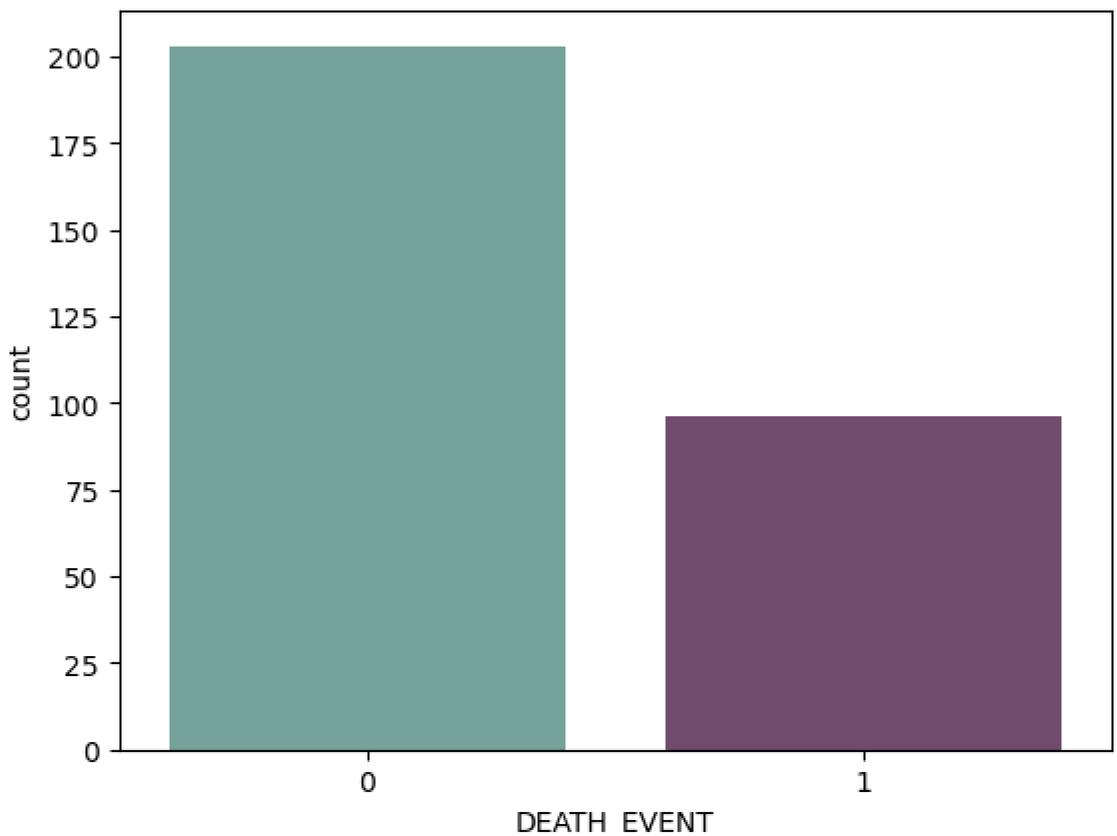
DATA ANALYSIS

Steps in data analysis and visulisation:

We begin our analysis by plotting a count plot of the targer attribute. A corelation matrix od the various attributes to examine the feature importance.

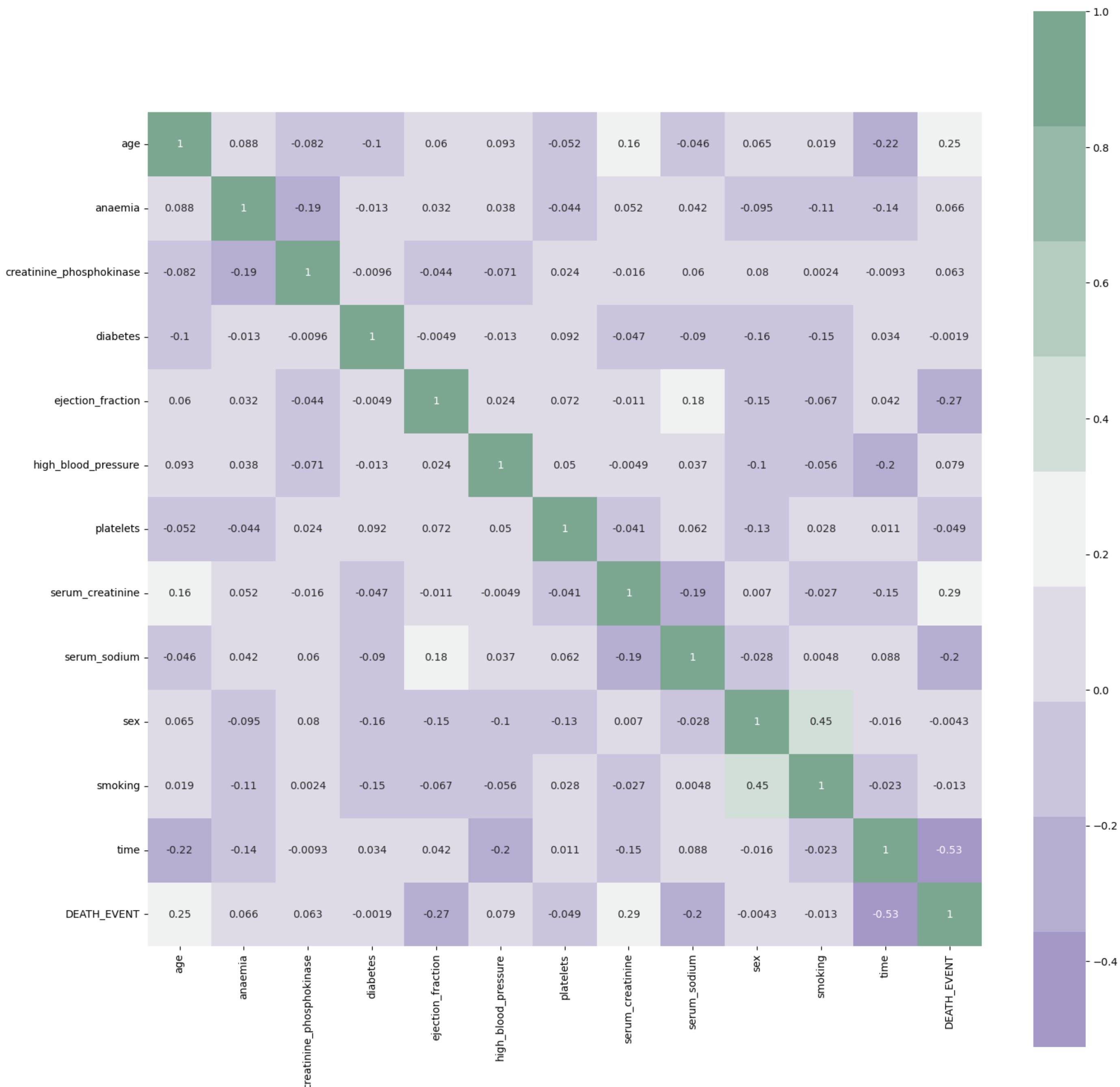
In [4]:
```
#first of all let us evaluate the target and find out if our data is imbalanced or not
cols= ["#6daa9f","#774571"]
sns.countplot(x= data["DEATH_EVENT"], palette= cols)
```

Out[4]:
```
<AxesSubplot:xlabel='DEATH_EVENT', ylabel='count'>
```



Point to note is that there is an imbalance in the data.

In [5]:
```
#Examaning a corelation matrix of all the features
cmap = sns.diverging_palette(275,150,  s=40, l=65, n=9)
corrmat = data.corr()
plt.subplots(figsize=(18,18))
sns.heatmap(corrmat,cmap= cmap,annot=True, square=True);
```



Notable points:

- Time of the patient's follow-up visit for the disease is crucial in as initial diagnosis with cardiovascular issue and treatment reduces the chances of any fatality. It holds and inverse relation.
- Ejection fraction is the second most important feature. It is quite expected as it is basically the efficiency of the heart.
- Age of the patient is the third most correlated feature. Clearly as heart's functioning declines with ageing

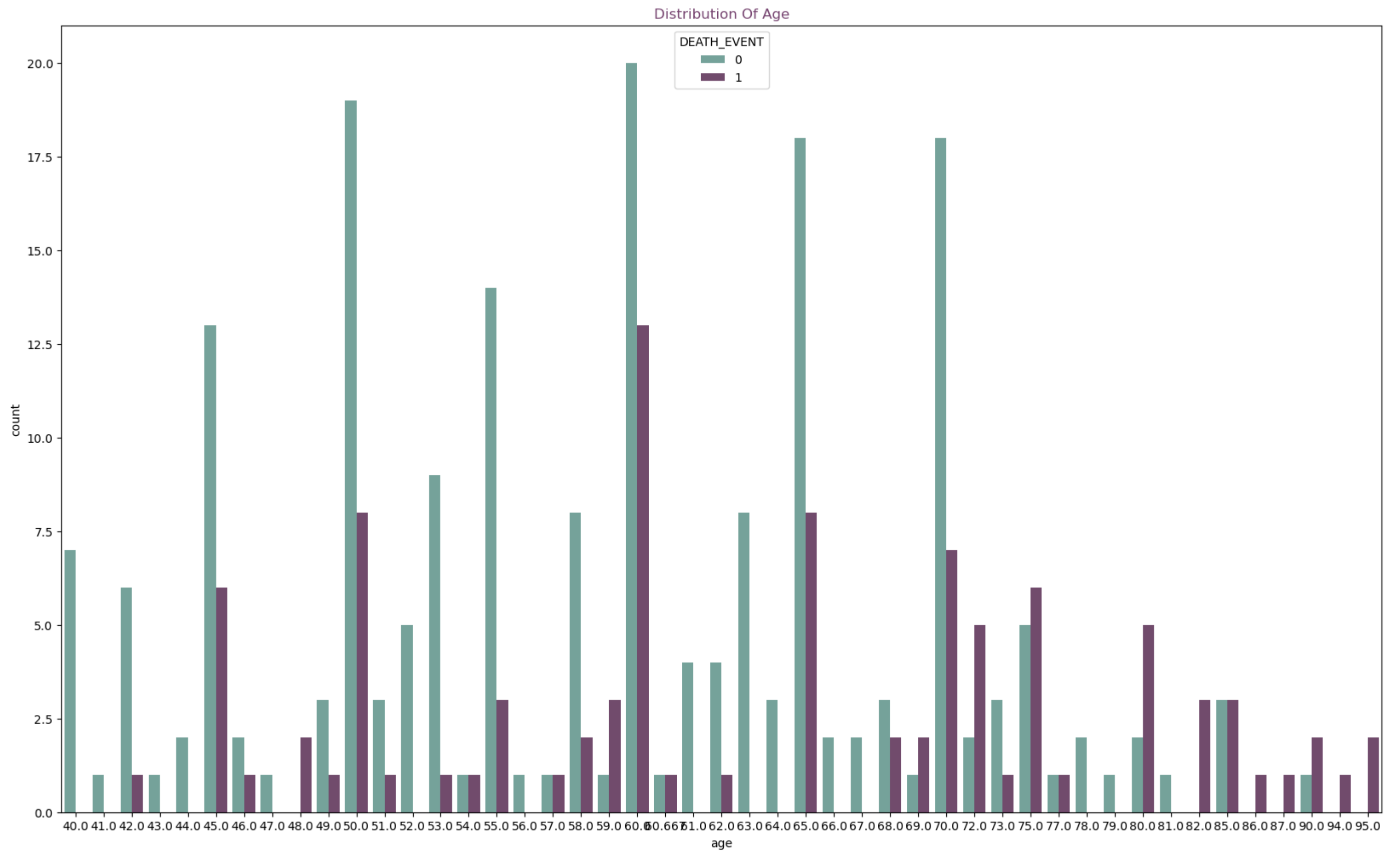
Next, we will examine the count plot of age.

In [6]:

```
#Evauating age distrivution
plt.figure(figsize=(20,12))
#colours =["#774571","#b398af","#f1f1f1" ,"#afcdc7", "#6daa9f"]
Days_of_week=sns.countplot(x=data['age'],data=data, hue ="DEATH_EVENT",palette = cols)
Days_of_week.set_title("Distribution Of Age", color="#774571")
```

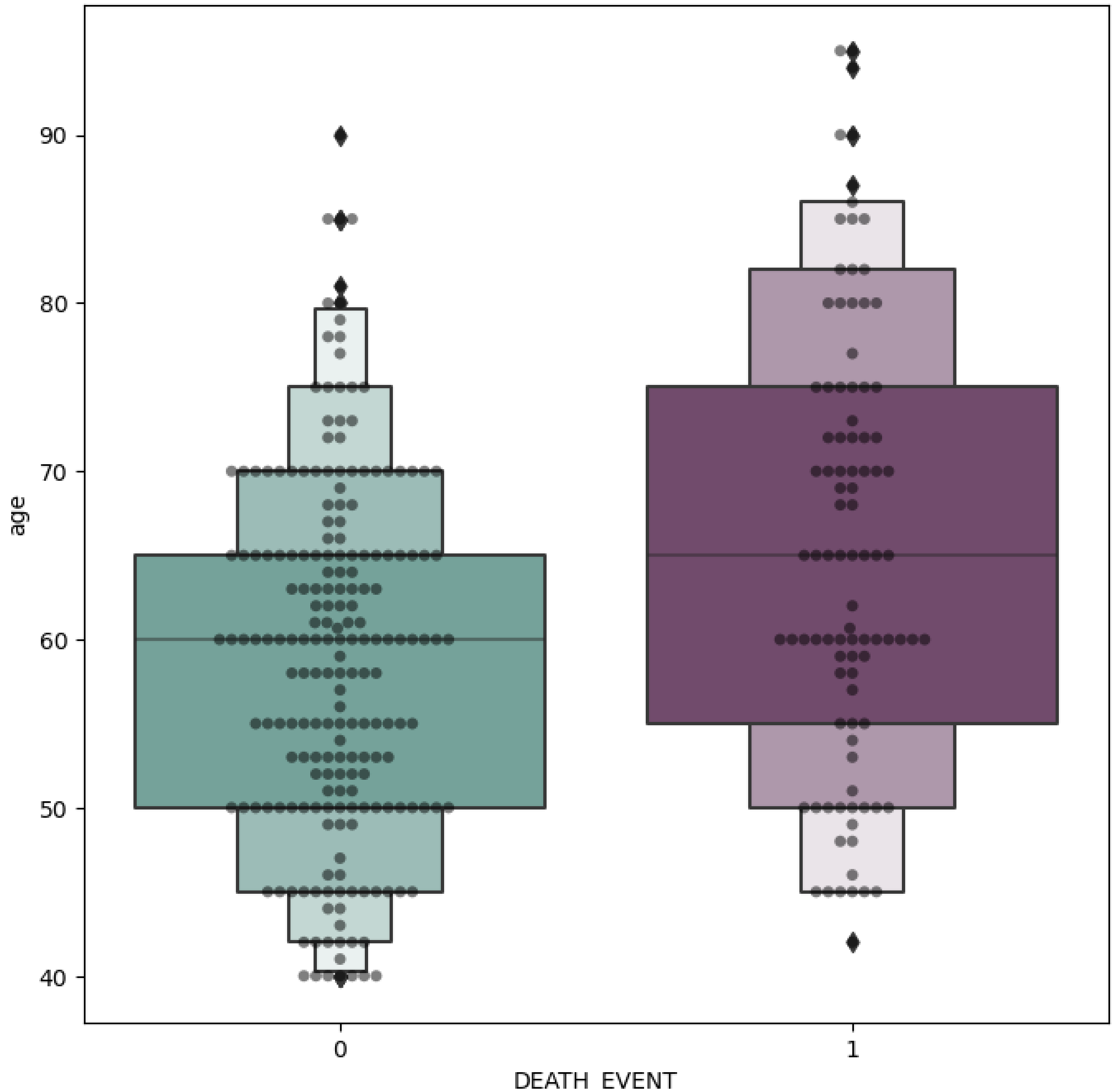
Out[6]:

```
Text(0.5, 1.0, 'Distribution Of Age')
```

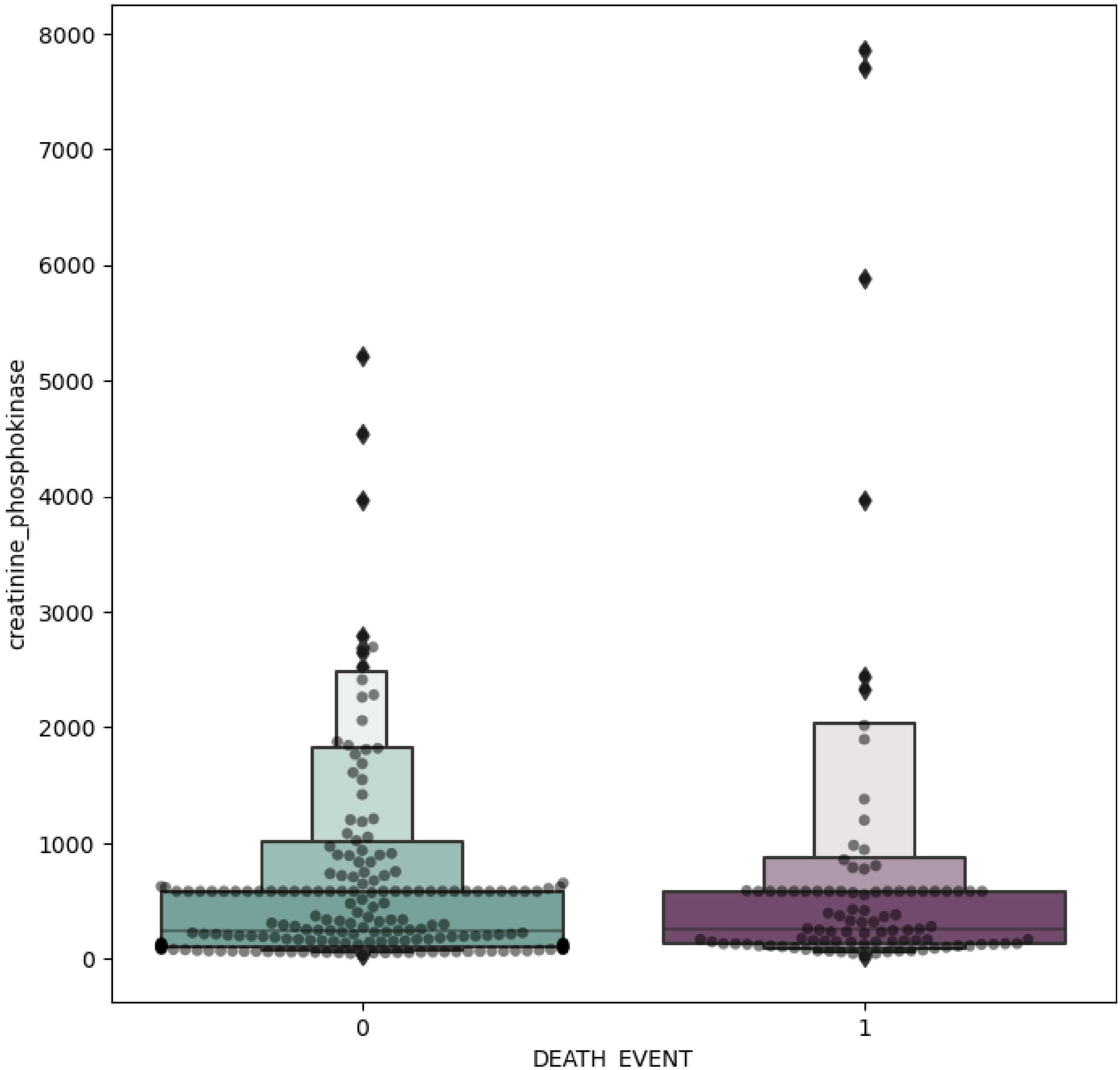


In [7]:

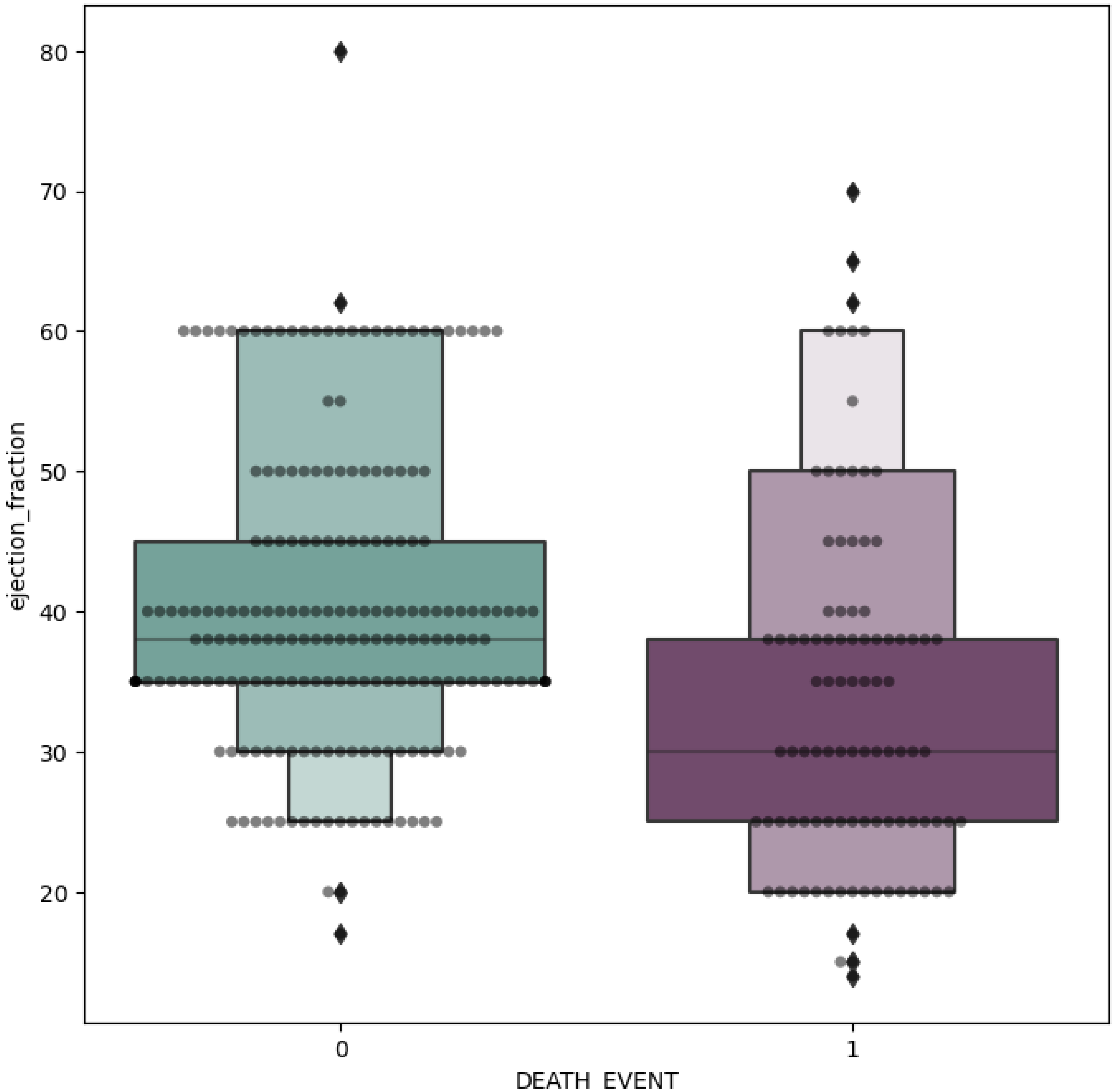
```
# Boxen and swarm plot of some non binary features.
feature = ["age","creatinine_phosphokinase","ejection_fraction","platelets","serum_creat
for i in feature:
    plt.figure(figsize=(8,8))
    sns.swarmplot(x=data["DEATH_EVENT"], y=data[i], color="black", alpha=0.5)
    sns.boxenplot(x=data["DEATH_EVENT"], y=data[i], palette=cols)
    plt.show()
```



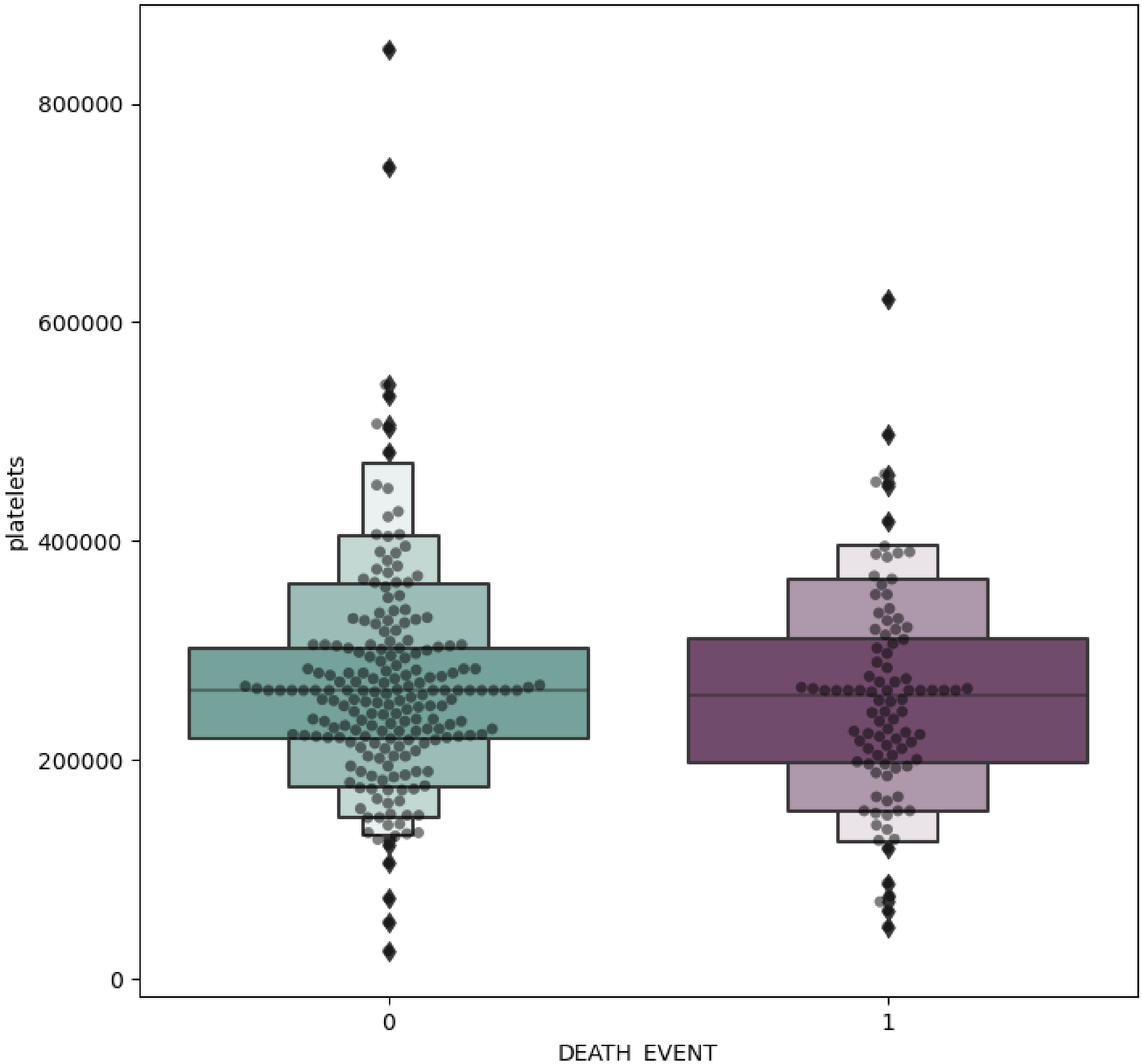
90
80
70
60
50
40
age
0
1
DEATH_EVENT

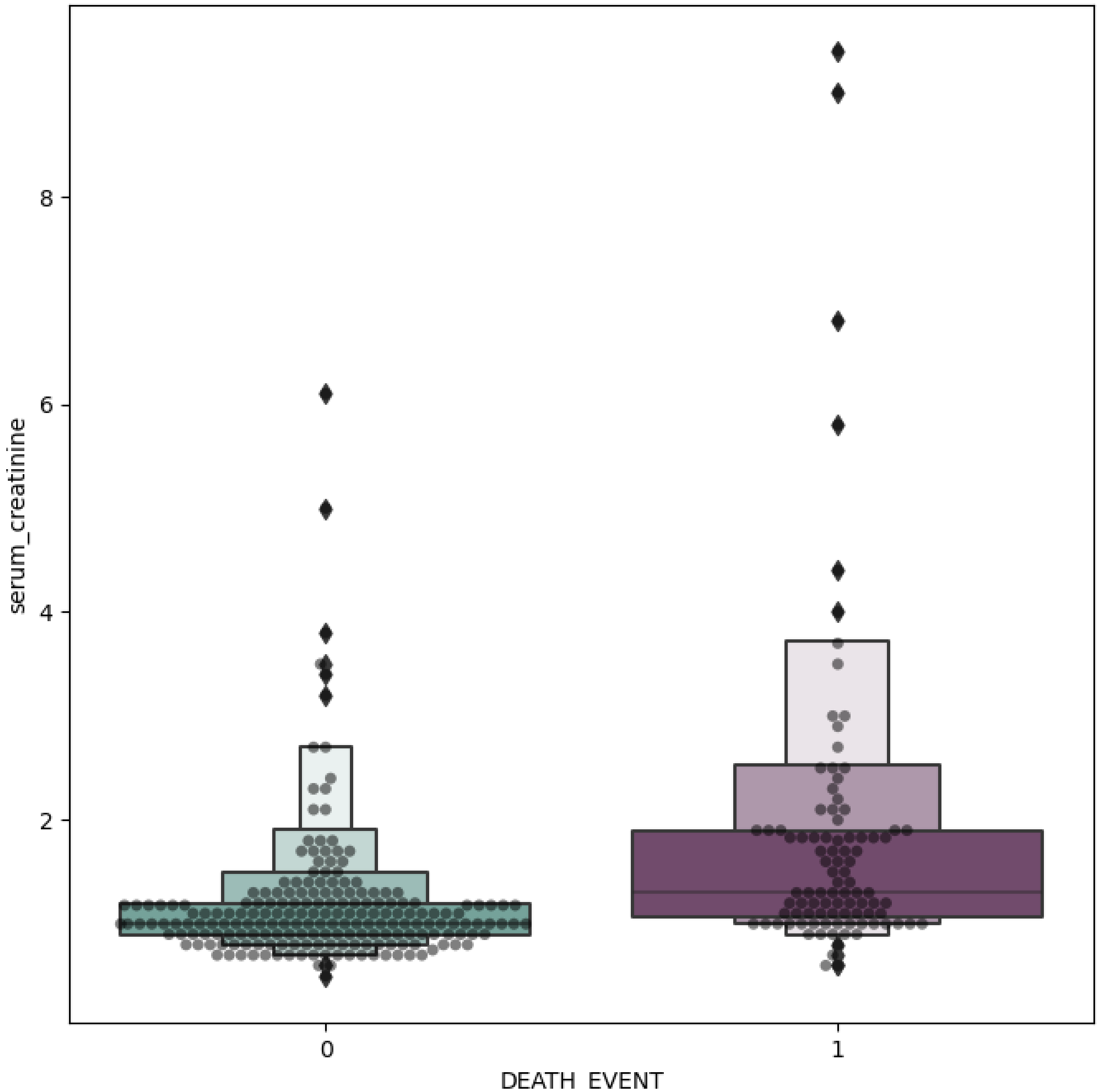


8000
7000
6000
5000
4000
3000
2000
1000
0
creatinine_phosphokinase
0
1
DEATH_EVENT

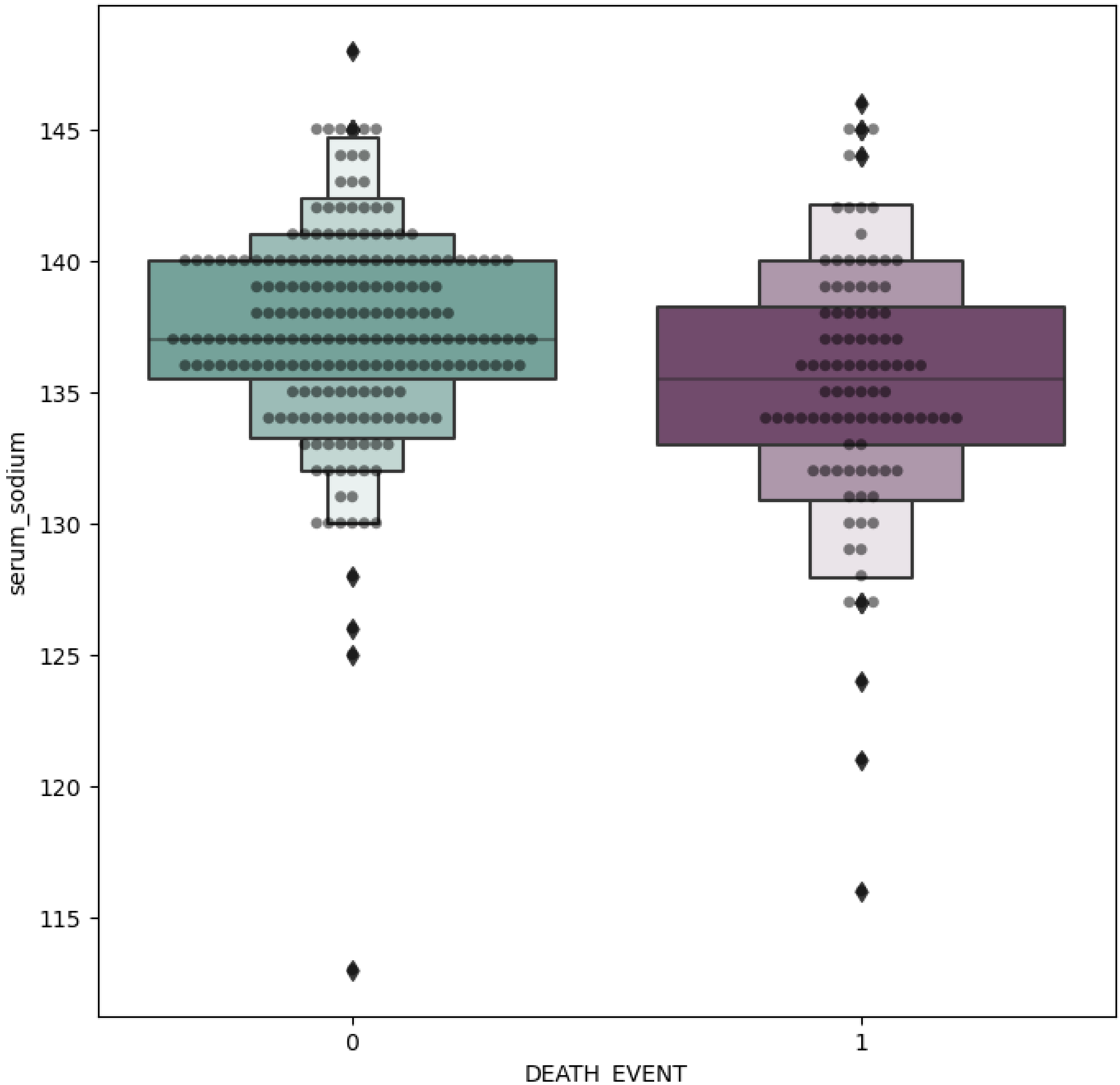


80
70
60
50
40
30
20
ejection_fraction
0
1
DEATH_EVENT

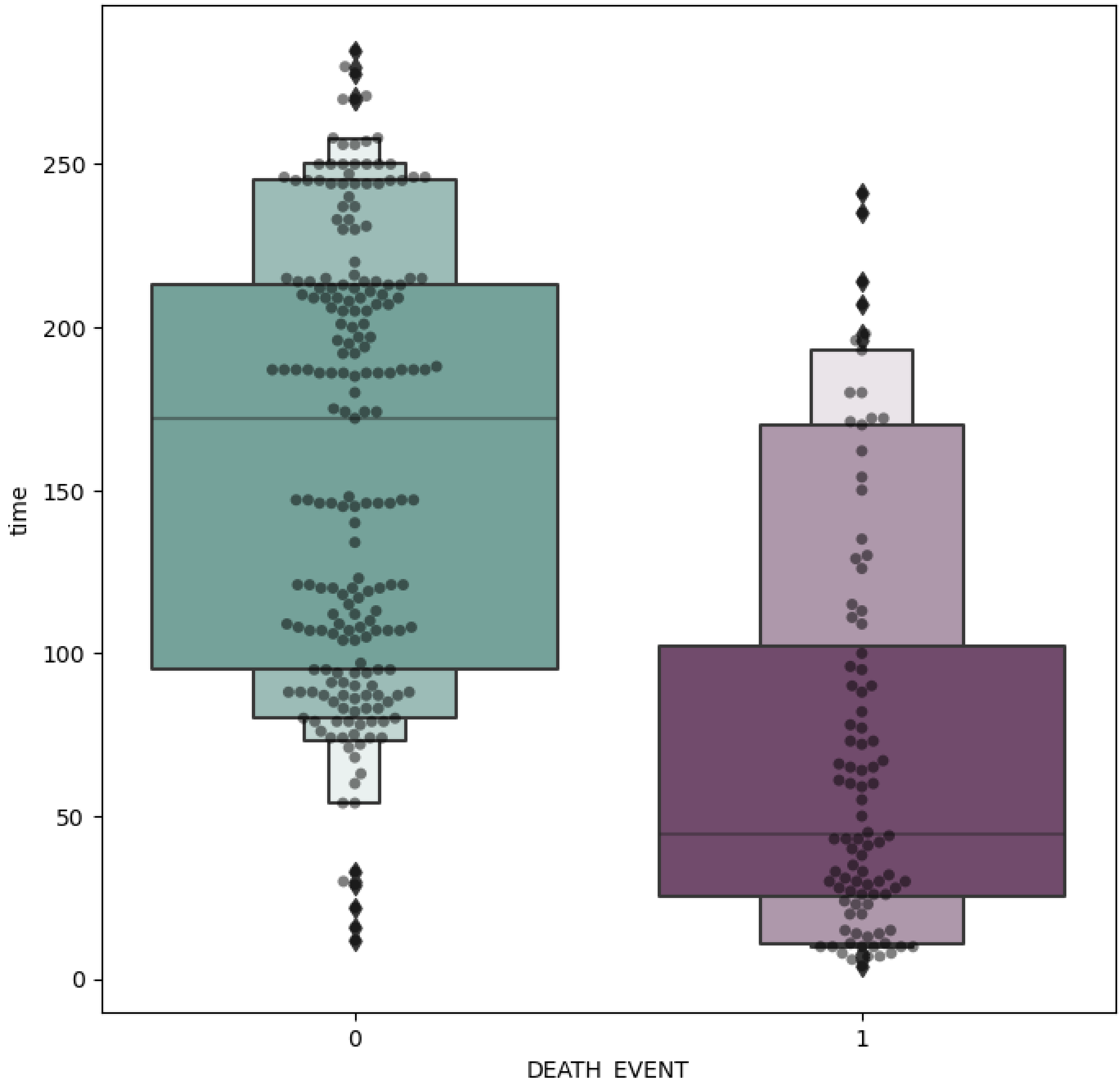

800000
600000
400000
200000
0
platelets
0
1
DEATH_EVENT



8
6
4
2
serum_creatinine
0
1
DEATH_EVENT



145
140
135
130
125
120
115
serum_sodium
0
1
DEATH_EVENT

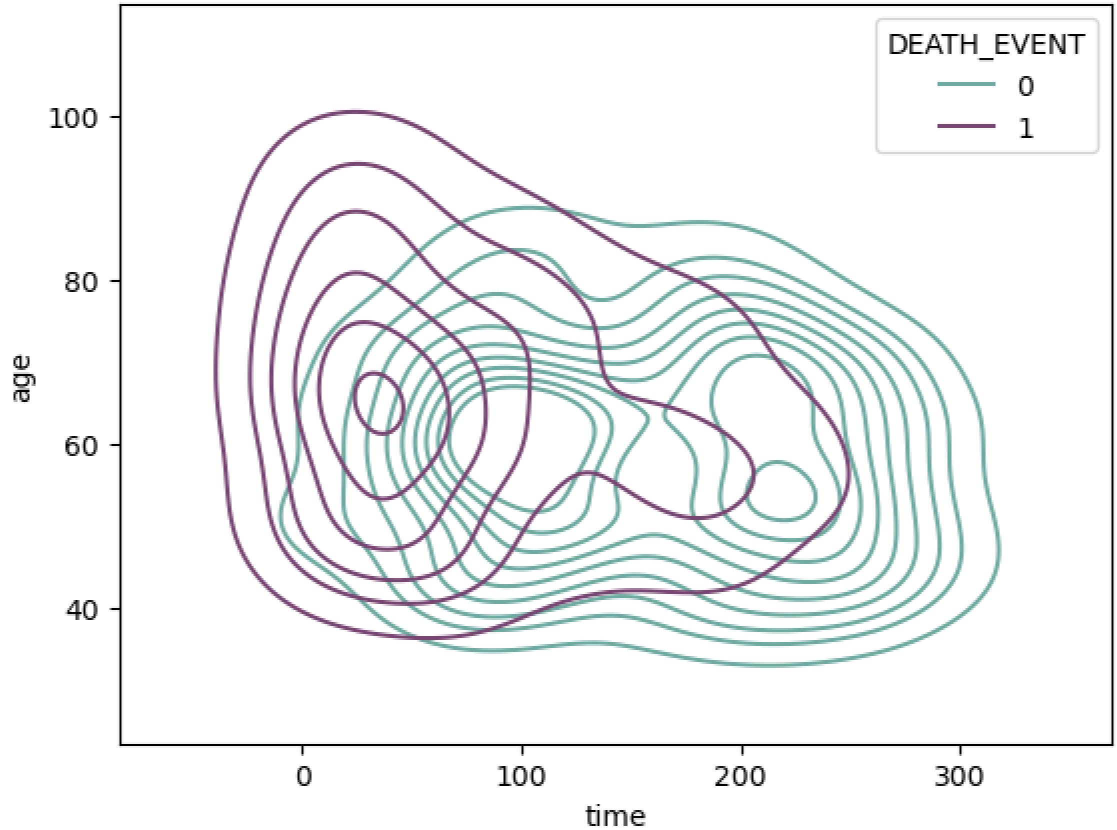


I spotted outliers on our dataset. I didn't remove them yet as it may lead to overfitting. Though we may end up with better statistics. In this case, with medical data, the outliers may be an important deciding factor.

Next, we examine the kdeplot of time and age as they both are significant features.

In [8]:
```
sns.kdeplot(x=data["time"], y=data["age"], hue =data["DEATH_EVENT"], palette=cols)
```

Out[8]:
```
<AxesSubplot:xlabel='time', ylabel='age'>
```



In [9]:
```
data.describe().T
```

Out[9]:

	count	mean	std	min	25%	50%	75%	max
age	299.0	60.833893	11.894809	40.0	51.0	60.0	70.0	95.0
anaemia	299.0	0.431438	0.496107	0.0	0.0	0.0	1.0	1.0
creatinine_phosphokinase	299.0	581.839465	970.287881	23.0	116.5	250.0	582.0	7861.0
diabetes	299.0	0.418060	0.494067	0.0	0.0	0.0	1.0	1.0
ejection_fraction	299.0	38.083612	11.834841	14.0	30.0	38.0	45.0	80.0
high_blood_pressure	299.0	0.351171	0.478136	0.0	0.0	0.0	1.0	1.0
platelets	299.0	263358.029264	97804.236869	25100.0	212500.0	262000.0	303500.0	850000.0
serum_creatinine	299.0	1.393880	1.034510	0.5	0.9	1.1	1.4	9.4
serum_sodium	299.0	136.625418	4.412477	113.0	134.0	137.0	140.0	148.0
sex	299.0	0.648829	0.478136	0.0	0.0	1.0	1.0	1.0
smoking	299.0	0.321070	0.467670	0.0	0.0	0.0	1.0	1.0
time	299.0	130.260870	77.614208	4.0	73.0	115.0	203.0	285.0
DEATH_EVENT	299.0	0.321070	0.467670	0.0	0.0	0.0	1.0	1.0

DATA PREPROCESSING

Steps involved in Data Preprocessing

- Dropping the outliers based on data analysis
- Assigning values to features as X and target as y

- Perform the scaling of the features
- Split test and training sets

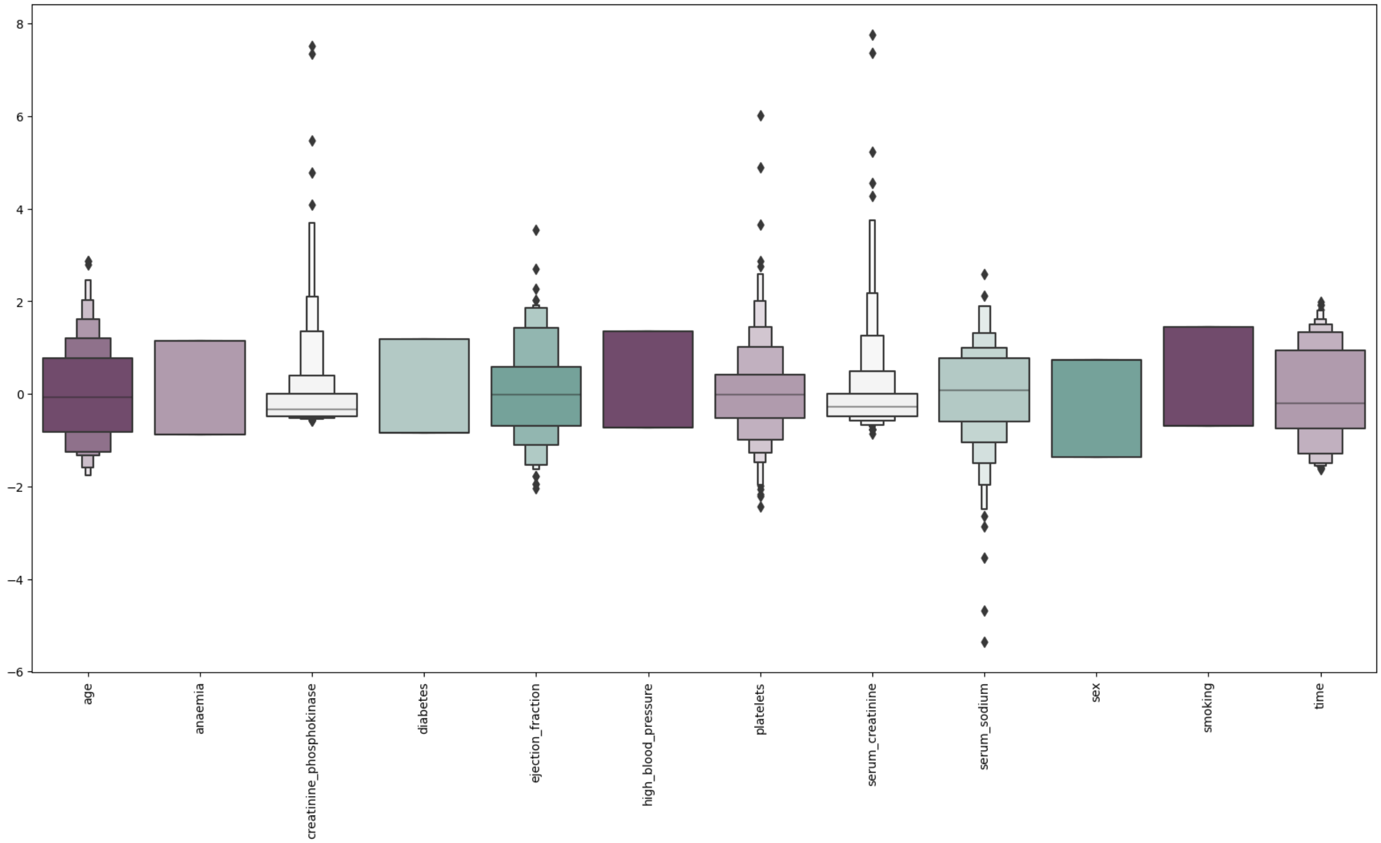
In [10]:
```
#assigning values to features as X and target as y
X=data.drop(["DEATH_EVENT"],axis=1)
y=data["DEATH_EVENT"]
```

In [11]:
```
#Set up a standard scaler for the features
col_names = list(X.columns)
s_scaler = preprocessing.StandardScaler()
X_df= s_scaler.fit_transform(X)
X_df = pd.DataFrame(X_df, columns=col_names)
X_df.describe().T
```

Out[11]:

	count	mean	std	min	25%	50%	75%	max
age	299.0	5.265205e-16	1.001676	-1.754448	-0.828124	-0.070223	0.771889	2.877170
anaemia	299.0	3.594301e-16	1.001676	-0.871105	-0.871105	-0.871105	1.147968	1.147968
creatinine_phosphokinase	299.0	3.713120e-18	1.001676	-0.576918	-0.480393	-0.342574	0.000166	7.514640
diabetes	299.0	1.113936e-16	1.001676	-0.847579	-0.847579	-0.847579	1.179830	1.179830
ejection_fraction	299.0	3.341808e-18	1.001676	-2.038387	-0.684180	-0.007077	0.585389	3.547716
high_blood_pressure	299.0	-4.841909e-16	1.001676	-0.735688	-0.735688	-0.735688	1.359272	1.359272
platelets	299.0	1.009969e-16	1.001676	-2.440155	-0.520870	-0.013908	0.411120	6.008180
serum_creatinine	299.0	-2.227872e-18	1.001676	-0.865509	-0.478205	-0.284552	0.005926	7.752020
serum_sodium	299.0	-8.627435e-16	1.001676	-5.363206	-0.595996	0.085034	0.766064	2.582144
sex	299.0	-5.940993e-18	1.001676	-1.359272	-1.359272	0.735688	0.735688	0.735688
smoking	299.0	-3.861645e-17	1.001676	-0.687682	-0.687682	-0.687682	1.454161	1.454161
time	299.0	-1.069379e-16	1.001676	-1.629502	-0.739000	-0.196954	0.938759	1.997038

In [12]:
```
#looking at the scaled features
colours =["#774571","#b398af","#f1f1f1" ,"#afcdc7", "#6daa9f"]
plt.figure(figsize=(20,10))
sns.boxenplot(data = X_df,palette = colours)
plt.xticks(rotation=90)
plt.show()
```



In [13]:

```
#spliting test and training sets
X_train, X_test, y_train,y_test = train_test_split(X_df,y,test_size=0.25,random_state=7)
```

MODEL BUILDING

In this project, we build an artificial neural network.

Following steps are involved in the model building

- Initialising the ANN
- Defining by adding layers
- Compiling the ANN
- Train the ANN

In [14]:

```
early_stopping = callbacks.EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True)

# Initialising the NN
model = Sequential()

# layers
model.add(Dense(units = 16, kernel_initializer = 'uniform', activation = 'relu', input_d
model.add(Dense(units = 8, kernel_initializer = 'uniform', activation = 'relu'))
model.add(Dropout(0.25))
model.add(Dense(units = 4, kernel_initializer = 'uniform', activation = 'relu'))
model.add(Dropout(0.5))
model.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
from keras.optimizers import SGD
# Compiling the ANN
model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

```
# Train the ANN
history = model.fit(X_train, y_train, batch_size = 32, epochs = 500,callbacks=[early_sto
```

```
Epoch 1/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 3s 131ms/step - accuracy: 0.6440 - loss: 0.6929 -
 val_accuracy: 0.6667 - val_loss: 0.6925
Epoch 2/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6339 - loss: 0.6921 -
 val_accuracy: 0.6667 - val_loss: 0.6919
Epoch 3/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.6419 - loss: 0.6913 -
 val_accuracy: 0.6667 - val_loss: 0.6914
Epoch 4/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.6837 - loss: 0.6899 -
 val_accuracy: 0.6667 - val_loss: 0.6908
Epoch 5/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.6223 - loss: 0.6903 -
 val_accuracy: 0.6667 - val_loss: 0.6902
Epoch 6/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.6677 - loss: 0.6882 -
 val_accuracy: 0.6667 - val_loss: 0.6896
Epoch 7/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.6884 - loss: 0.6864 -
 val_accuracy: 0.6667 - val_loss: 0.6889
Epoch 8/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.6445 - loss: 0.6872 -
 val_accuracy: 0.6667 - val_loss: 0.6882
Epoch 9/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.6396 - loss: 0.6864 -
 val_accuracy: 0.6667 - val_loss: 0.6873
Epoch 10/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6747 - loss: 0.6830 -
 val_accuracy: 0.6667 - val_loss: 0.6862
Epoch 11/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6947 - loss: 0.6809 -
 val_accuracy: 0.6667 - val_loss: 0.6848
Epoch 12/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.6410 - loss: 0.6817 -
 val_accuracy: 0.6667 - val_loss: 0.6831
Epoch 13/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.6232 - loss: 0.6821 -
 val_accuracy: 0.6667 - val_loss: 0.6812
Epoch 14/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.6163 - loss: 0.6787 -
 val_accuracy: 0.6667 - val_loss: 0.6785
Epoch 15/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.6166 - loss: 0.6741 -
 val_accuracy: 0.6667 - val_loss: 0.6749
Epoch 16/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.6421 - loss: 0.6688 -
 val_accuracy: 0.6667 - val_loss: 0.6700
Epoch 17/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.6113 - loss: 0.6706 -
 val_accuracy: 0.6667 - val_loss: 0.6639
Epoch 18/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.6233 - loss: 0.6553 -
 val_accuracy: 0.6667 - val_loss: 0.6559
Epoch 19/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.6374 - loss: 0.6515 -
 val_accuracy: 0.6667 - val_loss: 0.6461
Epoch 20/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.6624 - loss: 0.6335 -
 val_accuracy: 0.6667 - val_loss: 0.6341
Epoch 21/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6474 - loss: 0.6353 -
 val_accuracy: 0.6667 - val_loss: 0.6215
```

```
Epoch 22/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.5984 - loss: 0.6355 -
 val_accuracy: 0.6667 - val_loss: 0.6085
Epoch 23/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6496 - loss: 0.5958 -
 val_accuracy: 0.6667 - val_loss: 0.5946
Epoch 24/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.6497 - loss: 0.5913 -
 val_accuracy: 0.6667 - val_loss: 0.5816
Epoch 25/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.6470 - loss: 0.5672 -
 val_accuracy: 0.6667 - val_loss: 0.5702
Epoch 26/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.6678 - loss: 0.5519 -
 val_accuracy: 0.6667 - val_loss: 0.5592
Epoch 27/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.6683 - loss: 0.5596 -
 val_accuracy: 0.6667 - val_loss: 0.5490
Epoch 28/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.6305 - loss: 0.5628 -
 val_accuracy: 0.6667 - val_loss: 0.5419
Epoch 29/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.6318 - loss: 0.5165 -
 val_accuracy: 0.6667 - val_loss: 0.5353
Epoch 30/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.6287 - loss: 0.5583 -
 val_accuracy: 0.6667 - val_loss: 0.5329
Epoch 31/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.6449 - loss: 0.5171 -
 val_accuracy: 0.6667 - val_loss: 0.5312
Epoch 32/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.6497 - loss: 0.5437 -
 val_accuracy: 0.6667 - val_loss: 0.5304
Epoch 33/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.6607 - loss: 0.5086 -
 val_accuracy: 0.6667 - val_loss: 0.5299
Epoch 34/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.6422 - loss: 0.5231 -
 val_accuracy: 0.6667 - val_loss: 0.5300
Epoch 35/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6480 - loss: 0.5086 -
 val_accuracy: 0.6667 - val_loss: 0.5294
Epoch 36/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6487 - loss: 0.4659 -
 val_accuracy: 0.6667 - val_loss: 0.5288
Epoch 37/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.6186 - loss: 0.5191 -
 val_accuracy: 0.6667 - val_loss: 0.5281
Epoch 38/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.6193 - loss: 0.5148 -
 val_accuracy: 0.6667 - val_loss: 0.5268
Epoch 39/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6225 - loss: 0.5158 -
 val_accuracy: 0.6667 - val_loss: 0.5280
Epoch 40/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6823 - loss: 0.5175 -
 val_accuracy: 0.6667 - val_loss: 0.5284
Epoch 41/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.6108 - loss: 0.5244 -
 val_accuracy: 0.6667 - val_loss: 0.5291
Epoch 42/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6448 - loss: 0.4900 -
 val_accuracy: 0.6667 - val_loss: 0.5302
Epoch 43/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.6710 - loss: 0.4708 -
 val_accuracy: 0.6667 - val_loss: 0.5297
```

```
Epoch 44/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.6376 - loss: 0.5094 -
 val_accuracy: 0.6667 - val_loss: 0.5296
Epoch 45/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.6181 - loss: 0.5061 -
 val_accuracy: 0.6667 - val_loss: 0.5303
Epoch 46/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 0.6398 - loss: 0.4654 -
 val_accuracy: 0.6667 - val_loss: 0.5314
Epoch 47/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - accuracy: 0.6411 - loss: 0.4780 -
 val_accuracy: 0.6667 - val_loss: 0.5325
Epoch 48/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.6625 - loss: 0.4588 -
 val_accuracy: 0.6667 - val_loss: 0.5349
Epoch 49/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - accuracy: 0.6305 - loss: 0.4837 -
 val_accuracy: 0.6667 - val_loss: 0.5369
Epoch 50/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - accuracy: 0.6423 - loss: 0.5162 -
 val_accuracy: 0.6667 - val_loss: 0.5381
Epoch 51/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step - accuracy: 0.6649 - loss: 0.4574 -
 val_accuracy: 0.6667 - val_loss: 0.5380
Epoch 52/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step - accuracy: 0.6317 - loss: 0.5051 -
 val_accuracy: 0.6667 - val_loss: 0.5374
Epoch 53/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - accuracy: 0.6610 - loss: 0.4402 -
 val_accuracy: 0.6667 - val_loss: 0.5378
Epoch 54/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.6121 - loss: 0.4993 -
 val_accuracy: 0.6667 - val_loss: 0.5358
Epoch 55/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.6444 - loss: 0.5166 -
 val_accuracy: 0.6667 - val_loss: 0.5339
Epoch 56/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.6445 - loss: 0.4649 -
 val_accuracy: 0.6667 - val_loss: 0.5320
Epoch 57/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.6571 - loss: 0.4507 -
 val_accuracy: 0.6667 - val_loss: 0.5308
Epoch 58/500
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6390 - loss: 0.4482 -
 val_accuracy: 0.6667 - val_loss: 0.5311
```

In [15]:

```
val_accuracy = np.mean(history.history['val_accuracy'])
print("\n%s: %.2f%%" % ('val_accuracy', val_accuracy*100))
```

```
val_accuracy: 66.67%
```

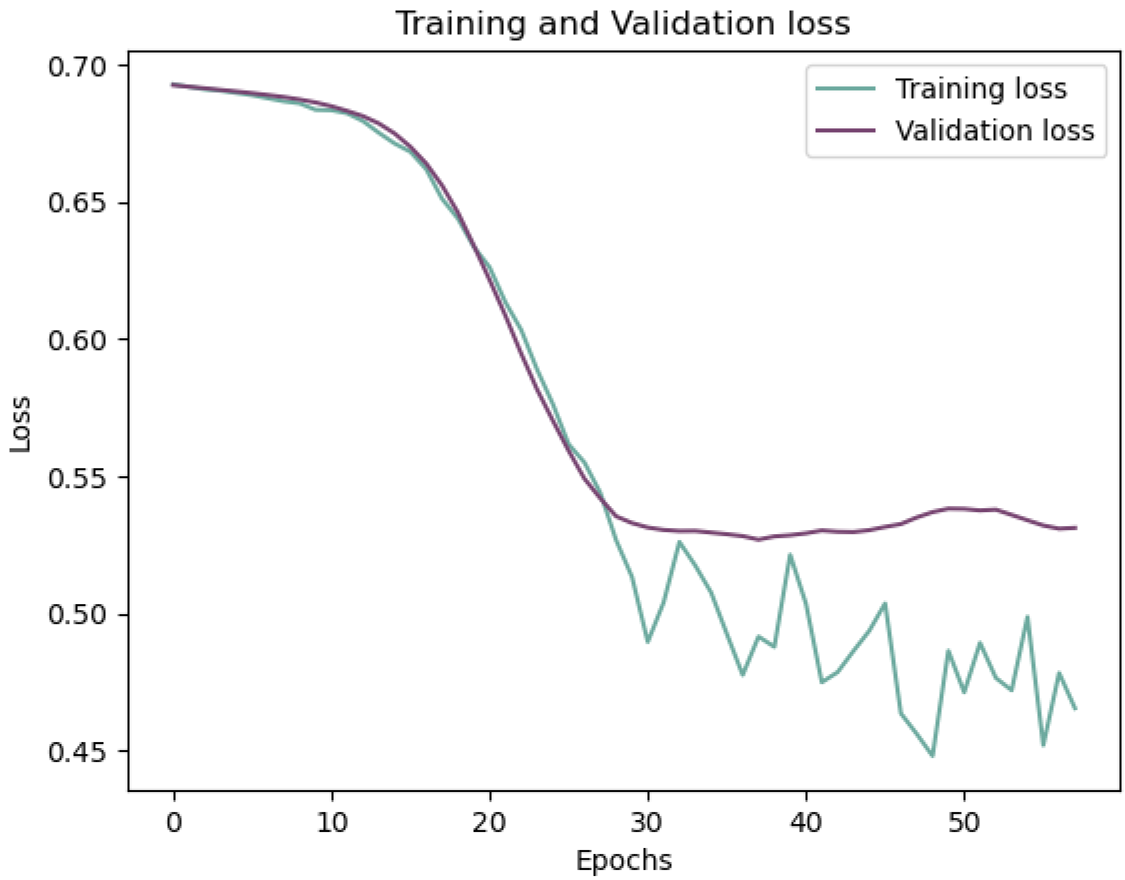
Plotting training and validation loss over epochs

In [16]:

```
history_df = pd.DataFrame(history.history)

plt.plot(history_df.loc[:, ['loss']], "#6daa9f", label='Training loss')
plt.plot(history_df.loc[:, ['val_loss']],"#774571", label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc="best")

plt.show()
```

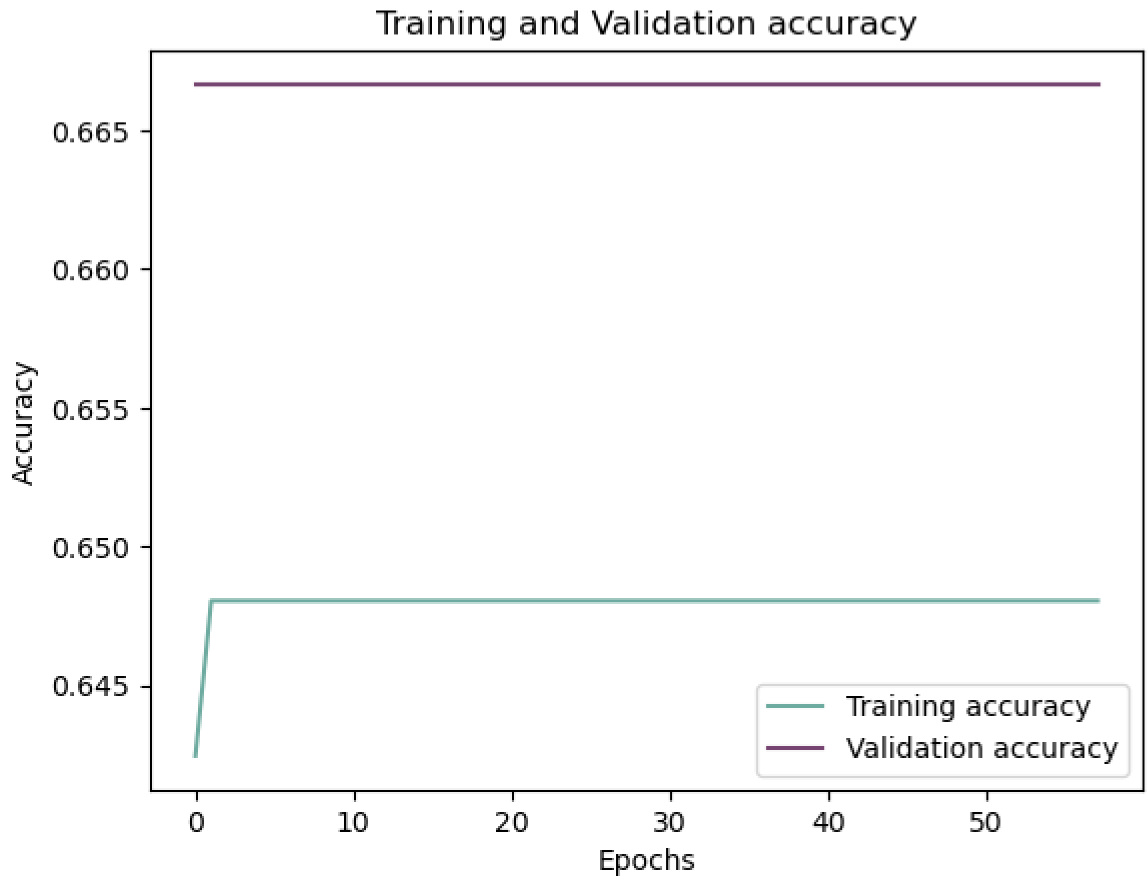


Plotting training and validation accuracy over epochs

In [17]:
```
history_df = pd.DataFrame(history.history)

plt.plot(history_df.loc[:, ['accuracy']], "#6daa9f", label='Training accuracy')
plt.plot(history_df.loc[:, ['val_accuracy']], "#774571", label='Validation accuracy')

plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



CONCLUSIONS

Concluding the model with:

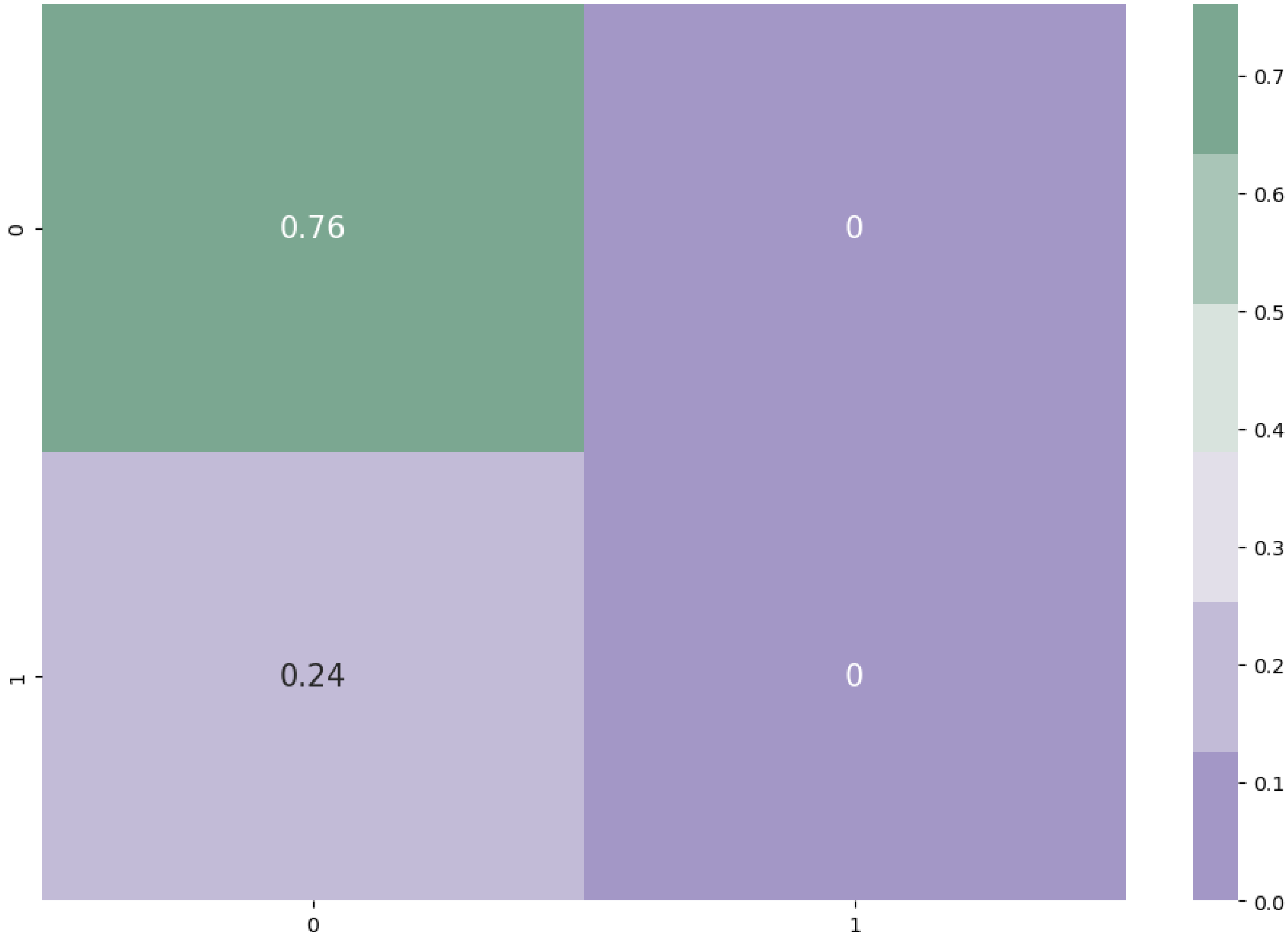
- Testing on the test set
- Evaluating the confusion matrix
- Evaluating the classification report

In [18]:
```
# Predicting the test set results
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5)
np.set_printoptions()
```

3/3 ━━━━━━━━━━━━━━━━━━━━ **0s** 59ms/step

In [19]:
```
# confusion matrix
cmap1 = sns.diverging_palette(275,150,  s=40, l=65, n=6)
plt.subplots(figsize=(12,8))
cf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(cf_matrix/np.sum(cf_matrix), cmap = cmap1, annot = True, annot_kws = {'size'
```

Out[19]: <AxesSubplot:>



In [20]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.76      1.00      0.86        57
           1       0.00      0.00      0.00        18

    accuracy                           0.76        75
   macro avg       0.38      0.50      0.43        75
weighted avg       0.58      0.76      0.66        75
```

THE END

In [ ]: