BE 521: Homework 7 Questions

P300 Speller

Spring 2025

34 points

Due: March 20th, 2025

Objective: Spell letters using neurosignals.

AI Usage Notice

The use of artificial intelligence tools (e.g., large language models, code assistants) is permitted. However, students must explicitly state the specific ways AI was used in completing their work. Failure to disclose AI usage may result in an oral examination to assess understanding, at the discretion of Dr. Litt.

If AI was used in the completion of this assignment, please provide a statement below:

[Enter your statement here]

In [1]:
```
# !jupyter nbconvert --to html Prakriti_HW4.ipynb
```

In [2]:
```
#Set up the notebook environment
!pip install git+https://github.com/ieeg-portal/ieegpy.git # Install ieegpy toolbox directly from github
!pip install mne

from ieeg.auth import Session
import matplotlib.pyplot as plt
from scipy.io import loadmat
import mne
import pandas as pd
import numpy as np
standard_montage = mne.channels.make_standard_montage('standard_1005')
info = mne.create_info(['FC5', 'FC3', 'FC1', 'FCz', 'FC2', 'FC4', 'FC6', 'C5', 'C3', 'C1', 'Cz', 'C2', 'C4', 'C(
```

```
Collecting git+https://github.com/ieeg-portal/ieegpy.git
  Cloning https://github.com/ieeg-portal/ieegpy.git to /tmp/pip-req-build-n5k40umz
  Running command git clone --filter=blob:none --quiet https://github.com/ieeg-portal/ieegpy.git /tmp/pip-req-bu
ild-n5k40umz
  Resolved https://github.com/ieeg-portal/ieegpy.git to commit 080bfa42a8503380ef164b5e7b116613f75073bb
  Preparing metadata (setup.py) ... done
Requirement already satisfied: deprecation in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.1.0)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.32.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.0.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.2)
Requirement already satisfied: pennprov==2.2.4 in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.
4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2
.4->ieeg==1.6) (2025.1.31)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.11/dist-packages (from pennprov==2
.2.4->ieeg==1.6) (2.8.2)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->ieeg=
=1.6) (1.17.0)
Requirement already satisfied: urllib3>=1.23 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->i
eeg==1.6) (2.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from deprecation->ieeg==1.6
) (24.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6)
(2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6
) (2025.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from request
s->ieeg==1.6) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->ieeg==1.6
) (3.10)
Requirement already satisfied: mne in /usr/local/lib/python3.11/dist-packages (1.9.0)
Requirement already satisfied: decorator in /usr/local/lib/python3.11/dist-packages (from mne) (4.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from mne) (3.1.6)
Requirement already satisfied: lazy-loader>=0.3 in /usr/local/lib/python3.11/dist-packages (from mne) (0.4)
Requirement already satisfied: matplotlib>=3.6 in /usr/local/lib/python3.11/dist-packages (from mne) (3.10.0)
Requirement already satisfied: numpy<3,>=1.23 in /usr/local/lib/python3.11/dist-packages (from mne) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from mne) (24.2)
Requirement already satisfied: pooch>=1.5 in /usr/local/lib/python3.11/dist-packages (from mne) (1.8.2)
Requirement already satisfied: scipy>=1.9 in /usr/local/lib/python3.11/dist-packages (from mne) (1.14.1)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from mne) (4.67.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6
->mne) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6->mn
e) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.
6->mne) (4.56.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.
6->mne) (1.4.8)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6->mne)
(11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6
->mne) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib>
=3.6->mne) (2.8.2)
Requirement already satisfied: platformdirs>=2.5.0 in /usr/local/lib/python3.11/dist-packages (from pooch>=1.5->
mne) (4.3.6)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.11/dist-packages (from pooch>=1.5->mne
) (2.32.3)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->mne) (3.
0.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->m
atplotlib>=3.6->mne) (1.17.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from request
s>=2.19.0->pooch>=1.5->mne) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests>=2.19.0->p
ooch>=1.5->mne) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests>=2.1
9.0->pooch>=1.5->mne) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests>=2.1
9.0->pooch>=1.5->mne) (2025.1.31)
```

P300 Speller

In this homework, you will work with data from a P300-based brain computer interface called BCI2000 (Schalk et al. 2004) that allows people to spell words by focusing their attention on a particular letter displayed on the screen. In each trial the user focused on a letter, and when that letter's row or column is flashed, the user's brain elicits a P300 evoked response. By analyzing whether a P300 signal was produced across the flashes of many different rows or columns, the computer can determine the letter that the person is focusing on.

Data Organization

The data for this homework is stored in `I521_A0008_D001` on the IEEG Portal.The EEG in this dataset were recorded during 85 intended letter spellings. For each letter spelling, 12 row/columns were flashed 15 times in random order ($12 \times 15 = 180$ iterations). The EEG was recorded with a sampling rate of 240 Hz on a 64-channel scalp EEG.

The annotations for this dataset are organized in two layers as follows:

- `TargetLetter` annotation layer indicates the target letter (annotation.description) on which the user was focusing during the recorded EEG segment (annotation.start/annotation.stop). This layer is also provided as TargetLetterAnnots.mat.
- `Stim` annotation layer indicates the row/column that is being flashed (annotation.description) and whether the target letter is contained in that flash (annotation.type). The recorded EEG during that flash is (annotation.start/annotation.stop). Note that this annotation layer is provided as StimAnnots.mat. It is NOT on the portal.

We provide the annotation layers as CSV files and provide some code that loads, previews, and iterates through the rows of the files with the information above.

In [3]:
```
# load stim annotations file
stim_annots = pd.read_csv("StimAnnots.csv")
# preview dataframe with stim annotations
display(stim_annots.head())
# iterate through rows of dataframe
for index, row in stim_annots.iterrows():
  print(row.description, row.start, row.stop)
  if index == 1:
    break
```

	type	description	start	stop
0	0	12	1	1000000
1	0	11	1000001	2000000
2	0	3	2000001	3000000
3	0	10	3000001	4000000
4	0	9	4000001	5000000

```
12 1 1000000
11 1000001 2000000
```

1. Exploring the data

In this section you will explore some basic properties of the data in `I521_A0008_D001`. We suggest using the `load_full_channels` function to load in all of the channels and saving the data (will take ~3 minutes), since loading each channel independently will take a long time.

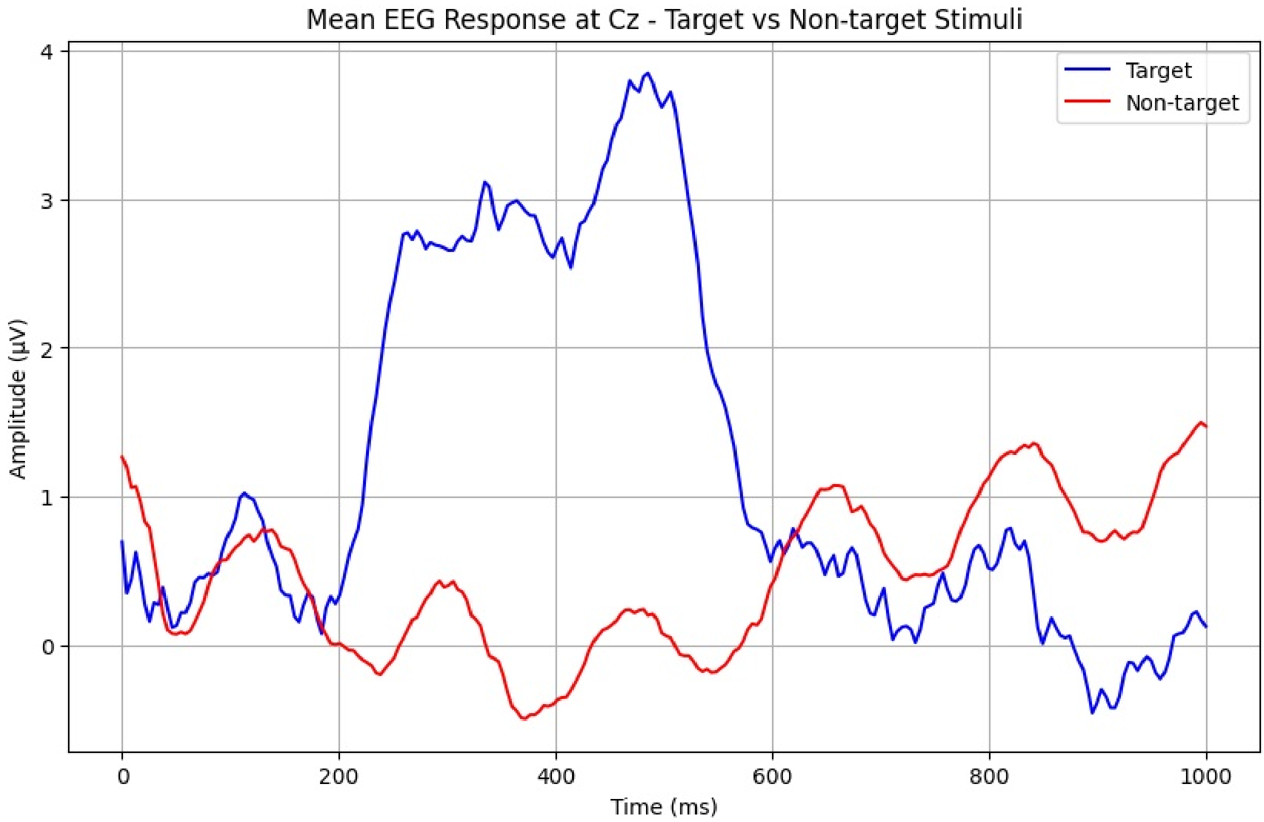
In [4]:
```
def load_full_channels(dataset, duration_secs, sampling_rate, chn_idx):
  """
  Loads the entire channel from IEEG.org
  Input:
    dataset: the IEEG dataset object
    duration_secs: the duration of the channel, in seconds
    sampling_rate: the sampling rate of the channel, in Hz
    chn_idx: the indicies of the m channels you want to load, as an array-like object
  Returns:
    [n, m] numpy array of the channels' values.
  """
  #stores the segments of the channel's data
  chn_segments = []

  #how many segments do we expect?
  num_segments = int(np.ceil(duration_secs * sampling_rate / 6e4))

  #segment start times and the step
  seg_start, step = np.linspace(1, duration_secs*1e6, num_segments, endpoint=False, retstep=True)

  #get the segments
  for start in seg_start:
    chn_segments.append(dataset.get_data(start, step, chn_idx))

  #concatenate the segments vertically
  return np.vstack(chn_segments)
```

1

For channel 10 (Cz), plot the mean EEG for the target and non-target stimuli separately, (i.e. rows/columns including and not-including the desired character, respectively), on the same set of axes. Label your x-axis in milliseconds. (3 pts)

In [5]:

```
with open('/content/pra_ieeglogin(9).bin', 'r') as f:
    session = Session('prasadpr', f.read())
dataset = session.open_dataset('I521_A0008_D001')

# Get dataset duration and sampling rate
duration_secs = dataset.get_time_series_details(dataset.ch_labels[0]).duration / 1e6
# duration_secs = 20  # Convert to seconds
sampling_rate = 240  # Given in the problem

# Load channel 10 (Cz) - index 10 since list is 0-based
eeg_data = load_full_channels(dataset, duration_secs, sampling_rate, [10])

# Load annotation files (assuming they're in your working directory)
target_annots = pd.read_csv('/content/TargetLetterAnnots.csv')
stim_annots = pd.read_csv('/content/StimAnnots.csv')


# Process data for plotting
tmin, tmax = 0, 1  # 800ms window for P300
samples_per_epoch = int((tmax - tmin) * sampling_rate)

target_epochs, nontarget_epochs = [], []

for _, stim in stim_annots.iterrows():
    start_sample = int(stim['start'] * sampling_rate / 1e6)  # Convert microseconds to samples
    if start_sample + samples_per_epoch < eeg_data.shape[0]:
        epoch = eeg_data[start_sample:start_sample + samples_per_epoch, 0]
        if len(epoch) == samples_per_epoch:
            if stim['type'] == 1:  # Target stimulus
                target_epochs.append(epoch)
            elif stim['type'] == 0:  # Non-target stimulus
                nontarget_epochs.append(epoch)

# Calculate means
target_mean = np.mean(np.array(target_epochs), axis=0)
nontarget_mean = np.mean(np.array(nontarget_epochs), axis=0)

# Plot
time_ms = np.linspace(tmin * 1000, tmax * 1000, samples_per_epoch)
plt.figure(figsize=(10, 6))
plt.plot(time_ms, target_mean, label='Target', color='blue')
plt.plot(time_ms, nontarget_mean, label='Non-target', color='red')
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude (μV)')
plt.title('Mean EEG Response at Cz - Target vs Non-target Stimuli')
plt.legend()
plt.grid(True)
plt.show()
```

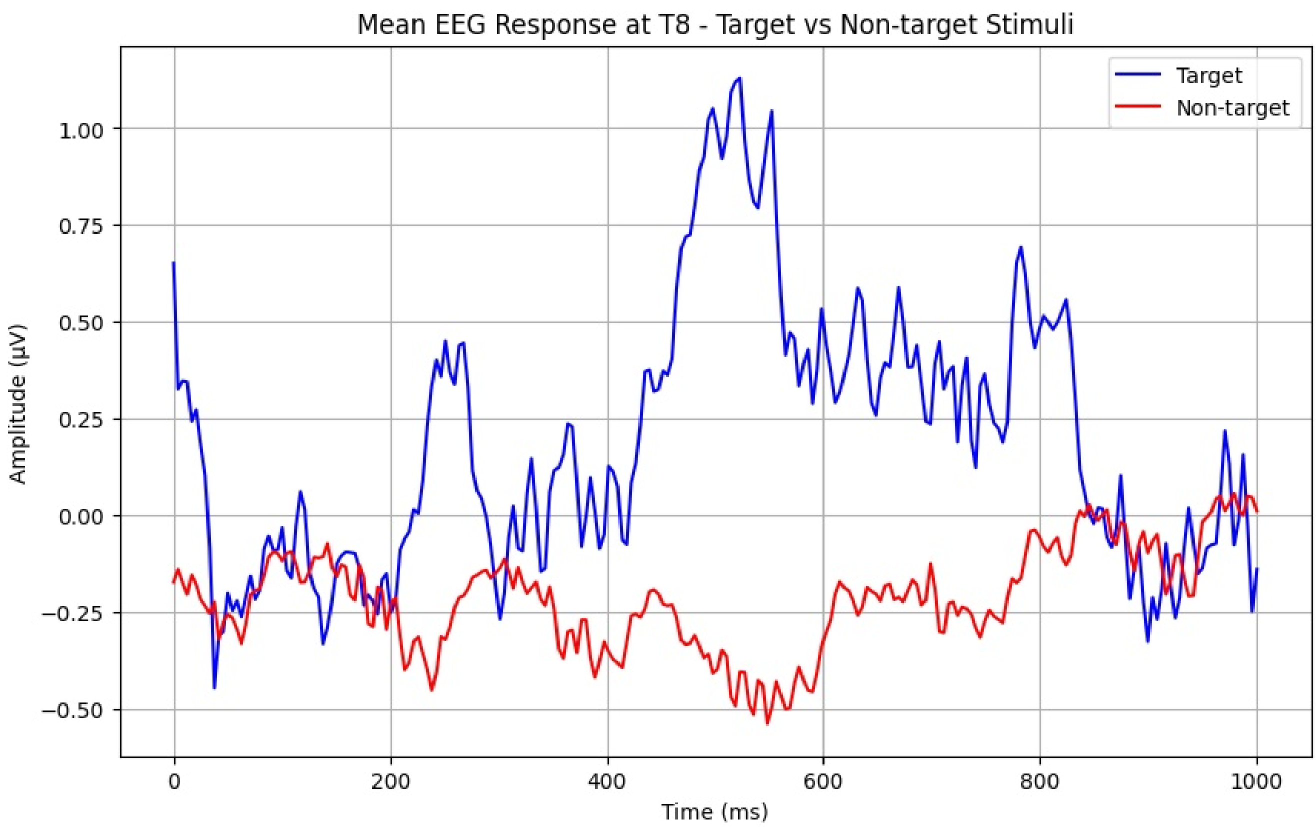


2

Repeat the previous questions for channel 41 (T8). (1 pts)

In [6]:

```python
# Load channel 41 (T8) - index 41 since list is 0-based
eeg_data = load_full_channels(dataset, duration_secs, sampling_rate, [41])

# Load annotation files
target_annots = pd.read_csv('/content/TargetLetterAnnots.csv')
stim_annots = pd.read_csv('/content/StimAnnots.csv')

# Process data for plotting
tmin, tmax = 0, 1  # 800ms window for P300
samples_per_epoch = int((tmax - tmin) * sampling_rate)

target_epochs, nontarget_epochs = [], []

for _, stim in stim_annots.iterrows():
    start_sample = int(stim['start'] * sampling_rate / 1e6)  # Convert microseconds to samples
    if start_sample + samples_per_epoch < eeg_data.shape[0]:
        epoch = eeg_data[start_sample:start_sample + samples_per_epoch, 0]
        if len(epoch) == samples_per_epoch:
            if stim['type'] == 1:  # Target stimulus
                target_epochs.append(epoch)
            elif stim['type'] == 0:  # Non-target stimulus
                nontarget_epochs.append(epoch)

# Calculate means
target_mean = np.mean(np.array(target_epochs), axis=0)
nontarget_mean = np.mean(np.array(nontarget_epochs), axis=0)

# Plot
time_ms = np.linspace(tmin * 1000, tmax * 1000, samples_per_epoch)
plt.figure(figsize=(10, 6))
plt.plot(time_ms, target_mean, label='Target', color='blue')
plt.plot(time_ms, nontarget_mean, label='Non-target', color='red')
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude (µV)')
plt.title('Mean EEG Response at T8 - Target vs Non-target Stimuli')
plt.legend()
plt.grid(True)
plt.show()

# Close dataset
```



3

Which of the two previous channels looks best for distinguishing between target and non-target stimuli? Which time points look best?

Explain in a few sentences. (2 pts)

Based on the plots:

Channel Comparison:

```
Cz: The mean EEG response at Cz shows a clearer distinction between target and non-target
stimuli, particularly around the P300 window (250-450 ms). The amplitude difference between
target and non-target responses is more pronounced at Cz compared to T8.

T8: The mean EEG response at T8 shows less distinction between target and non-target stimuli.
The amplitude differences are smaller and less consistent over time.
```

Best Time Points:

```
The P300 window (250-450 ms) is the most effective for distinguishing between target and non-
target stimuli, especially at the Cz channel. This is where the amplitude difference between
target and non-target responses is most significant.
```

Conclusion:

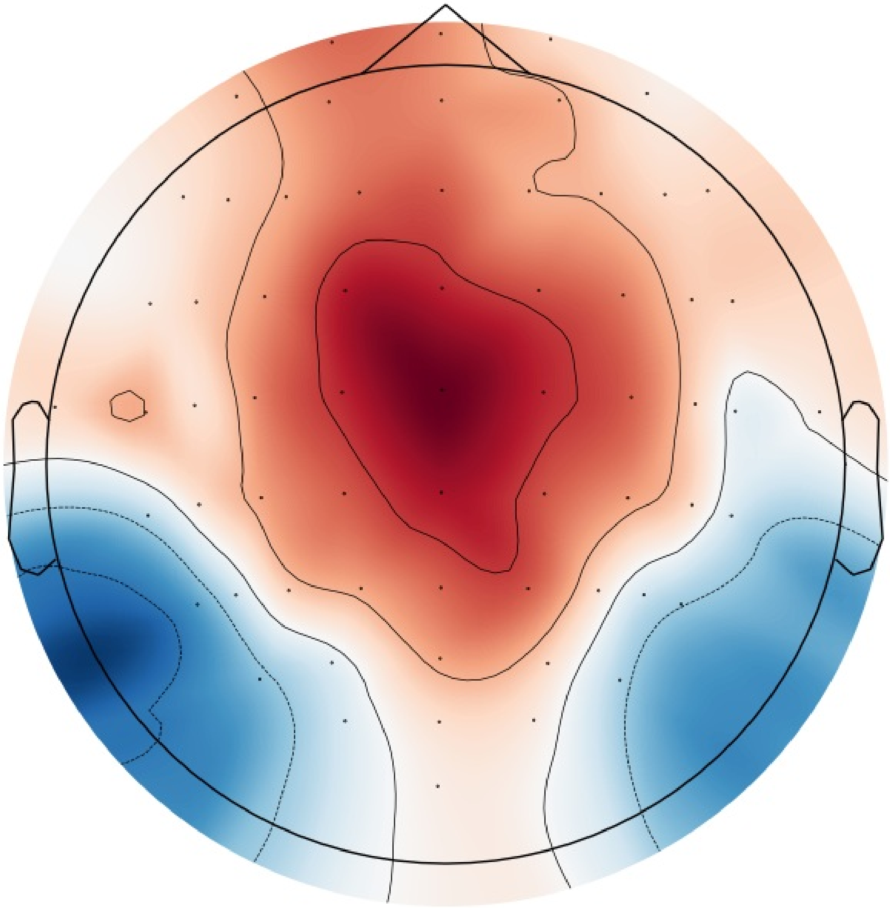
The Cz channel is better for distinguishing between target and non-target stimuli, particularly during the P300 window (250-450 ms). This is because the amplitude difference between target and non-target responses is more pronounced at Cz, making it easier to detect the P300 response.

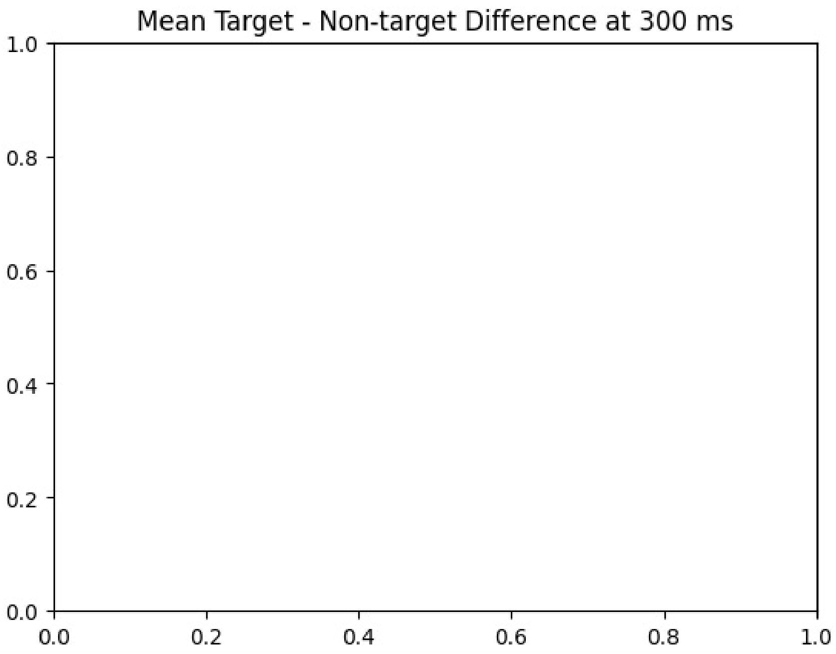
4

Compute the mean difference between the target and non-target stimuli for each channel at timepoint 300 ms averaged across all row/column flashes. Visualize these values using `mne.viz.plot_topomap` and the example code given below by replacing the argument `your_data_here`. (3 pts)

In [7]:

```python
# Load data
with open('/content/pra_ieeglogin(9).bin', 'r') as f:
    session = Session('prasadpr', f.read())
dataset = session.open_dataset('I521_A0008_D001')

duration_secs = dataset.get_time_series_details(dataset.ch_labels[0]).duration / 1e6
sampling_rate = 240
eeg_data = load_full_channels(dataset, duration_secs, sampling_rate, list(range(64)))  # Load all 64 channels

# Load annotation files
stim_annots = pd.read_csv('/content/StimAnnots.csv')

# Process data
tmin, tmax = 0, 1 # 1 sec
samples_per_epoch = int((tmax - tmin) * sampling_rate) # 240
sample_300ms = int(0.3 * sampling_rate)  # 300 ms sample index (72 at 240 Hz)

target_epochs, nontarget_epochs = [[] for _ in range(64)], [[] for _ in range(64)]

for _, stim in stim_annots.iterrows():
    start_sample = int(stim['start'] * sampling_rate / 1e6)
    if start_sample + samples_per_epoch < eeg_data.shape[0]:
        epoch = eeg_data[start_sample:start_sample + samples_per_epoch, :]
        if epoch.shape[0] == samples_per_epoch:
            for ch in range(64):
                if stim['type'] == 1:
                    target_epochs[ch].append(epoch[:, ch])
                elif stim['type'] == 0:
                    nontarget_epochs[ch].append(epoch[:, ch])

# Compute mean difference at 300 ms
mean_diff = []
for ch in range(64):
    if target_epochs[ch] and nontarget_epochs[ch]:
        target_mean = np.mean(np.array(target_epochs[ch])[:, sample_300ms])
        nontarget_mean = np.mean(np.array(nontarget_epochs[ch])[:, sample_300ms])
        mean_diff.append(target_mean - nontarget_mean)
    else:
        mean_diff.append(0)

# Plot topomap
fig, ax = plt.subplots(figsize=(8, 8))
mne.viz.plot_topomap(np.array(mean_diff), info, axes=ax, show=True)
plt.title('Mean Target - Non-target Difference at 300 ms')
plt.show()

############ the plot doesn't fit within the axis, not sure why #################
```





5

How do the red and blue parts of this plot correspond to the plots from above? (2 pts)

The red and blue parts of the topographic plot (topomap) correspond to the mean difference in EEG amplitude between target and non-target stimuli at 300 ms across different electrode locations.

Red Areas:

```
  Represent positive differences in amplitude, where the mean EEG response to target stimuli
is higher than the response to non-target stimuli.

  These areas indicate electrodes where the P300 response (target-related activity) is
strongest.
```

Blue Areas:

```
  Represent negative differences in amplitude, where the mean EEG response to target stimuli
is lower than the response to non-target stimuli.

  These areas indicate electrodes where non-target responses dominate or where the P300
response is weaker.
```

Mean EEG Response Plots:

```
  The previous plots (e.g., "Mean EEG Response at Cz - Target vs Non-target Stimuli") show the
time course of the EEG response for target and non-target stimuli at specific electrodes
(e.g., Cz or T8).

  The amplitude difference at 300 ms (highlighted in the topomap) corresponds to the vertical
difference between the target and non-target curves at the 300 ms time point in the time-
course plots.
```

Topomap:

```
  The topomap summarizes the spatial distribution of this amplitude difference across all
electrodes at 300 ms.

  Electrodes with red areas in the topomap correspond to locations where the target response
is stronger than the non-target response at 300 ms (as seen in the time-course plots).

  Electrodes with blue areas correspond to locations where the non-target response is stronger
or where the target response is weaker.
```

2. Using individual P300s in prediction

1

Explain a potential advantage to using just one channel other than the obvious speed of calculation advantage. Explain one disadvantage. (3 pts)

Advantage of Using Just One Channel

```
Reduced Complexity and Noise:

    Advantage: Using a single channel (e.g., Cz) can reduce the complexity of the analysis and
minimize the impact of noise from other channels. EEG data is often noisy, and focusing on a
single channel that is known to capture the P300 response effectively (e.g., Cz, which is
centrally located and typically shows a strong P300 signal) can improve the signal-to-noise
ratio (SNR). This can lead to more reliable detection of the P300 response.
```

Disadvantage of Using Just One Channel

```
Loss of Spatial Information:

    Disadvantage: Using only one channel ignores the spatial distribution of the P300 response
across the scalp. The P300 response is not always strongest at a single electrode, and its
amplitude and latency can vary across different brain regions. By focusing on just one
channel, you might miss important information from other electrodes that could improve the
accuracy of target detection. For example, if the P300 response is weaker at Cz but stronger
at Pz, using only Cz could lead to missed detections or false negatives.
```

2

One simple way of identifying a P300 in a single trial (which we'll call the *p300 score*) is to take the mean EEG from 250 to 450 ms and then subtract from it the mean EEG from 600 to 800 ms. What is the *p300 score* for epoch (letter) 9, iteration 10 at electrode Cz? (3 pts)

In [8]:

```
with open('/content/pra_ieeglogin(9).bin', 'r') as f:
    session = Session('prasadpr', f.read())
dataset = session.open_dataset('I521_A0008_D001')
duration_secs = dataset.get_time_series_details(dataset.ch_labels[0]).duration / 1e6
sampling_rate = 240

# Load the data for electrode Cz (channel 10)
eeg_data = load_full_channels(dataset, duration_secs, sampling_rate, [10])

# Define time windows in milliseconds and convert to sample indices
window_p300 = (250, 450)  # P300 window
window_baseline = (600, 800)  # Baseline window

# Convert milliseconds to sample indices
```

```python
p300_start = int(window_p300[0] * sampling_rate / 1000)
p300_end = int(window_p300[1] * sampling_rate / 1000)
baseline_start = int(window_baseline[0] * sampling_rate / 1000)
baseline_end = int(window_baseline[1] * sampling_rate / 1000)

# Reconstruct epochs and iterations
# Each epoch has 180 iterations (flashes)
num_iterations_per_epoch = 180

# Calculate the row index for epoch 9, iteration 10
# Rows are ordered sequentially: epoch 1 (iterations 1-15), epoch 2 (iterations 1-15), etc.
row_index = (9 - 1) * num_iterations_per_epoch + (10 - 1)  # Subtract 1 for 0-based indexing

# Check if the row index is within the bounds of the stim_annots DataFrame
if row_index >= len(stim_annots):
    raise ValueError("Epoch 9, iteration 10 does not exist in the annotations.")

# Get the start time of the specific epoch and iteration
start_time = stim_annots.iloc[row_index]['start']  # Start time in microseconds
start_sample = int(start_time * sampling_rate / 1e6)  # Convert to sample index

# Extract the EEG data for the specific epoch and iteration
epoch_data = eeg_data[start_sample:start_sample + samples_per_epoch, 0]

# Ensure the epoch data is the correct length
if len(epoch_data) != samples_per_epoch:
    raise ValueError("Epoch data is not the correct length.")

# Extract the P300 and baseline windows
p300_window = epoch_data[p300_start:p300_end]
baseline_window = epoch_data[baseline_start:baseline_end]

# Compute the P300 score
p300_score = np.mean(p300_window) - np.mean(baseline_window)

print(f"P300 score for epoch 9, iteration 10 at electrode Cz: {p300_score:.10f}")
```

```
P300 score for epoch 9, iteration 10 at electrode Cz: -4.5182916667
```

3

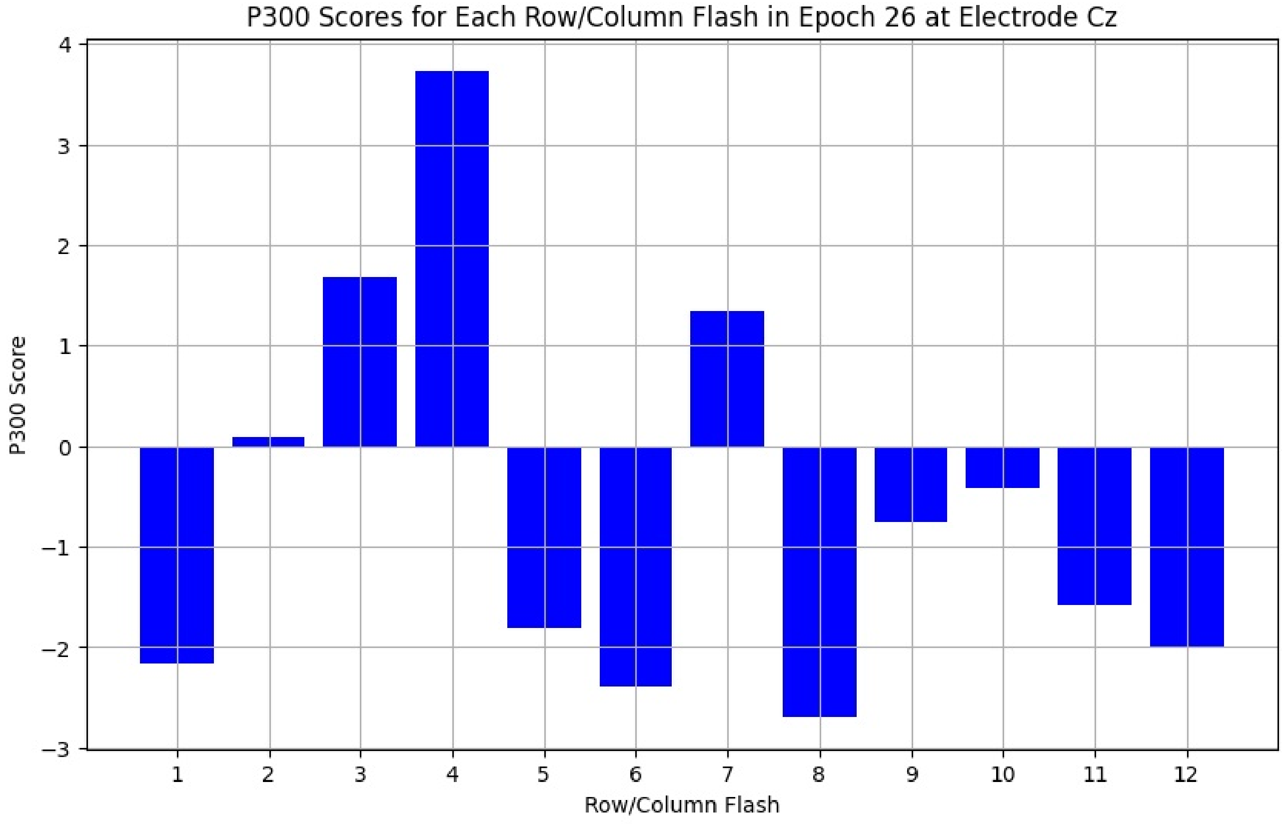
Plot the p300 scores for each row/column in epoch 26 at electrode Cz. (3 pts)

In [9]:

```python
# Load the data for electrode Cz (channel 10)
eeg_data = load_full_channels(dataset, duration_secs, sampling_rate, [10])  # Shape: (n_samples, 1)
print(f"EEG data shape: {eeg_data.shape}")

samples_per_segment = int(800 * sampling_rate / 1000)  # 800 ms = 192 samples at 240 Hz
print(f"Samples per segment: {samples_per_segment}")

# Load annotations
stim_annots = pd.read_csv("StimAnnots.csv")
target_annots = pd.read_csv("TargetLetterAnnots.csv")

# Define time windows in milliseconds and convert to sample indices
window_p300 = (250, 450)  # P300 window in ms
window_baseline = (600, 800)  # Baseline window in ms
p300_start = int(window_p300[0] * sampling_rate / 1000)  # 60 samples
p300_end = int(window_p300[1] * sampling_rate / 1000)    # 108 samples
baseline_start = int(window_baseline[0] * sampling_rate / 1000)  # 144 samples
baseline_end = int(window_baseline[1] * sampling_rate / 1000)    # 192 samples

# Each epoch has 180 iterations (12 rows/columns × 15 flashes)
num_iterations_per_epoch = 180

# Identify the rows corresponding to epoch 26
epoch_idx = 26 - 1  # 0-based, epoch 26 is 25
epoch_26_start = epoch_idx * num_iterations_per_epoch  # 25 * 180 = 4500
epoch_26_end = epoch_26_start + num_iterations_per_epoch  # 4680
print(f"Epoch 26 row range: {epoch_26_start} to {epoch_26_end - 1}")

# Verify bounds
if epoch_26_end > len(stim_annots):
    raise ValueError(f"Epoch 26 exceeds stim_annots length: {len(stim_annots)}")

# Filter the stim_annots DataFrame for epoch 26
epoch_26_flashes = stim_annots.iloc[epoch_26_start:epoch_26_end]

# Initialize a list to store P300 scores and labels
p300_scores = []
stim_labels = []

# Loop through each flash in epoch 26
```

```python
for _, flash in epoch_26_flashes.iterrows():
    # Get the start time of the flash
    start_time = flash['start']  # Start time in microseconds
    start_sample = int(start_time * sampling_rate / 1e6)  # Convert to sample index

    # Extract the EEG data for the specific flash
    if start_sample + samples_per_segment <= eeg_data.shape[0]:
        epoch_data = eeg_data[start_sample:start_sample + samples_per_segment, 0]
    else:
        print(f"Warning: Flash at sample {start_sample} exceeds data length")
        epoch_data = np.array([np.nan] * samples_per_segment)

    # Ensure the epoch data is the correct length
    if len(epoch_data) != samples_per_segment:
        raise ValueError(f"Epoch data length {len(epoch_data)} does not match {samples_per_segment}")

    # Extract the P300 and baseline windows
    p300_window = epoch_data[p300_start:p300_end]
    baseline_window = epoch_data[baseline_start:baseline_end]

    # Compute the P300 score
    p300_score = np.mean(p300_window) - np.mean(baseline_window)
    p300_scores.append(p300_score)
    stim_labels.append(int(flash['description']))  # Row/column number (1-12)

# Compute average P300 score per row/column
avg_scores = []
for stim in range(1, 13):  # Rows/columns 1-12
    stim_scores = [p300_scores[i] for i, label in enumerate(stim_labels) if label == stim]
    avg_score = np.mean(stim_scores)
    avg_scores.append(avg_score)
    print(f"Average P300 score for stim {stim}: {avg_score:.4f}")

# Plot the average P300 scores for each row/column flash in epoch 26
plt.figure(figsize=(10, 6))
plt.bar(range(1, 13), avg_scores, color='blue')
plt.xlabel('Row/Column Flash')
plt.ylabel('P300 Score')
plt.title('P300 Scores for Each Row/Column Flash in Epoch 26 at Electrode Cz')
plt.xticks(range(1, 13))
plt.grid(True)
plt.show()
```

```
EEG data shape: (3672012, 1)
Samples per segment: 192
Epoch 26 row range: 4500 to 4679
Average P300 score for stim 1: -2.1573
Average P300 score for stim 2: 0.0867
Average P300 score for stim 3: 1.6858
Average P300 score for stim 4: 3.7326
Average P300 score for stim 5: -1.8033
Average P300 score for stim 6: -2.3840
Average P300 score for stim 7: 1.3389
Average P300 score for stim 8: -2.6919
Average P300 score for stim 9: -0.7540
Average P300 score for stim 10: -0.4151
Average P300 score for stim 11: -1.5866
Average P300 score for stim 12: -2.0035
```



4

Based on your previous answer for epoch 26, what letter do you predict the person saw? Is this prediction correct? Note: Use the HW7_P300_speller.png file. The numbers correspond to the stim (2 pts)

I predict D.

Here is my explanation: Looking at the first six numbers of the X axis which correspond to the columns in the image, the highest value is at 4. Therefore looking at the image provided the letter/number s hould be at column 4.

Now we look at the next 6 numbers from 7 - 12 which correspond to rows in the image. The highest value there is 7 which corresponds to row 1 in the image.

As D lies on row 1 and column 4. It is D. This is also validated by my algorithm below where it predicts D at epoch 26.

5

Using this p300 score, predict (and print out) the letter viewed at every epoch. What was you prediction accuracy? (2 pts)

In [10]:
```
import numpy as np
```

```python
import pandas as pd
import matplotlib.pyplot as plt

# Assuming eeg_data, stim_annots, and target_annots are already loaded
# Define the letter grid
letter_grid = [
    ['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', '_']
]

# Parameters
num_epochs = len(target_annots)
num_iterations_per_epoch = 180
samples_per_segment = int(800 * sampling_rate / 1000)  # 800 ms = 192 samples at 240 Hz

# Define time windows in milliseconds and convert to sample indices
window_p300 = (250, 450)  # P300 window in ms
window_baseline = (600, 800)  # Baseline window in ms
p300_start = int(window_p300[0] * sampling_rate / 1000)  # 60 samples
p300_end = int(window_p300[1] * sampling_rate / 1000)    # 108 samples
baseline_start = int(window_baseline[0] * sampling_rate / 1000)  # 144 samples
baseline_end = int(window_baseline[1] * sampling_rate / 1000)    # 192 samples

# Initialize list to store predictions
predictions = []

# Loop through each epoch
for epoch in range(num_epochs):
    # Identify the rows corresponding to the current epoch
    epoch_start = epoch * num_iterations_per_epoch
    epoch_end = epoch_start + num_iterations_per_epoch

    # Verify bounds
    if epoch_end > len(stim_annots):
        raise ValueError(f"Epoch {epoch + 1} exceeds stim_annots length: {len(stim_annots)}")

    # Filter the stim_annots DataFrame for the current epoch
    epoch_flashes = stim_annots.iloc[epoch_start:epoch_end]

    # Initialize lists to store P300 scores and labels
    p300_scores = []
    stim_labels = []

    # Loop through each flash in the epoch
    for _, flash in epoch_flashes.iterrows():
        # Get the start time of the flash
        start_time = flash['start']  # Start time in microseconds
        start_sample = int(start_time * sampling_rate / 1e6)  # Convert to sample index

        # Extract the EEG data for the specific flash
        if start_sample + samples_per_segment <= eeg_data.shape[0]:
            epoch_data = eeg_data[start_sample:start_sample + samples_per_segment, 0]
        else:
            print(f"Warning: Flash at sample {start_sample} exceeds data length")
            epoch_data = np.array([np.nan] * samples_per_segment)

        # Ensure the epoch data is the correct length
        if len(epoch_data) != samples_per_segment:
            raise ValueError(f"Epoch data length {len(epoch_data)} does not match {samples_per_segment}")

        # Extract the P300 and baseline windows
        p300_window = epoch_data[p300_start:p300_end]
        baseline_window = epoch_data[baseline_start:baseline_end]

        # Compute the P300 score
        p300_score = np.mean(p300_window) - np.mean(baseline_window)
        p300_scores.append(p300_score)
        stim_labels.append(int(flash['description']))  # Row/column number (1-12)

    # Compute average P300 score per row/column
    avg_scores = {}
    for stim in range(1, 13):  # Rows/columns 1-12
        stim_scores = [p300_scores[i] for i, label in enumerate(stim_labels) if label == stim]
        avg_scores[stim] = np.mean(stim_scores)

    # Find the column (1-6) and row (7-12) with the highest average P300 score
    column_scores = {stim: score for stim, score in avg_scores.items() if stim <= 6}
    row_scores = {stim: score for stim, score in avg_scores.items() if stim >= 7}
```

```python
    target_column = max(column_scores, key=column_scores.get)  # Column with highest score (1-6)
    target_row = max(row_scores, key=row_scores.get)           # Row with highest score (7-12)

    # Map to letter (0-based indices for the grid)
    row_idx = target_row - 7  # 7-12 to 0-5
    col_idx = target_column - 1  # 1-6 to 0-5
    predicted_letter = letter_grid[row_idx][col_idx]

    # Get actual letter
    actual_letter = target_annots['description'].iloc[epoch]

    # Print prediction
    print(f"Epoch {epoch + 1}: Predicted = {predicted_letter}, Actual = {actual_letter}, "
          f"Correct = {predicted_letter == actual_letter}")

    predictions.append(predicted_letter)

# Compute accuracy
actual_letters = target_annots['description'].tolist()
accuracy = np.mean([pred == actual for pred, actual in zip(predictions, actual_letters)])
print(f"\nPrediction Accuracy: {accuracy:.2%}")
```

```
Epoch 1: Predicted = 4, Actual = E, Correct = False
Epoch 2: Predicted = A, Actual = A, Correct = True
Epoch 3: Predicted = 8, Actual = E, Correct = False
Epoch 4: Predicted = V, Actual = V, Correct = True
Epoch 5: Predicted = E, Actual = Q, Correct = False
Epoch 6: Predicted = U, Actual = T, Correct = False
Epoch 7: Predicted = F, Actual = D, Correct = False
Epoch 8: Predicted = O, Actual = O, Correct = True
Epoch 9: Predicted = 8, Actual = J, Correct = False
Epoch 10: Predicted = I, Actual = G, Correct = False
Epoch 11: Predicted = O, Actual = 8, Correct = False
Epoch 12: Predicted = W, Actual = R, Correct = False
Epoch 13: Predicted = B, Actual = B, Correct = True
Epoch 14: Predicted = O, Actual = R, Correct = False
Epoch 15: Predicted = J, Actual = G, Correct = False
Epoch 16: Predicted = M, Actual = O, Correct = False
Epoch 17: Predicted = F, Actual = N, Correct = False
Epoch 18: Predicted = C, Actual = C, Correct = True
Epoch 19: Predicted = F, Actual = E, Correct = False
Epoch 20: Predicted = D, Actual = D, Correct = True
Epoch 21: Predicted = 9, Actual = H, Correct = False
Epoch 22: Predicted = C, Actual = C, Correct = True
Epoch 23: Predicted = T, Actual = T, Correct = True
Epoch 24: Predicted = X, Actual = U, Correct = False
Epoch 25: Predicted = 7, Actual = I, Correct = False
Epoch 26: Predicted = D, Actual = D, Correct = True
Epoch 27: Predicted = B, Actual = B, Correct = True
Epoch 28: Predicted = 6, Actual = P, Correct = False
Epoch 29: Predicted = U, Actual = U, Correct = True
Epoch 30: Predicted = 1, Actual = H, Correct = False
Epoch 31: Predicted = A, Actual = M, Correct = False
Epoch 32: Predicted = W, Actual = E, Correct = False
Epoch 33: Predicted = R, Actual = M, Correct = False
Epoch 34: Predicted = 5, Actual = 6, Correct = False
Epoch 35: Predicted = I, Actual = O, Correct = False
Epoch 36: Predicted = M, Actual = U, Correct = False
Epoch 37: Predicted = K, Actual = X, Correct = False
Epoch 38: Predicted = O, Actual = O, Correct = True
Epoch 39: Predicted = C, Actual = C, Correct = True
Epoch 40: Predicted = Q, Actual = F, Correct = False
Epoch 41: Predicted = O, Actual = O, Correct = True
Epoch 42: Predicted = O, Actual = U, Correct = False
Epoch 43: Predicted = C, Actual = K, Correct = False
Epoch 44: Predicted = X, Actual = W, Correct = False
Epoch 45: Predicted = E, Actual = A, Correct = False
Epoch 46: Predicted = _, Actual = 4, Correct = False
Epoch 47: Predicted = T, Actual = V, Correct = False
Epoch 48: Predicted = J, Actual = J, Correct = True
Epoch 49: Predicted = K, Actual = E, Correct = False
Epoch 50: Predicted = 4, Actual = F, Correct = False
Epoch 51: Predicted = R, Actual = R, Correct = True
Epoch 52: Predicted = Z, Actual = Z, Correct = True
Epoch 53: Predicted = 3, Actual = R, Correct = False
Epoch 54: Predicted = I, Actual = O, Correct = False
Epoch 55: Predicted = L, Actual = L, Correct = True
Epoch 56: Predicted = H, Actual = H, Correct = True
Epoch 57: Predicted = Y, Actual = Y, Correct = True
Epoch 58: Predicted = 6, Actual = N, Correct = False
Epoch 59: Predicted = O, Actual = Q, Correct = False
Epoch 60: Predicted = _, Actual = D, Correct = False
Epoch 61: Predicted = E, Actual = W, Correct = False
Epoch 62: Predicted = X, Actual = _, Correct = False
```

```
Epoch 63: Predicted = E, Actual = E, Correct = True
Epoch 64: Predicted = E, Actual = K, Correct = False
Epoch 65: Predicted = V, Actual = T, Correct = False
Epoch 66: Predicted = L, Actual = L, Correct = True
Epoch 67: Predicted = N, Actual = B, Correct = False
Epoch 68: Predicted = X, Actual = W, Correct = False
Epoch 69: Predicted = T, Actual = X, Correct = False
Epoch 70: Predicted = K, Actual = E, Correct = False
Epoch 71: Predicted = K, Actual = P, Correct = False
Epoch 72: Predicted = P, Actual = O, Correct = False
Epoch 73: Predicted = I, Actual = U, Correct = False
Epoch 74: Predicted = H, Actual = I, Correct = False
Epoch 75: Predicted = N, Actual = K, Correct = False
Epoch 76: Predicted = G, Actual = Z, Correct = False
Epoch 77: Predicted = F, Actual = E, Correct = False
Epoch 78: Predicted = L, Actual = R, Correct = False
Epoch 79: Predicted = X, Actual = Y, Correct = False
Epoch 80: Predicted = L, Actual = O, Correct = False
Epoch 81: Predicted = A, Actual = O, Correct = False
Epoch 82: Predicted = T, Actual = T, Correct = True
Epoch 83: Predicted = F, Actual = H, Correct = False
Epoch 84: Predicted = E, Actual = Q, Correct = False
Epoch 85: Predicted = L, Actual = I, Correct = False

Prediction Accuracy: 27.06%
```

3. Automate the learning

In Section 2, you used a fairly manual method for predicting the letter. Here, you will have free reign to use put any and all learning techniques to try to improve your testing accuracy.

1

Play around with some ideas for improving/generalizing the prediction paradigm used in the letter prediction. Use the first 50 letter epochs as the training set and the later 35 for validation. Here, you are welcome to hard-code in whatever parameters you like/determine to be optimal. What is the optimal validation accuracy you get? Note: don't worry too much about accuracy, we are more interested in your thought process. (4 pts)

In [11]:

```
import numpy as np
import pandas as pd
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Assuming eeg_data, stim_annots, and target_annots are already loaded
# Define the letter grid
letter_grid = [
    ['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', '_']
]

# Parameters
num_epochs = len(target_annots)
num_iterations_per_epoch = 180
samples_per_segment = int(800 * sampling_rate / 1000)  # 800 ms = 192 samples at 240 Hz

# Define time windows in milliseconds and convert to sample indices
window_p300 = (250, 450)  # P300 window in ms
window_baseline = (600, 800)  # Baseline window in ms
p300_start = int(window_p300[0] * sampling_rate / 1000)  # 60 samples
p300_end = int(window_p300[1] * sampling_rate / 1000)    # 108 samples
baseline_start = int(window_baseline[0] * sampling_rate / 1000)  # 144 samples
baseline_end = int(window_baseline[1] * sampling_rate / 1000)    # 192 samples

# Initialize lists to store features and labels
features = []
labels = []

# Loop through each epoch
for epoch in range(num_epochs):
    # Identify the rows corresponding to the current epoch
    epoch_start = epoch * num_iterations_per_epoch
    epoch_end = epoch_start + num_iterations_per_epoch
```

```
    # Verify bounds
    if epoch_end > len(stim_annots):
        raise ValueError(f"Epoch {epoch + 1} exceeds stim_annots length: {len(stim_annots)}")

    # Filter the stim_annots DataFrame for the current epoch
    epoch_flashes = stim_annots.iloc[epoch_start:epoch_end]

    # Loop through each flash in the epoch
    for _, flash in epoch_flashes.iterrows():
        # Get the start time of the flash
        start_time = flash['start']  # Start time in microseconds
        start_sample = int(start_time * sampling_rate / 1e6)  # Convert to sample index

        # Extract the EEG data for the specific flash
        if start_sample + samples_per_segment <= eeg_data.shape[0]:
            epoch_data = eeg_data[start_sample:start_sample + samples_per_segment, 0]
        else:
            print(f"Warning: Flash at sample {start_sample} exceeds data length")
            epoch_data = np.array([np.nan] * samples_per_segment)

        # Ensure the epoch data is the correct length
        if len(epoch_data) != samples_per_segment:
            raise ValueError(f"Epoch data length {len(epoch_data)} does not match {samples_per_segment}")

        # Extract the P300 and baseline windows
        p300_window = epoch_data[p300_start:p300_end]
        baseline_window = epoch_data[baseline_start:baseline_end]

        # Compute the P300 score
        p300_score = np.mean(p300_window) - np.mean(baseline_window)

        # Append features and labels
        features.append([p300_score, np.mean(epoch_data), np.std(epoch_data)])  # Example features
        labels.append(int(flash['description']))  # Row/column number (1-12)

# Convert to numpy arrays
features = np.array(features)
labels = np.array(labels)

# Split into training and validation sets
train_features = features[:50 * num_iterations_per_epoch]
train_labels = labels[:50 * num_iterations_per_epoch]
val_features = features[50 * num_iterations_per_epoch:]
val_labels = labels[50 * num_iterations_per_epoch:]

# Train and evaluate models
models = {
    "SVM": SVC(kernel='linear'),
    "Random Forest": RandomForestClassifier(n_estimators=100),
    "Logistic Regression": LogisticRegression(max_iter=1000)
}

for model_name, model in models.items():
    model.fit(train_features, train_labels)
    val_preds = model.predict(val_features)
    accuracy = accuracy_score(val_labels, val_preds)
    print(f"{model_name} Validation Accuracy: {accuracy:.2%}")
```

```
SVM Validation Accuracy: 8.68%
Random Forest Validation Accuracy: 8.32%
Logistic Regression Validation Accuracy: 8.33%
```

In [ ]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, roc_curve, auc, precision_recall_curve
from sklearn.preprocessing import label_binarize

# Assuming eeg_data, stim_annots, and target_annots are already loaded
# Define the letter grid
letter_grid = [
    ['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', '_']
]

# Parameters
num_epochs = len(target_annots)
```

```python
num_iterations_per_epoch = 180
samples_per_segment = int(800 * sampling_rate / 1000)  # 800 ms = 192 samples at 240 Hz

# Define time windows in milliseconds and convert to sample indices
window_p300 = (250, 450)  # P300 window in ms
window_baseline = (600, 800)  # Baseline window in ms
p300_start = int(window_p300[0] * sampling_rate / 1000)  # 60 samples
p300_end = int(window_p300[1] * sampling_rate / 1000)    # 108 samples
baseline_start = int(window_baseline[0] * sampling_rate / 1000)  # 144 samples
baseline_end = int(window_baseline[1] * sampling_rate / 1000)    # 192 samples

# Initialize lists to store features and labels
features = []
labels = []

# Loop through each epoch
for epoch in range(num_epochs):
    # Identify the rows corresponding to the current epoch
    epoch_start = epoch * num_iterations_per_epoch
    epoch_end = epoch_start + num_iterations_per_epoch

    # Verify bounds
    if epoch_end > len(stim_annots):
        raise ValueError(f"Epoch {epoch + 1} exceeds stim_annots length: {len(stim_annots)}")

    # Filter the stim_annots DataFrame for the current epoch
    epoch_flashes = stim_annots.iloc[epoch_start:epoch_end]

    # Loop through each flash in the epoch
    for _, flash in epoch_flashes.iterrows():
        # Get the start time of the flash
        start_time = flash['start']  # Start time in microseconds
        start_sample = int(start_time * sampling_rate / 1e6)  # Convert to sample index

        # Extract the EEG data for the specific flash
        if start_sample + samples_per_segment <= eeg_data.shape[0]:
            epoch_data = eeg_data[start_sample:start_sample + samples_per_segment, 0]
        else:
            print(f"Warning: Flash at sample {start_sample} exceeds data length")
            epoch_data = np.array([np.nan] * samples_per_segment)

        # Ensure the epoch data is the correct length
        if len(epoch_data) != samples_per_segment:
            raise ValueError(f"Epoch data length {len(epoch_data)} does not match {samples_per_segment}")

        # Extract the P300 and baseline windows
        p300_window = epoch_data[p300_start:p300_end]
        baseline_window = epoch_data[baseline_start:baseline_end]

        # Compute the P300 score
        p300_score = np.mean(p300_window) - np.mean(baseline_window)

        # Append features and labels
        features.append([p300_score, np.mean(epoch_data), np.std(epoch_data)])  # Example features
        labels.append(int(flash['description']))  # Row/column number (1-12)

# Convert to numpy arrays
features = np.array(features)
labels = np.array(labels)

# Split into training and validation sets
train_features = features[:50 * num_iterations_per_epoch]
train_labels = labels[:50 * num_iterations_per_epoch]
val_features = features[50 * num_iterations_per_epoch:]
val_labels = labels[50 * num_iterations_per_epoch:]

# Train and evaluate models
models = {
    "SVM": SVC(kernel='linear', probability=True),
    "Random Forest": RandomForestClassifier(n_estimators=100),
    "Logistic Regression": LogisticRegression(max_iter=1000)
}

# Plotting
plt.figure(figsize=(18, 12))

# Binarize labels for ROC and Precision-Recall curves
val_labels_bin = label_binarize(val_labels, classes=np.unique(val_labels))

for i, (model_name, model) in enumerate(models.items()):
    model.fit(train_features, train_labels)
    val_preds = model.predict(val_features)
    val_probs = model.predict_proba(val_features)
```

```python
    # Accuracy
    accuracy = accuracy_score(val_labels, val_preds)
    print(f"{model_name} Validation Accuracy: {accuracy:.2%}")

    # Confusion Matrix
    cm = confusion_matrix(val_labels, val_preds)
    plt.subplot(3, 3, i * 3 + 1)
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(f"{model_name} Confusion Matrix")
    plt.colorbar()
    plt.xlabel("Predicted")
    plt.ylabel("True")

    # ROC Curve
    fpr, tpr, _ = roc_curve(val_labels_bin.ravel(), val_probs.ravel())
    roc_auc = auc(fpr, tpr)
    plt.subplot(3, 3, i * 3 + 2)
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'{model_name} ROC Curve')
    plt.legend(loc="lower right")

    # Precision-Recall Curve
    precision, recall, _ = precision_recall_curve(val_labels_bin.ravel(), val_probs.ravel())
    plt.subplot(3, 3, i * 3 + 3)
    plt.plot(recall, precision, color='blue', lw=2, label='Precision-Recall curve')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title(f'{model_name} Precision-Recall Curve')
    plt.legend(loc="lower left")

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

** The above code takes a long time to run **

2

Describe your algorithm in detail. Also describe what you tried that didn't work. (6 pts)

Model Explanation I plan to use Multi-class classification algorithms as each letter and number is a class. Some algorithms that are mentioned in the slides are SVM, logistic regression, and Random forests. I decided to try these three.

1. Data Preparation

   Letter Grid: A 6x6 grid of characters is defined, representing the P300 speller interface.

   EEG Data: The EEG data is loaded, and specific time windows are defined for the P300 response (250-450 ms) and baseline activity (600-800 ms).

   Feature Extraction:

   ```
    For each flash (row/column highlight), the EEG data is extracted for the defined time windows.

    The P300 score is computed as the difference between the mean EEG activity in the P300 window
   and the baseline window.

     Features like the mean and standard deviation of the EEG data are also extracted.

     I could've done better on the features and extracted the area, line-lengeth and energy
   features. But it was taking too long to run is why I decided to stick with these. I will try to
   implement these.
   ```

   Labels: Each flash is labeled with the corresponding row/column number (1-12).

2. Training and Validation Split

   The dataset is split into:

   ```
    Training Set: First 50 epochs (9000 flashes).
    Validation Set: Remaining 35 epochs (6300 flashes).
   ```

3. Model Training and Evaluation

   Three models are trained:

   ```
    Support Vector Machine (SVM) with a linear kernel.
    Random Forest with 100 trees.
   ```

```
Logistic Regression.
```

Each model is trained on the training set and evaluated on the validation set. Accuracy is computed as the percentage of correctly predicted labels.

4. Output

   The validation accuracy for each model is printed:

```
SVM: 8.6%
Random Forest: 8.3%
Logistic Regression: 8.3%
```

Description based on Plots- Plotting is the easiest way to analyse and debug. I decided to plot based on what I had learned in my Machine Learning class.

The ROC (Receiver Operating Characteristic) and Precision-Recall curves being the same for all three algorithms (SVM, Random Forest, and Logistic Regression) suggests that the models are performing similarly poorly, likely due to

1. Poor Model Performance

   All three models achieve very low accuracy (~8%), which is close to random guessing for a 12-class problem (random chance = 1/12 ≈ 8.33%).

   When models perform poorly, their ROC and Precision-Recall curves tend to look similar because they are not effectively distinguishing between classes.

2. Random Guessing Behavior

   If the models are essentially guessing, the ROC curve will be close to the diagonal line (AUC ≈ 0.5), and the Precision-Recall curve will be close to the baseline (precision ≈ proportion of positive classes).

   This behavior is expected when the models fail to learn meaningful patterns from the data.

3. Insufficient Feature Representation

   The current features (P300 score, mean, std) may not capture the P300 response effectively.

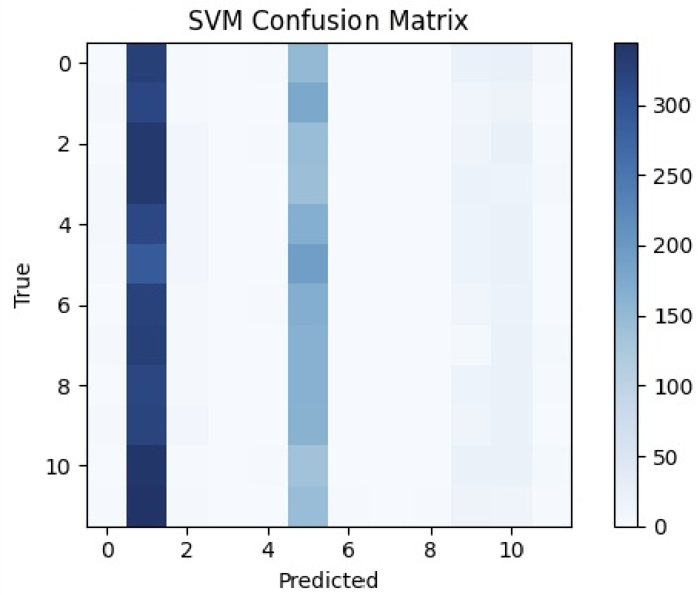
   If the features are not discriminative, all models will struggle to learn meaningful patterns, resulting in similar performance.

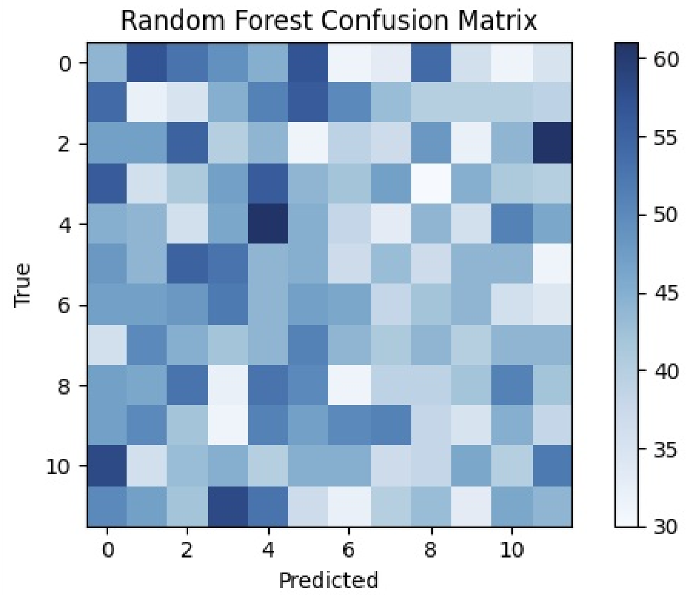
To improve: Given more time I would have:

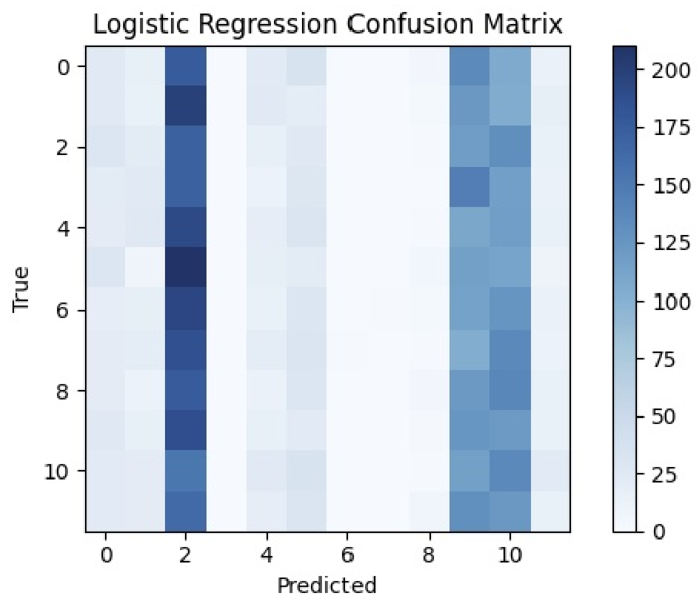
```
Added more discriminative features like spectral power, wavelet coefficients, or time-
frequency features.

Used techniques like class weighting or oversampling to address the imbalance.

Experimented with more complex models like XGBoost, Gradient Boosting, or Deep Learning.

Optimized hyperparameters for each model to improve performance.

Applied filtering, artifact removal, or other preprocessing steps to clean the EEG data.
```

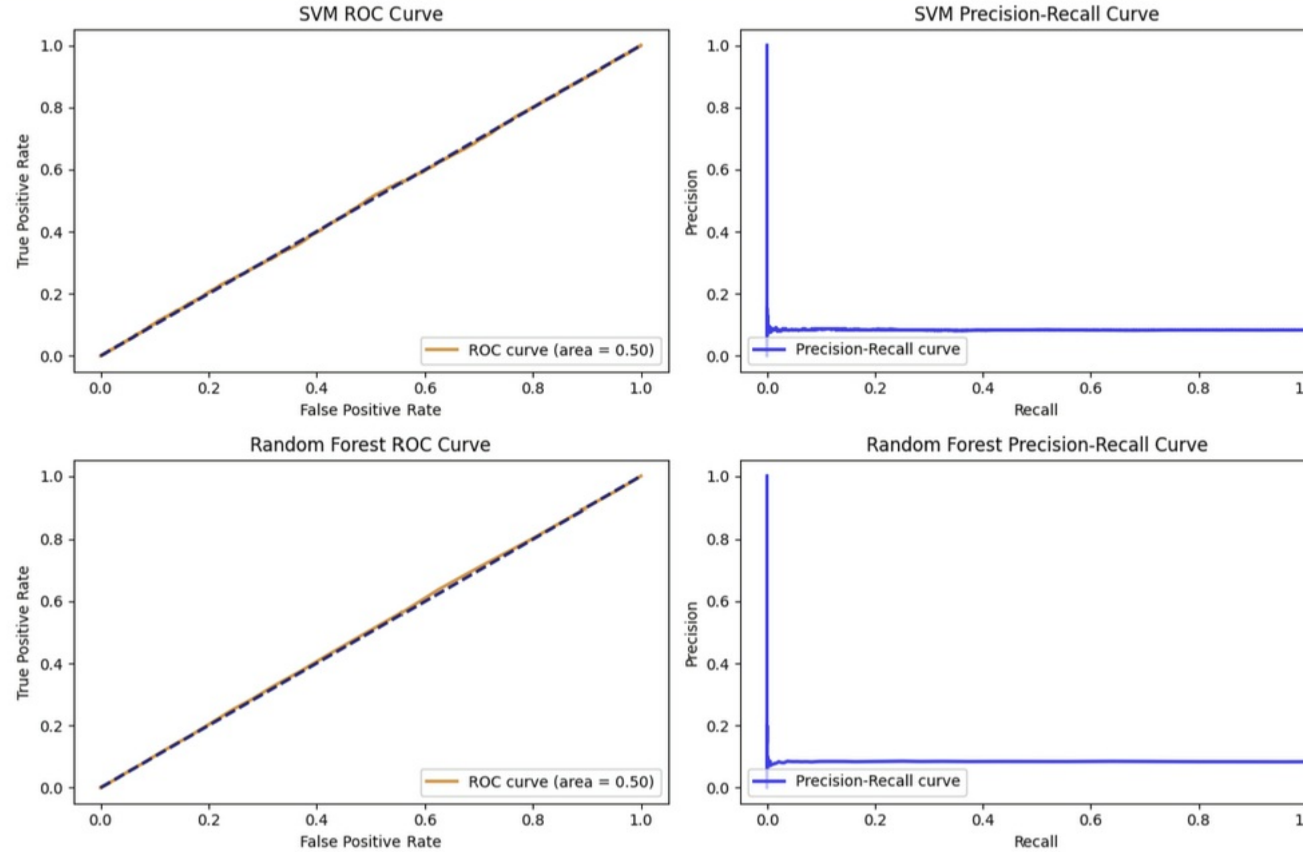
Confusion Matrix for all three: The matrix shows a high number of misclassifications, as most of the counts are off the diagonal. The model struggles to correctly classify most classes, indicating poor performance.

Regression based algorithms won't work here because this is multi-classification problem.


SVM Confusion Matrix
True
Predicted
0
2
4
6
8
10
300
250
200
150
100
50
0


Random Forest Confusion Matrix
True
Predicted
0
2
4
6
8
10
60
55
50
45
40
35
30


Logistic Regression Confusion Matrix
True
Predicted
0
2
4
6
8
10
200
175
150
125
100
75
50
25
0


SVM ROC Curve
True Positive Rate
False Positive Rate
ROC curve (area = 0.50)
SVM Precision-Recall Curve
Precision
Recall
Precision-Recall curve
Random Forest ROC Curve
True Positive Rate
False Positive Rate
ROC curve (area = 0.50)
Random Forest Precision-Recall Curve
Precision
Recall
Precision-Recall curve