

Forecasting Stress From Wearable Devices

- Author: Kevin Spring
- Blog post: <https://medium.com/@kevinjspring/stressful-conditions-6e6fc5eb018a>

Summary

We are HealthOn, a wearable device manufacturer focused on improving the health of our users. Our devices measures many physiological data in real time. Using publically available data, we determine the feasability of forecasting stress within a 5-minute period. The best models are only slightly better than guessing the stress or normal state of a user. More data needs to be collected on a variety of subjects under stressful conditions to improve the models.

We recommend:

1. Enhance Feature Measurement: Focus on measuring physiological features that have a high causality with stress, such as respiration rate, heart rate variability, body temperature, and electrodermal activity.
2. Expand Data Collection: Gather a more diverse and comprehensive dataset. Collecting data from a larger sample size of individuals, particularly in both stressful and normal conditions, will allow for a better understanding of the variations and patterns associated with stress.
3. Focus on Stressful Conditions: Prioritize data collection during high-stress situations or events. Collecting data from individuals undergoing stressful experiences, such as work-related stress, performance anxiety, or challenging life events. This targeted data collection will help train the models to better identify and predict stress states accurately.
4. Continuous Model Improvement: It is essential to continuously refine and enhance the machine learning models. Regularly analyze the performance of the models, identify areas for improvement, and iterate on the algorithms and techniques used.

Table of Contents

- [Introduction](#)
- [Business Problem](#)
- [Data](#)
- [Raw Data Feature Extraction](#)
- [Raw Data EDA and Preprocessing](#)
- [Preprocessed Data EDA](#)
- [Feature Engineering](#)
- [Model Data Generation](#)
- [Modeling](#)
- [Model Evaluation](#)
- [Conclusions](#)
- [Recommendations](#)
- [References](#)

Introduction

HealthOn is focused on improving the health of users through the development of wearable device technology. We currently have a range of smartwatches that collect heart rate, steps taken, and activity level. We are designing our next generation of smartwatches and want to identify which features to include that will take health monitoring to a new level.

Stress

Stress can have detrimental effects on both mental and physical health. When individuals experience prolonged or chronic stress, it can contribute to various mental health disorders such as anxiety, depression, and burnout. Additionally, stress can become physical symptoms and lead to conditions such as cardiovascular diseases, gastrointestinal problems, weakened immune system, and impaired cognitive functioning. Furthermore, chronic stress can exacerbate existing health conditions and hinder the recovery process. Additionally, stress-related issues contribute to increased healthcare costs, decreased job satisfaction, and lower overall economic growth [1](#).

Recognizing the significant economic costs, high prevalence, and negative impacts on individuals, it is crucial to prioritize stress management and promote strategies that reduce stress levels.

Business Problem

HealthOn is focused on improving the health of users through the development of wearable devices. The goal is to collect and analyze physiological data in real time to provide insights and interventions to improve users health.

Stress is a significant health related burden for individuals. It can lead to mental and somatic health issues. Physiological changes in the body are correlated with a person going into a stressful situation, for example, they will sweat more and their heart rate will increase.

We want to forecast if a user will be in a stressful state in the immediate future. Our device will then notify the user with recommendations to mitigate the stress.

Project Objective

This project aims to assess the feasibility of using physiological data to forecast a subject going into stress. It also aims to identify the most important features in detecting stress for the aim of improving wearable device development and design.

Stakeholders

- Project Manager
- Business executives

Data

Data Source

The [WESAD dataset](#) is a publicly available dataset designed for wearable stress and affect detection. It was collected during a lab study and includes physiological and motion data from 15 subjects. The data was recorded using two devices: a chest-worn device called RespiBAN and a wrist-worn device called Empatica E4 ([2](#)).

The RespiBAN device provides data on various sensors, including electrocardiogram (ECG), electrodermal activity (EDA), electromyogram (EMG), respiration, body temperature, and three-axis acceleration. All signals from this device are sampled at a rate of 700 Hz.

The Empatica E4 device captures data from sensors such as blood volume pulse (BVP), electrodermal activity (EDA), body temperature, and three-axis acceleration. The sampling rates for these signals are 64 Hz, 4 Hz, 4 Hz, and 32 Hz, respectively.

The dataset contains three affective states: neutral, stress, and amusement, but neutral and amusement were recharacterized as the same nonstress case (case 0). The neutral state serves as a baseline condition where subjects were engaged in standing or sitting while reading magazines. The amusement condition involves subjects watching a set of eleven funny video clips. The stress condition involves subjects participating in the Trier Social Stress Test (TSST), which includes public speaking and mental arithmetic.

Additionally, the dataset includes self-reports of the subjects obtained using established questionnaires, providing additional information about their experiences and perceptions.

For further details about the dataset structure, data format, study protocol, and self-report questionnaires, you can refer to the dataset's readme-file or the provided source.

Download instructions

The raw data can be downloaded as a [zip file \(2.1 GB\)](#) or available in a [Google Drive directory](#). After downloading the files store them in a top-level Google Drive directory as stress-prediction or modify the code below to reflect your working directory.

```
1 # Import the necessary packages for data loading and processing
2
3 import matplotlib.pyplot as plt # plotting
4 import numpy as np # linear algebra
5 import os # accessing directory structure
6 import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
7 import glob
8 import pickle
9
10 # install neurokit2 for signal processing
11 ! pip install neurokit2
12 import neurokit2 as nk
13
14 # ignore warnings
15 import warnings
16 warnings.filterwarnings("ignore")
```

```

Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting neurokit2
  Downloading neurokit2-0.2.4-py2.py3-none-any.whl (1.3 MB)
    1.3/1.3 MB 29.2 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from neurokit2) (1.22.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from neurokit2) (1.5.3)
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from neurokit2) (1.10.1)
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from neurokit2) (1.2.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from neurokit2) (3.7.1)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.0->neurokit2) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0.0->neurokit2) (3.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (1.0.7)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (4.39.3)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (1.4.4)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (23.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (8.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib->neurokit2) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->neurokit2) (2022.7.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib->neurokit2) (1.16.0)
Installing collected packages: neurokit2
Successfully installed neurokit2-0.2.4

```

```

1 # Mount Google Drive
2 from google.colab import drive
3 drive.mount('/content/drive', force_remount=True)

        Mounted at /content/drive

1 # View files in the directory stress-prediction
2 ! ls /content/drive/MyDrive/stress-prediction

        data  feature_importance.pdf  img  logs  models

1 # set path for data directory
2 os.chdir('/content/drive/MyDrive/stress-prediction')
3
4 # absolute path of data directory
5 PATH = os.path.join(os.path.abspath(os.getcwd()), 'data') # Path of data folder on Google Drive

```

▼ Open and clean up raw data

The data for the WESAD data is stored in separate folders based on the subject. Within each of these subject subfolders is a pickle (.pkl) file that contains all the physiological data collected on this subject. This is the file that will be loaded for modeling.

```

1 original_dir = os.path.join(PATH, 'original')
2 org_WESAD_dir = os.path.join(original_dir, 'WESAD')
3 print(org_WESAD_dir)

        /content/drive/MyDrive/stress-prediction/data/original/WESAD

1 class SubjectData:
2
3     """
4         This code defines a class called `SubjectData` that stores raw data from
5         the WESAD dataset and provides helper functions to extract wrist or
6         chest data.
7         Modified from https://github.com/WJMatthew/WESAD/blob/master/data\_wrangling.py
8
9     Attributes:
10        - `name`: Subject name.
11        - `subject_keys`: Keys for subject data.
12        - `signal_keys`: Keys for signal data.
13        - `chest_keys`: Keys for chest data.
14        - `wrist_keys`: Keys for wrist data.
15
16    Methods:
17        - `__init__(self, main_path, subject_number)`: Initializes the object with
18            the main path and subject number. Loads the raw data from the WESAD
19            dataset and sets the labels accordingly.
20        - `get_wrist_data(self)`: Retrieves the wrist data from the loaded data,
21            adds additional features, and includes the labels. Returns the updated data.

```

```

22     - `get_chest_data(self)`: Retrieves the chest data from the loaded data
23         and returns it.
24     ...
25     def __init__(self, main_path, subject_number):
26         self.name = f'S{subject_number}'
27         self.subject_keys = ['signal', 'label', 'subject']
28         self.signal_keys = ['chest', 'wrist']
29         self.chest_keys = ['ACC', 'ECG', 'EMG', 'EDA', 'Temp', 'Resp']
30         self.wrist_keys = ['BVP', 'EDA', 'TEMP']
31         with open(os.path.join(main_path, self.name) + '/' + self.name + '.pkl', 'rb') as file:
32             self.data = pickle.load(file, encoding='latin1')
33         self.labels = self.data['label']
34         # Set labels > or < 2 as non-stress (case 0)
35         self.labels[self.labels < 2] = 0
36         self.labels[self.labels > 2] = 0
37         # Set labels == 2 as stress state (case 1)
38         self.labels[self.labels == 2] = 1
39
40     def get_data(self):
41         data = self.data['signal']['chest']
42         # Modified to extract chest ECG data too
43         data.update({'ECG': self.data['signal']['chest']['ECG']})
44         data.update({'EMG': self.data['signal']['chest']['EMG']})
45         data.update({'HRV': self.data['signal']['wrist']['BVP']})
46         data['labels'] = self.labels
47         return data
48
49     def get_chest_data(self):
50         return self.data['signal']['chest']

1 def subject_data_import(directory):
2     """
3     WESAD raw data is stored in a pkl file of each specific subject.
4     This function scans through the directories and imports the pkl files.
5     It then calls the SubjectData object and stores the subject dictionaries.
6     """
7     import regex as re
8
9     regex = re.compile(r'\d+')
10    subject_li = []
11    ext = '.pkl' # pickle extension
12    # Walk through data directory and only return with specific extension
13    for path, dirc, files in os.walk(directory):
14        for name in files:
15            if name.endswith(ext):
16                # Extract the subject number
17                subject_number = regex.findall(name)
18
19                # Create SubjectData object
20                subject_li.append(SubjectData(directory, int(subject_number[0])))
21
22    return subject_li

1 # Store all subject data into a list of SubjectData objects
2 # Each SubjectData object represents a single subject
3 wesad_sd_li = subject_data_import(org_WESAD_dir)

```

▼ Raw Data Feature Extraction

NeuroKit2 is a Python library designed for the analysis and processing of physiological data, specifically focusing on electrocardiography (ECG), electrodermal activity (EDA), electromyography (EMG), and other biosignals. It provides a wide range of functionalities to clean, process, and analyze physiological data efficiently.

The main purpose of NeuroKit2 is to simplify the analysis pipeline for researchers and practitioners working with physiological data. It offers a high-level interface that abstracts complex operations and algorithms, allowing users to focus on their research questions rather than the technical details of signal processing.

One of the key features of NeuroKit2 is its ability to perform comprehensive cleaning and preprocessing of physiological data. It includes functionalities for handling artifacts and noise, detecting and correcting anomalies, filtering signals, and normalizing data. These preprocessing steps are crucial to ensure the quality and reliability of the physiological data before further analysis.

```

1 # Extract data and store in a list
2 # Will create a list of dictionaries for each subject
3 wesad_data_li = [ f.get_data() for f in wesad_sd_li ]

1 # Inspect features for the first subject
2 wesad_data_li[0].keys()

dict_keys(['ACC', 'ECG', 'EMG', 'EDA', 'Temp', 'Resp', 'HRV', 'labels'])

1 # Perform automated processing of bio signals using NeuroKit2
2 # bio_process() is a wrapper function for processing ECG, RSP, EDA, EMG, and other bio signals
3 # It automatically processes the provided bio signals and extracts relevant features
4
5 # Call the bio_process() function with the respective bio signals and sampling rate
6 processed_data, info = nk.bio_process(eda=wesad_data_li[0]['EDA'].reshape(-1),
7                                         rsp=wesad_data_li[0]['Resp'].reshape(-1),
8                                         ecg=wesad_data_li[0]['ECG'].reshape(-1),
9                                         sampling_rate=700)
10
11 # The processed_data variable contains the processed bio features such as
12 # cleaned signals, heart rate, R peaks, etc.
13 # The info variable contains a dictionary with additional information about
14 # the processed signals
15 # Both variables can be used for further analysis or visualization

```

```

1 # Add temperature to the processed_data dataframe
2 # This is added after because temperature is not processed with the other
3 # physiological signals
4 processed_data['TEMP'] = wesad_data_li[0]['Temp']

```

▼ Raw Data EDA and Preprocessing

```

1 # Data size
2 print(len(wesad_data_li[0]['EDA']))
3
4 # Drop rows that all data is Nan
5 processed_data.dropna(how='all', inplace=True)
6 len(processed_data)

3847200
3847200

1 # Inspect processed data dataframe for first subject
2 processed_data.info(verbose=True, show_counts=True)

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3847200 entries, 0 to 3847199
Data columns (total 44 columns):
 #   Column           Non-Null Count  Dtype  
 ---  -- 
 0   ECG_Raw          3847200 non-null   float64
 1   ECG_Clean         3847200 non-null   float64
 2   ECG_Rate          3847200 non-null   float64
 3   ECG_Quality       3847200 non-null   float64
 4   ECG_R_Peaks       3847200 non-null   int64  
 5   ECG_P_Peaks       3847200 non-null   int64  
 6   ECG_P_Onsets      3847200 non-null   int64  
 7   ECG_P_Offsets     3847200 non-null   int64  
 8   ECG_Q_Peaks       3847200 non-null   int64  
 9   ECG_R_Onsets      3847200 non-null   int64  
 10  ECG_R_Offsets     3847200 non-null   int64  
 11  ECG_S_Peaks       3847200 non-null   int64  
 12  ECG_T_Peaks       3847200 non-null   int64  
 13  ECG_T_Onsets      3847200 non-null   int64  
 14  ECG_T_Offsets     3847200 non-null   int64  
 15  ECG_Phase_Atrial  3846321 non-null   float64
 16  ECG_Phase_Completion_Atrial 3847200 non-null   float64
 17  ECG_Phase_Ventricular 3846229 non-null   float64
 18  ECG_Phase_Completion_Ventricular 3847200 non-null   float64
 19  RSP_Raw            3847200 non-null   float64
 20  RSP_Clean          3847200 non-null   float64
 21  RSP_Amplitude      3847200 non-null   float64
 22  RSP_Rate            3847200 non-null   float64
 23  RSP_RVT             3847200 non-null   float64

```

```

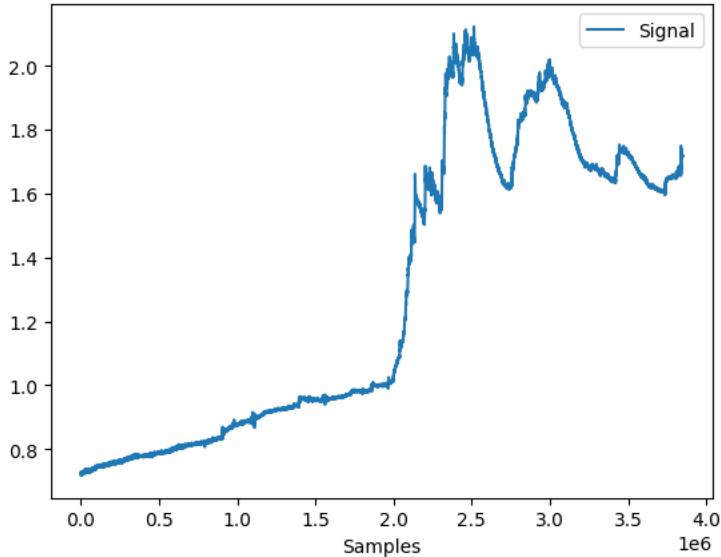
24 RSP_Phase           3842297 non-null float64
25 RSP_Phase_Completion 3847200 non-null float64
26 RSP_Symmetry_PeakTrough 3847200 non-null float64
27 RSP_Symmetry_RiseDecay 3847200 non-null float64
28 RSP_Peaks            3847200 non-null int64
29 RSP_Troughs           3847200 non-null int64
30 EDA_Raw              3847200 non-null float64
31 EDA_Clean             3847200 non-null float64
32 EDA_Tonic             3847200 non-null float64
33 EDA_Phasic            3847200 non-null float64
34 SCR_Onsets            3847200 non-null int64
35 SCR_Peaks             3847200 non-null int64
36 SCR_Height             3847200 non-null float64
37 SCR_Amplitude          3847200 non-null float64
38 SCR_RiseTime           3847200 non-null float64
39 SCR_Recovery            3847200 non-null int64
40 SCR_RecoveryTime        3847200 non-null float64
41 RSA_P2T                3847200 non-null float64
42 RSA_Gates               3847200 non-null float64
43 TEMP                   3847200 non-null float32
dtypes: float32(1), float64(27), int64(16)
memory usage: 1.2 GB

```

```

1 # inspect the cleaned EDA signal
2 nk.signal_plot(processed_data['EDA_Clean'])
3 plt.savefig('/content/drive/MyDrive/stress-prediction/img/eda_clean.pdf')

```



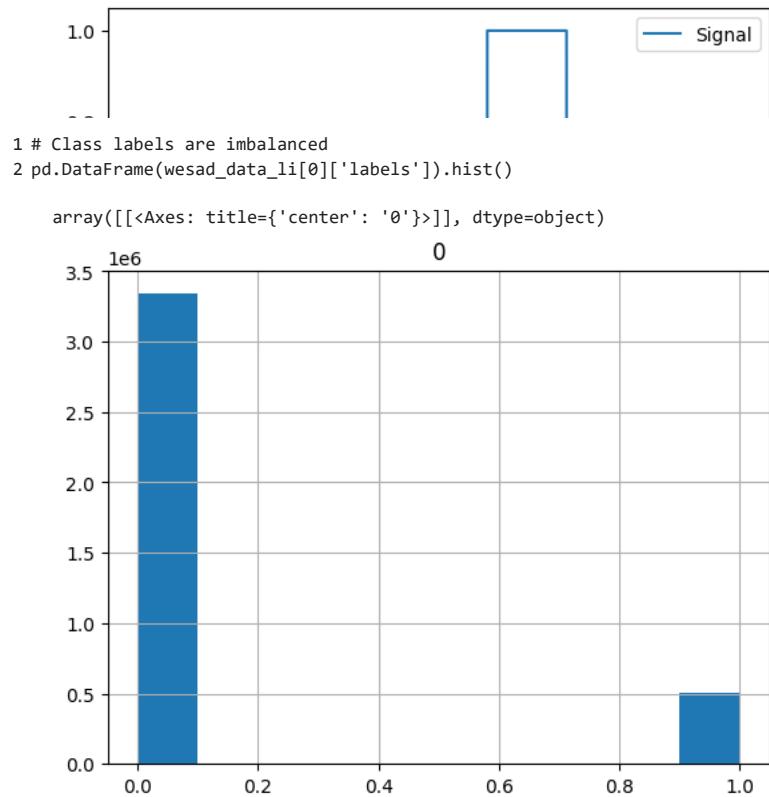
```

1 # What are the unique label IDs?
2 np.unique(wesad_data_li[0]['labels'])

array([0, 1], dtype=int32)

1 # Plot the signal
2 nk.signal_plot(wesad_data_li[0]['labels'])

```



```

1 # Class labels are imbalanced
2 pd.DataFrame(wesad_data_li[0]['labels']).hist()
array([[[<Axes: title={'center': '0'}>]], dtype=object)

1 # Find events using the events_find() function from NeuroKit2
2 # Only on the first subject in exploration
3
4 ***
5 The event_channel parameter represents the channel containing the events
6 In this case, it is extracted from the 'labels' column of the `wesad_data_li` dataset
7 The function automatically detects and selects events based on specified parameters
8 These parameters include threshold, duration, onset time, etc., which can be adjusted as needed
9 The function returns a dictionary containing arrays of event onsets, durations, labels, and conditions (if specified)
10 The 'onset' array represents the onset times of the events
11 The 'duration' array represents the durations of the events
12 The 'label' array contains the event identifiers
13 The 'conditions' array contains optional information about the event conditions
14 The resulting 'events' dictionary can be used for further analysis or visualization
15 ***
16 # Call the events_find() function with the event channel from the `wesad_data_li` dataset
17 events = nk.events_find(event_channel=wesad_data_li[0]['labels'])
18 events

{'onset': array([2234999]),
 'duration': array([507500]),
 'label': array(['1'], dtype='<U21')}

1 # Plot the event when stress is induced for the first subject
2 plot = nk.events_plot(events, wesad_data_li[0]['labels'])
3 plt.savefig('/content/drive/MyDrive/stress-prediction/img/signal_ex.pdf')

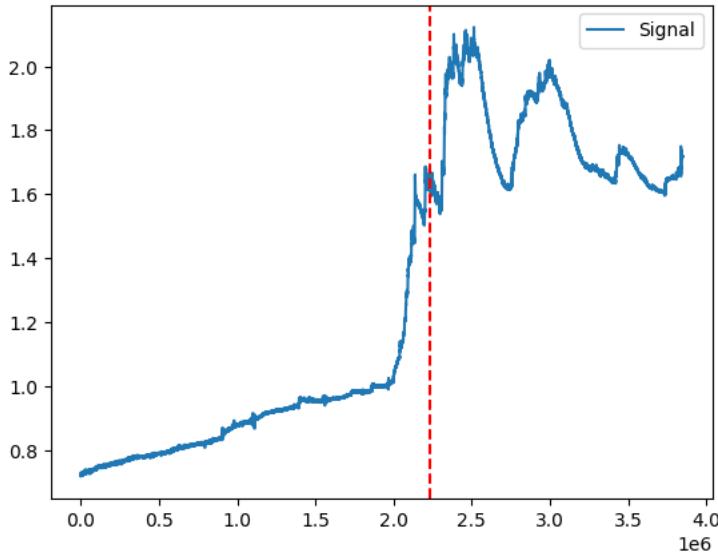
```



```

1 # Plot the signal event with the cleaned EDA signal
2 plot = nk.events_plot(events, processed_data['EDA_Clean'])
3 plt.savefig('/content/drive/MyDrive/stress-prediction/img/eda_w_signal.pdf')

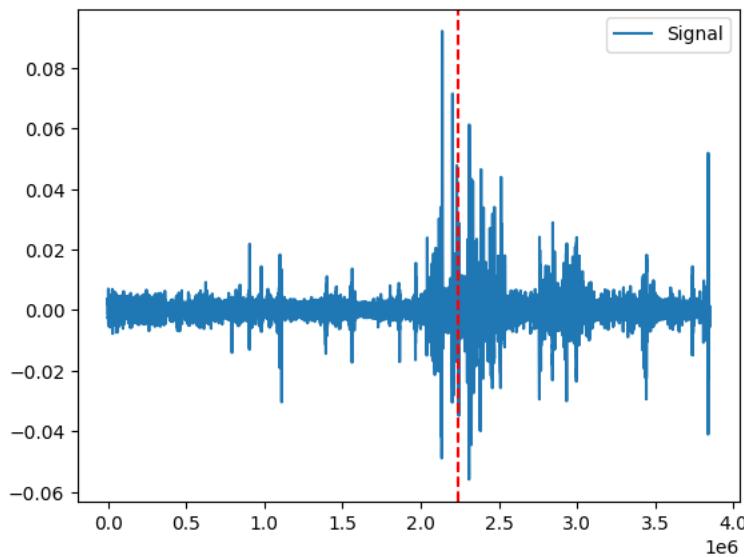
```



```

1 # Plot the signal event with the EDA Phase signal of the first subject
2 plot = nk.events_plot(events, processed_data['EDA_Phasic'])
3 plt.savefig('/content/drive/MyDrive/stress-prediction/img/eda_phase_signal.pdf')

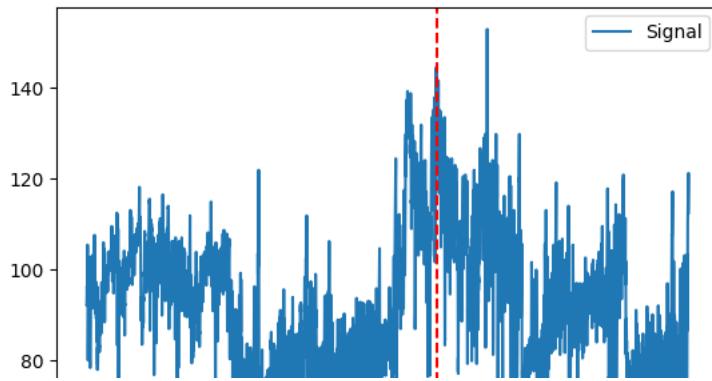
```



```

1 # Plot the signal event with the ECG rate signal of the first subject
2 nk.events_plot(events, processed_data['ECG_Rate'])
3 plt.savefig('/content/drive/MyDrive/stress-prediction/img/ECG_Rate_signal.pdf')

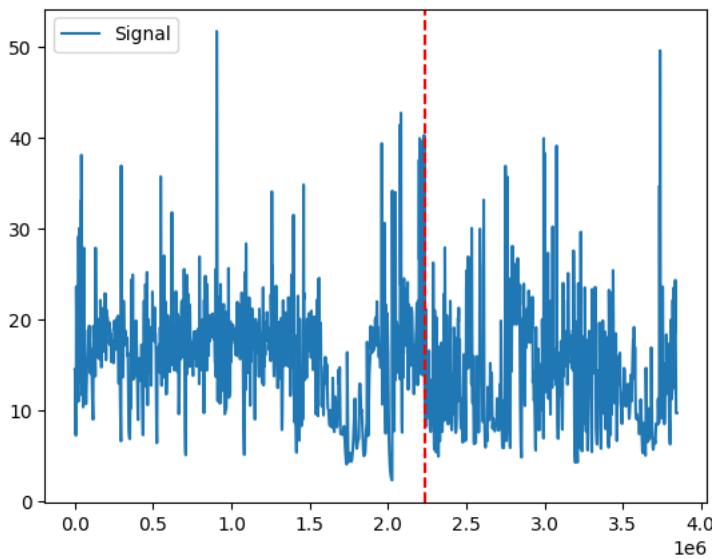
```



```

1 # Plot the signal event with the respiration rate of the first subject
2 nk.events_plot(events, processed_data['RSP_Rate'])
3 plt.savefig('/content/drive/MyDrive/stress-prediction/img/RSP_Rate_signal.pdf')

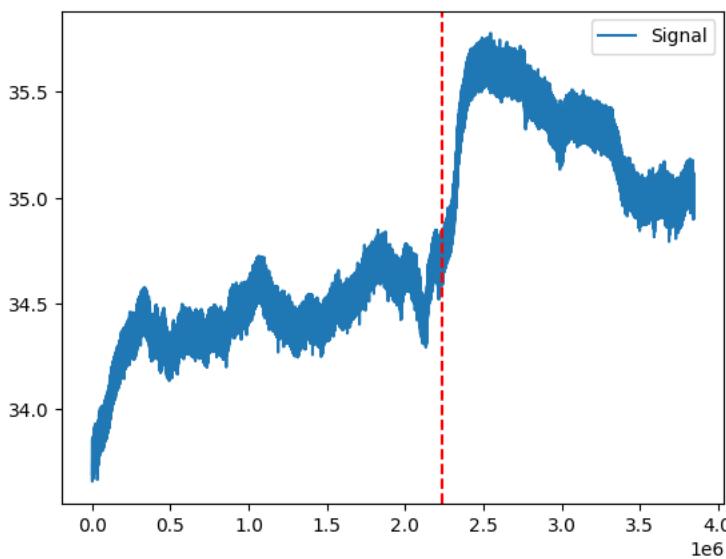
```



```

1 # Plot the signal event with the body temperature of the first subject
2 nk.events_plot(events, processed_data['TEMP'])
3 plt.savefig('/content/drive/MyDrive/stress-prediction/img/TEMP_signal.pdf')

```



```

1 # View all labels in the processed_data
2 processed_data.keys()

```

```
Index(['ECG_Raw', 'ECG_Clean', 'ECG_Rate', 'ECG_Quality', 'ECG_R_Peaks',
       'ECG_P_Peaks', 'ECG_P_Onsets', 'ECG_P_Offsets', 'ECG_Q_Peaks',
       'ECG_R_Onsets', 'ECG_R_Offsets', 'ECG_S_Peaks', 'ECG_T_Peaks',
       'ECG_T_Onsets', 'ECG_T_Offsets', 'ECG_Phase_Atrial',
       'ECG_Phase_Completion_Atrial', 'ECG_Phase_Ventricular',
       'ECG_Phase_Completion_Ventricular', 'RSP_Raw', 'RSP_Clean',
       'RSP_Amplitude', 'RSP_Rate', 'RSP_RVT', 'RSP_Phase',
       'RSP_Phase_Completion', 'RSP_Symmetry_PeakTrough',
       'RSP_Symmetry_RiseDecay', 'RSP_Peaks', 'RSP_Troughs', 'EDA_Raw',
       'EDA_Clean', 'EDA_Tonic', 'EDA_Phasic', 'SCR_Onsets', 'SCR_Peaks',
       'SCR_Height', 'SCR_Amplitude', 'SCR_RiseTime', 'SCR_Recovery',
       'SCR_RecoveryTime', 'RSA_P2T', 'RSA_Gates', 'TEMP'],
      dtype='object')
```

```
1 # Correlation Matrix of all features in the processed_data dataset
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 corr = processed_data.corr()
6 plt.figure(figsize=(16,10))
7 sns.heatmap(corr);
```

```

1 # From the correlation matrix the following features look differentiable
2 # and not highly correlated with each other
3
4 # RSP_Rate: Respiration rate
5 # EDA_Phasic: The rapid/sudden changes in the EDA response
6 # ECG_Rate: Heart rate
7 # TEMP: body temperature
8
9 processed_data_subset = processed_data[['RSP_Rate',
10                                'EDA_Phasic',
11                                'ECG_Rate',
12                                'TEMP']]
13 processed_data_subset
14

```

	RSP_Rate	EDA_Phasic	ECG_Rate	TEMP
0	14.481092	-0.002589	91.999226	33.695862
1	14.481092	-0.002616	91.999226	33.741333
2	14.481092	-0.002641	91.999226	33.717072
3	14.481092	-0.002665	91.999226	33.741333
4	14.481092	-0.002688	91.999226	33.747406
...
3847195	9.690817	-0.000035	112.299465	35.015808
3847196	9.690817	-0.000032	112.299465	35.018921
3847197	9.690817	-0.000029	112.299465	35.020447
3847198	9.690817	-0.000027	112.299465	34.932495
3847199	9.690817	-0.000024	112.299465	34.944824

3847200 rows × 4 columns

g as RC F

```

1 def modeling_data_process(subject_data):
2     """
3     Process and prepare the modeling data for further analysis.
4
5     Parameters:
6     subject_data (dict): Dictionary containing subject's physiological data.
7
8     Returns:
9     processed_data_subset (DataFrame): Processed and subsetted modeling data.
10    """
11    import warnings
12    warnings.filterwarnings("ignore")
13
14    # Perform bio-processing on the physiological signals (EDA, RSP, ECG) using NeuroKit2
15    processed_data, info = nk.bio_process(edap=subject_data['EDA'].reshape(-1),
16                                         rsp=subject_data['Resp'].reshape(-1),
17                                         ecg=subject_data['ECG'].reshape(-1),
18                                         sampling_rate=700)
19
20    # Add the 'Temp' signal to the processed data
21    processed_data['Temp'] = pd.DataFrame.from_dict(subject_data['Temp'])
22
23    # Subset the processed data to select specific features of interest
24    processed_data_subset = processed_data[['RSP_Rate', 'EDA_Phasic', 'ECG_Rate', 'Temp']]
25
26    # Add the labels to the processed data
27    processed_data_subset['label'] = pd.DataFrame.from_dict(subject_data['labels'])
28
29    # Return the processed and subsetted data
30    return processed_data_subset
31
32
33 # Create a list of processed subject data
34 model_data = [modeling_data_process(subject) for subject in wesad_data_li]
35
36
37 # Inspect first subject of preprocessed data
38 model_data[0].info()

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3847200 entries, 0 to 3847199
Data columns (total 5 columns):
 #   Column      Dtype  
--- 
 0   RSP_Rate    float64
 1   EDA_Phasic  float64
 2   ECG_Rate    float64
 3   Temp        float32
 4   label       int32  
dtypes: float32(1), float64(3), int32(1)
memory usage: 117.4 MB

1 # Save the preprocessed data for modeling notebook use
2
3 import os
4 import joblib
5
6 # Define the directory path
7 directory = "data/pickle"
8
9 # Create the directory if it doesn't exist
10 if not os.path.exists(directory):
11     os.makedirs(directory)
12
13 # Pickle and save preprocessed data
14 joblib.dump(model_data, os.path.join(directory, "WESAD_model_data.pickle"))

['data/pickle/WESAD_model_data.pickle']

```

▼ Preprocessed Data EDA

The data being used in this section is the preprocessed data where specific features from the physiological data has been extracted. Further statistical features will be extracted to produce the final dataset for model fitting.

After loading the data, this notebook will prepare the data for modeling by splitting it into training, validation, and testing sets, and then normalize the data to ensure that our model is able to learn from it effectively. We'll also use a class weight to give the minority class a weight boost to address class imbalance.

Finally, we'll train and evaluate our model on the testing set and compute various metrics such as accuracy, precision, recall, and F1-score to measure its performance.

```

1 # Import required libraries
2 import os
3 import pickle
4 import joblib
5
6 # Mount Google Drive to access data
7 from google.colab import drive
8 drive.mount('/content/drive', force_remount=True)
9
10 # Change working directory to the main file directory on Google Drive
11 os.chdir('/content/drive/MyDrive/stress-prediction')
12
13 # Define the path of the data directory on Google Drive
14 PATH = os.path.join(os.path.abspath(os.getcwd()), 'data')
15
16 # Load the pickled data using joblib
17 data = joblib.load(f'{PATH}/pickle/WESAD_model_data.pickle')

Mounted at /content/drive

```

▼ Exploratory Data Analysis

Now that we have loaded our pickled data, let's take a look at its structure. The 'data' variable is a list of two elements: a pandas dataframe and a numpy array. We can access the dataframe using `data[0]`. To get information about the first subject's preprocessed data.

```

1 # Structure of single dataframe in the list
2 data[0].info()

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3847200 entries, 0 to 3847199
Data columns (total 5 columns):
 #   Column      Dtype  
--- 
 0   RSP_Rate    float64 
 1   EDA_Phasic  float64 
 2   ECG_Rate    float64 
 3   Temp        float32 
 4   label        int32  
dtypes: float32(1), float64(3), int32(1)
memory usage: 117.4 MB

1 # Find the size and length of each dataframe
2 import pandas as pd
3
4 # Calculate duration of each dataframe in minutes
5 for i, df in enumerate(data):
6     duration = len(df) / (700 * 60)
7     df['duration'] = duration
8
9 # Create a table showing the duration of each dataframe
10 df_lengths = pd.DataFrame({'Subject': [f"{i}" for i in range(len(data))],
11                             'Observations': [len(df) for df in data],
12                             '(min)': [round(df['duration'][0]) for df in data]})
13 #print(df_lengths.to_string(index=False))
14 df_lengths.head(15)

```

	Subject	Observations	(min)
0	0	3847200	92
1	1	3883600	92
2	2	3875900	92
3	3	3663100	87
4	4	3941700	94
5	5	3676400	88
6	6	4144000	99
7	7	4255300	101
8	8	4545100	108
9	9	4496100	107
10	10	4380600	104
11	11	4949700	118
12	12	3666600	87
13	13	3826200	91
14	14	3656100	87

▼ Statistical Feature Engineering

We will compute statistical features that capture the patterns in the data that are relevant to predicting stress. Specifically, we will compute the mean and standard deviation of EDA over different time intervals using a rolling window function.

To compute these features, we define a function called `compute_features`. This function takes as input a dataframe and a list of columns to exclude from feature computation. It computes rolling mean and standard deviation features for all columns except those in the `exclude_cols` list. We define two window sizes, 60 seconds and 300 seconds, which correspond to intervals of one and five minutes, respectively.

We apply the `compute_features` function to each dataframe in the list using a list comprehension, creating a new list of dataframes with the computed features. Finally, we drop any rows with missing values in the resulting dataframes to prepare them for modeling.

```

1 # Define a function to compute mean, standard deviation of past 1 and 5 min
2 def compute_features(df, Hz_sampled, exclude_cols=[]):
3     # Create a copy of the input DataFrame
4     df = df.copy()
5
6     # Compute rolling mean and standard deviation features for all columns

```

```

7     # except those in exclude_cols
8     windows = [int(60 * Hz_sampled), int(300 * Hz_sampled)]
9
10    for col in df.columns:
11        if col not in exclude_cols:
12            for window in windows:
13                df[f'{col}_mean_{window // Hz_sampled}s'] = df[col].rolling(window=window).mean()
14                df[f'{col}_std_{window // Hz_sampled}s'] = df[col].rolling(window=window).std()
15
16    # Drop rows with missing values
17    df.dropna(inplace=True)
18    return df
19
20 # Apply the function to each dataframe in the list
21 data_frames_list = [compute_features(df, 700, ['label', 'duration']) for df in data]
22

```

```

1 def trim_and_chop(df, Hz_sampled, duration):
2     """
3         Trim the length of each subject dataframe to be divisible by
4         the desired sequence length, and remove any excess rows.
5
6         Args:
7             df (DataFrame): The subject's raw dataframe
8             Hz_sampled (int): The frequency of the data
9             duration (int): The desired duration of each sequence in seconds
10
11        Returns:
12            df_trimmed (DataFrame): The trimmed dataframe
13        """
14
15        # Calculate the desired length of each sequence
16        target_length = Hz_sampled * duration
17
18        # Check if the length of the dataframe is already divisible by the target length
19        remainder = len(df) % target_length
20
21        if remainder != 0:
22            # Calculate the number of rows to trim from the top and bottom
23            top_trim = remainder // 2
24            bottom_trim = remainder - top_trim
25
26            # Trim the dataframe
27            df_trimmed = df.iloc[top_trim: -bottom_trim]
28            return df_trimmed
29        else:
30            return df
31
32 # Apply function to each dataframe in the list
33 trimmed_df_list = [trim_and_chop(df, 700, (5*60)) for df in data_frames_list]

```

```

1 # Display all the new features
2 trimmed_df_list[0].info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 3570000 entries, 243599 to 3813598
Data columns (total 22 columns):
 #   Column           Dtype  
---  --  
0   RSP_Rate          float64
1   EDA_Phasic        float64
2   ECG_Rate          float64
3   Temp              float32
4   label              int32  
5   duration          float64
6   RSP_Rate_mean_60s float64
7   RSP_Rate_std_60s  float64
8   RSP_Rate_mean_300s float64
9   RSP_Rate_std_300s float64
10  EDA_Phasic_mean_60s float64
11  EDA_Phasic_std_60s float64
12  EDA_Phasic_mean_300s float64
13  EDA_Phasic_std_300s float64
14  ECG_Rate_mean_60s  float64
15  ECG_Rate_std_60s  float64
16  ECG_Rate_mean_300s float64
17  ECG_Rate_std_300s  float64
18  Temp_mean_60s     float64
19  Temp_std_60s      float64

```

```

20 Temp_mean_300s      float64
21 Temp_std_300s       float64
dtypes: float32(1), float64(20), int32(1)
memory usage: 599.2 MB

```

```
1 trimmed_df_list[0].describe(include='all').T
```

	count	mean	std	min	25%	50%	75%	max
RSP_Rate	3570000.0	1.461197e+01	5.467430e+00	2.319673	10.168142	1.500523e+01	18.249221	51.660517
EDA_Phasic	3570000.0	1.652568e-06	5.241135e-03	-0.055904	-0.001296	-6.506016e-05	0.001155	0.092057
ECG_Rate	3570000.0	9.173783e+01	1.577418e+01	53.915276	79.151481	9.108286e+01	102.182506	152.727273
Temp	3570000.0	3.482161e+01	4.541593e-01	34.136749	34.414764	3.464474e+01	35.306641	35.778046
label	3570000.0	1.421569e-01	3.492110e-01	0.000000	0.000000	0.000000e+00	0.000000	1.000000
duration	3570000.0	9.160000e+01	7.389645e-13	91.600000	91.600000	9.160000e+01	91.600000	91.600000
RSP_Rate_mean_60s	3570000.0	1.463973e+01	3.476750e+00	5.527036	12.206068	1.526531e+01	17.405408	22.693054
RSP_Rate_std_60s	3570000.0	3.843627e+00	1.723224e+00	0.875032	2.514544	3.613787e+00	4.842836	9.056521
RSP_Rate_mean_300s	3570000.0	1.478697e+01	2.597034e+00	7.838479	13.046097	1.558837e+01	17.080891	18.760963
RSP_Rate_std_300s	3570000.0	4.612333e+00	1.121828e+00	2.299659	3.677238	4.585956e+00	5.536561	7.192086
EDA_Phasic_mean_60s	3570000.0	7.119763e-08	2.339368e-04	-0.002361	-0.000035	1.647863e-07	0.000033	0.002526
EDA_Phasic_std_60s	3570000.0	3.549953e-03	3.844122e-03	0.000874	0.001402	1.858297e-03	0.003403	0.021228
EDA_Phasic_mean_300s	3570000.0	1.068569e-08	4.862331e-05	-0.000496	-0.000007	5.467897e-08	0.000007	0.000516
EDA_Phasic_std_300s	3570000.0	3.836964e-03	3.556489e-03	0.001026	0.001583	2.035786e-03	0.004704	0.014766
ECG_Rate_mean_60s	3570000.0	9.183885e+01	1.340504e+01	66.549417	79.042463	9.216362e+01	100.697578	131.021011
ECG_Rate_std_60s	3570000.0	7.667732e+00	2.994821e+00	2.507606	5.391551	7.175648e+00	9.490561	17.599181
ECG_Rate_mean_300s	3570000.0	9.224123e+01	1.198909e+01	74.696954	80.164089	9.198639e+01	100.569761	119.287726
ECG_Rate_std_300s	3570000.0	9.614290e+00	2.869071e+00	5.112240	7.046066	9.069524e+00	11.600488	19.363593
Temp_mean_60s	3570000.0	3.481724e+01	4.538684e-01	34.234607	34.410448	3.463725e+01	35.311926	35.631838
Temp_std_60s	3570000.0	3.756672e-02	1.260727e-02	0.026510	0.031285	3.445325e-02	0.038565	0.129836
Temp_mean_300s	3570000.0	3.479784e+01	4.570649e-01	34.087082	34.391481	3.460227e+01	35.305932	35.607152
Temp_std_300s	3570000.0	7.382311e-02	5.542976e-02	0.033025	0.044111	5.705469e-02	0.074091	0.332044

```

1 ## Include Subject for each dataframe to prepare for concatenation
2 import warnings
3 warnings.filterwarnings("ignore")
4 for i, df in enumerate(trimmed_df_list):
5     # Set subject as categorical
6     df['subject'] = i
7     df['subject'] = df['subject'].astype('category')

```

▼ Modeling Data Generation

Train-Test Split

We are splitting our dataset into training, validation, and testing sets. We are concatenating all of the dataframes together, down-sampling the data to a target frequency of 1Hz, and dropping any columns that have any missing values.

Then, we are splitting the data into training, validation, and testing sets by subject. This is important to ensure that we have a balanced distribution of subjects in each dataset. We are using a 60:20:20 split, where 60% of the subjects are in the training set, 20% are in the validation set, and 20% are in the testing set.

We are standardizing the data using a StandardScaler object, which scales the data to have a mean of 0 and a standard deviation of 1. This is important to ensure that all of the features are on the same scale and that no one feature dominates the others.

Finally, we are converting the subject column to a categorical variable, which is required for training the model with the fit method.

```

1 import pandas as pd
2 import numpy as np
3 from sklearn.model_selection import train_test_split
4 from sklearn.preprocessing import StandardScaler
5
6 # Concatenate the dataframes together
7 df = pd.concat(trimmed_df_list)
8
9 # Downsample the dataframe
10 sample_hz = 700
11 target_hz = 1
12 resample_factor = int(sample_hz / target_hz)
13 df = df.iloc[::resample_factor, :]
14
15 # Drop any columns that have any missing values
16 df = df.dropna(axis=1, how='any')
17
18 # Split the data into training, testing, and validation sets by subject
19 train_dfs = []
20 test_dfs = []
21 val_dfs = []
22
23 # Define the percentage split for training, validation, and testing data
24 train_pct = 0.6
25 val_pct = 0.2
26 test_pct = 0.2
27
28 # Get a list of unique subject IDs
29 subject_ids = np.unique(df['subject'])
30
31 # Randomly shuffle the subject IDs
32 np.random.seed(42)
33 np.random.shuffle(subject_ids)
34
35 # Calculate the number of subjects for each dataset
36 num_train = int(len(subject_ids) * train_pct)
37 num_val = int(len(subject_ids) * val_pct)
38 num_test = int(len(subject_ids)) - num_train - num_val
39
40 # Split the subject IDs into training, validation, and testing sets
41 train_subjects = subject_ids[:num_train]
42 val_subjects = subject_ids[num_train:num_train+num_val]
43 test_subjects = subject_ids[num_train+num_val:]
44
45 # Split the data into training, validation, and testing sets by subject ID
46 train_df = df[df['subject'].isin(train_subjects)]
47 val_df = df[df['subject'].isin(val_subjects)]
48 test_df = df[df['subject'].isin(test_subjects)]
49
50 # Standardize the data
51 scaler = StandardScaler()
52 cols_to_transform = [col for col in train_df.columns if col not in ['subject', 'label']]
53 train_df[cols_to_transform] = scaler.fit_transform(train_df[cols_to_transform])
54 test_df[cols_to_transform] = scaler.transform(test_df[cols_to_transform])
55 val_df[cols_to_transform] = scaler.transform(val_df[cols_to_transform])
56
57 # Set subject as categorical
58 train_df['subject'] = train_df['subject'].astype('category')
59 test_df['subject'] = test_df['subject'].astype('category')
60 val_df['subject'] = val_df['subject'].astype('category')

1 train_df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 48600 entries, 243599 to 3612349
Data columns (total 23 columns):
 #   Column            Non-Null Count  Dtype  
---  -- 
 0   RSP_Rate          48600 non-null   float64
 1   EDA_Phasic        48600 non-null   float64
 2   ECG_Rate          48600 non-null   float64
 3   Temp              48600 non-null   float64
 4   label              48600 non-null   int32  
 5   duration           48600 non-null   float64
 6   RSP_Rate_mean_60s 48600 non-null   float64
 7   RSP_Rate_std_60s  48600 non-null   float64

```

```

8   RSP_Rate_mean_300s    48600 non-null  float64
9   RSP_Rate_std_300s     48600 non-null  float64
10  EDA_Phasic_mean_60s   48600 non-null  float64
11  EDA_Phasic_std_60s    48600 non-null  float64
12  EDA_Phasic_mean_300s  48600 non-null  float64
13  EDA_Phasic_std_300s   48600 non-null  float64
14  ECG_Rate_mean_60s     48600 non-null  float64
15  ECG_Rate_std_60s      48600 non-null  float64
16  ECG_Rate_mean_300s    48600 non-null  float64
17  ECG_Rate_std_300s     48600 non-null  float64
18  Temp_mean_60s         48600 non-null  float64
19  Temp_std_60s          48600 non-null  float64
20  Temp_mean_300s        48600 non-null  float64
21  Temp_std_300s         48600 non-null  float64
22  subject               48600 non-null  category
dtypes: category(1), float64(21), int32(1)
memory usage: 8.4 MB

```

```

1 # Display which subjects are in which dataset group
2 print(train_df['subject'].unique())
3 print(val_df['subject'].unique())
4 print(test_df['subject'].unique())

[0, 1, 2, 5, 8, 9, 11, 13, 14]
Categories (9, int64): [0, 1, 2, 5, ..., 9, 11, 13, 14]
[4, 7, 10]
Categories (3, int64): [4, 7, 10]
[3, 6, 12]
Categories (3, int64): [3, 6, 12]

1 # Total length of each dataset. Since validation and test datasets
2 # have less subjects, they will have less observations
3 print(train_df.shape)
4 print(val_df.shape)
5 print(test_df.shape)

(48600, 23)
(16500, 23)
(15000, 23)

```

▼ Label Class Imbalance

Class imbalance is a problem in machine learning where the distribution of classes in the training data is not balanced. In some cases, one class may have significantly fewer samples than the other. This can lead to a model that is biased towards the majority class and performs poorly on the minority class.

To address this issue, we can use class weight correction. Class weight correction assigns a weight to each class based on its frequency in the training data. The weight is higher for the minority class and lower for the majority class. This way, the model is encouraged to pay more attention to the minority class during training.

In this code section, we first check the balance of the two label cases (0 is neutral and 1 is stress) in our training data. We find that there is a class imbalance as the number of stress samples is significantly smaller than the number of non-stress samples.

We calculate the class weights by passing the training labels to the `compute_sample_weight` function and setting the `class_weight` parameter to `'balanced'`. This computes the weight for each class based on the inverse of its frequency in the training data.

Next, we define a dictionary that maps each class label to its corresponding weight. We set the weight for the minority (stress) class to be higher than the majority (non-stress) class.

Finally, we plot a bar graph to visualize the class imbalance in our training data. We can see that the stress class has significantly fewer samples than the non-stress class.

```

1 # Balance of the two label cases. 0 is neutral and 1 is stress
2 # Indicates class imbalance
3 train_df['label'].value_counts()

0    42610
1    5990
Name: label, dtype: int64

1 # Class weight correction
2 import matplotlib.pyplot as plt
3 import numpy as np
4

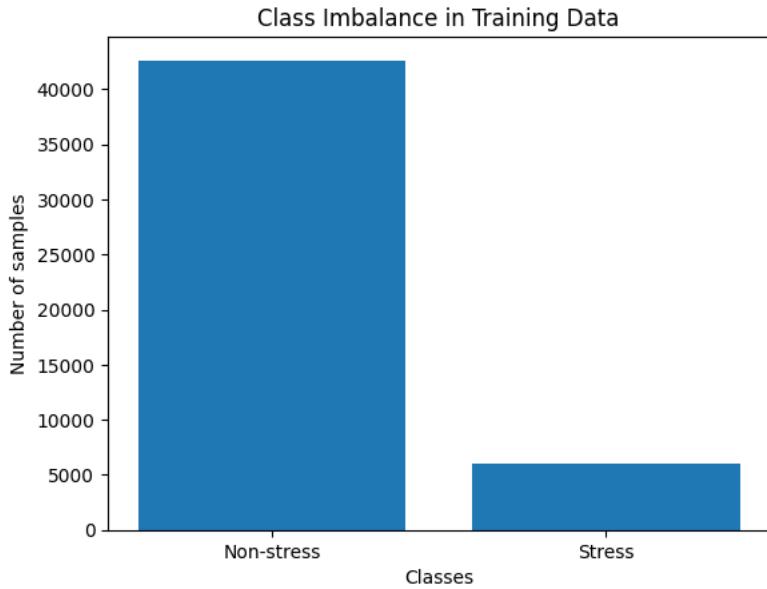
```

```

5 y_train = train_df['label']
6
7 # Calculate class counts
8 unique_classes, class_counts = np.unique(y_train, return_counts=True)
9 class_percentages = (class_counts / class_counts.sum()) * 100
10
11 # Define class weight dictionary
12 zero_count = class_counts[0]
13 one_count = class_counts[1]
14 class_weights_dict = {0: 1 / zero_count, 1: 1 / one_count}
15
16 # Print class percentages
17 for class_id, percentage in zip(unique_classes, class_percentages):
18     print(f"Class {class_id} ({'Non-stress' if class_id == 0 else 'Stress'}): {percentage:.2f}%")
19
20 # Create a bar plot for class counts
21 plt.bar(unique_classes, class_counts, tick_label=['Non-stress', 'Stress'])
22 plt.xlabel('Classes')
23 plt.ylabel('Number of samples')
24 plt.title('Class Imbalance in Training Data')
25 plt.savefig('img/imbalanced_labels.png', dpi=300, bbox_inches='tight')
26

```

Class 0 (Non-stress): 87.67%
 Class 1 (Stress): 12.33%



▼ Data Generator

The data generated from previous processing steps are structured to be fed into a neural network model. This is done using the `TimeseriesGenerator` class from the `keras` preprocessing module. The data is first divided into batches, with each batch containing a fixed number of sequences of a fixed length. The sequence length is determined based on the desired duration of the time series (in minutes), the sample frequency, and the target sequence length (in rows). For each set, the data is trimmed at the beginning and end to ensure that only full-length sequences are included.

Finally, the `TimeseriesGenerator` class is used to create a generator object for each set of data. The generator object iteratively returns batches of sequence data and corresponding labels, with the start index of each sequence in the batch determined by the generator based on the specified sequence length and stride.

The `TimeseriesGenerator` creates sequences of fixed length from the input data, with each sequence overlapping with the previous one by a certain amount. This overlap is controlled by the `stride` parameter. Stride refers to the number of rows to move forward between each sequence generated by the `TimeseriesGenerator`.

Here, `stride` is set to 1 minute, which means that each sequence in the generator will be 5 minutes long (the `duration` parameter), and will start 1 minute after the previous one ended. This means that each sequence in the generator will overlap with the previous one by 4 minutes.

Using a stride like this can help capture more information from the data, since each sequence will contain some data that was also present in the previous sequence.

```

1 from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
2 from tensorflow.keras.preprocessing.sequence import pad_sequences
3
4 # Define batch size and sequence length
5 duration = 5 # minutes
6 batch_size = 10
7 seq_length = target_hz * 60 * duration # full length of sequence
8
9 # Calculate the offset in rows
10 offset_minutes = 5
11 offset_rows = offset_minutes * target_hz * 60 # seconds
12
13 # Stride
14 stride = 1 * target_hz * 60 # 1 minute strides
15
16 # Extract values from DataFrame
17 train_data = train_df.drop(['label', 'subject', 'duration'], axis=1).values
18 val_data = val_df.drop(['label', 'subject', 'duration'], axis=1).values
19 test_data = test_df.drop(['label', 'subject', 'duration'], axis=1).values
20
21 # Trim front and end of the data
22 train_data = train_data[:-seq_length]
23 train_labels = train_df['label'][seq_length:].values
24 val_data = val_data[:-seq_length]
25 val_labels = val_df['label'][seq_length:].values
26 test_data = test_data[:-seq_length]
27 test_labels = test_df['label'][seq_length:].values
28
29 # Create data generators
30 train_gen = TimeseriesGenerator(train_data,
31                               train_labels,
32                               length=seq_length,
33                               batch_size=batch_size,
34                               shuffle=False)
35
36 val_gen = TimeseriesGenerator(val_data,
37                               val_labels,
38                               length=seq_length,
39                               batch_size=batch_size)
40
41 test_gen = TimeseriesGenerator(test_data,
42                               test_labels,
43                               length=seq_length,
44                               batch_size=batch_size)

```

```

1 # Display the shape of train_gen, val_gen, and test_gen
2 data_shape = train_gen[0][0].shape
3 print(data_shape)

```

(10, 300, 20)

Data Generator Shape

The shape (10, 300, 20) indicates a 3-dimensional tensor with 10 samples, each sample containing a sequence of 300 timesteps and 20 features per timestep. This means that the data is organized as 10 batches of sequences, where each sequence is a segment of 300 timesteps with 20 features per timestep. This format is suitable for input to neural network models that expect sequential data with a fixed number of timesteps and features per timestep.

▼ Modeling

We will explore different machine learning models, including 6 neural network models, a Random Forest model, and an XGBoost model, to predict stress based on time series data. Time series data can be challenging to model due to its sequential nature and the need to capture temporal dependencies. Therefore, we will focus on using neural networks, which have proven to be effective in modeling time series data. Neural networks are capable of automatically learning relevant features and patterns from the data, which can be used to make accurate predictions.

```

1 # Define helper function to plot the training and validation accuracy and loss
2 def plot_evaluation(model, file_name=None):
3     '''
4         Plots the Training and validation accuracy as well as the training and validation loss

```

```

5    over the number of epochs specified.
6
7    Args:
8        model: history object returned from the fit() method of the Keras model
9        file_name: filename to save the plot
10
11    Returns:
12        None
13        ''
14
15    # Import necessary libraries
16    import matplotlib.pyplot as plt
17
18    # Get training and validation accuracy and loss from the model history
19    acc = model.history['binary_accuracy']
20    val_acc = model.history['val_binary_accuracy']
21    loss = model.history['loss']
22    val_loss = model.history['val_loss']
23    epochs_range = range(1, len(loss) + 1)
24
25    # Create subplots for accuracy and loss
26    fig, ax = plt.subplots(1, 2, figsize=(24, 8))
27
28    # Plot training and validation accuracy
29    ax[0].plot(epochs_range, acc, 'g-', label='Training Accuracy')
30    ax[0].plot(epochs_range, val_acc, 'r-', label='Validation Accuracy')
31    ax[0].legend(loc='lower right')
32    ax[0].title.set_text('Training and Validation Accuracy')
33
34    # Plot training and validation loss
35    ax[1].plot(epochs_range, loss, 'g-', label='Training Loss')
36    ax[1].plot(epochs_range, val_loss, 'r-', label='Validation Loss')
37    ax[1].legend(loc='upper right')
38    ax[1].title.set_text('Training and Validation Loss')
39
40    # Customize the figure
41    fig.patch.set_facecolor('white')
42
43    # If a file path is provided, Save the plot
44    if save_file:
45        # Create directories if they don't exist
46        directory = os.path.dirname(file_name)
47        if not os.path.exists(directory):
48            os.makedirs(directory)
49        # Save the plot
50        plt.savefig(f'{file_name}')
51        # Display plot
52        plt.show()

```

```

1 # Define function to evaluate the model and plot the training and validation metrics
2 def evaluate_model(model_filepath, history_filepath, val_data, test_data, save_plot_path):
3     ''
4     Evaluate a Keras model and plot the training and validation metrics.
5
6     Args:
7         model_filepath: path to the Keras model file
8         history_filepath: path to the pickle file containing the training history of the model
9         val_data: validation data generator
10        test_data: testing data generator
11        save_plot_path: path to save the plot
12
13    Returns:
14        The trained Keras model and the training history object
15        ''
16
17    # Import necessary libraries
18    from tensorflow.keras.models import load_model
19    import joblib
20
21    # Load the trained Keras model
22    model = load_model(model_filepath)
23
24    # Evaluate the model on the validation set
25    model_eval_val = model.evaluate(val_data, use_multiprocessing=True, batch_size=batch_size)
26
27    # Load the training fit history from the pickle file
28    fit_history = joblib.load(history_filepath)

```

```

29 # Plot the training and validation metrics
30 plot_evaluation(fit_history, save_plot_path)
31
32 # Evaluate the model on the test set and print the validation and test accuracy
33 model_eval_test = model.evaluate(test_data, use_multiprocessing=True, batch_size=batch_size)
34 print(f'Validation Binary Accuracy is: {model_eval_val[1]:.2f}')
35 print(f'Testing Binary Accuracy is: {model_eval_test[1]:.2f}')
36
37 # Return the trained model and the training history object
38 return model, fit_history
39

```

```

1 # Callbacks Function
2 def callbacks(checkpoint_file, patience=3):
3     """
4     This function defines a set of callbacks that can be used during the training
5     of a neural network to monitor the performance and behavior of the training
6     process.
7     """
8
9     # import necessary libraries
10    from keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard
11
12    # Callback and Early Stopping
13    callbacks = [EarlyStopping(monitor='val_loss',
14                                verbose=1,
15                                patience=patience,
16                                restore_best_weights=True,
17                                mode='min'),
18
19                ModelCheckpoint(checkpoint_file,
20                                save_best_only=True,
21                                verbose=0),
22
23                TensorBoard(log_dir='./logs', histogram_freq=1,
24                                embeddings_freq=1,
25                                update_freq='epoch')]
26
27    return callbacks

```

```

1 def confusion_matrix_plot(model, data, save_file=None):
2
3     """
4     Computes and plots the confusion matrix for a given model and data.
5     It first makes predictions using the model on the given data, and then
6     computes the confusion matrix using sklearn.metrics.confusion_matrix.
7     The resulting matrix is then plotted using seaborn.heatmap. If a save_file
8     argument is provided, the plot is saved as an image.
9     """
10
11    import seaborn as sns
12    import matplotlib.pyplot as plt
13    from sklearn.metrics import confusion_matrix
14
15    # Predict the labels for the data using the trained model
16    y_pred_probs = model.predict(data)
17    y_pred = (y_pred_probs > 0.5).astype(int)
18
19    # Get the true labels for the data
20    y_true = []
21    for _, batch_labels in data:
22        y_true.extend(batch_labels)
23    y_true = np.array(y_true)
24
25    # Compute the confusion matrix using sklearn.metrics
26    cm = confusion_matrix(y_true, y_pred)
27
28    # Plot the confusion matrix using seaborn and matplotlib
29    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
30    plt.xlabel('Predicted')
31    plt.ylabel('True')
32
33    # Save the plot to a file if specified
34    if save_file:
35        # Create directories if they don't exist
36        directory = os.path.dirname(save_file)
37        if not os.path.exists(directory):

```

```

37         os.makedirs(directory)
38     # Save the plot
39     plt.savefig(f'{save_file}')
40
41     # Display the plot
42     plt.show()

1 # Classification Report Helper Function
2
3 def classification_report_output(model, data, save_file=None):
4     """
5         This function computes the classification report of a model's performance
6         on a given dataset. It first uses the trained model to predict the labels
7         of the data, and then computes the precision, recall, f1-score, and support
8         for each class using sklearn's classification_report function.
9         The output is a Pandas dataframe containing these metrics for each class.
10        If a file path is provided, the report is saved as a CSV file.
11        Finally, the report is printed to the console.
12        """
13    # Import necessary libraries
14    from sklearn.metrics import classification_report
15
16    # Make predictions on the data using the model
17    y_pred_probs = model.predict(data)
18    y_pred = (y_pred_probs > 0.5).astype(int)
19
20    # Extract the true labels from the data generator
21    y_true = []
22    for _, batch_labels in data:
23        y_true.extend(batch_labels)
24    y_true = np.array(y_true)
25
26    # Compute the classification report as a dictionary and convert to a Pandas dataframe
27    report = classification_report(y_true, y_pred, output_dict=True)
28    report_df = pd.DataFrame(report).transpose()
29
30    # If a file path is provided, save the report as a CSV file
31    if save_file:
32        # Create directories if they don't exist
33        directory = os.path.dirname(save_file)
34        if not os.path.exists(directory):
35            os.makedirs(directory)
36
37        report_df.to_csv(save_file, index=True)
38
39    # Print the report to the console
40    print(report_df)
41

```

▼ Baseline Model (Model 0)

This baseline neural network model is a simple Densely Connected Network that starts by flattening the time series data and then runs it through two Dense layers. The model has a single output for binary classification. The loss function used is binary_crossentropy and the optimizer used is Adam with a learning rate of 0.001. The model is evaluated using binary accuracy metric. This model is a quick and cheap way to test the data generator and the evaluation metrics.

Evaluation Metrics

The F1-score balances both precision and recall. Similar to the recall, the F1-score for class 0 remains relatively high across all datasets, indicating a good balance between precision and recall. However, the F1-score for class 1 varies, with the lowest value in the validation dataset. Overall, the model performs well in identifying non-stress instances but struggles to accurately detect stress instances. Further improvements are necessary to enhance the model's ability to classify stress accurately.

```

1 """
2 Model_0: Densely Connected Network
3 This neural network starts by flattening the data dn then runs it through two
4 Dense layers.
5 """
6
7 no_features = train_gen[0][0].shape[2]
8
9 from tensorflow import keras

```

```

10 from tensorflow.keras.optimizers import Adam
11 from keras.layers import Dense, Flatten
12
13 # Build Model 0
14 inputs = keras.Input(shape=(seq_length, no_features))
15 x = Flatten()(inputs)
16 x = Dense(16, activation='relu')(x)
17 outputs = Dense(1)(x)
18 model_0 = keras.Model(inputs, outputs, name='model_0')
19
20 # Compile Model 0
21 model_0.compile(loss='binary_crossentropy',
22                  optimizer=Adam(learning_rate=0.001),
23                  metrics=['binary_accuracy'])
24
25
26 model_0.summary()

```

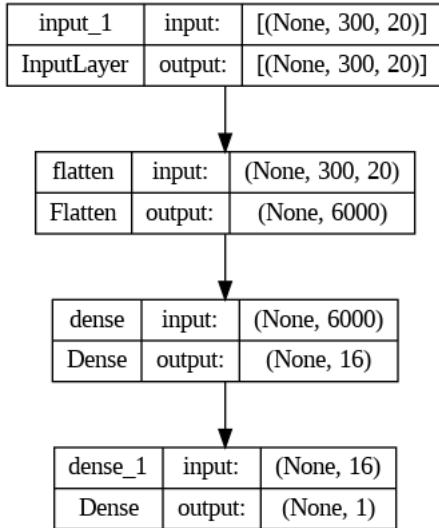
Model: "model_0"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[None, 300, 20]	0
flatten (Flatten)	(None, 6000)	0
dense (Dense)	(None, 16)	96016
dense_1 (Dense)	(None, 1)	17
<hr/>		
Total params: 96,033		
Trainable params: 96,033		
Non-trainable params: 0		

```

1 # Figure showing neural network
2 from keras.utils import plot_model
3
4 # assuming you have defined a model called "model"
5 plot_model(model_0, to_file='models/model_0.png', show_shapes=True, show_layer_names=True)

```



```

1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_0_filepath = 'models/model_dense.keras'
4 history_0_filepath = 'models/trainHistoryDict/model_0_dense_history.pkl'

1 # Fit the model to the training data
2 history_0 = model_0.fit(train_gen,
3                         validation_data=val_gen,
4                         shuffle=True,
5                         epochs=epochs,
6                         class_weight=class_weights_dict,
7                         callbacks=callbacks(model_0_filepath,

```

```

8                               patience=5),
9
10
11 # Pickle the Traning Fit History
12 with open(history_0_filepath, 'wb') as file_pi:
13     pickle.dump(history_0, file_pi)

Epoch 1/200
4800/4800 [=====] - 23s 4ms/step - loss: 1.6737e-04 - binary_accuracy: 0.7183 - val_loss: 7.0890 - val_binary_a
Epoch 2/200
4800/4800 [=====] - 18s 4ms/step - loss: 1.2191e-04 - binary_accuracy: 0.7301 - val_loss: 4.8773 - val_binary_a
Epoch 3/200
4800/4800 [=====] - 17s 4ms/step - loss: 1.0364e-04 - binary_accuracy: 0.8411 - val_loss: 6.3220 - val_binary_a
Epoch 4/200
4800/4800 [=====] - 17s 4ms/step - loss: 8.5044e-05 - binary_accuracy: 0.8369 - val_loss: 7.1377 - val_binary_a
Epoch 5/200
4800/4800 [=====] - 17s 4ms/step - loss: 7.9502e-05 - binary_accuracy: 0.8359 - val_loss: 5.5864 - val_binary_a
Epoch 6/200
4800/4800 [=====] - 18s 4ms/step - loss: 6.7938e-05 - binary_accuracy: 0.8659 - val_loss: 3.7385 - val_binary_a
Epoch 7/200
4800/4800 [=====] - 18s 4ms/step - loss: 1.0183e-04 - binary_accuracy: 0.9003 - val_loss: 3.8656 - val_binary_a
Epoch 8/200
4800/4800 [=====] - 17s 4ms/step - loss: 8.1530e-05 - binary_accuracy: 0.9133 - val_loss: 5.7047 - val_binary_a
Epoch 9/200
4800/4800 [=====] - 17s 4ms/step - loss: 7.1674e-05 - binary_accuracy: 0.9149 - val_loss: 6.2411 - val_binary_a
Epoch 10/200
4800/4800 [=====] - 17s 4ms/step - loss: 6.0856e-05 - binary_accuracy: 0.9246 - val_loss: 5.7594 - val_binary_a
Epoch 11/200
4784/4800 [=====>.] - ETA: 0s - loss: 6.5843e-05 - binary_accuracy: 0.9124Restoring model weights from the end of
4800/4800 [=====] - 17s 4ms/step - loss: 6.6190e-05 - binary_accuracy: 0.9124 - val_loss: 5.3326 - val_binary_a
Epoch 11: early stopping

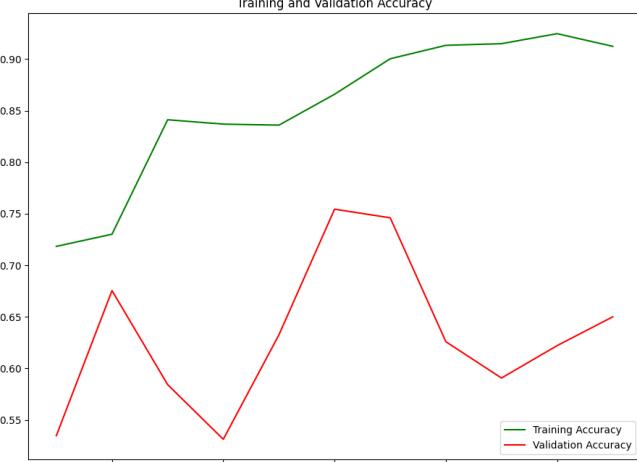
```

```

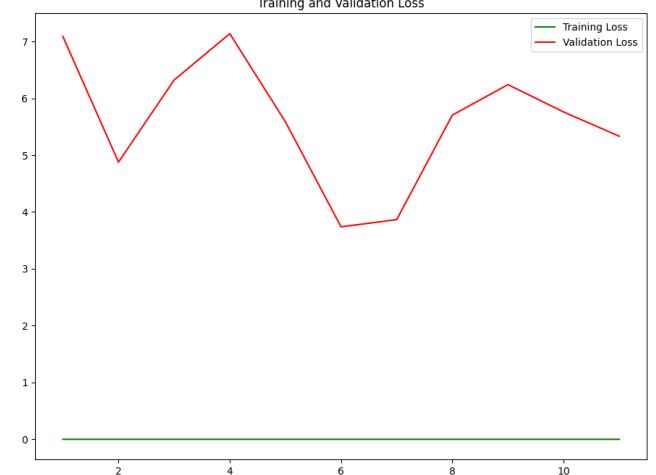
1 # Load and Evaluate Model
2 model_0, history_0 = evaluate_model(model_0_filepath,
3                                     history_0_filepath,
4                                     val_gen,
5                                     test_gen,
6                                     'models/evaluate/model_0_evaluation.pdf')

```

1590/1590 [=====] - 5s 3ms/step - loss: 3.7385 - binary_accuracy: 0.7544



Training and Validation Loss



1440/1440 [=====] - 6s 4ms/step - loss: 4.2598 - binary_accuracy: 0.7203

Validation Binary Accuracy is: 0.75

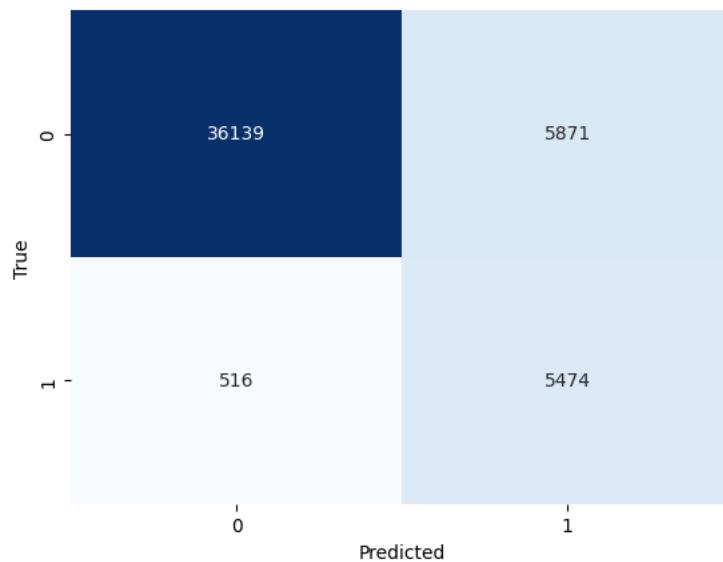
Testing Binary Accuracy is: 0.72

```

1 confusion_matrix_plot(model_0, train_gen, save_file='models/evaluate/model_0_train_confusion_matrix.pdf')
2 classification_report_output(model_0, train_gen, save_file=None)
3 confusion_matrix_plot(model_0, val_gen, save_file='models/evaluate/model_0_val_confusion_matrix.pdf')
4 classification_report_output(model_0, val_gen, save_file=None)
5 confusion_matrix_plot(model_0, test_gen, save_file='models/evaluate/model_0_test_confusion_matrix.pdf')
6 classification_report_output(model_0, test_gen, save_file=None)

```

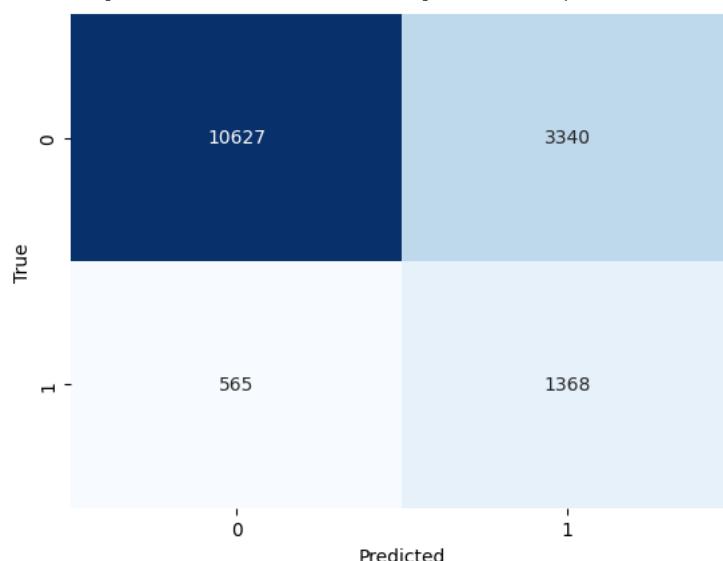
4800/4800 [=====] - 7s 1ms/step



4800/4800 [=====] - 7s 1ms/step

	precision	recall	f1-score	support
0	0.985923	0.860248	0.918808	42010.000000
1	0.482503	0.913856	0.631555	5990.000000
accuracy	0.866938	0.866938	0.866938	0.866938
macro avg	0.734213	0.887052	0.775181	48000.000000
weighted avg	0.923100	0.866938	0.882961	48000.000000

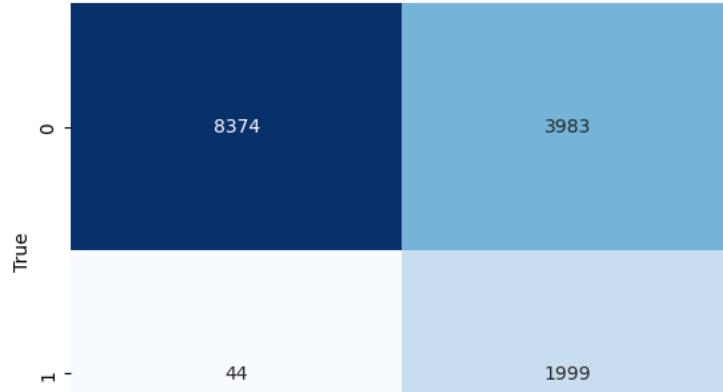
1590/1590 [=====] - 2s 1ms/step



1590/1590 [=====] - 2s 1ms/step

	precision	recall	f1-score	support
0	0.949518	0.760865	0.844787	13967.000000
1	0.290569	0.707708	0.411986	1933.000000
accuracy	0.754403	0.754403	0.754403	0.754403
macro avg	0.620043	0.734287	0.628387	15900.000000
weighted avg	0.869408	0.754403	0.792171	15900.000000

1440/1440 [=====] - 2s 1ms/step



Double-click (or enter) to edit

```

1 # Hyperparameter Tuning
2 def nn_lstm_hyperparameter_tuning(build_model_fn,
3                                     lstm_units_list,
4                                     weight_penalty_list,
5                                     learning_rates_list,
6                                     callback_filepath):
7     """
8         Performs hyperparameter tuning for an LSTM-based neural network model.
9         It takes in a function that constructs the model with given
10        hyperparameters, as well as lists of `lstm_units`, `weight_penalty`,
11        and `learning_rates` to loop through. For each combination of
12        hyperparameters, the function trains the model on the training set and
13        evaluates its performance on the validation set using the F1-score. The
14        function returns the best model, hyperparameters, and history of the
15        training process. The `callbacks` function is also called to save the
16        best model based on the validation loss.
17    """
18    import numpy as np
19    from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
20
21    # Placeholder for the best model and its corresponding hyperparameters
22    best_model = None
23    best_lstm_units = None
24    best_learning_rate = None
25    best_penalty = None
26    best_val_metric = -np.inf # Set initial value to negative infinity
27
28    # Loop through all combinations of LSTM units and learning rates
29    for lstm_units in lstm_units_list:
30        for learning_rate in learning_rates_list:
31            for weight_penalty in weight_penalty_list:
32                # Display the variables
33                print(f'lstm_units: {lstm_units}, learning_rate: {learning_rate}, weight_penalty: {weight_penalty}')
34
35                # Build model
36                model = build_model_fn(lstm_units, learning_rate)
37
38                # Set weight penalty
39                class_weights_dict[1] = class_weights_dict[1] * weight_penalty
40
41                # Train the model on the training set
42                history = model.fit(train_gen,
43                                     shuffle=False,
44                                     validation_data=val_gen,
45                                     epochs=epochs,
46                                     class_weight=class_weights_dict,
47                                     callbacks=callbacks(callback_filepath,
48                                         patience=5))
49
50                # Evaluate the model on the validation set
51                y_val_true = np.concatenate([y for x, y in val_gen], axis=0)
52                y_val_pred_probs = model.predict(val_gen)
53                y_val_pred = (y_val_pred_probs > 0.5).astype(int)
54
55                # Calculate the performance metric(s) of interest (e.g., F1-score)
56                val_f1_score = f1_score(y_val_true, y_val_pred)
57
58                # Update the best model and hyperparameters if the current model is better
59                if val_f1_score > best_val_metric:
60                    best_history = history
61                    best_val_metric = val_f1_score
62                    best_model = model
63                    best_lstm_units = lstm_units
64                    best_learning_rate = learning_rate
65                    best_penalty = weight_penalty
66
67                # Print the best LSTM units and learning rate
68                print(f"Best LSTM units: {best_lstm_units}")
69                print(f"Best learning rate: {best_learning_rate}")
70                print(f"Best weight_penalty: {best_penalty}")
71
72    return history, best_model, best_lstm_units, best_learning_rate, best_penalty
73

```

▼ Model 1

Model 1 includes a Long Short-Term Memory (LSTM) neural network. LSTM models are a type of recurrent neural network that is particularly suited for time series data due to their ability to remember and use past information while processing new data points. LSTMs use special units, called memory cells, that can store information over a long period of time and selectively forget or update that information based on the current input. This allows the model to capture long-term dependencies in the data, which is often important in time series forecasting or classification tasks. LSTMs have been shown to outperform traditional machine learning models on a variety of time series problems.

Compared to Model 0, which is a simple densely connected neural network, the LSTM model is expected to have better performance in modeling time series data because it can capture the sequential dependencies within the data.

Evaluation Metrics

Model 1 generally has similar precision and recall compared to Model 0 compared to Model 0. Model 1 exhibits similar F1-scores for the stress class in all datasets. Overall, Model 1 demonstrates similar performance in correctly identifying stress instances compared to Model 0.

```

1 # Model 1
2 def build_lstm_model(lstm_units, learning_rate):
3     from tensorflow import keras
4     from tensorflow.keras.optimizers import Adam
5     from keras.layers import Dense, LSTM
6
7     inputs = keras.Input(shape=(seq_length, no_features))
8     x = LSTM(units=lstm_units,
9               activation='tanh',
10              dropout=0.5)(inputs)
11    outputs = Dense(1, activation='sigmoid')(x)
12    model = keras.Model(inputs, outputs)
13    model.summary()
14
15    model.compile(loss='binary_crossentropy',
16                  optimizer=Adam(learning_rate=learning_rate),
17                  metrics=['binary_accuracy'])
18
19    return model

1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_1_filepath = 'models/model_1_lstm.keras'
4 history_1_filepath = 'models/trainHistoryDict/model_1_lstm_history.pkl'

1
2 lstm_units_list = [32]
3 learning_rates_list = [0.001]
4 class_weight_penalty= [1]
5 # Fit the model to the training data
6
7 history_1, model_1, best_lstm_units, best_learning_rate, best_penalty = nn_lstm_hyperparameter_tuning(
8                                         build_lstm_model,
9                                         lstm_units_list,
10                                        class_weight_penalty,
11                                        learning_rates_list,
12                                        model_1_filepath)
13
14 # Pickle the Traning Fit History
15 with open(history_1_filepath, 'wb') as file_pi:
16     pickle.dump(history_1, file_pi)

lstm_units: 32, learning_rate: 0.001, weight_penalty: 1
Model: "model"



| Layer (type)         | Output Shape    | Param # |
|----------------------|-----------------|---------|
| input_2 (InputLayer) | [None, 300, 20] | 0       |
| lstm (LSTM)          | (None, 32)      | 6784    |
| dense_2 (Dense)      | (None, 1)       | 33      |


=====
Total params: 6,817
Trainable params: 6,817
Non-trainable params: 0

Epoch 1/200
4800/4800 [=====] - 56s 11ms/step - loss: 1.5179e-05 - binary_accuracy: 0.8609 - val_loss: 1.0099 - val_binary_

```

```

Epoch 2/200
4800/4800 [=====] - 52s 11ms/step - loss: 1.1607e-05 - binary_accuracy: 0.8653 - val_loss: 1.1487 - val_binary_
Epoch 3/200
4800/4800 [=====] - 52s 11ms/step - loss: 1.0389e-05 - binary_accuracy: 0.8779 - val_loss: 1.2764 - val_binary_
Epoch 4/200
4800/4800 [=====] - 53s 11ms/step - loss: 9.6219e-06 - binary_accuracy: 0.8962 - val_loss: 1.4549 - val_binary_
Epoch 5/200
4800/4800 [=====] - 52s 11ms/step - loss: 9.4354e-06 - binary_accuracy: 0.8927 - val_loss: 1.4923 - val_binary_
Epoch 6/200
4797/4800 [=====>..] - ETA: 0s - loss: 8.2412e-06 - binary_accuracy: 0.9193Restoring model weights from the end of
4800/4800 [=====] - 52s 11ms/step - loss: 8.2361e-06 - binary_accuracy: 0.9193 - val_loss: 1.3391 - val_binary_
Epoch 6: early stopping
1590/1590 [=====] - 7s 4ms/step
Best LSTM units: 32
Best learning rate: 0.001
Best weight_penalty: 1

```

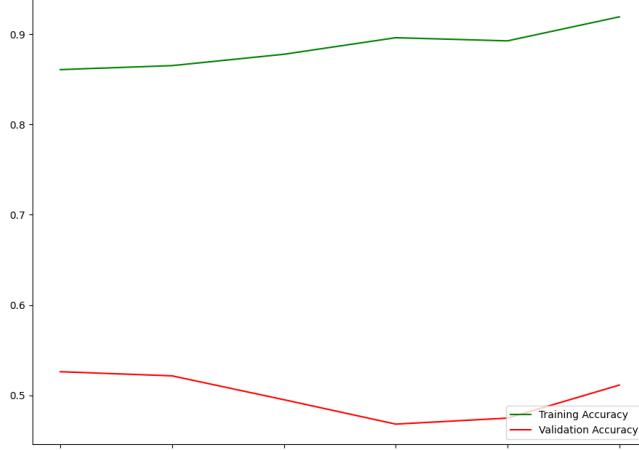
```

1 # Load and Evaluate Model
2 model_1, history_1 = evaluate_model(model_1_filepath,
3                                     history_1_filepath,
4                                     val_gen,
5                                     test_gen,
6                                     'models/evaluate/model_1_evaluation.pdf')

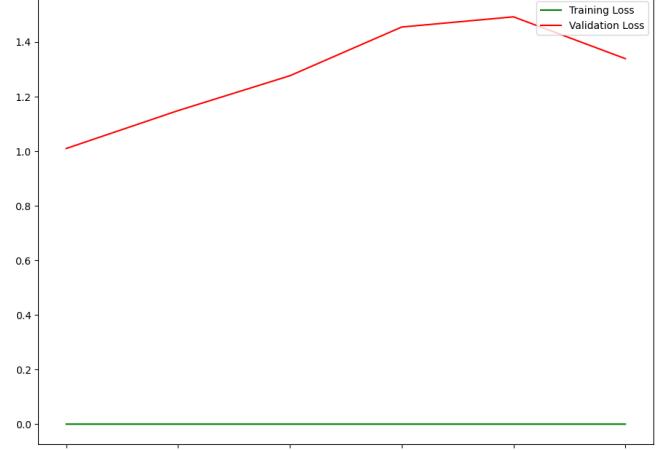
```

```
1590/1590 [=====] - 10s 6ms/step - loss: 1.0099 - binary_accuracy: 0.5262
```

Training and Validation Accuracy



Training and Validation Loss



```
1440/1440 [=====] - 8s 5ms/step - loss: 0.6930 - binary_accuracy: 0.7222
```

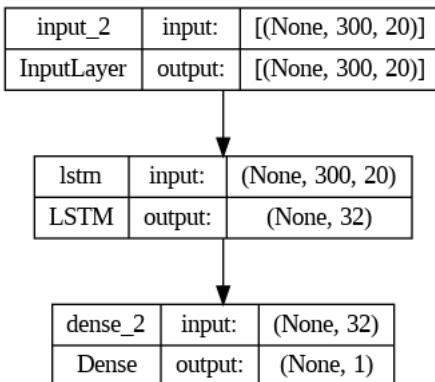
Validation Binary Accuracy is: 0.53

Testing Binary Accuracy is: 0.72

```

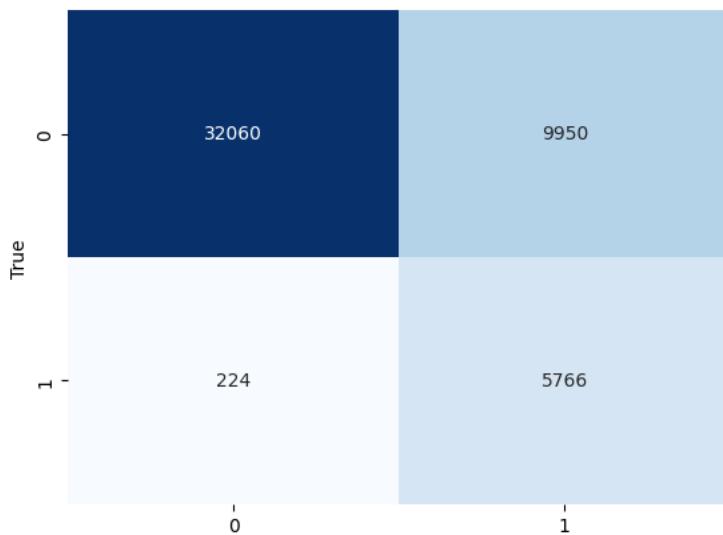
1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_1, to_file='models/model_1_arch.png', show_shapes=True, show_layer_names=True)

```



```
1 confusion_matrix_plot(model_1, train_gen, save_file='models/evaluate/model_1_train_confusion_matrix.pdf')
2 classification_report_output(model_1, train_gen, save_file=None)
3 confusion_matrix_plot(model_1, val_gen, save_file='models/evaluate/model_1_val_confusion_matrix.pdf')
4 classification_report_output(model_1, val_gen, save_file=None)
5 confusion_matrix_plot(model_1, test_gen, save_file='models/evaluate/model_1_test_confusion_matrix.pdf')
6 classification_report_output(model_1, test_gen, save_file=None)
```

```
4800/4800 [=====] - 21s 4ms/step
```



▼ Model 2

This is a deeper LSTM-based neural network architecture compared to model 1. The model consists of two LSTM layers with a dropout of 0.5 and an output layer. The first LSTM layer takes input of shape (300, 20) and returns a sequence of the same shape, which is then fed into the second LSTM layer. The second LSTM layer has half the number of units compared to the first LSTM layer. The output layer has a sigmoid activation function, which makes it suitable for binary classification problems. The dropout layers help to reduce overfitting, and the return_sequences parameter allows the model to process sequences of variable length.

Evaluation Metrics

Based on the results obtained, model 2 has not shown any significant improvement compared to models 0 and 1.

Model 2 architecture includes two LSTM layers with a lower number of LSTM units in the second layer compared to the first layer. The addition of the second LSTM layer is meant to provide more complex representations of the input data, which could help capture temporal patterns more effectively. Also, the use of the dropout layers can prevent overfitting and improve generalization.

Overall, Model 2 demonstrates lower performance in correctly identifying stress instances and maintaining similar performance for non-stress instances compared to Models 0 and 1. This suggests that Model 2 may be less effective in stress detection compared to the previous models.

```
1 # Model 2
2 def build_lstm_2_model(lstm_units, learning_rate):
3     import keras
4     from keras.layers import LSTM, Dense
5     from keras.models import Sequential
6     from tensorflow.keras.optimizers import Adam
7
8     # Build the model
9     inputs = keras.Input(shape=(seq_length, no_features))
10
11    # LSTM layer
12    x = LSTM(lstm_units,
13              activation='tanh',
14              input_shape=(seq_length, no_features),
15              return_sequences = True,
16              dropout=0.5)(inputs) # Dropout
17
18    # Second LSTM Layer
19    x = LSTM(lstm_units//2,
20              activation='tanh',
21              dropout=0.5)(x) # Dropout
22
23    outputs = Dense(1, 'sigmoid')(x)
24    model = keras.Model(inputs, outputs)
25
26    # Compile
27    model.compile(loss='binary_crossentropy',
28                  optimizer=Adam(learning_rate=learning_rate),
29                  metrics=['binary_accuracy'])
30
31    model.summary()
32
33    return model
```

```

1 model_2_filepath = 'models/model_2_multilSTM.keras'
2 history_2_filepath = 'models/trainHistoryDict/model_2_dense_history.pkl'
3 lstm_units_list = [64]
4 #learning_rates_list = [best_learning_rate]
5 #class_weight_penalty= [best_penalty]
    1440/1440 [=====] - 6s 4ms/step
1 # Fit the model to the training data
2 history_2, model_2, best_lstm_units, best_learning_rate, best_penalty = nn_lstm_hyperparameter_tuning(
3                                         build_lstm_2_model,
4                                         lstm_units_list,
5                                         class_weight_penalty,
6                                         learning_rates_list,
7                                         model_2_filepath)
8 # Pickle the Training Fit History
9 with open(history_2_filepath, 'wb') as file_pi:
10     pickle.dump(history_2, file_pi)

lstm_units: 64, learning_rate: 0.001, weight_penalty: 1
Model: "model_1"



| Layer (type)         | Output Shape      | Param # |
|----------------------|-------------------|---------|
| input_3 (InputLayer) | [(None, 300, 20)] | 0       |
| lstm_1 (LSTM)        | (None, 300, 64)   | 21760   |
| lstm_2 (LSTM)        | (None, 32)        | 12416   |
| dense_3 (Dense)      | (None, 1)         | 33      |


Total params: 34,209
Trainable params: 34,209
Non-trainable params: 0

Epoch 1/200
4800/4800 [=====] - 96s 19ms/step - loss: 1.9660e-05 - binary_accuracy: 0.7830 - val_loss: 0.9801 - val_binary_
Epoch 2/200
4800/4800 [=====] - 91s 19ms/step - loss: 1.6394e-05 - binary_accuracy: 0.8086 - val_loss: 1.3390 - val_binary_
Epoch 3/200
4800/4800 [=====] - 89s 19ms/step - loss: 1.1769e-05 - binary_accuracy: 0.8815 - val_loss: 1.0293 - val_binary_
Epoch 4/200
4800/4800 [=====] - 90s 19ms/step - loss: 1.1006e-05 - binary_accuracy: 0.9033 - val_loss: 1.1082 - val_binary_
Epoch 5/200
4800/4800 [=====] - 89s 19ms/step - loss: 1.1740e-05 - binary_accuracy: 0.8686 - val_loss: 1.0664 - val_binary_
Epoch 6/200
4800/4800 [=====] - 90s 19ms/step - loss: 9.9572e-06 - binary_accuracy: 0.9016 - val_loss: 0.9258 - val_binary_
Epoch 7/200
4800/4800 [=====] - 89s 19ms/step - loss: 1.1597e-05 - binary_accuracy: 0.8996 - val_loss: 0.9978 - val_binary_
Epoch 8/200
4800/4800 [=====] - 90s 19ms/step - loss: 1.0906e-05 - binary_accuracy: 0.9013 - val_loss: 1.2510 - val_binary_
Epoch 9/200
4800/4800 [=====] - 89s 19ms/step - loss: 1.0670e-05 - binary_accuracy: 0.8956 - val_loss: 1.4198 - val_binary_
Epoch 10/200
4800/4800 [=====] - 90s 19ms/step - loss: 8.4408e-06 - binary_accuracy: 0.9249 - val_loss: 1.5860 - val_binary_
Epoch 11/200
4800/4800 [=====] - ETA: 0s - loss: 9.5429e-06 - binary_accuracy: 0.9163Restoring model weights from the end of
4800/4800 [=====] - 90s 19ms/step - loss: 9.5429e-06 - binary_accuracy: 0.9163 - val_loss: 1.0405 - val_binary_
Epoch 11: early stopping
1590/1590 [=====] - 12s 7ms/step
Best LSTM units: 64
Best learning rate: 0.001
Best weight_penalty: 1

```

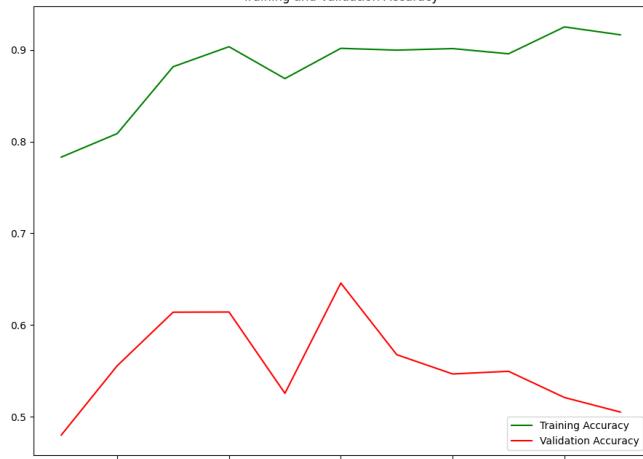
```

1 # Load and Evaluate Model
2 model_2, history_2 = evaluate_model(model_2_filepath,
3                                     history_2_filepath,
4                                     val_gen,
5                                     test_gen,
6                                     'models/evaluate/model_2_evaluation.pdf')

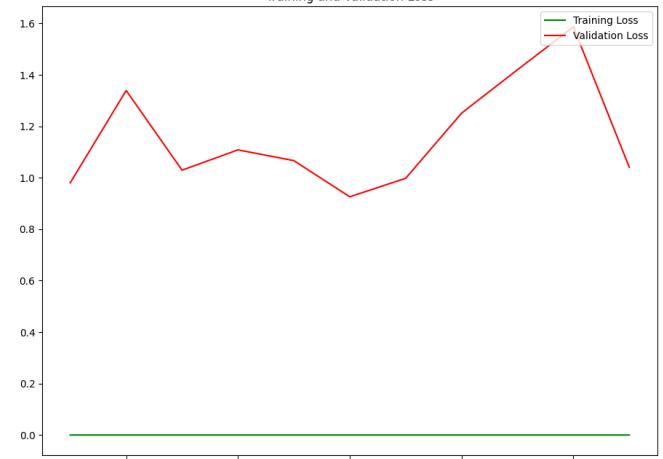
```

```
1590/1590 [=====] - 14s 8ms/step - loss: 0.9258 - binary_accuracy: 0.6458
```

Training and Validation Accuracy



Training and Validation Loss

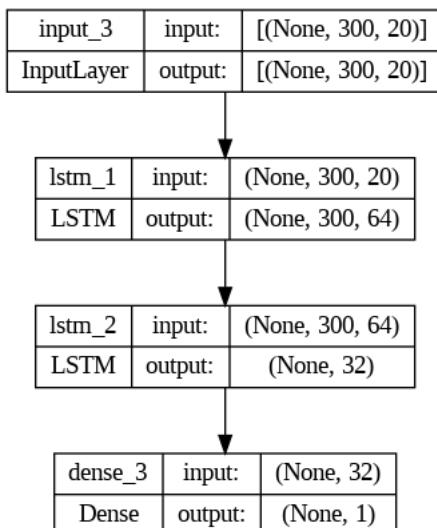


```
1440/1440 [=====] - 12s 8ms/step - loss: 0.7254 - binary_accuracy: 0.7335
```

Validation Binary Accuracy is: 0.65

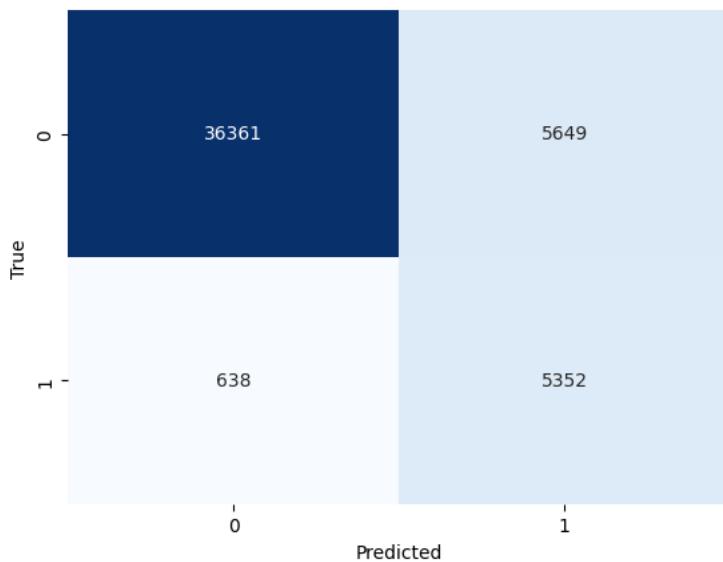
Testing Binary Accuracy is: 0.73

```
1 # Plot model visualization
2 from keras.utils import plot_model
3
4 plot_model(model_2, to_file='models/model_2_arch.png', show_shapes=True, show_layer_names=True)
```



```
1 confusion_matrix_plot(model_2, train_gen, save_file='models/evaluate/model_2_train_confusion_matrix.pdf')
2 classification_report_output(model_2, train_gen, save_file=None)
3 confusion_matrix_plot(model_2, val_gen, save_file='models/evaluate/model_2_val_confusion_matrix.pdf')
4 classification_report_output(model_2, val_gen, save_file=None)
5 confusion_matrix_plot(model_2, test_gen, save_file='models/evaluate/model_2_test_confusion_matrix.pdf')
6 classification_report_output(model_2, test_gen, save_file=None)
```

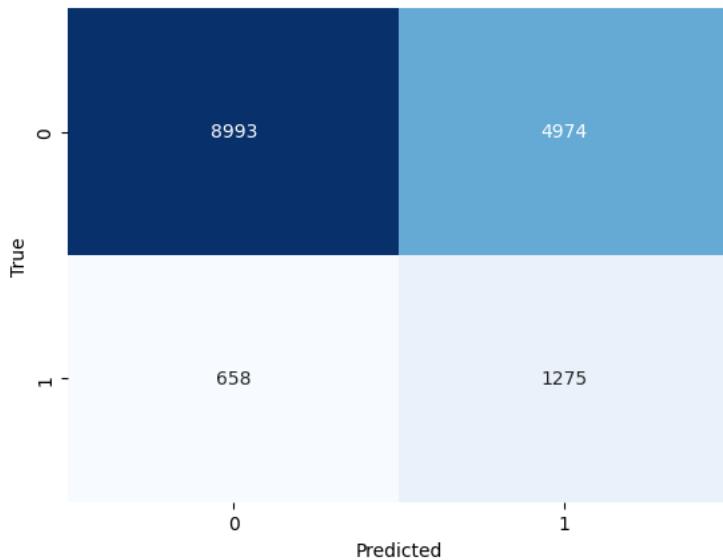
4800/4800 [=====] - 36s 7ms/step



4800/4800 [=====] - 36s 7ms/step

	precision	recall	f1-score	support
0	0.982756	0.865532	0.920427	42010.000000
1	0.486501	0.893489	0.629981	5990.000000
accuracy	0.869021	0.869021	0.869021	0.869021
macro avg	0.734629	0.879511	0.775204	48000.000000
weighted avg	0.920828	0.869021	0.884182	48000.000000

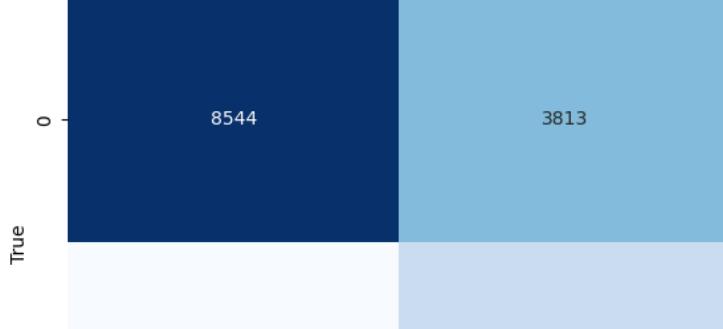
1590/1590 [=====] - 12s 8ms/step



1590/1590 [=====] - 12s 7ms/step

	precision	recall	f1-score	support
0	0.931821	0.643875	0.761538	13967.000000
1	0.204033	0.659596	0.311660	1933.000000
accuracy	0.645786	0.645786	0.645786	0.645786
macro avg	0.567927	0.651736	0.536599	15900.000000
weighted avg	0.843342	0.645786	0.706845	15900.000000

1440/1440 [=====] - 11s 7ms/step



▼ Model 3

The model architecture of Model 3 is similar to Model 2, with the addition of BatchNormalization layers after each LSTM layer. BatchNormalization is a technique that normalizes the inputs of each layer, making the training process more efficient and reducing the risk of overfitting.

Model 3 is expected to perform better, as it includes BatchNormalization layers to further improve the model's training stability and generalization ability.

Evaluation Metrics

Model 3 has lower precision for non-stress class compared to Models 0, 1, and 2. It also has worse performance according to F1-score of the stress class (class 1). Overall, Model 3 demonstrates lower performance compared to the previous models in terms of precision and F1-score and would not be effective in classifying for our business problem.

```
1 ## Model 3
2 def build_lstm_2_norm(lstm_units, learning_rate):
3     import keras
4     from keras.layers import LSTM, Dense, Dropout
5     from tensorflow.keras.optimizers import Adam
6     from keras.layers.normalization.batch_normalization_v1 import BatchNormalization
7
8     # Build the model
9     inputs = keras.Input(shape=(seq_length, no_features))
10
11    # LSTM layer
12    x = LSTM(lstm_units,
13              activation='tanh',
14              input_shape=(seq_length, no_features),
15              return_sequences=True,
16              dropout=0.5)(inputs) # Dropout
17    # Second LSTM Layer
18    x = BatchNormalization()(x)
19    x = LSTM(lstm_units//2,
20              activation='tanh',
21              return_sequences=False, # Set return_sequences=False
22              dropout=0.5)(x) # Dropout
23    x = BatchNormalization()(x)
24    # Dense layer for binary prediction
25    outputs = Dense(1, activation='sigmoid')(x)
26
27    model = keras.Model(inputs, outputs)
28
29    # Compile
30    model.compile(loss='binary_crossentropy',
31                  optimizer=Adam(learning_rate=learning_rate),
32                  metrics=['binary_accuracy'])
33
34    model.summary()
35
36    return model

1 model_3_filepath = 'models/model_3_multilSTM.keras'
2 history_3_filepath = 'models/trainHistoryDict/model_3_history.pkl'
3 #lstm_units_list = [32, 64, 128]
4 #learning_rates_list = [0.0001, 0.001, 0.01, 0.1]
5 #class_weight_penalty = [1, 2, 10, 100, 1000]

1 # Fit the model and tune hyperparameters
2 history_3, model_3, best_lstm_units, best_learning_rate, best_penalty = nn_lstm_hyperparameter_tuning(
3         build_lstm_2_norm,
4         lstm_units_list,
5         class_weight_penalty,
6         learning_rates_list,
7         model_3_filepath)
8 # Pickle the Training Fit History
9 with open(history_3_filepath, 'wb') as file_pi:
10     pickle.dump(history_3, file_pi)

lstm_units: 64, learning_rate: 0.001, weight_penalty: 1
Model: "model_2"
```

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[None, 300, 20]	0
lstm_3 (LSTM)	(None, 300, 64)	21760
batch_normalization (BatchN ormalization)	(None, 300, 64)	256
lstm_4 (LSTM)	(None, 32)	12416
batch_normalization_1 (BatchNormalization)	(None, 32)	128
dense_4 (Dense)	(None, 1)	33

Total params: 34,593
Trainable params: 34,401
Non-trainable params: 192

Epoch 1/200
4800/4800 [=====] - 106s 21ms/step - loss: 3.0316e-05 - binary_accuracy: 0.5828 - val_loss: 0.6252 - val_binary_accuracy: 0.4974
Epoch 2/200
4800/4800 [=====] - 100s 21ms/step - loss: 2.9850e-05 - binary_accuracy: 0.4974 - val_loss: 0.6841 - val_binary_accuracy: 0.4710
Epoch 3/200
4800/4800 [=====] - 99s 21ms/step - loss: 3.0261e-05 - binary_accuracy: 0.4710 - val_loss: 0.6906 - val_binary_accuracy: 0.5090
Epoch 4/200
4800/4800 [=====] - 99s 21ms/step - loss: 3.0447e-05 - binary_accuracy: 0.5090 - val_loss: 0.6739 - val_binary_accuracy: 0.5306
Epoch 5/200
4800/4800 [=====] - 99s 21ms/step - loss: 3.0754e-05 - binary_accuracy: 0.5306 - val_loss: 0.6891 - val_binary_accuracy: 0.5112
Epoch 6/200
4798/4800 [=====>..] - ETA: 0s - loss: 3.1130e-05 - binary_accuracy: 0.5110 Restoring model weights from the end of epoch 5
4800/4800 [=====] - 98s 20ms/step - loss: 3.1124e-05 - binary_accuracy: 0.5112 - val_loss: 0.6887 - val_binary_accuracy: 0.5112
Epoch 6: early stopping
1590/1590 [=====] - 12s 7ms/step
Best LSTM units: 64
Best learning rate: 0.001
Best weight_penalty: 1

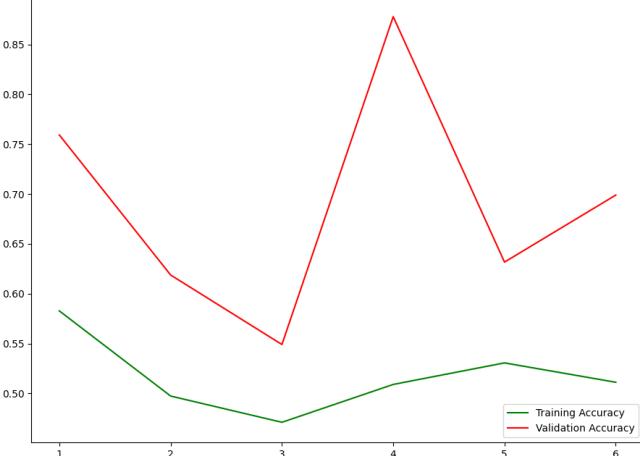
```

1 # Load and Evaluate Model
2 model_3, history_3 = evaluate_model(model_3_filepath,
3                                     history_3_filepath,
4                                     val_gen,
5                                     test_gen,
6                                     'models/evaluate/model_3_evaluation.pdf')

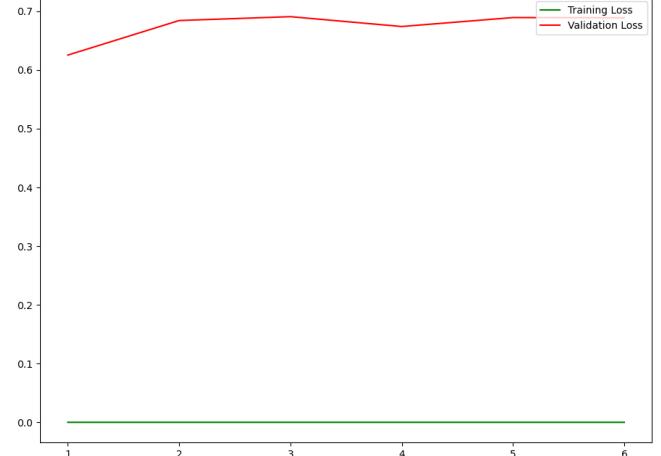
```

1590/1590 [=====] - 14s 8ms/step - loss: 0.6252 - binary_accuracy: 0.7592

Training and Validation Accuracy



Training and Validation Loss



1440/1440 [=====] - 12s 8ms/step - loss: 0.6834 - binary_accuracy: 0.5833

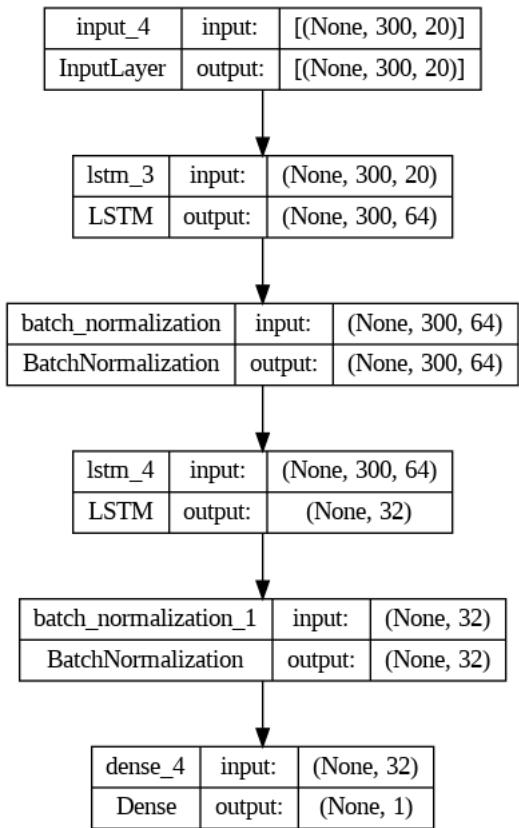
Validation Binary Accuracy is: 0.76

Testing Binary Accuracy is: 0.58

```

1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_3, to_file='models/model_3_arch.png', show_shapes=True, show_layer_names=True)

```

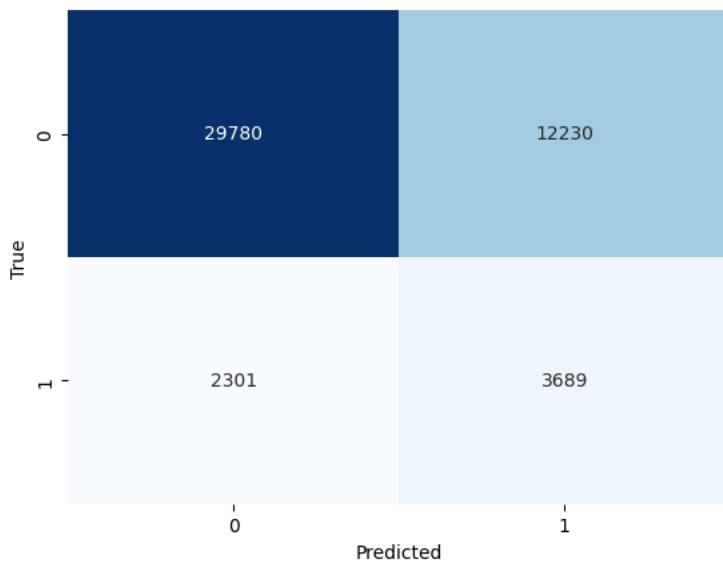


```

1 confusion_matrix_plot(model_3, train_gen, save_file='models/evaluate/model_3_train_confusion_matrix.pdf')
2 classification_report_output(model_3, train_gen, save_file=None)
3 confusion_matrix_plot(model_3, val_gen, save_file='models/evaluate/model_3_val_confusion_matrix.pdf')
4 classification_report_output(model_3, val_gen, save_file=None)
5 confusion_matrix_plot(model_3, test_gen, save_file='models/evaluate/model_3_test_confusion_matrix.pdf')
6 classification_report_output(model_3, test_gen, save_file=None)

```

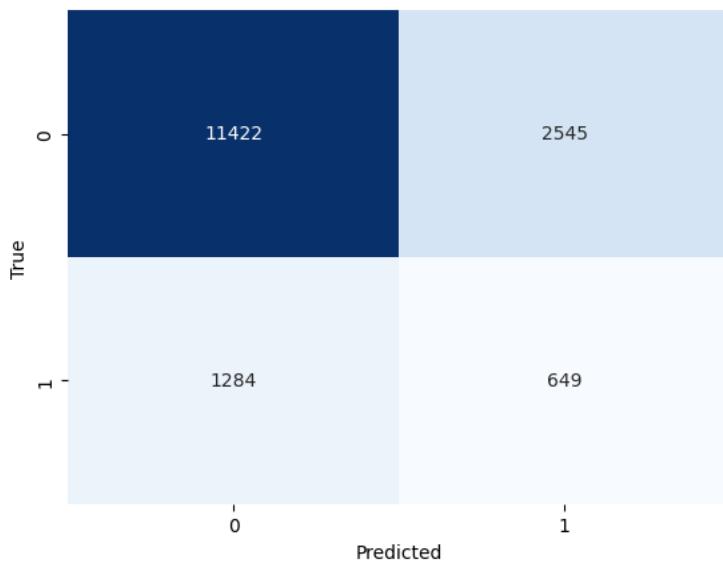
```
4800/4800 [=====] - 36s 7ms/step
```



```
4800/4800 [=====] - 36s 7ms/step
```

	precision	recall	f1-score	support
0	0.928275	0.708879	0.803876	42010.000000
1	0.231736	0.615860	0.336757	5990.000000
accuracy	0.697271	0.697271	0.697271	0.697271
macro avg	0.580005	0.662369	0.570316	48000.000000
weighted avg	0.841353	0.697271	0.745584	48000.000000

```
1590/1590 [=====] - 12s 8ms/step
```



```
1590/1590 [=====] - 12s 7ms/step
```

	precision	recall	f1-score	support
0	0.898945	0.817785	0.856447	13967.000000
1	0.203193	0.335748	0.253169	1933.000000
accuracy	0.759182	0.759182	0.759182	0.759182
macro avg	0.551069	0.576766	0.554808	15900.000000
weighted avg	0.814361	0.759182	0.783105	15900.000000

```
1440/1440 [=====] - 11s 7ms/step
```

Model 4

Model 4 is a combination of a convolutional layer and an LSTM layer. The convolutional layer is used to extract features from the input data. The output of the convolutional layer is then passed through a max pooling layer to reduce the dimensionality of the features. This process is repeated with another convolutional layer and max pooling layer. The output of the last max pooling layer is then fed into the LSTM layer, which can capture the temporal dependencies in the data.

The output of the LSTM layer is then flattened and passed through a dense layer with a ReLU activation function. A dropout layer is added to prevent overfitting, and the final output layer uses a sigmoid activation function for binary classification.

Evaluation

Model 4 performs poorly for both classes, as it predicts only the non-stress class (class 0) and ignores the stress class (class 1). It has lower precision, recall, and F1-score for the non-stress class compared to the other models. Model 4 is not able to effectively predict the stress instances.

```

1 # Model 4
2 def build_lstm_conv_lstm(lstm_units, learning_rate):
3     import keras
4     from keras.layers import LSTM, Dense, Dropout, Conv1D, MaxPooling1D, Flatten
5     from tensorflow.keras.optimizers import Adam
6
7     # Define the input shape
8     inputs = keras.Input(shape=(seq_length, no_features))
9
10    # Add a 1D convolutional layer to extract features
11    x = Conv1D(filters=lstm_units, kernel_size=9, activation='relu', padding='same')(inputs)
12    x = MaxPooling1D(pool_size=2)(x)
13    x = Conv1D(filters=lstm_units*2, kernel_size=3, activation='relu', padding='same')(x)
14    x = MaxPooling1D(pool_size=2)(x)
15
16    # LSTM layer
17    x = LSTM(lstm_units, activation='tanh', return_sequences=True)(x)
18
19    # Flatten the output
20    x = Flatten()(x)
21
22    # Dense layer
23    x = Dense(lstm_units//2, activation='relu')(x)
24
25    # Dropout layer
26    x = Dropout(0.5)(x)
27
28    # Add the output layer
29    outputs = Dense(1, activation='sigmoid')(x)
30
31    # Create the model
32    model = keras.Model(inputs=inputs, outputs=outputs)
33
34    # Compile the model
35    model.compile(loss='binary_crossentropy',
36                  optimizer=Adam(learning_rate),
37                  metrics=['binary_accuracy'])
38
39    model.summary()
40
41    return model
42

1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_4_filepath = 'models/model_4_conv_lstm.keras'
4 history_4_filepath = 'models/trainHistoryDict/model_4_history.pkl'
5 #lstm_units_list = [32, 64, 128]
6 #learning_rates_list = [0.01]

1 # Fit the model and tune hyperparameters
2 history_4, model_4, best_lstm_units, best_learning_rate, best_penalty = nn_lstm_hyperparameter_tuning(
3                                         build_lstm_conv_lstm,
4                                         lstm_units_list,
5                                         class_weight_penalty,
6                                         learning_rates_list,
7                                         model_4_filepath)
8 # Pickle the Training Fit History
9 with open(history_4_filepath, 'wb') as file_pi:
10    pickle.dump(history_4, file_pi)

lstm_units: 64, learning_rate: 0.001, weight_penalty: 1
Model: "model_3"



| Layer (type)                 | Output Shape      | Param # |
|------------------------------|-------------------|---------|
| input_5 (InputLayer)         | [(None, 300, 20)] | 0       |
| conv1d (Conv1D)              | (None, 300, 64)   | 11584   |
| max_pooling1d (MaxPooling1D) | (None, 150, 64)   | 0       |


```

```

)
conv1d_1 (Conv1D)           (None, 150, 128)      24704
max_pooling1d_1 (MaxPooling 1D) (None, 75, 128)      0
lstm_5 (LSTM)               (None, 75, 64)        49408
flatten_1 (Flatten)         (None, 4800)          0
dense_5 (Dense)             (None, 32)            153632
dropout (Dropout)           (None, 32)            0
dense_6 (Dense)             (None, 1)              33
=====

```

```

Total params: 239,361
Trainable params: 239,361
Non-trainable params: 0

```

```

Epoch 1/200
4800/4800 [=====] - 47s 8ms/step - loss: 2.7694e-05 - binary_accuracy: 0.6628 - val_loss: 0.6757 - val_binary_a
Epoch 2/200
4800/4800 [=====] - 38s 8ms/step - loss: 2.8915e-05 - binary_accuracy: 0.5525 - val_loss: 0.6772 - val_binary_a
Epoch 3/200
4800/4800 [=====] - 38s 8ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5448 - val_loss: 0.6775 - val_binary_a
Epoch 4/200
4800/4800 [=====] - 37s 8ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5431 - val_loss: 0.6776 - val_binary_a
Epoch 5/200
4800/4800 [=====] - 38s 8ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_a
Epoch 6/200
4794/4800 [=====>..] - ETA: 0s - loss: 2.8929e-05 - binary_accuracy: 0.5423Restoring model weights from the end of
4800/4800 [=====] - 37s 8ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_a
Epoch 6: early stopping
1590/1590 [=====] - 5s 3ms/step
Best LSTM units: 64
Best learning rate: 0.001
Best weight_penalty: 1

```

1 # Load and Evaluate Model

```

2 model_4, history_4 = evaluate_model(model_4_filepath,
3                                     history_4_filepath,
4                                     val_gen,
5                                     test_gen,
6                                     'models/evaluate/model_4_evaluation.pdf')

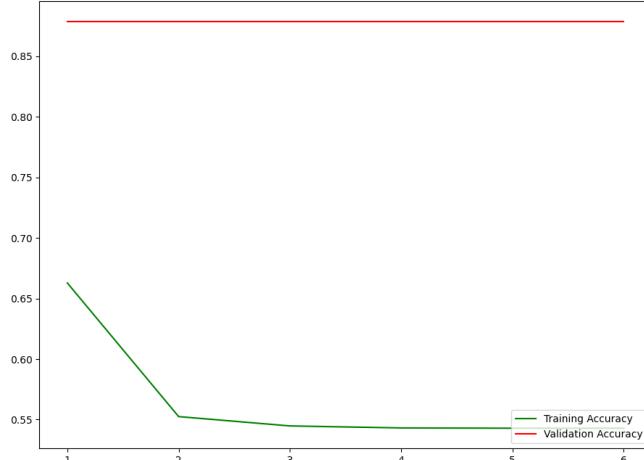
```

```

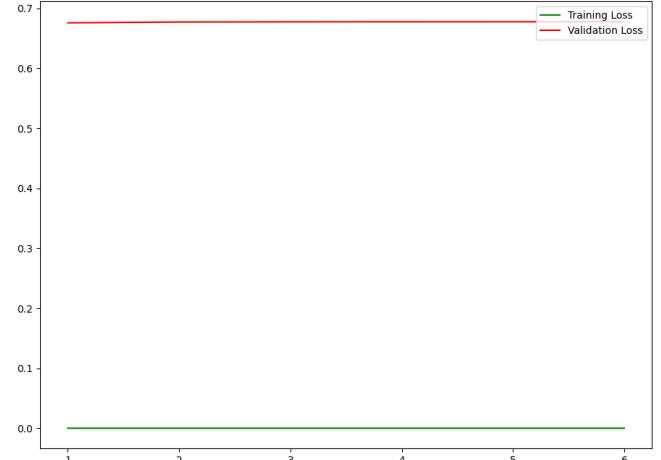
1590/1590 [=====] - 7s 4ms/step - loss: 0.6757 - binary_accuracy: 0.8784

```

Training and Validation Accuracy



Training and Validation Loss



```

1440/1440 [=====] - 6s 4ms/step - loss: 0.6767 - binary_accuracy: 0.8581

```

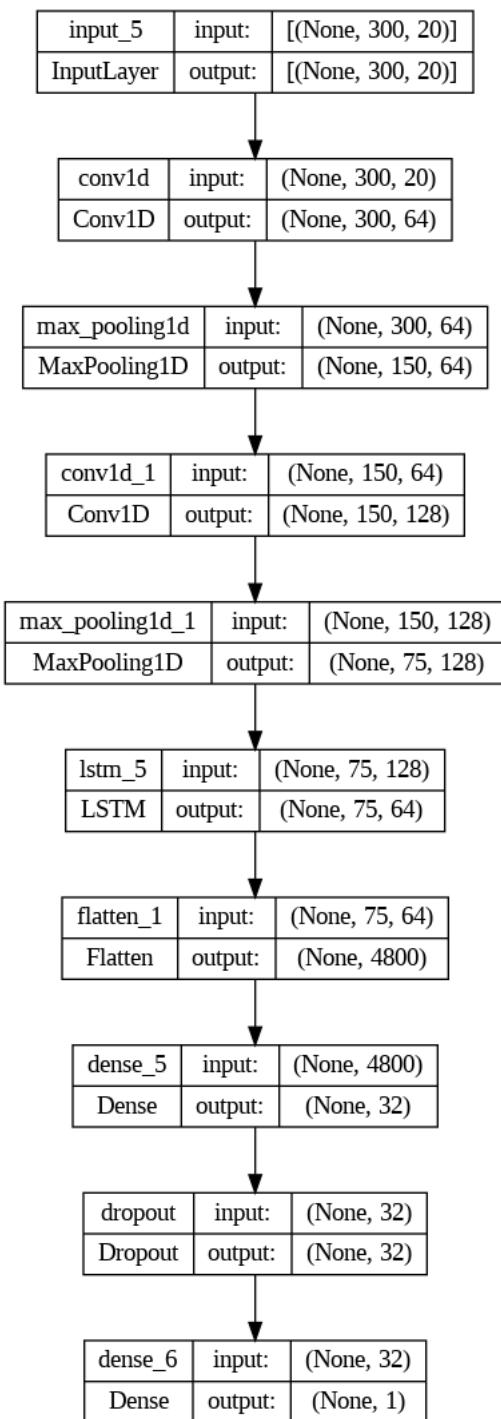
Validation Binary Accuracy is: 0.88

Testing Binary Accuracy is: 0.86

```

1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_4, to_file='models/model_4_arch.png', show_shapes=True, show_layer_names=True)

```

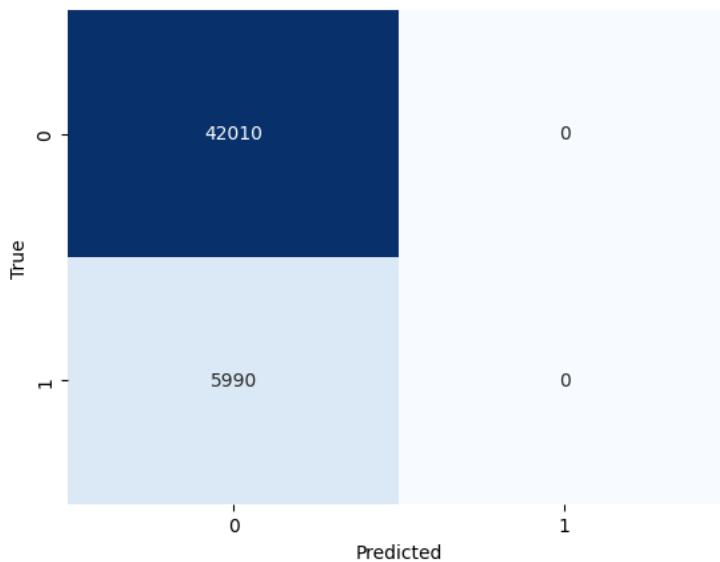


```

1 confusion_matrix_plot(model_4, train_gen, save_file='models/evaluate/model_4_train_confusion_matrix.pdf')
2 classification_report_output(model_4, train_gen, save_file=None)
3 confusion_matrix_plot(model_4, val_gen, save_file='models/evaluate/model_4_val_confusion_matrix.pdf')
4 classification_report_output(model_4, val_gen, save_file=None)
5 confusion_matrix_plot(model_4, test_gen, save_file='models/evaluate/model_4_test_confusion_matrix.pdf')
6 classification_report_output(model_4, test_gen, save_file=None)

```

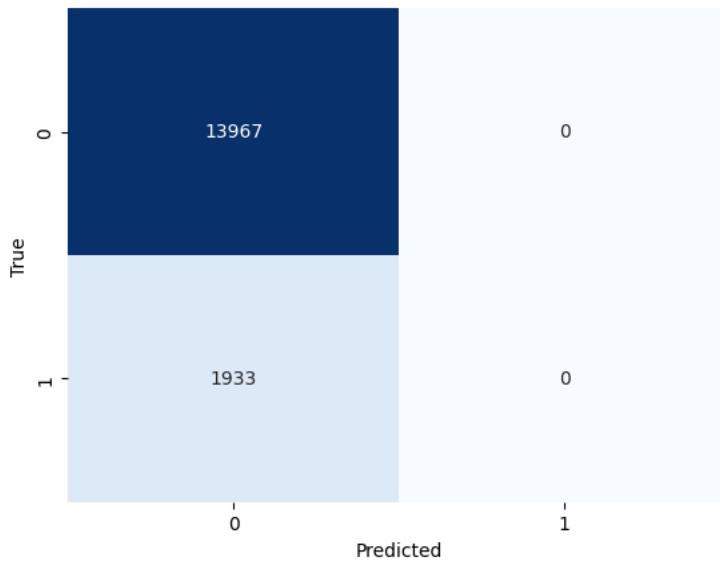
```
4800/4800 [=====] - 14s 3ms/step
```



```
4800/4800 [=====] - 13s 3ms/step
```

	precision	recall	f1-score	support
0	0.875208	1.000000	0.933452	42010.000000
1	0.000000	0.000000	0.000000	5990.000000
accuracy	0.875208	0.875208	0.875208	0.875208
macro avg	0.437604	0.500000	0.466726	48000.000000
weighted avg	0.765990	0.875208	0.816965	48000.000000

```
39/1590 [.....] - ETA: 4s /usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1
 _warn_prf(average, modifier, msg_start, len(result))
1590/1590 [=====] - 4s 3ms/step
```



```
1590/1590 [=====] - 4s 3ms/step
```

	precision	recall	f1-score	support
0	0.878428	1.000000	0.935280	13967.000000
1	0.000000	0.000000	0.000000	1933.000000
accuracy	0.878428	0.878428	0.878428	0.878428
macro avg	0.439214	0.500000	0.467640	15900.000000
weighted avg	0.771635	0.878428	0.821576	15900.000000

```
36/1440 [.....] - ETA: 4s /usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1
 _warn_prf(average, modifier, msg_start, len(result))
1440/1440 [=====] - 4s 3ms/step
```



▼ Model 5

This model, which we will call model 5, is a combination of a Transformer block and an LSTM layer. The Transformer block is designed to learn the dependencies between the time steps of the input sequence and to generate a representation of the input sequence that is more informative and easier to process. The LSTM layer then takes the output of the Transformer block and learns to make predictions based on the temporal dependencies in the input sequence.

The input shape of the model is the same as the previous models, but the key difference is that it utilizes a Transformer block as the initial layer. The Transformer block has shown significant performance improvements over traditional recurrent neural networks like LSTMs, which were used in the previous models.

Evaluation

Overall, Model 5 performs poorly for both classes, as it predicts only the non-stress class (class 0) and ignores the stress class (class 1). It has lower precision, recall, and F1-score for the non-stress class compared to the other models. Model 5 is not able to effectively predict the stress instances.

```

1 # Model 5: Transformer LSTM
2 def build_transform_lstm_1(lstm_units, learning_rate):
3     from tensorflow.keras.layers import Input, Dense, LSTM, Dropout, TransformerBlock
4     from tensorflow.keras.models import Model
5
6     # Define input shape
7     input_shape = train_gen[0][0].shape[2]
8
9     # Define input layer
10    # Define the input shape
11    inputs = Input(seq_length, shape=input_shape)
12
13    # Add transformer block
14    transformer_output = TransformerBlock(64, 2)(inputs)
15
16    # Add LSTM layer
17    plt.xcorr = LSTM(lstm_units)(transformer_output)
18
19    # Add output layer
20    outputs = Dense(1, activation='sigmoid')(x)
21
22    # Define model
23    model = Model(inputs=inputs, outputs=outputs)
24
25    # Compile model
26    model.compile(loss='binary_crossentropy',
27                  optimizer=Adam(learning_rate),
28                  metrics=['binary_accuracy'])
29
30    model.summary()
31
32    return model
33

1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_5_filepath = 'models/model_5_conv_lstm.keras'
4 history_5_filepath = 'models/trainHistoryDict/model_5_history.pkl'
5 #lstm_units_list = [32, 64, 128]
6 #learning_rates_list = [0.01]

1 # Fit the model and tune hyperparameters
2 history_5, model_5, best_lstm_units, best_learning_rate, best_penalty = nn_lstm_hyperparameter_tuning(
3                                         build_lstm_conv_lstm,
4                                         lstm_units_list,
5                                         class_weight_penalty,
6                                         learning_rates_list,
7                                         model_5_filepath)
8 # Pickle the Training Fit History
9 with open(history_5_filepath, 'wb') as file_pi:
10     pickle.dump(history_5, file_pi)

lstm_units: 64, learning_rate: 0.001, weight_penalty: 1
Model: "model_4"

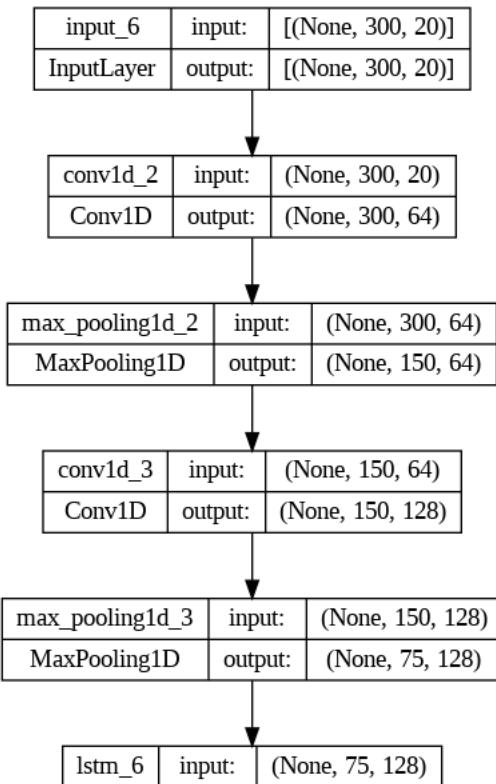
```

Layer (type)	Output Shape	Param #
input_6 (InputLayer)	[(None, 300, 20)]	0
conv1d_2 (Conv1D)	(None, 300, 64)	11584
max_pooling1d_2 (MaxPooling 1D)	(None, 150, 64)	0
conv1d_3 (Conv1D)	(None, 150, 128)	24704
max_pooling1d_3 (MaxPooling 1D)	(None, 75, 128)	0
lstm_6 (LSTM)	(None, 75, 64)	49408
flatten_2 (Flatten)	(None, 4800)	0
dense_7 (Dense)	(None, 32)	153632
dropout_1 (Dropout)	(None, 32)	0
dense_8 (Dense)	(None, 1)	33

Total params: 239,361
Trainable params: 239,361
Non-trainable params: 0

Epoch 1/200
4800/4800 [=====] - 42s 8ms/step - loss: 2.6487e-05 - binary_accuracy: 0.8542 - val_loss: 0.6590 - val_binary_a
Epoch 2/200
4800/4800 [=====] - 38s 8ms/step - loss: 2.8936e-05 - binary_accuracy: 0.6526 - val_loss: 0.6735 - val_binary_a
Epoch 3/200
4800/4800 [=====] - 38s 8ms/step - loss: 2.8917e-05 - binary_accuracy: 0.5652 - val_loss: 0.6767 - val_binary_a
Epoch 4/200
4800/4800 [=====] - 37s 8ms/step - loss: 2.8914e-05 - binary_accuracy: 0.5473 - val_loss: 0.6774 - val_binary_a
Epoch 5/200
4800/4800 [=====] - 37s 8ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5437 - val_loss: 0.6775 - val_binary_a
Epoch 6/200
4800/4800 [=====] - ETA: 0s - loss: 2.8913e-05 - binary_accuracy: 0.5431Restoring model weights from the end of
4800/4800 [=====] - 37s 8ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5431 - val_loss: 0.6776 - val_binary_a
Epoch 6: early stopping
1590/1590 [=====] - 5s 3ms/step
Best LSTM units: 64
Best learning rate: 0.001
Best weight_penalty: 1

```
1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_5, to_file='models/model_5_arch.png', show_shapes=True, show_layer_names=True)
```

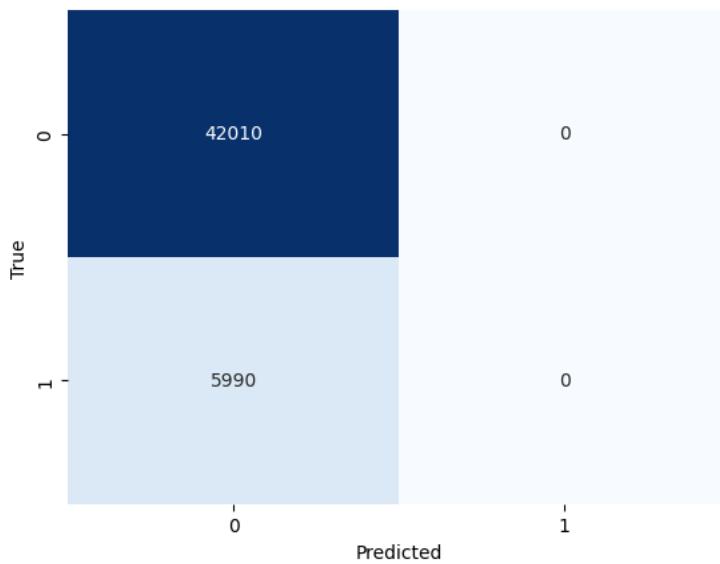


```

1 confusion_matrix_plot(model_5, train_gen, save_file='models/evaluate/model_5_train_confusion_matrix.pdf')
2 classification_report_output(model_5, train_gen, save_file=None)
3 confusion_matrix_plot(model_5, val_gen, save_file='models/evaluate/model_5_val_confusion_matrix.pdf')
4 classification_report_output(model_5, val_gen, save_file=None)
5 confusion_matrix_plot(model_5, test_gen, save_file='models/evaluate/model_5_test_confusion_matrix.pdf')
6 classification_report_output(model_5, test_gen, save_file=None)

```

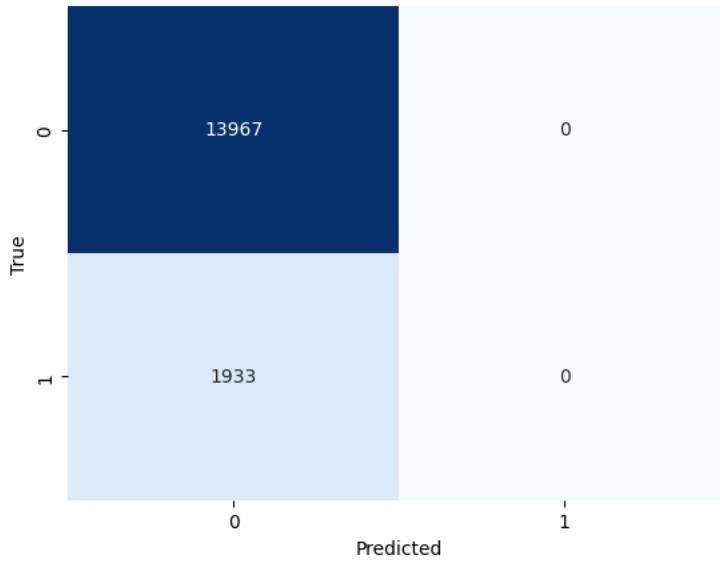
```
4800/4800 [=====] - 13s 3ms/step
```



```
4800/4800 [=====] - 14s 3ms/step
```

	precision	recall	f1-score	support
0	0.875208	1.000000	0.933452	42010.000000
1	0.000000	0.000000	0.000000	5990.000000
accuracy	0.875208	0.875208	0.875208	0.875208
macro avg	0.437604	0.500000	0.466726	48000.000000
weighted avg	0.765990	0.875208	0.816965	48000.000000

```
39/1590 [.....] - ETA: 4s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1, outputted by _warn_prf(average, modifier, msg_start, len(result))  
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 0, outputted by _warn_prf(average, modifier, msg_start, len(result))  
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1, outputted by _warn_prf(average, modifier, msg_start, len(result))  
1590/1590 [=====] - 4s 3ms/step
```



```
1590/1590 [=====] - 5s 3ms/step
```

	precision	recall	f1-score	support
0	0.878428	1.000000	0.935280	13967.000000
1	0.000000	0.000000	0.000000	1933.000000
accuracy	0.878428	0.878428	0.878428	0.878428
macro avg	0.439214	0.500000	0.467640	15900.000000
weighted avg	0.771635	0.878428	0.821576	15900.000000

```
38/1440 [.....] - ETA: 3s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1, outputted by _warn_prf(average, modifier, msg_start, len(result))  
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 0, outputted by _warn_prf(average, modifier, msg_start, len(result))  
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and hence omitted for label 1, outputted by _warn_prf(average, modifier, msg_start, len(result))  
1440/1440 [=====] - 4s 3ms/step
```

▼ Model 6

Model 6 is a bidirectional LSTM model which uses an LSTM layer that processes the input sequence in both directions, forward and backward. The model takes the input shape, and then applies two bidirectional LSTM layers, with 50% dropout rate. The final LSTM layer is then connected to a dense layer with a ReLU activation function. Finally, there is an output layer with a sigmoid activation function.

This model differs from the previous models in that it includes a bidirectional LSTM layer. The bidirectional layer allows the model to capture not only the past but also the future context of the input sequence. This is particularly useful when predicting events where future events may influence the final outcome. However, if the prediction solely depends on the past sequence, a bidirectional LSTM may not be the best model for the data.

Evaluation

The classification report shows that Model 6 performs poorly for both classes. It has low precision, recall, and F1-score for the stress class, indicating that it struggles to correctly predict stress instances. It also has relatively low precision, recall, and F1-score for the non-stress class, indicating that it may have difficulty distinguishing between stress and non-stress instances.

```

1 # Model 6: Bidirectional LSTM
2 def build_bidirection_lstm(lstm_units, learning_rate):
3     import keras
4     from keras.layers import LSTM, Dense, Bidirectional
5     from tensorflow.keras.optimizers import Adam
6
7     # Define the input shape
8     inputs = keras.Input(shape=(seq_length, no_features))
9
10    # Next model add a Dense layer here
11    x = Bidirectional(LSTM(lstm_units,
12                           activation='tanh',
13                           return_sequences=True,
14                           dropout=0.5))(inputs)
15    x = Bidirectional(LSTM(lstm_units,
16                           activation='tanh',
17                           #return_sequences=True,
18                           dropout=0.5))(x)
19    x = Dense(units=batch_size//2,
20              activation='relu')(x)
21
22    outputs = Dense(units=1, activation='sigmoid')(x)
23
24    # Build the model
25    model = keras.Model(inputs=inputs, outputs=outputs)
26
27    # Compile the model
28    model.compile(loss='binary_crossentropy',
29                  optimizer=Adam(learning_rate),
30                  metrics=['binary_accuracy'])
31
32    model.summary()
33
34    return model

1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_6_filepath = 'models/model_6_bidirectional.keras'
4 history_6_filepath = 'models/trainHistoryDict/model_6_history.pkl'
5 #lstm_units_list = [32, 64, 128]
6 #learning_rates_list = [0.0001, 0.001, 0.01, 0.1]
7 class_weight_penalty= [1] #[1, 2, 10, 100, 1000]

1 # Fit the model and tune hyperparameters
2 history_6, model_6, best_lstm_units, best_learning_rate, best_penalty = nn_lstm_hyperparameter_tuning(
3                                         build_bidirection_lstm,
4                                         lstm_units_list,
5                                         class_weight_penalty,
6                                         learning_rates_list,
7                                         model_6_filepath)
8 # Pickle the Training Fit History
9 with open(history_6_filepath, 'wb') as file_pi:
10     pickle.dump(history_6, file_pi)

lstm_units: 64, learning_rate: 0.001, weight_penalty: 1
Model: "model_5"

```

Layer (type)	Output Shape	Param #
--------------	--------------	---------

```
=====
input_7 (InputLayer)      [(None, 300, 20)]      0
bidirectional (Bidirectiona (None, 300, 128)      43520
1)
bidirectional_1 (Bidirectio (None, 128)          98816
nal)
dense_9 (Dense)          (None, 5)                645
dense_10 (Dense)         (None, 1)                6
=====
```

Total params: 142,987
Trainable params: 142,987
Non-trainable params: 0

Epoch 1/200
4800/4800 [=====] - 178s 36ms/step - loss: 1.3697e-05 - binary_accuracy: 0.8737 - val_loss: 0.7350 - val_binary
Epoch 2/200
4800/4800 [=====] - 169s 35ms/step - loss: 1.2941e-05 - binary_accuracy: 0.8864 - val_loss: 0.5033 - val_binary
Epoch 3/200
4800/4800 [=====] - 168s 35ms/step - loss: 1.0869e-05 - binary_accuracy: 0.9360 - val_loss: 1.0188 - val_binary
Epoch 4/200
4800/4800 [=====] - 168s 35ms/step - loss: 9.3878e-06 - binary_accuracy: 0.9002 - val_loss: 1.0766 - val_binary
Epoch 5/200
4800/4800 [=====] - 168s 35ms/step - loss: 7.9902e-06 - binary_accuracy: 0.9410 - val_loss: 1.1299 - val_binary
Epoch 6/200
4800/4800 [=====] - 168s 35ms/step - loss: 6.6757e-06 - binary_accuracy: 0.9376 - val_loss: 1.8557 - val_binary
Epoch 7/200
4799/4800 [=====>.] - ETA: 0s - loss: 6.0842e-06 - binary_accuracy: 0.9441 Restoring model weights from the end of
4800/4800 [=====] - 168s 35ms/step - loss: 6.0830e-06 - binary_accuracy: 0.9441 - val_loss: 1.8370 - val_binary
Epoch 7: early stopping
1590/1590 [=====] - 23s 13ms/step
Best LSTM units: 64
Best learning rate: 0.001
Best weight_penalty: 1

<

>

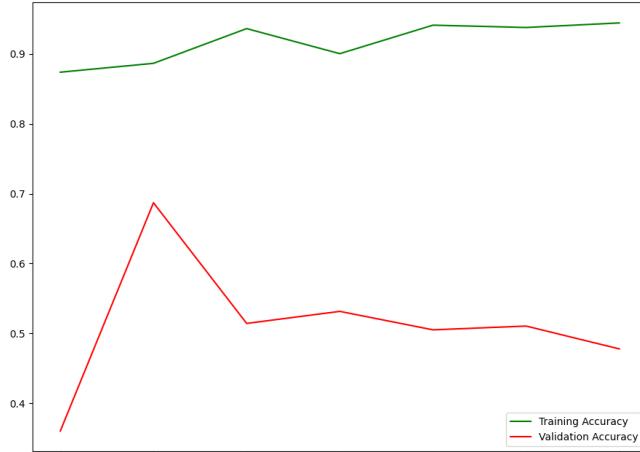
1 # Fit the model to the training data

2 # Load and Evaluate Model

```
3 model_6, history_6 = evaluate_model(model_6_filepath,
4                                     history_6_filepath,
5                                     val_gen,
6                                     test_gen,
7                                     'models/evaluate/model_6_evaluation.pdf')
```

1590/1590 [=====] - 24s 14ms/step - loss: 0.5033 - binary_accuracy: 0.6869

Training and Validation Accuracy



Training and Validation Loss

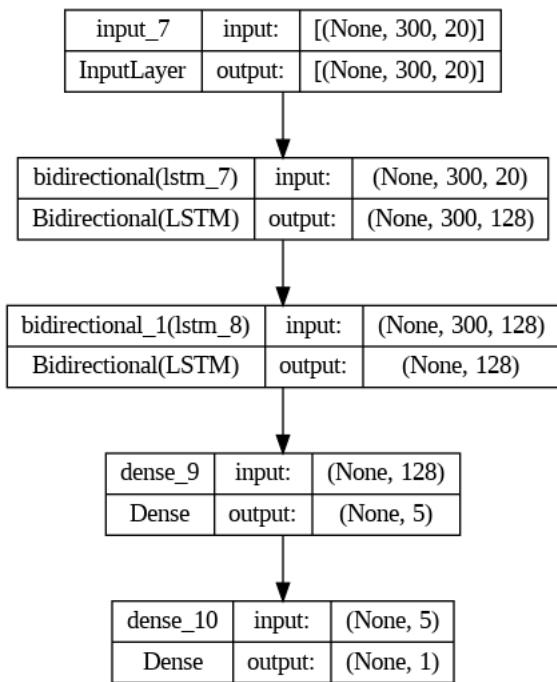


1440/1440 [=====] - 21s 15ms/step - loss: 0.3687 - binary_accuracy: 0.7607

Validation Binary Accuracy is: 0.69

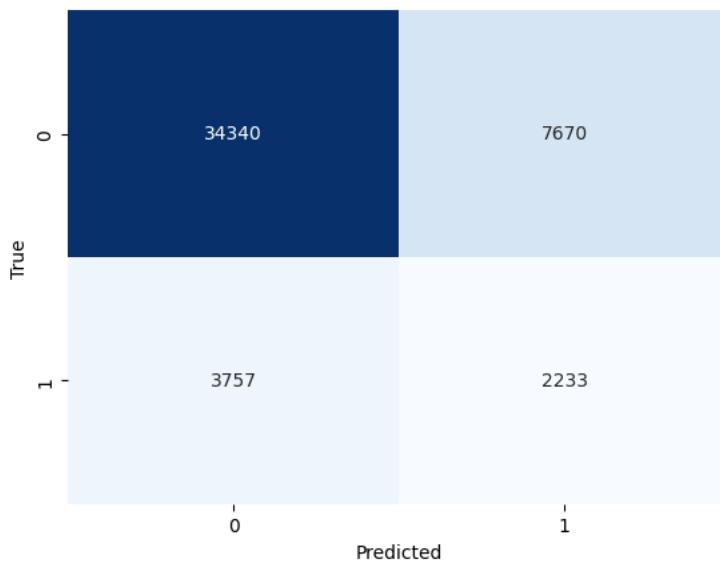
Testing Binary Accuracy is: 0.76

```
1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_6, to_file='models/model_6_arch.png', show_shapes=True, show_layer_names=True)
```



```
1 confusion_matrix_plot(model_6, train_gen, save_file='models/evaluate/model_6_train_confusion_matrix.pdf')
2 classification_report_output(model_6, train_gen, save_file=None)
3 confusion_matrix_plot(model_6, val_gen, save_file='models/evaluate/model_6_val_confusion_matrix.pdf')
4 classification_report_output(model_6, val_gen, save_file=None)
5 confusion_matrix_plot(model_6, test_gen, save_file='models/evaluate/model_6_test_confusion_matrix.pdf')
6 classification_report_output(model_6, test_gen, save_file=None)
```

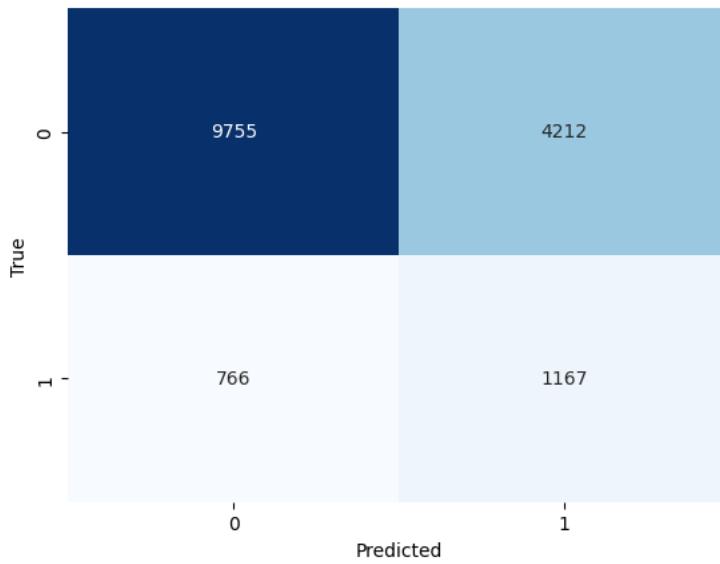
```
4800/4800 [=====] - 66s 13ms/step
```



```
4800/4800 [=====] - 66s 14ms/step
```

	precision	recall	f1-score	support
0	0.901383	0.817424	0.857353	42010.000000
1	0.225487	0.372788	0.281004	5990.000000
accuracy	0.761938	0.761938	0.761938	0.761938
macro avg	0.563435	0.595106	0.569179	48000.000000
weighted avg	0.817037	0.761938	0.785430	48000.000000

```
1590/1590 [=====] - 22s 14ms/step
```



```
1590/1590 [=====] - 22s 14ms/step
```

precision	recall	f1-score	support
-----------	--------	----------	---------

▼ Model 7

Model 7 is a timeseries classification model that uses a transformer architecture. The transformer encoder is composed of a normalization and attention layer, followed by a feedforward neural network. The attention layer is applied to the inputs twice, and a residual connection is added to the output. The feedforward part consists of a convolutional layer and another residual connection. The global average pooling layer is added after the transformer blocks, followed by a fully connected network with a single output.

Compared to the previous models, model 7 uses a transformer architecture instead of LSTM layers, which may be better suited for timeseries classification tasks. Additionally, it uses a multi-head attention mechanism, which allows the model to focus on different parts of the input sequence simultaneously. Finally, it uses global average pooling instead of LSTM or bidirectional LSTM layers, which reduces the number of parameters in the model and can help prevent overfitting.

Evaluation

Model 7 performs poorly for both classes. It has low precision, recall, and F1-score for the non-stress class, indicating that it fails to predict non-stress instances. It has high recall for the stress class, but very low precision and F1-score, indicating that it incorrectly predicts many instances as stress. This is because it is only predicting the stress class (class 1).

```

1 ''
2 Model_7: Timeseries classification with a transformer model
3 ''
4 from tensorflow import keras
5 from tensorflow.keras import layers
6 from keras.models import Sequential
7 from keras.layers import LSTM, Dense, Dropout
8 from tensorflow.keras.optimizers import Adam
9 from keras.callbacks import EarlyStopping, ModelCheckpoint
10
11 # Transformer
12 def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout=0):
13     # Normalization and Attention
14     x = layers.LayerNormalization(epsilon=1e-6)(inputs)
15     x = layers.MultiHeadAttention(key_dim=head_size,
16                                    num_heads=num_heads,
17                                    dropout=dropout)(x, x)
18     x = layers.Dropout(dropout)(x)
19     res = x + inputs
20
21     # Feed Forward Part
22     x = layers.LayerNormalization(epsilon=1e-6)(res)
23     x = layers.Conv1D(filters=ff_dim, kernel_size=1, activation="relu")(x)
24     x = layers.Dropout(dropout)(x)
25     x = layers.Conv1D(filters=inputs.shape[-1], kernel_size=1)(x)
26     return x + res
27
28 def build_transformer_model(
29     input_shape,
30     n_classes,
31     head_size,
32     num_heads,
33     ff_dim,
34     num_transformer_blocks,
35     mlp_units,
36     dropout=0,
37     mlp_dropout=0,
38 ):
39     inputs = keras.Input(shape=(seq_length, no_features))
40     x = inputs
41     for _ in range(num_transformer_blocks):
42         x = transformer_encoder(x, head_size, num_heads, ff_dim, dropout)
43
44     x = layers.GlobalAveragePooling1D(data_format="channels_last")(x)
45     for dim in mlp_units:
46         x = layers.Dense(dim, activation="relu")(x)
47         x = layers.Dropout(mlp_dropout)(x)
48     outputs = layers.Dense(n_classes, activation="softmax")(x)
49     return keras.Model(inputs, outputs)
50
51 # Build model
52 model_7 = build_transformer_model(
53     seq_length,
54     n_classes=1,
55     head_size=256,
56     num_heads=4, # Number of Transformer repeats
57     ff_dim=4,
58     num_transformer_blocks=4,
59     mlp_units=[128],
60     mlp_dropout=0.4,
61     dropout=0.5,
62 )
63 best_learning_rate = 0.001
64
65 # Compile Model
66 model_7.compile(
67     loss="binary_crossentropy",
68     optimizer=Adam(learning_rate=best_learning_rate),
69     metrics=["binary_accuracy"],
70 )
71 model_7.summary()

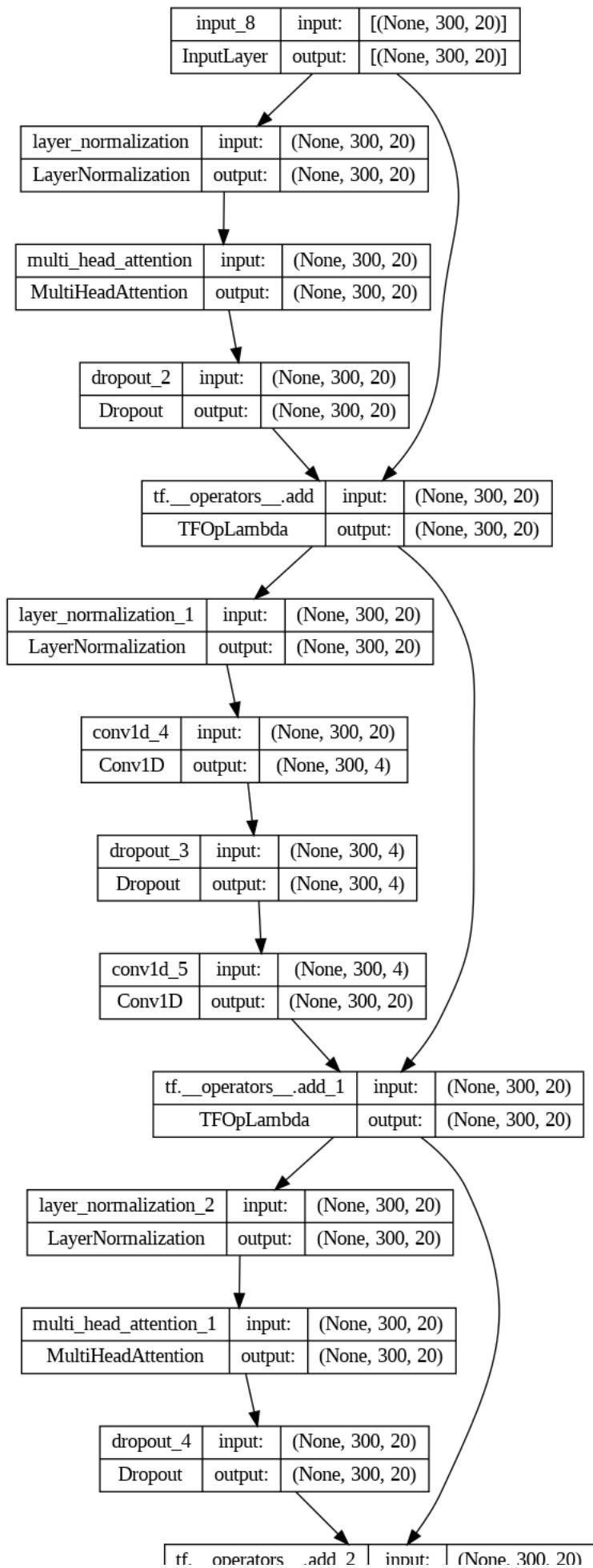
```

tf.__operators__.add_4 (TFOpLa (None, 300, 20) mbda)	0	['dropout_6[0][0]', 'tf.__operators__.add_3[0][0]']	
layer_normalization_5 (LayerNo rmalization)	40	['tf.__operators__.add_4[0][0]']	
conv1d_8 (Conv1D)	(None, 300, 4)	84	['layer_normalization_5[0][0]']
dropout_7 (Dropout)	(None, 300, 4)	0	['conv1d_8[0][0]']
conv1d_9 (Conv1D)	(None, 300, 20)	100	['dropout_7[0][0]']
tf.__operators__.add_5 (TFOpLa (None, 300, 20) mbda)	0	['conv1d_9[0][0]', 'tf.__operators__.add_4[0][0]']	
layer_normalization_6 (LayerNo rmalization)	40	['tf.__operators__.add_5[0][0]']	
multi_head_attention_3 (MultiH eadAttention)	(None, 300, 20)	85012	['layer_normalization_6[0][0]', 'layer_normalization_6[0][0]']
dropout_8 (Dropout)	(None, 300, 20)	0	['multi_head_attention_3[0][0]']
tf.__operators__.add_6 (TFOpLa (None, 300, 20) mbda)	0	['dropout_8[0][0]', 'tf.__operators__.add_5[0][0]']	
layer_normalization_7 (LayerNo rmalization)	40	['tf.__operators__.add_6[0][0]']	
conv1d_10 (Conv1D)	(None, 300, 4)	84	['layer_normalization_7[0][0]']
dropout_9 (Dropout)	(None, 300, 4)	0	['conv1d_10[0][0]']
conv1d_11 (Conv1D)	(None, 300, 20)	100	['dropout_9[0][0]']
tf.__operators__.add_7 (TFOpLa (None, 300, 20) mbda)	0	['conv1d_11[0][0]', 'tf.__operators__.add_6[0][0]']	
global_average_pooling1d (Glob alAveragePooling1D)	(None, 20)	0	['tf.__operators__.add_7[0][0]']
dense_11 (Dense)	(None, 128)	2688	['global_average_pooling1d[0][0]']
dropout_10 (Dropout)	(None, 128)	0	['dense_11[0][0]']
dense_12 (Dense)	(None, 1)	129	['dropout_10[0][0]']

```
Total params: 343,921
Trainable params: 343,921
Non-trainable params: 0
```

```
1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_7_filepath = 'models/model_7_transformer.keras'
4 history_7_filepath = 'models/trainHistoryDict/model_7_history.pkl'

1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_7, to_file='models/model_7_arch.png', show_shapes=True, show_layer_names=True)
```



```

1 # Fit the model to the training data
2 history_7 = model_7.fit(train_gen,
3                         validation_data=val_gen,
4                         shuffle=False,
5                         epochs=epochs,
6                         class_weight=class_weights_dict,
7                         callbacks=callbacks(model_7_filepath,
8                                              patience=5))
9
10 # Pickle the Traning Fit History
11 with open(history_7_filepath, 'wb') as file_pi:
12     pickle.dump(history_7, file_pi)

Epoch 1/200
4800/4800 [=====] - 231s 45ms/step - loss: 1.2382e-05 - binary_accuracy: 0.1248 - val_loss: 17.8575 - val_binary_accuracy: 0.1248
Epoch 2/200
4800/4800 [=====] - 217s 45ms/step - loss: 1.7404e-05 - binary_accuracy: 0.1248 - val_loss: 1.6641 - val_binary_accuracy: 0.1248
Epoch 3/200
4800/4800 [=====] - 216s 45ms/step - loss: 1.7328e-05 - binary_accuracy: 0.1248 - val_loss: 13.0358 - val_binary_accuracy: 0.1248
Epoch 4/200
4800/4800 [=====] - 216s 45ms/step - loss: 2.7016e-05 - binary_accuracy: 0.1248 - val_loss: 0.9621 - val_binary_accuracy: 0.1248
Epoch 5/200
4800/4800 [=====] - 215s 45ms/step - loss: 1.6603e-05 - binary_accuracy: 0.1248 - val_loss: 2.2355 - val_binary_accuracy: 0.1248
Epoch 6/200
4800/4800 [=====] - 216s 45ms/step - loss: 1.5391e-05 - binary_accuracy: 0.1248 - val_loss: 2.6232 - val_binary_accuracy: 0.1248
Epoch 7/200
4800/4800 [=====] - 215s 45ms/step - loss: 1.7029e-05 - binary_accuracy: 0.1248 - val_loss: 16.6839 - val_binary_accuracy: 0.1248
Epoch 8/200
4800/4800 [=====] - 214s 45ms/step - loss: 1.3612e-05 - binary_accuracy: 0.1248 - val_loss: 3.5823 - val_binary_accuracy: 0.1248
Epoch 9/200
4799/4800 [=====>..] - ETA: 0s - loss: 2.3809e-05 - binary_accuracy: 0.1248Restoring model weights from the end of epoch 9
4800/4800 [=====] - 214s 45ms/step - loss: 2.3805e-05 - binary_accuracy: 0.1248 - val_loss: 3.0948 - val_binary_accuracy: 0.1248
Epoch 9: early stopping

```

1 # Load and Evaluate Model

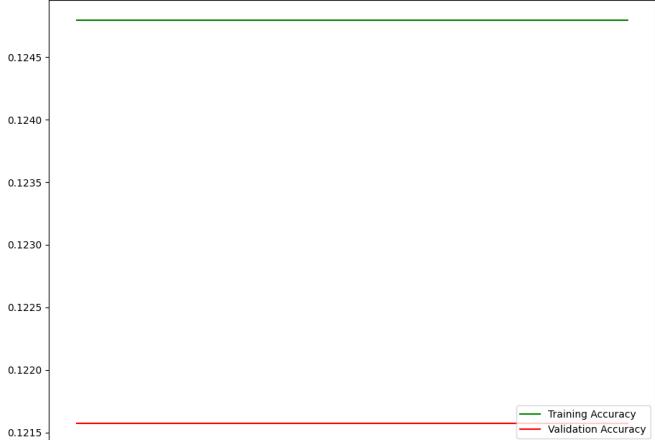
```

2 model_7, history_7 = evaluate_model(model_7_filepath,
3                                     history_7_filepath,
4                                     val_gen,
5                                     test_gen,
6                                     'models/evaluate/model_7_evaluation.pdf')

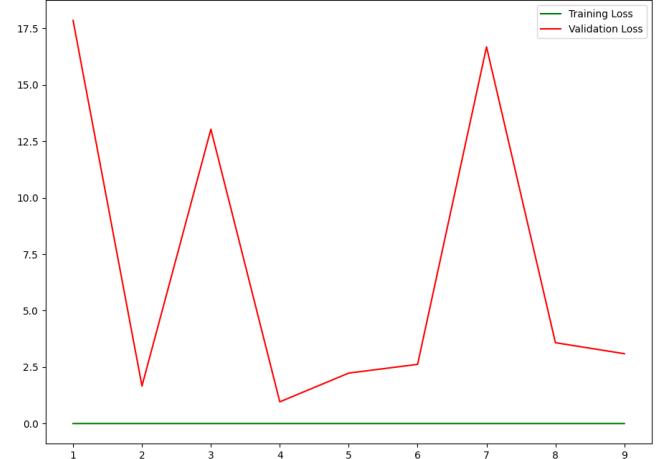
```

1590/1590 [=====] - 24s 15ms/step - loss: 0.9621 - binary_accuracy: 0.1216

Training and Validation Accuracy



Training and Validation Loss



1440/1440 [=====] - 21s 15ms/step - loss: 0.6001 - binary_accuracy: 0.1419

Validation Binary Accuracy is: 0.12

Testing Binary Accuracy is: 0.14

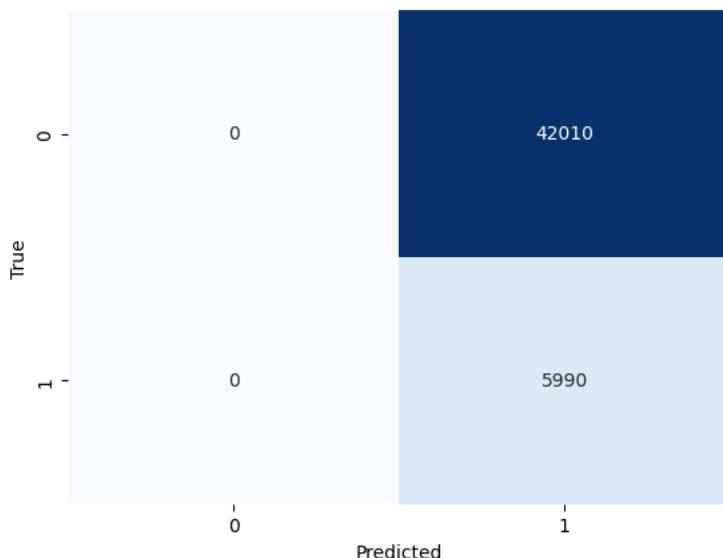
```

1 confusion_matrix_plot(model_7, train_gen, save_file='models/evaluate/model_7_train_confusion_matrix.pdf')
2 classification_report_output(model_7, train_gen, save_file=None)
3 confusion_matrix_plot(model_7, val_gen, save_file='models/evaluate/model_7_val_confusion_matrix.pdf')

```

```
4 classification_report_output(model_7, val_gen, save_file=None)
5 confusion_matrix_plot(model_7, test_gen, save_file='models/evaluate/model_7_test_confusion_matrix.pdf')
6 classification_report_output(model_7, test_gen, save_file=None)
```

```
4800/4800 [=====] - 66s 14ms/step
```



```
4800/4800 [=====] - 65s 13ms/step
```

	precision	recall	f1-score	support
0	0.00000	0.00000	0.00000	42010.00000
1	0.124792	1.00000	0.221893	5990.00000
accuracy	0.124792	0.124792	0.124792	0.124792
macro avg	0.062396	0.500000	0.110946	48000.00000
weighted avg	0.015573	0.124792	0.027690	48000.00000

4/1590 [.....] - ETA: 29s /usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: l
_warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_warn_prf(average, modifier, msg_start, len(result))
1590/1590 [=====] - 21s 13ms/step

▼ Model 8

This architecture combines the strengths of LSTM and Transformer layers. The LSTM layer captures sequential patterns, while the Transformer blocks enable the model to learn both local and global dependencies in the timeseries data. This combination can potentially improve the model's ability to detect stress patterns in the input timeseries.

Evaluation

Model 8 performs very poorly for the stress classe. It fails to predict any instances of stress correctly, as it only predicts the normal state (class 0).

```
1 # Model 8
2 def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout=0):
3     # Normalization and Attention
4     x = layers.LayerNormalization(epsilon=1e-6)(inputs)
5     x = layers.MultiHeadAttention(key_dim=head_size,
6                                     num_heads=num_heads,
7                                     dropout=dropout)(x, x)
8     x = layers.Dropout(dropout)(x)
9     res = x + inputs
10
11    # Feed Forward Part
12    x = layers.LayerNormalization(epsilon=1e-6)(res)
13    x = layers.Conv1D(filters=ff_dim, kernel_size=1, activation="relu")(x)
14    x = layers.Dropout(dropout)(x)
15    x = layers.Conv1D(filters=inputs.shape[-1], kernel_size=1)(x)
16    return x + res
17
18 def build_transformer_lstm_model(input_shape, n_classes, head_size, num_heads, ff_dim, num_transformer_blocks, lstm_units, mlp_units, drop
19     inputs = keras.Input(shape=input_shape)
20     x = inputs
21
22     # LSTM layer
23     lstm_out = layers.LSTM(lstm_units, return_sequences=True)(x)
24
25     # Transformer blocks
26     for _ in range(num_transformer_blocks):
```

```

27     x = transformer_encoder(lstm_out, head_size, num_heads, ff_dim, dropout)
28
29 # Flatten and MLP layers
30 x = layers.Flatten()(x)
31 for dim in mlp_units:
32     x = layers.Dense(dim, activation="relu")(x)
33     x = layers.Dropout(mlp_dropout)(x)
34 outputs = layers.Dense(n_classes, activation="sigmoid")(x)
35 return keras.Model(inputs, outputs)
36
37 # Define model hyperparameters
38 seq_length = 300
39 n_classes = 1
40 head_size = 256
41 num_heads = 4
42 ff_dim = 4
43 num_transformer_blocks = 4
44 lstm_units = 64
45 mlp_units = [128]
46 mlp_dropout = 0.4
47 dropout = 0.5
48
49 # Build the model
50 model_8 = build_transformer_lstm_model(
51     input_shape=(seq_length, train_gen[0][0].shape[2]),
52     n_classes=n_classes,
53     head_size=head_size,
54     num_heads=num_heads,
55     ff_dim=ff_dim,
56     num_transformer_blocks=num_transformer_blocks,
57     lstm_units=lstm_units,
58     mlp_units=mlp_units,
59     dropout=dropout,
60     mlp_dropout=mlp_dropout
61 )
62
63 # Compile the model
64 best_learning_rate = 0.001
65 model_8.compile(
66     loss="binary_crossentropy",
67     optimizer=Adam(learning_rate=best_learning_rate),
68     metrics=["binary_accuracy"]
69 )
70
71 # Print the model summary
72 model_8.summary()
73

```

Model: "model_7"

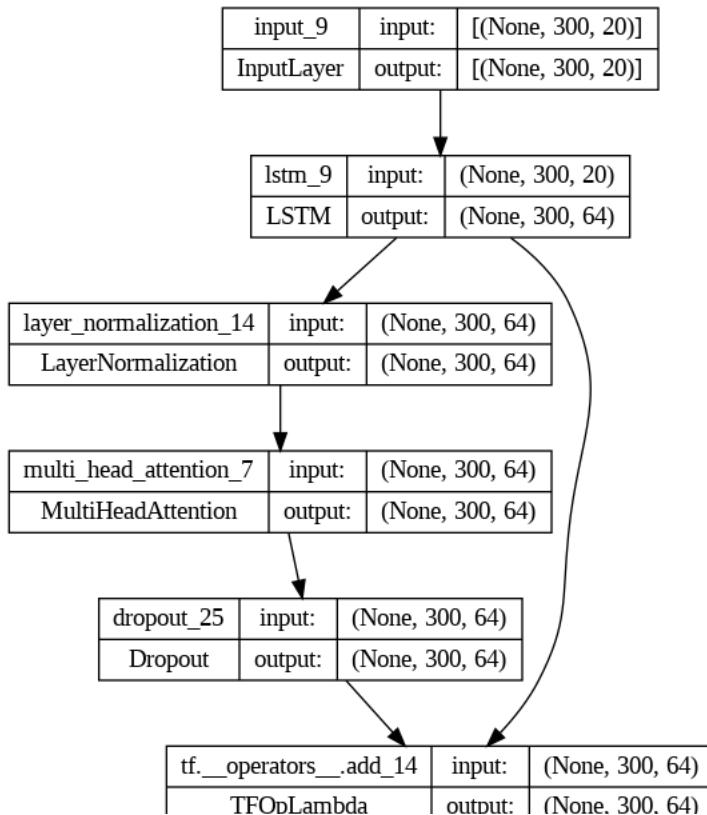
Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_9 (InputLayer)	[(None, 300, 20)]	0	[]
lstm_9 (LSTM)	(None, 300, 64)	21760	['input_9[0][0]']
layer_normalization_14 (LayerNormalization)	(None, 300, 64)	128	['lstm_9[0][0]']
multi_head_attention_7 (MultiHeadAttention)	(None, 300, 64)	265280	['layer_normalization_14[0][0]', 'layer_normalization_14[0][0]']
dropout_25 (Dropout)	(None, 300, 64)	0	['multi_head_attention_7[0][0]']
tf.__operators__.add_14 (TFOpLambda)	(None, 300, 64)	0	['dropout_25[0][0]', 'lstm_9[0][0]']
layer_normalization_15 (LayerNormalization)	(None, 300, 64)	128	['tf.__operators__.add_14[0][0]']
conv1d_18 (Conv1D)	(None, 300, 4)	260	['layer_normalization_15[0][0]']
dropout_26 (Dropout)	(None, 300, 4)	0	['conv1d_18[0][0]']
conv1d_19 (Conv1D)	(None, 300, 64)	320	['dropout_26[0][0]']
tf.__operators__.add_15 (TFOpLambda)	(None, 300, 64)	0	['conv1d_19[0][0]', 'tf.__operators__.add_14[0][0]']
flatten_3 (Flatten)	(None, 19200)	0	['tf.__operators__.add_15[0][0]']

```
dense_13 (Dense)           (None, 128)      2457728      ['flatten_3[0][0]']
dropout_27 (Dropout)        (None, 128)      0           ['dense_13[0][0]']
dense_14 (Dense)           (None, 1)        129         ['dropout_27[0][0]']

=====
Total params: 2,745,733
Trainable params: 2,745,733
Non-trainable params: 0
```

```
1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_8_filepath = 'models/model_8_transformer.keras'
4 history_8_filepath = 'models/trainHistoryDict/model_8_history.pkl'

1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_8, to_file='models/model_8_arch.png', show_shapes=True, show_layer_names=True)
```



```

1 # Fit the model to the training data
2 history_8 = model_8.fit(train_gen,
3                         validation_data=val_gen,
4                         shuffle=False,
5                         epochs=epochs,
6                         class_weight=class_weights_dict,
7                         callbacks=callbacks(model_8_filepath,
8                                              patience=5))
9
10 # Pickle the Traning Fit History
11 with open(history_8_filepath, 'wb') as file_pi:
12     pickle.dump(history_8, file_pi)

Epoch 1/200
4800/4800 [=====] - 110s 22ms/step - loss: 8.3306e-04 - binary_accuracy: 0.8017 - val_loss: 3597.5811 - val_binary_accuracy: 0.5427
Epoch 2/200
4800/4800 [=====] - 103s 22ms/step - loss: 4.2024e-05 - binary_accuracy: 0.6275 - val_loss: 1.4618 - val_binary_accuracy: 0.5429
Epoch 3/200
4800/4800 [=====] - 102s 21ms/step - loss: 2.8915e-05 - binary_accuracy: 0.5575 - val_loss: 2.3882 - val_binary_accuracy: 0.5431
Epoch 4/200
4800/4800 [=====] - 103s 21ms/step - loss: 2.8911e-05 - binary_accuracy: 0.5456 - val_loss: 2.3887 - val_binary_accuracy: 0.5429
Epoch 5/200
4800/4800 [=====] - 103s 21ms/step - loss: 2.8911e-05 - binary_accuracy: 0.5431 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 6/200
4800/4800 [=====] - 105s 22ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5425 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 7/200
4800/4800 [=====] - 105s 22ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5427 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 8/200
4800/4800 [=====] - 104s 22ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 9/200
4800/4800 [=====] - 103s 21ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 10/200
4800/4800 [=====] - 103s 21ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 11/200
4800/4800 [=====] - 106s 22ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 12/200
4800/4800 [=====] - 104s 22ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 13/200
4798/4800 [=====] - ETA: 0s - loss: 2.8918e-05 - binary_accuracy: 0.5427Restoring model weights from the end of epoch 13
4800/4800 [=====] - 104s 22ms/step - loss: 2.8913e-05 - binary_accuracy: 0.5429 - val_loss: 0.6776 - val_binary_accuracy: 0.5429
Epoch 13: early stopping

```

| Dense | Output. | (None, 120) |

```

1 # Load and Evaluate Model
2 model_8, history_8 = evaluate_model(model_8_filepath,
3                                     history_8_filepath,
4                                     ....)

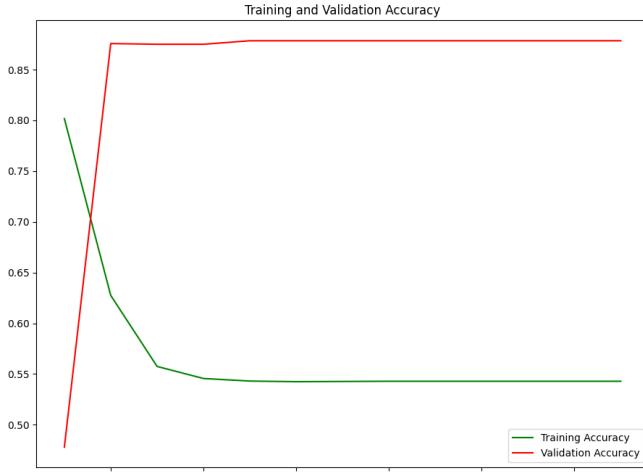
```

```

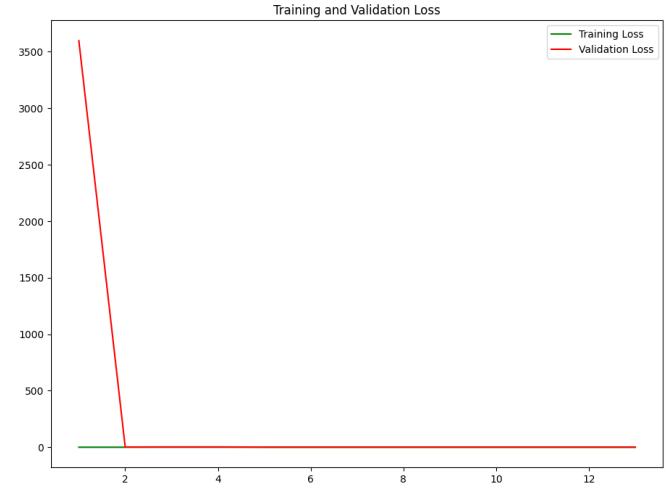
4         val_gen,
5         test_gen,
6         'models/evaluate/model_8_evaluation.pdf')

```

1590/1590 [=====] - 14s 8ms/step - loss: 0.6776 - binary_accuracy: 0.8784



Training and Validation Loss



1440/1440 [=====] - 12s 8ms/step - loss: 0.6784 - binary_accuracy: 0.8581

Validation Binary Accuracy is: 0.88

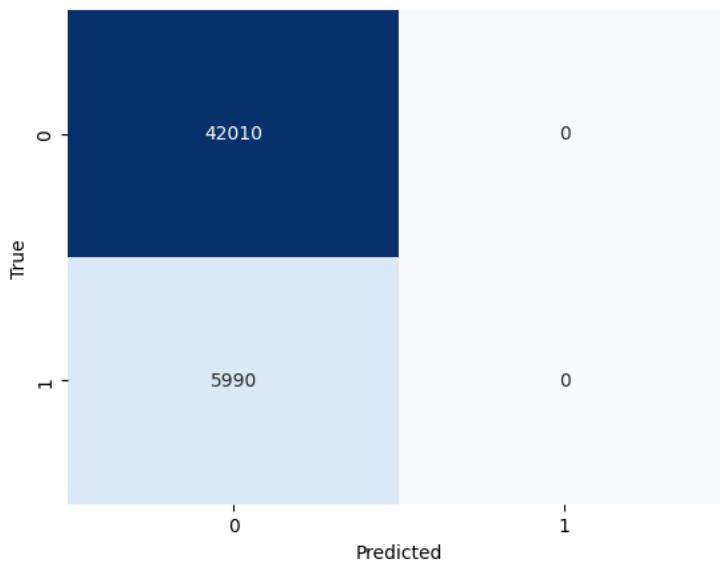
Testing Binary Accuracy is: 0.86

```

1 confusion_matrix_plot(model_8, train_gen, save_file='models/evaluate/model_8_train_confusion_matrix.pdf')
2 classification_report_output(model_8, train_gen, save_file=None)
3 confusion_matrix_plot(model_8, val_gen, save_file='models/evaluate/model_8_val_confusion_matrix.pdf')
4 classification_report_output(model_8, val_gen, save_file=None)
5 confusion_matrix_plot(model_8, test_gen, save_file='models/evaluate/model_8_test_confusion_matrix.pdf')
6 classification_report_output(model_8, test_gen, save_file=None)

```

```
4800/4800 [=====] - 32s 7ms/step
```



```
4800/4800 [=====] - 32s 7ms/step
```

	precision	recall	f1-score	support
0	0.875208	1.000000	0.933452	42010.000000
1	0.000000	0.000000	0.000000	5990.000000

```
accuracy 0.875208 0.875208 0.875208 0.875208
```

```
macro avg 0.437604 0.500000 0.466726 48000.000000
```

```
weighted avg 0.765990 0.875208 0.816965 48000.000000
```

```
6/1590 [.....] - ETA: 16s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

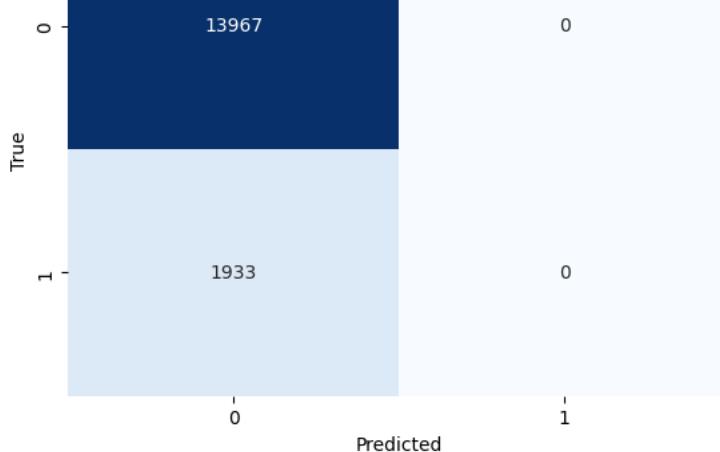
```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
_warn_prf(average, modifier, msg_start, len(result))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
1590/1590 [=====] - 11s 7ms/step
```



```
1590/1590 [=====] - 11s 7ms/step
```

	precision	recall	f1-score	support
0	0.878428	1.000000	0.935280	13967.000000
1	0.000000	0.000000	0.000000	1933.000000

```
accuracy 0.878428 0.878428 0.878428 0.878428
```

```
macro avg 0.439214 0.500000 0.467640 15900.000000
```

```
weighted avg 0.771635 0.878428 0.821576 15900.000000
```

```
15/1440 [.....] - ETA: 10s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0. This will change in version 0.23 if you rely on this metric. Fix your pipeline.
```

```
1440/1440 [=====] - 10s 7ms/step
```



▼ Model 9

This is a valid implementation of the [TS-Transformer](#) model using the Keras API in TensorFlow. It consists of a combination of an LSTM layer and several Transformer blocks followed by a fully connected layer to perform binary classification.

The model is defined using the `build_ts_transformer()` function, which takes in the input shape, number of classes, hyperparameters such as head size, number of heads, feedforward dimension, number of transformer blocks, LSTM units, and MLP units.

Evaluation

Model 9 improves upon model 8 but not by much. Model 9 predicted only a few instances of class 1 correctly. Overall, the model's accuracy is high for classifying non-stress instances (class 0), but it fails to detect instances of stress (class 1).

```
!           !
1 from tensorflow import keras
2 from tensorflow.keras import layers
3 from tensorflow.keras.optimizers import Adam
4 from keras.callbacks import EarlyStopping, ModelCheckpoint
5
6 # Define function to build the TS-Transformer model
7 def build_ts_transformer(input_shape, n_classes, head_size, num_heads, ff_dim, num_transformer_blocks, lstm_units, mlp_units, dropout=0, mlp_dropout=0.4):
8     # LSTM layer
9     inputs = keras.Input(shape=input_shape)
10    lstm_out = layers.LSTM(lstm_units, return_sequences=True)(inputs)
11
12    # Transformer blocks
13    x = lstm_out
14    for _ in range(num_transformer_blocks):
15        x = transformer_encoder(x, head_size, num_heads, ff_dim, dropout)
16
17    # Flatten and MLP layers
18    x = layers.Flatten()(x)
19    for dim in mlp_units:
20        x = layers.Dense(dim, activation="relu")(x)
21        x = layers.Dropout(mlp_dropout)(x)
22
23    # Output layer
24    outputs = layers.Dense(n_classes, activation="sigmoid")(x)
25
26    # Create the model
27    model = keras.Model(inputs=inputs, outputs=outputs)
28
29    return model
30
31 # Define the hyperparameters
32 seq_length = 300
33 n_classes = 1
34 head_size = 256
35 num_heads = 4
36 ff_dim = 4
37 num_transformer_blocks = 4
38 lstm_units = 64
39 mlp_units = [128]
40 mlp_dropout = 0.4
41 dropout = 0.5
42 epochs = 20
43 batch_size = 32
44 learning_rate = 0.001
45
46 # Build the model
47 model_9 = build_ts_transformer(
48     input_shape=(seq_length, train_gen[0][0].shape[2]),
49     n_classes=n_classes,
50     head_size=head_size,
51     num_heads=num_heads,
52     ff_dim=ff_dim,
53     num_transformer_blocks=num_transformer_blocks,
54     lstm_units=lstm_units,
55     mlp_units=mlp_units,
56     dropout=dropout,
57     mlp_dropout=mlp_dropout
58 )
59
60 # Compile the model
61 best_learning_rate = 0.001
```

```

62 model_9.compile(
63     loss="binary_crossentropy",
64     optimizer=Adam(learning_rate=best_learning_rate),
65     metrics=["binary_accuracy"]
66 )
67
68 # Print the model summary
69 model_9.summary()
70

Model: "model_8"
-----
```

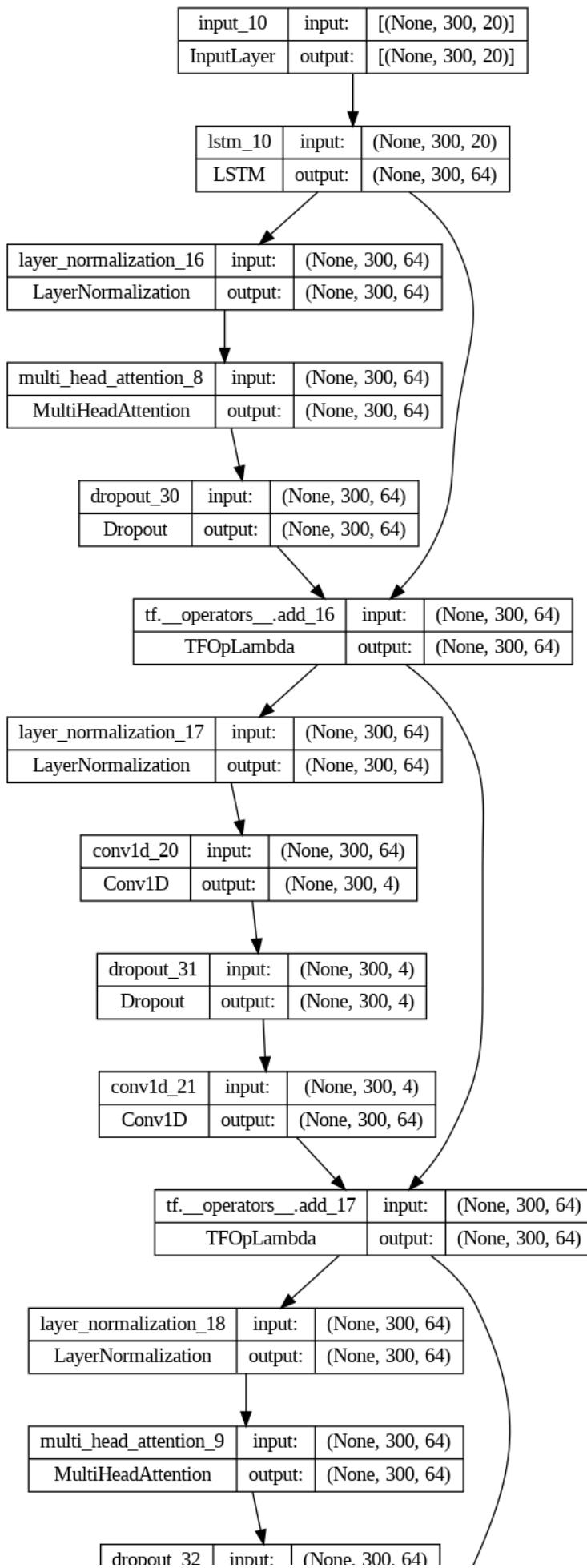
Layer (type)	Output Shape	Param #	Connected to
input_10 (InputLayer)	[None, 300, 20]	0	[]
lstm_10 (LSTM)	(None, 300, 64)	21760	['input_10[0][0]']
layer_normalization_16 (LayerN ormalization)	(None, 300, 64)	128	['lstm_10[0][0]']
multi_head_attention_8 (MultiH eadAttention)	(None, 300, 64)	265280	['layer_normalization_16[0][0]', 'layer_normalization_16[0][0]']
dropout_30 (Dropout)	(None, 300, 64)	0	['multi_head_attention_8[0][0]']
tf.__operators__.add_16 (TFOpL ambda)	(None, 300, 64)	0	['dropout_30[0][0]', 'lstm_10[0][0]']
layer_normalization_17 (LayerN ormalization)	(None, 300, 64)	128	['tf.__operators__.add_16[0][0]']
conv1d_20 (Conv1D)	(None, 300, 4)	260	['layer_normalization_17[0][0]']
dropout_31 (Dropout)	(None, 300, 4)	0	['conv1d_20[0][0]']
conv1d_21 (Conv1D)	(None, 300, 64)	320	['dropout_31[0][0]']
tf.__operators__.add_17 (TFOpL ambda)	(None, 300, 64)	0	['conv1d_21[0][0]', 'tf.__operators__.add_16[0][0]']
layer_normalization_18 (LayerN ormalization)	(None, 300, 64)	128	['tf.__operators__.add_17[0][0]']
multi_head_attention_9 (MultiH eadAttention)	(None, 300, 64)	265280	['layer_normalization_18[0][0]', 'layer_normalization_18[0][0]']
dropout_32 (Dropout)	(None, 300, 64)	0	['multi_head_attention_9[0][0]']
tf.__operators__.add_18 (TFOpL ambda)	(None, 300, 64)	0	['dropout_32[0][0]', 'tf.__operators__.add_17[0][0]']
layer_normalization_19 (LayerN ormalization)	(None, 300, 64)	128	['tf.__operators__.add_18[0][0]']
conv1d_22 (Conv1D)	(None, 300, 4)	260	['layer_normalization_19[0][0]']
dropout_33 (Dropout)	(None, 300, 4)	0	['conv1d_22[0][0]']
conv1d_23 (Conv1D)	(None, 300, 64)	320	['dropout_33[0][0]']
tf.__operators__.add_19 (TFOpL ambda)	(None, 300, 64)	0	['conv1d_23[0][0]', 'tf.__operators__.add_18[0][0]']
layer_normalization_20 (LayerN ormalization)	(None, 300, 64)	128	['tf.__operators__.add_19[0][0]']
multi_head_attention_10 (Multi HeadAttention)	(None, 300, 64)	265280	['layer_normalization_20[0][0]', 'layer_normalization_20[0][0]']

```

1 # Parameters for fitting and saving model training data
2 epochs = 200
3 model_9_filepath = 'models/model_9_transformer.keras'
4 history_9_filepath = 'models/trainHistoryDict/model_9_history.pkl'

1 from keras.utils import plot_model
2
3 # assuming you have defined a model called "model"
4 plot_model(model_9, to_file='models/model_8_arch.png', show_shapes=True, show_layer_names=True)

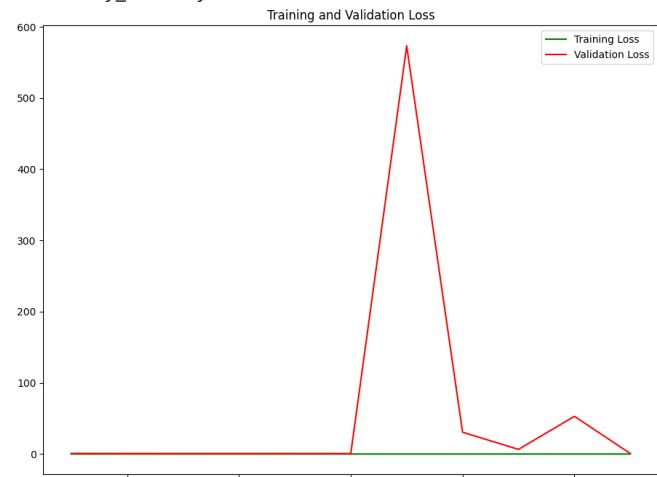
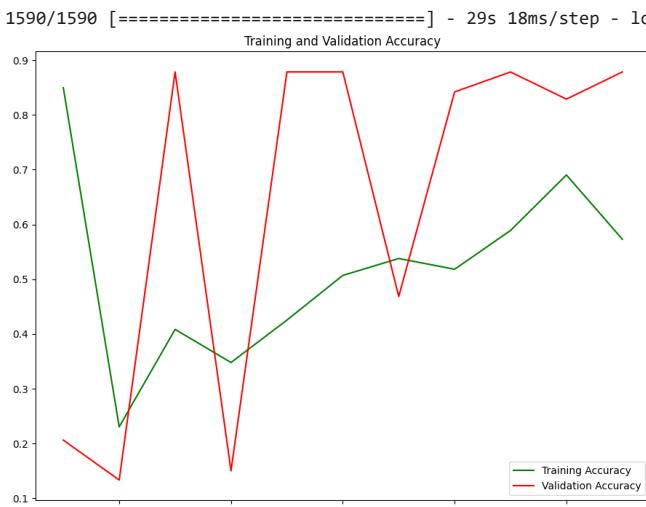
```



```
1 # Fit the model to the training data
2 history_9 = model_9.fit(train_gen,
3                         validation_data=val_gen,
4                         shuffle=False,
5                         epochs=epochs,
6                         class_weight=class_weights_dict,
7                         callbacks=callbacks(model_9_filepath,
8                                              patience=5))
9
10 # Pickle the Traning Fit History
11 with open(history_9_filepath, 'wb') as file_pi:
12     pickle.dump(history_9, file_pi)

Epoch 1/200
4800/4800 [=====] - 270s 53ms/step - loss: 0.0057 - binary_accuracy: 0.8497 - val_loss: 0.8269 - val_binary_accuracy: 0.2303
Epoch 2/200
4800/4800 [=====] - 253s 53ms/step - loss: 3.0539e-05 - binary_accuracy: 0.2303 - val_loss: 0.6956 - val_binary_accuracy: 0.4083
Epoch 3/200
4800/4800 [=====] - 253s 53ms/step - loss: 2.8953e-05 - binary_accuracy: 0.4083 - val_loss: 0.6726 - val_binary_accuracy: 0.3478
Epoch 4/200
4800/4800 [=====] - 251s 52ms/step - loss: 1.4335e-04 - binary_accuracy: 0.3478 - val_loss: 0.6797 - val_binary_accuracy: 0.4253
Epoch 5/200
4800/4800 [=====] - 253s 53ms/step - loss: 2.8838e-05 - binary_accuracy: 0.4253 - val_loss: 0.6646 - val_binary_accuracy: 0.5069
Epoch 6/200
4800/4800 [=====] - 253s 53ms/step - loss: 2.8847e-05 - binary_accuracy: 0.5069 - val_loss: 0.6615 - val_binary_accuracy: 0.5378
Epoch 7/200
4800/4800 [=====] - 251s 52ms/step - loss: 2.8293e-05 - binary_accuracy: 0.5378 - val_loss: 573.1028 - val_binary_accuracy: 0.5181
Epoch 8/200
4800/4800 [=====] - 254s 53ms/step - loss: 0.0065 - binary_accuracy: 0.5181 - val_loss: 30.6064 - val_binary_accuracy: 0.5890
Epoch 9/200
4800/4800 [=====] - 253s 53ms/step - loss: 2.8882e-05 - binary_accuracy: 0.5890 - val_loss: 6.5214 - val_binary_accuracy: 0.6903
Epoch 10/200
4800/4800 [=====] - 251s 52ms/step - loss: 2.9327e-05 - binary_accuracy: 0.6903 - val_loss: 52.9466 - val_binary_accuracy: 0.5729
Epoch 11/200
4800/4800 [=====] - ETA: 0s - loss: 2.8939e-05 - binary_accuracy: 0.5729Restoring model weights from the end of epoch 10
4800/4800 [=====] - 251s 52ms/step - loss: 2.8939e-05 - binary_accuracy: 0.5729 - val_loss: 0.6763 - val_binary_accuracy: 0.5729
Epoch 11: early stopping
```

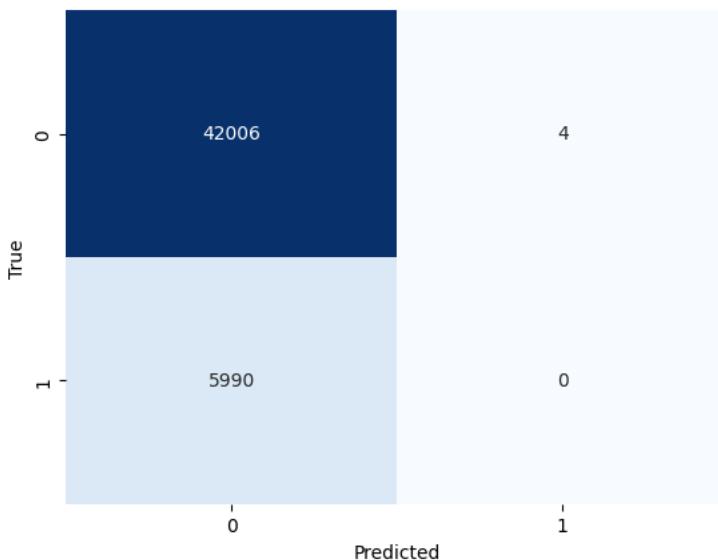
```
1 model_9, history_9 = evaluate_model(model_9_filepath,
2                                     history_9_filepath,
3                                     val_gen,
4                                     test_gen,
5                                     'models/evaluate/model_9_evaluation.pdf')
```



```
1440/1440 [=====] - 26s 18ms/step - loss: 0.6778 - binary_accuracy: 0.8581  
Validation Binary Accuracy is: 0.88  
Testing Binary Accuracy is: 0.86
```

```
1 confusion_matrix_plot(model_9, train_gen, save_file='models/evaluate/model_9_train_confusion_matrix.pdf')
2 classification_report_output(model_9, train_gen, save_file=None)
3 confusion_matrix_plot(model_9, val_gen, save_file='models/evaluate/model_9_val_confusion_matrix.pdf')
4 classification_report_output(model_9, val_gen, save_file=None)
5 confusion_matrix_plot(model_9, test_gen, save_file='models/evaluate/model_9_test_confusion_matrix.pdf')
6 classification_report_output(model_9, test_gen, save_file=None)
```

```
4800/4800 [=====] - 78s 16ms/step
```



```
4800/4800 [=====] - 78s 16ms/step
```

	precision	recall	f1-score	support
0	0.875198	0.999905	0.933404	42010.000000
1	0.000000	0.000000	0.000000	5990.000000
accuracy	0.875125	0.875125	0.875125	0.875125

▼ Model 10 XGBoost

XGBoost (Extreme Gradient Boosting) is a powerful machine learning algorithm that is widely used for various types of predictive modeling tasks. XGBoost is an ensemble learning method that combines the predictions of multiple weak prediction models to create a strong predictive model. It uses a boosting technique, where each subsequent model focuses on correcting the mistakes made by the previous models. This iterative process helps XGBoost to continually improve its predictions.

XGBoost provides a measure of feature importance, which can help you understand which features from the wearable device data are most influential in predicting stress levels. This information can guide feature selection and potentially lead to insights about the underlying factors contributing to stress.

Evaluation

The XGBoost model performs well overall. The model performs well in terms of accuracy and precision for both the validation and testing sets. The F1-score for the stress class (class 1) is higher than all previous models.

Feature Importance

The XGBoost feature importance results indicate that physiological measurements related to heart rate, electrodermal activity, respiration rate, and temperature are important factors for predicting stress using the XGBoost model. The longer-term averages and standard deviations (300-second window) seem to carry more weight in the prediction compared to the shorter-term (60-second window) features.

```
1590/1590 [=====] - 26s 16ms/step
```

```
1 # XGBoost and Random Forests
2 import numpy as np
3 from xgboost import XGBClassifier
4 from sklearn.ensemble import RandomForestClassifier
5 from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
6
7 # Extract features and labels from the TimeseriesGenerator
8 X_train = np.concatenate([batch[0] for batch in train_gen])
9 y_train = np.concatenate([batch[1] for batch in train_gen])
10
11 X_val = np.concatenate([batch[0] for batch in val_gen])
12 y_val = np.concatenate([batch[1] for batch in val_gen])
13
14 X_test = np.concatenate([batch[0] for batch in test_gen])
15 y_test = np.concatenate([batch[1] for batch in test_gen])
16
17 # Reshape the data into a 2D format
18 X_train_2d = X_train.reshape(X_train.shape[0], -1)
19 X_val_2d = X_val.reshape(X_val.shape[0], -1)
20 X_test_2d = X_test.reshape(X_test.shape[0], -1)
```

```

1 X_train_2d.shape
(48000, 6000)

1 len(X_train)
48000

        precision    recall  f1-score   support

1 X_train = train_df.drop(['label', 'subject', 'duration'], axis=1)
2 y_train = train_df['label']
3
4 X_val = val_df.drop(['label', 'subject', 'duration'], axis=1)
5 y_val = val_df['label']
6
7 X_test = test_df.drop(['label', 'subject', 'duration'], axis=1)
8 y_test = test_df['label']

     warn_pref/average  modifier  mco_start  len(result)\

1 from sklearn.model_selection import GridSearchCV
2 import joblib
3
4 # Define the hyperparameters to search over
5 param_grid = {
6     'tree_method': ['gpu_hist'],
7     'max_depth': [3, 6, 9],
8     'n_estimators': [50, 100, 150],
9     'learning_rate': [0.01, 0.1, 0.5],
10    'class_weight': [None, class_weights_dict]
11 }
12
13 #Train an XGBoost model:
14 xgb_model = XGBClassifier(random_state=42)
15
16 # Create the GridSearchCV object
17 model_10 = GridSearchCV(
18     estimator=xgb_model,
19     param_grid=param_grid,
20     cv=5,
21     error_score='raise',
22     scoring='accuracy',
23     verbose=3,
24     n_jobs=-1 #all available GPU cores
25 )
26
27 # Fit the GridSearchCV object to the training data
28 model_10.fit(X_train, y_train)
29
30 # Save model
31 model_10_best = model_10.best_estimator_
32 joblib.dump(model_10_best, 'models/model_10.pkl')

      Fitting 5 folds for each of 54 candidates, totalling 270 fits
      ['models/model_10.pkl']

1 # Print the best hyperparameters and the corresponding validation accuracy
2 y_train_pred = model_10.predict(X_train)
3 y_val_pred = model_10.predict(X_val)
4 y_test_pred = model_10.predict(X_test)
5 print("Best hyperparameters:", model_10.best_params_)
6 print("Train accuracy:", accuracy_score(y_train, y_train_pred))
7 print("Validation accuracy:", accuracy_score(y_val, y_val_pred))
8
9 # Print confusion matrix for training set
10 print("Confusion matrix (training set):\n", confusion_matrix(y_train, y_train_pred))
11
12 # Print classification report for training set
13 print("Classification report (training set):\n", classification_report(y_train, y_train_pred))
14
15 # Print confusion matrix for validation set
16 print("Confusion matrix (validation set):\n", confusion_matrix(y_val, y_val_pred))
17
18 # Print classification report for validation set
19 print("Classification report (validation set):\n", classification_report(y_val, y_val_pred))
20
21 # Print confusion matrix for testing set
22 print("Confusion matrix (testing set):\n", confusion_matrix(y_test, y_test_pred))

```

```

23
24 # Print classification report for testing set
25 print("Classification report (testing set):\n", classification_report(y_test, y_test_pred))

Best hyperparameters: {'class_weight': None, 'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 150, 'tree_method': 'gpu_hist'}
Train accuracy: 0.998991769547325
Validation accuracy: 0.9194545454545454
Confusion matrix (training set):
[[42583  27]
 [ 22 5968]]
Classification report (training set):
      precision    recall  f1-score   support

          0       1.00     1.00     1.00    42610
          1       1.00     1.00     1.00     5990

   accuracy                           1.00    48600
  macro avg       1.00     1.00     1.00    48600
weighted avg       1.00     1.00     1.00    48600

Confusion matrix (validation set):
[[13686  881]
 [ 448 1485]]
Classification report (validation set):
      precision    recall  f1-score   support

          0       0.97     0.94     0.95    14567
          1       0.63     0.77     0.69     1933

   accuracy                           0.92    16500
  macro avg       0.80     0.85     0.82    16500
weighted avg       0.93     0.92     0.92    16500

Confusion matrix (testing set):
[[12107  850]
 [ 616 1427]]
Classification report (testing set):
      precision    recall  f1-score   support

          0       0.95     0.93     0.94    12957
          1       0.63     0.70     0.66     2043

   accuracy                           0.90    15000
  macro avg       0.79     0.82     0.80    15000
weighted avg       0.91     0.90     0.90    15000

1 from sklearn.metrics import confusion_matrix
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 y_pred = model_10.best_estimator_.predict(X_test)
6 cm = confusion_matrix(y_test, y_pred)
7
8 sns.heatmap(cm, annot=True, cmap='YlGn', fmt='d')
9 plt.title('Confusion Matrix')
10 plt.xlabel('Predicted Label')
11 plt.ylabel('True Label')
12 plt.savefig('models/evaluate/model_10_confusion_matrix.png')

```

Confusion Matrix



```
1 # Feature Importance
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import warnings
5
6 # Disable all warnings
7 warnings.filterwarnings("ignore", category=UserWarning, module="matplotlib")
8
9 # Get the booster object
10 booster = model_10.best_estimator_.get_booster()
11
12 # Get the feature importance scores
13 feature_importances = booster.get_score(importance_type="weight")
14
15 # Create a dataframe to store the feature importance scores
16 fi_df = pd.DataFrame({'feature': list(feature_importances.keys()),
17                       'importance': list(feature_importances.values())})
18
19 # Sort the dataframe by feature importance scores in descending order
20 fi_df = fi_df.sort_values(by='importance', ascending=False)
21
22 # Print the top ten most important features
23 print(fi_df.head(10))
24
25 # Create a bar graph of feature importance
26 plt.figure(figsize=(10, 6))
27 plt.barh(fi_df['feature'], fi_df['importance'], color='lightgreen')
28 plt.xlabel('Importance')
29 plt.ylabel('Feature')
30 plt.title('Feature Importance')
31
32 # Save the plot as a PNG image
33 plt.savefig('img/feature_importance.png', dpi=300, bbox_inches='tight')
```

```
    feature  importance
7    EDA_Phasic_std_300s      145.0
```

▼ Model 11

The Random Forest model is a powerful machine learning algorithm that can be used for classification tasks, such as forecasting stress using wearable device data. Random Forest is an ensemble learning method that combines multiple decision trees to make predictions. Each decision tree is built independently on different subsets of the data, and the final prediction is made by aggregating the predictions of all individual trees. This ensemble approach helps improve the model's accuracy and robustness.

Evaluation

Model 11 performed better in recall and F1-score compared to all previous models.

Feature importance

Similar to model 10 (XGBoost), model 11 indicates the most important features are the extracted statistical features for mean heart rate (ECG) and electrodermal activity over 5-min and 1-min intervals.

```
1 # Import necessary libraries
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6
7 # Load the dataset
8 df = pd.read_csv('heart_rate_mean.csv')
9
10 # Check the first few rows of the dataset
11 df.head()
12
13 # Check the shape of the dataset
14 df.shape
15
16 # Check the data types of the columns
17 df.dtypes
18
19 # Check for missing values
20 df.isnull().sum()
21
22 # Check for duplicates
23 df.duplicated().sum()
24
25 # Check for outliers
26 df.describe()
27
28 # Check for correlations
29 df.corr()
30
31 # Check for missing values again
32 df.isnull().sum()
33
34 # Check for duplicates again
35 df.duplicated().sum()
36
37 # Check for outliers again
38 df.describe()
39
40 # Check for correlations again
41 df.corr()
42
43 # Check for missing values again
44 df.isnull().sum()
45
46 # Check for duplicates again
47 df.duplicated().sum()
48
49 # Check for outliers again
50 df.describe()
51
52 # Check for correlations again
53 df.corr()
54
55 # Check for missing values again
56 df.isnull().sum()
57
58 # Check for duplicates again
59 df.duplicated().sum()
60
61 # Check for outliers again
62 df.describe()
63
64 # Check for correlations again
65 df.corr()
66
67 # Check for missing values again
68 df.isnull().sum()
69
70 # Check for duplicates again
71 df.duplicated().sum()
72
73 # Check for outliers again
74 df.describe()
75
76 # Check for correlations again
77 df.corr()
78
79 # Check for missing values again
80 df.isnull().sum()
81
82 # Check for duplicates again
83 df.duplicated().sum()
84
85 # Check for outliers again
86 df.describe()
87
88 # Check for correlations again
89 df.corr()
90
91 # Check for missing values again
92 df.isnull().sum()
93
94 # Check for duplicates again
95 df.duplicated().sum()
96
97 # Check for outliers again
98 df.describe()
99
100 # Check for correlations again
101 df.corr()
102
103 # Check for missing values again
104 df.isnull().sum()
105
106 # Check for duplicates again
107 df.duplicated().sum()
108
109 # Check for outliers again
110 df.describe()
111
112 # Check for correlations again
113 df.corr()
114
115 # Check for missing values again
116 df.isnull().sum()
117
118 # Check for duplicates again
119 df.duplicated().sum()
120
121 # Check for outliers again
122 df.describe()
123
124 # Check for correlations again
125 df.corr()
126
127 # Check for missing values again
128 df.isnull().sum()
129
130 # Check for duplicates again
131 df.duplicated().sum()
132
133 # Check for outliers again
134 df.describe()
135
136 # Check for correlations again
137 df.corr()
138
139 # Check for missing values again
140 df.isnull().sum()
141
142 # Check for duplicates again
143 df.duplicated().sum()
144
145 # Check for outliers again
146 df.describe()
147
148 # Check for correlations again
149 df.corr()
150
151 # Check for missing values again
152 df.isnull().sum()
153
154 # Check for duplicates again
155 df.duplicated().sum()
156
157 # Check for outliers again
158 df.describe()
159
160 # Check for correlations again
161 df.corr()
162
163 # Check for missing values again
164 df.isnull().sum()
165
166 # Check for duplicates again
167 df.duplicated().sum()
168
169 # Check for outliers again
170 df.describe()
171
172 # Check for correlations again
173 df.corr()
174
175 # Check for missing values again
176 df.isnull().sum()
177
178 # Check for duplicates again
179 df.duplicated().sum()
180
181 # Check for outliers again
182 df.describe()
183
184 # Check for correlations again
185 df.corr()
186
187 # Check for missing values again
188 df.isnull().sum()
189
190 # Check for duplicates again
191 df.duplicated().sum()
192
193 # Check for outliers again
194 df.describe()
195
196 # Check for correlations again
197 df.corr()
198
199 # Check for missing values again
200 df.isnull().sum()
201
202 # Check for duplicates again
203 df.duplicated().sum()
204
205 # Check for outliers again
206 df.describe()
207
208 # Check for correlations again
209 df.corr()
210
211 # Check for missing values again
212 df.isnull().sum()
213
214 # Check for duplicates again
215 df.duplicated().sum()
216
217 # Check for outliers again
218 df.describe()
219
220 # Check for correlations again
221 df.corr()
222
223 # Check for missing values again
224 df.isnull().sum()
225
226 # Check for duplicates again
227 df.duplicated().sum()
228
229 # Check for outliers again
230 df.describe()
231
232 # Check for correlations again
233 df.corr()
234
235 # Check for missing values again
236 df.isnull().sum()
237
238 # Check for duplicates again
239 df.duplicated().sum()
240
241 # Check for outliers again
242 df.describe()
243
244 # Check for correlations again
245 df.corr()
246
247 # Check for missing values again
248 df.isnull().sum()
249
250 # Check for duplicates again
251 df.duplicated().sum()
252
253 # Check for outliers again
254 df.describe()
255
256 # Check for correlations again
257 df.corr()
258
259 # Check for missing values again
260 df.isnull().sum()
261
262 # Check for duplicates again
263 df.duplicated().sum()
264
265 # Check for outliers again
266 df.describe()
267
268 # Check for correlations again
269 df.corr()
270
271 # Check for missing values again
272 df.isnull().sum()
273
274 # Check for duplicates again
275 df.duplicated().sum()
276
277 # Check for outliers again
278 df.describe()
279
280 # Check for correlations again
281 df.corr()
282
283 # Check for missing values again
284 df.isnull().sum()
285
286 # Check for duplicates again
287 df.duplicated().sum()
288
289 # Check for outliers again
290 df.describe()
291
292 # Check for correlations again
293 df.corr()
294
295 # Check for missing values again
296 df.isnull().sum()
297
298 # Check for duplicates again
299 df.duplicated().sum()
300
301 # Check for outliers again
302 df.describe()
303
304 # Check for correlations again
305 df.corr()
306
307 # Check for missing values again
308 df.isnull().sum()
309
310 # Check for duplicates again
311 df.duplicated().sum()
312
313 # Check for outliers again
314 df.describe()
315
316 # Check for correlations again
317 df.corr()
318
319 # Check for missing values again
320 df.isnull().sum()
321
322 # Check for duplicates again
323 df.duplicated().sum()
324
325 # Check for outliers again
326 df.describe()
327
328 # Check for correlations again
329 df.corr()
330
331 # Check for missing values again
332 df.isnull().sum()
333
334 # Check for duplicates again
335 df.duplicated().sum()
336
337 # Check for outliers again
338 df.describe()
339
340 # Check for correlations again
341 df.corr()
342
343 # Check for missing values again
344 df.isnull().sum()
345
346 # Check for duplicates again
347 df.duplicated().sum()
348
349 # Check for outliers again
350 df.describe()
351
352 # Check for correlations again
353 df.corr()
354
355 # Check for missing values again
356 df.isnull().sum()
357
358 # Check for duplicates again
359 df.duplicated().sum()
360
361 # Check for outliers again
362 df.describe()
363
364 # Check for correlations again
365 df.corr()
366
367 # Check for missing values again
368 df.isnull().sum()
369
370 # Check for duplicates again
371 df.duplicated().sum()
372
373 # Check for outliers again
374 df.describe()
375
376 # Check for correlations again
377 df.corr()
378
379 # Check for missing values again
380 df.isnull().sum()
381
382 # Check for duplicates again
383 df.duplicated().sum()
384
385 # Check for outliers again
386 df.describe()
387
388 # Check for correlations again
389 df.corr()
390
391 # Check for missing values again
392 df.isnull().sum()
393
394 # Check for duplicates again
395 df.duplicated().sum()
396
397 # Check for outliers again
398 df.describe()
399
400 # Check for correlations again
401 df.corr()
402
403 # Check for missing values again
404 df.isnull().sum()
405
406 # Check for duplicates again
407 df.duplicated().sum()
408
409 # Check for outliers again
410 df.describe()
411
412 # Check for correlations again
413 df.corr()
414
415 # Check for missing values again
416 df.isnull().sum()
417
418 # Check for duplicates again
419 df.duplicated().sum()
420
421 # Check for outliers again
422 df.describe()
423
424 # Check for correlations again
425 df.corr()
426
427 # Check for missing values again
428 df.isnull().sum()
429
430 # Check for duplicates again
431 df.duplicated().sum()
432
433 # Check for outliers again
434 df.describe()
435
436 # Check for correlations again
437 df.corr()
438
439 # Check for missing values again
440 df.isnull().sum()
441
442 # Check for duplicates again
443 df.duplicated().sum()
444
445 # Check for outliers again
446 df.describe()
447
448 # Check for correlations again
449 df.corr()
450
451 # Check for missing values again
452 df.isnull().sum()
453
454 # Check for duplicates again
455 df.duplicated().sum()
456
457 # Check for outliers again
458 df.describe()
459
460 # Check for correlations again
461 df.corr()
462
463 # Check for missing values again
464 df.isnull().sum()
465
466 # Check for duplicates again
467 df.duplicated().sum()
468
469 # Check for outliers again
470 df.describe()
471
472 # Check for correlations again
473 df.corr()
474
475 # Check for missing values again
476 df.isnull().sum()
477
478 # Check for duplicates again
479 df.duplicated().sum()
480
481 # Check for outliers again
482 df.describe()
483
484 # Check for correlations again
485 df.corr()
486
487 # Check for missing values again
488 df.isnull().sum()
489
490 # Check for duplicates again
491 df.duplicated().sum()
492
493 # Check for outliers again
494 df.describe()
495
496 # Check for correlations again
497 df.corr()
498
499 # Check for missing values again
500 df.isnull().sum()
501
502 # Check for duplicates again
503 df.duplicated().sum()
504
505 # Check for outliers again
506 df.describe()
507
508 # Check for correlations again
509 df.corr()
510
511 # Check for missing values again
512 df.isnull().sum()
513
514 # Check for duplicates again
515 df.duplicated().sum()
516
517 # Check for outliers again
518 df.describe()
519
520 # Check for correlations again
521 df.corr()
522
523 # Check for missing values again
524 df.isnull().sum()
525
526 # Check for duplicates again
527 df.duplicated().sum()
528
529 # Check for outliers again
530 df.describe()
531
532 # Check for correlations again
533 df.corr()
534
535 # Check for missing values again
536 df.isnull().sum()
537
538 # Check for duplicates again
539 df.duplicated().sum()
540
541 # Check for outliers again
542 df.describe()
543
544 # Check for correlations again
545 df.corr()
546
547 # Check for missing values again
548 df.isnull().sum()
549
550 # Check for duplicates again
551 df.duplicated().sum()
552
553 # Check for outliers again
554 df.describe()
555
556 # Check for correlations again
557 df.corr()
558
559 # Check for missing values again
560 df.isnull().sum()
561
562 # Check for duplicates again
563 df.duplicated().sum()
564
565 # Check for outliers again
566 df.describe()
567
568 # Check for correlations again
569 df.corr()
570
571 # Check for missing values again
572 df.isnull().sum()
573
574 # Check for duplicates again
575 df.duplicated().sum()
576
577 # Check for outliers again
578 df.describe()
579
580 # Check for correlations again
581 df.corr()
582
583 # Check for missing values again
584 df.isnull().sum()
585
586 # Check for duplicates again
587 df.duplicated().sum()
588
589 # Check for outliers again
590 df.describe()
591
592 # Check for correlations again
593 df.corr()
594
595 # Check for missing values again
596 df.isnull().sum()
597
598 # Check for duplicates again
599 df.duplicated().sum()
600
601 # Check for outliers again
602 df.describe()
603
604 # Check for correlations again
605 df.corr()
606
607 # Check for missing values again
608 df.isnull().sum()
609
610 # Check for duplicates again
611 df.duplicated().sum()
612
613 # Check for outliers again
614 df.describe()
615
616 # Check for correlations again
617 df.corr()
618
619 # Check for missing values again
620 df.isnull().sum()
621
622 # Check for duplicates again
623 df.duplicated().sum()
624
625 # Check for outliers again
626 df.describe()
627
628 # Check for correlations again
629 df.corr()
630
631 # Check for missing values again
632 df.isnull().sum()
633
634 # Check for duplicates again
635 df.duplicated().sum()
636
637 # Check for outliers again
638 df.describe()
639
640 # Check for correlations again
641 df.corr()
642
643 # Check for missing values again
644 df.isnull().sum()
645
646 # Check for duplicates again
647 df.duplicated().sum()
648
649 # Check for outliers again
650 df.describe()
651
652 # Check for correlations again
653 df.corr()
654
655 # Check for missing values again
656 df.isnull().sum()
657
658 # Check for duplicates again
659 df.duplicated().sum()
660
661 # Check for outliers again
662 df.describe()
663
664 # Check for correlations again
665 df.corr()
666
667 # Check for missing values again
668 df.isnull().sum()
669
670 # Check for duplicates again
671 df.duplicated().sum()
672
673 # Check for outliers again
674 df.describe()
675
676 # Check for correlations again
677 df.corr()
678
679 # Check for missing values again
680 df.isnull().sum()
681
682 # Check for duplicates again
683 df.duplicated().sum()
684
685 # Check for outliers again
686 df.describe()
687
688 # Check for correlations again
689 df.corr()
690
691 # Check for missing values again
692 df.isnull().sum()
693
694 # Check for duplicates again
695 df.duplicated().sum()
696
697 # Check for outliers again
698 df.describe()
699
700 # Check for correlations again
701 df.corr()
702
703 # Check for missing values again
704 df.isnull().sum()
705
706 # Check for duplicates again
707 df.duplicated().sum()
708
709 # Check for outliers again
710 df.describe()
711
712 # Check for correlations again
713 df.corr()
714
715 # Check for missing values again
716 df.isnull().sum()
717
718 # Check for duplicates again
719 df.duplicated().sum()
720
721 # Check for outliers again
722 df.describe()
723
724 # Check for correlations again
725 df.corr()
726
727 # Check for missing values again
728 df.isnull().sum()
729
730 # Check for duplicates again
731 df.duplicated().sum()
732
733 # Check for outliers again
734 df.describe()
735
736 # Check for correlations again
737 df.corr()
738
739 # Check for missing values again
740 df.isnull().sum()
741
742 # Check for duplicates again
743 df.duplicated().sum()
744
745 # Check for outliers again
746 df.describe()
747
748 # Check for correlations again
749 df.corr()
750
751 # Check for missing values again
752 df.isnull().sum()
753
754 # Check for duplicates again
755 df.duplicated().sum()
756
757 # Check for outliers again
758 df.describe()
759
760 # Check for correlations again
761 df.corr()
762
763 # Check for missing values again
764 df.isnull().sum()
765
766 # Check for duplicates again
767 df.duplicated().sum()
768
769 # Check for outliers again
770 df.describe()
771
772 # Check for correlations again
773 df.corr()
774
775 # Check for missing values again
776 df.isnull().sum()
777
778 # Check for duplicates again
779 df.duplicated().sum()
780
781 # Check for outliers again
782 df.describe()
783
784 # Check for correlations again
785 df.corr()
786
787 # Check for missing values again
788 df.isnull().sum()
789
790 # Check for duplicates again
791 df.duplicated().sum()
792
793 # Check for outliers again
794 df.describe()
795
796 # Check for correlations again
797 df.corr()
798
799 # Check for missing values again
800 df.isnull().sum()
801
802 # Check for duplicates again
803 df.duplicated().sum()
804
805 # Check for outliers again
806 df.describe()
807
808 # Check for correlations again
809 df.corr()
810
811 # Check for missing values again
812 df.isnull().sum()
813
814 # Check for duplicates again
815 df.duplicated().sum()
816
817 # Check for outliers again
818 df.describe()
819
820 # Check for correlations again
821 df.corr()
822
823 # Check for missing values again
824 df.isnull().sum()
825
826 # Check for duplicates again
827 df.duplicated().sum()
828
829 # Check for outliers again
830 df.describe()
831
832 # Check for correlations again
833 df.corr()
834
835 # Check for missing values again
836 df.isnull().sum()
837
838 # Check for duplicates again
839 df.duplicated().sum()
840
841 # Check for outliers again
842 df.describe()
843
844 # Check for correlations again
845 df.corr()
846
847 # Check for missing values again
848 df.isnull().sum()
849
850 # Check for duplicates again
851 df.duplicated().sum()
852
853 # Check for outliers again
854 df.describe()
855
856 # Check for correlations again
857 df.corr()
858
859 # Check for missing values again
860 df.isnull().sum()
861
862 # Check for duplicates again
863 df.duplicated().sum()
864
865 # Check for outliers again
866 df.describe()
867
868 # Check for correlations again
869 df.corr()
870
871 # Check for missing values again
872 df.isnull().sum()
873
874 # Check for duplicates again
875 df.duplicated().sum()
876
877 # Check for outliers again
878 df.describe()
879
880 # Check for correlations again
881 df.corr()
882
883 # Check for missing values again
884 df.isnull().sum()
885
886 # Check for duplicates again
887 df.duplicated().sum()
888
889 # Check for outliers again
890 df.describe()
891
892 # Check for correlations again
893 df.corr()
894
895 # Check for missing values again
896 df.isnull().sum()
897
898 # Check for duplicates again
899 df.duplicated().sum()
900
901 # Check for outliers again
902 df.describe()
903
904 # Check for correlations again
905 df.corr()
906
907 # Check for missing values again
908 df.isnull().sum()
909
910 # Check for duplicates again
911 df.duplicated().sum()
912
913 # Check for outliers again
914 df.describe()
915
916 # Check for correlations again
917 df.corr()
918
919 # Check for missing values again
920 df.isnull().sum()
921
922 # Check for duplicates again
923 df.duplicated().sum()
924
925 # Check for outliers again
926 df.describe()
927
928 # Check for correlations again
929 df.corr()
930
931 # Check for missing values again
932 df.isnull().sum()
933
934 # Check for duplicates again
935 df.duplicated().sum()
936
937 # Check for outliers again
938 df.describe()
939
940 # Check for correlations again
941 df.corr()
942
943 # Check for missing values again
944 df.isnull().sum()
945
946 # Check for duplicates again
947 df.duplicated().sum()
948
949 # Check for outliers again
950 df.describe()
951
952 # Check for correlations again
953 df.corr()
954
955 # Check for missing values again
956 df.isnull().sum()
957
958 # Check for duplicates again
959 df.duplicated().sum()
960
961 # Check for outliers again
962 df.describe()
963
964 # Check for correlations again
965 df.corr()
966
967 # Check for missing values again
968 df.isnull().sum()
969
970 # Check for duplicates again
971 df.duplicated().sum()
972
973 # Check for outliers again
974 df.describe()
975
976 # Check for correlations again
977 df.corr()
978
979 # Check for missing values again
980 df.isnull().sum()
981
982 # Check for duplicates again
983 df.duplicated().sum()
984
985 # Check for outliers again
986 df.describe()
987
988 # Check for correlations again
989 df.corr()
990
991 # Check for missing values again
992 df.isnull().sum()
993
994 # Check for duplicates again
995 df.duplicated().sum()
996
997 # Check for outliers again
998 df.describe()
999
1000 # Check for correlations again
1001 df.corr()
```

```

20
21 # Print confusion matrix for testing set
22 print("Confusion matrix (testing set):\n", confusion_matrix(y_test, y_test_pred))
23
24 # Print classification report for testing set
25 print("Classification report (testing set):\n", classification_report(y_test, y_test_pred))

Best hyperparameters: {'class_weight': {0: 2.346866932644919e-05, 1: 0.0001669449081803005}, 'max_depth': 9, 'n_estimators': 100}
Train accuracy: 0.9822633744855968
Validation accuracy: 0.8909009090909090
Confusion matrix (training set):
[[41748  862]
 [  0 5990]]
Classification report (training set):
precision    recall    f1-score   support
          0       1.00      0.98      0.99     42610
          1       0.87      1.00      0.93      5990

accuracy                           0.98     48600
macro avg       0.94      0.99      0.96     48600
weighted avg    0.98      0.98      0.98     48600

Confusion matrix (validation set):
[[13588  979]
 [ 821 1112]]
Classification report (validation set):
precision    recall    f1-score   support
          0       0.94      0.93      0.94     14567
          1       0.53      0.58      0.55      1933

accuracy                           0.89     16500
macro avg       0.74      0.75      0.75     16500
weighted avg    0.89      0.89      0.89     16500

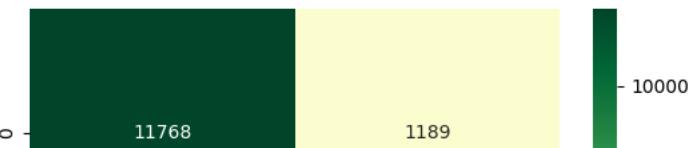
Confusion matrix (testing set):
[[11768 1189]
 [ 431 1612]]
Classification report (testing set):
precision    recall    f1-score   support
          0       0.96      0.91      0.94     12957
          1       0.58      0.79      0.67     2043

accuracy                           0.89     15000
macro avg       0.77      0.85      0.80     15000
weighted avg    0.91      0.89      0.90     15000

1 from sklearn.metrics import confusion_matrix
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 y_pred = model_11.best_estimator_.predict(x_test)
6 cm = confusion_matrix(y_test, y_pred)
7
8 sns.heatmap(cm, annot=True, cmap='YlGn', fmt='d')
9 plt.title('Confusion Matrix')
10 plt.xlabel('Predicted Label')
11 plt.ylabel('True Label')
12 plt.savefig('models/evaluate/model_11_confusion_matrix.png')

```

Confusion Matrix



```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import warnings
4
5 # Disable all warnings
6 warnings.filterwarnings("ignore")
7
8 # Replace 'X' with your actual feature data or feature names
9 feature_data = X_train # Replace with your feature data or feature names
10 feature_names = feature_data.columns.values # Replace with your feature names
11
12 # Get the feature importance scores
13 feature_importances = model_11.best_estimator_.feature_importances_
14
15 # Create a dictionary of feature importance scores with corresponding feature names
16 feature_scores = dict(zip(feature_names, feature_importances))
17
18 # Create a dataframe to store the feature importance scores
19 fi_df = pd.DataFrame.from_dict(feature_scores, orient='index', columns=['importance'])
20
21 # Sort the dataframe by feature importance scores in descending order
22 fi_df = fi_df.sort_values(by='importance', ascending=False)
23
24 # Print the top ten most important features
25 print(fi_df.head(10))
26
27 # Create a bar graph of feature importance
28 plt.figure(figsize=(10, 6))
29 plt.barh(fi_df.index, fi_df['importance'], color='lightgreen')
30 plt.xlabel('Importance')
31 plt.ylabel('Feature')
32 plt.title('Feature Importance')
33
34 # Save the plot as a PNG image
35 plt.savefig('models/evaluate/model11_feature_importance.png', dpi=300, bbox_inches='tight')
36
```

```

        importance
ECG_Rate_mean_300s    0.194850
EDA_Phasic_std_300s   0.181640
ECG_Rate_mean_60s     0.149783
ECG_Rate_std_300s    0.090755
EDA_Phasic_std_60s    0.065071
RSP_Rate_std_300s    0.046917
ECG_Rate               0.040938
Temp_mean_300s        0.036136

```

Model Evaluation

```

1 def testing_metric_table(models, test_data):
2     """
3         This function computes the classification report of multiple models' performance
4         on a given dataset. It iterates over the models, uses each model to predict the labels
5         of the data, and computes the accuracy, precision, recall, and F1-score for each class
6         using sklearn's classification_report function. The output is a Pandas dataframe containing
7         these metrics for class 1, with each model's metrics appended.
8     """
9     # Initialize an empty list to store the metrics for each model
10    metrics_list = []
11
12    # Generate model names
13    model_names = [f'Model {i}' for i in range(len(models))]
14
15    for model, model_name in zip(models, model_names):
16        # Check the type of test data
17        if isinstance(test_data, TimeseriesGenerator):
18            # For neural network models using TimeseriesGenerator
19            y_pred_probs = model.predict(test_data)
20            y_pred = (y_pred_probs > 0.5).astype(int)
21            y_true = np.concatenate([batch_labels for _, batch_labels in test_data])
22
23        # Compute the classification report
24        report = classification_report(y_true, y_pred, output_dict=True)
25        class_1_metrics = report['1']
26
27        # Append the metrics for the current model to the metrics list
28        metrics_list.append(class_1_metrics)
29
30    # Create a Pandas dataframe with the metrics for all models
31    metrics_df = pd.DataFrame(metrics_list, index=model_names)
32
33    return metrics_df
34
35 # Make a list of all the models
36 model_names_nn = [f'model_{i}' for i in range(10)]
37 models = [globals()[name] for name in model_names_nn]
38
39 # Make the metrics table of the testing data for all the models
40 metrics_table_nn = testing_metric_table(models, test_gen)
41 print(metrics_table_nn)
42
43
440/1440 [=====] - 2s 1ms/step
440/1440 [=====] - 7s 5ms/step
440/1440 [=====] - 11s 8ms/step
440/1440 [=====] - 11s 8ms/step
440/1440 [=====] - 4s 3ms/step
 38/1440 [.....] - ETA: 3s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unc
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
 _warn_prf(average, modifier, msg_start, len(result))
1440/1440 [=====] - 4s 3ms/step
 7/1440 [.....] - ETA: 26s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unc
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
 _warn_prf(average, modifier, msg_start, len(result))
1440/1440 [=====] - 20s 14ms/step
1440/1440 [=====] - 19s 13ms/step
 17/1440 [.....] - ETA: 9s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Unc
 _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c

```

```

_warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_warn_prf(average, modifier, msg_start, len(result))
1440/1440 [=====] - 10s 7ms/step
8/1440 [.....] - ETA: 22s/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: Ur
_WARN_PRF(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_WARN_PRF(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_WARN_PRF(average, modifier, msg_start, len(result))
1440/1440 [=====] - 24s 16ms/step
precision    recall   f1-score  support
Model 0  0.334169  0.978463  0.498193     2043
Model 1  0.336833  0.989232  0.502549     2043
Model 2  0.346193  0.988253  0.512762     2043
Model 3  0.207422  0.686735  0.318610     2043
Model 4  0.000000  0.000000  0.000000     2043
Model 5  0.000000  0.000000  0.000000     2043
Model 6  0.330596  0.670093  0.442755     2043
Model 7  0.141875  1.000000  0.248495     2043
Model 8  0.000000  0.000000  0.000000     2043
Model 9  0.000000  0.000000  0.000000     2043
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_WARN_PRF(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_WARN_PRF(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision and F-score are ill-c
_WARN_PRF(average, modifier, msg_start, len(result))

```

Model Evaluation

Metrics used

The purpose of this analysis is to be able to forecast a stressful state for a user of our wearable device hardware. In this case it is important to avoid false negative predictions while limiting false positives.

Precision

Precision, also known as the Positive Predictive Value (PPV), measures the accuracy of positive predictions made by a classifier. It is the ratio of true positives to all predicted positives. Precision focuses on the performance of the classifier when it predicts the positive class. In other words, precision quantifies how well the model avoids false positives.

Precision is particularly useful in scenarios where the cost of false positives (incorrectly identifying negative instances as positive) is high. A higher precision indicates a lower rate of false positives, while a lower precision suggests a higher rate of false positives.

Recall

Recall, also known as sensitivity or true positive rate, is a metric used to evaluate the proportion of actual positive instances (true positives) that are correctly identified by the model. Recall is particularly useful in certain scenarios, such as when the cost of false negatives is high. For example, missing the detection of stress. In such cases, the focus is on identifying as many positive instances as possible, even if it means sacrificing precision. A higher recall indicates a better ability to capture positive instances, while a lower recall suggests missing more positive instances.

F1-score

The F1-score is a metric commonly used to evaluate the performance of a classification model, particularly when dealing with imbalanced datasets. It combines the precision and recall metrics into a single score to provide a balanced assessment of the model's performance.

The F1-score is calculated as the harmonic mean of precision and recall, with equal importance given to both metrics. It is especially useful when the dataset has a significant class imbalance, such as in this dataset with 80% of the labels as non-stress. In such cases, accuracy alone can be misleading, as a model may achieve a high accuracy by simply predicting the majority class for all instances.

The F1-score takes into account the precision and recall values for that particular class. It measures how well the model performs in correctly identifying the positive instances (label 1) while minimizing both false positives and false negatives. A higher F1-score indicates better overall performance in terms of correctly identifying positive instances and avoiding false positives and false negatives.

Model Results

Model	Recall	F1-Score
Model 0	0.98	0.50
Model 1	0.99	0.50

Model	Recall	F1-Score
Model 2	0.99	0.51
Model 3	0.69	0.32
Model 4	0.00	0.00
Model 5	0.00	0.00
Model 6	0.67	0.44
Model 7	1.00	0.25
Model 8	0.00	0.00
Model 9	0.00	0.00
Model10	0.70	0.66
Model11	0.79	0.67

- In terms of F1-score of the stress class, model 11 performed better than all other models.
- Models 0, 1, and 2 demonstrate relatively high recall values, indicating that they are effective at correctly predicting the stress cases.
- Models 3, 6, 10, and 11 show moderate to high recall values, suggesting they also perform well in identifying positive cases.
- Models 4, 5, 8, and 9 have recall values of 0, indicating that they fail to identify any positive cases.
- Models 4, 5, 8, and 9 have F1-scores of 0, indicating poor performance in correctly forecasting stress.
- Model 7 achieves perfect recall (1.0), meaning it correctly identifies all positive cases, but has a low F1-score, indicating it is only predicting positive cases.

Overall, model 11 is the best performing model as it has a high F1-score and indicates a good balance between predicting normal and stress classes.

Feature Importance

Random Forest Classifiers allows us to extract the most important weighted features in the model. According to Model 10 the most important features are:

Feature
1 ECG_Rate_mean_300s
2 EDA_Phasic_std_300s
3 ECG_Rate_mean_60s
4 ECG_Rate_std_300s
5 EDA_Phasic_std_60s

Conclusions

Physiological data, such as heart rate, electrodermal activity (EDA), body temperature, and respiration rate, can provide valuable insights into the emotional and physical state of a person. These physiological parameters are influenced by the autonomic nervous system, which regulates the body's response to various stimuli and can reflect changes in emotional arousal, stress levels, and physical well-being.

- Heart Rate: Heart rate is the number of times the heart beats per minute and is influenced by factors such as physical exertion, stress, and emotional arousal. Higher heart rate can indicate increased physiological arousal, which may be associated with emotions like excitement, anxiety, or fear. Changes in heart rate variability (HRV), the variation in time intervals between heartbeats, can also provide information about emotional regulation and stress levels.
- Electrodermal Activity (EDA): EDA measures the electrical conductance of the skin, which is influenced by sweat gland activity. EDA is commonly used as an indicator of sympathetic nervous system activity, which is associated with emotional arousal and stress. Increased EDA may reflect heightened emotional responses, such as excitement, fear, or anxiety.
- Body Temperature: Body temperature can fluctuate based on environmental conditions, physical activity, and emotional states. Increased body temperature may occur during periods of physical exertion, stress, or emotional arousal. Conversely, decreased body temperature may indicate relaxation or a lower emotional state.
- Respiration Rate: Respiration rate refers to the number of breaths taken per minute. Emotional and physical states can impact respiration patterns. For instance, during states of stress or anxiety, respiration rate may increase, leading to rapid and shallow breathing. In contrast, during calm or relaxed states, respiration rate tends to be slower and deeper.

Machine learning algorithms and statistical techniques can be applied to these data to develop models that predict emotional states, stress levels, or physical conditions, such as stress forecasting.

The best models according to the F1-score and recall is Model 11 using a Random Forest Classifier.

Model 10 (XGBoost) and Model 2 (deep LSTM) are considered to be only slightly better than guessing because their performance metrics of recall and F1-score are not significantly higher than random chance. In the context of stress prediction, these models exhibit limited predictive power and may not provide reliable or accurate forecasts.

Additionally, the performance of these models may be attributed to the limited dataset used for training. Both models were trained on a specific

