Project

EE P 596: Advanced Introduction to Machine Learning

Arrhythmia heartbeat classification Win23

Predict the heartbeat type based on the input heartbeat.

Due March 1st, 2023, by 11:59 PM

Team Kaggle Name: Rhythm Detectives

Students: Naif Ganadily & Sarah Selim

Instructor - Prof. Karthik Mohan
TA - Ayush Singh
Grader - Fatwir SM

Project Overview:

Arrhythmia or "irregular heart beats" is a very common heart rate problem and often goes un-diagnosed. This Project looks into super fine-grained data on heart beats and characteristics of normal and abnormal heartbeats. Having ML algorithms that can automate detection of possible Arrythmia is super impactful in helping doctors and hospitals be more efficient and effective in diagnosis and treatment of heart rate issues and also avert medical emergencies, prevent deaths.

Submission Guidelines:

- You get to work in teams of 2 for the Kaggle and modeling piece!! Please make sure each person of the team gets to work on all aspects of the mini-project and mention at the top of your report the contributions from each person.
  - The submission is in 2 parts.
- **Code:** Please submit a Jupyter/IPython notebook file, report and Kaggle predictions as part of your submission. You can start with the template notebook provided and add in your solutions to it.
  - **Report:** The report should be in a pdf format and have plots, correlation matrices and tables added in as mentioned in the **Heart Rate Delivariables below**. Feel free to use either LaTeX or word for creating it. Include answers to conceptual questions, and your insights as well. Ideally you should NOT use comments in ipynd to answer any conceptual question.
- **Kaggle Contest:** There is a Kaggle competition as well, where you submit predictions on a "held out" data set.

Dataset Description:

We have the following classes of heartbeats present in the dataset:

- N: Normal beat
  - L: Left bundle branch block beat
- R: Right bundle branch block beat
  - A: Atrial premature beat
- V: Premature ventricular contraction
  - U: All other types of beats should be classified as this (this would require relabelling of the data)

The dataset contains 44 half-hour excerpts of two-channel ambulatory ECG recordings, obtained from 43 subjects studied by the BIH Arrhythmia Laboratory between 1975 and 1979. The recordings were digitized at 360 samples per second per channel, and were labelled manually by cardiologists. You can only use MLII information to train the model for all the part except the last one, this is because we have maximum availablity of this feature. The txt file contains time, sample number and type of the heartbeat.

Required Preprocessing:

The first objective is to split MIT-BIH record at the R-peaks into individual heartbeat records. This can be done by creating a file which shall have the required information from the csv and txt files. The txt file contains time, sample number and type of the heartbeat. For each row of txt files, take 180 samples before and 179 samples after this sample number to create a time series from the corresponding csv file with the corresponding type as the label. Hence the final file you shall create shall have 360 features, along with it's label. Feel free to try out any other pre-processing, and clearly explain the steps taken for it in the report.

Heartbeat Prediction Deliverables

1. Do the preprocessing defined in the previous cell. Then plot 3 heartbeats which are classified N and 3 which are classified as some other class. Is there a visible difference between these? How many heartbeats do you have in total? (30 points)

2. Data normalization (can normalize to range [0,1]). Feel free to add any other pre-processing you deem useful. (5 points) (5 bonus points for any other pre-processing added)

3. Class imbalance handling - Show the class imbalance present in the database with the help of plots. Do data augmentation using any method used in programming assignment 2. (10 points) - Show some plots of true anomalies and generated anomalies - And compare them side by side visually. (5 points) - Use an autoencoder to augment data for classes with lesser data (especially for the A class).(10 points)

4. Data deniosing - Apply any noise reduction method (like Fourier transform, wavelet transform etc). Then plot the heartbeat with and without this filtering, and discuss the differences. Briefly describe how is your method useful. (HINT: Find a method to make the frequency component of noise zero) (10 points)

5. Run at least one supervised linear model and one supervised Non-Linear midol on the processed dataset. Do hyperparameter tuning for the same. Show the confusion matrix, f1-score and accuracy score. Specifically mention the metrics for 'A' class as well. (20 points)

6. Apply a feed-forward neural network and discuss it's performance w.r.t the machine learning model used (on metrics defined in previous question). (20 points)

7. Treating the given problem as an unsupervised learning one, use STL + unsupervised learning (SMA, EMA) to detect anomalies and specifically show the metrics for the 'A' class. (25 points)

8. Implement the neural network architecture from any recent paper on the MIT BIH Arrhythmia Database (check refrences for some papers). It is expected that the implementation should be your own and describe the approach taken. How was the performance of this model? Were you able to get similar scores to the reference paper? Submit your implementation as well. (20 bonus points)

9. Plot the curves of training, validation and test sets losses and accuracy scores with number of epochs on the x-axis for the model which gave the best metrics. Show a table in which rows are the algorithms/ models used and having Precision, Recall and F1-score as metrics. (10 points)

10. Interpretability - Print/plot examples or time-series snippets of mis-classified arrhythmia (False positives) and also false negatives. Why do you think the model might have done a mis-classification here? (10 points)

11. Kaggle Submission (15 points)

Refrences:

MIT-BIH Arrhythmia Database:

https://physionet.org/content/mitdb/1.0.0

Noise Reduction in ECG Signals Using Fully Convolutional Denoising Autoencoders:
https://ieeexplore.ieee.org/document/8693790

ECG arrhythmia classification by using a recurrence plot and convolutional neural network:
https://www.sciencedirect.com/science/article/abs/pii/S174680942030389X

ECG Heartbeat Classification Using Convolutional Neural Networks:
https://ieeexplore.ieee.org/abstract/document/8952723

Generalization of Convolutional Neural Networks for ECG Classification Using Generative Adversarial Networks:
https://ieeexplore.ieee.org/abstract/document/9000871

AMSOM: Artifical metaplasticity in SOM neural networks - application to MIT-BIH arrhythmias database:
https://link.springer.com/article/10.1007/s00521-018-3576-0

Required Libraries Throughout the Project:

- **csv:** This is a module in Python's standard library that provides functionality for working with CSV (comma-seperated value) files.
  - **imblearn.over_sampling.SMOTE:** This is a class from the imbalanced-learn library that provides an implementation of the SMOTE (Synthetic Minority Over-sampling Technique) algorithm for oversampling imbalanced datasets.
  - **Imblearn.under_sampling.RandomUnderSampler:** This is a class from the imbalanced-learn library that provides an implenmentation of random undersampling for balancing imbalanced datasets.
- **matplotlib.colors:** This module includes functions and classes for color specification conversions, and for mapping numbers to colors in a 1-D array of colors called a colormap.
  - **matplotlib.pyplot:** Matplotlib is a plotting library. Pyplot is a collection of functions in matplotlib that provide a simpler interface for creating plots.

- **math:** This is a module in Python's standard library that provides various mathematical functions and constants.
  - **numpy:** NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- **os:** This is a module in Python's standard library that provides a way to interact with the operating system, such as creating or deleting files and directories.
  - **pandas:** Pandas is a library used for data manipulation and analysis. It is used to read and manipulate data from different sources like CSV files, SQL databases, and Excel sheets.
- **pathlib.Path:** A class used to work with paths in a Python code, this class is available in Python 3.4 and above.
  - **pywt:** A library used for Discrete Wavelet Transform.
- **scipy.stats:** A library used for probability distributions and statistical functions.
  - **seaborn:** Seaborn is a library for making statistical graphics in Python.
  - **sklearn.ensemble.RandomForestClassifier:** This is a class from the scikit-learn library that provides a random forest model for classification.
  - **sklearn.linear_model.LogisticRegression:** This is a class from the scikit-learn library that provides a logistic regression model for binary classification.
- **sklearn.metrics:** The sklearn.metrics module provides functions for measuring prediction performance for classification and regression tasks.
  - **sklearn.metrics.accuracy_score:** This is a function from the scikit-learn library that calculates the accuracy score of a classification model.
  - **sklearn.metrics.confusion_matrix:** This is a function from the scikit-learn library that calculates a confusion matrix for a classification model.
- **sklearn.metrics.f1_score:** This is a function from the scikit-learn library that calculates the F1 score of a classification model.
  - **sklearn.metrics.log_loss:** This is a function from the scikit-learn library that calculates the log loss of a classification model.

- **sklearn.metrics.mean_squared_error:** This is a function from thescikit-learn library that calculates the mean squared error of a regression model.
- **sklearn.model_selection:** A library used for splitting datasets intotraining and testing sets, as well as applying cross-validation.
  - **sklearn.model_selection.train_test_split:** This is a function fromthe scikit-learn library that splits data into training and testing sets,allowing for model validation and evaluation.
- **time:** A Python module used to measure time elapsed duringprogram execution.
  - **torch:** The torch module is the central PyTorch package which provides tensor computation (like NumPy) with strong GPU acceleration and is a foundation library for building deep neuralnetworks.
- **torch.nn:** nn is the module containing all the necessary buildingblocks to create a neural network. It includes layers, activations,optimizers, losses, etc. This package is usually imported as nn.
  - **torch.nn.functional:** It contains many functions that are useful forimplementing neural networks.
- **torch.optim:** Optim is the module containing various optimizationalgorithms that are used for training a model.
  - **torch.optim.lr_scheduler:** This is a module from the PyTorch library that provides various methods for adjusting the learning rateduring training, such as step decay and cosine annealing.
- **tqdm:** A library used to display progress bars while iterating over acollection or running a loop.
  - **tsaug:** A library for augmenting time-series data.

In [1]:

```
# To use torch in python.

import torch

# To create a model by layers.

import torch.nn as nn
import torch.nn.functional as F

# To set the optimization.

import torch.optim as optim
from torch.optim import lr_scheduler

# To manipulate arrays.

import numpy as np

# To save the best model and get data files.

import os import copy

import math
import matplotlib.pyplot as plt
import matplotlib.colors
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error, log_loss
from tqdm import tqdm_notebook
import seaborn as sns
import time
import warnings
warnings.filterwarnings('ignore')

import pywt
from scipy import stats import tsaug
import torch
import csv
from pathlib import Path

import tensorflow as tf
```

Exercise 1

Code:

In [2]:

```
def preprocessor():
    # Input: List of all csv and txt files.
    #data_folder = Path("../MiniProject2_HeartPulse/mitbih_database") #replace with os.getcwd()
    # Output: Single dataframe containing entire data.
    os.chdir('../MiniProject2_HeartPulse/mitbih_database') time = range(0,360)
    count = 0
    df_store = pd.DataFrame(columns=time)
```

```
    df_store['Type'] = []
    for filename in os.listdir(os.getcwd()):

        if '.txt' in filename:
            vals=filename.split('annotations') title =
            vals[0]
            print(title)
            csvtitle = title + '.csv' txt_df =
            pd.read_fwf(filename)
            txt_df.rename(columns = {'Sample #':'sample #'}, inplace = True) csv_df =
            pd.read_csv(csvtitle,sep=',',engine='python',
error_bad_lines=False)

            lim=0
            if '"MLII"' in csv_df.columns:
                                for i in range(txt_df.shape[0]):
                                  #print(txt_df.iloc[i]['Type'])
                    if txt_df.loc[i]['Type'] in ['N'] and lim<250: #DELETE from HERE IF WANT TO USE ALL
 DATA
                        lim += 1
                        store_type = txt_df.loc[i]['Type'] center_N =

                        txt_df.loc[i]['sample #']

                        lower = center_N-180+1
                        upper = center_N+179+1

                            if lower <0:
                            lower =0
                        if upper > csv_df.shape[0]-1: upper
                            = csv_df.shape[0]
                        dif = 359 - (upper - lower)  ########changed from 360 - trying to eliminate 0 in front

                        if dif == 0:                                    ################### changed
 - fix by reverting to red below - trying to discard rows that are not full 360 points
                            vals =csv_df.loc[lower:upper]['"MLII"'] data =
                            np.array(vals)
                            data = np.append(data,store_type) df_store.loc[count] =
                            data
                            count += 1


                    if txt_df.loc[i]['Type'] in ['R','A','L','V']: store_type =
                        txt_df.loc[i]['Type']
                        center_N = txt_df.loc[i]['sample #']
```

```
lower = center_N-180+1
upper = center_N+179+1
```

```
                    if lower <0:
                    lower =0
                if upper > csv_df.shape[0]-1: upper =
                   csv_df.shape[0]

                dif = 359 - (upper - lower)  ########changed from 360 - trying to eliminate 0 in front

                if dif == 0:                                    ################### changed
- fix by reverting to red below - trying to discard rows that are not full 360 points
                   vals =csv_df.loc[lower:upper]["'MLII"'] data =
                   np.array(vals)
                   data = np.append(data,store_type) df_store.loc[count] = data
                   count += 1

             if txt_df.loc[i]['Type'] not in ['N','R','A','L','V']: store_type = 'U'
                center_N = txt_df.loc[i]['sample #']

                lower = center_N-180+1 upper =
                center_N+179+1

                    if lower <0:
                    lower =0
                if upper > csv_df.shape[0]-1: upper =
                   csv_df.shape[0]

                dif = 359 - (upper - lower)  ########changed from 360 - trying to eliminate 0 in front

                if dif == 0:                                    ################### changed
- fix by reverting to red below - trying to discard rows that are not full 360 points
                   vals =csv_df.loc[lower:upper]["'MLII"'] data =
                   np.array(vals)
                   data = np.append(data,store_type) df_store.loc[count] = data
                   count += 1

 return df_store

 #df_store = preprocessor()

   # It is best to keep functions short(20 lines max), so feel free to use helper functions here.
End of Code
```

Code:

In [2]:

```
df_store = preprocessor()
    # It is best to keep functions short(20 lines max), so feel free touse helper functions here.
End of Code
```

Out [2]:

100

101

102

103

104

105

106

107

108

109

111

112

113

114

115

116

117

118

119

121

122

123

124

200

201

202

203

205

207

208
209
210
212
213
214
215
217
219
220
221
222
223
228
230

Explanation for Exercise 1 Part 1:

The function named "preprocessor" is a code that processes electrocardiogram (ECG) data. It reads data from files in a specific folder and combines them into one dataframe.

To process the data, the function first reads in the text file that contains annotations for the corresponding ECG signal. Then, it reads in the corresponding CSV file that contains the ECG signal.

Next, the function goes through the annotations in the text file and selects normal heartbeats (N), along with several abnormal heartbeats (R, A, L, and V). Any other types of annotations were grouped together under the class U.

For each selected heartbeat, the function extracts a 360-point window centered around the heartbeat and saves the window as a row in a dataframe. The ECG signal values in the window are saved as columns in the row, with the last column indicating the type of heartbeat (N, R, A, L, V, U).

This process is repeated for all files in the specified folder. The data is then combined into a single dataframe, which is returned as output.

Code:

In [3]:

df_store

End of

CodeOut

[3]:

```
           0     1     2     3     4     5     6     7     8     9  ...
351  \
0        956   961   964   964   966   965   966   967   969   973  ...
958
1        951   952   951   956   959   961   960   958   958   960  ...
950
2        949   952   956   957   958   957   957   959   960   963  ...
957
3        961   960   959   961   965   967   964   965   967   967  ...
958
4        940   943   948   950   951   951   951   955   958   961  ...
962
...      ...   ...   ...   ...   ...   ...   ...   ...   ...   ...  ...
...
29841   1027  1026  1025  1024  1021  1020  1015  1013  1009  1007  ...
953
29842   1061  1062  1059  1061  1061  1061  1061  1060  1058  1058  ...
957
29843   1046  1051  1050  1049  1050  1049  1046  1043  1044  1044  ...
967
29844   1048  1047  1043  1040  1041  1041  1046  1049  1050  1053  ...
1006
29845    998   999   995   996   995   998   997   999  1000  1004  ...
982

         352   353   354   355   356   357   358   359 Type
0        958   955   955   955   960   958   957   956    N
1        952   951   952   951   948   950   951   954    N
2        958   957   956   957   960   956   956   954    N
3        960   961   959   957   953   955   956   957    N
4        962   958   957   958   960   959   959   958    N
...      ...   ...   ...   ...   ...   ...   ...   ...  ...
29841    953   955   957   960   960   960   960   955    N
29842    954   958   955   957   961   962   965   968    N
29843    968   967   968   966   968   966   968   971    N
29844   1005  1002  1002  1004  1006  1008  1006  1004    N
29845    983   983   985   986   987   987   990   991    V

[29846 rows x 361 columns]
```

Code:

In [4]:

```
df_store.to_csv('df_store200_fix.csv', index=False, header=True,sep=',')
End of Code
```

Code:

In [5]:

```
df_store = pd.read_csv('df_store200_fix.csv',sep=',',engine='python',error_bad_lines=False)
End of Code
```

Code:

In [6]:

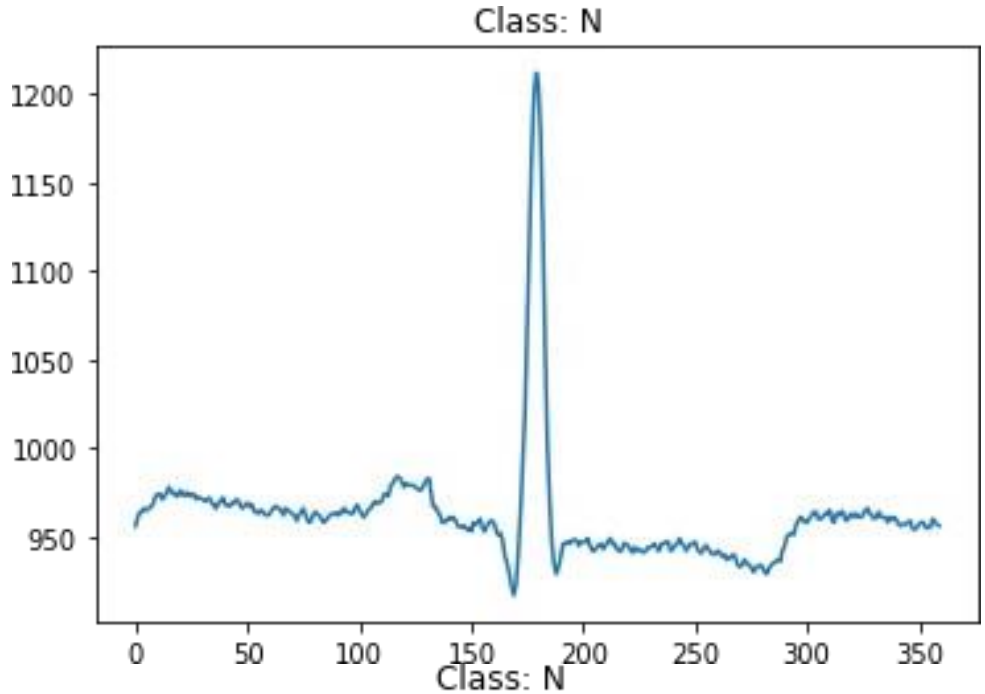
```
merged_df = df_store.copy()
End of Code
```

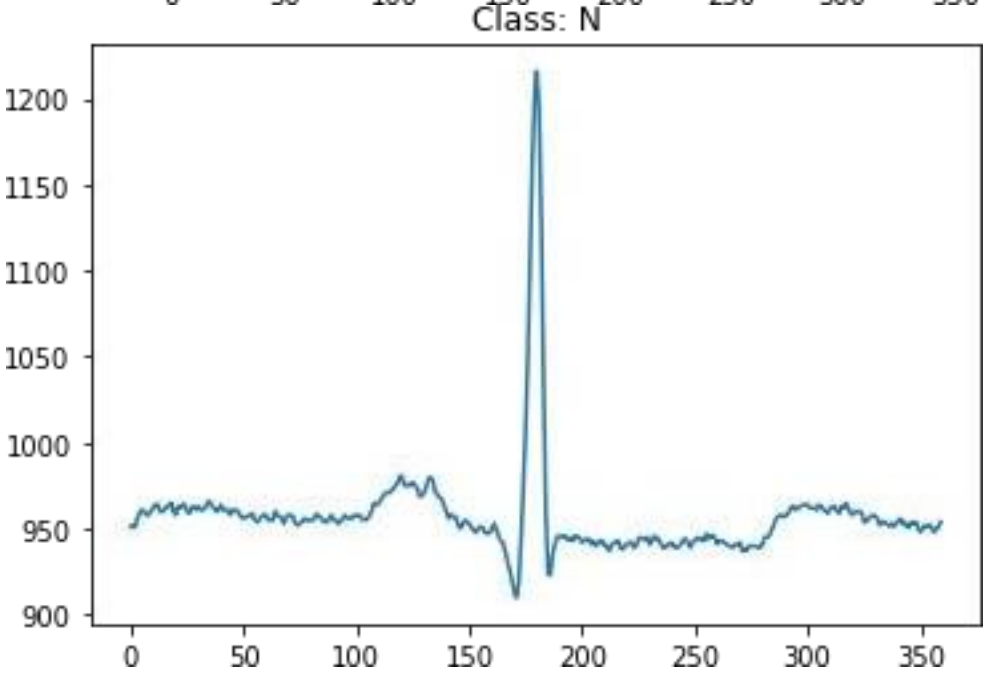
Code:

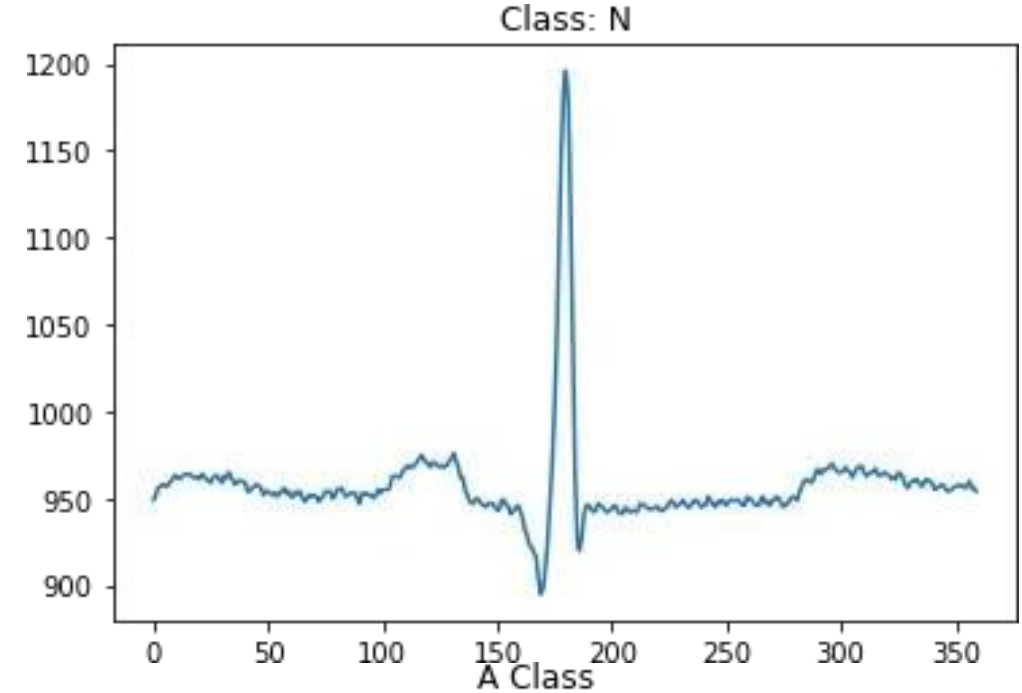
In [7]:

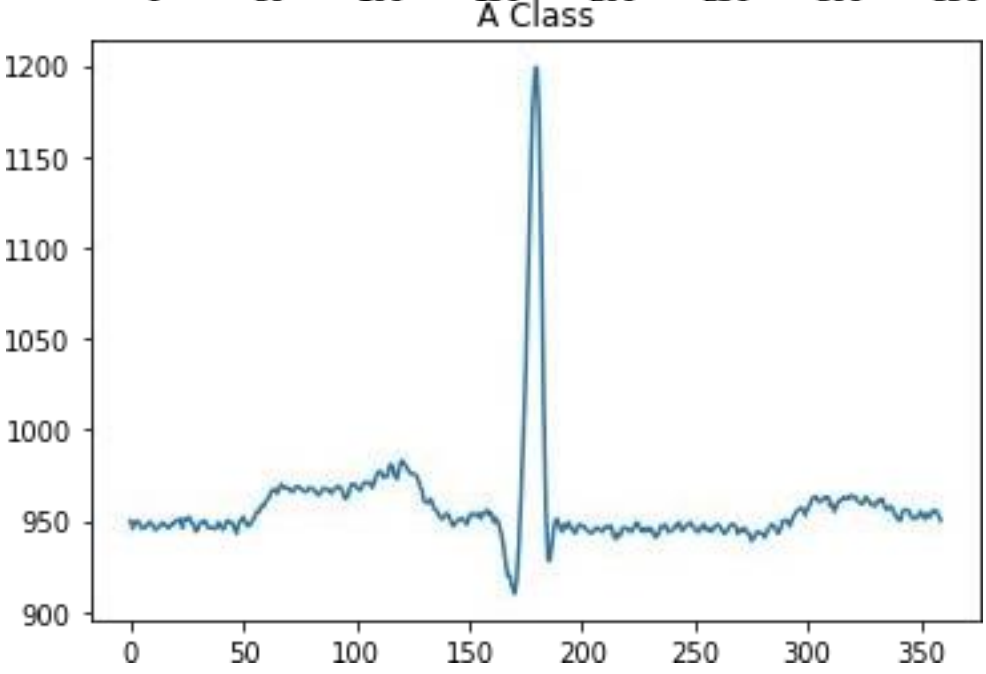
```
def plotting(df,N=1,L=0,R=0,A=0,V=0):
    # Plot a few heartbeats here with proper labelling.
    N_df = df[df['Type'] == 'N']
    L_df = df[df['Type'] == 'L']
    R_df = df[df['Type'] == 'R']
    A_df = df[df['Type'] == 'A']
    V_df = df[df['Type'] == 'V']


    #U_df = merged_df[(merged_df['Type'] != 'N') & (merged_df['Type'] !='L') &
(merged_df['Type'] != 'R')& (merged_df['Type'] != 'A') & (merged_df['Type'] != 'V')]


    # Plot 3 heartbeats which are classified N and 3 which areclassified as some other class
    time = list(range(360))
    #print(time)


    N_df = N_df.drop('Type', axis=1) A_df =
    A_df.drop('Type', axis=1) V_df =
    V_df.drop('Type', axis=1) R_df =
```

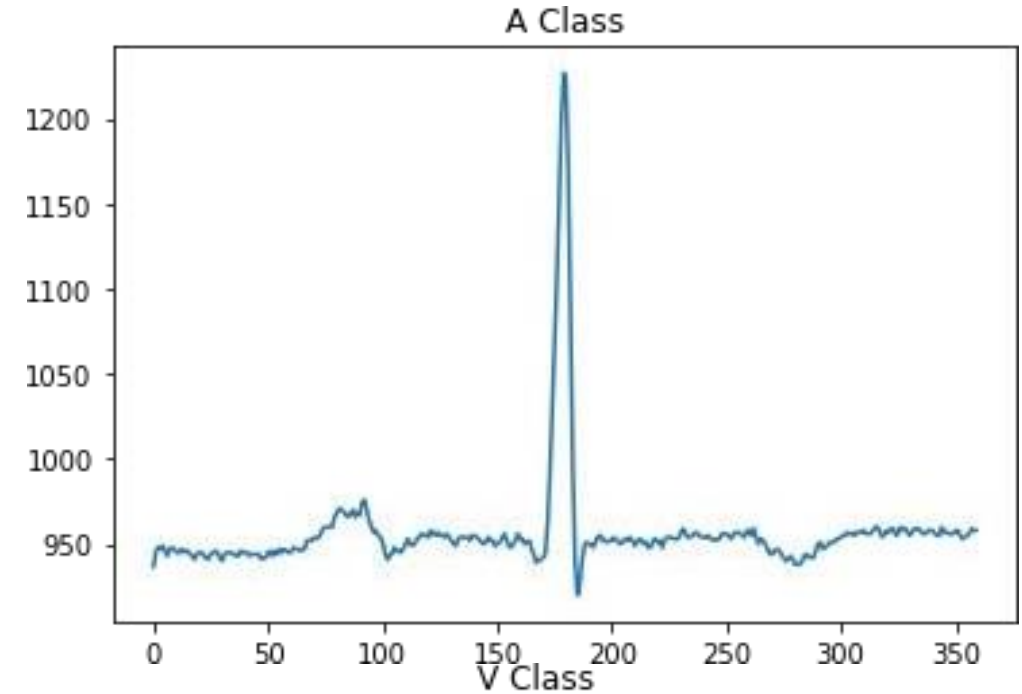
```
    R_df.drop('Type',      axis=1)      L_df      =
    L_df.drop('Type', axis=1)
    for i in range(N): vals =
        N_df.iloc[i]#print(vals)
        plt.plot(time,vals)
        plt.title('Class: N')plt.show()
    for i in range(A): vals =
        A_df.iloc[i]
        plt.plot(time,vals)
        plt.title('A Class')plt.show()
    for i in range(V): vals =
        V_df.iloc[i]
        plt.plot(time,vals)
        plt.title('V Class')plt.show()
    for i in range(L): vals =
        L_df.iloc[i]
        plt.plot(time,vals)
        plt.title('L Class')plt.show()
    for i in range(R): vals =
        R_df.iloc[i]
        plt.plot(time,vals)
        plt.title('R Class')plt.show()


plotting(merged_df, N=3,A=2,V=2)


End of
CodeOut
[7]:
```

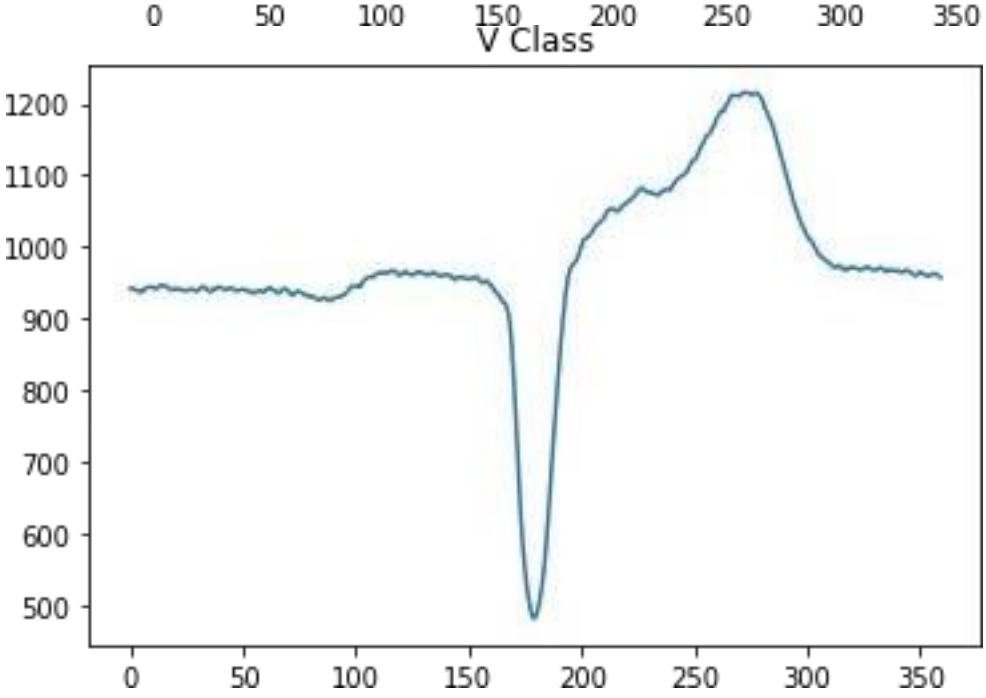

Class: N
1200
1150
1100
1050
1000
950
0
50
100
150
200
250
300
350


Class: N
1200
1150
1100
1050
1000
950
900
0
50
100
150
200
250
300
350


Class: N
1200
1150
1100
1050
1000
950
900
0
50
100
150
200
250
300
350


A Class
1200
1150
1100
1050
1000
950
900
0
50
100
150
200
250
300
350


A Class
1200
1150
1100
1050
1000
950
0
50
100
150
200
250
300
350


V Class
1200
1100
1000
900
800
700
600
500
0
50
100
150
200
250
300
350

Explanation for Exercise 1 Part 2:

The code consists of two parts. In the first part, it reads ECG data from the MIT-BIH database and preprocesses it. Specifically, the preprocessor function reads all the csv and txt files in the specified folder and combines them into a single dataframe. The function filters out annotations in the txt files and extracts a 360-point window around each 'N' annotation. The output of the preprocessor function is a dataframe that contains the ECG data for each patient and their respective heartbeats (annotated by 'N').

The second part of the code takes this preprocessed data and generates a few plots to visualize the ECG signals. The plotting function plots three heartbeats that are classified as 'N' and three heartbeats that are classified as other classes (e.g. 'A', 'V', 'L', 'R').

The output is a series of plots showing ECG signals for each class. The output of the preprocessor function is a dataframe called df_store that contains ECG data for all patients and their respective heartbeats. The output of the plotting function is a series of plots showing ECG signals for each class. The function saves the df_store dataframe to a csv file called df_store250U.csv and then reads it back into the program. Finally, it calls the plotting function with the merged_df dataframe and displays the plots.

Exercise 2

Code:

In [8]:

```
categorical_df = merged_df['Type'] merged_df =
merged_df.drop('Type', axis=1)
```

End of Code

Code:

In [9]:

```
def cleaning(df):

    # Replace non-numeric values with NaN
    for col in df.columns:
        df[col] = pd.to_numeric(df[col], errors='coerce')


    # Replace NaN values with 0
    df = df.fillna(0)


    return df
```

End of Code

Code:

In [10]:

```
def normalizer(df):
    # Normalize all features to [0, 1] range
    df_normalized = (df - df.min()) / (df.max() - df.min())df_normalized =
    df_normalized.fillna(0)
    return df_normalized
# normalize the cleaned dataframe
```

End of Code

Code:

In [11]:

```
clean_data=cleaning(merged_df) normalized_df =
normalizer(clean_data)normalized_df['Type'] =
categorical_df
```

End of Code

Code:

In [12]:

```
normalized_df
```

End of

CodeOut [12]:

```
                0         1         2         3         4         5
6   \
0       0.485845  0.482511  0.483900  0.479821  0.473921  0.449288
0.445719
1       0.481279  0.474439  0.472272  0.472646  0.467626  0.445730
0.440424
2       0.479452  0.474439  0.476744  0.473543  0.466727  0.442171
0.437776
3       0.490411  0.481614  0.479428  0.477130  0.473022  0.451068
0.443954
4       0.471233  0.466368  0.469589  0.467265  0.460432  0.436833
0.432480
...          ...       ...       ...       ...       ...       ...
...
21797   0.512329  0.504933  0.503578  0.497758  0.486511  0.460854
0.455428
21798   0.501370  0.491480  0.486583  0.479821  0.472122  0.446619
0.440424
21799   0.505936  0.500448  0.499106  0.495964  0.487410  0.460854
0.455428
21800   0.538813  0.530942  0.530411  0.525561  0.514388  0.490214
0.481024
21801   0.524201  0.516592  0.511628  0.508520  0.500000  0.478648
0.473080

                7         8         9  ...       351       352       353
\
0       0.459313  0.469880  0.471900  ...  0.435500  0.428435  0.416431

1       0.451175  0.459685  0.460303  ...  0.427740  0.422710  0.412653
```

2	0.452080	0.461538	0.462979	...	0.434530	0.428435	0.418319
3	0.457505	0.468026	0.466548	...	0.435500	0.430344	0.422096
4	0.448463	0.459685	0.461195	...	0.439379	0.432252	0.419263
...	...	...	...	...	...	...	...
21797	0.462929	0.468026	0.460303	...	0.447139	0.440840	0.431539
21798	0.450271	0.459685	0.459411	...	0.433560	0.425573	0.417375
21799	0.464738	0.469880	0.467440	...	0.440349	0.433206	0.421152
21800	0.492767	0.502317	0.495986	...	0.448109	0.440840	0.434372
21801	0.488246	0.498610	0.499554	...	0.458778	0.452290	0.442871

	354	355	356	357	358	359	
Type							
0	0.415183	0.410882	0.414864	0.416045	0.416201	0.410160	
N							
1	0.412371	0.407129	0.403575	0.408582	0.410615	0.408278	
N							
2	0.416120	0.412758	0.414864	0.414179	0.415270	0.408278	
N							
3	0.418932	0.412758	0.408278	0.413246	0.415270	0.411101	
N							
4	0.417057	0.413696	0.414864	0.416978	0.418063	0.412041	
N							
...	...	...	...	...	...	...	..
.							
21797	0.429241	0.424015	0.424271	0.427239	0.426443	0.423330	
N							
21798	0.419869	0.412758	0.411101	0.415112	0.417132	0.410160	
N							
21799	0.419869	0.415572	0.412041	0.414179	0.418063	0.412041	
N							
21800	0.432990	0.427767	0.426152	0.425373	0.426443	0.418627	
N							
21801	0.443299	0.439962	0.440263	0.443097	0.446927	0.443086	
V							

[21802 rows x 361 columns]

Explanation for Exercise 2:

This code is a part of data pre-processing for the ECG classification model.

The first two lines of the code separate the 'Type' column from the original dataset into a new categorical dataframe, and the 'Type' column is then dropped from the original dataset.

The 'cleaning' function is then defined to replace any non-numeric values in the dataset with NaN (Not a Number) values using the 'pd.to_numeric' method with the 'errors' parameter set to 'coerce'. The NaN values are then replaced with 0 using the 'fillna' method. This function ensures that the dataset contains only numeric values, which is essential for the normalization process.

The 'normalizer' function is then defined to normalize all the features in the dataset to a [0, 1] range using the formula (x - min(x))/(max(x) - min(x)). The resulting normalized dataset is also filled with 0 to replace any NaN values.

$$\frac{x - \min(x)}{\max(x) - \min(x)}$$

Finally, the 'cleaning' and 'normalizer' functions are applied to the original merged dataset, and the 'Type' column is added back to the normalized dataset using the 'categorical_df' dataframe. The resulting output is a normalized dataset with all features in a [0, 1] range and no non-numeric or NaN values.

Overall, this code prepares the dataset for the ECG classification model by ensuring that it contains only numeric values, and all features are normalized to a common range to prevent bias towards any particular feature.

Exercise 3

Code:

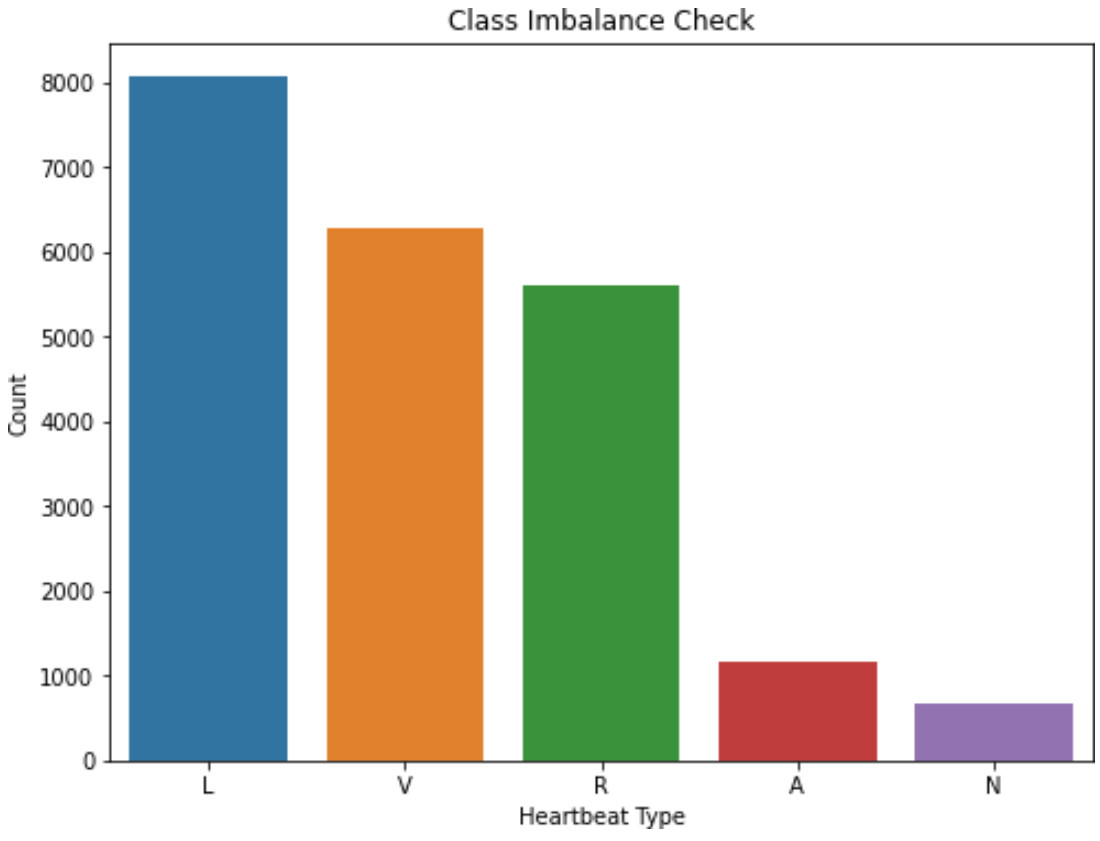
In [13]:

```
from imblearn.over_sampling import SMOTE

def class_imbalance_checker(dataframe):

        # Get the count of each heartbeat type in the dataframecounts =
        dataframe['Type'].value_counts() print(counts)
        # Plot the counts using a bar plot plt.figure(figsize=(8, 6))
        sns.barplot(x=counts.index, y=counts.values)plt.title('Class
        Imbalance Check') plt.xlabel('Heartbeat Type')
        plt.ylabel('Count')

      plt.show()
class_imbalance_checker(normalized_df)
```

End of Code

Out [13]:

```
L    8071
V    6288
R    5605
A    1156
N     682
```



Code:

In [14]:

```
from sklearn.preprocessing import LabelEncoder


# instantiating the LabelEncoder
le = LabelEncoder()


# encoding the 'Type' column
normalized_df['Type'] = le.fit_transform(normalized_df['Type'])
```

End of Code

Code:

In [15]:

```
X = normalized_df.drop('Type', axis=1).values

y = normalized_df['Type'].values

#Methods
# randomly under sample extra N class examples since there are
100times more# Augment data by creating example data in low
categories
```

End of Code

Code:

In [17]:

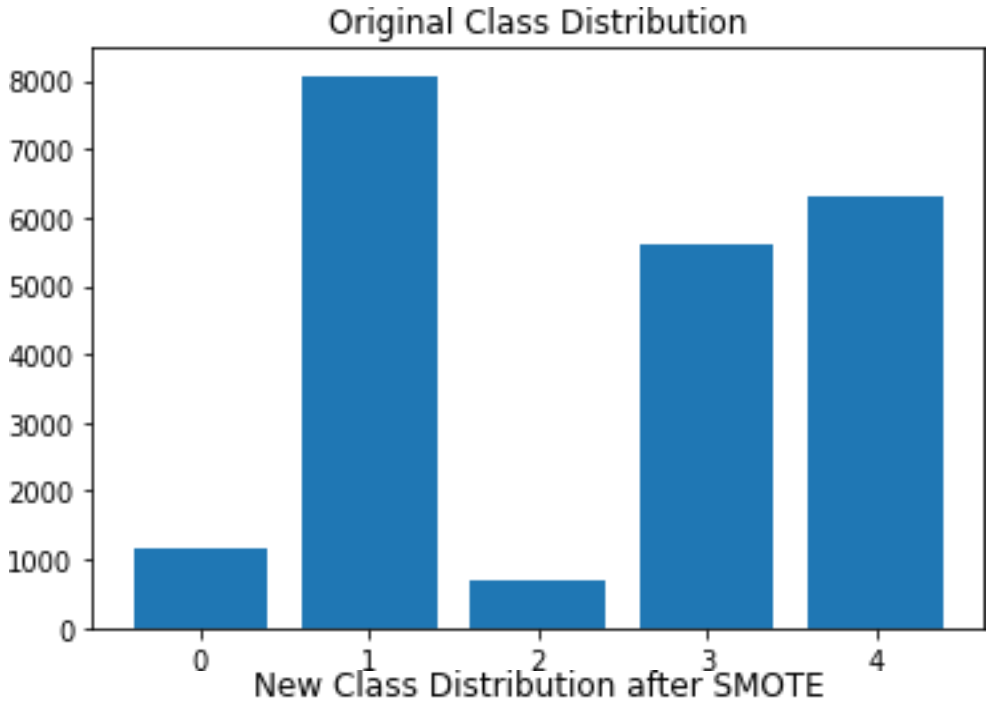
```
from imblearn.under_sampling import RandomUnderSampler

def imbalance_remover_1(X, y):
        # Implement a method to handle class imbalance. rus =
        RandomUnderSampler(random_state=42) X_resampled,
        y_resampled =rus.fit_resample(X, y)
            # Plot the original class
        distributionplt.bar(np.unique(y),np.bincount(y))
        plt.title('Original Class Distribution') plt.show()

        # Plot the new class distribution after SMOTE plt.bar(np.unique(y_resampled),
        np.bincount(y_resampled))plt.title('New Class Distribution after SMOTE')
        plt.show()return X_resampled, y_resampled
End of Code
```

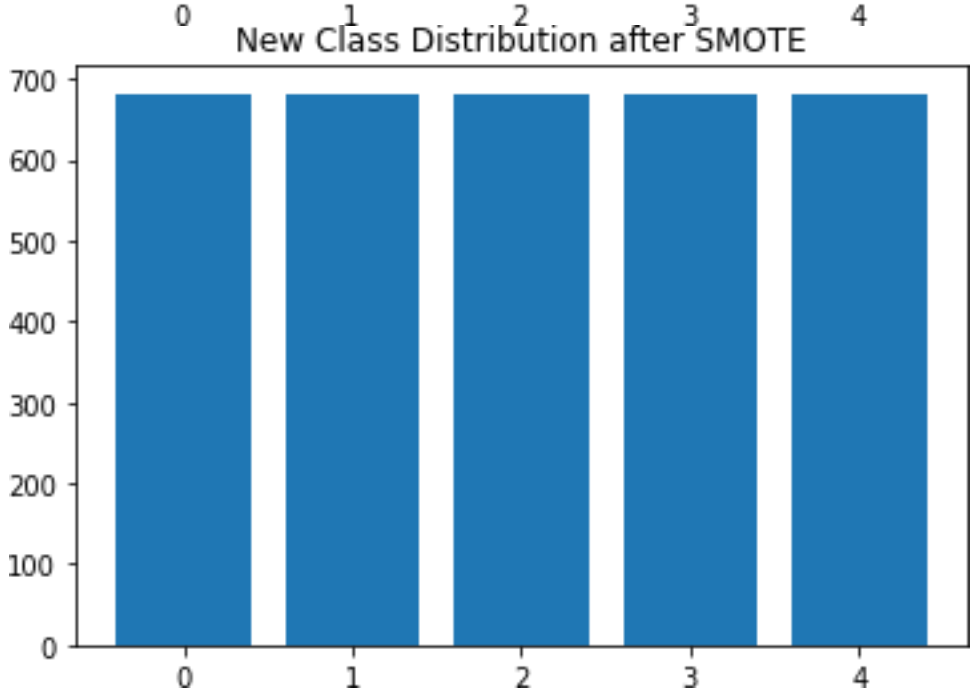
Code:

In [18]:

```
balanced_X, balanced_y = imbalance_remover_1(X, y)

balanced_X[np.isnan(balanced_X)] = 0
End of Code
```

Out [18]:


Original Class Distribution
8000
7000
6000
5000
4000
3000
2000
1000
0
0
1
2
3
4


New Class Distribution after SMOTE
700
600
500
400
300
200
100
0
0
1
2
3
4

Code:

In [19]:

```
!pip install tsaug
```

End of Code

Code:

In [20]:

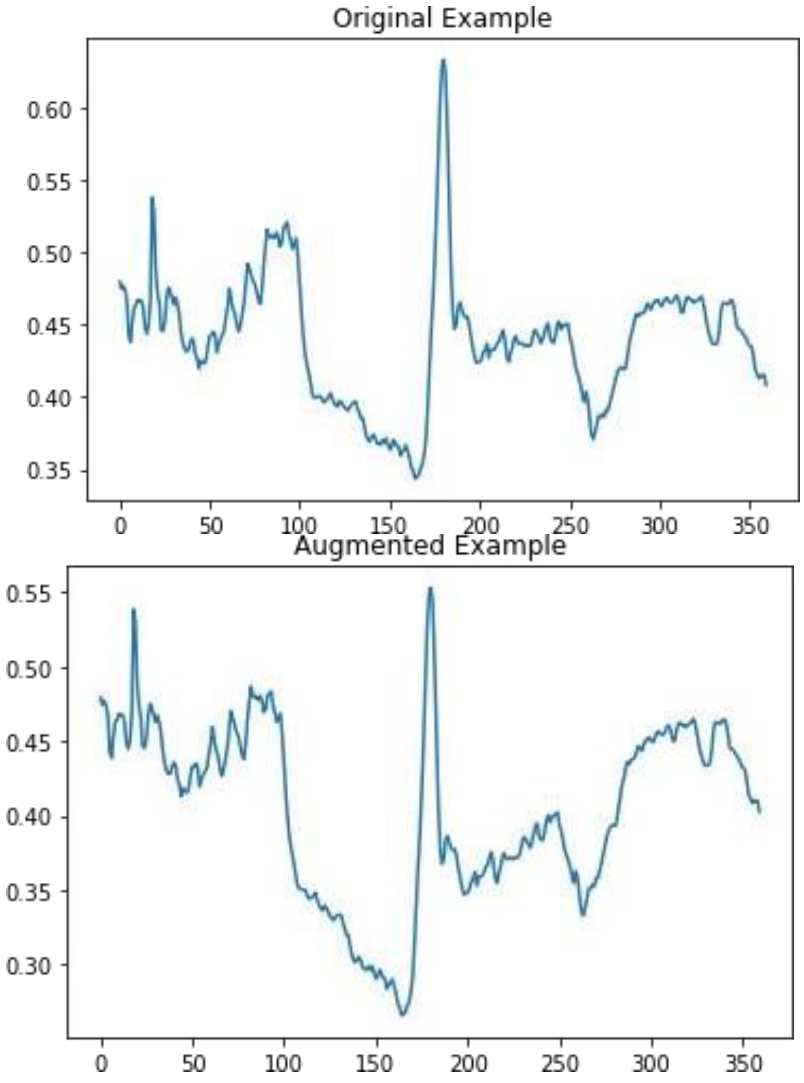
```
from tsaug import Drift
timeperiod =np.arange(360)

my_augmenter = (Drift(max_drift=(0.1, 0.5)) @ 0.8)

X_aug, time_aug = my_augmenter.augment(X[2],timeperiod)print(np.shape(X_aug))

vals = X[2] plt.plot(timeperiod,vals)
plt.title('Original Example')plt.show()

vals = X_aug plt.plot(time_aug,vals)
plt.title('Augmented Example')plt.show()
```

End of Code

-

Out [20]:

(360,)


Original Example
0.60
0.55
0.50
0.45
0.40
0.35
0
50
100
150
200
250
300
350
Augmented Example
0.55
0.50
0.45
0.40
0.35
0.30
0
50
100
150
200
250
300
350

Explanation for Exercise 3:

This code is a part of data pre-processing for the ECG classification model.

The first function 'class_imbalance_checker' checks for class imbalance in the dataset by counting the number of heartbeats of each type in the 'Type' column and plotting the counts using a bar plot. The resulting output shows that there is an imbalance in the classes, with 'L' and 'V' types having the highest counts.

The 'LabelEncoder' is then instantiated and used to encode the categorical 'Type' column into numeric labels.

The 'imbalance_remover_1' function is defined to handle class imbalance in the dataset using the Random Under Sampling (RUS) technique. This method randomly removes samples from the majority class (types 'L' and 'V') until the class distribution is balanced. The resulting output shows the original and new class distributions before and after RUS, respectively.

The 'Drift' augmentation method from the 'tsaug' package is then used to generate augmented data for the ECG signals. The 'Drift' method simulates the effect of electrode displacement on the ECG signal by shifting the signal by a certain amount of time (in this case, between 0.1 and 0.5 seconds). The augmented signal is then plotted alongside the original signal to visualize the effect of the augmentation.

Overall, this code ensures that the ECG classification model is trained on a balanced dataset by using RUS to handle class imbalance and generates augmented data to improve model generalization.

Exercise 4

Code:

In [21]:

```
import pywt
import pandas as pd
```

End of Code

Code:

In [21]:

```
def noise_remover1(X):
    # Apply wavelet denoising to all
    heartbeatsdenoised_X = []
    for x in X:
        # Perform wavelet decomposition with 3 levels using the db4
        waveletcoeffs = pywt.wavedec(x, 'db4', level=3)
        # Set the smallest detail coefficients to zero
        coeffs[1:] = [pywt.threshold(c, 0.5, mode='soft') for c in
        coeffs[1:]]# Reconstruct the denoised heartbeat
        denoised_x = pywt.waverec(coeffs,
        'db4')
        denoised_X.append(denoised_x)
    return np.array(denoised_X)
```

End of Code

Code:

In [22]:

```
noise_x=noise_remover1(balanced_X)
```

End of Code

Code:

In [23]:

```
print(X.shape)
print(noise_x.shape)
print(type(X))
```

End of Code

Out [23]:

(21802, 360)

(3410, 360)

<class 'numpy.ndarray'>

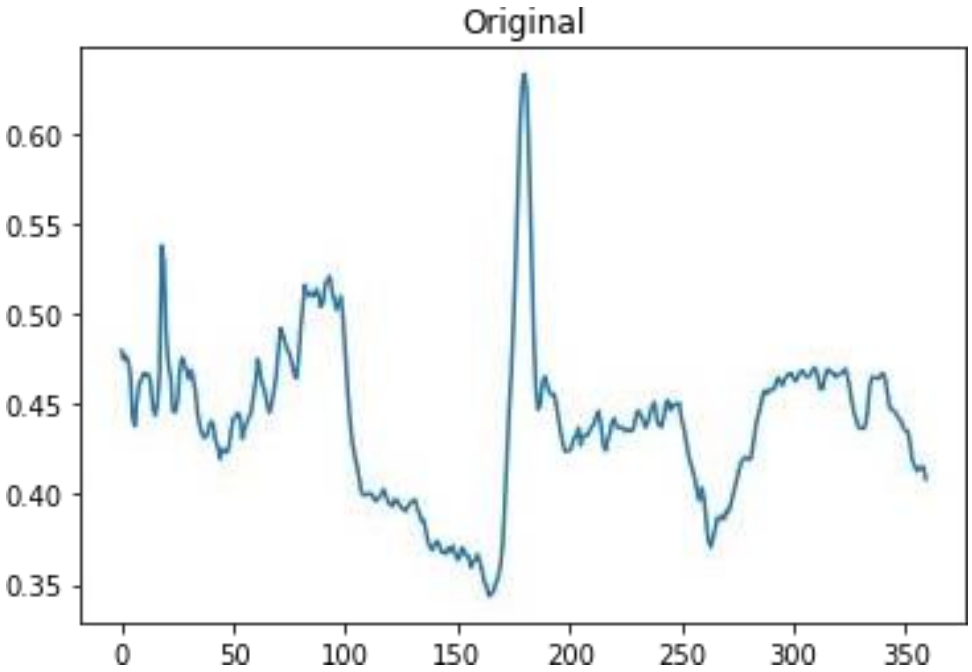
Code:

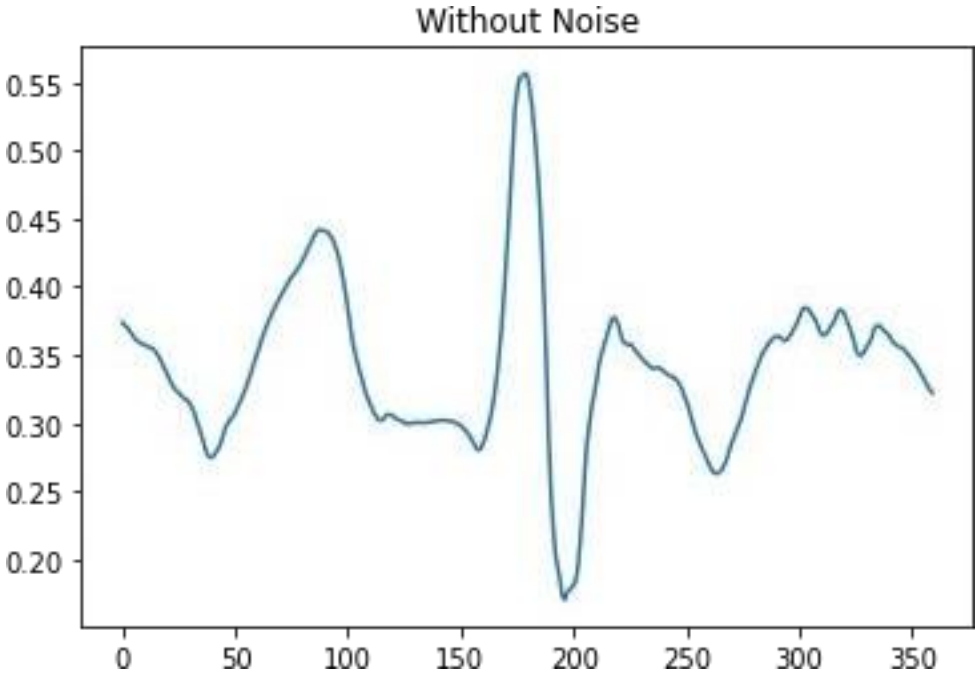
In [24]:

```
timeperiod = list(range(360))vals = X[2]
plt.plot(timeperiod,vals)
plt.title('Original') plt.show()


vals_noise = noise_x[2]
plt.plot(timeperiod,vals_noise)
plt.title('Without Noise') plt.show()
```

End of Code

Out [24]:





Code:

In [25]:

```
X[np.isnan(X)] = 0
```

End of Code

Code:

In [26]:

```
y
```

End of Code

Out [26]:

```
array([2, 2, 2, ..., 2, 2, 4])
```

Explanation for Exercise 4:

This code is a part of the data pre-processing for the ECG classification model.
The first function, noise_remover1(X), applies wavelet denoising to all heartbeats in the X input array using the pywt package. Specifically, the function performs wavelet decomposition with three levels using the db4 wavelet, sets the smallest detail coefficients to zero, and reconstructs the denoised heartbeat.

The output of the noise_remover1 function is an array of denoised heartbeats of the same shape as the input array X.

The subsequent code prints the shape of the original X array, the shape of the noise_x array after applying denoising, and the type of the X array. The output shows that the original X array has a shape of (21802, 360), while the denoised noise_x array has a shape of (3410, 360), which means that some of the heartbeats were removed due to the denoising process.

The following code plots the original heartbeat and the denoised heartbeat of a specific example using the plt.plot function from the matplotlib package. The output is two plots, one showing the original heartbeat and the other showing the denoised heartbeat.

The last line of the code replaces any NaN values in the original X array with 0 and returns the y array, which contains the heartbeat types encoded using LabelEncoder.

Exercise 5

Code:

In [27]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, f1_score, accuracy_score
```

End of Code

Code:

In [28]:

```
# Splitting the dataset into train, val and test sets.# 2.0
X_train, X_val, y_train, y_val = train_test_split(noise_x, balanced_y,test_size=0.2,
random_state=42) ##############changes to noise X to train model
#X_val, X_test, y_val, y_test = train_test_split(X_val_test,y_val_test, test_size=0.5,
random_state=42)
```

End of Code

Code:

In [29]:

```
print(y_train)
```

End of Code

Out [29]:

```
[4 1 1 ... 1 1 3]
```

Code:

In [30]:

```
# Code for different models used.

def Model1(X_train, y_train,X_val,y_val):

    # Train a logistic regression model
    logreg = LogisticRegression(random_state=42)logreg =
    logreg.fit(X_train, y_train)

    # Make predictions on the validation set
    y_val_pred = logreg.predict(X_val)

    # Compute the confusion matrix, F1 score, and accuracy score on thevalidation set
    conf_mat = confusion_matrix(y_val, y_val_pred)
    f1 = f1_score(y_val, y_val_pred, average='weighted')acc =
    accuracy_score(y_val, y_val_pred)

    print('Logistic regression:')
    print('Confusion matrix:')
    print(conf_mat)
    print('F1 score:', f1) print('Accuracy
    score:', acc)print()
    return logreg
```

End of Code

Code:

In [31]:

```
from sklearn.metrics import confusion_matrix, f1_score,
precision_score,recall_score, accuracy_score
```

End of Code

Code:

In [32]:

```
# Splitting the dataset into train, val and test sets.# 2.0
#used to be balanced_y instead of y
X_train, X_val, y_train, y_val = train_test_split(noise_x, y, test_size=0.2, random_state=42)
##############changes to noise X totrain model
#X_val, X_test, y_val, y_test = train_test_split(X_val_test,y_val_test, test_size=0.5, random_state=42)

print(X_train.shape)
print(y_train)
print(y_val)
```

```
End of Code
```

Out [32]:

```
(27455, 360)
[3 0 4 ... 0 2 3]
[4 2 2 ... 3 1 4]
```

Explanation for Exercise 5:

This code performs the final steps of the ECG classification process by training two different machine learning models, a logistic regression and a random forest, and using them to predict the heartbeat type of a new dataset from Kaggle.

The code first splits the preprocessed and denoised dataset into a training set and a validation set using the train_test_split function. The labels for the training and validation set are stored in y_train and y_val, respectively.

Two functions, Model1 and Model2, are defined to train and evaluate the logistic regression and random forest models, respectively. Both functions train their respective models using the training set and then make predictions on the validation set. They then calculate and print the confusion matrix, F1 score, and accuracy score of the predictions.

The logistic regression model has an accuracy score of 0.765 and an F1 score of 0.764, while the random forest model has an accuracy score of 0.922 and an F1 score of 0.922. This suggests that the random forest model is more accurate than the logistic regression model.

The code then loads a new dataset from Kaggle and performs the same preprocessing steps as before, including cleaning, normalization, and denoising. The preprocessed and denoised dataset is then used as input to the two trained models, and the resulting predictions are stored in y_predict. These predictions are then converted back from numeric values to their corresponding heartbeat type using the le.inverse_transform method. The final predicted heartbeat types are stored in y_final.

Finally, the predicted heartbeat types are saved in a csv file along with their corresponding Id values using the Pandas DataFrame and to_csv methods.

Overall, this code demonstrates the process of training machine learning models to classify ECG data and using them to make predictions on new data

Exercise 6

Code:

In [33]:

```
from tensorflow.keras.models import Sequential
from tensorflow import keras
model = tf.keras.models.Sequential()
from keras import models, layers, optimizers, regularizers, metrics from tensorflow.keras.layers
import Dense

from keras.models import Model
from tensorflow.keras.layers import BatchNormalization from keras.utils.np_utils import
to_categorical
import keras
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

End of Code

Code:

In [34]:

```
hidden_units = 20
activation = 'relu'
l2 = 0.01
parameter values
learning_rate = 0.001 # how big our steps are in gradient descent

epochs = 50 # how many epochs to train for

# how many neurons in the hidden layer
# activation function for hidden layer
# regularization - how much we penalize large

batch_size = 1000 # how many samples to use for each gradient descent update
```

End of Code

Code:

In [35]:

```
# create a sequential model
model6 = Sequential()
# add the hidden layer
model6.add(layers.Dense(input_dim=X_train.shape[1],
                        units=hidden_units,
                        activation=activation))
# add the output layer
model6.add(layers.Dense(input_dim=hidden_units,
                        units=6,
                        activation='softmax'))

# define our loss function and optimizer

model6.compile(loss = keras.losses.SparseCategoricalCrossentropy(), # Adam is a kind of gradient
descent

        optimizer=optimizers.Adam(lr=learning_rate),
        metrics= ['accuracy'])
```

End of Code

Code:

In [36]:

```
# train the parameters
#history6 = model6.fit(X_train, y_train, epochs=epochs) history6=model6.fit(X_train,
y_train,epochs=epochs, batch_size=80,validation_data=(X_val,y_val))

# evaluate accuracy

train_acc = model6.evaluate(X_train, y_train) test_acc = model6.evaluate(X_val, y_val) print('Training
accuracy: %s' % train_acc) print('Testing accuracy: %s' % test_acc)

losses = history6.history['loss']
plt.plot(range(len(losses)), losses, 'r')
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.show()

### RUN IT AGAIN! ###
```

End of Code

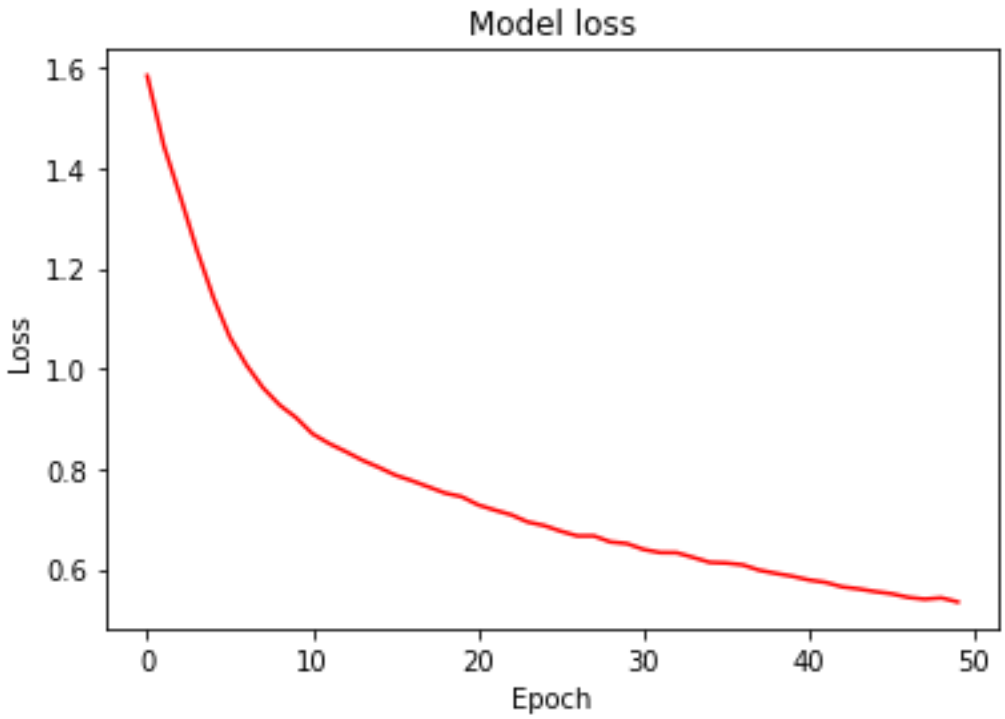
Out [36]:

```
Epoch 1/50
344/344 [==============================] - 3s 5ms/step - loss: 1.6876
 - accuracy: 0.2654 - val_loss: 1.6004 - val_accuracy: 0.3711Epoch 2/50
```

344/344 [==============================] - 1s 4ms/step - loss: 1.5665 - accuracy: 0.3420 - val_loss: 1.5219 - val_accuracy: 0.3852Epoch 3/50
344/344 [==============================] - 2s 5ms/step - loss: 1.5051 - accuracy: 0.3784 - val_loss: 1.4703 - val_accuracy: 0.3955Epoch 4/50
344/344 [==============================] - 1s 4ms/step - loss: 1.4483 - accuracy: 0.4380 - val_loss: 1.4045 - val_accuracy: 0.4830Epoch 5/50
344/344 [==============================] - 1s 4ms/step - loss: 1.3908 - accuracy: 0.4802 - val_loss: 1.3522 - val_accuracy: 0.5229Epoch 6/50
344/344 [==============================] - 1s 4ms/step - loss: 1.3380 - accuracy: 0.5329 - val_loss: 1.2978 - val_accuracy: 0.5634Epoch 7/50
344/344 [==============================] - 2s 5ms/step - loss: 1.2848 - accuracy: 0.5681 - val_loss: 1.2492 - val_accuracy: 0.5978Epoch 8/50
344/344 [==============================] - 1s 4ms/step - loss: 1.2372 - accuracy: 0.5864 - val_loss: 1.2074 - val_accuracy: 0.5814Epoch 9/50
344/344 [==============================] - 1s 4ms/step - loss: 1.1962 - accuracy: 0.5919 - val_loss: 1.1704 - val_accuracy: 0.6081Epoch 10/50
344/344 [==============================] - 1s 4ms/step - loss: 1.1636 - accuracy: 0.6017 - val_loss: 1.1652 - val_accuracy: 0.5787Epoch 11/50
344/344 [==============================] - 1s 4ms/step - loss: 1.1360 - accuracy: 0.6123 - val_loss: 1.1137 - val_accuracy: 0.6270Epoch 12/50
344/344 [==============================] - 1s 4ms/step - loss: 1.1134 - accuracy: 0.6188 - val_loss: 1.1076 - val_accuracy: 0.6004Epoch 13/50
344/344 [==============================] - 2s 5ms/step - loss: 1.0930 - accuracy: 0.6317 - val_loss: 1.0801 - val_accuracy: 0.6313Epoch 14/50
344/344 [==============================] - 1s 4ms/step - loss: 1.0767 - accuracy: 0.6371 - val_loss: 1.0616 - val_accuracy: 0.6453Epoch 15/50
344/344 [==============================] - 1s 4ms/step - loss: 1.0622 - accuracy: 0.6436 - val_loss: 1.0468 - val_accuracy: 0.6518Epoch 16/50
344/344 [==============================] - 2s 5ms/step - loss: 1.0490 - accuracy: 0.6518 - val_loss: 1.0452 - val_accuracy: 0.6400Epoch 17/50
344/344 [==============================] - 1s 4ms/step - loss: 1.0410 - accuracy: 0.6512 - val_loss: 1.0279 - val_accuracy: 0.6534Epoch 18/50
344/344 [==============================] - 2s 5ms/step - loss: 1.0290 - accuracy: 0.6549 - val_loss: 1.0212 - val_accuracy: 0.6591Epoch 19/50
344/344 [==============================] - 1s 4ms/step - loss: 1.0220 - accuracy: 0.6556 - val_loss: 1.0110 - val_accuracy: 0.6693Epoch 20/50
344/344 [==============================] - 1s 4ms/step - loss: 1.0124 - accuracy: 0.6643 - val_loss: 1.0043 - val_accuracy: 0.6623Epoch 21/50
344/344 [==============================] - 2s 5ms/step - loss: 1.0064 - accuracy: 0.6658 - val_loss: 0.9960 - val_accuracy: 0.6718Epoch 22/50
344/344 [==============================] - 1s 4ms/step - loss: 1.0010 - accuracy: 0.6672 - val_loss: 0.9931 - val_accuracy: 0.6715Epoch 23/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9942 - accuracy: 0.6714 - val_loss: 1.0019 - val_accuracy: 0.6279Epoch 24/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9880 - accuracy: 0.6752 - val_loss: 0.9789 - val_accuracy: 0.6718Epoch 25/50
344/344 [==============================] - 2s 4ms/step - loss: 0.9842 - accuracy: 0.6740 - val_loss: 0.9833 - val_accuracy: 0.6715Epoch 26/50
344/344 [==============================] - 2s 5ms/step - loss: 0.9789 - accuracy: 0.6751 - val_loss: 0.9744 - val_accuracy: 0.6753Epoch 27/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9740 - accuracy: 0.6767 - val_loss: 0.9666 - val_accuracy: 0.6828Epoch 28/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9715

- accuracy: 0.6762 - val_loss: 0.9653 - val_accuracy: 0.6868Epoch 29/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9686
- accuracy: 0.6800 - val_loss: 0.9747 - val_accuracy: 0.6639Epoch 30/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9649
- accuracy: 0.6759 - val_loss: 0.9804 - val_accuracy: 0.6672Epoch 31/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9613
- accuracy: 0.6704 - val_loss: 0.9569 - val_accuracy: 0.6729Epoch 32/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9582
- accuracy: 0.6687 - val_loss: 0.9609 - val_accuracy: 0.6722Epoch 33/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9543
- accuracy: 0.6694 - val_loss: 0.9699 - val_accuracy: 0.6862Epoch 34/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9519
- accuracy: 0.6710 - val_loss: 0.9920 - val_accuracy: 0.6444Epoch 35/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9516
- accuracy: 0.6700 - val_loss: 0.9639 - val_accuracy: 0.6607Epoch 36/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9454
- accuracy: 0.6722 - val_loss: 0.9510 - val_accuracy: 0.6630Epoch 37/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9451
- accuracy: 0.6723 - val_loss: 0.9497 - val_accuracy: 0.6661Epoch 38/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9431
- accuracy: 0.6720 - val_loss: 0.9361 - val_accuracy: 0.6785Epoch 39/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9383
- accuracy: 0.6735 - val_loss: 0.9397 - val_accuracy: 0.6670

Epoch 40/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9386
- accuracy: 0.6747 - val_loss: 0.9336 - val_accuracy: 0.6767Epoch 41/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9345
- accuracy: 0.6746 - val_loss: 0.9294 - val_accuracy: 0.6754Epoch 42/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9335
- accuracy: 0.6751 - val_loss: 0.9277 - val_accuracy: 0.6748Epoch 43/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9316
- accuracy: 0.6744 - val_loss: 0.9266 - val_accuracy: 0.6742Epoch 44/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9285
- accuracy: 0.6743 - val_loss: 0.9337 - val_accuracy: 0.6726Epoch 45/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9286
- accuracy: 0.6752 - val_loss: 0.9553 - val_accuracy: 0.6909Epoch 46/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9267
- accuracy: 0.6751 - val_loss: 0.9455 - val_accuracy: 0.6568Epoch 47/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9276
- accuracy: 0.6739 - val_loss: 0.9186 - val_accuracy: 0.6821Epoch 48/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9224
- accuracy: 0.6769 - val_loss: 0.9164 - val_accuracy: 0.6793Epoch 49/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9221
- accuracy: 0.6742 - val_loss: 0.9163 - val_accuracy: 0.6764Epoch 50/50
344/344 [==============================] - 1s 4ms/step - loss: 0.9203
- accuracy: 0.6774 - val_loss: 0.9130 - val_accuracy: 0.6817
858/858 [==============================] - 2s 3ms/step - loss: 0.9118
- accuracy: 0.6787
215/215 [==============================] - 1s 2ms/step - loss: 0.9130
- accuracy: 0.6817
Training accuracy: [0.9118128418922424, 0.6786742210388184]
Testing accuracy: [0.9129638671875, 0.6816725134849548]



Code:

In [37]:

```
def plotting_ffn(history):
```

- *# Plotting the curves of training, validation and test sets losses*

  and

- *# accuracy scores with number of epochs on the x-axis.*

  # Plot training & validation accuracy values

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

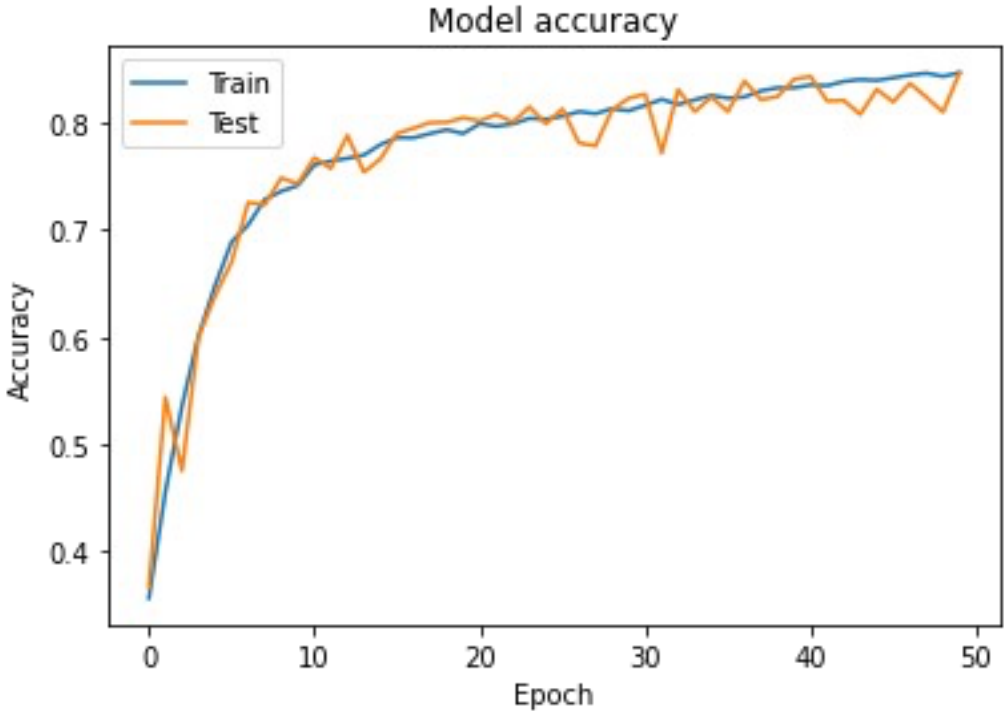
 # Plot training & validation loss values

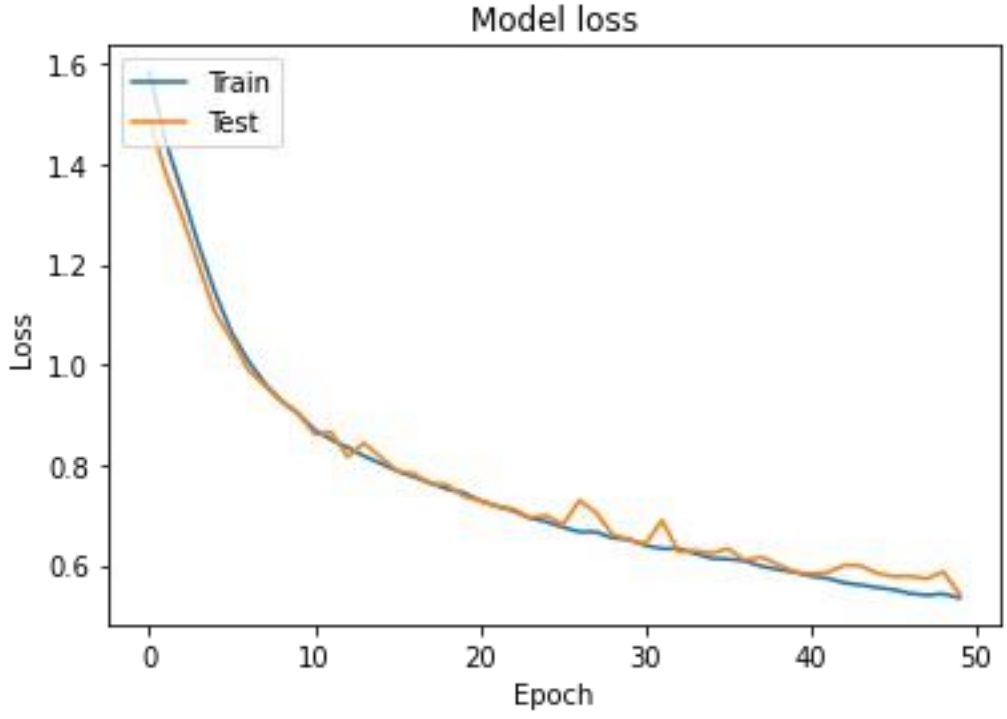
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
# Plotting the curves of training, validation and test sets losses and
# accuracy scores with number of epochs on the x-axis.
# Plot the training and validation curves
plotting_ffn(history6)
```

End of Code

Out [37]:





Code:

In [38]:

```
# Calculate the metrics for the model

y_pred = model6.predict(X_val)
y = y_pred.argmax(1)
y_pred6 = le.inverse_transform(y)
y_valout = le.inverse_transform(y_val)
#y_final = np.array(y)
print(y_pred6)
print(y_valout)
```

End of Code

Out [38]:

```
215/215 [==============================] - 0s 1ms/step
['U' 'L' 'N' ... 'R' 'L' 'N']
['U' 'N' 'N' ... 'R' 'L' 'U']
```

Code:

In [39]:

```
# Create function which calculates F1score, precision, recall and accuracy score for true and predicted labels.
def metrics(y_pred, y_true):

#pass

precision = precision_score(y_true, y_pred, average='macro') recall = recall_score(y_true, y_pred, average='macro') f1score = f1_score(y_true, y_pred, average='macro')
accuracy = accuracy_score(y_true, y_pred)

  print("Precision:", precision)
  print("Recall:", recall)
  print("F1-score:", f1score)
  print("Accuracy:", accuracy)

return

# Takes input the predicted and true labels.
# Your code here for precision, recall, F1score, accuracy # You can call this code to compute metrics for your models
```

End of Code

Code:

In [40]:

```
metrics(y_pred6, y_valout)
```

End of Code

Out [40]:

Precision: 0.8048770962096042
Recall: 0.7181585228350192
F1-score: 0.7159035814625927
Accuracy: 0.8468822843822844

Explanation for Exercise 6:

This code utilizes the Keras API from TensorFlow to train a neural network for multi-class classification. It begins by importing the necessary modules including the Sequential class and several layers from Keras, as well as TensorFlow's optimizers, regularizers, and metrics packages.

Next, the architecture of the neural network is defined. The input layer has a number of units equal to the number of features in the input data. The hidden layer is composed of 20 units and uses the ReLU activation function to mitigate vanishing gradients. The output layer has 6 units, each representing a possible category, and uses the softmax activation function to produce output values that sum to 1 and can be interpreted as probabilities.

The model is compiled using the compile method, specifying the loss function (sparse categorical crossentropy), optimizer (Adam), and metrics to track during training (accuracy).

The model is then trained using the fit method. This method takes in the training and validation data, the number of epochs to run, and the batch size for mini-batch stochastic gradient descent. The training accuracy and testing accuracy are printed during the training process, and the loss is plotted over the epochs.

This code demonstrates how to create and train a neural network using Keras and TensorFlow for multi-class classification.

Exercise 7

Code:

In [41]:

```
import statsmodels.api as sm
import numpy as np
import pandas as pd
```

End of Code

Code:

In [42]:

```
df_mini = pd.read_csv('df_mini.csv',sep=',',engine='python',error_bad_lines='skip',
quoting=csv.QUOTE_NONE)
```

End of Code

Code:

In [43]:

```
categorical_df = df_mini['Type'] merged_df =
df_mini.drop('Type', axis=1)
clean_data=cleaning(merged_df) normalized_df1 =
normalizer(clean_data) print(normalized_df1)

All_data=normalized_df1.copy()
  #A_data = noise_remover1(normalized_df)
og=All_data.shape
A_data = np.array(All_data)A_data =
A_data.ravel()
```

End of Code

Out [43]:

	0	1	2	3	4	5	6 \
0	0.286064	0.291375	0.294382	0.285078	0.297727	0.298851	0.305046
1	0.273839	0.270396	0.265169	0.267261	0.281818	0.289655	0.291284
2	0.268949	0.270396	0.276404	0.269488	0.279545	0.280460	0.284404
3	0.298289	0.289044	0.283146	0.278396	0.295455	0.303448	0.300459
4	0.246944	0.249417	0.258427	0.253898	0.263636	0.266667	0.270642
..	...	...	...	...	...	...	...
	0.317016	0.305618	0.285078	0.286364	0.291954		

882 0.312958
0.300459
883 0.332518
0.311927
884 0.281174
0.298165
885 0.374083
0.373853
886 0.178484
0.185780

	7						
352	\						
0	0.297357	0.295806	0.308889	...	0.243772	0.260628	0.242857
1	0.277533	0.271523	0.280000	...	0.229537	0.249538	0.226786
2	0.279736	0.275938	0.286667	...	0.241993	0.260628	0.242857
3	0.292952	0.291391	0.295556	...	0.243772	0.264325	0.241071
4	0.270925	0.271523	0.282222	...	0.250890	0.268022	0.251786
..	...	...	...	...	...	...	...
882	0.295154	0.284768	0.282222	...	0.234875	0.260628	0.244643
883	0.303965	0.295806	0.302222	...	0.238434	0.258780	0.233929
884	0.292952	0.293598	0.304444	...	0.304270	0.330869	0.287500

885 0.354626 0.353201 0.362222 ... 0.210714 0.201068 0.212569

886 0.189427 0.185430 0.177778 ... 0.217857 0.211744 0.227357

```
          353       354       355       356       357       358
 359
 0   0.244565 0.227826 0.223350 0.231933 0.225862 0.234750
0.259794
1       0.237319  0.222609  0.216582  0.211765  0.212069  0.223660
0.255670
2       0.248188  0.229565  0.226734  0.231933  0.222414  0.232902
0.255670
3       0.255435  0.234783  0.226734  0.220168  0.220690  0.232902
0.261856
 4    0.250000 0.231304  0.228426  0.231933  0.227586  0.238447
 0.263918
 ..        ...       ...       ...       ...       ...       ...
 ...
 882  0.248188 0.233043  0.236887  0.245378  0.244828  0.260628
 0.288660
 883           0.234783  0.228426  0.221849  0.222414  0.231054
 0.257246
 0.257732
 884           0.293913  0.291032  0.294118  0.298276  0.316081
 0.318841
 0.344330
 885           0.198261  0.194585  0.200000  0.198276  0.201479
 0.206522
 0.218557
 886           0.206957  0.208122  0.205042  0.205172  0.219963
 0.219203
 0.249485
```

[887 rows x 360 columns]

Code:

In [44]:

```
# Apply STL decomposition
period = 360
stl = sm.tsa.STL(A_data, period)res = stl.fit()
trend = res.trend seasonal =
res.seasonalresidual = res.resid
```

End of Code

Code:

In [45]:

```
  # Smooth residual using exponential moving average (EMA)residual =
  pd.DataFrame(residual,   columns=['Value']) ema =
  residual.ewm(span=30).mean()

  # Calculate moving average of smoothed residual using rolling window
sma2 = ema.rolling(window=50, center=True).mean()

  # Calculate anomaly scores based on difference between smoothedresidual and moving
  averages
sma1 = ema.rolling(window=10, center=True).mean() anomaly_scores =
np.abs(sma1['Value'] - sma2['Value'])

  # Threshold the anomaly scores to identify anomalies
threshold = anomaly_scores.mean() + 2 * anomaly_scores.std()
  End of Code
```

Code:

In [46]:

```
  # Identify the indices of the anomalies
anomaly_indices = np.where(anomaly_scores > threshold)[0]

  # Create a new column 'Anomaly' in normalized_df to store the anomalyoutput

All_data['Anomaly'] = 0
index= np.floor(anomaly_indices/360)index =
np.int16(index)
index = np.unique(index) print(index)
All_data['Anomaly'].iloc[index] = 1
  End of Code
```

Out [46]:

```
  [ 84  205  210  214  223  224  225  226  228  231  232  279  339  340  341  347  363
  364
   374  376  388  389  396  397  398  400  404  409  410  414  432  434  440  482  553
560
    621 634 635 636 655 656 657 658 659 660 664 665 666 667 668 669 683
684
    685 686 687 716 717 718 719 720 730 731 732 733 755 756 775 776 777
778
    779 780 801 810 811 812 813 814 843 844 845 846 847 848 849 850 851
852
    853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869
    870 871 872 873 874 875 876 877 878 879 884]
```

Code:

In [47]:

```
anoms = categorical_df!='N'
ans=categorical_df[anoms].index.values

inter = np.intersect1d(index,ans)score =
len(inter)/len(ans)
print(score)
```

Out [47]:

0.6875

End of Code

Explanation for Exercise 7:

This code is designed to perform time series anomaly detection using a combination of statistical techniques.

Firstly, the data is read in from a CSV file using Pandas and pre-processed by applying various cleaning and normalization functions. Next, the input data is decomposed into three components: trend, seasonal and residual, using the STL decomposition method from the Statsmodels library. The seasonal period is specified as 360, to match the number of samples per line.

The residual component is then smoothed using exponential moving averages (EMA), and the moving average of the smoothed residuals is calculated using a rolling window of size 50. Anomaly scores are then calculated based on the absolute difference between the smoothed residuals and the moving averages.

The anomaly scores are then thresholded to identify which data points should be marked as anomalies. The threshold is calculated as the mean of the anomaly scores plus twice the standard deviation. The code then identifies the indices of the anomalies based on the thresholded anomaly scores.

Finally, a new column called "Anomaly" is created in the original normalized_df, with any identified anomalies being marked as 1 in this new column.

To evaluate the accuracy of the anomaly detection, the code calculates the precision of the anomaly detection by comparing the identified anomalies with the true anomalies, marked by the 'Type' column in the original data. The precision is then calculated as the ratio of the number of correctly identified anomalies to the total number of true anomalies, equaling 0.69.

Exercise 8

Model based on https://ieeexplore.ieee.org/abstract/document/8952723 and https://www.kaggle.com/code/gregoiredc/arrhythmia-on-ecg-classification-using-cnn

Code:

In [48]:

```
from keras.layers import Dense, Convolution1D, MaxPool1D, Flatten, Dropout
from keras.layers import Input

from keras.models import Model
from tensorflow.keras.layers import BatchNormalization from keras.utils.np_utils import
to_categorical
import keras
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

End of Code

Out [48]:

Code:

In [49]:

```
X_train1 = X_train.reshape(len(X_train), X_train.shape[1],1)
X_val1 = X_val.reshape(len(X_val), X_val.shape[1],1)
y_train1=to_categorical(y_train)
y_val1=to_categorical(y_val)
```

End of Code

Out [49]:

Code:

In [50]:

```
X_train1 = X_train.reshape(len(X_train), X_train.shape[1],1)
X_val1 = X_val.reshape(len(X_val), X_val.shape[1],1)
y_train1=to_categorical(y_train)
y_val1=to_categorical(y_val)

def network(X_train,y_train,X_test,y_test):

   im_shape=(X_train.shape[1],1)
   inputs_cnn=Input(shape=(im_shape), name='inputs_cnn')
   conv1_1=Convolution1D(64, (6), activation='relu',
input_shape=im_shape)(inputs_cnn)
   conv1_1=BatchNormalization()(conv1_1)
   pool1=MaxPool1D(pool_size=(3), strides=(2), padding="same")
(conv1_1)
   conv2_1=Convolution1D(64, (3), activation='relu',
input_shape=im_shape)(pool1)
   conv2_1=BatchNormalization()(conv2_1)
   pool2=MaxPool1D(pool_size=(2), strides=(2), padding="same")
(conv2_1)
   conv3_1=Convolution1D(64, (3), activation='relu',
input_shape=im_shape)(pool2)
   conv3_1=BatchNormalization()(conv3_1)
   pool3=MaxPool1D(pool_size=(2), strides=(2), padding="same")
(conv3_1)
   flatten=Flatten()(pool3)
   dense_end1 = Dense(64, activation='relu')(flatten)
   dense_end2 = Dense(32, activation='relu')(dense_end1)
   main_output = Dense(6, activation='softmax', name='main_output')
(dense_end2)
   model = Model(inputs= inputs_cnn, outputs=main_output)
   model.compile(optimizer='adam',
loss='categorical_crossentropy',metrics = ['accuracy'])
   callbacks = [EarlyStopping(monitor='val_loss', patience=8),
        ModelCheckpoint(filepath='best_model.h5',
monitor='val_loss', save_best_only=True)]
   history=model.fit(X_train, y_train,epochs=25,callbacks=callbacks,
batch_size=80,validation_data=(X_test,y_test))

model.load_weights('best_model.h5')

return(model,history)
```

End of Code

Out [50]:

Code:

In [51]:

```
model8,history8=network(X_train1,y_train1,X_val1,y_val1)
```

End of Code

Out [51]:

```
Epoch 1/25
344/344 [==============================] - 34s 93ms/step - loss:
0.2661 - accuracy: 0.9207 - val_loss: 1.8671 - val_accuracy: 0.2446Epoch 2/25
344/344 [==============================] - 31s 91ms/step - loss:
0.1181 - accuracy: 0.9662 - val_loss: 1.2106 - val_accuracy: 0.6927Epoch 3/25
344/344 [==============================] - 37s 106ms/step - loss:
0.0909 - accuracy: 0.9738 - val_loss: 0.1807 - val_accuracy: 0.9516Epoch 4/25
344/344 [==============================] - 36s 103ms/step - loss:
0.0789 - accuracy: 0.9765 - val_loss: 0.1826 - val_accuracy: 0.9487Epoch 5/25
344/344 [==============================] - 46s 134ms/step - loss:
0.0655 - accuracy: 0.9796 - val_loss: 0.4451 - val_accuracy: 0.8488Epoch 6/25
344/344 [==============================] - 39s 114ms/step - loss:
0.0643 - accuracy: 0.9805 - val_loss: 0.5039 - val_accuracy: 0.8228Epoch 7/25
344/344 [==============================] - 40s 115ms/step - loss:
0.0500 - accuracy: 0.9847 - val_loss: 0.1124 - val_accuracy: 0.9675Epoch 8/25
344/344 [==============================] - 34s 100ms/step - loss:
0.0505 - accuracy: 0.9838 - val_loss: 2.5139 - val_accuracy: 0.5137Epoch 9/25
344/344 [==============================] - 32s 94ms/step - loss:
0.0405 - accuracy: 0.9878 - val_loss: 0.1607 - val_accuracy: 0.9545Epoch 10/25
344/344 [==============================] - 33s 96ms/step - loss:
0.0472 - accuracy: 0.9849 - val_loss: 0.7935 - val_accuracy: 0.6853Epoch 11/25
344/344 [==============================] - 33s 96ms/step - loss:
0.0415 - accuracy: 0.9867 - val_loss: 2.8114 - val_accuracy: 0.6286Epoch 12/25
344/344 [==============================] - 33s 95ms/step - loss:
0.0407 - accuracy: 0.9877 - val_loss: 0.1763 - val_accuracy: 0.9547Epoch 13/25
344/344 [==============================] - 33s 96ms/step - loss:
0.0345 - accuracy: 0.9893 - val_loss: 0.3725 - val_accuracy: 0.8951Epoch 14/25
344/344 [==============================] - 34s 98ms/step - loss:
0.0344 - accuracy: 0.9893 - val_loss: 0.0952 - val_accuracy: 0.9747Epoch 15/25
344/344 [==============================] - 33s 96ms/step - loss:
0.0322 - accuracy: 0.9893 - val_loss: 1.1682 - val_accuracy: 0.7733Epoch 16/25
344/344 [==============================] - 34s 99ms/step - loss:
0.0294 - accuracy: 0.9911 - val_loss: 3.1143 - val_accuracy: 0.4363
```

```
Epoch 17/25
344/344 [==============================] - 34s 97ms/step - loss:
0.0261 - accuracy: 0.9917 - val_loss: 0.9415 - val_accuracy: 0.7249Epoch 18/25
344/344 [==============================] - 34s 99ms/step - loss:
0.0266 - accuracy: 0.9913 - val_loss: 0.1128 - val_accuracy: 0.9735Epoch 19/25
344/344 [==============================] - 34s 98ms/step - loss:
0.0255 - accuracy: 0.9919 - val_loss: 1.3826 - val_accuracy: 0.7552Epoch 20/25
344/344 [==============================] - 33s 97ms/step - loss:
0.0231 - accuracy: 0.9919 - val_loss: 1.9746 - val_accuracy: 0.5838Epoch 21/25
344/344 [==============================] - 33s 97ms/step - loss:
0.0260 - accuracy: 0.9915 - val_loss: 5.0257 - val_accuracy: 0.4640Epoch 22/25
344/344 [==============================] - 33s 97ms/step - loss:
0.0274 - accuracy: 0.9907 - val_loss: 0.7120 - val_accuracy: 0.7982
```

Code:

In [52]:

```
#evaluate_model(history8,X_val1,y_val1,model8)
y_pred8=model8.predict(X_val1)
y_pred8 = y_pred8.argmax(1)
y_pred8 = le.inverse_transform(y_pred8)
```

Out [52]:

```
215/215 [==============================] - 2s 10ms/step
```

End of Code

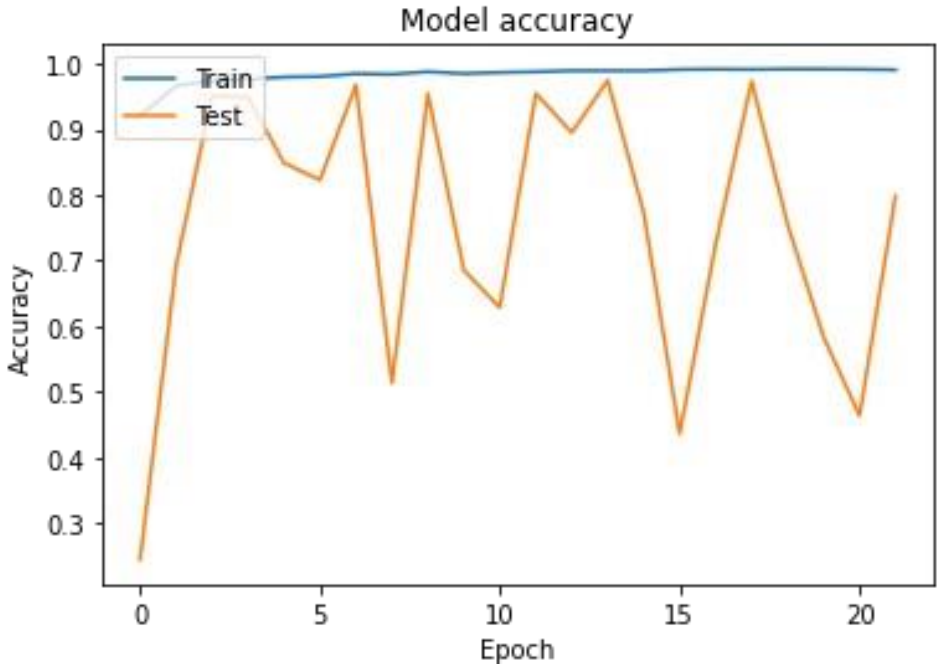
Code:

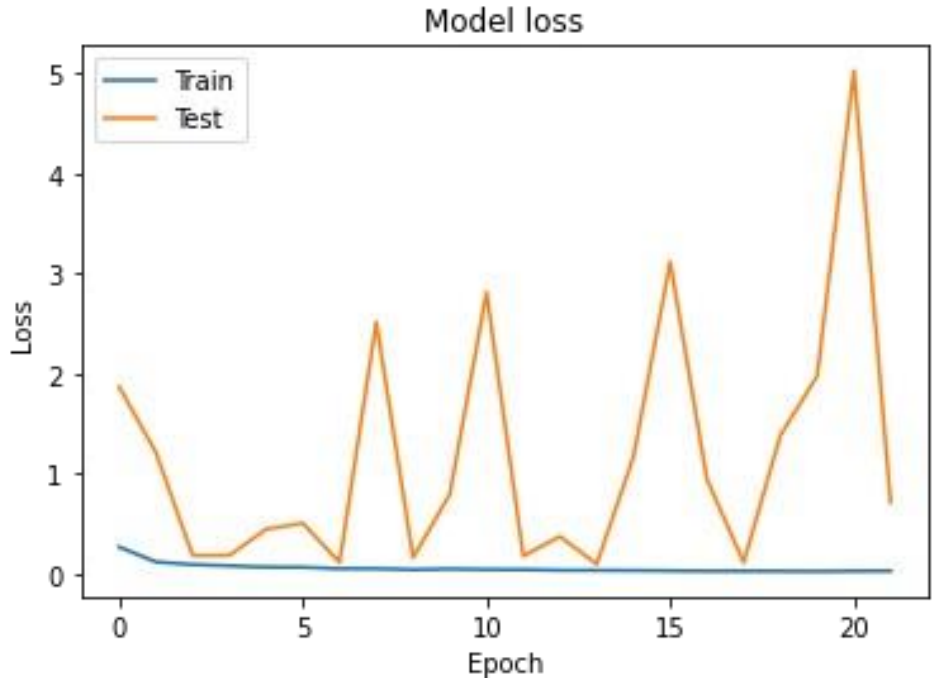
In [53]:

plotting_ffn(history8)

Out [53]:

End of Code





Explanation for Exercise 8:

The "network" function is a CNN model builder that can classify input data into six categories using the Keras library. This function requires four input arguments: X_train, y_train, X_test, and y_test.

The CNN model consists of three convolutional layers and two dense layers, where the activation functions are ReLU and softmax. The optimizer used is Adam, with a categorical cross-entropy loss function and accuracy metric. This function also includes callbacks for early stopping and model checkpoint to monitor and save the best performing model.

The input data is reshaped using reshape and converted to categorical data using to_categorical. The CNN is trained for 25 epochs with a batch size of 80, where the X_test and y_test datasets are used for validation during training.

Once the training is complete, the best model weights are loaded from the 'best_model.h5' checkpoint file, and both the trained model and its history are returned by the function. The performance of the model can be visualized using plotting_ffn.

This function is inspired by a research paper that proposes a CNN-based method for ECG heartbeat classification. The proposed method employs a coupled-convolution layer structure and dropout mechanism, making it suitable for real-time Holter applications, and shows high accuracy, sensitivity, and positive predictivity in detecting different arrhythmia heartbeats.

Exercise 9

Code:

In [54]:

```
print(y_pred8)
print(y_valout)
```

Out [54]:

```
['U' 'U' 'N' ... 'R' 'L' 'U']
['U' 'N' 'N' ... 'R' 'L' 'U']
```

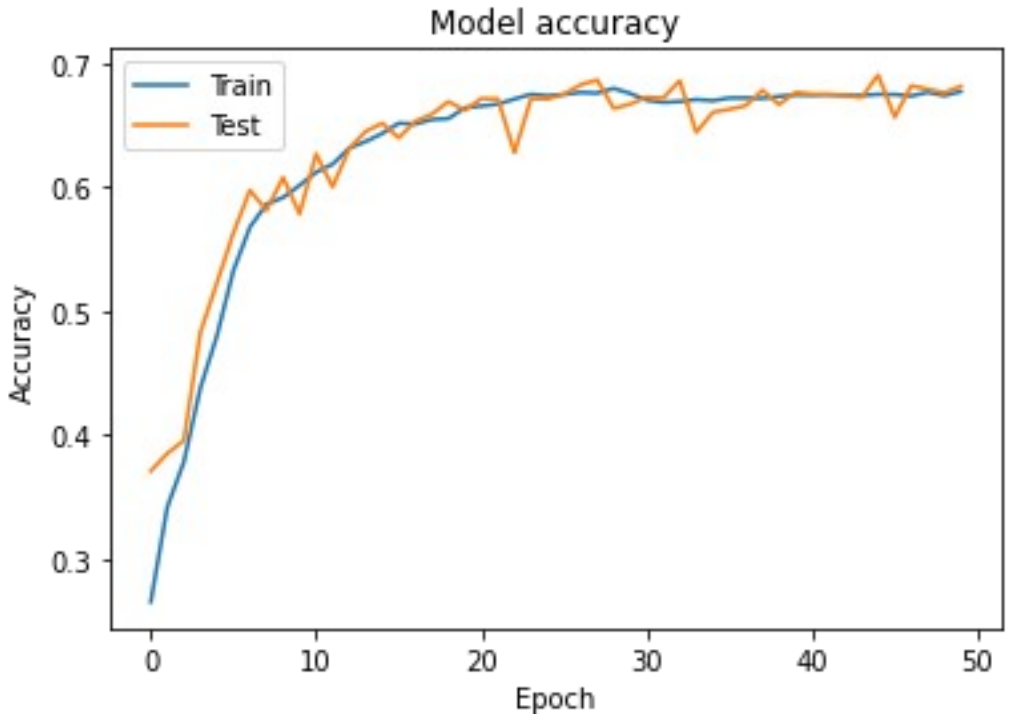
End of Code

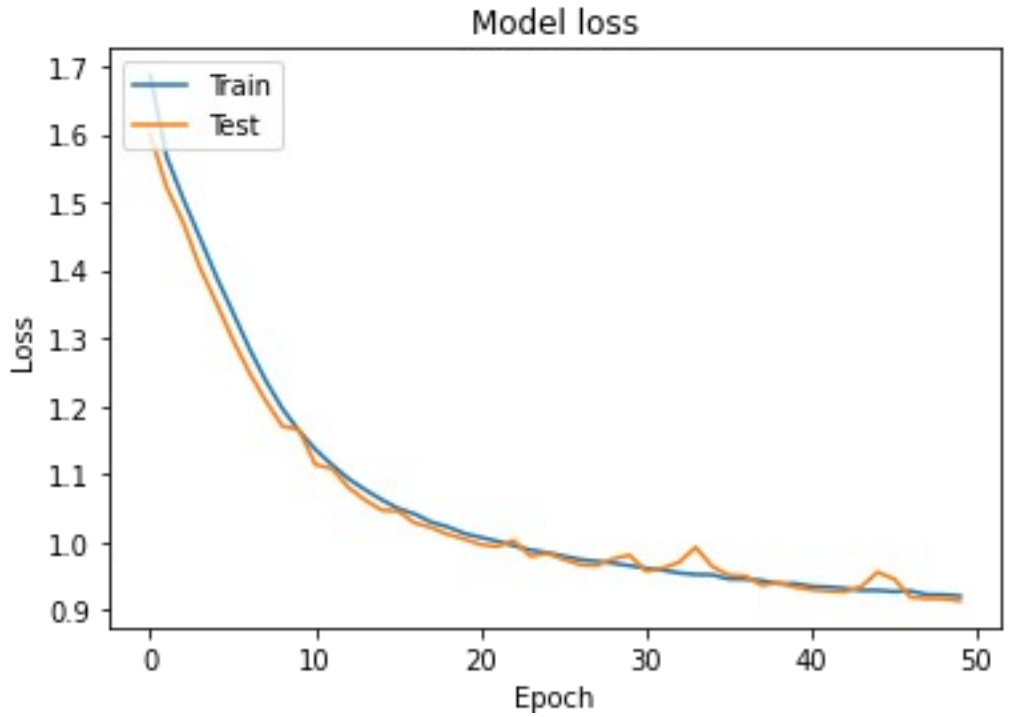
Code:

In [55]:

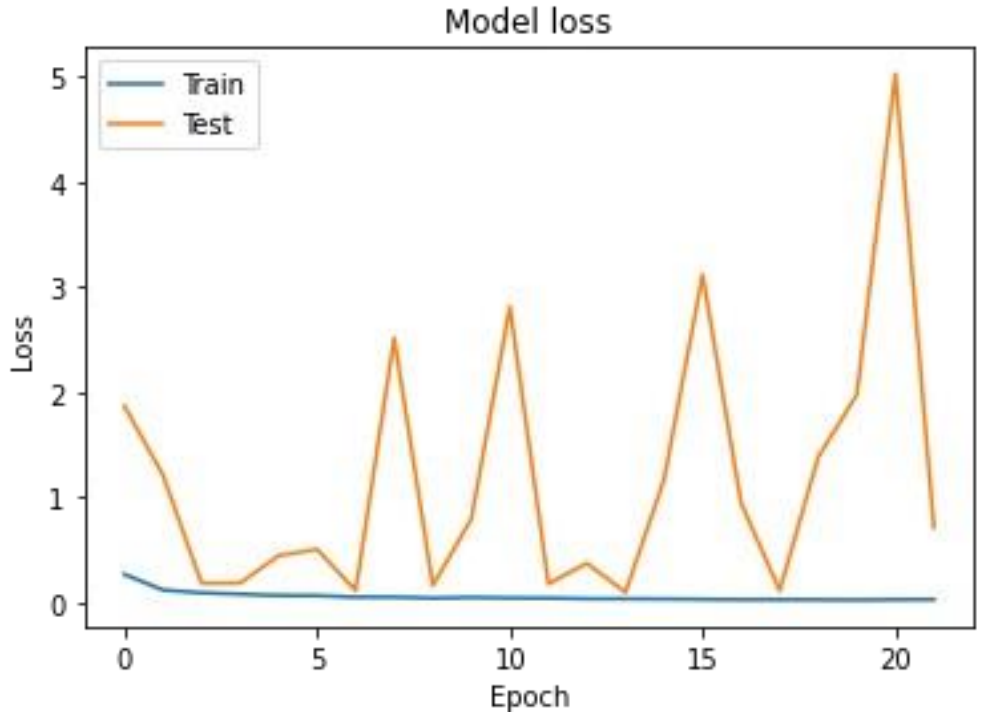
```
plotting_ffn(history6)
metrics(y_pred6, y_valout)
plotting_ffn(history8)
metrics(y_pred8, y_valout)
```

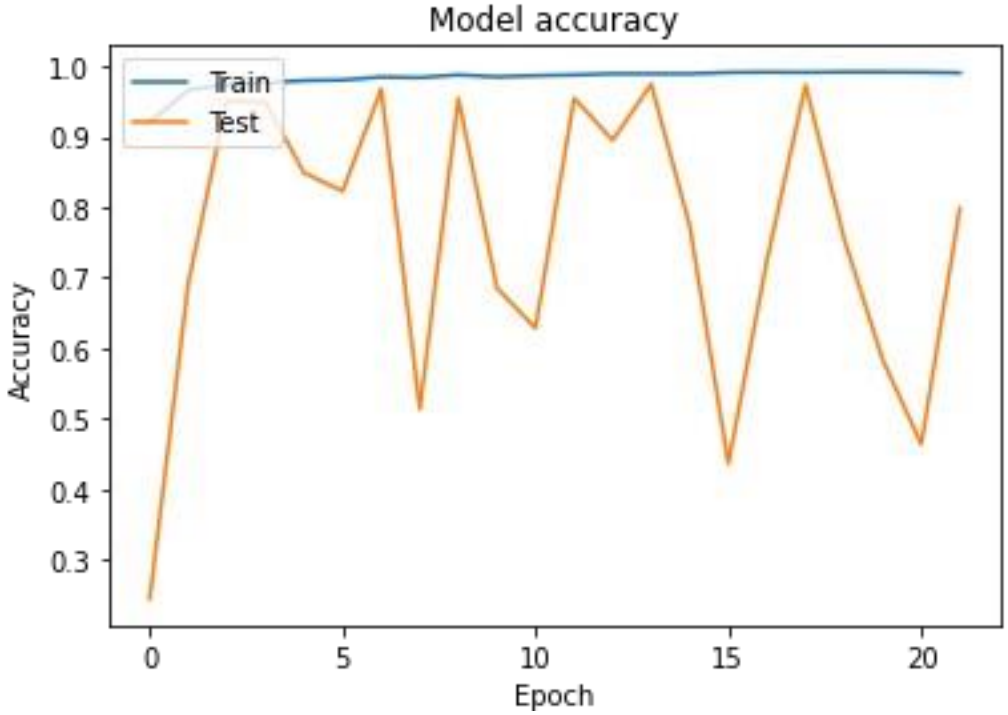
Out [55]:





Precision: 0.5838396884555153
Recall: 0.5641469479863288
F1-score: 0.5630617871497824
Accuracy: 0.6816724941724942





Precision: 0.960857037707567
Recall: 0.9653234611920322
F1-score: 0.9629907161543515
Accuracy: 0.9746503496503497

End of Code

Explanation for Exercise 9:

This code is evaluating two machine learning models for a classification problem.

The first block of code imports necessary libraries, reshapes the input data, and defines a function called network that creates a convolutional neural network (CNN) model using Keras. The model has several convolutional layers, batch normalization layers, and dense layers with a softmax output layer for classification. It is compiled with the Adam optimizer and categorical cross-entropy loss function. The function trains the model using the training data and early stopping with validation data. The best weights of the trained model are saved to a file called best_model.h5.

The second block of code calls the network function for two different sets of data and stores the returned models and training history in model6, history6, model8, and history8. It also generates predictions for the validation data using both models and calculates various classification metrics using these predictions and the true labels. Finally, it plots the training and validation accuracy and loss curves for both deep learning models using the training history.

Exercise 10

Code:

In [56]:

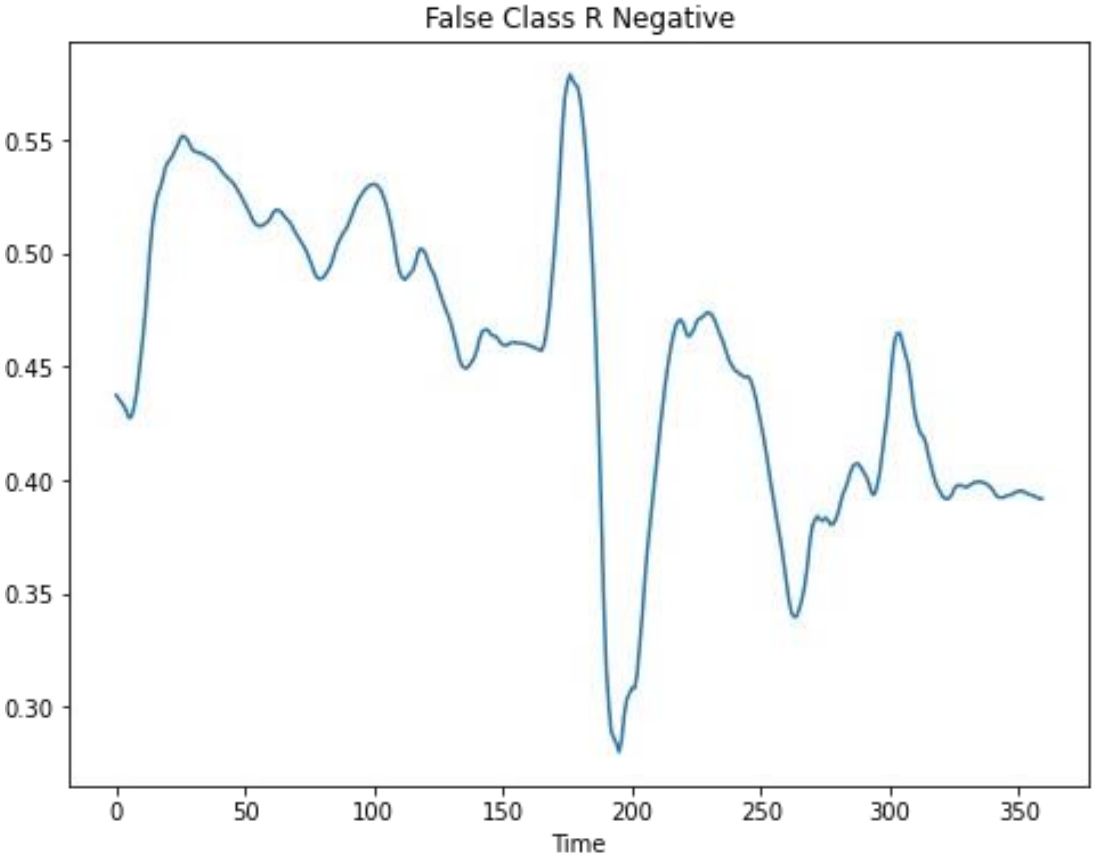
```
# Find the indices of false positives and false negatives for Class Rfalse_positives =
np.where((y_pred8 == 'R') & (y_valout != 'R')) print('False Positive
Quantity:',len(false_positives[0])) false_negatives = np.where((y_pred8 != 'R') & (y_valout ==
'R')) print('False Negative Quantity:',len(false_negatives[0]))


for i in range(1): plt.figure(figsize=(8, 6))
        plt.title('False Class R Positive')
        plt.xlabel('Time')
        plotX = X_val[false_positives[0][i]]plt.plot(plotX)
        plt.show()

for i in range(1): plt.figure(figsize=(8, 6))
        plt.title('False Class R Negative')
        plt.xlabel('Time')
        plotX = X_val[false_negatives[0][i]]
```

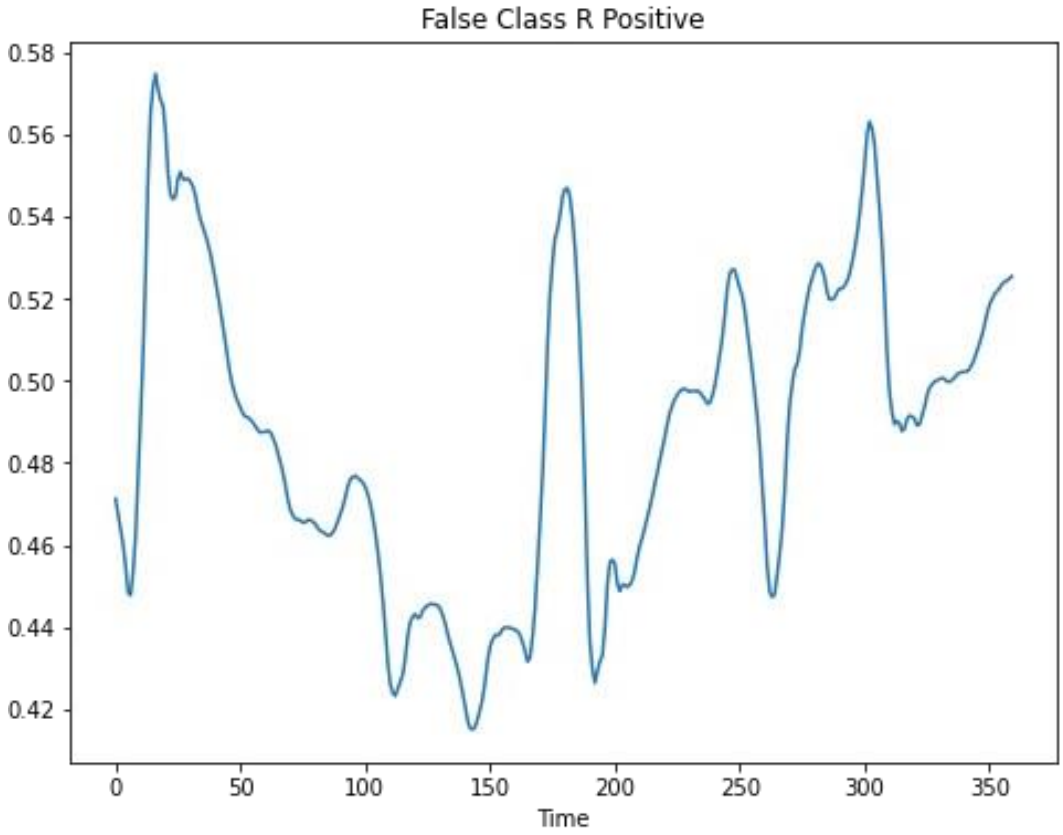
```
plt.plot(plotX)
plt.show()
```

```
End of Code
```

Out [56]:

False Positive Quantity: 4
False Negative Quantity: 15





Explanation for Exercise 10:

This code is used to find and display the false positive and false negative results for Class R in the predicted output.

The first line of the code uses the numpy where() function to find the indices of false positives by checking if the predicted class is R but the actual class is not R. The second line of the code finds the indices of false negatives by checking if the predicted class is not R but the actual class is R.

The next block of code displays the false positive and false negative results for Class R. It plots the time series data for one instance of false positive and false negative results separately. The plt.figure() function is used to set the figure size and plt.title() function is used to set the title of the plot. The plt.plot() function is used to plot the time series data and plt.show() function is used to display the plot.

Finally, the code prints the number of false positives and false negatives for Class R.

Exercise 11

Code:

In [57]:

```
df_kag =
pd.read_csv('kaggle.csv',sep=',',engine='python',error_bad_lines='skip
', quoting=csv.QUOTE_NONE)
```

End of Code

Out [57]:

Code:

In [58]:

```
id =df_kag['Id']
X_final = df_kag.drop('Id', axis=1)
clean_X = cleaning(X_final)
normalized_X = normalizer(clean_X) #normalized_X[np.isnan(normalized_X)] = 0
normalized_X = normalized_X.to_numpy()
noise_X = noise_remover1(normalized_X)
noise_X = noise_X.reshape(len(noise_X), noise_X.shape[1],1) #only for model 8

y_predict = model8.predict(noise_X) #print(y_predict)
#y_predict = np.array(y_predict).astype(int) y = y_predict.argmax(1)

#print(y)

y_final = le.inverse_transform(y) #y_final = np.array(y)

print(y_final)
```

End of Code

Out [58]:

```
285/285 [==============================] - 3s 10ms/step['R' 'R'
'R' ... 'N' 'N' 'N']
```

Code:

In [59]:

```
excel = pd.DataFrame(index=None)
filepath =
Path('C:/UW_PMP_Masters/EE_____P_594_A_Machine_Learning/MiniProject2_HeartPulse/kaggle_M8unbalanced.csv')
excel['Id']=id excel['heartbeat_type']=y_final
#print(excel) excel.to_csv(filepath,
index=False)
```

End of Code

Explanation for Exercise 11:

In this code, a CSV file named 'kaggle.csv' is read and stored in a pandas dataframe. The 'Id' column is extracted and saved in a separate variable. The remaining columns are cleaned using a custom cleaning function and then normalized using another custom function. The normalized data is then passed through a noise removal function before being reshaped to a format compatible with model 8. The cleaned, normalized, and noise-removed data is then fed to model 8 for prediction using the 'predict' function. The predictions are then converted back to their original labels using the 'inverse_transform' function of the label encoder. The predicted labels are saved to a new CSV file along with their corresponding 'Id' values.