lab_eeg_partial

September 30, 2025

1 Lab: Source Localization for EEG

EEG or Electroencephalography is a powerful tool for neuroscientists in understanding brain activity. In EEG, a patient wears a headset with electrodes that measures voltages at a number of points on the scalp. These voltages arise from ionic currents within the brain. A common *inverse problem* is to estimate the which parts of the brain caused the measured response. Source localization is useful in understanding which parts of the brain are involved in certain tasks. A key challenge in this inverse problem is that the number of unknowns (possible locations in the brain) is much larger than the number of measurements. In this lab, we will use LASSO regression on a real EEG dataset to overcome this problem and determine the brain region that is active under an auditory stimulus.

In addition to the concepts in the prostate LASSO demo you will learn to: * Represent responses of multi-channel time-series data, such as EEG, using linear models * Perform LASSO and Ridge regression * Select the regularization level via cross-validation * Visually compare the sparsity between the solutions

We first download standard packages.

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     import pickle

     from sklearn.linear_model import Lasso, Ridge, ElasticNet
     from sklearn.metrics import r2_score
     from sklearn.model_selection import train_test_split
```

1.1 Load the Data

The data in this lab is taken from one of the sample datasets in the MNE website. The sample data is a recording from one subject who experienced some auditory stimulus on the left ear.

The raw data is very large (`1.5G`) and also requires that you install the `mne` python package. To make this lab easier, I have extracted and processed a small section of the data. The following command will download a `pickle` file `eeg_dat.p` to your local machine. If you do want to create the data yourself, the program to create the data is in this directory in the github repository.

```
[2]: fn_src ='https://drive.google.com/uc?
       ↪export=download&id=1RzQpKONOcXSMxH2ZzOI4iVMiTgD6ttSl'
     fn_dst ='eeg_dat.p'
```

```python
import os
from six.moves import urllib

if os.path.isfile(fn_dst):
    print('File %s is already downloaded' % fn_dst)
else:
    print('Fetching file %s [53MB].  This may take a minute..' % fn_dst)
    urllib.request.urlretrieve(fn_src, fn_dst)
    print('File %s downloaded' % fn_dst)
```

```
Fetching file eeg_dat.p [53MB].  This may take a minute..
File eeg_dat.p downloaded
```

Now run the following command which will get the data from the `pickle` file.

```python
[3]: import pickle
     fn = 'eeg_dat.p'
     with open(fn, 'rb') as fp:
         [X,Y] = pickle.load(fp)
```

To understand the data, there are three key variables: * `nt` = number of time steps that we measure data * `nchan` = number of channels (i.e. electrodes) measured in each time step * `ncur` = number of currents in the brain that we want to estimate.

Each current comes from one brain region (called a *voxel*) in either the `x`, `y` or `z` direction. So,

`nvoxels = ncur / 3`

The components of the `X` and `Y` matrices are: * `Y[i,k]` = electric field measurement on channel `i` at time `k` * `X[i,j]` = sensitivity of channel `i` to current `j`.

Using `X.shape` and `Y.shape` compute and print `nt`, `nchan`, `ncur` and `nvoxels`.

```python
[4]: # TODO
     nt = X.shape[1]
     ncur = Y.shape[1]
     nchan = X.shape[0]
     nvoxels = ncur / 3
```

1.2 Ridge Regression

Our goal is to estimate the currents in the brain from the measurements `Y`. One simple linear model is:

`Y[i,k]  = \sum_j X[i,j]*W[j,k]+ b[k]`

where `W[j,k]` is the value of current `j` at time `k` and `b[k]` is a bias. We can solve for the current matrix `W` via linear regression.

Howeever, there is a problem: * There are `nt x ncur` unknowns in `W` * There are only `nt x nchan` measurements in `Y`.

In this problem, we have:

```
number of measurements  << number of unknowns
```

We need to use regularization in these circumstances. We first try Ridge regression.

First split the data into training and test. Use the `train_test_split` function with `test_size=0.33`.

```
[6]: # TODO
     Xtr,Xts,Ytr,Yts = train_test_split(X,Y,test_size=0.33)
```

Use the `Ridge` regression object in `sklearn` to fit the model on the training data. Use a regularization, `alpha=1`.

```
[7]: # TODO
     regr = Ridge(alpha=1)
     regr.fit(Xtr,Ytr)
```

```
[7]: Ridge(alpha=1)
```

Preict the values `Y` on both the training and test data. Use the `r2_score` method to measure the R^2 value on both the training and test. You will see that R^2 value is large for the training data, it is very low for the test data. This suggest that even with regularization, the model is over-fitting the data.

```
[9]: # TODO
     Ytr_pred = regr.predict(Xtr)
     Yts_pred = regr.predict(Xts)

     rsq_tr = r2_score(Ytr,Ytr_pred)
     rsq_ts = r2_score(Yts,Yts_pred)
     print(rsq_tr,rsq_ts)
```

```
0.5822515386815218 0.1922804672352655
```

Next, try to see if we can get a better R^2 score using different values of `alpha`. Use cross-validation to measure the test R^2 for 20 `alpha` values logarithmically spaced from 10^{-2} to 10^{2} (use `np.logspace()`). You can use regular cross-validation. You do not need to do K-fold.

```
[13]: # TODO
      rsq_ts_list =[]
      alphas = np.logspace(-2,2,20)

      for alpha in np.logspace(-2,2,20):
          regr = Ridge(alpha=alpha)
          regr.fit(Xtr,Ytr)
          Yts_pred = regr.predict(Xts)
          rsq_ts = r2_score(Yts,Yts_pred)
          rsq_ts_list.append(rsq_ts)
```

Plot the test `R^2` vs. `alpha`. And print the maximum test `R^2`. You should see that the maximum test `R^2` is still not very high.

```
[18]: # TODO
plt.plot(alphas,rsq_ts_list)
plt.xscale('log')
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.title('R^2 vs. alpha')
plt.show()
print("max r^2",np.max(rsq_ts_list))
# print the alpha that gives the maximum R^2
print("alpha",alphas[np.argmax(rsq_ts_list)])
```
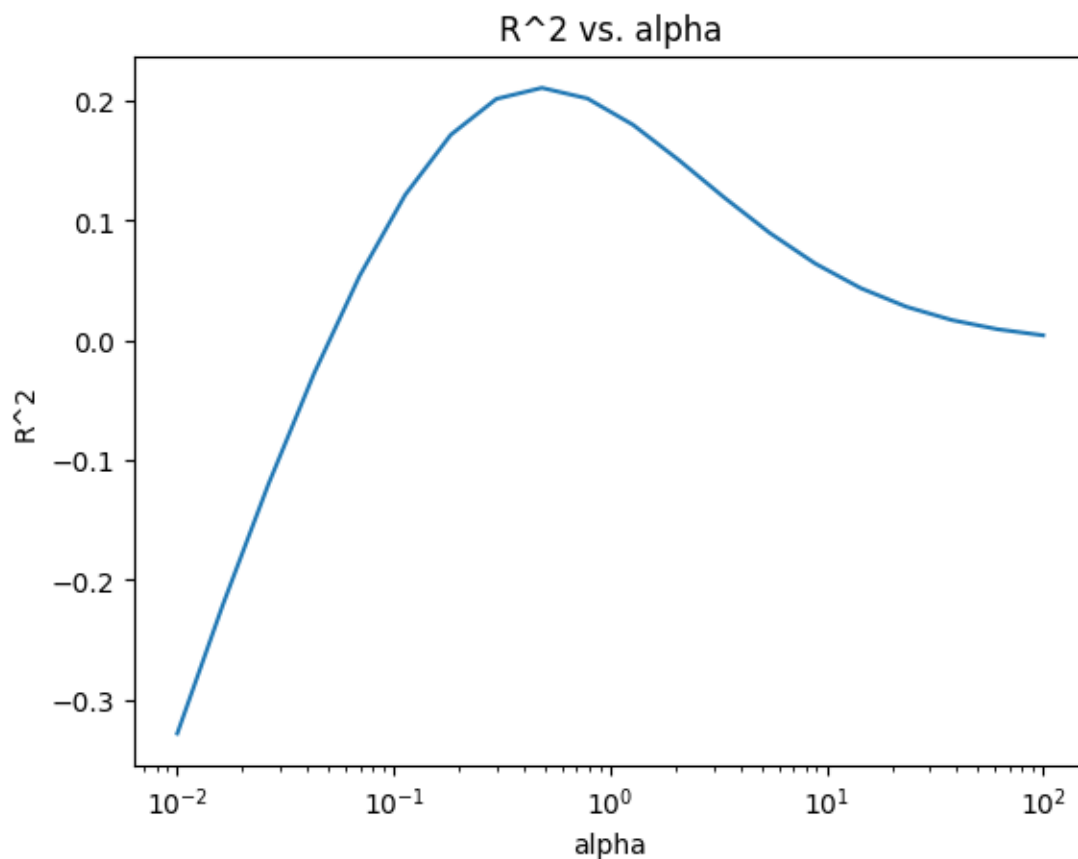


```
max r^2 0.21089181817743335
alpha 0.4832930238571752
```

Now, let's take a look at the solution.

- Find the optimal regularization `alpha` from the cross-validation
- Re-fit the model at the optimal `alpha`

- Get the current matrix `W` from the coefficients in the linear model. These are stored in `regr.coef_`. You may need a transpose
- For each current `j` compute `Wrms[j] = sqrt( sum_k W[j,k]**2 )` which is root mean squared current.

You will see that the vector `Wrms` is not sparse. This means that the solution that is found with Ridge regression finds currents in all locations.

```
[19]: # TODO
      regr = Ridge(alpha=alphas[np.argmax(rsq_ts_list)])
      regr.fit(Xtr,Ytr)
      W = regr.coef_
      Wrms = np.sqrt(np.sum(W**2,axis=1))
      print(Wrms)
```

```
[1.13225946 1.17108924 1.20482549 1.23274345 1.25551175 1.27525788
 1.29526182 1.31928293 1.3506413  1.39135951 1.44167926 1.50016114
 1.56424733 1.63100657 1.69779792 1.76271078 1.82480395 1.88421634
 1.94222232 2.00120356 2.06446052 2.13572027 2.21829293 2.31401334
 2.42231179 2.53980493 2.66061469 2.7773461  2.88239579 2.96924033
 3.03341142 3.07298606 3.08852003 3.08248662 3.0583753  3.0197143
 2.96931919 2.90897861 2.83965058 2.7620054  2.67703681 2.58645656
 2.49271742 2.39868942 2.30717576 2.22051344 2.1404134  2.06802153
 2.00403125 1.94862542 1.90119318 1.85998977 1.82209082 1.78387759
 1.74203784 1.6947565  1.64265893 1.58910975 1.53958071 1.50010953
 1.47522826 1.46624468 1.47080708 1.48397069 1.50006893 1.51434164
 1.52377459 1.52712339 1.52445462 1.51659678 1.5047395  1.49024327
 1.47460407 1.4593274  1.4456801  1.43431936 1.42501577 1.41666407
 1.40765632 1.39647954 1.38228304 1.36517729 1.34615634 1.32670968
 1.30830442]
```

1.3 LASSO Regression

We can improve the estimate by imposing sparsity. Biologically, we know that only a limited number of brain regions should be involved in the reponse to a particular stimuli. As a result, we would expect that the current matrix `W[j,k]` to be zero for most values `j,k`. We can impose this constraint using LASSO regularization.

Re-fit the training data using the `Lasso` model with `alpha=1e-3`. Also set `max_iter=100` and `tol=0.01`. The LASSO solver is much slower, so this make take a minute.

```
[20]: # TODO
      lasso = Lasso(alpha=1e-3,max_iter=100,tol=0.01)
```

Now, test the model on the test data and measure the `R^2` value. You should get a much better fit than with the Ridge regression solution.

```
[22]: # TODO
      lasso.fit(Xtr,Ytr)
      Yts_pred = lasso.predict(Xts)
```

```
rsq_ts = r2_score(Yts,Yts_pred)
print(rsq_ts)
```

0.17412652112244573

We can now search for the optimal `alpha`. Use cross-validation to find the `alpha` logarithically space between `alpha=10^{-3}` and `alpha=10^{-4}`. Each fit takes some time, so use only 5 values of `alpha`. Also for each `alpha` store the current matrix. This way, you will not have to re-fit the model.

```
[32]:  # TODO
       rsq_ts_list = []
       W_list = []   # Store weight matrices for each alpha
       alphas = np.logspace(-4,-3,5)

       for alpha in alphas:
           lasso = Lasso(alpha=alpha, max_iter=100, tol=0.01)
           lasso.fit(Xtr, Ytr)

           W = lasso.coef_
           W_list.append(W)

           Yts_pred = lasso.predict(Xts)
           rsq_ts = r2_score(Yts, Yts_pred)
           rsq_ts_list.append(rsq_ts)

           print(f"Alpha: {alpha:.2e}, Test R²: {rsq_ts:.4f}")
```

```
Alpha: 1.00e-04, Test R²: 0.3243
Alpha: 1.78e-04, Test R²: 0.3759
Alpha: 3.16e-04, Test R²: 0.3675
Alpha: 5.62e-04, Test R²: 0.2964
Alpha: 1.00e-03, Test R²: 0.1741
```

Plot the `r^2` value vs. `alpha`. Print the optimal `r^2`. You should see it is much higher than with the best Ridge Regression case.

```
[33]:  # TODO
       plt.figure(figsize=(8, 6))
       plt.plot(alphas, rsq_ts_list)
       plt.xscale('log')
       plt.xlabel('Alpha')
       plt.ylabel('Test R²')
       plt.title('LASSO: Test R² vs. Alpha')
       plt.grid(True, alpha=0.3)
       plt.show()

       optimal_idx = np.argmax(rsq_ts_list)
       optimal_alpha = alphas[optimal_idx]
```
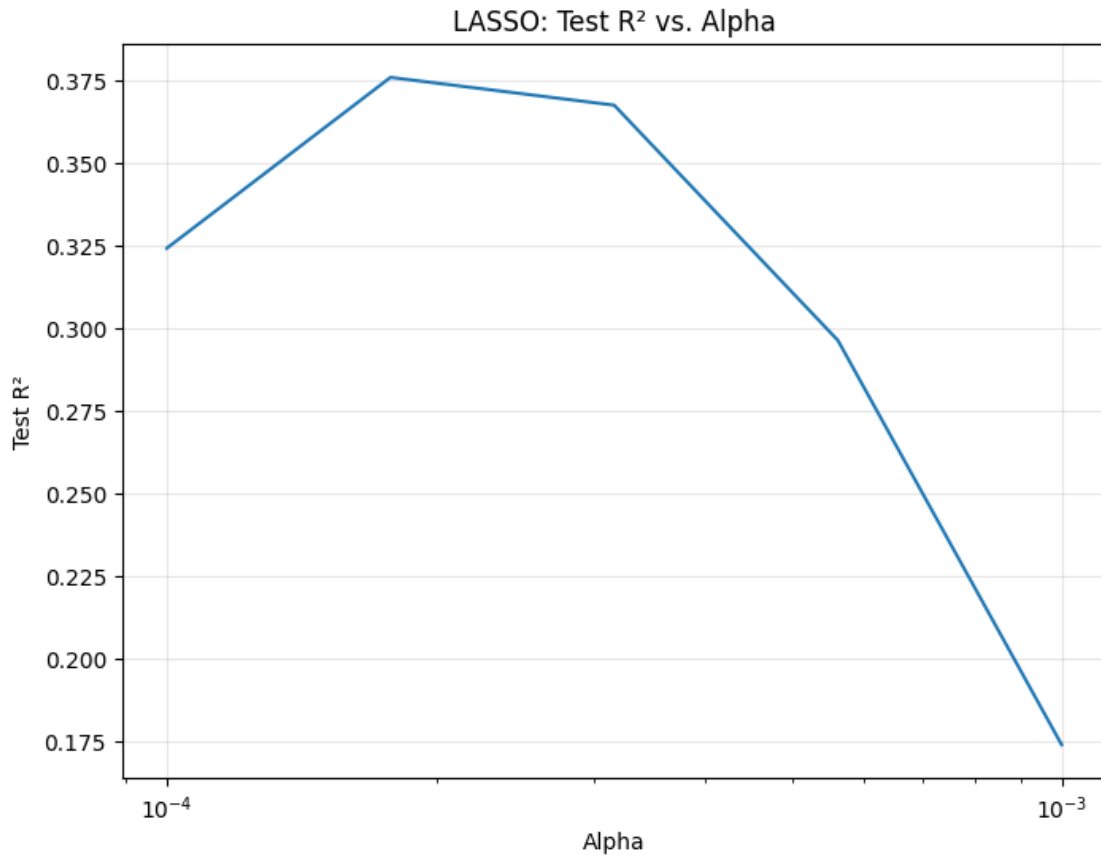
```
max_r2 = rsq_ts_list[optimal_idx]

print(f"Maximum test R²: {max_r2:.4f}")
print(f"Optimal alpha: {optimal_alpha:.2e}")
```

LASSO: Test R² vs. Alpha


```
Maximum test R²: 0.3759
Optimal alpha: 1.78e-04
```

Display the current matrix `W` for the optimal `alpha` as you did in the Ridge Regression case. You will see that is much sparser.

```
[34]: # TODO
W_optimal = W_list[optimal_idx]

# Compute RMS current for each brain region
Wrms_optimal = np.sqrt(np.sum(W_optimal**2, axis=1))

print(f"Using optimal alpha = {optimal_alpha:.2e}")
print(f"Shape of weight matrix W: {W_optimal.shape}")
print(f"Number of non-zero currents: {np.sum(Wrms_optimal > 1e-6)}")
```

```python
print(f"Sparsity: {(1 - np.sum(Wrms_optimal > 1e-6) / len(Wrms_optimal)) * 100:.
 ↪1f}%")

# Display the RMS current values
print(f"\nRMS current values (first 20):")
print(Wrms_optimal[:20])

# Plot the RMS current distribution
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(Wrms_optimal, 'b-', linewidth=1)
plt.xlabel('Current Index')
plt.ylabel('RMS Current')
plt.title('LASSO: RMS Current Distribution')
plt.grid(True, alpha=0.3)

plt.subplot(1, 2, 2)
plt.hist(Wrms_optimal[Wrms_optimal > 1e-6], bins=20, alpha=0.7, color='blue')
plt.xlabel('RMS Current')
plt.ylabel('Frequency')
plt.title('Histogram of Non-zero RMS Currents')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```
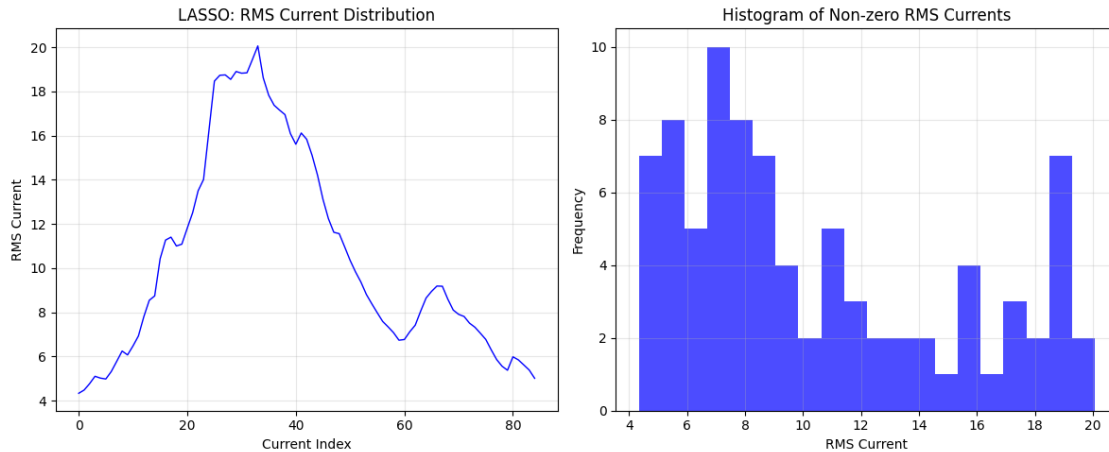
```
Using optimal alpha = 1.78e-04
Shape of weight matrix W: (85, 22494)
Number of non-zero currents: 85
Sparsity: 0.0%

RMS current values (first 20):
[ 4.3320326   4.47938459  4.76443726  5.10173297  5.01937372  4.97944555
  5.31675673  5.77614206  6.24473644  6.07018219  6.47490879  6.9273322
  7.80094011  8.54248482  8.74092861 10.4217325  11.26797873 11.40231711
 11.00099266 11.08416378]
```

1.4 More fun

If you want to more on this lab: * Install the MNE python package. This is an amazing package with many tools for processing EEG data. * In particular, you can use the above results to visualize where in the brain the currents sources are. * You can also improve the fitting with more regularization. For example, we know that the currents will be non-zero in groups: If the current is non-zero for one time, it is likely to non-zero for all time. You can use the Group LASSO method. * You can combine these results to make predictions about what the patient is seeing or hearing or thinking.

```python
import mne
import warnings
warnings.filterwarnings('ignore')

# Create MNE info structure for our EEG data
n_channels = X.shape[0]
ch_names = [f'EEG{i:03d}' for i in range(n_channels)]
ch_types = ['eeg'] * n_channels
info = mne.create_info(ch_names=ch_names, sfreq=1000, ch_types=ch_types)

# Convert our measurement data to MNE Evoked object
evoked = mne.EvokedArray(Y, info, tmin=0)

# Plot the EEG data
evoked.plot(spatial_colors=True, time_unit='ms')

# Create source space coordinates (simplified brain model)
n_sources = X.shape[1]
n_voxels = n_sources // 3

# Generate brain coordinates
```

```python
phi = np.linspace(0, np.pi, int(np.sqrt(n_voxels)))
theta = np.linspace(0, 2*np.pi, int(np.sqrt(n_voxels)))

brain_coords = []
for i in range(n_voxels):
    p_idx = i % len(phi)
    t_idx = (i // len(phi)) % len(theta)
    x = 0.08 * np.sin(phi[p_idx]) * np.cos(theta[t_idx])
    y = 0.08 * np.sin(phi[p_idx]) * np.sin(theta[t_idx])
    z = 0.08 * np.cos(phi[p_idx])
    brain_coords.append([x, y, z])

brain_coords = np.array(brain_coords[:n_voxels])

# Combine x,y,z currents for each voxel from LASSO solution
voxel_activity = np.zeros(n_voxels)
for i in range(n_voxels):
    if 3*i+2 < len(Wrms_optimal):
        voxel_activity[i] = np.sqrt(Wrms_optimal[3*i]**2 +
                                    Wrms_optimal[3*i+1]**2 +
                                    Wrms_optimal[3*i+2]**2)

# Visualize brain sources
fig = plt.figure(figsize=(15, 5))

# 3D brain plot
ax1 = fig.add_subplot(1, 3, 1, projection='3d')
scatter = ax1.scatter(brain_coords[:, 0], brain_coords[:, 1], brain_coords[:,
  ↪2],
                      c=voxel_activity, s=50, cmap='hot', alpha=0.8)
ax1.set_title('Brain Source Activity')
ax1.set_xlabel('X (m)')
ax1.set_ylabel('Y (m)')
ax1.set_zlabel('Z (m)')
plt.colorbar(scatter, ax=ax1, shrink=0.6)

# Top view
ax2 = fig.add_subplot(1, 3, 2)
scatter2 = ax2.scatter(brain_coords[:, 0], brain_coords[:, 1],
                       c=voxel_activity, s=50, cmap='hot', alpha=0.8)
ax2.set_title('Top View')
ax2.set_xlabel('X (m)')
ax2.set_ylabel('Y (m)')
ax2.set_aspect('equal')
plt.colorbar(scatter2, ax=ax2)

# Side view
```

```
ax3 = fig.add_subplot(1, 3, 3)
scatter3 = ax3.scatter(brain_coords[:, 1], brain_coords[:, 2],
                       c=voxel_activity, s=50, cmap='hot', alpha=0.8)
ax3.set_title('Side View')
ax3.set_xlabel('Y (m)')
ax3.set_ylabel('Z (m)')
ax3.set_aspect('equal')
plt.colorbar(scatter3, ax=ax3)

plt.tight_layout()
plt.show()

# Print top active sources
print("Top 5 most active brain regions:")
top_sources = np.argsort(voxel_activity)[-5:][::-1]
for i, src_idx in enumerate(top_sources):
    coord = brain_coords[src_idx]
    print(f"{i+1}. Voxel {src_idx}: Activity = {voxel_activity[src_idx]:.4f}")
    print(f"    Location: ({coord[0]*1000:.1f}, {coord[1]*1000:.1f},␣
 ↪{coord[2]*1000:.1f}) mm")
```


EEG (305 channels)   N_ave=1
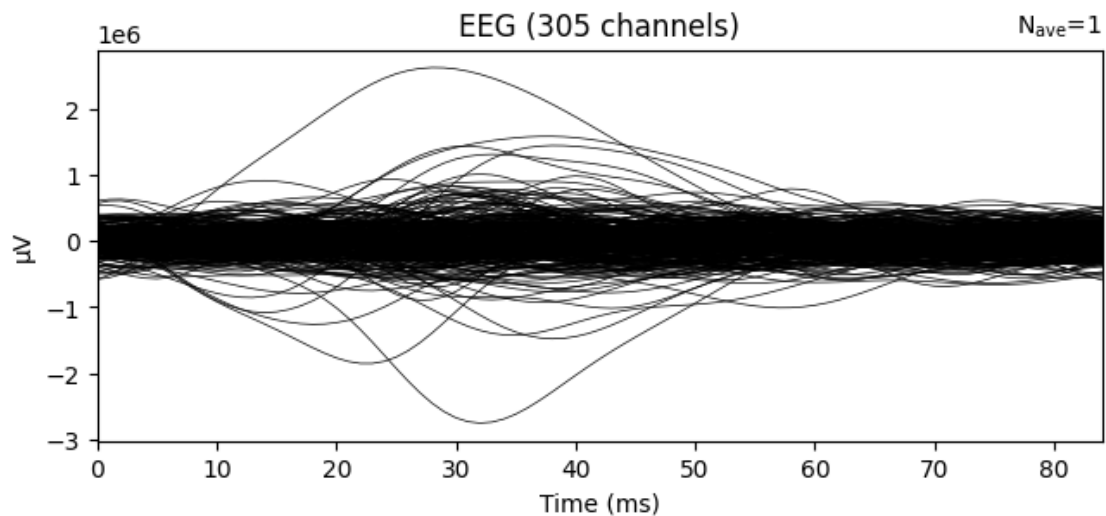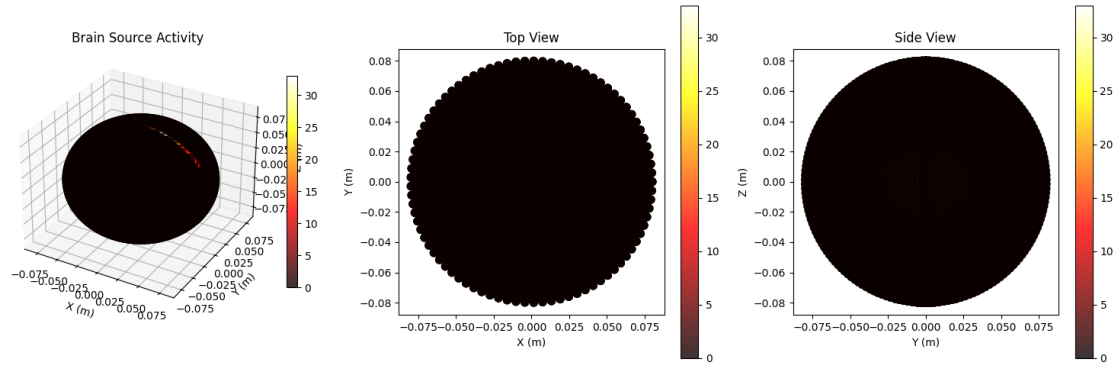
Top 5 most active brain regions:
1. Voxel 10: Activity = 32.9753
   Location: (28.9, 0.0, 74.6) mm
2. Voxel 11: Activity = 32.6602
   Location: (31.6, 0.0, 73.5) mm
3. Voxel 9: Activity = 32.4442
   Location: (26.1, 0.0, 75.6) mm
4. Voxel 8: Activity = 30.9127
   Location: (23.3, 0.0, 76.5) mm
5. Voxel 12: Activity = 29.7222
   Location: (34.3, 0.0, 72.3) mm