

```
In [1]: %matplotlib inline
import os, shutil
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as scio
from scipy.fft import fft, ifft, fftfreq
from scipy.signal import stft, ricker, cwt, butter, iirnotch, lfilter
import scipy.stats as stats
import pywt
from tqdm import tqdm
import pandas as pd
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import shuffle

from sklearn.inspection import DecisionBoundaryDisplay
from matplotlib.colors import ListedColormap

import torch

import mne

from statsmodels.stats.weightstats import ztest
```

```
In [ ]:
```

```
In [2]: mode = 'notebook' #notebook, commit, colab

import plotly.offline as pyo
import plotly.graph_objs as go
from plotly.offline import iplot

from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot

if mode == 'colab':
    import cufflinks as cf
    cf.go_offline()
    init_notebook_mode(connected=False)

def configure_plotly_browser_state():
    import IPython
    display(IPython.core.display.HTML('''
        <script src="/static/components/requirejs/require.js"></script>
        <script>
            requirejs.config({
                paths: {
                    base: '/static/base',
                    plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',
                },
            });
        </script>
    '''))

import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
import plotly.io as pio

if mode == 'commit':
    pio.renderers.default = "png"
```

```
In [3]: mne.set_log_level(verbose='ERROR')
```

# Dataset Processing

- Source: [P300 speller with ALS patients \(<http://bncl-horizon-2020.eu/database/data-sets>\)](http://bncl-horizon-2020.eu/database/data-sets)
- Paper: [Attention and P300-based BCI performance in people with amyotrophic lateral sclerosis \(<https://doi.org/10.3389/fnhum.2013.00732>\)](https://doi.org/10.3389/fnhum.2013.00732)

## Extract from the paper (Experimental protocol):

Scalp EEG signals were recorded (g.MOBILAB, g.tec, Austria) from eight channels according to 10–10 standard (Fz, Cz, Pz, Oz, P3, P4, PO7 and PO8; Chatrian et al., 1985; Krusienski et al., 2006) using active electrodes (g.Ladybird, g.tec, Austria). All channels were referenced to the right earlobe and grounded to the left mastoid. The EEG signal was digitized at 256 Hz. Data acquisition and stimuli delivery were managed by the BCI2000 framework (Schalk et al., 2004).

Participants were required to copy spell seven predefined words of five characters each (runs), by controlling a P300 speller (Farwell and Donchin, 1988). The latter consisted of a 6 by 6 matrix containing alphanumeric characters (Figure 1A). Rows and columns on the interface were randomly intensified for 125 ms, with an inter stimulus interval (ISI) of 125 ms, yielding a 250 ms lag between the appearance of two stimuli (stimulus onset asynchrony, SOA). For each character selection (trial) all rows and columns were intensified 10 times (stimuli repetitions) thus each single item on the interface was intensified 20 times. Participants were seated facing a 15" computer screen placed at eye level approximately one meter in front of them. The angular distance subtended by the speller was of 15 degrees. A single flash of a letter at the beginning of each trial cued the target to focus. In the first three runs (15 trials in total) EEG data was stored to perform a calibration of the BCI classifier. Thus no feedback was provided to the participant up to this point. A stepwise linear discriminant analysis (SWLDA) was applied to the data from the three calibration runs (i.e., runs 1–3) to determine the classifier weights (i.e., classifier coefficients) (Krusienski et al., 2006). These weights were then applied during the subsequent four testing runs (i.e., testing set; runs 4–7) when participants were provided with feedback. EEG potentials between 0 and 800 ms after each stimulus onset were decimated by replacing each sequence of 12 samples with their mean value and used for the analysis. The next four runs (20 trials in total) characterized a testing phase in which feedback was provided by showing each spelled character. In cases of error the feedback was represented by a dot (instead of the wrongly typed character) to minimize frustration of the participants.

```
In [4]: def load_data(filepath):
    """
    Loads EEG signal from the dataset
    """
    sample_matlab_data = scio.loadmat(filepath) # Structured Matlab Array containing all information
    return sample_matlab_data

basedir = 'DataFiles'
```

```
In [5]: patient_data = []
for file in os.listdir(basedir):
    filepath = os.path.join(basedir, file)
    patient_data.append(load_data(filepath))

print(len(patient_data))
```

8

```
In [6]: patient_data[0]
```

```
Out[6]: {'__header__': b'MATLAB 5.0 MAT-file, Platform: PCWIN, Created on: Tue Nov 25
12:46:00 2014',
 '__version__': '1.0',
 '__globals__': [],
 'data': array([[(array([[array(['Fz'], dtype='<U2'), array(['Cz'], dtype='<U
2'),
          array(['Pz'], dtype='<U2'), array(['Oz'], dtype='<U2'),
          array(['P3'], dtype='<U2'), array(['P4'], dtype='<U2'),
          array(['PO7'], dtype='<U3'), array(['PO8'], dtype='<U3')]],,
          dtype=object), array([-1.21978372, -0.32769319, -1.2199583
1, ..., -2.65680554,
          7.37202505, -0.27890203],
         [-1.42616224, -4.18996393, -5.65458822, ..., -9.45534598,
          -3.67053188, -5.48506363],
         [-1.31524453, -6.87075831, -8.55992067, ..., -14.26798757,
          -11.63200572, -8.92900433],
         ...,
         [-3.96511643, 5.70748743, 1.11047492, ..., -5.27998942,
          9.13724881, -0.85580654],
         [-0.5562302, 6.01519786, 3.32213104, ..., -0.67030739,
          8.2749988, 2.10576813],
         [4.22622461, 6.77563943, 6.12924025, ..., 5.20342677,
          6.29436858, 6.15256883]]), array([[0],
         [0],
         [0],
         ...,
         [0],
         [0],
         [0],
         [0]]], dtype=uint8), array([[0],
         [0],
         [0],
         ...,
         [0],
         [0],
         [0],
         [0]]], dtype=int16), array([[ 1801, 11529, 21257, 30985,
40713, 51473, 61201, 70929,
          80657, 90385, 101145, 110873, 120601, 130329, 140057, 1508
17,
          160545, 170273, 180001, 189729, 200489, 210217, 219945, 2296
73,
          239401, 250161, 259889, 269617, 279345, 289073, 299833, 3095
61,
          319289, 329017, 338745]]), array([[array(['NonTarget'], dtype='<U9'),
          array(['Target'], dtype='<U6')]], dtype=object), array([[array(['Row1'], dtype='<U4'), array(['Row2'], dtype='<U4'),
          array(['Row3'], dtype='<U4'), array(['Row4'], dtype='<U4'),
          array(['Row5'], dtype='<U4'), array(['Row6'], dtype='<U4'),
          array(['Col1'], dtype='<U4'), array(['Col2'], dtype='<U4'),
          array(['Col3'], dtype='<U4'), array(['Col4'], dtype='<U4'),
          array(['Col5'], dtype='<U4'), array(['Col6'], dtype='<U
4')]],
          dtype=object), array(['male'], dtype='<U4'), array(['55'],
          dtype='<U2'), array(['13'], dtype='<U2'), array(['spinal'],
          dtype='<U6))]], dtype=[('channels', 'O'), ('X', 'O'), ('y', 'O'), ('y_stim', 'O'), ('t
ime', 'O')])]
```

```
rial', '0'), ('classes', '0'), ('classes_stim', '0'), ('gender', '0'), ('age', '0'), ('ALSfrs', '0'), ('onsetALS', '0')))}
```

```
In [7]: def get_channel_name(patient, index):
    """
    Returns the name of the EEG channel for the corresponding index
    """
    return patient['data'][0][0]['channels'][0][index]
print(get_channel_name(patient_data[0], 1))

['Cz']
```

```
In [8]: def get_channel_index(patient, channel_name):
    """
    Returns the index number of a channel
    """
    indices = [data[0].lower() for data in patient['data'][0][0]['channels'][0]]
    return indices.index(channel_name.lower())
print(get_channel_index(patient_data[0], 'Fz'))
```

```
0
```

```
In [9]: print("Data shape")
print(f"Signal: {patient_data[0]['data'][0][0]['X'].shape}")
print(f"Target/Class label: {patient_data[0]['data'][0][0]['y'].shape}")
print(f"Stimulus type: {patient_data[0]['data'][0][0]['y_stim'].shape}")

Data shape
Signal: (347704, 8)
Target/Class label: (347704, 1)
Stimulus type: (347704, 1)
```

## Parameters of the EEG Signal

```
In [10]: # Functions for EEG data extraction and manipulation

sample_frequency = 256 #Hz
bandpass_filter = (0, 30) # Hz
stimulus_duration = 125 #ms
interstimulus_interval = 125 #ms
total_stimulus = 12 # 6 rows and 6 columns
number_of_times_each_stimulus_appeared_per_trial = 10
total_stimulus_appeared_per_trial = total_stimulus * number_of_times_each_stimulus_appeared_per_trial

trial_duration = (stimulus_duration * total_stimulus_appeared_per_trial) + \
(interstimulus_interval * (total_stimulus_appeared_per_trial - 1))

samples_per_trial = (sample_frequency * trial_duration)//1000

print(f"Each trial duration: {trial_duration} ms or {samples_per_trial} samples")
```

Each trial duration: 29875 ms or 7648 samples

```
In [11]: def get_trials(sample, remove_calibration_data=True):
    """
        Returns all trial data for a patient
    """
    data = sample['data'][0][0]
    signal = data['X']
    stimulus = data['y_stim']
    target = data['y']

    x, y_stimulus, y_target = [], [], []
    # trial_start = 0
    for i in range(data['trial'].shape[1]):
        trial_start = data['trial'][0, i]
        trial_end = trial_start+samples_per_trial

        # Fetch signal
        x.append(signal[trial_start:trial_end])
        # Fetch stimulus
        y_stimulus.append(stimulus[trial_start:trial_end, 0])
        # Fetch target
        y_target.append(target[trial_start:trial_end, 0])

    x, y_stimulus, y_target = np.array(x), np.array(y_stimulus), np.array(y_target)
    if remove_calibration_data:
        x, y_stimulus, y_target = x[3:], y_stimulus[3:], y_target[3:]
    return x, y_stimulus, y_target

X, y_stim, y_target = get_trials(patient_data[0])
print(X.shape, y_stim.shape, y_target.shape)
print(np.unique(y_stim))
print(np.unique(y_target))
```

(32, 7648, 8) (32, 7648) (32, 7648)  
[ 0 1 2 3 4 5 6 7 8 9 10 11 12]  
[0 1 2]

```
In [12]: number_of_trials_per_run = 5 # each run is a word and each trial is a character of the word
number_of_runs = 7 # total number of words

total_characters_with_calibration = number_of_trials_per_run * number_of_runs
total_characters_without_calibration = number_of_trials_per_run * number_of_runs - 3

def get_dataset(dataset):
    """
    Converts the matlab file of all patients to a single numpy array and returns the dataset
    Returns: Numpy array -> input shape: (patient_id, run, trial, sample, channels),
    stimulus -> (patient_id, run, trial, sample)
    target -> (patient_id, run, trial, sample)
    """
    x, y_stim, y_target = [], [], []
    for i in range(len(dataset)):
        patient = dataset[i]
        x_trial, y_stim_trial, y_target_trial = get_trials(patient, remove_calibration_data=False)

        # Reshape array to group trials by run
        x_trial = x_trial.reshape(-1, number_of_trials_per_run, x_trial.shape[1], x_trial.shape[2])
        y_stim_trial = y_stim_trial.reshape(-1, number_of_trials_per_run, y_stim_trial.shape[1])
        y_target_trial = y_target_trial.reshape(-1, number_of_trials_per_run, y_target_trial.shape[1])

        # Add to array
        x.append(x_trial)
        y_stim.append(y_stim_trial)
        y_target.append(y_target_trial)

    return np.array(x), np.array(y_stim), np.array(y_target)
```

```
X, y_stim, y_target = get_dataset(patient_data)

print(X.shape, y_stim.shape, y_target.shape)
print(np.unique(y_stim))
print(np.unique(y_target))
```

```
(8, 7, 5, 7648, 8) (8, 7, 5, 7648) (8, 7, 5, 7648)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12]
[0 1 2]
```

## Mapping for row/column value to character

```
In [13]: words = np.array([chr(i) for i in range(ord('A'), ord('Z')+1)] + [str(i) for i in range(1, 10)] + ['-'])
word_matrix = np.reshape(words, (6, 6))
word_matrix_row = np.arange(7, 13)
word_matrix_column = np.arange(1, 7)
print(word_matrix)
word_to_tup = {}
tup_to_word = {}
for i in range(word_matrix.shape[0]):
    for j in range(word_matrix.shape[1]):
        tup = (word_matrix_row[i], word_matrix_column[j])
        word_to_tup[word_matrix[i, j]] = tup
        tup_to_word[str(tup[0]) + "," + str(tup[1])] = word_matrix[i, j]

print("\nCharacter to indices")
print(word_to_tup)
print("\n Indices to Characters")
print(tup_to_word)
```

```
[[ 'A' 'B' 'C' 'D' 'E' 'F']
 ['G' 'H' 'I' 'J' 'K' 'L']
 ['M' 'N' 'O' 'P' 'Q' 'R']
 ['S' 'T' 'U' 'V' 'W' 'X']
 ['Y' 'Z' '1' '2' '3' '4']
 ['5' '6' '7' '8' '9' '-']]
```

Character to indices

```
{'A': (7, 1), 'B': (7, 2), 'C': (7, 3), 'D': (7, 4), 'E': (7, 5), 'F': (7, 6), 'G': (8, 1), 'H': (8, 2), 'I': (8, 3), 'J': (8, 4), 'K': (8, 5), 'L': (8, 6), 'M': (9, 1), 'N': (9, 2), 'O': (9, 3), 'P': (9, 4), 'Q': (9, 5), 'R': (9, 6), 'S': (10, 1), 'T': (10, 2), 'U': (10, 3), 'V': (10, 4), 'W': (10, 5), 'X': (10, 6), 'Y': (11, 1), 'Z': (11, 2), '1': (11, 3), '2': (11, 4), '3': (11, 5), '4': (11, 6), '5': (12, 1), '6': (12, 2), '7': (12, 3), '8': (12, 4), '9': (12, 5), '-': (12, 6)}
```

Indices to Characters

```
{'7,1': 'A', '7,2': 'B', '7,3': 'C', '7,4': 'D', '7,5': 'E', '7,6': 'F', '8,1': 'G', '8,2': 'H', '8,3': 'I', '8,4': 'J', '8,5': 'K', '8,6': 'L', '9,1': 'M', '9,2': 'N', '9,3': 'O', '9,4': 'P', '9,5': 'Q', '9,6': 'R', '10,1': 'S', '10,2': 'T', '10,3': 'U', '10,4': 'V', '10,5': 'W', '10,6': 'X', '11,1': 'Y', '11,2': 'Z', '11,3': '1', '11,4': '2', '11,5': '3', '11,6': '4', '12,1': '5', '12,2': '6', '12,3': '7', '12,4': '8', '12,5': '9', '12,6': '-'}
```

```
In [14]: target_class = 2
non_target_class = 1
def convert_stimulus_to_word(y_stim, y_target):
    """
        Takes all consecutive stimulus (row & column number of BCI) and outputs the
        corresponding word that was
        displayed.
        Input shape: (Patients, run, trial, samples)
        Output shape: (Patients, run, trial, character)
    """
    y_stim = y_stim.reshape(-1)
    y_target = y_target.reshape(-1)
    target_stimulus = y_stim[y_target == target_class]
    vals = []
    for i in range(len(target_stimulus)):
        if len(vals) == 0 or vals[-1] != target_stimulus[i]:
            vals.append(target_stimulus[i])

    character_indices = []
    characters = []
    for i in range(0, len(vals)-1, 2):
        c = vals[i]
        r = vals[i+1]
        character_indices.append((r, c))
        characters.append(tup_to_word(str(r) + "," + str(c)))

    non_repeating_chars = []
    for c in characters:
        if len(non_repeating_chars) == 0 or non_repeating_chars[-1] != c:
            non_repeating_chars.append(c)

    words = []
    for i in range(0, len(non_repeating_chars), 5):
        words.append(''.join(non_repeating_chars[i:i+5]))
    return words

for i in range(X.shape[0]):
    print(f"Words displayed to patient: {i+1}")
    print(convert_stimulus_to_word(y_stim[0], y_target[0]))
```

```
Words displayed to patient: 1
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
Words displayed to patient: 2
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
Words displayed to patient: 3
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
Words displayed to patient: 4
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
Words displayed to patient: 5
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
Words displayed to patient: 6
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
Words displayed to patient: 7
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
Words displayed to patient: 8
['WGAMN', 'F70DY', 'KPUAB', 'AJOCY', 'IJYVN', '06A7Y', 'GPD']
```

## High level overview of the dataset

First we will visualize the EEG signals collected from different patients. We will do a visual inspection to see some of the characteristics of the dataset e.g. what are the minimum and maximum amplitude of the signals for different patients, are there any visible artifacts in the signal, etc.

```
In [15]: get_channel_index(patient_data[0], 'PO7')
```

```
Out[15]: 6
```

```
In [16]: def view_signal(signal, title='', channels_to_show=[], highpass=None, lowpass=None, y=None, scalings='auto'):
    signal = signal.reshape(-1, signal.shape[-1]).transpose()
    channels = ['P3', 'P07', 'Fz', 'Cz', 'Pz', 'Oz', 'P4', 'P08']
    channel_indices = [get_channel_index(patient_data[0], name) for name in channels]
    #     channel_indices = [i for i in range(signal.shape[0])]
    #     channels = [get_channel_name(patient_data[0], i)[0] for i in channel_indices]
    if len(channels_to_show) == 0:
        channels_to_show = [i for i in range(len(channels))]
    else:
        channels_to_show = [channels.index(channel) for channel in channels_to_show]

    info = mne.create_info(channels, sample_frequency, ch_types='eeg')
    data = mne.io.RawArray(signal[channel_indices, :], info)

    target_descriptions = ['No stimulus', 'Non-Target', 'Target']
    target_description_indices = {i:target_descriptions[i] for i in range(len(target_descriptions))}
    target_description_names = {target_descriptions[i]: i for i in range(len(target_descriptions))}

    if not (y is None):
        y = y.reshape(-1)
        target = np.where(y == target_description_names['Target'])[0]
        non_targets = np.where(y == target_description_names['Non-Target'])[0]
        onsets = []
        duration = []
        description = []

        for i in range(len(target)):
            if i == 0:
                onsets.append(target[i])
                description.append('T')
            else:
                if target[i] != target[i-1]+1:
                    duration.append(target[i-1] - onsets[-1])
                    onsets.append(target[i])
                    description.append('T')
                elif i == len(target) - 1:
                    duration.append(target[i] - onsets[-1])

        for i in range(len(non_targets)):
            if i == 0:
                onsets.append(non_targets[i])
                description.append('N')
            else:
                if non_targets[i] != non_targets[i-1]+1:
                    duration.append(non_targets[i-1] - onsets[-1])
                    onsets.append(non_targets[i])
                    description.append('N')
                elif i == len(non_targets) - 1:
                    duration.append(non_targets[i] - onsets[-1])
```

```
onsets = np.array(onsets)/sample_frequency
duration = np.array(duration)/sample_frequency

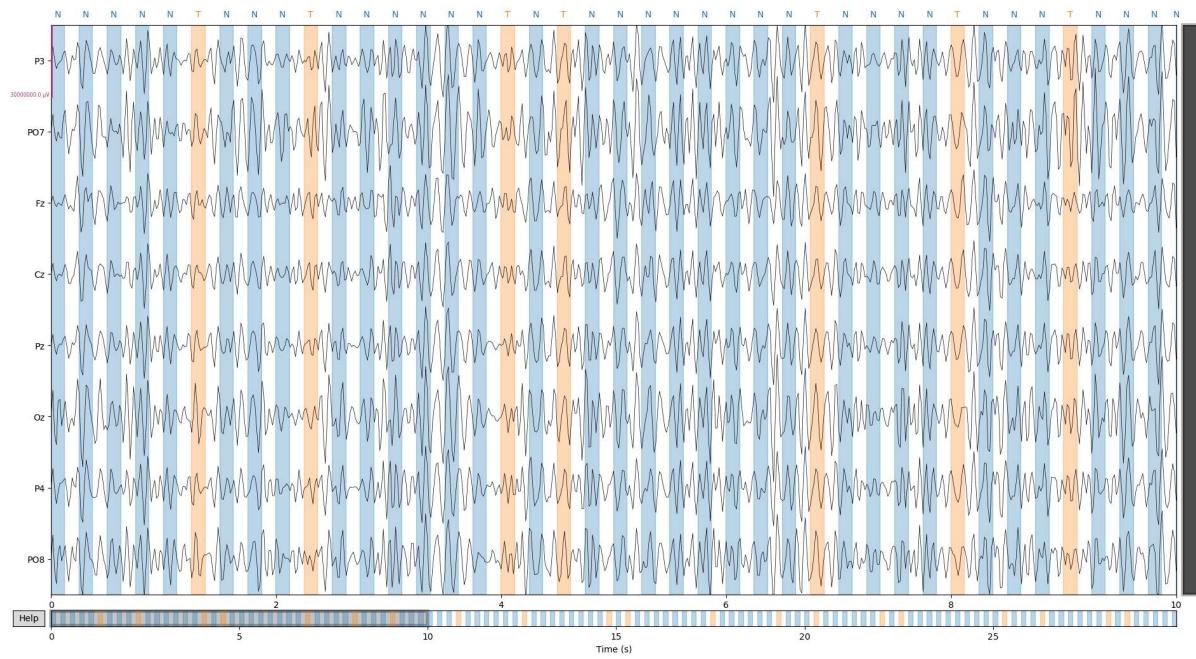
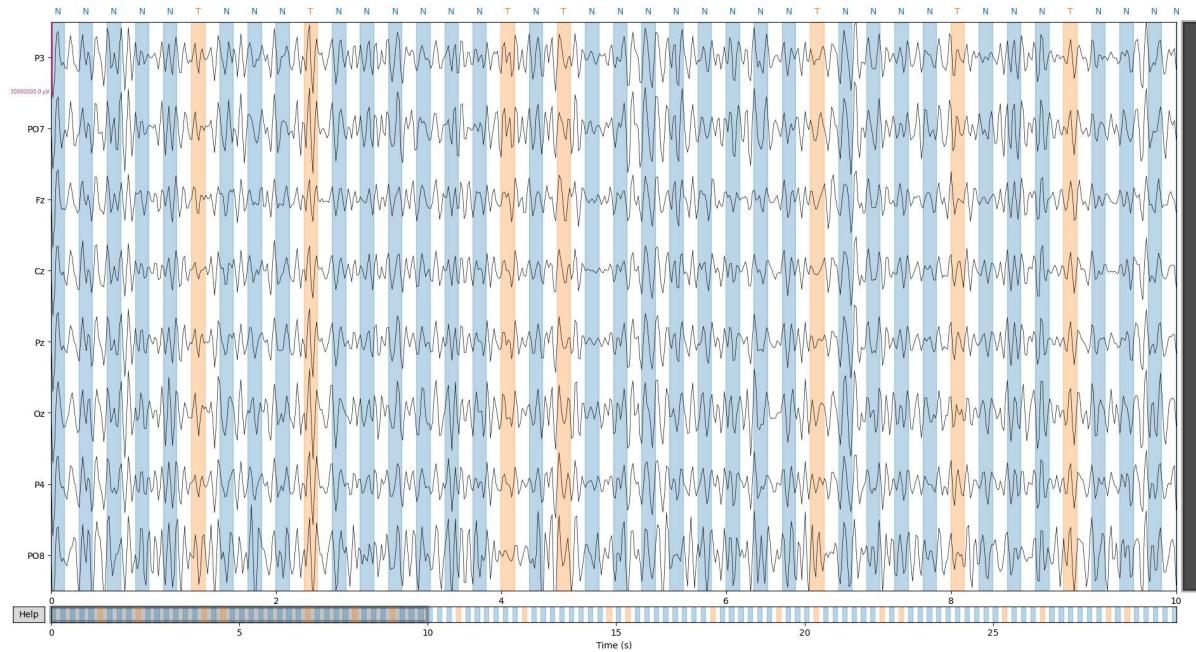
data.set_annotations(mne.Annotations(onsets, duration, description))

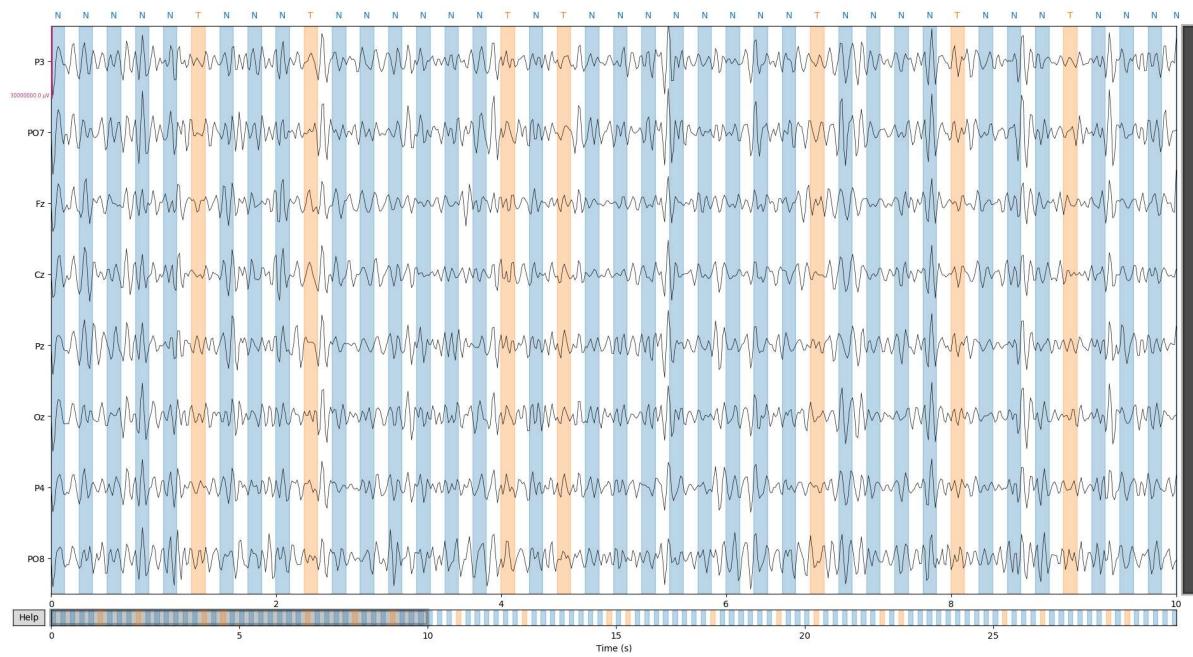
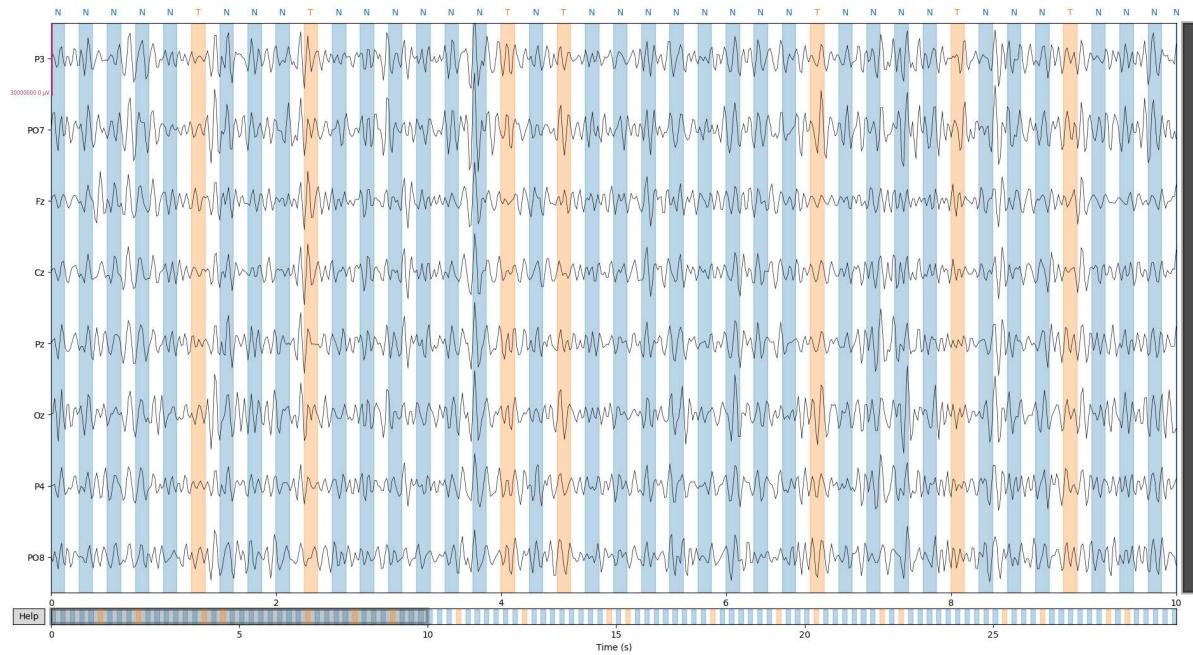
data.plot(scalings=scalings, title=title, order=channels_to_show, highpass
=highpass, lowpass=lowpass)
```

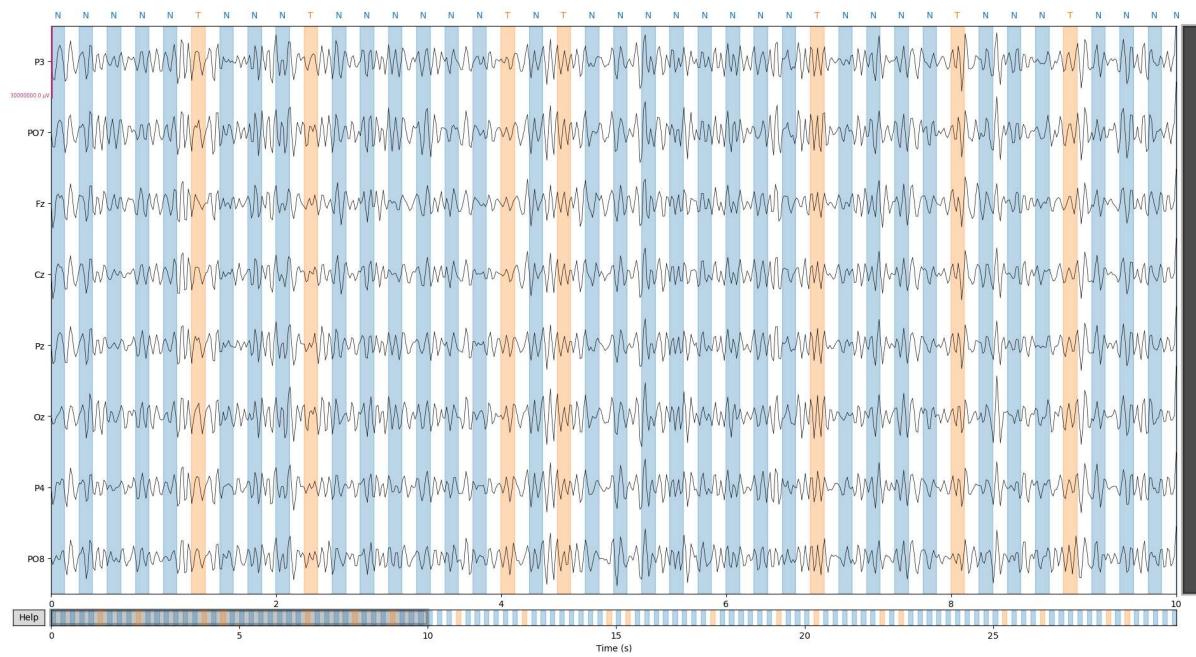
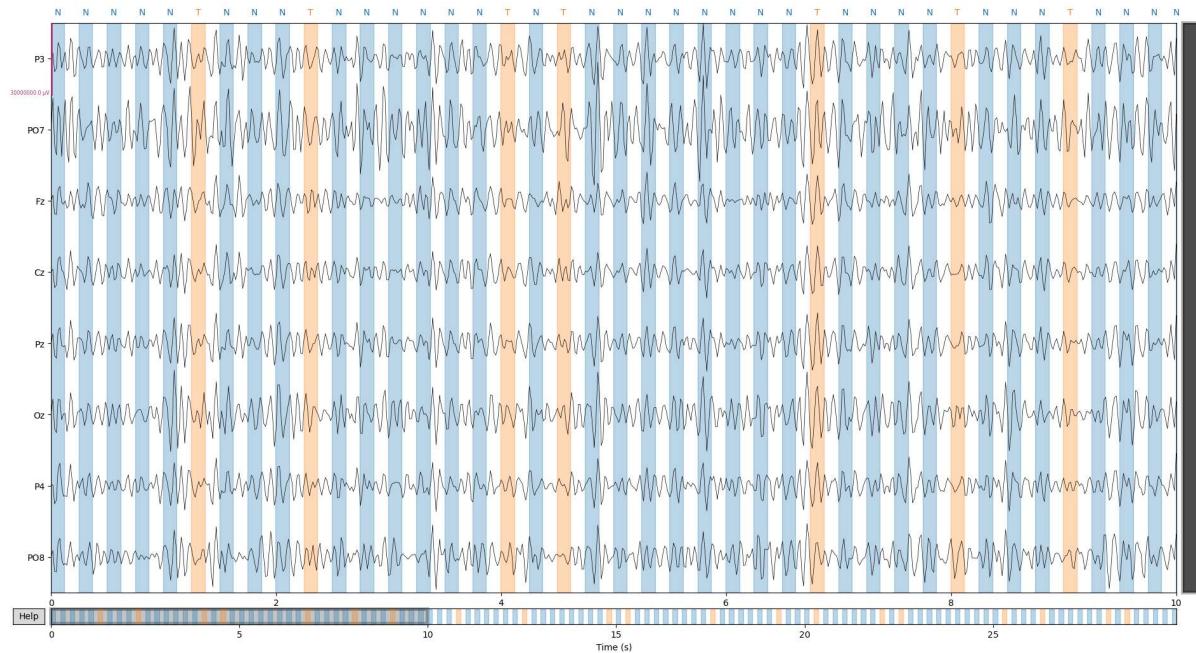
In [17]: X.shape, y\_target.shape

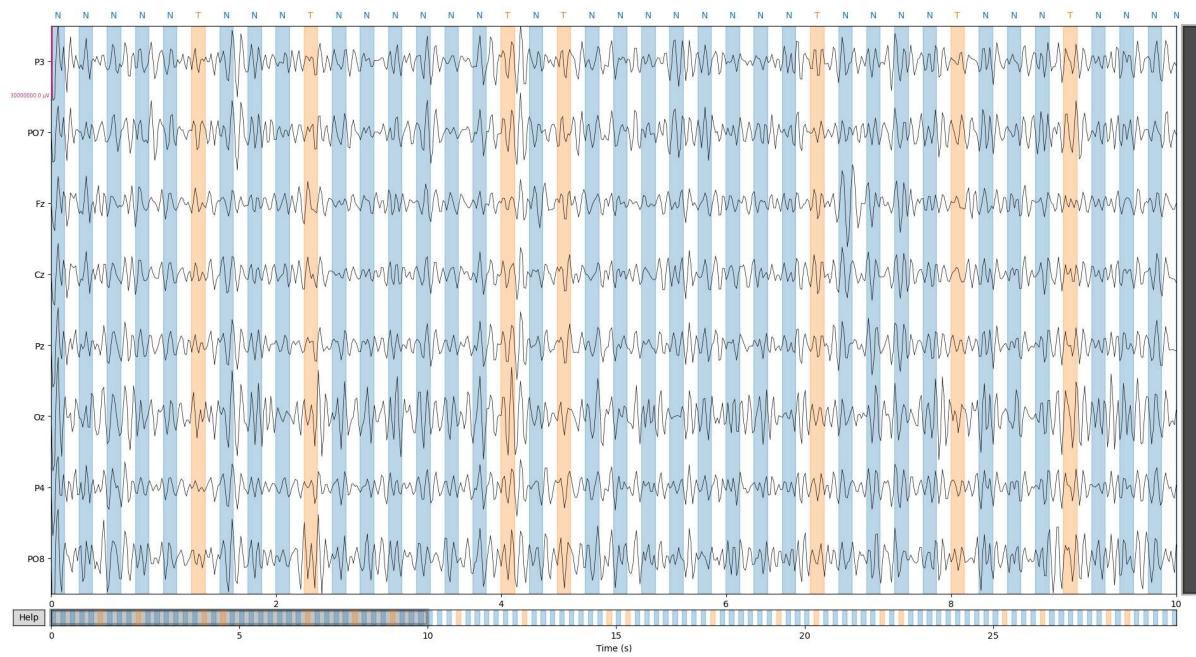
Out[17]: ((8, 7, 5, 7648, 8), (8, 7, 5, 7648))

```
In [18]: for i in range(X.shape[0]):  
    print(f"Patient {i+1}")  
    view_signal(X[i, 0, 0, :, :], title=f'Patient {i+1}', lowpass=20, highpass  
=8, y=y_target[i], scalings=dict(eeg=15))
```

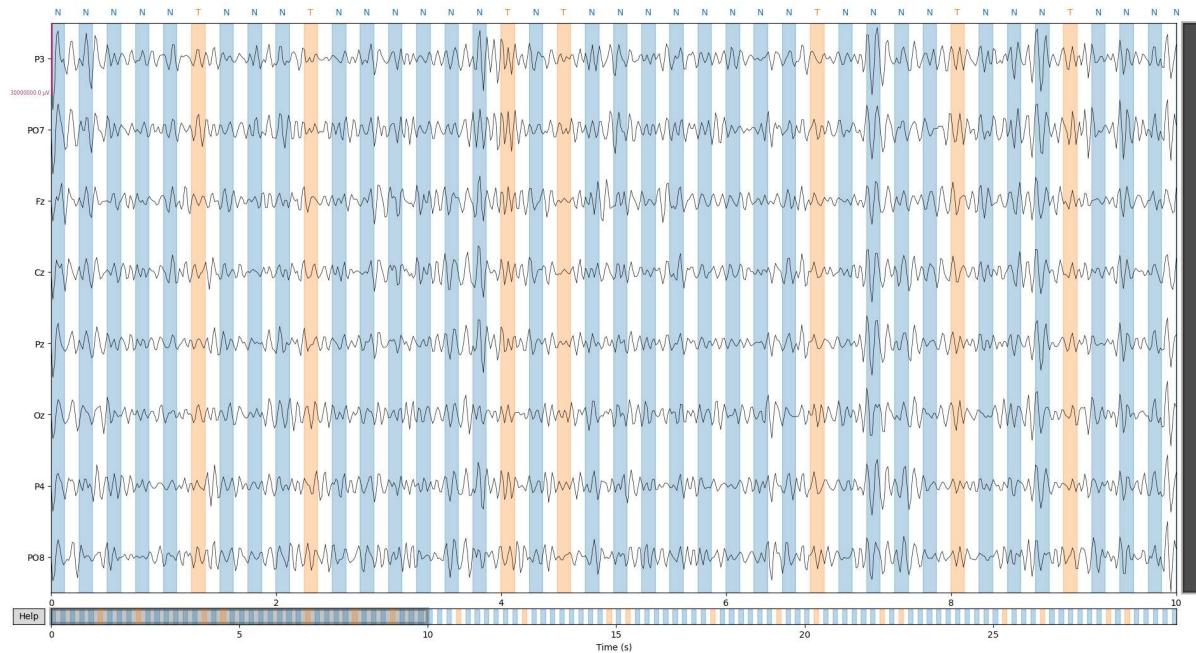
**Patient 1****Patient 2****Patient 3**

**Patient 4****Patient 5**

**Patient 6****Patient 7**



Patient 8



```
In [19]: # # View in new window
# %matplotlib qt
# patient = 7
# view_signal(X[patient, 0, 0], title='Patient 1', highpass=0.1, Lowpass=10, y=y_target[patient])
```

```
In [ ]:
```

## Removing Artifacts - Part I

In this section we will follow the same procedure mentioned in the paper to remove the artifacts and visualize the EEG signal.

### Extract from the paper (Data Analysis):

EEG data was high pass and low pass filtered with cut off frequencies of 0.1 Hz and 10 Hz respectively using a 4th order Butterworth filter. In addition, a notch filter was used to remove 50 Hz contamination due to the AC interference. Data was divided into 1000 ms long epochs starting with the onset of each stimulus. Epochs in which peak amplitude was higher than 70  $\mu$ V or lower than -70  $\mu$ V were identified as artifacts and removed. A baseline correction was done based on the average EEG activity within 200 ms immediately preceding each epoch. The average waveform for both target and non-target epochs was computed for each trial in order to assess P300 peak amplitude. Particularly, amplitude of the P300 potential in Cz was defined as the highest value of the difference between target and non-target average waveforms in the time interval 250–700 ms

```
In [20]: def butter_bandpass(lowcut, highcut, fs, order=4):
    return butter(order, [lowcut, highcut], fs=fs, btype='band')

def butter_bandpass_filter(data, lowcut, highcut, fs, order=4, axis=-1):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data, axis=axis)
    return y

def notch_filter(data, frequency, fs, axis=-1, quality_factor=30.0):
    b, a = iirnotch(frequency, quality_factor, fs)
    return lfilter(b, a, data, axis=axis)
```

```
In [21]: highpass_filter = 0.1 #Hz
lowpass_filter = 10 #Hz
notch_frequency = 50 #Hz
```

### Filter the signal

```
In [22]: X_filtered = butter_bandpass_filter(X, highpass_filter, lowpass_filter, sample_
_frequency, order=4, axis=3)
X_filtered = notch_filter(X_filtered, notch_frequency, sample_frequency, axis=
3)
print(X_filtered.shape)

(8, 7, 5, 7648, 8)
```

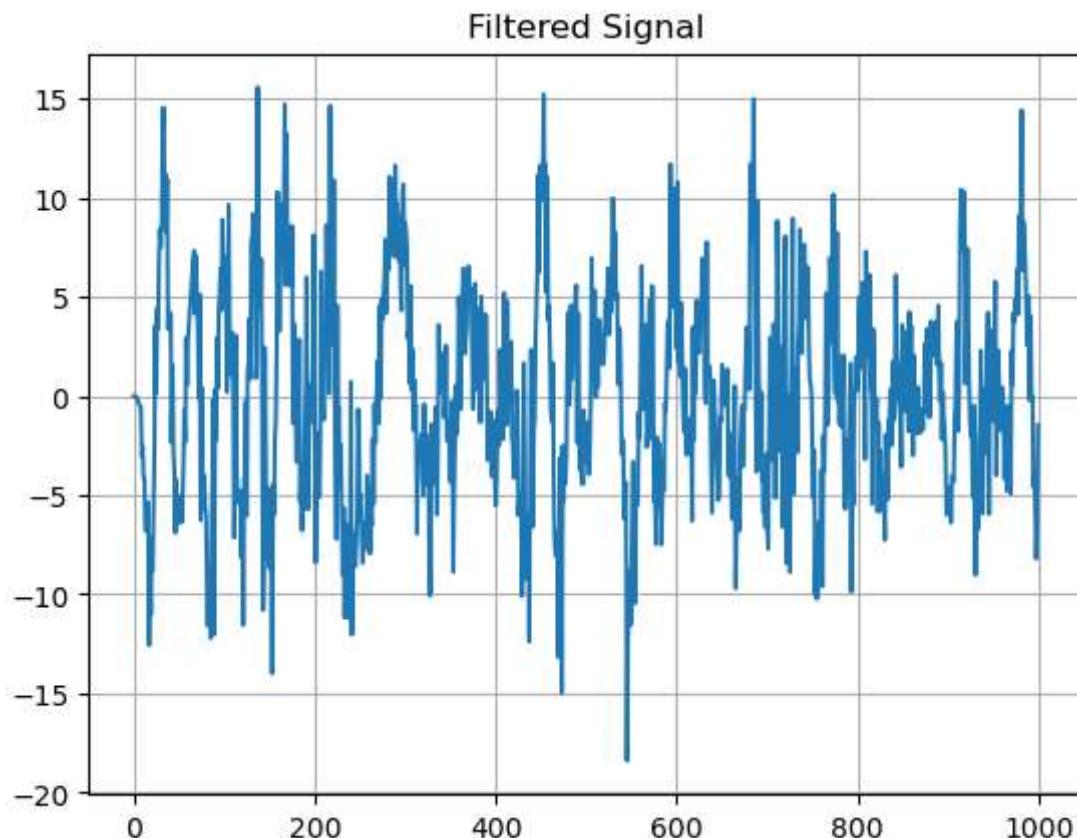
### Remove amplitude greater than 70uV or less than -70uv

```
In [23]: minimum_amplitude = -70
maximum_amplitude = 70

X_filtered[(X_filtered < minimum_amplitude) | (X_filtered > maximum_amplitude)] = 0
print(X_filtered.shape)

(8, 7, 5, 7648, 8)
```

```
In [24]: plt.figure()
plt.title("Filtered Signal")
plt.plot(X_filtered.reshape((X_filtered.shape[0], -1, X_filtered.shape[1]))[1, :1000, 0])
plt.grid()
plt.show()
```



## Baseline correction

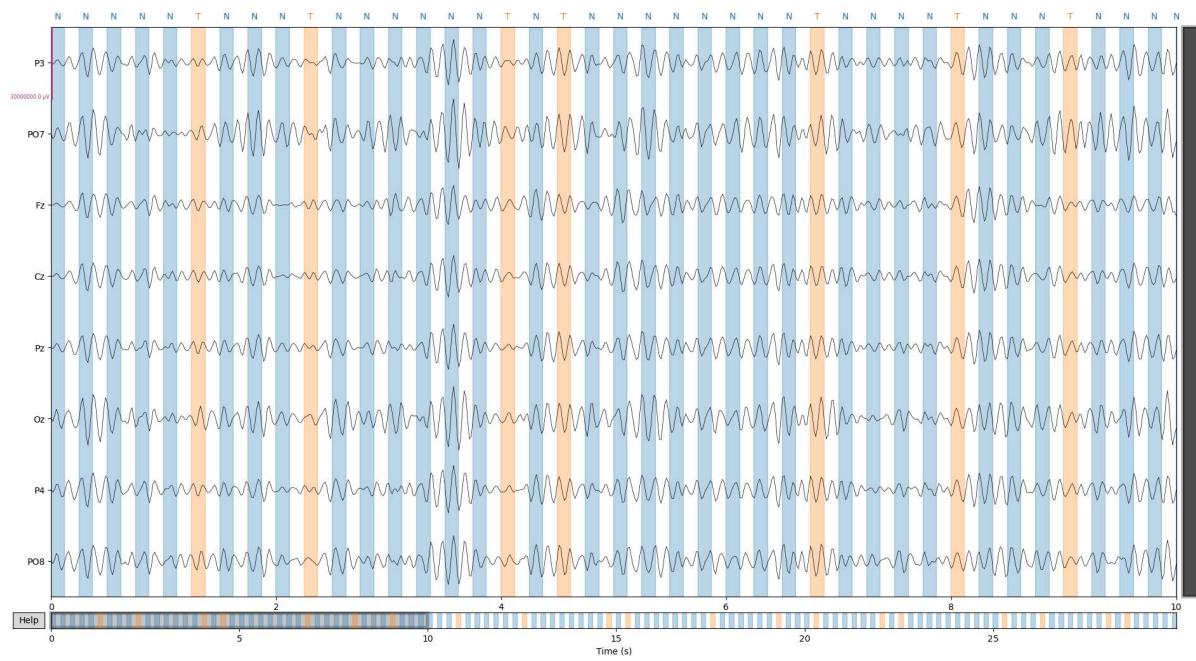
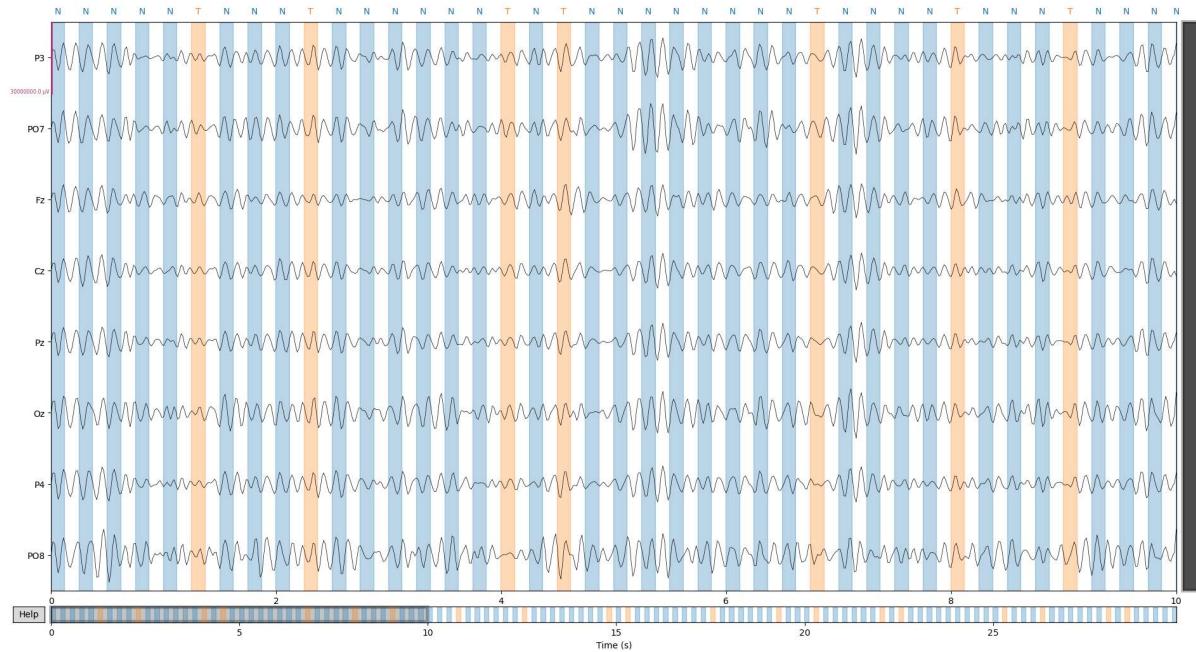
```
In [25]: baseline_duration = 200 #ms
baseline_samples = int(sample_frequency*baseline_duration/1000)
epoch_duration = 1000 #ms
epoch_samples = int(sample_frequency*epoch_duration/1000)

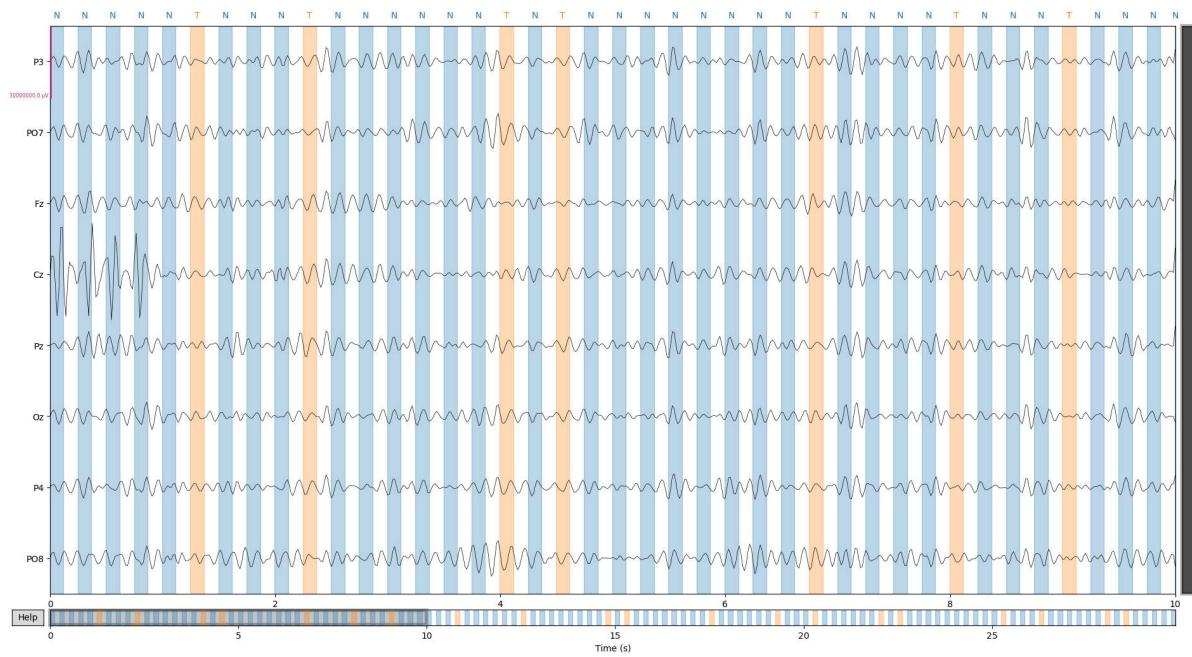
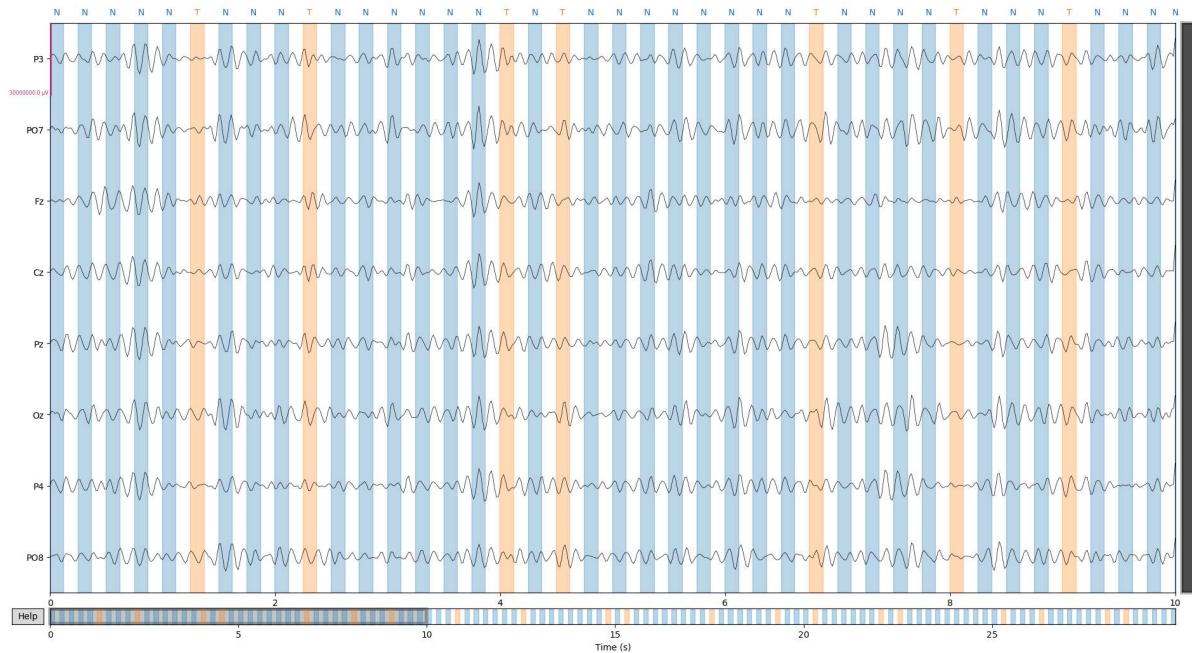
X_filtered = X_filtered.reshape((X_filtered.shape[0], -1, X_filtered.shape[-1]))
y_filtered = y_target.reshape((y_target.shape[0], -1))

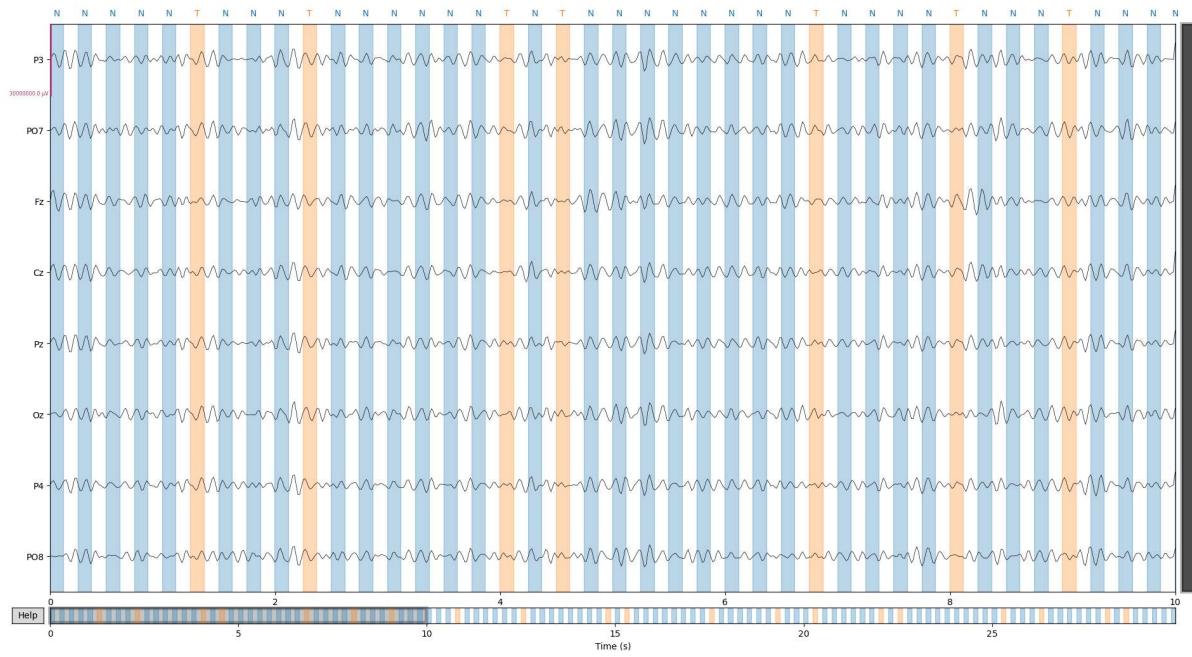
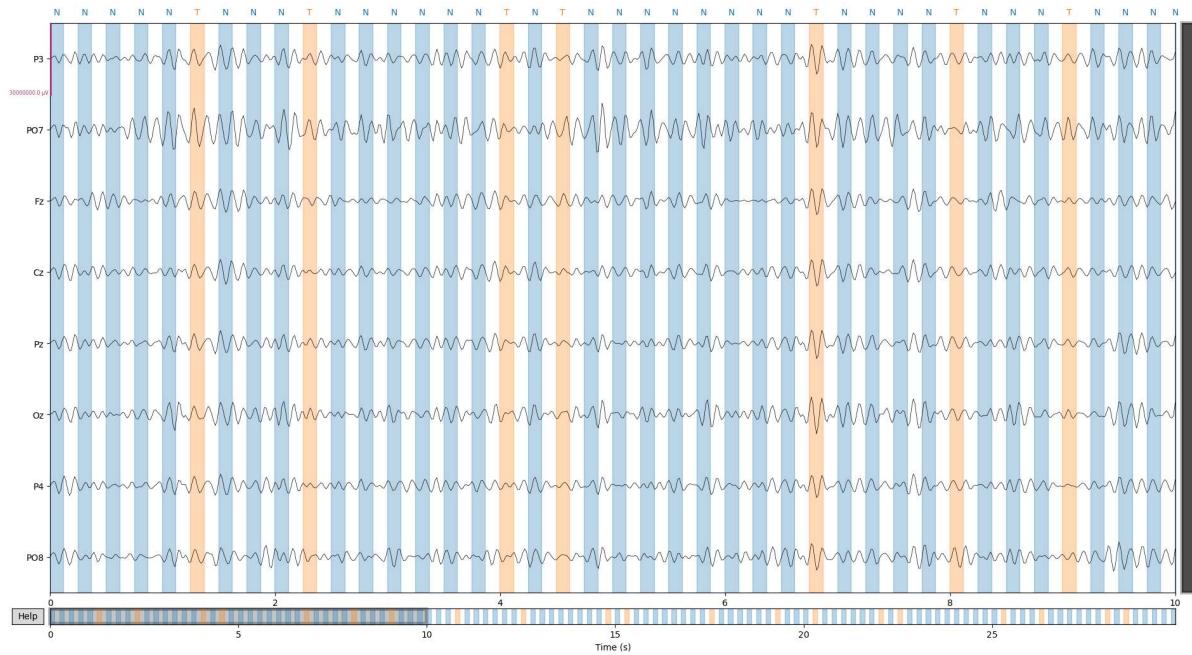
for patient in range(X_filtered.shape[0]):
    for channel in range(X_filtered.shape[-1]):
        i = 0
        while i < X_filtered.shape[1]:
            if y_filtered[patient, i] != 0: # Stimulus onset
                average = 0
                if i > baseline_samples:
                    average = np.mean(X_filtered[patient, i-baseline_samples:i+epoch_samples, channel])
                X_filtered[patient, i:i+epoch_samples, channel] = X_filtered[patient, i:i+epoch_samples, channel] - average
                i = i + epoch_samples
            else:
                i += 1
del y_filtered
X_filtered = X_filtered.reshape(X.shape)
print(X_filtered.shape)
```

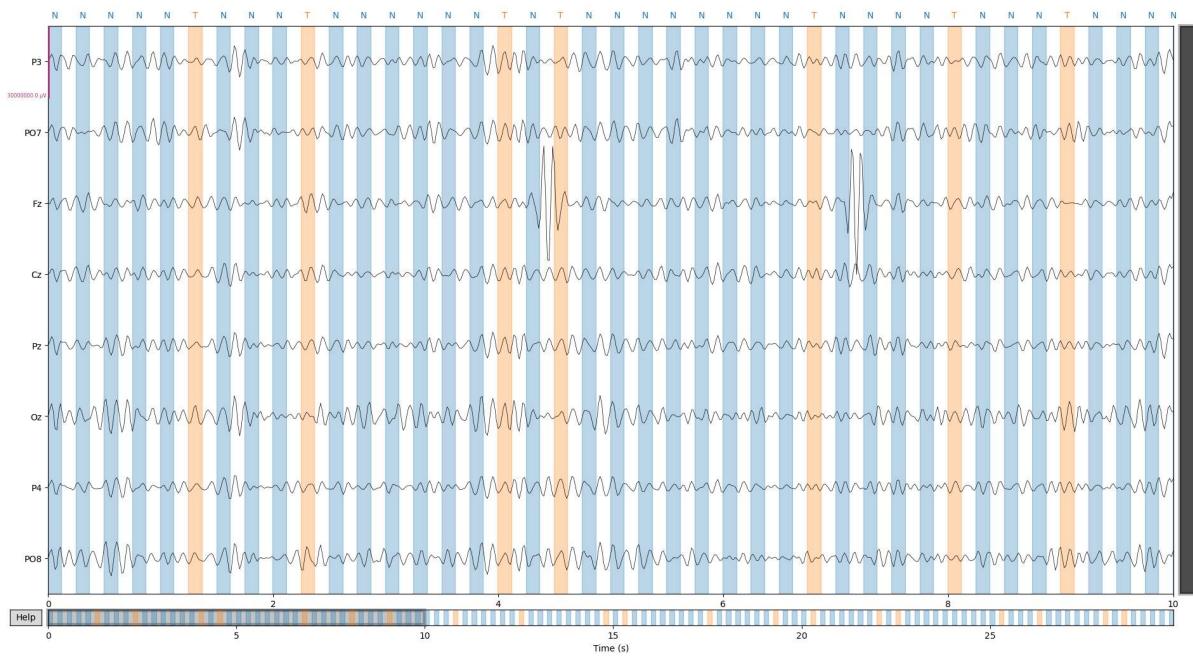
(8, 7, 5, 7648, 8)

```
In [26]: %matplotlib inline
for i in range(X.shape[0]):
    print(f"Patient {i+1}")
    view_signal(X_filtered[i, 0, 0, :, :], title=f'Patient {i+1}', lowpass=20,
    highpass=8, y=y_target[i],
    scalings=dict(eeg=15))
```

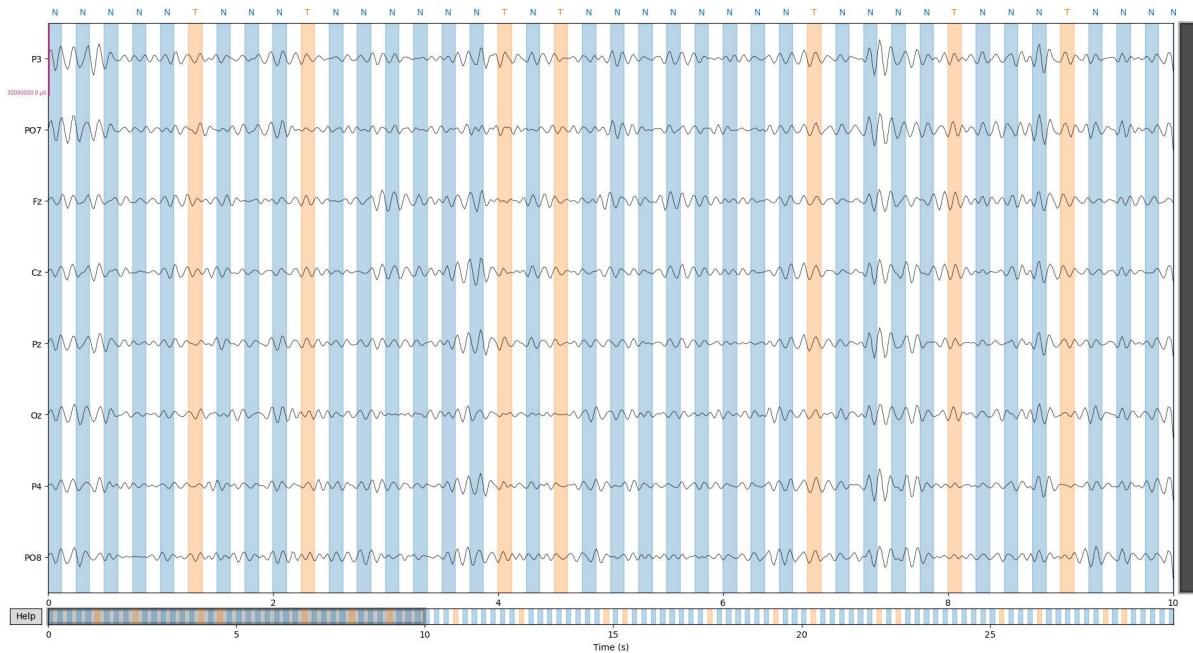
**Patient 1****Patient 2****Patient 3**

**Patient 4****Patient 5**

**Patient 6****Patient 7**



Patient 8



```
In [27]: # %matplotlib qt
# view_signal(X_filtered[1, 0:2, 0:5, :, :], title=f'Patient {i+1}', y=y_target[i])
```

## Calculating average waveform of each onset

In the paper, the waveform that occurred in each trial between 200-700 ms after each onset was collected from 'Cz' channel. The average waveform was calculated for each onset type (Target and Non-target). The highest value of the difference between the average target and non-target waveform was considered as the P300 amplitude.

```
In [343]: waveform_start_time_after_onset = 0
waveform_end_time_after_onset = 1000
waveform_start_sample_after_onset = int(sample_frequency * waveform_start_time_
_after_onset / 1000)
waveform_end_sample_after_onset = 17*12
#waveform_end_sample_after_onset = int(sample_frequency * waveform_end_time_af-
ter_onset / 1000)

for i in range(X_filtered.shape[0]):
    target_waveforms = []
    non_target_waveforms = []
    x = X_filtered[i, :, :, :, get_channel_index(patient_data[0], 'Cz')].resha-
pe(-1)
    y = y_target[i].reshape(-1)

    for j in range(1, y.size):
        if (y[j] != 0) and (y[j-1] != y[j]): # start of onset
            if j + waveform_end_sample_after_onset < y.size:
                if y[j] == 1:
                    non_target_waveforms.append(x[j+waveform_start_sample_afte-
r_onset:j+waveform_end_sample_after_onset])
                else:
                    target_waveforms.append(x[j+waveform_start_sample_after_-
onset:j+waveform_end_sample_after_onset])

    target_waveforms = np.array(target_waveforms)
    non_target_waveforms = np.array(non_target_waveforms)

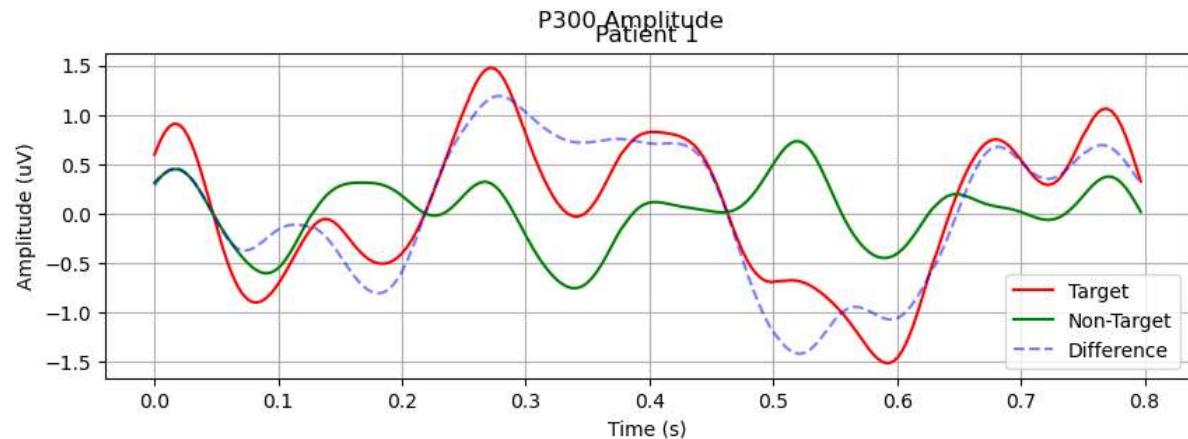
    x_axis = np.linspace(0, target_waveforms.shape[1]/sample_frequency, target_-
waveforms.shape[1])
    print(target_waveforms.shape, non_target_waveforms.shape)
    target_waveforms = np.mean(target_waveforms, axis=0)
    non_target_waveforms = np.mean(non_target_waveforms, axis=0)

plt.figure(figsize=(10,3))
if i == 0:
    plt.suptitle("P300 Amplitude")
    plt.title(f"Patient {i+1}")

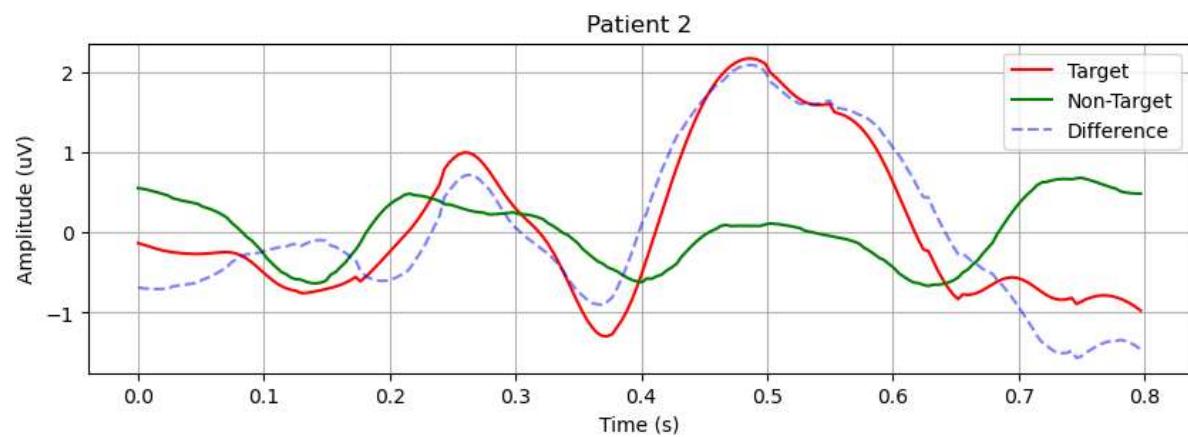
    plt.plot(x_axis, target_waveforms, 'r-', label='Target')
    plt.plot(x_axis, non_target_waveforms, 'g-', label='Non-Target')
    plt.plot(x_axis, target_waveforms - non_target_waveforms, 'b--', alpha=0.
5, label='Difference')

    plt.xlabel("Time (s)")
    plt.ylabel("Amplitude (uV)")
    plt.legend()
    plt.grid()
    plt.subplots_adjust(hspace=0.5)
    plt.show()
```

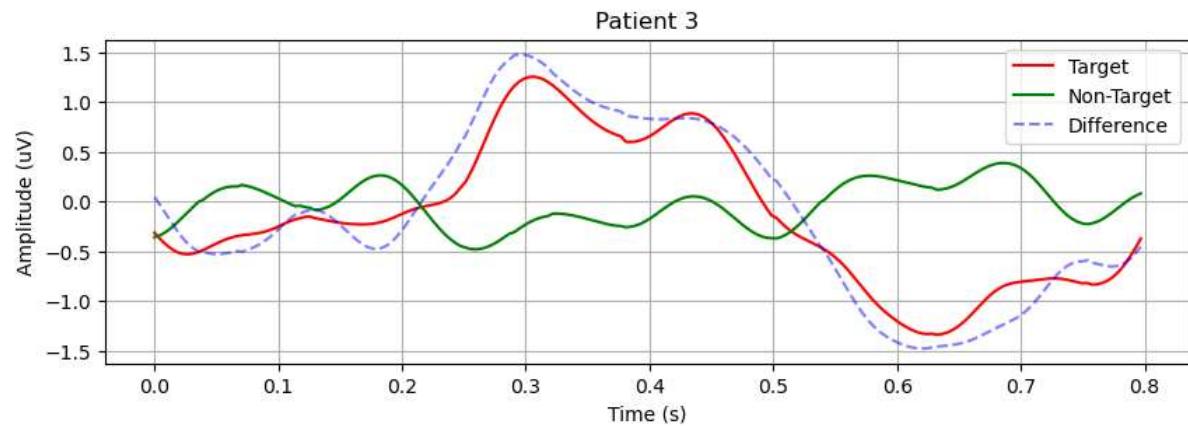
(700, 204) (3496, 204)



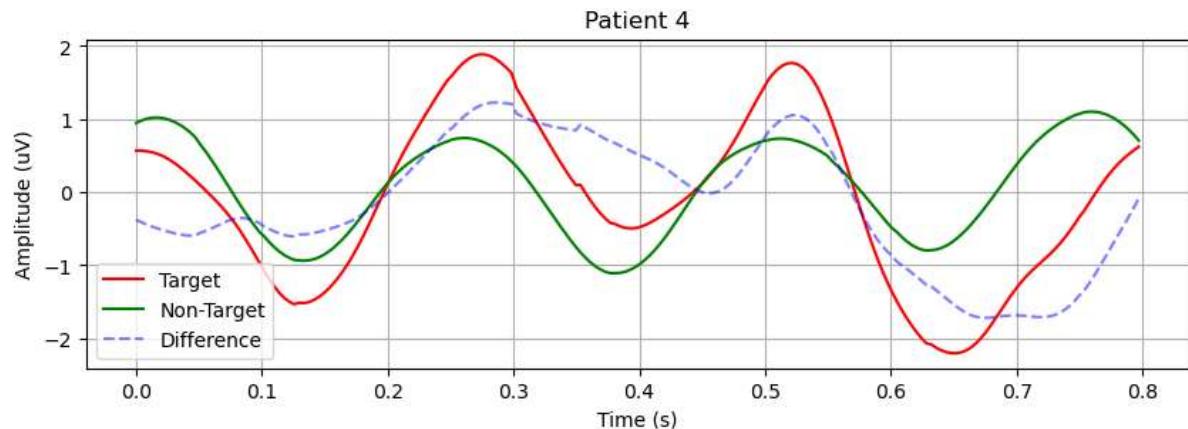
(700, 204) (3496, 204)



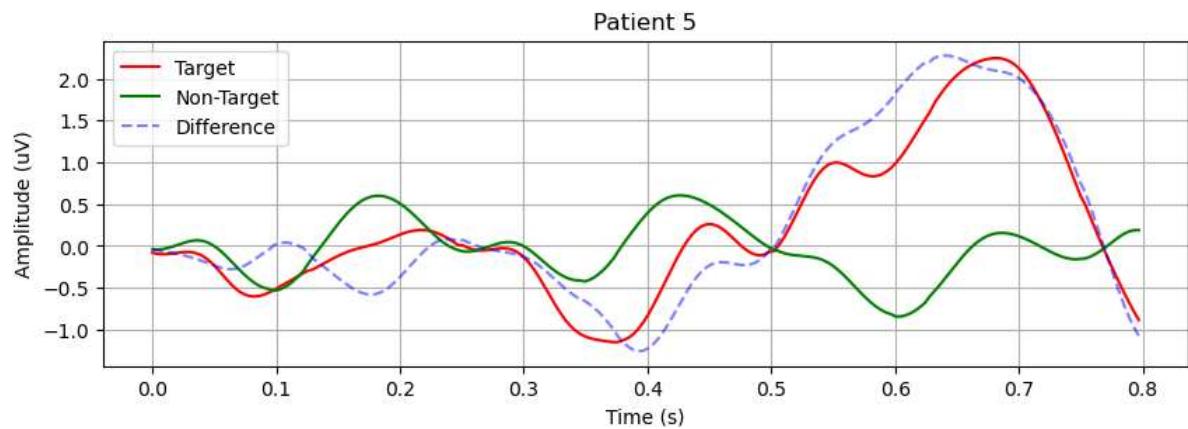
(700, 204) (3496, 204)



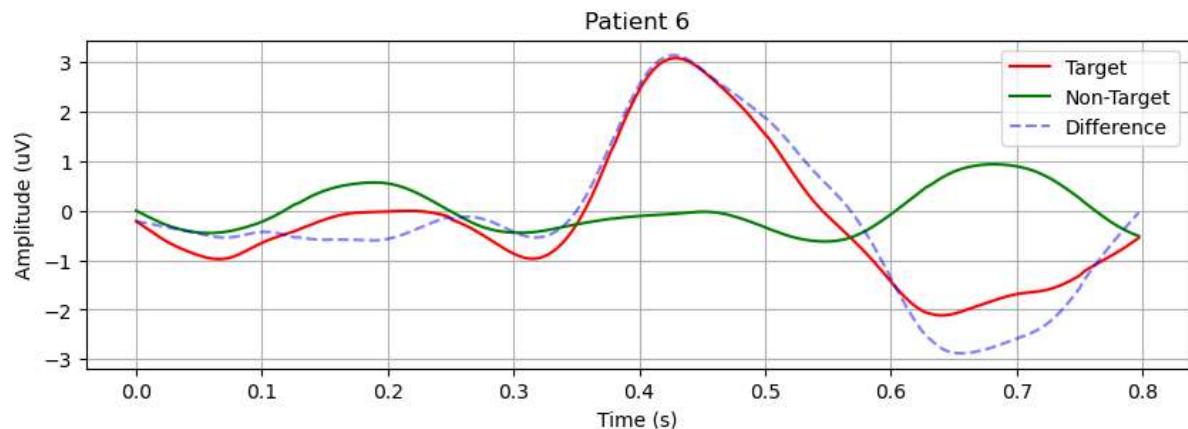
(700, 204) (3496, 204)



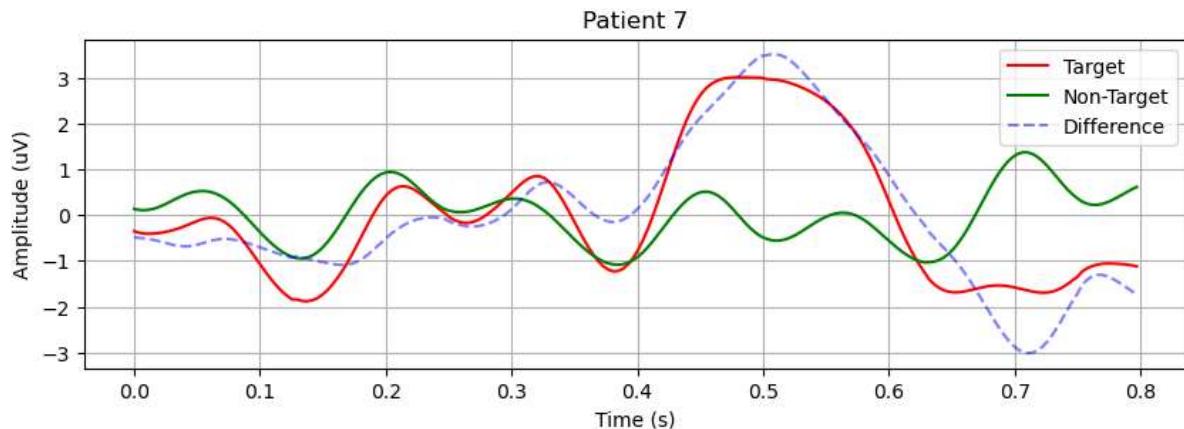
(700, 204) (3496, 204)



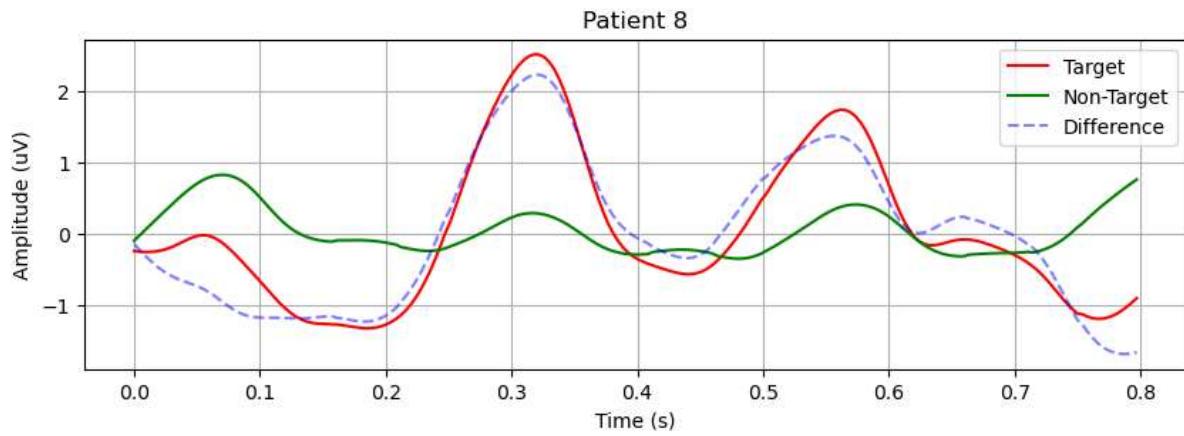
(700, 204) (3496, 204)



(700, 204) (3496, 204)



(700, 204) (3496, 204)



## Classification - Part 1

### Extract from the paper (Single trial classification)

To provide an estimate of the classifier accuracy we considered the binary classification problem target vs. non-target (Blankertz et al., 2011) that takes into account the correct classification of a target or of a non-target. Frequency filtering, data segmentation and artifact rejection were conducted as in P300 morphology section. EEG data were then resampled by replacing each sequence of 12 samples with their mean value, yielding  $17 \times 8$  samples per epoch (eight being the number of channels), which were concatenated in a feature vector (Krusienski et al., 2006). A seven-fold cross-validation was used to evaluate the binary accuracy (BA) of the classifier on each participant's dataset. For each iteration we applied a SWLDA on the testing dataset (consisting of six words) to extract the 60 most significant control features (Draper and Smith, 1998) and we assessed the BA on the training dataset (the remaining word).

```
In [347]: def create_dataset(x, y, show_progress=False, get_delayed_samples=0, get_fft=False):
    """
        Input Shape: x: (Patients, Trial, Run, Samples, channels), y: (Patients, Trial, Run, Samples)
        Output Shape: (Data, Samples, channels), (Data, Samples)
    """
    waveform_start_time_after_onset = 0
    waveform_start_sample_after_onset = int(sample_frequency * waveform_start_time_after_onset / 1000)

    stimulus_samples = 256*stimulus_duration//1000
    waveform_end_sample_after_onset = stimulus_samples + get_delayed_samples

    x = x.reshape((x.shape[0], x.shape[1] * x.shape[2] * x.shape[3], x.shape[4]))
    y = y.reshape((y.shape[0], y.shape[1] * y.shape[2] * y.shape[3]))

    x_result = []
    y_result = []

    for patient in range(x.shape[0]):
        val = range(x.shape[1])
        if show_progress:
            val = tqdm(val)
        for sample in val:
            if (y[patient, sample] != 0) and (y[patient, sample-1] != y[patient, sample]): # start of onset
                if sample + waveform_end_sample_after_onset < y.shape[-1]:
                    x_result.append(x[i, sample+waveform_start_sample_after_onset:sample+waveform_end_sample_after_onset, :])
                    y_result.append(y[patient, sample])

    return np.array(x_result), np.array(y_result)
```

```
In [360]: # As per paper number of samples are 17*12 which and then a running average of  
# 12 samples is calculated to obtain 17 samples  
samples = 17*12  
additional_samples = samples - (256*stimulus_duration//1000)  
x, y = create_dataset(X_filtered, y_target, True, additional_samples)  
x.shape
```

```
100% |  
267680/267680 [00:00<00:00, 467139.42it/s]  
100% |  
267680/267680 [00:00<00:00, 468765.28it/s]  
100% |  
267680/267680 [00:00<00:00, 466326.06it/s]  
100% |  
267680/267680 [00:00<00:00, 468775.66it/s]  
100% |  
267680/267680 [00:00<00:00, 468776.64it/s]  
100% |  
267680/267680 [00:00<00:00, 466274.55it/s]  
100% |  
267680/267680 [00:00<00:00, 467139.23it/s]  
100% |  
267680/267680 [00:00<00:00, 463097.89it/s]
```

```
Out[360]: (33576, 204, 8)
```

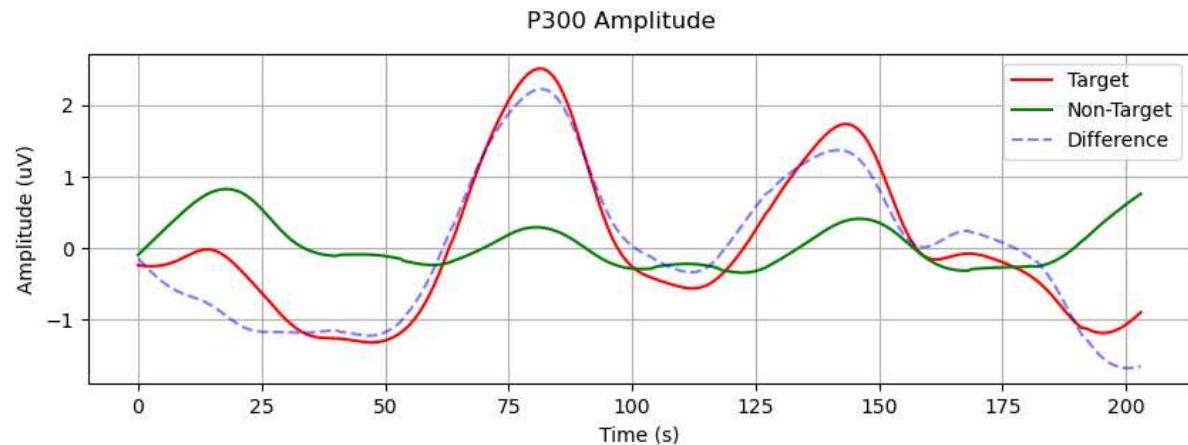
```
In [361]: target_waveforms = np.mean(x[np.where(y == 2)[0], :, get_channel_index(patient_data[0], 'Cz')], axis=0)

non_target_waveforms = np.mean(x[np.where(y == 1)[0], :, get_channel_index(patient_data[0], 'Cz')], axis=0)

plt.figure(figsize=(10,3))

plt.suptitle("P300 Amplitude")
plt.plot(target_waveforms, 'r-', label='Target')
plt.plot(non_target_waveforms, 'g-', label='Non-Target')
plt.plot(target_waveforms - non_target_waveforms, 'b--', alpha=0.5, label='Difference')

plt.xlabel("Time (s)")
plt.ylabel("Amplitude (uV)")
plt.legend()
plt.grid()
plt.subplots_adjust(hspace=0.5)
plt.show()
```

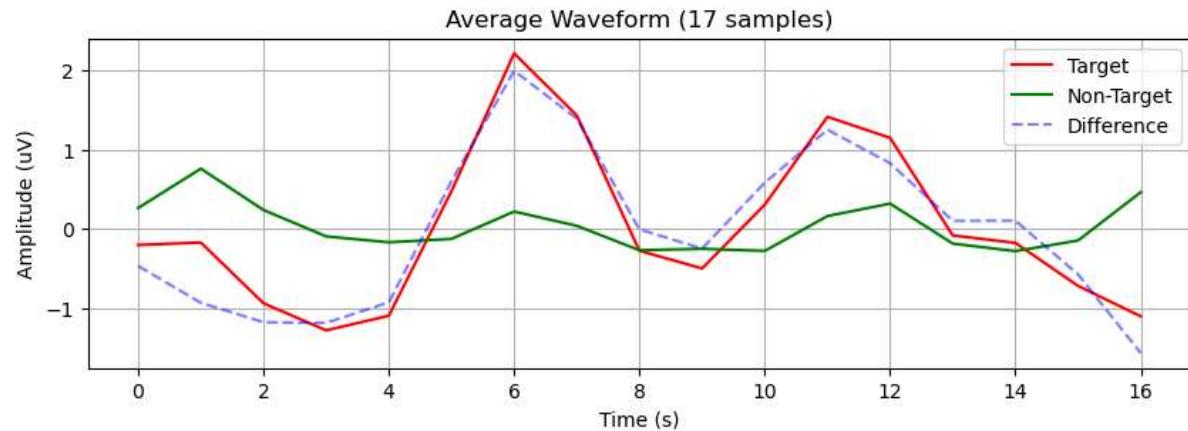


```
In [362]: target_waveforms = np.reshape(target_waveforms, (17, 12)).mean(axis=1)
print(target_waveforms.shape)
non_target_waveforms = np.reshape(non_target_waveforms, (17, 12)).mean(axis=1)

plt.figure(figsize=(10, 3))
plt.title("Average Waveform (17 samples)")
plt.plot(target_waveforms, 'r-', label='Target')
plt.plot(non_target_waveforms, 'g-', label='Non-Target')
plt.plot(target_waveforms - non_target_waveforms, 'b--', alpha=0.5, label='Difference')

plt.xlabel("Time (s)")
plt.ylabel("Amplitude (uV)")
plt.legend()
plt.grid()
plt.subplots_adjust(hspace=0.5)
plt.show()
```

(17,)



## Create Dataset:

1. Labels: Non-Target(1), Target (2)
2. Input Size:  $17 \times 8 = 136$

```
In [373]: total_samples_per_class = 17*12 # Number of samples in each waveform
additional_samples = total_samples_per_class - int(256*stimulus_duration/1000)
# Additional sample
x, y = create_dataset(X_filtered, y_target, get_delayed_samples=additional_samples, show_progress=True)
y = y - 1
print(x.shape)

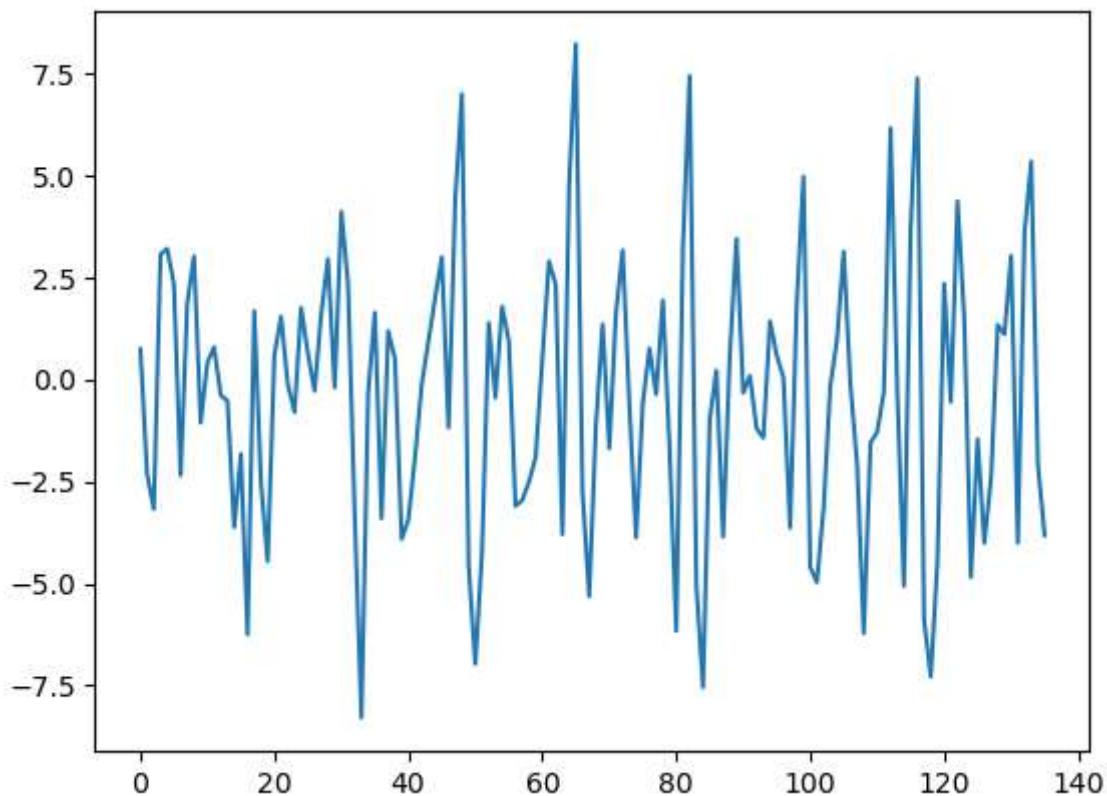
# Bring channel to second axis for ease of visualization
x = x.transpose((0, 2, 1))

# Calculate average of consecutive 12 samples to get a sample of size 17x8
x = x.reshape((x.shape[0], x.shape[1], 17, 12)).mean(axis=-1)
x, y = shuffle(x, y, random_state=12432)
X_data = x.copy()
y_data = y.copy()
x = x.reshape((x.shape[0], -1))
print(x.shape)
```

```
100%|██████████| 267680/267680 [00:00<00:00, 459921.84it/s]
100%|██████████| 267680/267680 [00:00<00:00, 465508.20it/s]
100%|██████████| 267680/267680 [00:00<00:00, 462303.30it/s]
100%|██████████| 267680/267680 [00:00<00:00, 467143.11it/s]
100%|██████████| 267680/267680 [00:00<00:00, 465518.81it/s]
100%|██████████| 267680/267680 [00:00<00:00, 463904.81it/s]
100%|██████████| 267680/267680 [00:00<00:00, 463093.31it/s]
100%|██████████| 267680/267680 [00:00<00:00, 461506.68it/s]

(33576, 204, 8)
(33576, 136)
```

```
In [374]: plt.figure()
plt.plot(x[2, :])
plt.show()
```



```
In [375]: np.unique(y)
```

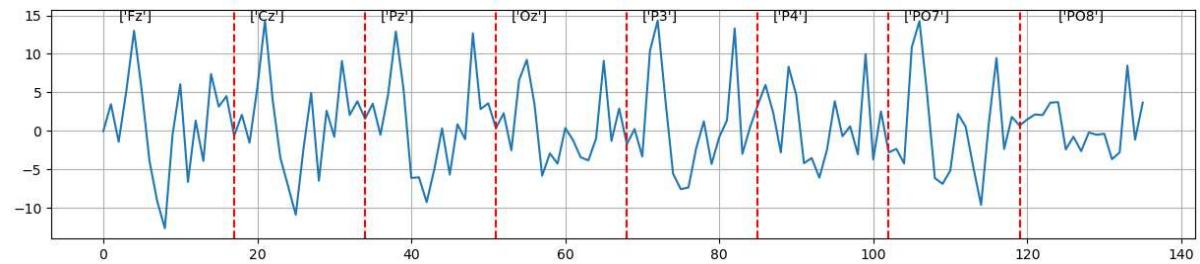
```
Out[375]: array([0, 1], dtype=uint8)
```

```
In [376]: # Balancing the dataset
data_per_class = 5600
indices = np.concatenate((np.where(y == 0)[0][:data_per_class], np.where(y == 1)[0][:data_per_class]))
x = x[indices, :]
y = y[indices]
```

```
In [377]: X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=6150)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

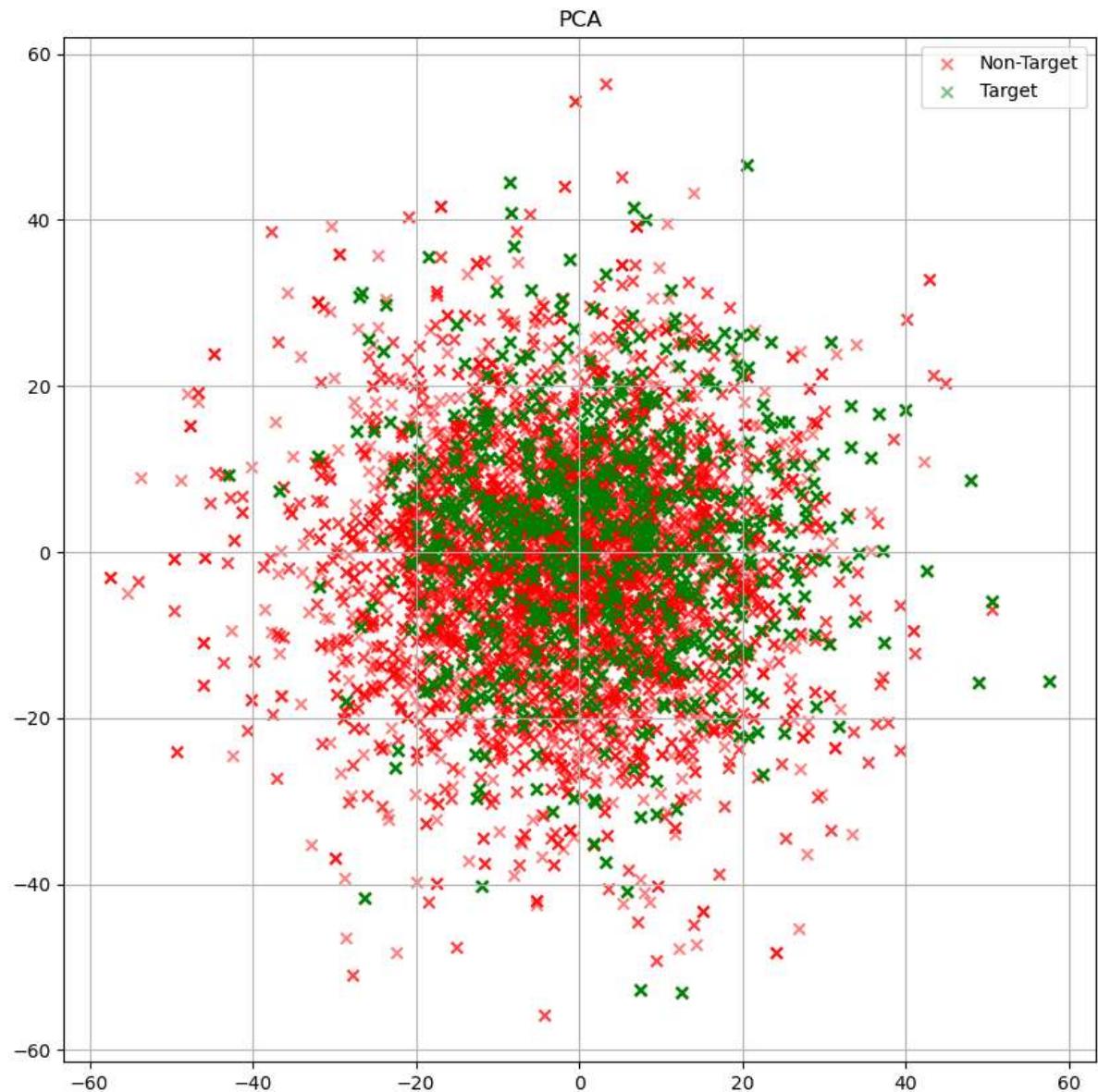
```
(8960, 136) (8960,) (2240, 136) (2240,)
```

```
In [396]: plt.figure(figsize=(15, 3))
plt.plot(x[0, :])
a = np.max(x[0])
for i in range(1, 8):
    p = x.shape[-1]//8*i
    plt.axvline(x=p, color='red', linestyle='--')
    plt.text(p-15, a, get_channel_name(patient_data[0], i-1))
    if i == 7:
        plt.text(p+5, a, get_channel_name(patient_data[0], i))
plt.grid()
plt.show()
```



## PCA for visualization

```
In [397]: a = PCA(n_components=2).fit(x).transform(x)
plt.figure(figsize=(10, 10))
plt.title("PCA")
plt.scatter(a[y == 0, 0], a[y == 0, 1], label='Non-Target', color='r', alpha=0.5, marker='x')
plt.scatter(a[y == 1, 0], a[y == 1, 1], label='Target', color='g', alpha=0.5, marker='x')
plt.grid()
plt.legend()
plt.show()
```



# Stepwise LDA

Source: [\(https://sebastianraschka.com/Articles/2014\\_python\\_lda.html\)](https://sebastianraschka.com/Articles/2014_python_lda.html)

LDA is a supervised machine learning algorithm that reduces the number of dimensions in feature space of a dataset. However, the number of reduced dimension is equal to the minimum value between the number of features in the feature space minus one and total number of classes minus one. Since, this classification algorithm consists of just 2 classes(Target and Non-Target) the reduced number of feature is equal to 1.

Therefore, in order to extract multiple important features using LDA we need to compute Stepwise Linear Discriminant Analysis(SWLDA) which is a form of LDA but calculates multiple important features in order to reduce the dimensionality of the feature space.

## Overview

### Summarizing the LDA approach in 5 steps:

Listed below are the 5 general steps for performing a linear discriminant analysis; we will explore them in more detail in the following sections.

1. Compute the d-dimensional mean vectors for the different classes from the dataset.
1. Compute the scatter matrices (in-between-class and within-class scatter matrix).
1. Compute the eigenvectors ( $e_1, e_2, \dots, e_d$ ) and corresponding eigenvalues ( $\lambda_1, \lambda_2, \dots, \lambda_d$ ) for the scatter matrices.
1. Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a  $d \times k$  dimensional matrix  $W$  (where every column represents an eigenvector).
1. Use this  $d \times k$  eigenvector matrix to transform the samples onto the new subspace. This can be summarized by the matrix multiplication:  $Y = X \times W$  (where  $X$  is a  $n \times d$ -dimensional matrix representing the  $n$  samples, and  $y$  are the transformed  $n \times k$ -dimensional samples in the new subspace).

### Compute the d-dimensional mean vectors for the different classes from the dataset.

Calculate Mean of feature vector for all signals of each class.

**Shape of Mean Vector:**  $M \times N$  where  $M$  is no. of classes and  $N$  is No. of features

## 2. Compute the Scatter Matrices

Now, we will compute the two  $4 \times 4$ -dimensional matrices: The within-class and the between-class scatter matrix.

### a. Within Class Scatter Matrix, $S_W$ :

The within-class scatter matrix  $S_W$  is computed by the following equation:

$$S_W = \sum_{i=1}^c S_i$$

where 'c' is the no. of available classes and

$$S_i = \sum_{x \in D_i}^n (x - m_i)(x - m_i)^T$$

Here,

$x$  = A feature vector that belongs to the  $i^{th}$  class

$D_i$  = The set of feature vectors(signals) in the  $i^{th}$  class

$n$  = Total no. of feature vectors/signals in the  $i^{th}$  class

$m_i$  = Mean vector for the  $i^{th}$  class

$$m_i = \frac{1}{n_i} \sum_{x \in D_i}^n x_k$$

Here,

$x_k$  =  $k^{th}$  feature vector/signal belonging to the  $i^{th}$  class

### b. Between class Scatter matrix, $S_B$ :

The between-class scatter matrix  $S_B$  is computed by the following equation:

$$S_B = \sum_{i=1}^c N_i(m_i - m)(m_i - m)^T$$

Here,

$c$  = Number of Total Classes available

$N_i$  = Size(Total feature vectors/signals) in  $i^{th}$  class

$m_i$  = Mean vector of the  $i^{th}$  class

$m$  = Overall Mean(For all classes)

## 3. Solving the generalized eigenvalue problem for the matrix $S_W^{-1} S_B$

1. Next, we will solve the generalized eigenvalue problem for the matrix  $S_W^{-1} S_B$  to obtain the linear discriminants.
2. Then we will perform a quick check that the eigenvector-eigenvalue calculation is correct and satisfy the equation:

$$Av = \lambda v$$

Here,

$$A = S_W^{-1} S_B$$

$v$  = Eigenvector

$\lambda$  = eigenvalue

## 4. Selecting linear discriminants for the new feature subspace

### 4.1 Sorting the eigenvectors by decreasing eigenvalues

Remember from the introduction that we are not only interested in merely projecting the data into a subspace that improves the class separability, but also reduces the dimensionality of our feature space, (where the eigenvectors will form the axes of this new feature subspace).

However, the eigenvectors only define the directions of the new axis, since they have all the same unit length 1.

So, in order to decide which eigenvector(s) we want to drop for our lower-dimensional subspace, we have to take a look at the corresponding eigenvalues of the eigenvectors. Roughly speaking, the eigenvectors with the lowest eigenvalues bear the least information about the distribution of the data, and those are the ones we want to drop. The common approach is to rank the eigenvectors from highest to lowest corresponding eigenvalue and choose the top  $k$  eigenvectors.

### 4.2 Choosing $k$ eigenvectors with the largest eigenvalues

After sorting the eigenpairs by decreasing eigenvalues, it is now time to construct our  $k \times d$ -dimensional eigenvector matrix  $W$  (here  $N \times 2$ : based on the 2 most informative eigenpairs) and thereby reducing the initial  $N$ -dimensional feature space into a 2-dimensional feature subspace.

## 5. Transforming the samples onto the new subspace

In the last step, we use the  $4 \times 2$ -dimensional matrix  $WW$  that we just computed to transform our samples onto the new subspace via the equation:

$$Y = XxW$$

(where  $X$  is a  $n \times d$ -dimensional matrix representing the  $n$  samples, and  $Y$  are the transformed  $n \times k$ -dimensional samples in the new subspace).

```
In [295]: class StepwiseLDA:
    def __init__(self, n_components):
        self.n_components = n_components
        self.classes = []
        self.eigenpairs = []
        self.weights = None

    def transform(self, x):
        if self.weights is None:
            raise ValueError("Please fit the model before using it")

        return x.dot(self.weights)

    def fit(self, x, y):
        self.classes = np.unique(y)

        # Calculate mean vectors
        mean_vectors = []
        for c in self.classes:
            mean_vectors.append(np.mean(x[y == c], axis=0))
        mean_vectors = np.array(mean_vectors)

        s_w = np.zeros((x.shape[1],x.shape[1])) # Create scatter within matrix for each feature
        s_b = np.zeros((x.shape[1],x.shape[1])) # Between Class Scatter Matrix
        overall_mean = np.mean(x, axis=0).reshape(x.shape[1],1) # Mean for all classes

        for cl, mv in zip(self.classes, mean_vectors): # For each class and its corresponding mean vector
            mv = mv.reshape(x.shape[1],1) # transpose mean vector of the current class
            x_target = x[y==cl] # Calculate size/total no. of feature vectors in the current class
            n = x_target.shape[0]

            # Calculate within-class Scatter matrix
            class_sc_mat = np.zeros((x.shape[1],x.shape[1]))
            # scatter matrix for every class
            for row in x_target:
                row = row.reshape(x.shape[1],1) # make column vectors
                class_sc_mat += (row-mv).dot((row-mv).T)

            s_w += class_sc_mat # sum class scatter matrices

            # calculate between-class scatter matrix
            s_b += n * (mv - overall_mean).dot((mv - overall_mean).T) # Calculate Between class scatter matrix
```

```
# Compute Eigen Value and Eigenvector for the dot product between the
two matrices
eig_vals, eig_vecs = np.linalg.eig(np.linalg.inv(s_w).dot(s_b))

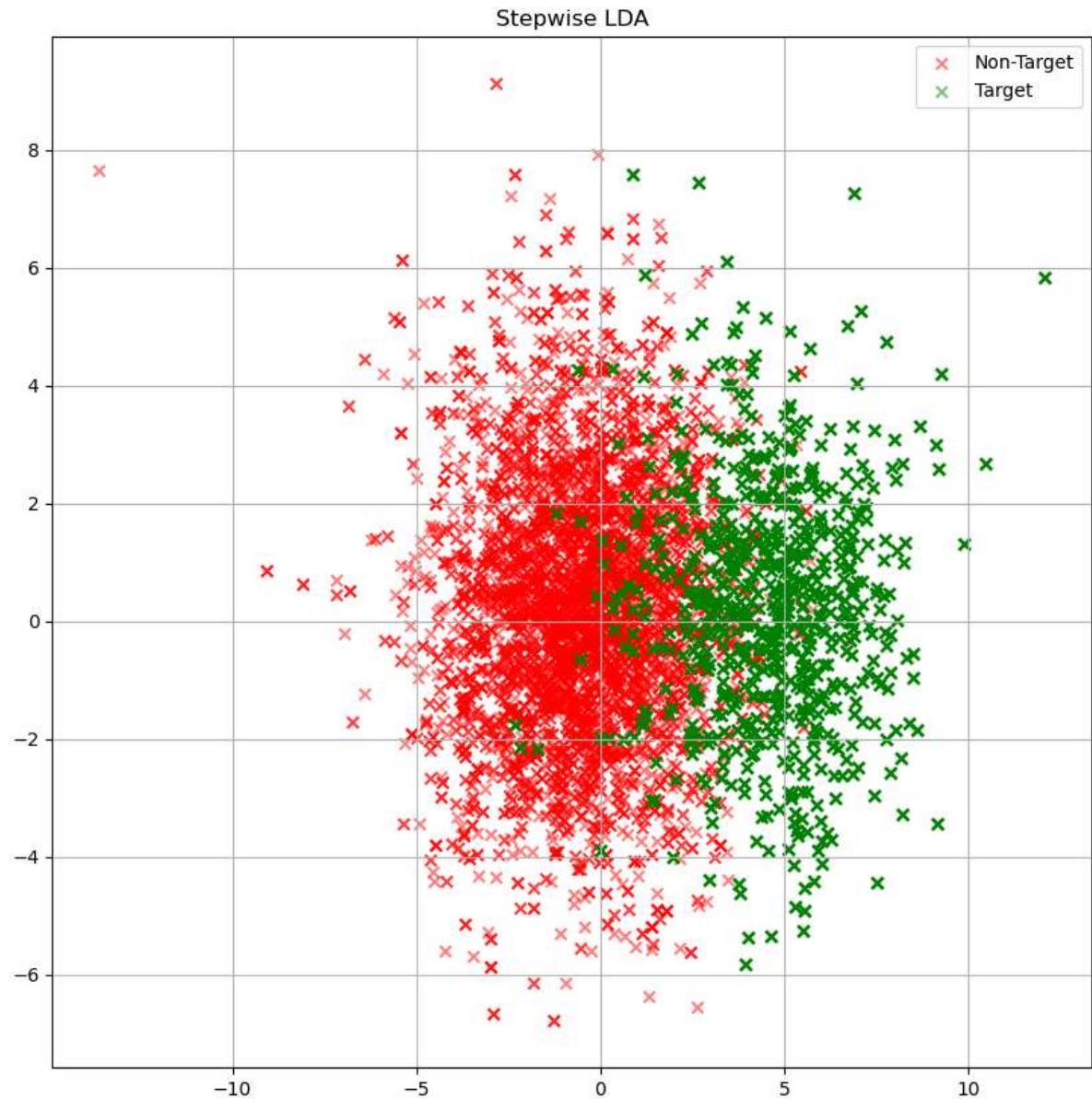
# Make a List of (eigenvalue, eigenvector) tuples
self.eigenpairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range
(len(eig_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
self.eigenpairs = sorted(self.eigenpairs, key=lambda k: k[0], reverse=
True)

# Calculate weights
self.weights = self.eigenpairs[0][1].reshape(x.shape[1],1)
#W = np.hstack((, eig_pairs[1][1].reshape(x_tr_an.shape[1],1)))
for i in range(1, self.n_components):
    self.weights = np.hstack((self.weights, self.eigenpairs[i][1].resh
ape(x.shape[1],1)))

return self
```

```
In [398]: a = StepwiseLDA(n_components=2).fit(x, y).transform(x)
plt.figure(figsize=(10, 10))
plt.title("Stepwise LDA")
plt.scatter(a[y == 0, 0].real, a[y == 0, 1].real, label='Non-Target', color='r', alpha=0.5, marker='x')
plt.scatter(a[y == 1, 0].real, a[y == 1, 1].real, label='Target', color='g', alpha=0.5, marker='x')
plt.grid()
plt.legend()
plt.show()
```



# Random Forest Classification

We will perform Random Forest Classification using the following datasets:

1. **Simple waveform:** Here we will use the individual waveforms as the input data.
2. **PCA Feature:** Here, we will use PCA on each individual waveform to extract important features and perform classification on those feature as inputs.
3. **Stepwise LDA Feature:** Here, we will use Step-Wise LDA on each individual waveform to extract important features and perform classification on those feature as inputs.

## 1. Simple Waveform

```
In [399]: clf = RandomForestClassifier(max_depth=2, random_state=26784)
clf.fit(X_train, y_train)
print(f"Accuracy: {clf.score(X_test, y_test)*100:.5f}%")
```

Accuracy: 75.66964%

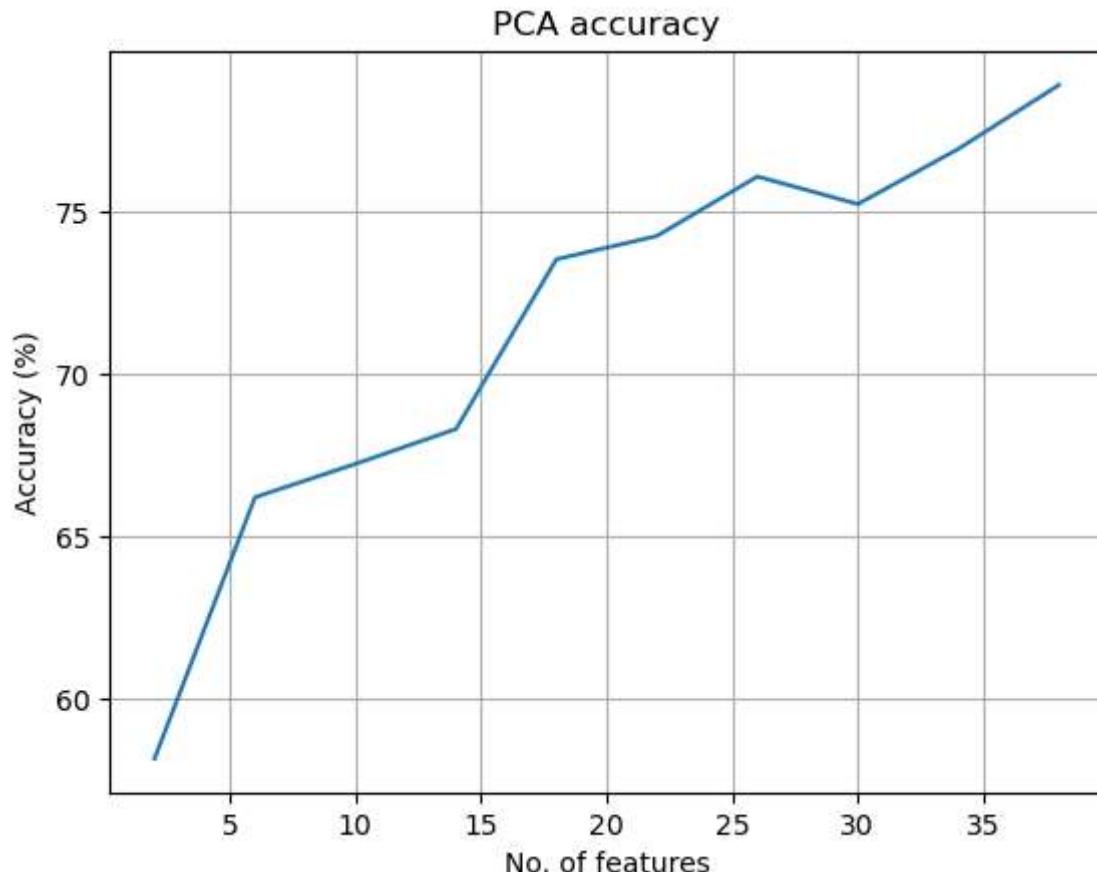
## 2. PCA

```
In [400]: input_features = np.arange(2, 42, 4)
accuracies = []

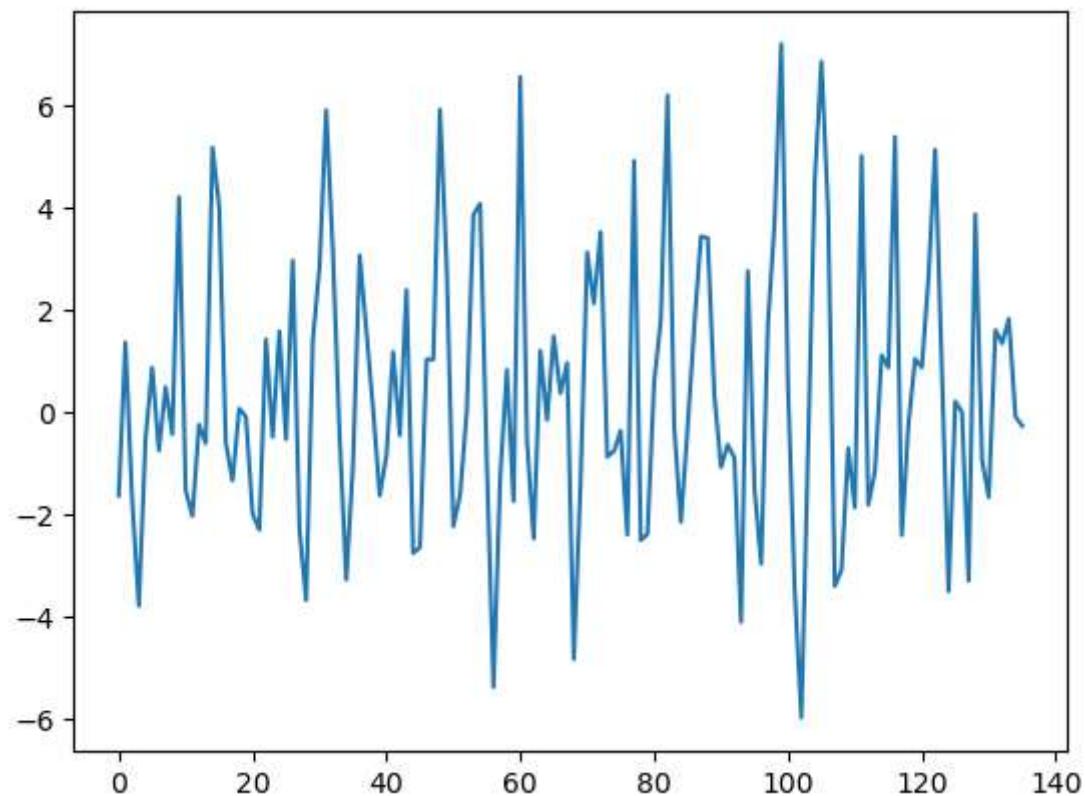
for f in tqdm(input_features):
    pca = PCA(n_components=f)
    pca.fit(X_train, y_train)
    a, b = pca.transform(X_train), pca.transform(X_test)
    clf = RandomForestClassifier(max_depth=2, random_state=26784)
    clf.fit(a, y_train)
    accuracies.append(clf.score(b, y_test)*100)

plt.figure()
plt.title("PCA accuracy")
plt.plot(input_features, accuracies)
plt.grid()
plt.xlabel("No. of features")
plt.ylabel("Accuracy (%)")
plt.show()
```

100% |   
 | 10/10 [00:09<00:00, 1.11it/s]



```
In [401]: plt.figure()  
plt.plot(X_train[0, :])  
plt.show()
```



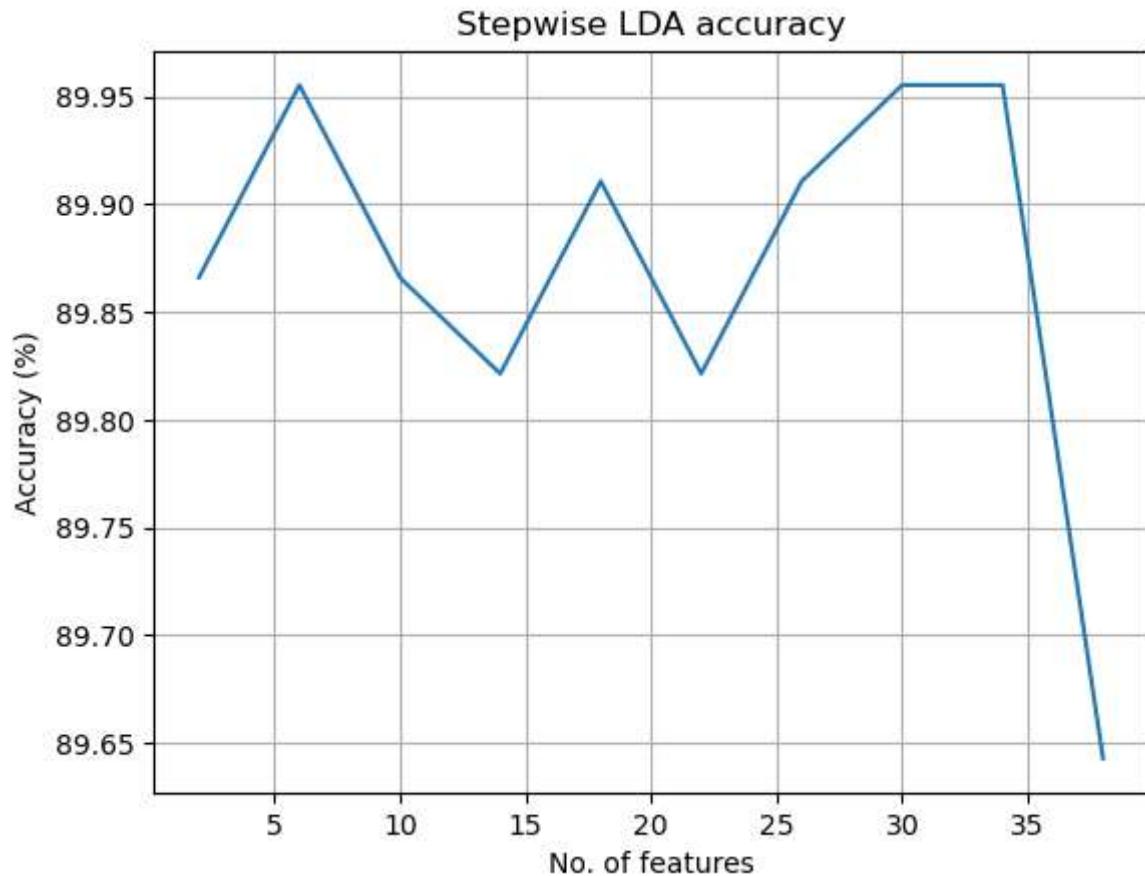
### 3. Stepwise LDA

```
In [402]: input_features = np.arange(2, 42, 4)
accuracies = []

for f in tqdm(input_features):
    lda = StepwiseLDA(f)
    lda.fit(X_train, y_train)
    a, b = lda.transform(X_train), lda.transform(X_test)
    clf = RandomForestClassifier(max_depth=2, random_state=26784)
    clf.fit(a.real, y_train)
    accuracies.append(clf.score(b.real, y_test)*100)

plt.figure()
plt.title("Stepwise LDA accuracy")
plt.plot(input_features, accuracies)
plt.grid()
plt.xlabel("No. of features")
plt.ylabel("Accuracy (%)")
plt.show()
```

100% |   
 | 10/10 [00:09<00:00, 1.04it/s]



In [ ]:

```
In [403]: a = StepwiseLDA(n_components=2).fit(x, y).transform(x)
clf = RandomForestClassifier(max_depth=2, random_state=26784)
clf.fit(a.real, y)

plt.figure()

ax = plt.subplot(1, 1, 1)
ax.set_title("Random Forest Classifier")

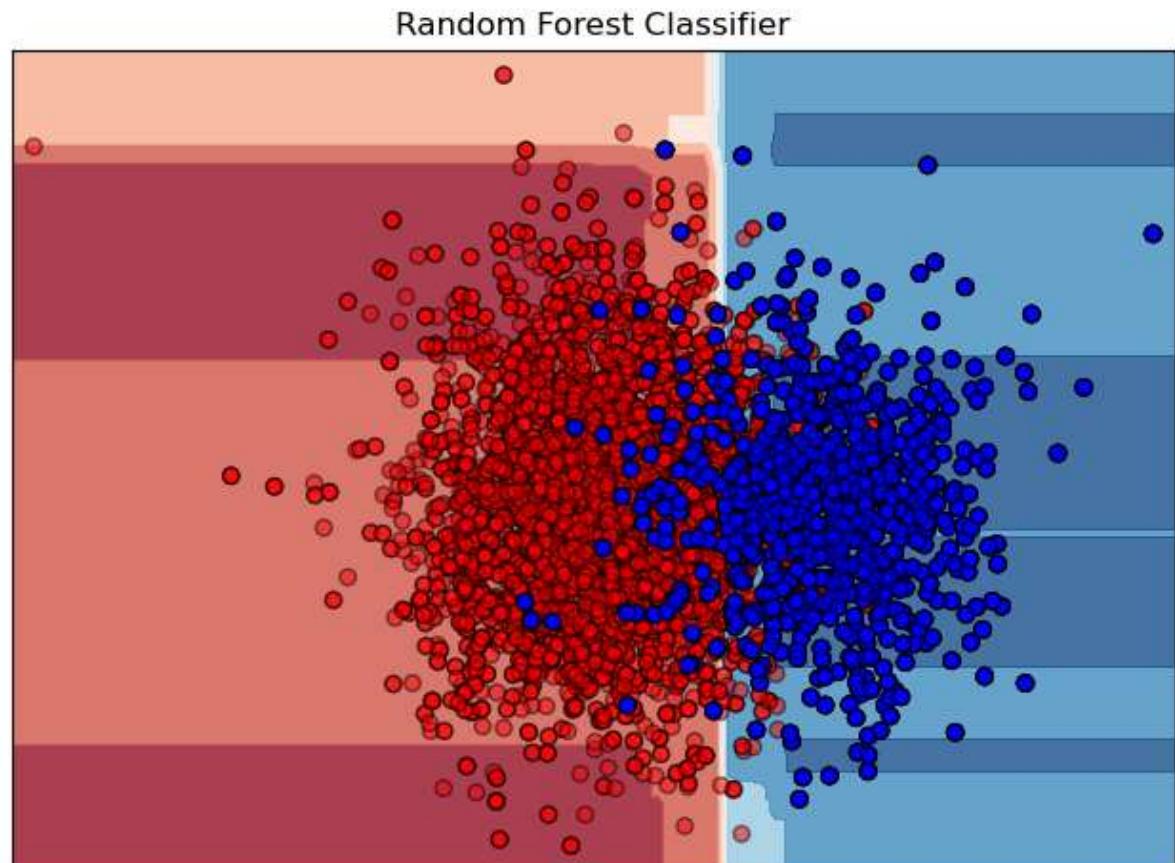
DecisionBoundaryDisplay.from_estimator(
    clf, a.real, cmap=plt.cm.RdBu, alpha=0.8, ax=ax, eps=0.5
)

ax.scatter(a[:, 0].real, a[:, 1].real, c=y, alpha=0.5, edgecolors="k", cmap=ListedColormap(["#FF0000", "#0000FF"]))

x_min, x_max = a[:, 0].real.min() - 0.5, a[:, 0].real.max() + 0.5
y_min, y_max = a[:, 1].real.min() - 0.5, a[:, 1].real.max() + 0.5

ax.set_xlim(x_min, x_max)
ax.set_xlim(x_min, x_max)
ax.set_ylim(y_min, y_max)
ax.set_xticks(())
ax.set_yticks(())

plt.tight_layout()
plt.show()
```



Stepwise LDA provides the best classification accuracy when the number of features is around 35-40. However, the classification accuracy is around 90%.

## Distribution of the data

```
In [405]: cols = 2
rows = X_data.shape[1]/cols
if int(rows) != rows:
    rows = int(rows) + 1
else:
    rows = int(rows)

current = 0

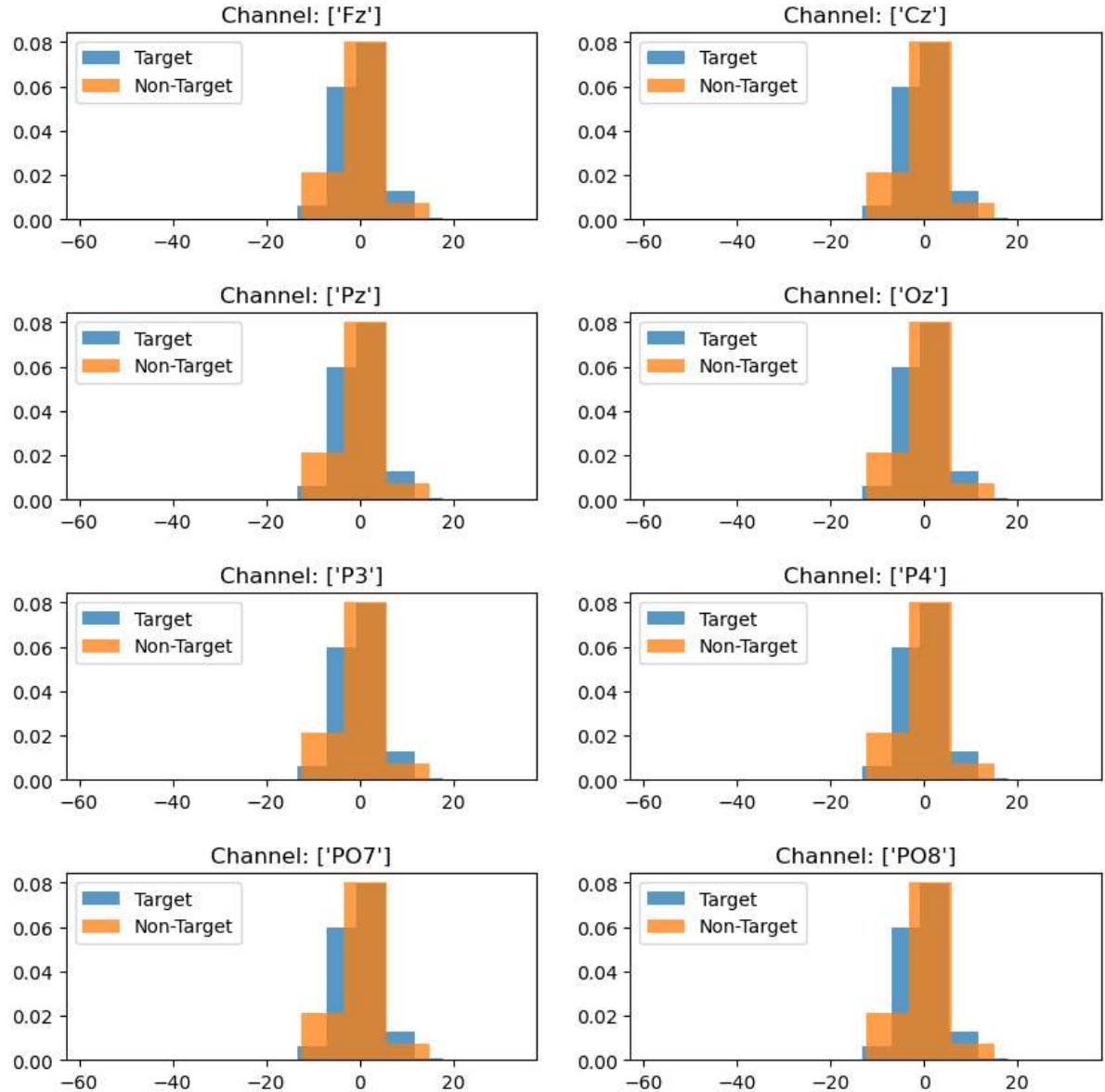
fig, ax = plt.subplots(rows, cols, figsize=(10, 10))
plt.suptitle("Data distribution")
for j in range(rows):
    for k in range(cols):
        if current >= X_data.shape[1]:
            break
        ax[j, k].set_title(f"Channel: {get_channel_name(patient_data[0], current)}")

        ax[j, k].hist(X_data[y_data == 1].reshape(-1)+1e-6, density=True, label='Target', alpha=0.75)
        ax[j, k].hist(X_data[y_data == 0].reshape(-1)+1e-6, density=True, label='Non-Target', alpha=0.75)
        ax[j, k].legend()

        current += 1

plt.subplots_adjust(hspace=0.5)
plt.show()
```

## Data distribution



## Statistical significance of P300

P300 should have a positive deflection in the waveform generated between 200-700 ms after the onset of a stimulus. As seen from the average waveforms calculated above (P300 Amplitude) we do see that the target signal of each channel has a positive peak compared to non-target signal. We would like to check the significance level of this statement.

To calculate whether target waveforms have higher amplitude, we will first calculate the average waveform from a number of sample signals. We will then calculate the average positive peak amplitude of the average waveform. This will provide us the highest peak amplitude of the waveform. We will then test on average, whether the peak amplitude of a target waveform is higher than a non-target waveform.

Therefore, we would like to test the following - Target waveforms have a lower or equal peak amplitude than non-target waveforms. i.e.

$$H_0 : \mu_{target} - \mu_{nontarget} \leq 0$$
$$H_1 : \mu_{target} - \mu_{nontarget} > 0$$

```
In [406]: x_target = X_data[np.where(y_data == 1)[0]]  
x_nontarget = X_data[np.where(y_data == 0)[0]]  
print(x_target.shape)
```

```
(5600, 8, 17)
```

```
In [408]: mean_calculation_data_size = 100
sample_size = 50

np.random.seed(34324)

def view_distribution():
    for channel in range(x_target.shape[1]):
        ma_t = []
        ma_nt = []
        for _ in range(sample_size):
            # Get average waveform from sample
            t = np.mean(x_target[ np.random.choice(np.arange(x_target.shape[0]), size=mean_calculation_data_size, replace=False), channel, :], axis=0)
            nt = np.mean(x_nontarget[ np.random.choice(np.arange(x_nontarget.shape[0]), size=mean_calculation_data_size, replace=False), channel, :], axis=0)

            # Remove negative portion of average waveform as P300 deflection is positive
            t[t<0] = 0
            nt[nt<0] = 0

            # Calculate the average amplitude of each signal
            ma_nt.append(np.mean(nt))
            ma_t.append(np.mean(t))

        mean_target_amplitude = np.mean(ma_t)
        std_target_amplitude = np.std(ma_t)

        mean_nontarget_amplitude = np.mean(ma_nt)
        std_nontarget_amplitude = np.std(ma_nt)

        print(f"Target Mean: {mean_target_amplitude}, Std: {std_target_amplitude}")
        print(f"Non-Target Mean: {mean_nontarget_amplitude}, Std: {std_nontarget_amplitude}")

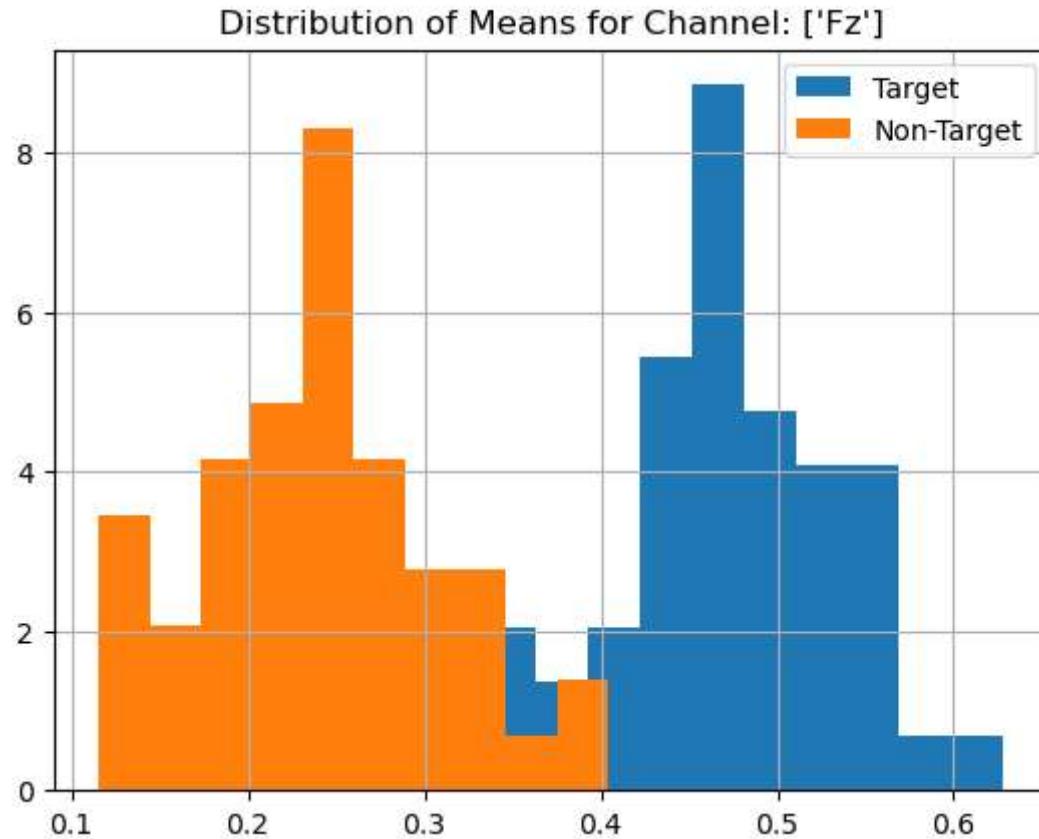
        # T-Statistics
        confidence = 0.95
        d = stats.ttest_ind(ma_t, ma_nt, equal_var=False, alternative='greater')
        print(f"Confidence interval: {d.confidence_interval(confidence)}, T-Score: {d.statistic:.6f}, Degree of freedom: {d.df}, P-Value: {d.pvalue}")
        if d.pvalue > (1-confidence):
            print(f"With a significance level of {1-confidence:.5f}, the hypothesis is accepted")
        else:
            print(f"With a significance level of {1-confidence:.5f}, the hypothesis is rejected")

plt.figure()
```

```
plt.title(f"Distribution of Means for Channel: {get_channel_name(patient_data[0], channel)}")
plt.hist(ma_t, density=True, label='Target')
plt.hist(ma_nt, density=True, label='Non-Target')
plt.grid()
plt.legend()
plt.show()

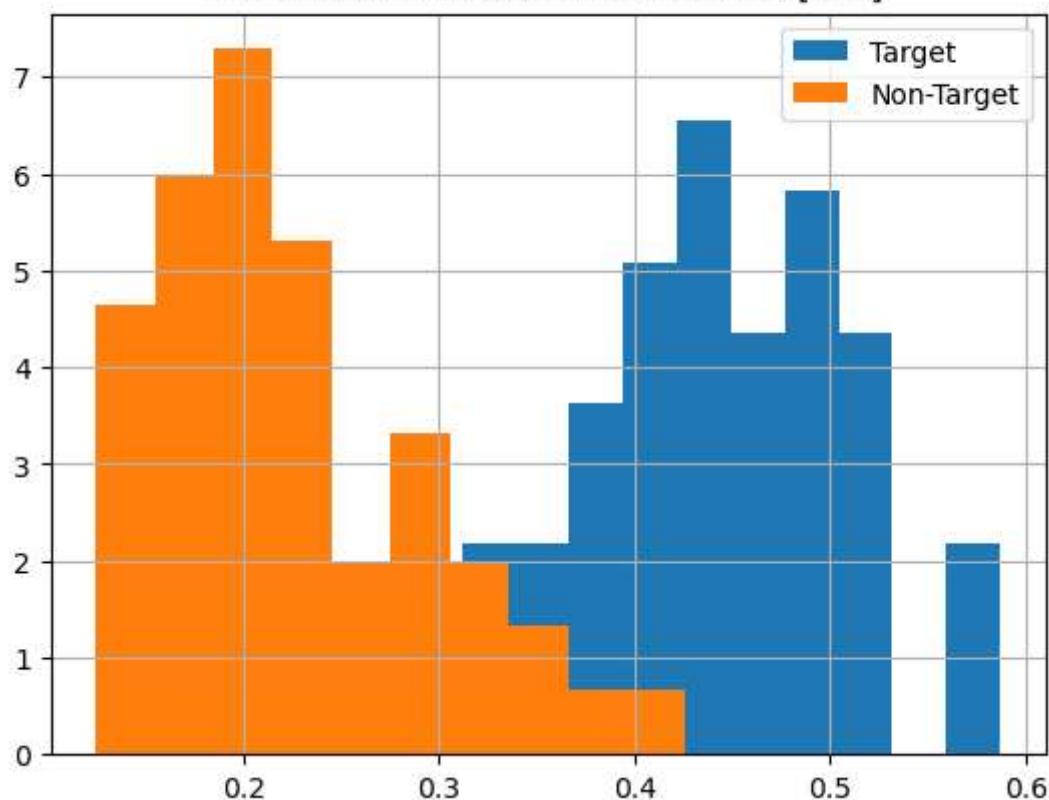
view_distribution()
```

Target Mean: 0.4746667094587482, Std: 0.06067926240884431  
Non-Target Mean: 0.24015819804121472, Std: 0.06664372629531629  
Confidence interval: ConfidenceInterval(low=0.21312602098023875, high=inf), T-Score: 18.213308, Degree of freedom: 97.15095202987126, P-Value: 2.162793025238947e-33  
With a significance level of 0.05000, the hypothesis is rejected



Target Mean: 0.4430455069649211, Std: 0.06409750127575528  
Non-Target Mean: 0.2240366996275501, Std: 0.06755833438070388  
Confidence interval: ConfidenceInterval(low=0.1969165145731703, high=inf), T-Score: 16.462072, Degree of freedom: 97.73024464785895, P-Value: 3.0944242509278306e-30  
With a significance level of 0.05000, the hypothesis is rejected

## Distribution of Means for Channel: ['Cz']



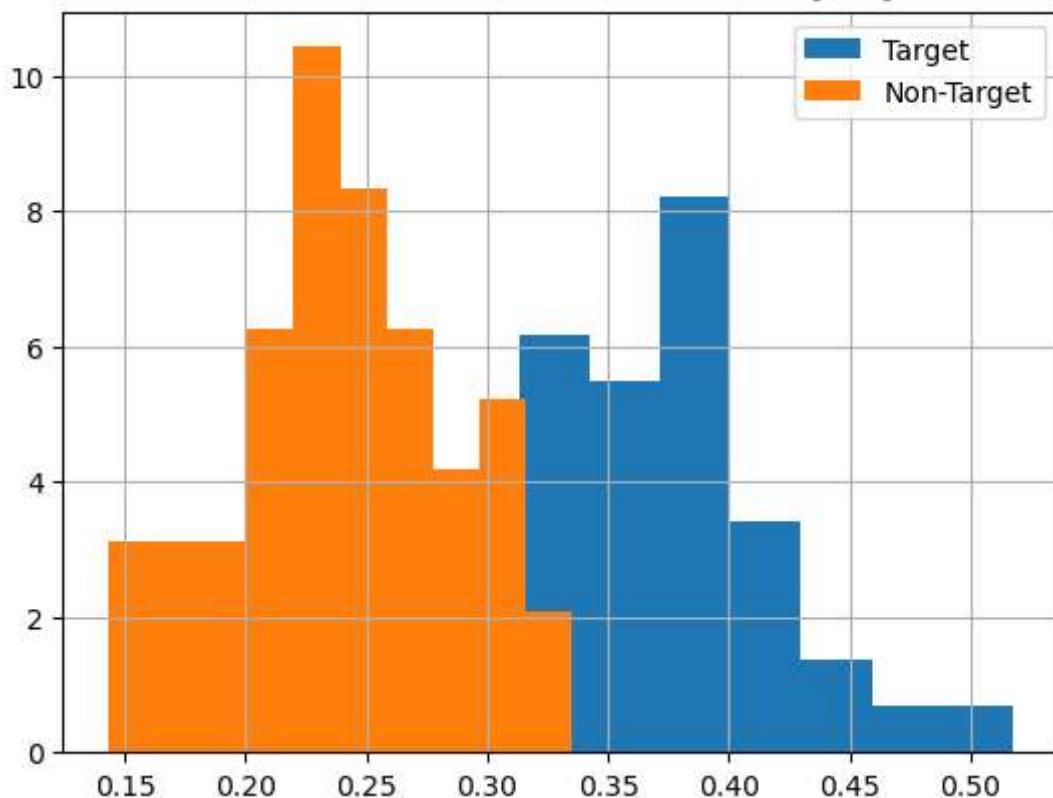
Target Mean: 0.35340510703701644, Std: 0.05866412000737728

Non-Target Mean: 0.2407385726597356, Std: 0.04620404466447245

Confidence interval: ConfidenceInterval(low=0.09494282355909965, high=inf), T-Score: 10.561368, Degree of freedom: 92.89907503939554, P-Value: 6.790690380754123e-18

With a significance level of 0.05000, the hypothesis is rejected

## Distribution of Means for Channel: ['Pz']



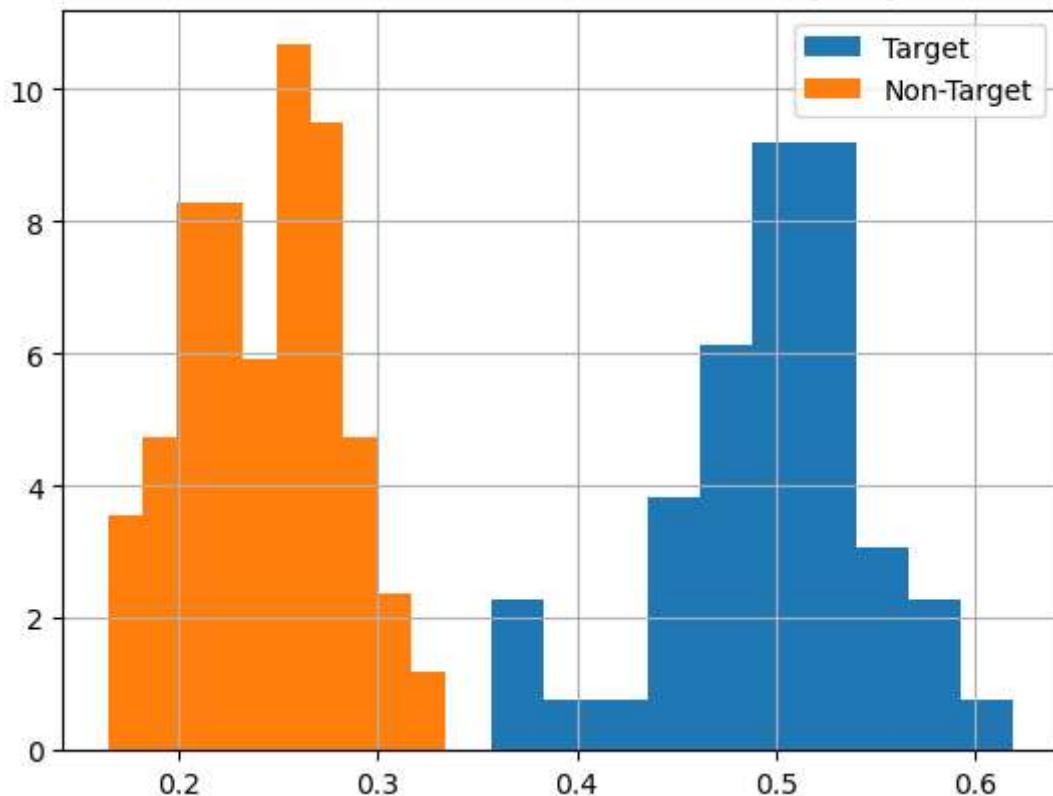
Target Mean: 0.49529928174690974, Std: 0.05260673046086157

Non-Target Mean: 0.24200223010514071, Std: 0.03872916141635347

Confidence interval: ConfidenceInterval(low=0.23778738231398838, high=inf), T-Score: 27.142240, Degree of freedom: 90.05510589000879, P-Value: 1.955583020  
1937045e-45

With a significance level of 0.05000, the hypothesis is rejected

## Distribution of Means for Channel: ['Oz']



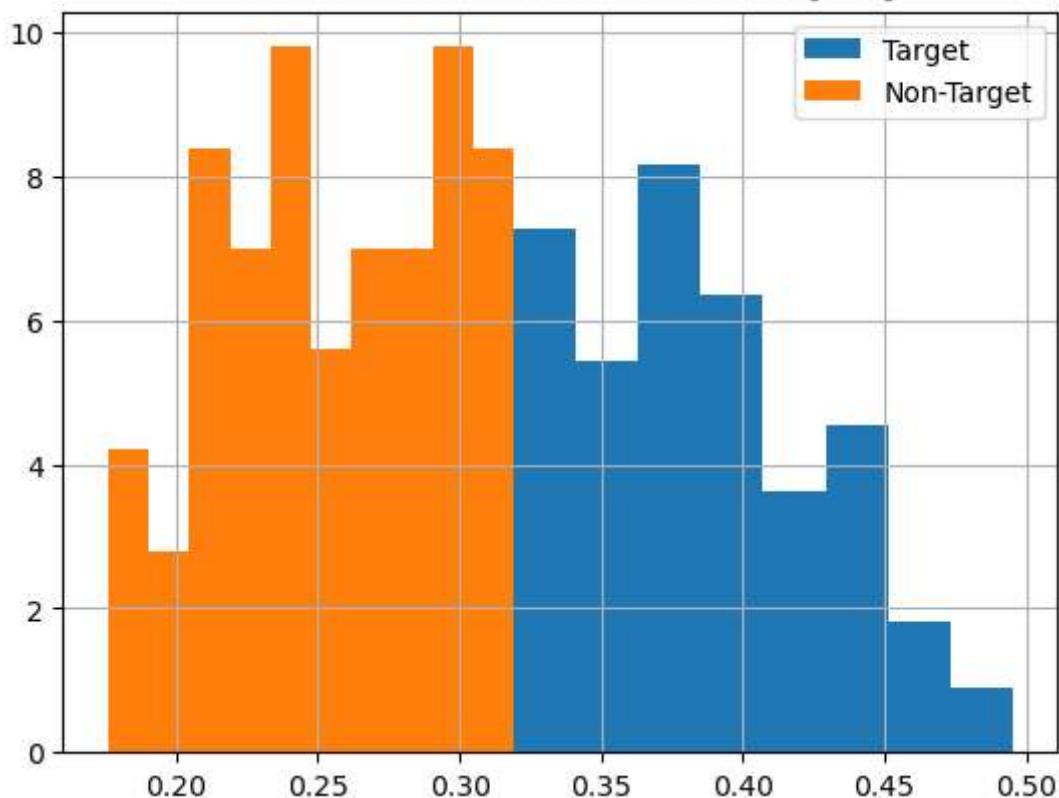
Target Mean: 0.3725125640378665, Std: 0.05026233400631502

Non-Target Mean: 0.25610144642838334, Std: 0.03899375926360597

Confidence interval: ConfidenceInterval(low=0.10131146219656093, high=inf), T-Score: 12.809607, Degree of freedom: 92.29859352311449, P-Value: 1.718673102  
0436986e-22

With a significance level of 0.05000, the hypothesis is rejected

## Distribution of Means for Channel: ['P3']



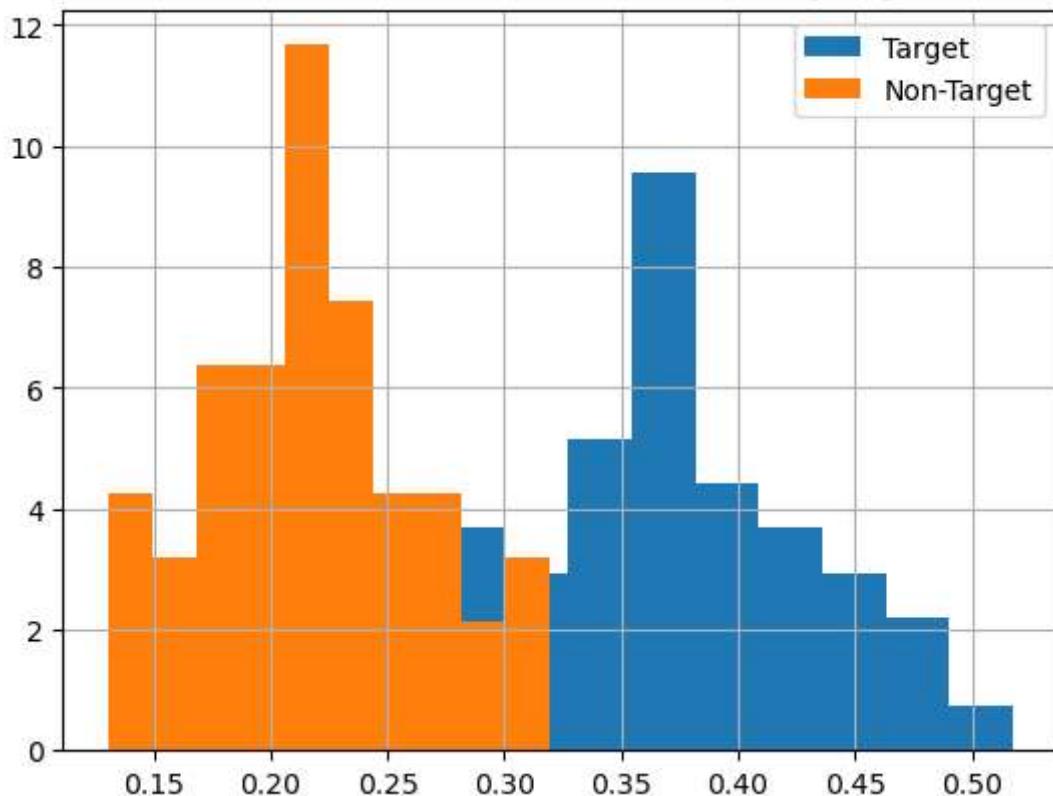
Target Mean: 0.3734285999855696, Std: 0.05765315502427675

Non-Target Mean: 0.21836594083244593, Std: 0.044657076692342576

Confidence interval: ConfidenceInterval(low=0.13775280894872133, high=inf), T-Score: 14.884195, Degree of freedom: 92.23450576280811, P-Value: 1.476920273  
1543146e-26

With a significance level of 0.05000, the hypothesis is rejected

## Distribution of Means for Channel: ['P4']



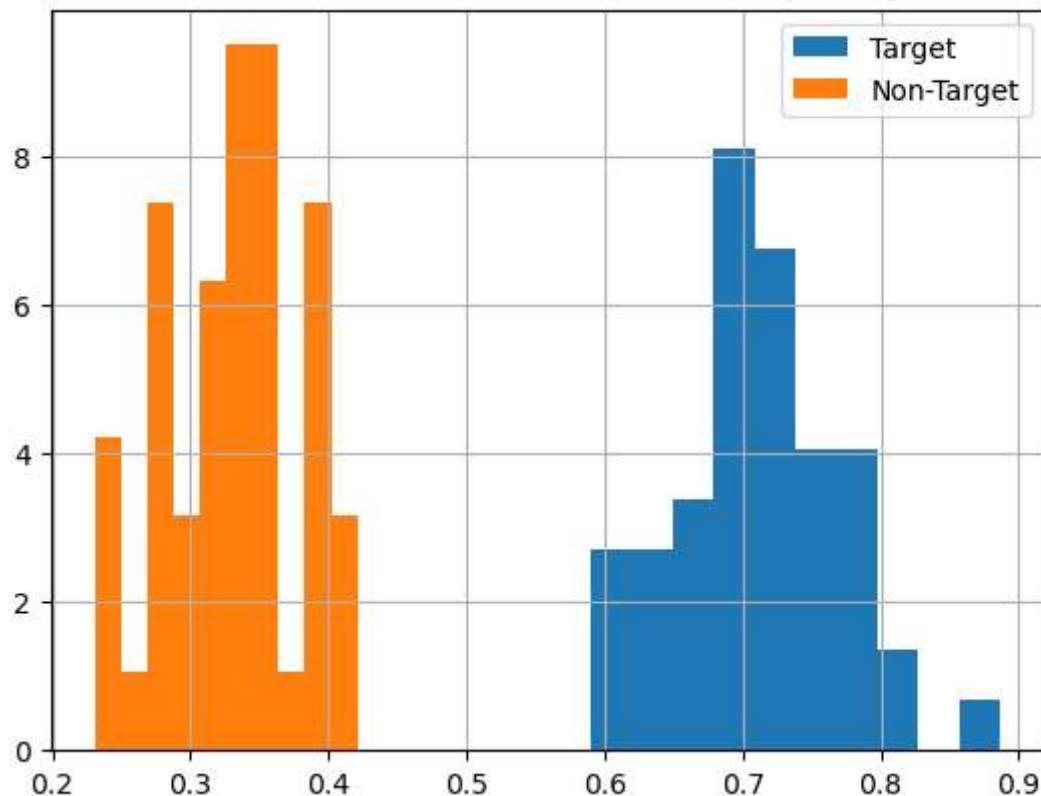
Target Mean: 0.7103176227230938, Std: 0.0615426358446131

Non-Target Mean: 0.33223501553066626, Std: 0.04716664288166952

Confidence interval: ConfidenceInterval(low=0.35967697828337236, high=inf), T-Score: 34.132528, Degree of freedom: 91.79739465211715, P-Value: 2.972129746508494e-54

With a significance level of 0.05000, the hypothesis is rejected

## Distribution of Means for Channel: ['PO7']



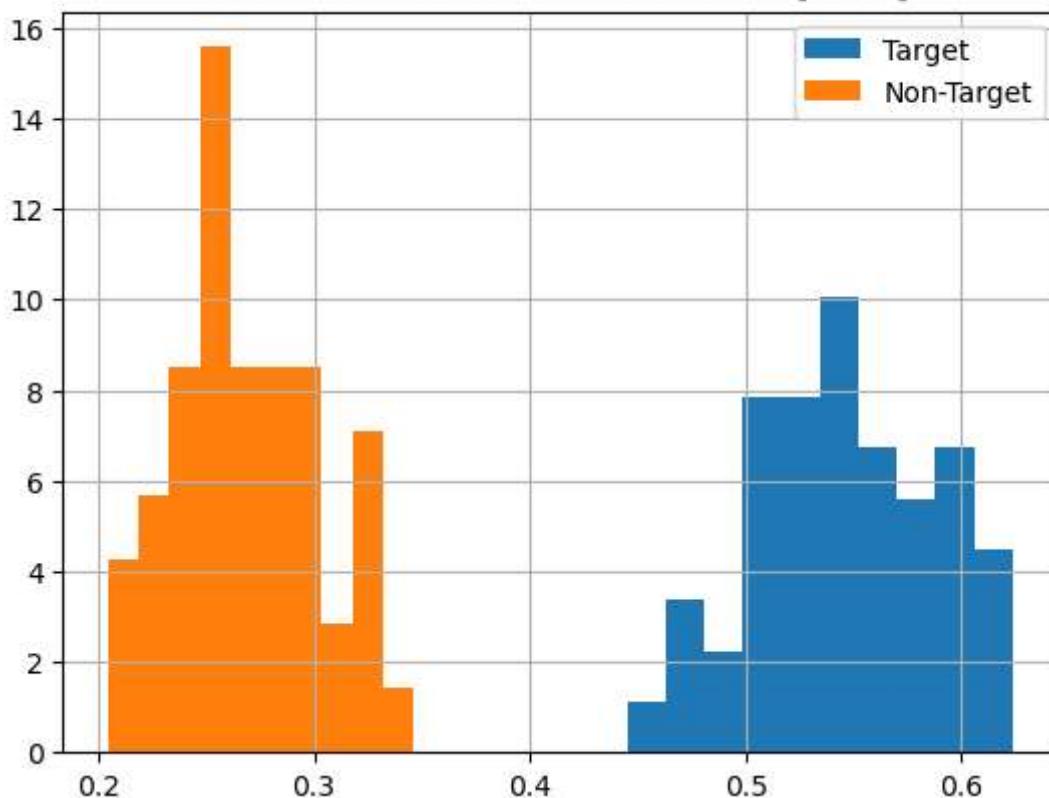
Target Mean: 0.5458552446787204, Std: 0.041959881236350166

Non-Target Mean: 0.26745967183162395, Std: 0.032602831804938254

Confidence interval: ConfidenceInterval(low=0.26578290011561495, high=inf), T-Score: 36.674196, Degree of freedom: 92.36089310836287, P-Value: 3.805948243951511e-57

With a significance level of 0.05000, the hypothesis is rejected

### Distribution of Means for Channel: ['PO8']



We can see from the above distribution, that the hypothesis that the P300 amplitude of a target signal is higher in all regions, particularly in Oz (occipital), PO7, and PO8 (Parieto-occipital) regions. So, the null hypothesis is rejected for all channels

### Statistical significance of P300 for individual waveform

In the above example, we collected a small sample and extracted the average waveform from the sample. We then used the mean waveform as our sample data. We calculated the mean amplitude of the average waveform to find the significance level. But now we will use individual waveform as our sample and check the significance level of P300.

Therefore, we would like to test the following - Target waveforms have a higher peak amplitude than non-target waveforms. i.e.

$$H_0 : \mu_{target} - \mu_{nontarget} \leq 0$$

$$H_1 : \mu_{target} - \mu_{nontarget} > 0$$

In [409]: sample\_size = 50

```

np.random.seed(34324)

def view_distribution():
    for channel in range(x_target.shape[1]):
        ma_t = []
        ma_nt = []
        for _ in range(sample_size):
            # Get average waveform from sample
            t = x_target[ np.random.choice(np.arange(x_target.shape[0]), size=1, replace=False), channel, :]
            nt = x_nontarget[ np.random.choice(np.arange(x_nontarget.shape[0]), size=1, replace=False), channel, :]

            # Remove negative portion of average waveform as P300 deflection is positive
            t[t<0] = 0
            nt[nt<0] = 0

            # Calculate the average amplitude of each signal
            ma_nt.append(np.mean(nt))
            ma_t.append(np.mean(t))

        mean_target_amplitude = np.mean(ma_t)
        std_target_amplitude = np.std(ma_t)

        mean_nontarget_amplitude = np.mean(ma_nt)
        std_nontarget_amplitude = np.std(ma_nt)

        print(f"Target Mean: {mean_target_amplitude}, Std: {std_target_amplitude}")
        print(f"Non-Target Mean: {mean_nontarget_amplitude}, Std: {std_nontarget_amplitude}")

        # T-Statistics
        confidence = 0.95
        d = stats.ttest_ind(ma_t, ma_nt, equal_var=False, alternative='greater')
        print(f"Confidence interval: {d.confidence_interval(confidence)}, T-Score: {d.statistic:.6f}, Degree of freedom: {d.df}, P-Value: {d.pvalue}")
        if d.pvalue > (1-confidence):
            print(f"With a significance level of {1-confidence:.5f}, the hypothesis is accepted")
        else:
            print(f"With a significance level of {1-confidence:.5f}, the hypothesis is rejected")

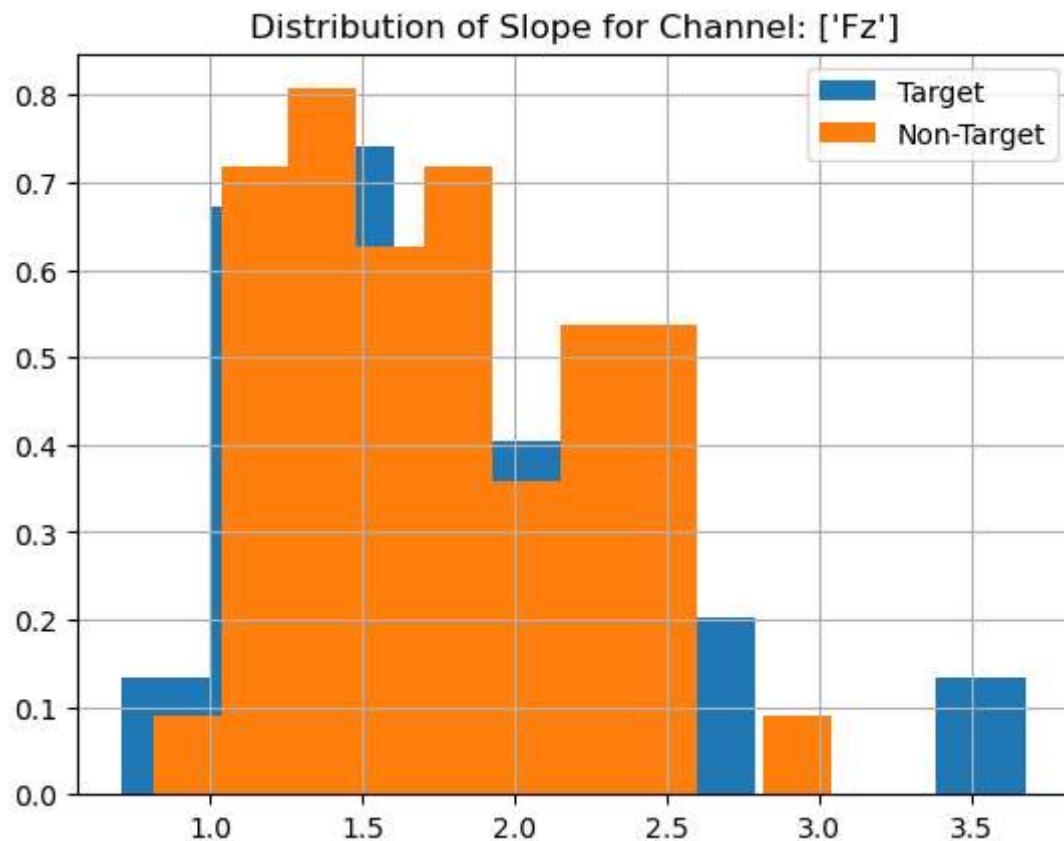
plt.figure()
plt.title(f"Distribution of Slope for Channel: {get_channel_name(patient_data[0], channel)}")

```

```
plt.hist(ma_t, density=True, label='Target')
plt.hist(ma_nt, density=True, label='Non-Target')
plt.grid()
plt.legend()
plt.show()

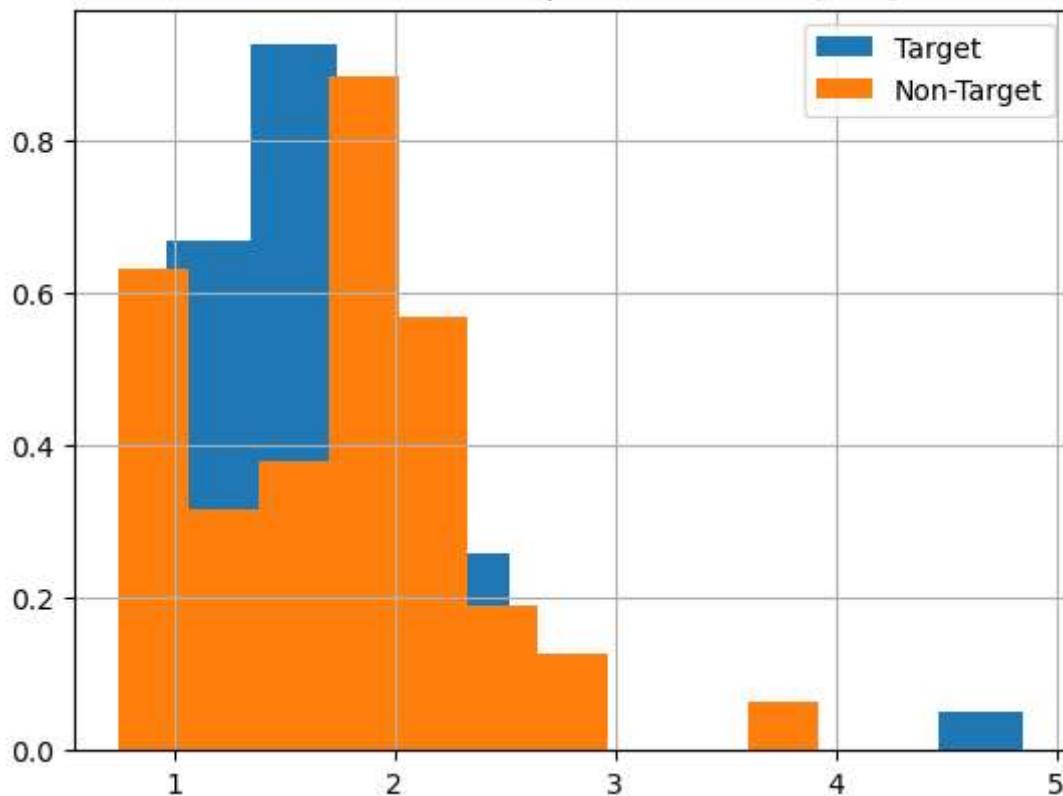
view_distribution()
```

Target Mean: 1.7757286746052214, Std: 0.6039467004548362  
Non-Target Mean: 1.757631679358547, Std: 0.4876469488151088  
Confidence interval: ConfidenceInterval(low=-0.16612234625335642, high=inf),  
T-Score: 0.163195, Degree of freedom: 93.83461827019623, P-Value: 0.435357828  
5921234  
With a significance level of 0.05000, the hypothesis is accepted



Target Mean: 1.7229771446465694, Std: 0.6192546002621162  
Non-Target Mean: 1.7469403342880043, Std: 0.6094589246443836  
Confidence interval: ConfidenceInterval(low=-0.23007587930674636, high=inf),  
T-Score: -0.193060, Degree of freedom: 97.97509491700993, P-Value: 0.57634434  
56802365  
With a significance level of 0.05000, the hypothesis is accepted

## Distribution of Slope for Channel: ['Cz']



Target Mean: 1.703224966352861, Std: 0.5683094164770924

Non-Target Mean: 1.6051949046809277, Std: 0.43781754507998993

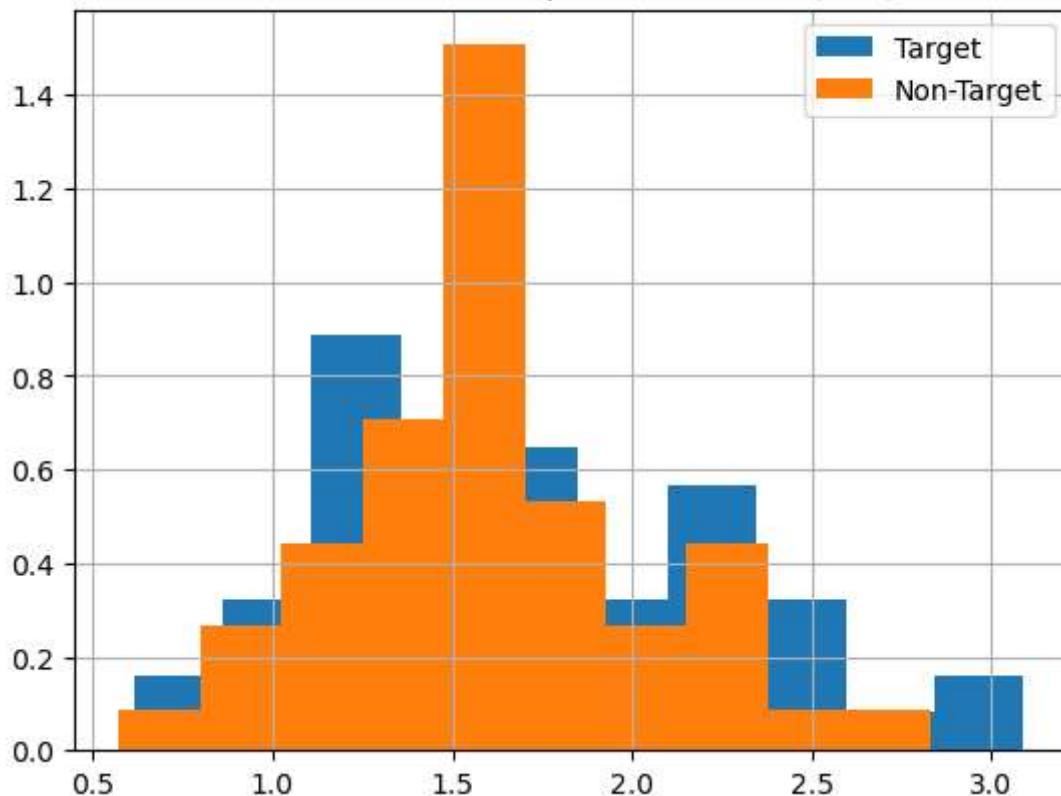
Confidence interval: ConfidenceInterval(low=-0.07225798694967697, high=inf),

T-Score: 0.956527, Degree of freedom: 92.0120687949657, P-Value: 0.1706562837

7350206

With a significance level of 0.05000, the hypothesis is accepted

## Distribution of Slope for Channel: ['Pz']



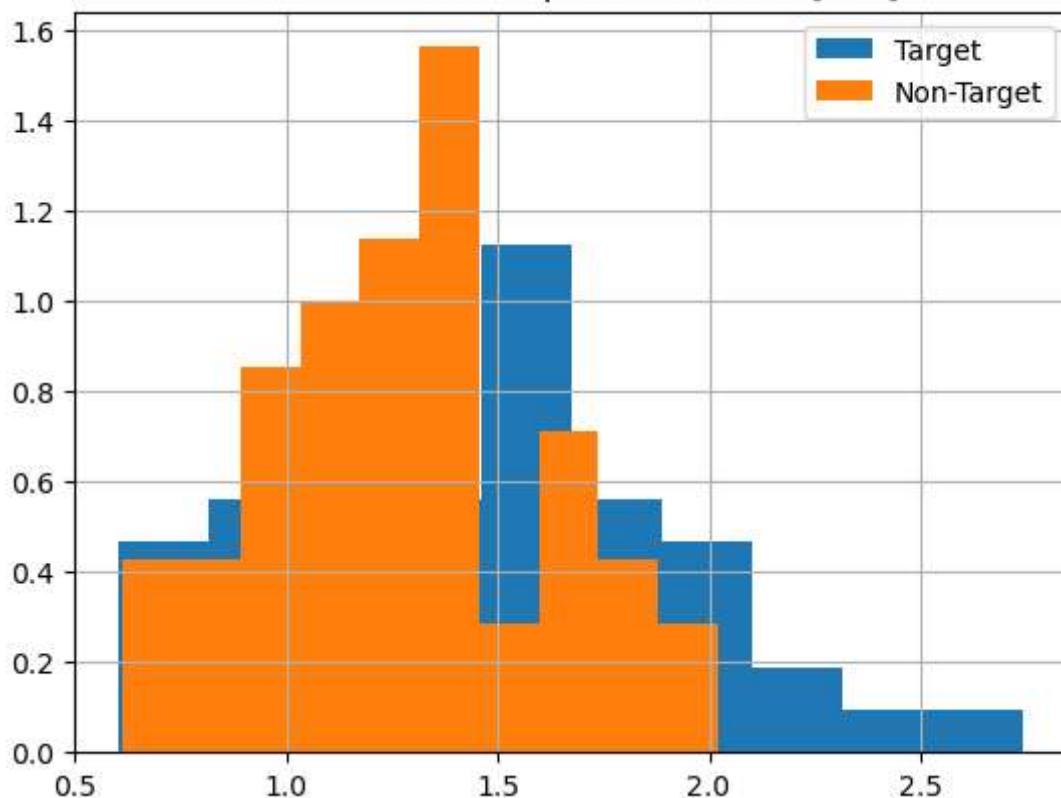
Target Mean: 1.4463395989933727, Std: 0.46337513612409925

Non-Target Mean: 1.2734128427544869, Std: 0.3256753918597596

Confidence interval: ConfidenceInterval(low=0.03842303454836368, high=inf), T-Score: 2.137253, Degree of freedom: 87.91404666185488, P-Value: 0.0176770471  
1345975

With a significance level of 0.05000, the hypothesis is rejected

## Distribution of Slope for Channel: ['Oz']



Target Mean: 1.6660037054365981, Std: 0.5050721905639041

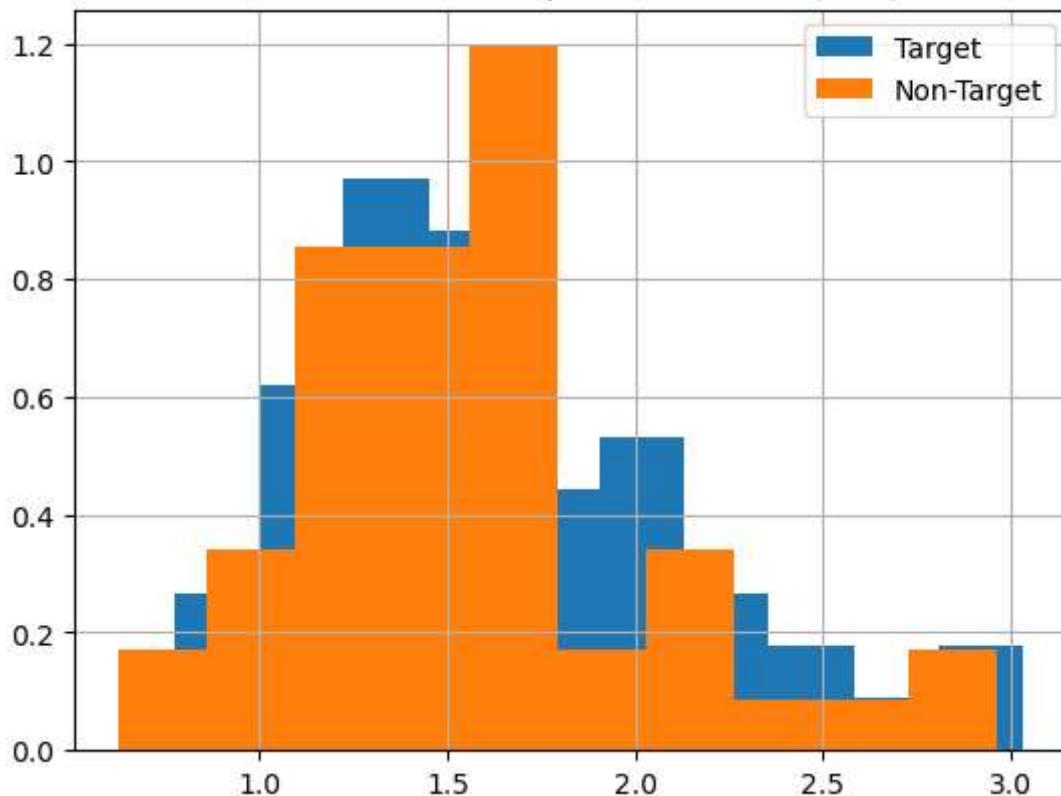
Non-Target Mean: 1.5636259004868387, Std: 0.47288533361117724

Confidence interval: ConfidenceInterval(low=-0.06176136811145666, high=inf),

T-Score: 1.035771, Degree of freedom: 97.57811971884588, P-Value: 0.151434519  
57433365

With a significance level of 0.05000, the hypothesis is accepted

## Distribution of Slope for Channel: ['P3']



Target Mean: 1.5822122994472028, Std: 0.4693581708174683

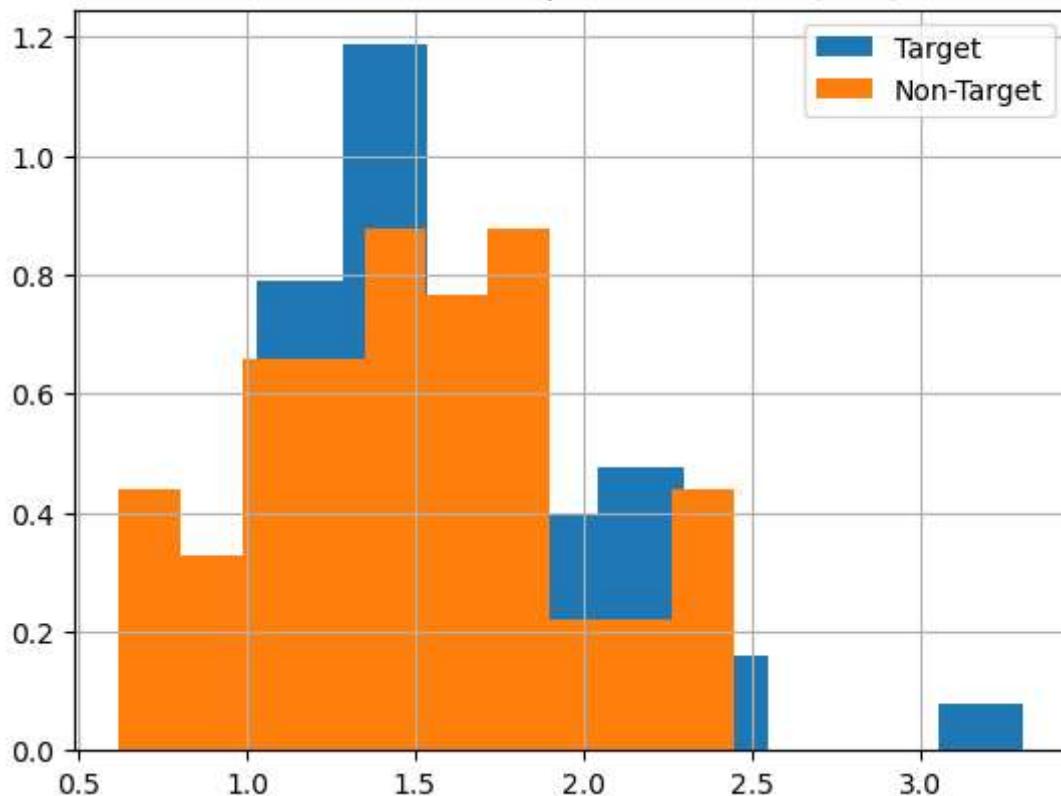
Non-Target Mean: 1.4957646947093632, Std: 0.4510824364362753

Confidence interval: ConfidenceInterval(low=-0.06798078015472045, high=inf),

T-Score: 0.929575, Degree of freedom: 97.84582354504433, P-Value: 0.177439322  
03970747

With a significance level of 0.05000, the hypothesis is accepted

## Distribution of Slope for Channel: ['P4']

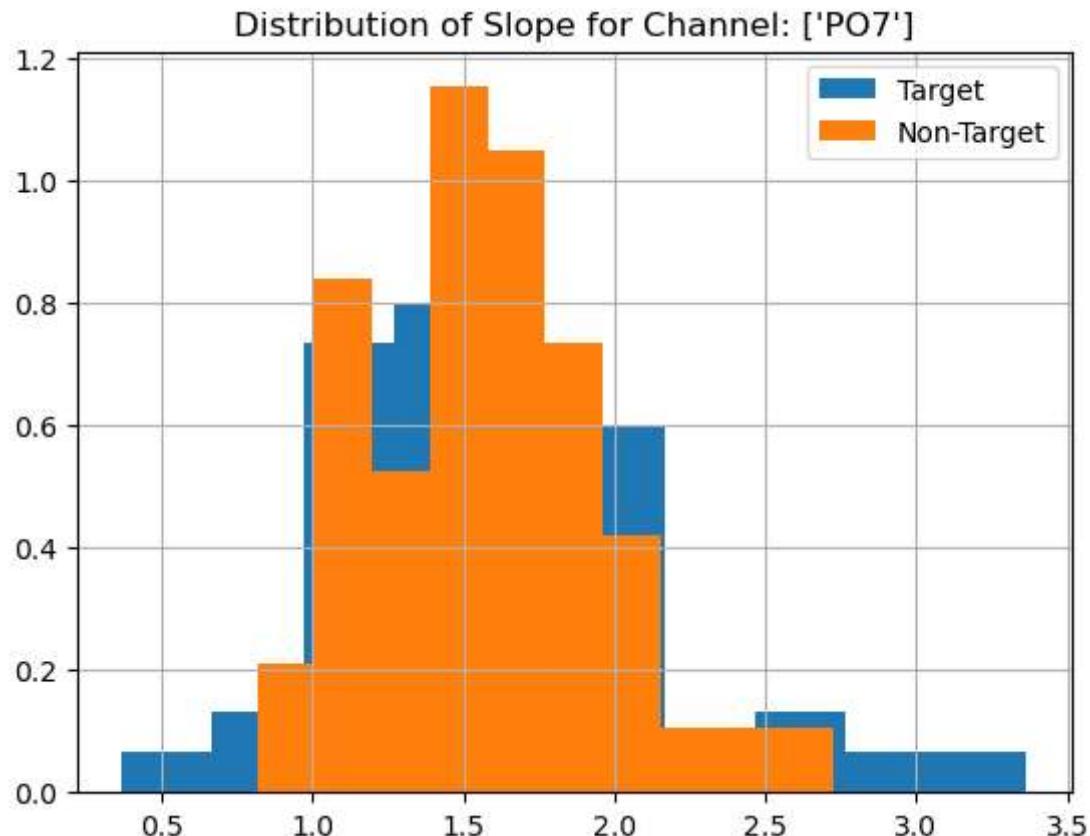


Target Mean: 1.6066030407183163, Std: 0.5267489352415895

Non-Target Mean: 1.5873330250181223, Std: 0.39253402842968754

Confidence interval: ConfidenceInterval(low=-0.13668789800598746, high=inf),  
T-Score: 0.205336, Degree of freedom: 90.59465194575809, P-Value: 0.418885008  
82826646

With a significance level of 0.05000, the hypothesis is accepted



Target Mean: 1.4212659408932398, Std: 0.43980892063267235

Non-Target Mean: 1.2336618203493774, Std: 0.38362088368974845

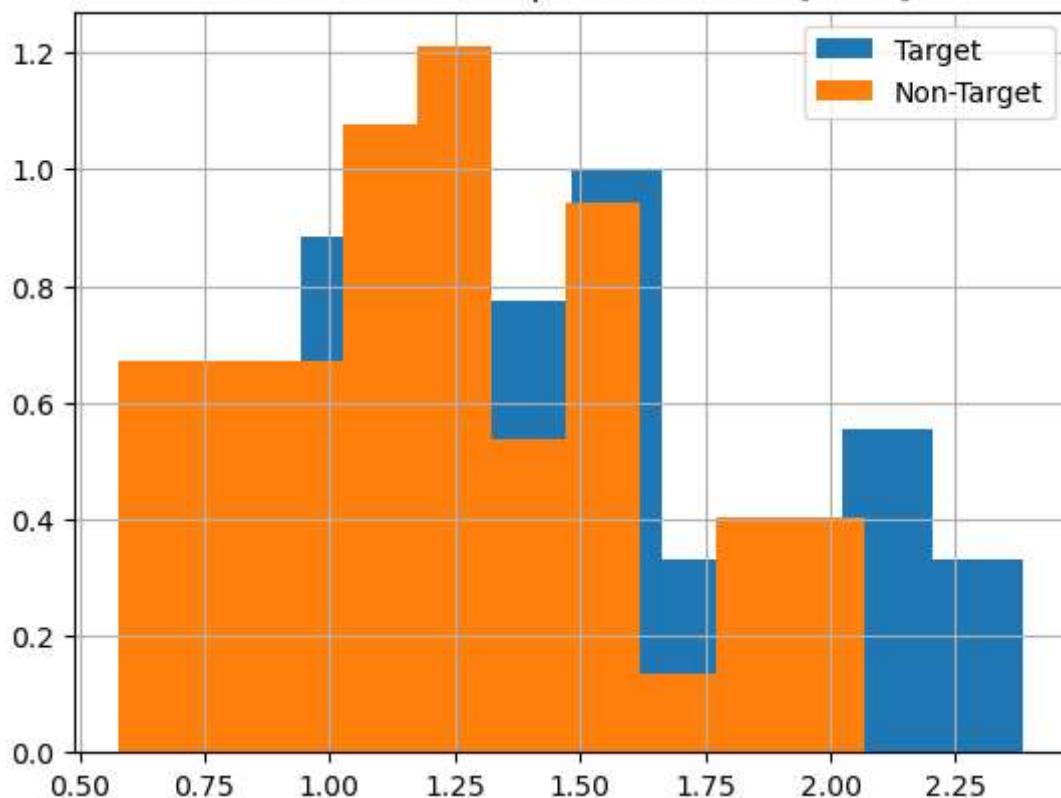
Confidence interval: ConfidenceInterval(low=0.049135601037613275, high=inf),

T-Score: 2.250195, Degree of freedom: 96.22440138112383, P-Value: 0.013357092

3850459

With a significance level of 0.05000, the hypothesis is rejected

### Distribution of Slope for Channel: ['PO8']



Even though the P300 amplitude was significant when we collected a bunch of samples for each class and calculated their average waveforms, it is not significant when we calculate the P300 amplitude of individual amplitude. However, it is significant in Oz channel. Let us check if deep neural networks can perform a better classification. For input we will use the following datasets:

1. Simple waveform
2. Stepwise LDA

## 1. Create Dataset

```
In [410]: X_train, X_test = torch.tensor(X_train, requires_grad=False).float(), torch.tensor(X_test, requires_grad=False).float()
y_train, y_test = torch.tensor(y_train, requires_grad=False).type(torch.LongTensor), torch.tensor(y_test, requires_grad=False).type(torch.LongTensor)
```

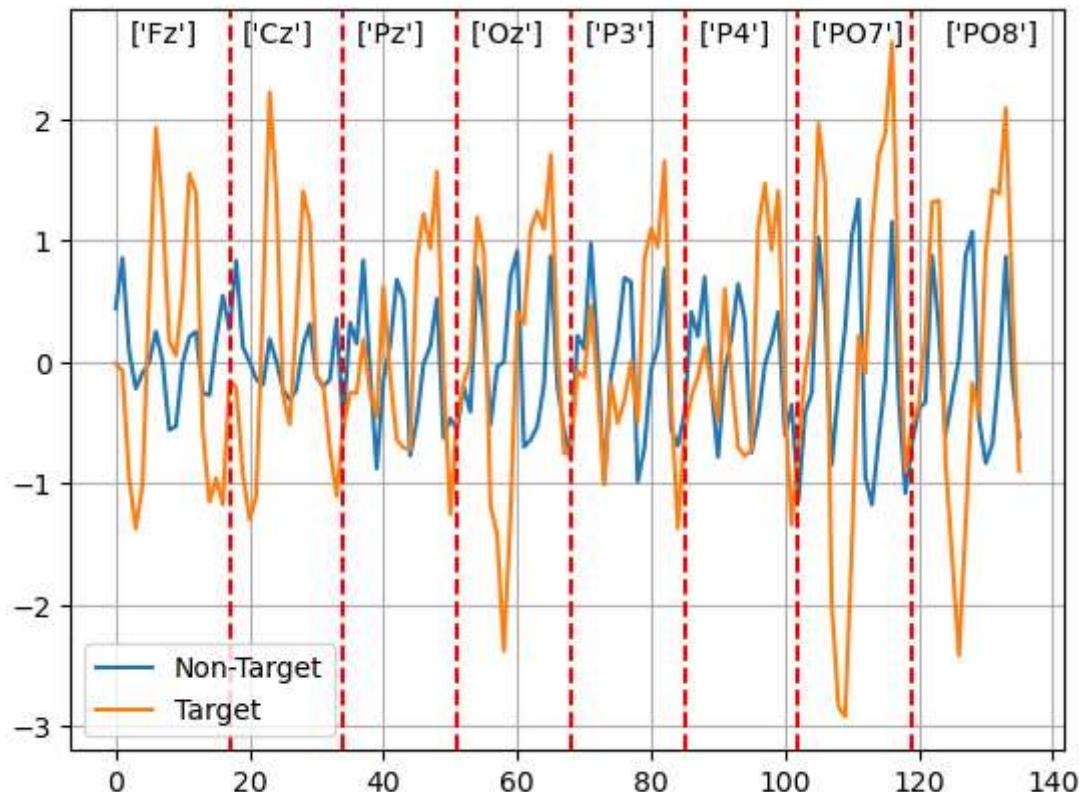
```
In [413]: plt.figure()

plt.plot(torch.mean(X_train[np.where(y_train == 0)[0]], axis=0).numpy(), label='Non-Target')
plt.plot(torch.mean(X_train[np.where(y_train == 1)[0]], axis=0).numpy(), label='Target')

a = np.max(torch.mean(X_train[np.where(y_train == 1)[0]], axis=0).numpy())
for i in range(1, 8):
    p = X_train.shape[-1]//8*i
    plt.axvline(x=p, color='red', linestyle='--')
    plt.text(p-15, a, get_channel_name(patient_data[0], i-1))
    if i == 7:
        plt.text(p+5, a, get_channel_name(patient_data[0], i))

plt.grid()
plt.legend()

plt.show()
```



## 2. Training function

```
In [48]: @torch.no_grad()
def calculate_accuracy(prediction, true_labels):
    prediction_classes = torch.argmax(prediction, axis=1)
    return (prediction_classes == true_labels).float().mean()

def train(epochs, batch_size, loss_function, optimizer, model, x, y, print_every=200, learning_rate_decay=None):
    train_accuracy = []
    train_loss = []
    vals = tqdm(range(epochs), desc=f'Epoch: 0, train loss: -, Accuracy: -')

    scheduler = None
    if not (learning_rate_decay is None):
        scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=learning_rate_decay)

    for epoch in vals:
        avg_cost = 0

        # Shuffle training data
        dataset = torch.randperm(x.shape[0])

        # For each batch
        for batch in range(0, dataset.size()[0], batch_size):
            model.train()

            range_end = batch+batch_size
            if range_end + batch_size >= dataset.size()[0]:
                range_end = range_end+batch_size
            # Get train and test data
            x_batch = x[dataset[batch:range_end]]
            y_batch = y[dataset[batch:range_end]]

            # Forward pass
            optimizer.zero_grad()
            prediction = model(x_batch)

            # Calculate Loss
            loss = loss_function(prediction, y_batch)

            # Backpropagation
            loss.backward() # Calculate gradient
            optimizer.step() # Upgrade gradients

            # Calculate metrics
            train_loss.append(loss.item())
            train_accuracy.append(calculate_accuracy(prediction, y_batch))

            if batch % print_every == 0:
                vals.set_description(f'Epoch: {epoch}, train loss: {train_loss[-1]}, accuracy: {train_accuracy[-1]}')
```

```
[ -1]:.5f}, Accuracy: {train_accuracy[-1]*100:.3f}')
```

```
    if not (scheduler is None):
        scheduler.step() # Learning rate decay
```

```
return train_loss, train_accuracy
```

### 3. Neural Network Model

```
In [49]: class ModelDNN(torch.nn.Module):
    def __init__(self, sample_x, total_classes, first_layer_neuron=1024, hidden_layers_neurons=[512, ], activation=torch.nn.ReLU):
        super(ModelDNN, self).__init__()

        # Reshape signal so that it has only two dimensions:
        sample_x = self.get_valid_x(sample_x)
        print(f"Layer 1 input: {sample_x.shape}")

        # Layer 1- Fully Connected Layer
        self.fc1 = torch.nn.Linear(sample_x.shape[-1], first_layer_neuron)
        self.act1 = activation()
        self.drop1 = torch.nn.Dropout(0.4)

        self.layer1 = torch.nn.Sequential(self.fc1, self.act1, self.drop1)

        self.layer1.eval()
        with torch.no_grad():
            out = self.layer1(sample_x)
            print(f"Layer 1 output: {out.shape}")
        self.layer1.train()

        self.layers = torch.nn.ModuleList()
        self.layers.append(self.layer1)
        self.activation_layer_name = [type(self.act1).__name__]

        # Add additional hidden layers
        for i in range(len(hidden_layers_neurons)):
            hidden_fc = torch.nn.Linear(out.shape[-1], hidden_layers_neurons[i])
            hidden_act = activation()
            hidden_drop = torch.nn.Dropout(0.2)
            hidden_layer = torch.nn.Sequential(hidden_fc, hidden_act, hidden_drop)

            hidden_layer.eval()
            with torch.no_grad():
                out = hidden_layer(out)
                print(f"Layer {i+2} output: {out.shape}")
            hidden_layer.train()
            self.layers.append(hidden_layer)
            self.activation_layer_name.append(type(hidden_act).__name__)

        # Add output layer
        self.out_fc = torch.nn.Linear(out.shape[1], total_classes, bias=False)

        self.out_fc.eval()
        with torch.no_grad():
            out = self.out_fc(out)
            print(f"Final output: {out.shape}")
        self.out_fc.train()
```

```
def forward(self, x):
    out = self.get_valid_x(x)

    for layer in self.layers:
        out = layer(out)

    return self.out_fc(out)

def get_valid_x(self, x):
    if len(x.shape) > 2:
        x = x.reshape((x.shape[0], -1))
    return x

def show_weight_distribution(self):
    params = []
    for param in self.parameters():
        if len(param.shape) >= 2:
            params.append(param)
    cols = 2
    rows = len(params)/2
    if int(rows) != rows:
        rows = int(rows) + 1
    else:
        rows = int(rows)

    fig, ax = plt.subplots(rows, cols, figsize=(10, 10))
    plt.suptitle("Weight Distribution")

    current = 0
    for i in range(rows):
        for j in range(cols):
            ax[i, j].set_title(f"Param: {current+1}")
            ax[i, j].hist(params[current].detach().numpy().reshape(-1))
            ax[i, j].grid()
            current += 1

    plt.subplots_adjust(hspace=0.5)
    plt.show()

def show_weights(self):
    self.eval()
    with torch.no_grad():
        cols = 2
        rows = len(model.layers)/cols
        if rows == int(rows):
            rows = int(rows)
        else:
            rows = int(rows) + 1

        fig, ax = plt.subplots(rows, cols, figsize=(10, 10))
        plt.suptitle("Layer Weights")
```

```

        current = 0
        current_i = 0
        current_j = 0
        for layer in self.layers:
            if current >= len(model.layers):
                break
            params = layer.parameters()
            for p in params:
                if current >= len(model.layers):
                    break
                if len(p.shape) == 2:
                    weights = p.numpy()
                    ax[current_i, current_j].set_title(f"Weight - Layer {current+1}")
                    im = ax[current_i, current_j].pcolormesh(weights, cmap='gray')
                    fig.colorbar(im, ax=ax[current_i, current_j], orientation='vertical')

            # Draw vertical Line where neurons are dead
            zeros = np.where(np.sum(weights) == 0)[0]
            for ind in zeros:
                ax[current_i, current_j].axvline(x=ind, color='b')
                current += 1
            if current_j + 1 >= cols:
                current_j = 0
                current_i += 1
            else:
                current_j += 1

        plt.subplots_adjust(hspace=0.5)
        plt.show()

    self.train()

```

```

def show_output_distribution(self, x):
    self.eval()
    with torch.no_grad():
        out = self.get_valid_x(x)
        activated_outs = []
        for i in range(len(self.layers)):
            module = self.layers[i]
            for layer in module:
                out = layer(out)
                if type(layer).__name__ == self.activation_layer_name[i]:
                    activated_outs.append(out)

        cols = 2
        rows = len(activated_outs)/2
        if int(rows) != rows:
            rows = int(rows) + 1

```

```
    else:
        rows = int(rows)

        fig, ax = plt.subplots(rows, cols, figsize=(10, 10))
        plt.suptitle("Activated Output distribution")

        current = 0
        for i in range(rows):
            for j in range(cols):
                if current >= len(activated_outs):
                    break
                ax[i, j].set_title(f"Output: {current+1}")
                ax[i, j].hist(activated_outs[current].detach().numpy().reshape(-1))
                ax[i, j].grid()
                current += 1

        plt.subplots_adjust(hspace=0.5)
        plt.show()

    self.train()
```

## 4. Classification - Simple Waveform

```
In [414]: model = ModelDNN(X_test, np.unique(y_test).size, first_layer_neuron=512, hidden_layers_neurons=[128, 64, 32, 16])
epochs = 500
batch_size = 32
learning_rate = 1e-2

loss, accuracy = train(
    epochs=epochs,
    batch_size=batch_size,
    loss_function=torch.nn.CrossEntropyLoss(),
    optimizer=torch.optim.Adam(params=model.parameters(), lr=learning_rate),
    model=model,
    x=X_train,
    y=y_train,
    print_every=200,
    learning_rate_decay=0.9
)

running_mean = epochs
plt.figure(figsize=(10, 10))
plt.suptitle("Training")

plt.subplot(2, 1, 1)
plt.title("Loss")
plt.plot(np.array(loss).reshape(-1, running_mean).mean(axis=1))
plt.grid()

plt.subplot(2, 1, 2)
plt.title("Accuracy")
plt.plot(np.array(accuracy).reshape(-1, running_mean).mean(axis=1))

plt.grid()

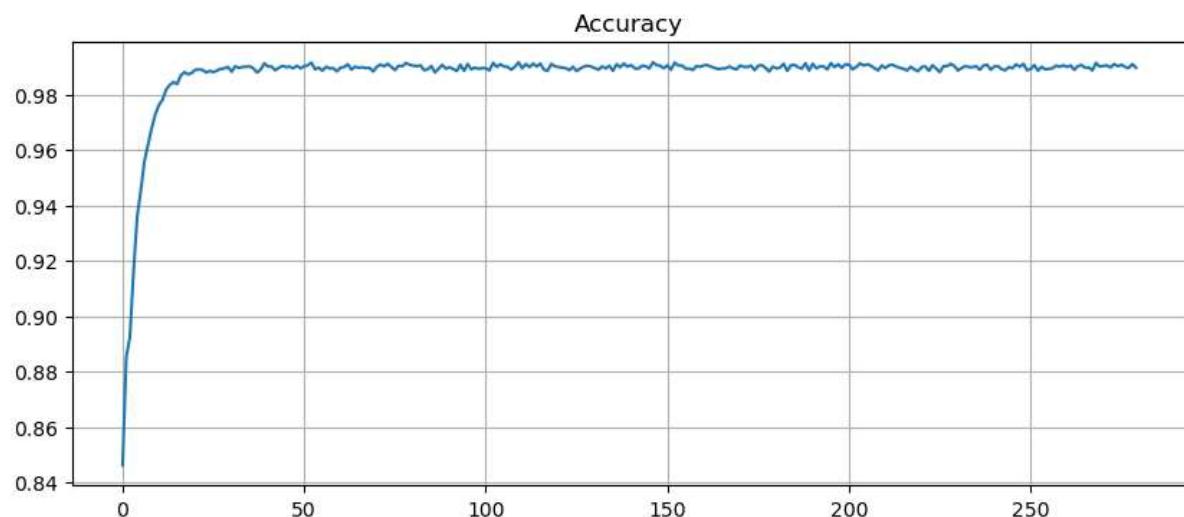
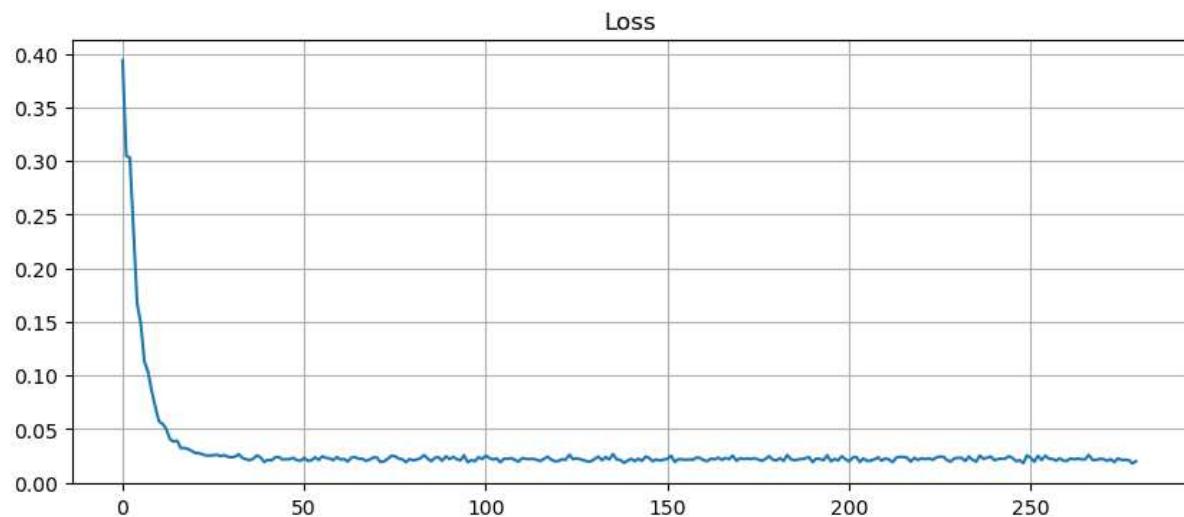
plt.subplots_adjust(hspace=0.5)

plt.show()
```

```
Layer 1 input: torch.Size([2240, 136])
Layer 1 output: torch.Size([2240, 512])
Layer 2 output: torch.Size([2240, 128])
Layer 3 output: torch.Size([2240, 64])
Layer 4 output: torch.Size([2240, 32])
Layer 5 output: torch.Size([2240, 16])
Final output: torch.Size([2240, 2])
```

```
Epoch: 499, train loss: 0.06286, Accuracy: 93.750000: 100% |██████████| 500/500 [08:14<00:00, 1.01it/s]
```

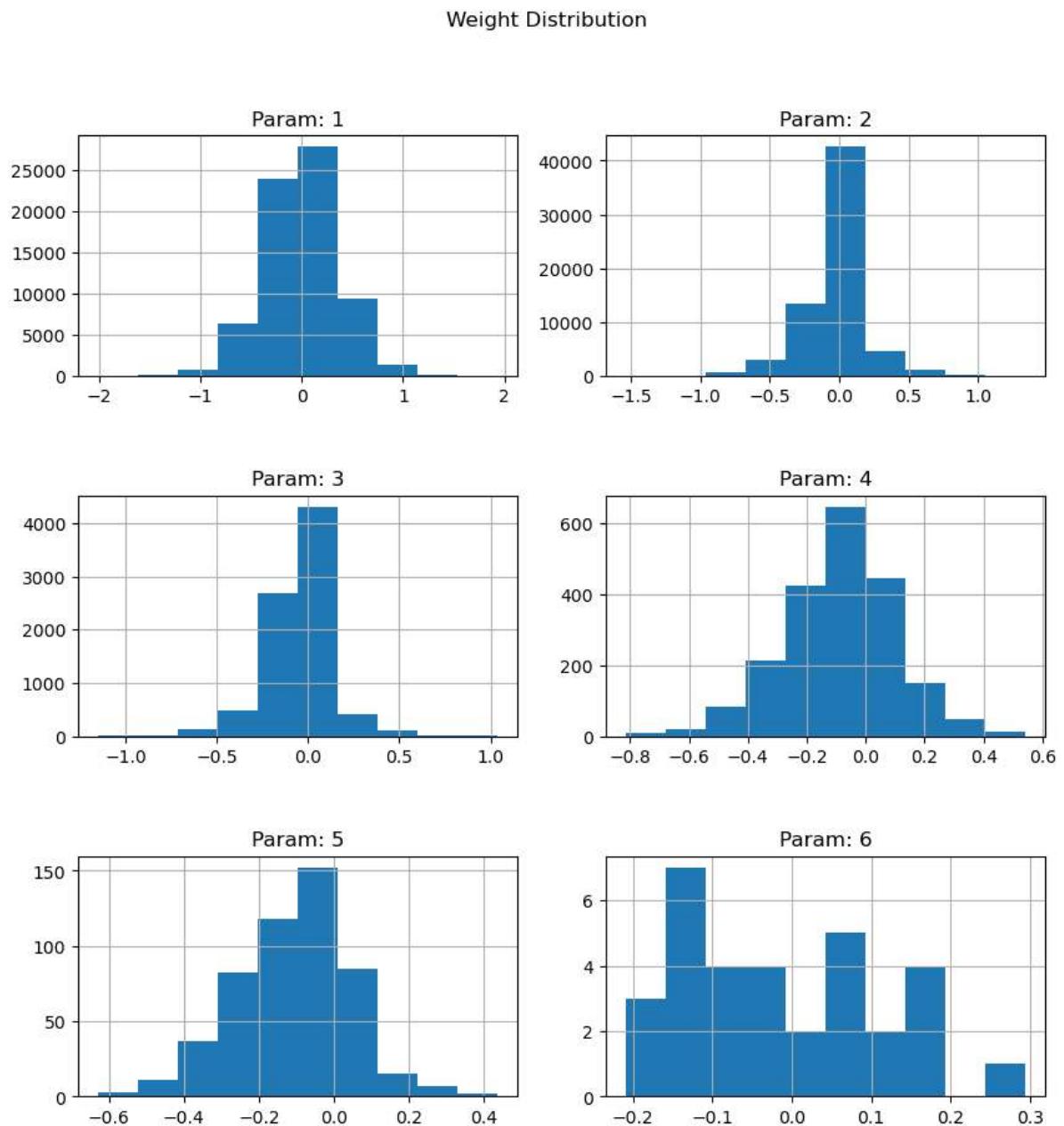
Training



```
In [415]: with torch.no_grad():
    print(f"Accuracy: {calculate_accuracy(model(X_test), y_test)*100:.3f}%")
```

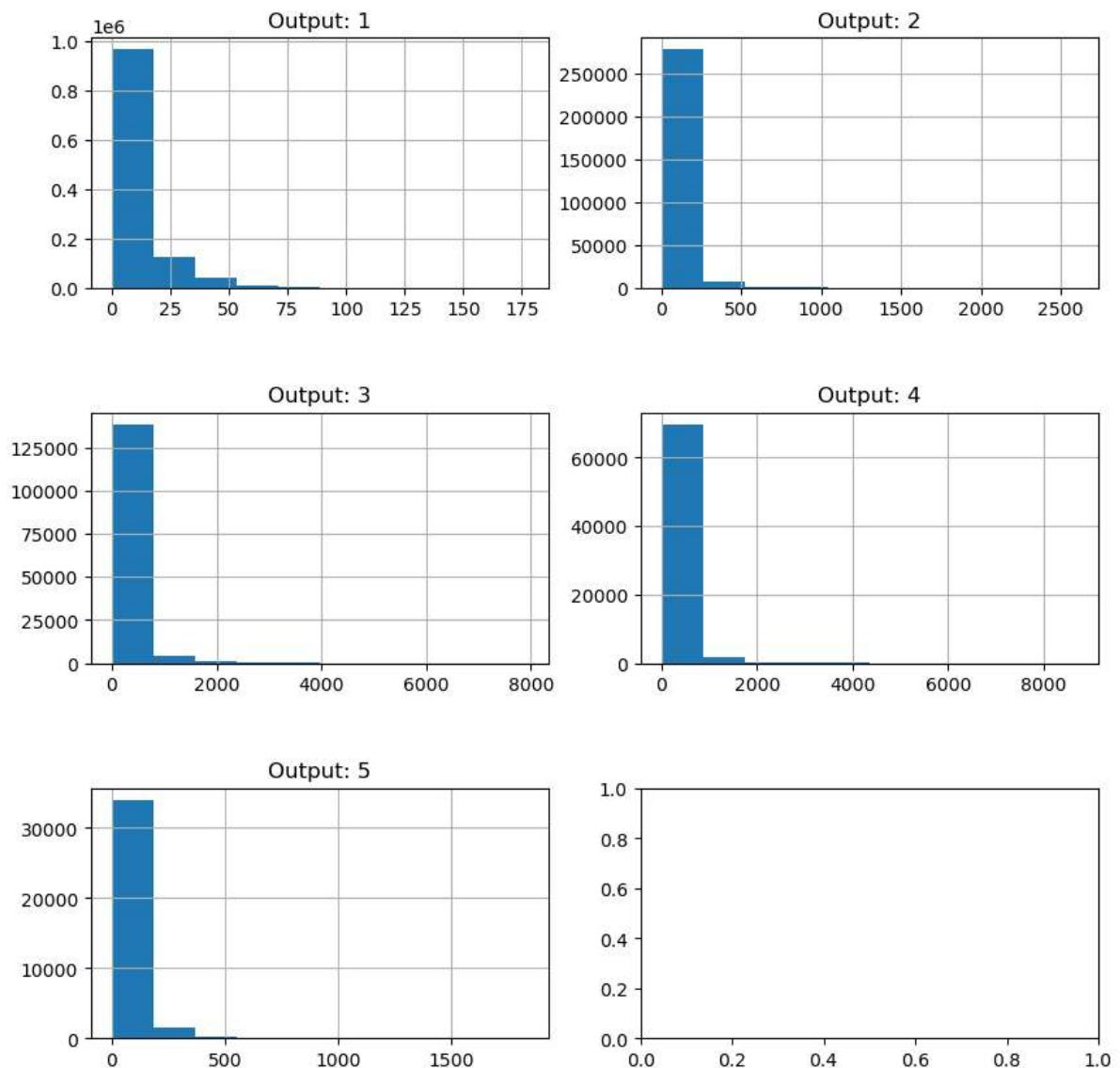
Accuracy: 97.946%

```
In [418]: model.show_weight_distribution()
```

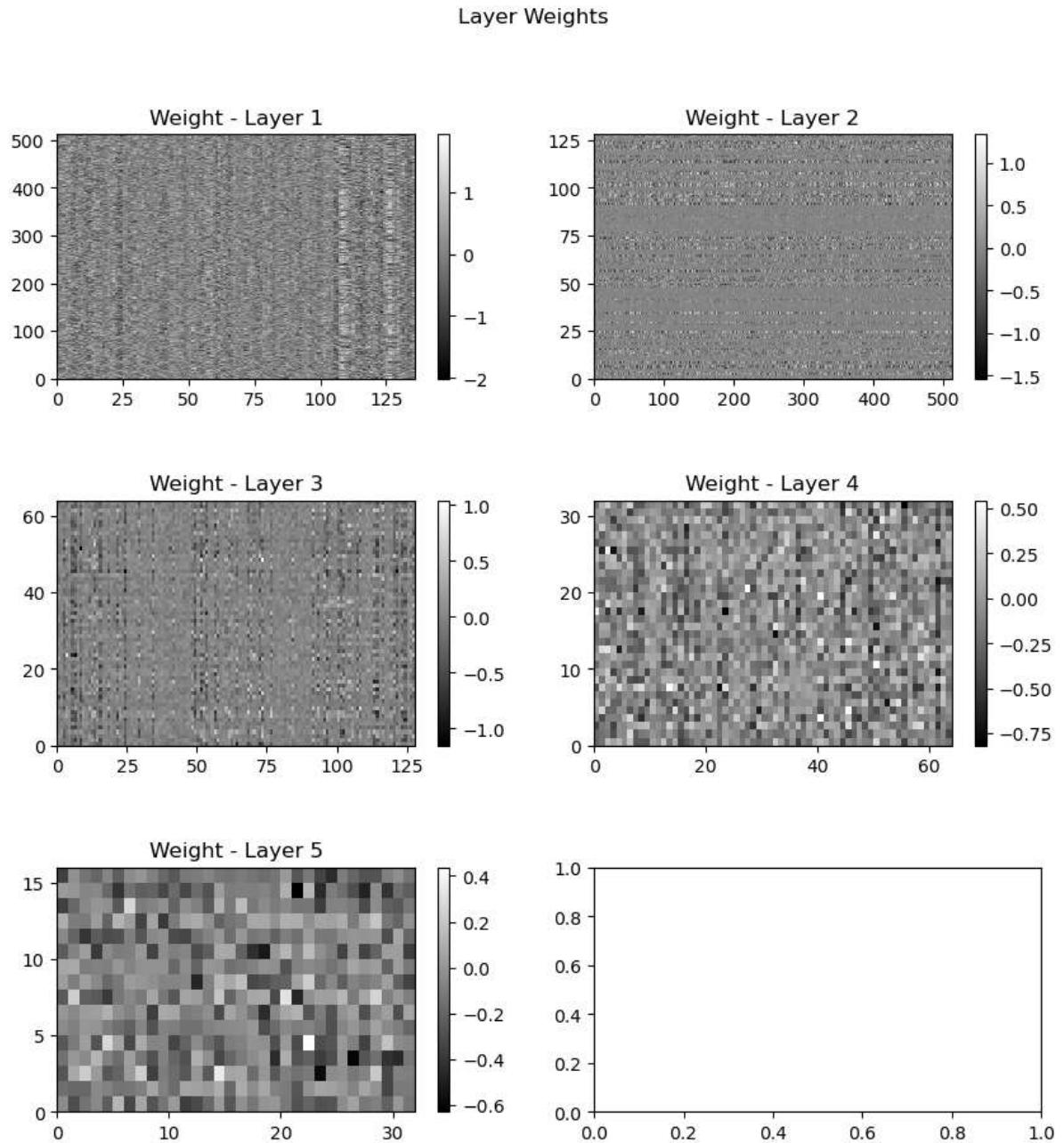


```
In [419]: model.show_output_distribution(X_test)
```

Activated Output distribution



```
In [420]: model.show_weights()
```



## 5. Classification - Stepwise LDA (35 features)

```
In [421]: lda = StepwiseLDA(n_components=35)
lda.fit(X_train.numpy(), y_train.numpy())
model = ModelDNN(
    torch.tensor(lda.transform(X_test.numpy()).real).float(),
    np.unique(y_test.numpy()).size,
    first_layer_neuron=512,
    hidden_layers_neurons=[128, 64, 32, 16],
    activation=torch.nn.Tanh
)
epochs = 50
batch_size = 32
learning_rate = 1e-3

loss, accuracy = train(
    epochs=epochs,
    batch_size=batch_size,
    loss_function=torch.nn.CrossEntropyLoss(),
    optimizer=torch.optim.Adam(params=model.parameters(), lr=learning_rate),
    model=model,
    x=torch.tensor(lda.transform(X_train.numpy()).real).float(),
    y=y_train,
    print_every=200,
    learning_rate_decay=0.95
)

running_mean = epochs
plt.figure(figsize=(10, 10))
plt.suptitle("Training")

plt.subplot(2, 1, 1)
plt.title("Loss")
plt.plot(np.array(loss).reshape(-1, running_mean).mean(axis=1))
plt.grid()

plt.subplot(2, 1, 2)
plt.title("Accuracy")
plt.plot(np.array(accuracy).reshape(-1, running_mean).mean(axis=1))

plt.grid()

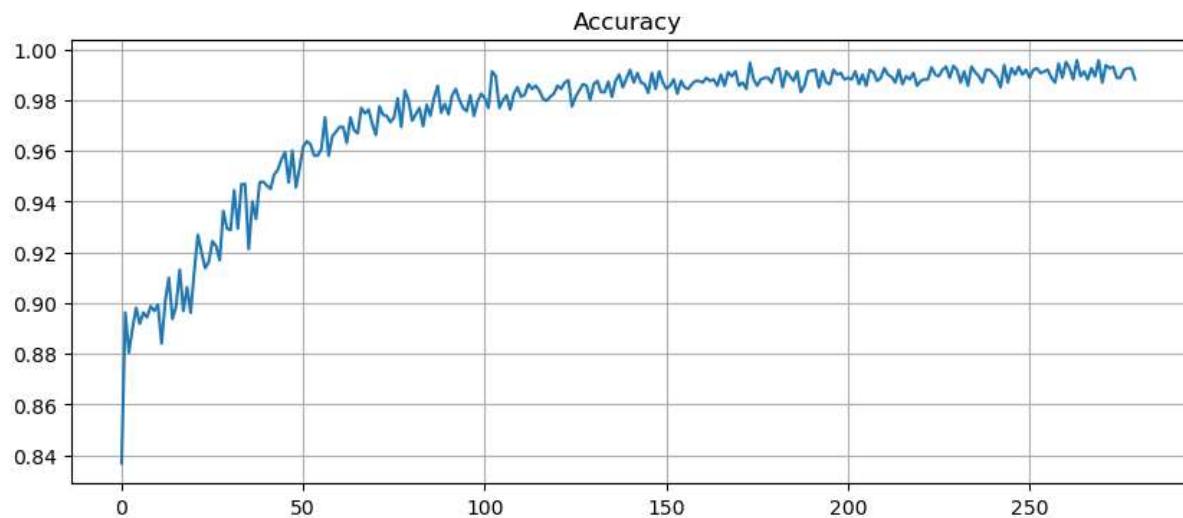
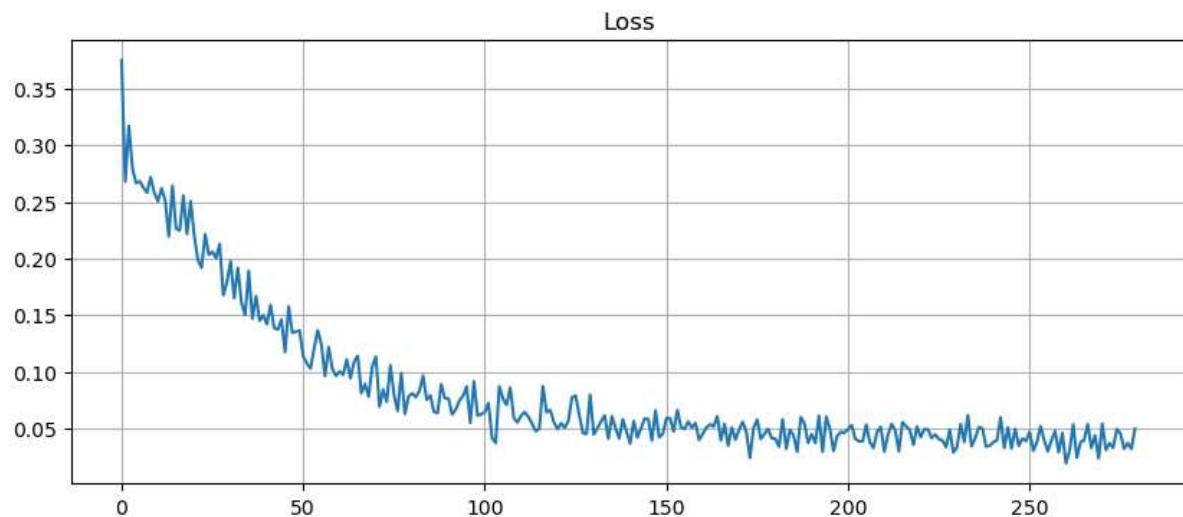
plt.subplots_adjust(hspace=0.5)

plt.show()
```

```
Layer 1 input: torch.Size([2240, 35])
Layer 1 output: torch.Size([2240, 512])
Layer 2 output: torch.Size([2240, 128])
Layer 3 output: torch.Size([2240, 64])
Layer 4 output: torch.Size([2240, 32])
Layer 5 output: torch.Size([2240, 16])
Final output: torch.Size([2240, 2])
```

```
Epoch: 49, train loss: 0.00611, Accuracy: 100.000000: 100%|██████████|
██████████| 50/50 [00:42<00:00, 1.17it/s]
```

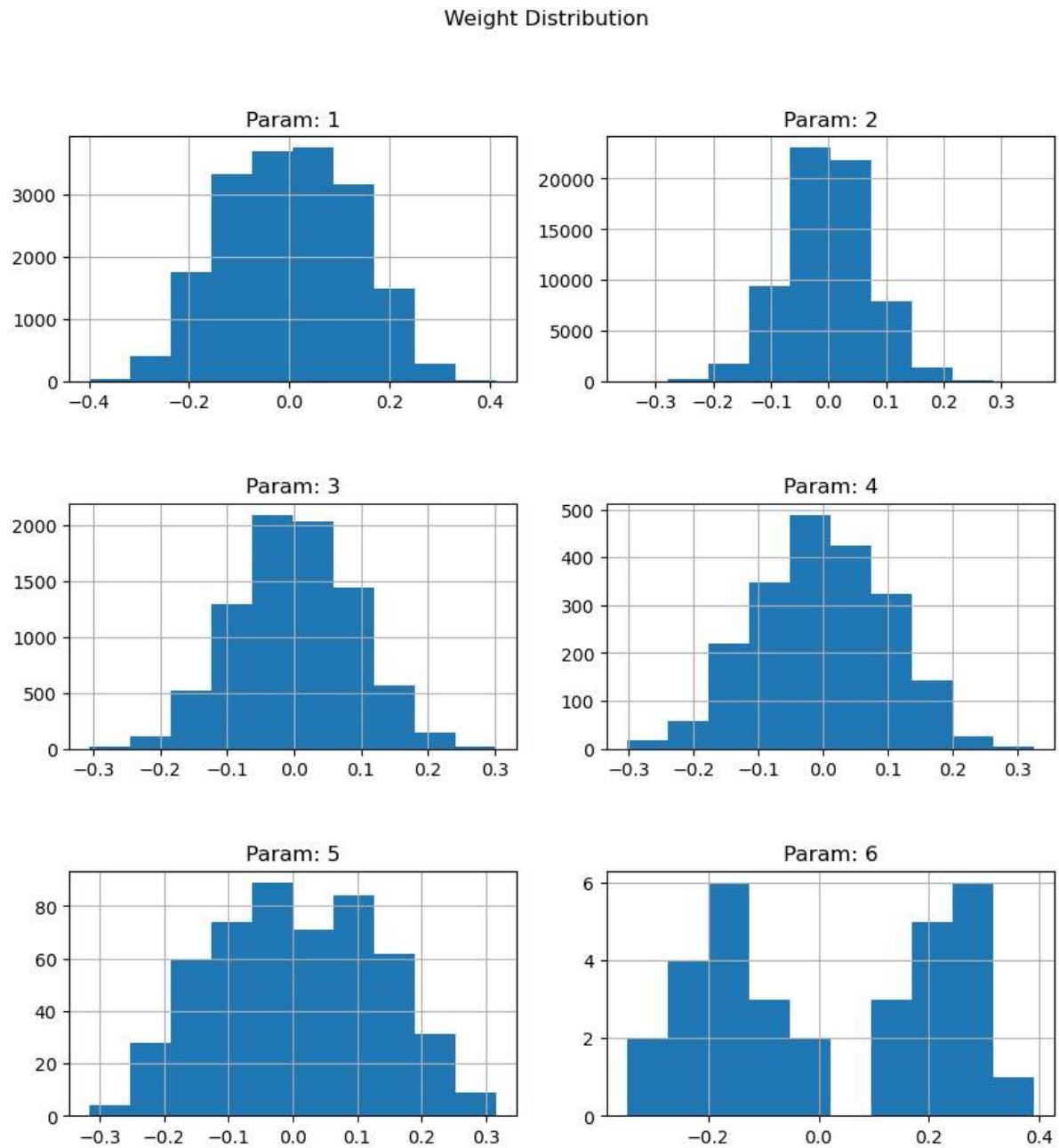
Training



```
In [422]: with torch.no_grad():
    print(f"Accuracy: {calculate_accuracy(model(torch.tensor(lda.transform(X_t
est.numpy()).real).float()), y_test)*100:.3f}%")
```

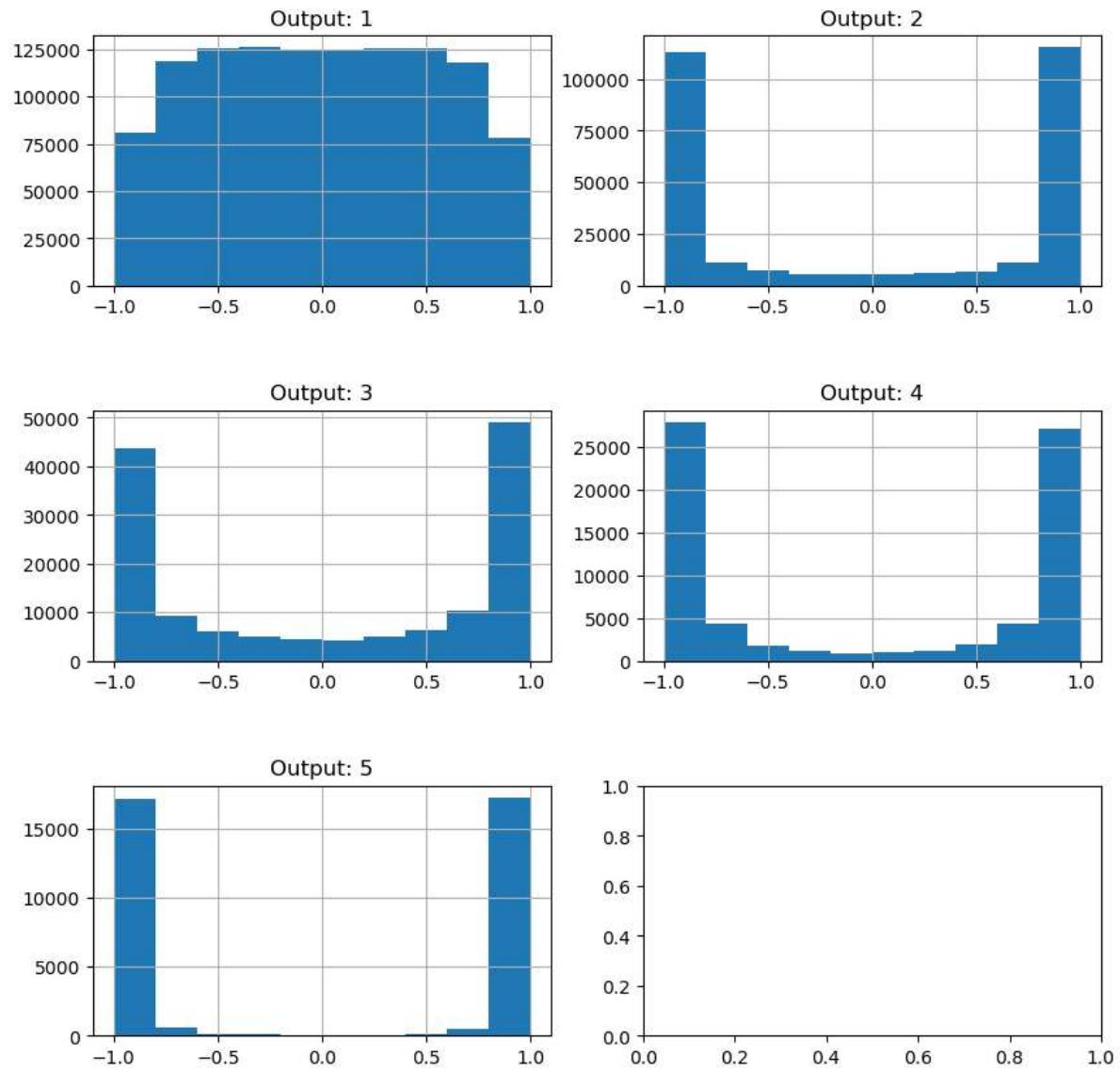
Accuracy: 97.634%

```
In [423]: model.show_weight_distribution()
```

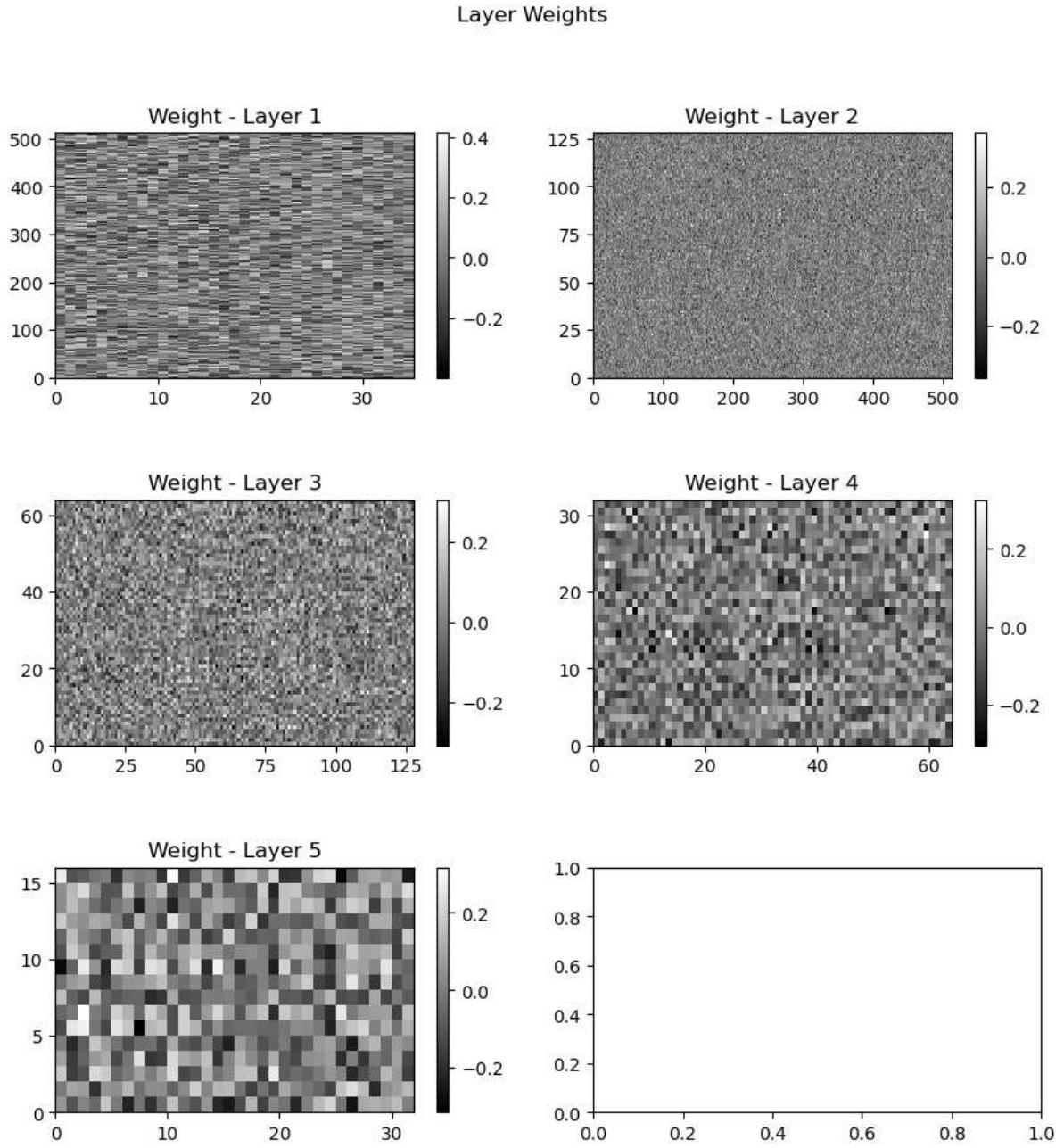


```
In [424]: model.show_output_distribution(torch.tensor(lda.transform(X_test.numpy()).real).float())
```

Activated Output distribution



In [425]: `model.show_weights()`



Deep Neural Networks also provide a classification accuracy over 97%. DNNs perform much better than RFC. However, if we reduce the dimensionality using Stepwise LDA, we obtain similar accuracy as the raw signal but we gain an increase in speed of classification.

Earlier we saw that the statistical significance of P300 amplitude was prominent in Oz, PO7, and PO8 region. Therefore, we will now use the data of these channels only to perform classification using Neural Networks and Stepwise LDA.

## Classification using Pz, PO7, and PO8

In [ ]: