lab5

September 28, 2020

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
```

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
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
# TODO
nt = len(Y[0])
ncur = len(X[0])
nchan = len(Y)
nvoxels = ncur / 3
print(nt)
print(ncur)
print(nchan)
print(nvoxels)
```

```
85
22494
305
7498.0
```

1.1 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`.

```
[52]:   # 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`.

```
[53]:   # TODO
        from sklearn.preprocessing import StandardScaler
        xscal = StandardScaler()
        yscal = StandardScaler()

        # Xtr1 = xscal.fit_transform(Xtr)
        # Xts1 = xscal.transform(Xts)
        # Ytr1 = yscal.fit_transform(Ytr)
        # Yts1 = yscal.transform(Yts)

        Xtr1 = Xtr
        Xts1 = Xts
        Ytr1 = Ytr
        Yts1 = Yts

        regr = Ridge(alpha=1)

        regr.fit(Xtr1, Ytr1)
        yhat1 = regr.predict(Xtr1)
        yhat2 = regr.predict(Xts1)
```

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.

```
[54]: # TODO
      rsq_tr = r2_score(Ytr1, yhat1)
      rsq_ts = r2_score(Yts1, yhat2)
      print('Traing R^2  = %f' % rsq_tr)
      print('Test R^2   = %f' % rsq_ts)
```

```
Traing R^2  = 0.609913
Test R^2   = 0.189119
```

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.

```
[55]: # TODO
      alphas = np.logspace(-2,2,10)
      nalpha = len(alphas)
      rsq = np.zeros(nalpha)

      for i, alpha in enumerate(alphas):
          # Fit on the training data
          reg = Ridge(alpha=alpha)
          reg.fit(Xtr1, Ytr1)

          # Score on the test data
          yhat1 = reg.predict(Xts1)
          rsq[i] = r2_score(Yts1, yhat1)
```
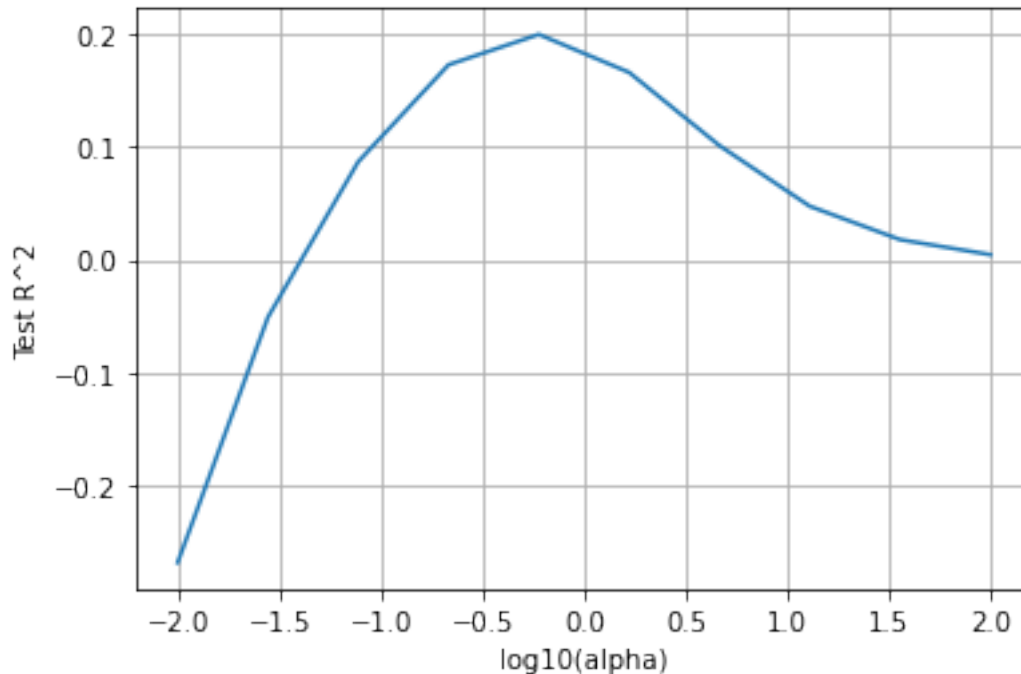
Plot the test R^2 vs. alpha. And print the maximum test R^2. You should see that the maximum test R^2 is still not very high.

```
[56]: plt.plot(np.log10(alphas), rsq)
      plt.xlabel('log10(alpha)')
      plt.ylabel('Test R^2')
      plt.grid()
      plt.show()

      print('Maximum test R^2:  %f' % np.max(rsq))
```

```
Maximum test R^2:  0.199352
```

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.

```
[57]: # TODO
      index = np.argmax(rsq)
      op_alpha = alphas[index]
      reg = Ridge(alpha=alpha)
      reg.fit(Xtr1, Ytr1)
      W = reg.coef_.T
      Wrms = np.sum(W**2, axis = 1)**(1/2)

      print(Wrms)
      plt.plot(np.flipud(np.sort(np.abs(Wrms))))
      plt.xlabel('Sorted var index')
      plt.ylabel('Coeff Mag')
```

```
plt.grid()
plt.legend(['Ridge, normal rule'])
plt.tight_layout()
```

[0.00728155 0.00296452 0.00475282 … 0.00092336 0.00126816 0.00165524]
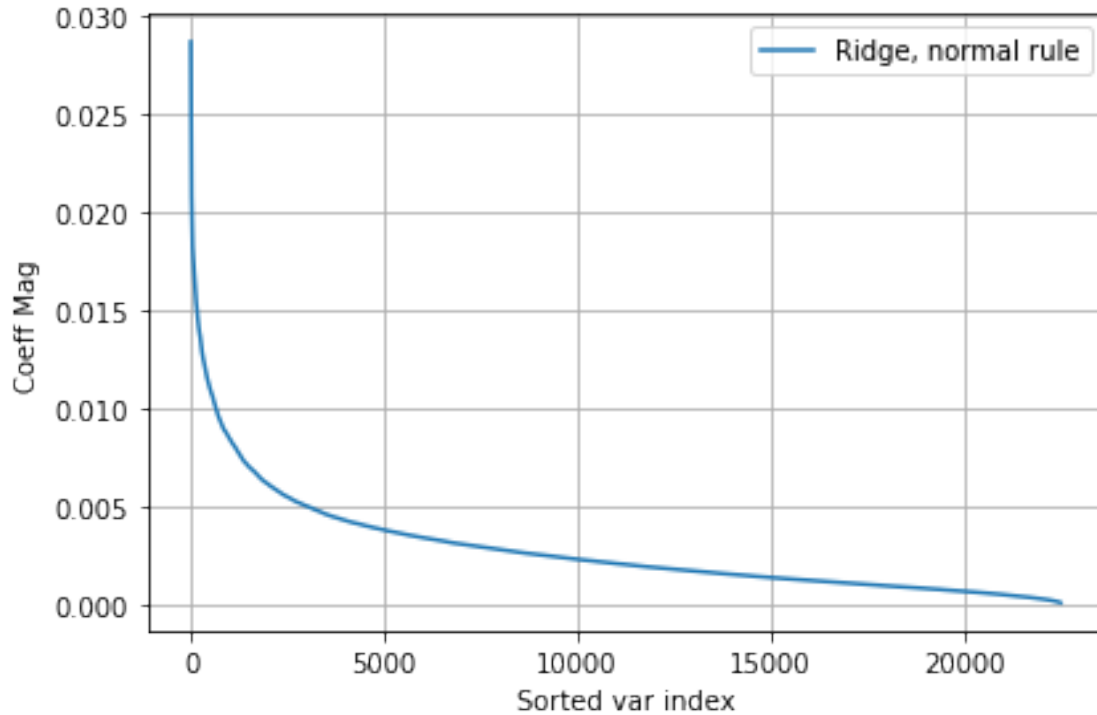
1.2 LASSO Regression

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.

[58]:
```
# TODO
alpha = 1e-3

# Create Lasso object and fit on training data
reg = Lasso(alpha=alpha,max_iter=100,tol=0.01)
reg.fit(Xtr1, Ytr1)
```

[58]: Lasso(alpha=0.001, 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.

```
[59]:  # Predict and score on test
       yhat1 = reg.predict(Xts1)
       rsq = r2_score(Yts1, yhat1)
       print(rsq)
```

0.1837083309498796

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.

```
[61]:  # TODO
       # Alpha values to test
       alphas = np.logspace(-4,-3,5)
       nalpha = len(alphas)

       # Run the cross-validation
       rsq = np.zeros(nalpha)
       for i, alpha in enumerate(alphas):

           # Fit on the training data
           reg = Lasso(alpha=alpha,max_iter=100,tol=0.01)
           reg.fit(Xtr1, Ytr1)

           # Score on the test data
           yhat1 = reg.predict(Xts1)
           rsq[i] = r2_score(Yts1, yhat1)
           print("finish step : %f" % i)
       print("All Clear")
```
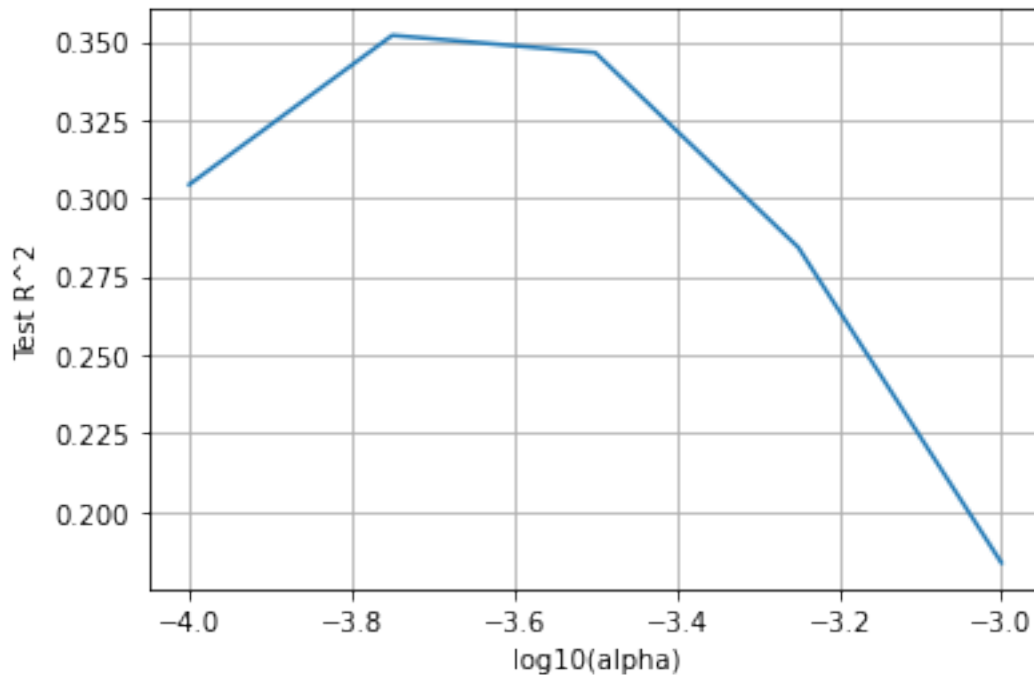
```
finish step : 0.000000
finish step : 1.000000
finish step : 2.000000
finish step : 3.000000
finish step : 4.000000
All Clear
```

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.

```
[62]:  # TODO
       plt.plot(np.log10(alphas), rsq)
       plt.xlabel('log10(alpha)')
       plt.ylabel('Test R^2')
       plt.grid()
```

7

```
plt.show()

print('Maximum test R^2:  %f' % np.max(rsq))
```



```
Maximum test R^2:  0.351957
```

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

```
[63]: # TODO
index = np.argmax(rsq)
op_alpha = alphas[index]
reg = Ridge(alpha=alpha)
reg.fit(Xtr1, Ytr1)
W = reg.coef_.T
Wrms = np.sum(W**2, axis = 1)**(1/2)

print(Wrms)
plt.plot(np.flipud(np.sort(np.abs(Wrms))))
plt.xlabel('Sorted var index')
plt.ylabel('Coeff Mag')
plt.grid()
plt.legend(['Lasso, normal rule'])
plt.tight_layout()
```

```
[6.20551993 2.70348098 3.0549586  … 0.82681025 0.48309581 0.76145471]
```
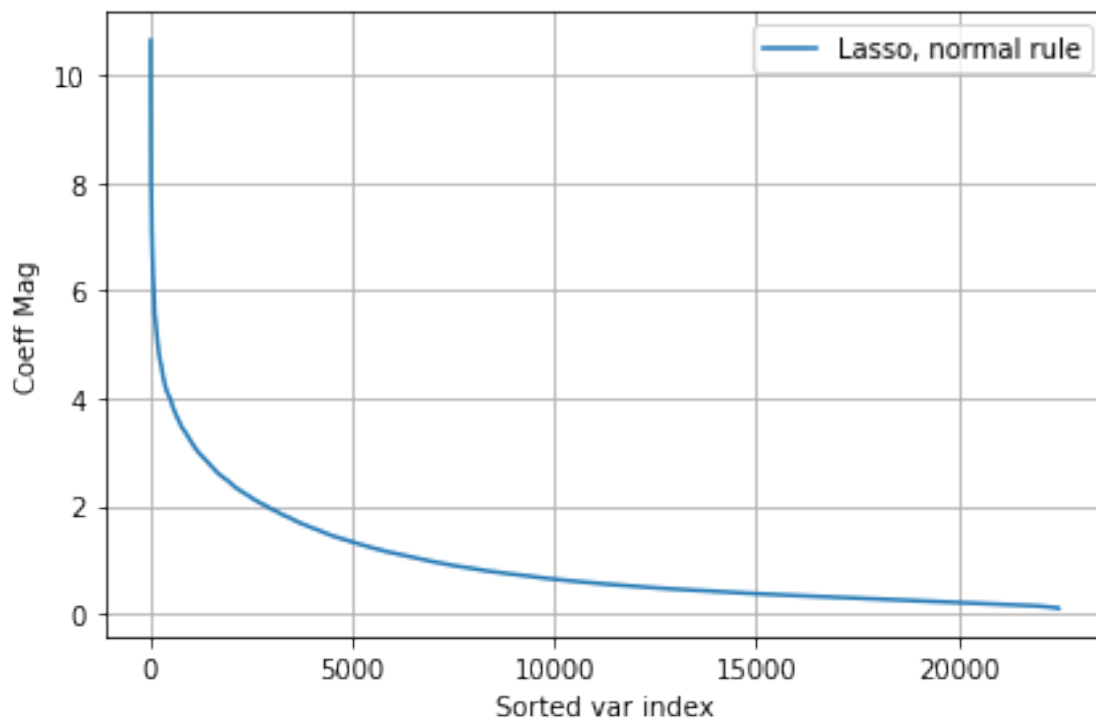
1.3 More fun

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.

[ ]: