

# Lab: Model Selection for Neural Data

Machine learning is a key tool for neuroscientists to understand how sensory and motor signals are encoded in the brain. In addition to improving our scientific understanding of neural phenomena, understanding neural encoding is critical for brain machine interfaces. In this lab, you will use linear regression with feature selection for performing some simple analysis on real neural signals.

Before doing this lab, you should review the ideas in the [polynomial model selection demo \(./polyfit.ipynb\)](#). In addition to the concepts in that demo, you will learn to:

- Load MATLAB data
- Formulate models of different complexities using heuristic model selection
- Fit a linear model for the different model orders (= # of features)
- Select the optimal features via cross-validation

## Loading the data

The data in this lab comes from neural recordings described in:

Stevenson, Ian H., et al. "Statistical assessment of the stability of neural movement representations." *Journal of neurophysiology* 106.2 (2011): 764-774 (<http://jn.physiology.org/content/106/2/764.short>)

Neurons are the basic information processing units in the brain. Neurons communicate with one another via *spikes* or *action potentials* which are brief events where voltage in the neuron rapidly rises then falls. These spikes trigger the electro-chemical signals between one neuron and another. In this experiment, the spikes were recorded from 196 neurons in the primary motor cortex (M1) of a monkey using an electrode array implanted onto the surface of a monkey's brain. During the recording, the monkey performed several reaching tasks and the position and velocity of the hand was recorded as well.

The goal of the experiment is to try to *read the monkey's brain*: That is, predict the hand position from the neural signals from the motor cortex.

We first load the basic packages.

```
In [1]: import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

The full data is available on the CRCNS website <http://crcns.org/data-sets/movements/dream> (<http://crcns.org/data-sets/movements/dream>). This website has a large number of great datasets and can be used for projects as well. To make this lab easier, I have pre-processed the data slightly and placed it in the file `StevensonV2.mat`, which is a MATLAB file. You will need to have this file downloaded in the directory you are working on.

Since MATLAB is widely-used, python provides method for loading MATLAB `mat` files. We can use these commands to load the data as follows.

```
In [2]: import scipy.io
        mat_dict = scipy.io.loadmat('StevensonV2.mat')
```

The returned structure, `mat_dict`, is a dictionary with each of the MATLAB variables that were saved in the `.mat` file. Use the `.keys()` method to list all the variables.

```
In [5]: #TODO
        print (mat_dict.keys(), type(mat_dict.keys()))

        dict_keys(['__header__', '__version__', '__globals__', 'Publication', 'timeBase', 'spikes', 'time', 'handVel', 'handPos', 'target', 'startBins', 'targets', 'startBinned']) <class 'dict_keys'>
```

We extract two variables, `spikes` and `handPos`, from the dictionary `mat_dict`, which represent the recorded spikes per neuron and the hand position. We take the transpose of the spikes data so that it is in the form  $\text{time bins} \times \text{number of neurons}$ . For the `handPos` data, we take the second component which is the position of monkey's hand.

```
In [6]: X0 = mat_dict['spikes'].T
        y0 = mat_dict['handPos'][0,:]
```

The spikes matrix will be a  $n_t \times n_{\text{neuron}}$  matrix where  $n_t$  is the number of time bins and  $n_{\text{neuron}}$  is the number of neurons. Each entry `spikes[k,j]` is the number of spikes in time bin  $k$  from neuron  $j$ . Use the `shape` method to find  $n_t$  and  $n_{\text{neuron}}$  and print the values.

```
In [8]: # TODO
        print('nt = {}, and nneuron = {}'.format(X0.shape[0], X0.shape[1]))

        nt = 15536, and nneuron = 196
```

Now extract the `time` variable from the `mat_dict` dictionary. Reshape this to a 1D array with  $n_t$  components. Each entry `time[k]` is the starting time of the time bin  $k$ . Find the sampling time `tsamp` which is the time between measurements, and `ttotal` which is the total duration of the recording.

```
In [18]: # TODO
time = mat_dict['time'].T.reshape(X0.shape[0])
time_shift = time[1:] - time[:time.shape[0]-1]
tsamp = sum(time_shift)/len(time_shift)
ttotal = time[-1]- time[0]

print('sampling time = {}, and total recording duration = {}'.format(tsamp,ttotal))
```

sampling time = 0.05, and total recording duration = 776.75

## Linear fitting on all the neurons

First divide the data into training and test with approximately half the samples in each. Let  $X_{tr}$  and  $y_{tr}$  denote the training data and  $X_{ts}$  and  $y_{ts}$  denote the test data.

```
In [21]: # TODO
from sklearn.model_selection import train_test_split

Xtr, Xts, ytr, yts = train_test_split(X0, y0, test_size=0.5)
```

Now, we begin by trying to fit a simple linear model using *all* the neurons as predictors. To this end, use the `sklearn.linear_model` package to create a regression object, and fit the linear model to the training data.

```
In [22]: from sklearn.linear_model import LinearRegression

# TODO
all_neuron_model = LinearRegression().fit(Xtr, ytr)
```

Measure and print the normalized RSS on the test data.

```
In [100]: # TODO
ypred = all_neuron_model.predict(Xts)
all_neuron_RSS = np.dot((yts - ypred).T, (yts - ypred))
all_neuron_S = np.sum((yts - np.mean(yts))**2)
all_neuron_NRSS = all_neuron_RSS/all_neuron_S

print('normalized RRS on test data = {}'.format(all_neuron_NRSS))

normalized RRS on test data = 3.829564649502364e+21
```

You should see that the test error is enormous -- the model does not generalize to the test data at all.

# Linear Fitting with Heuristic Model Selection

The above shows that we need a way to reduce the model complexity. One simple idea is to select only the neurons that individually have a high correlation with the output.

Write code which computes the coefficient of determination,  $R_k^2$ , for each neuron  $k$ . Plot the  $R_k^2$  values.

You can use a for loop over each neuron, but if you want to make efficient code try to avoid the for loop and use [python broadcasting](#) ([../Basics/numpy\\_axes\\_broadcasting.ipynb](#)).

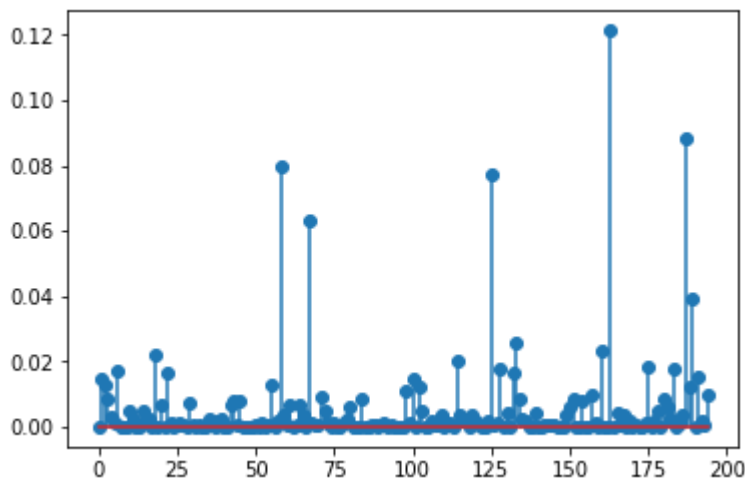
```
In [125]: # TODO
import scipy

y_normal = y0 - np.mean(y0)
X_normal = X0 - np.mean(X0, axis=0)

R_2 = (np.dot(y_normal.T, X_normal)**2) / ((np.dot(y_normal.T, y_normal))*(np.
sum(X_normal**2, axis=0)))
R_2 = R_2[~np.isnan(R_2)] #removing the nan value
plt.stem(np.arange(len(R_2)), R_2)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:7: RuntimeWarning: invalid value encountered in true_divide
import sys
```

```
Out[125]: <StemContainer object of 3 artists>
```



We see that many neurons have low correlation and can probably be discarded from the model.

Use the `np.argsort()` command to find the indices of the  $d=50$  neurons with the highest  $R_k^2$  value. Put the  $d$  indices into an array `Ise1`. Print the indices of the neurons with the 10 highest correlations.

```
In [126]: d = 50 # Number of neurons to use

# TODO
Isel = np.argsort(R_2)[-d:]

print("The neurons with the ten highest R^2 values = {}".format(Isel[-10:]))

The neurons with the ten highest R^2 values = [114  18 160 133 189  67 125  5
8 187 163]
```

Fit a model using only the  $d$  neurons selected in the previous step and print both the test RSS per sample and the normalized test RSS.

```
In [127]: # TODO
X0S = X0[:,Isel]
y0S = y0

XtrS, XtsS, ytrS, ytsS = train_test_split(X0S, y0S, test_size=0.5)
s_neuron_model = LinearRegression().fit(XtrS, ytrS)

ypredS = s_neuron_model.predict(XtsS)
s_neuron_RSS = np.dot((ytsS - ypredS).T, (ytsS - ypredS))
s_neuron_S = np.sum((ytsS - np.mean(ytsS))**2)
s_neuron_NRSS = s_neuron_RSS/s_neuron_S

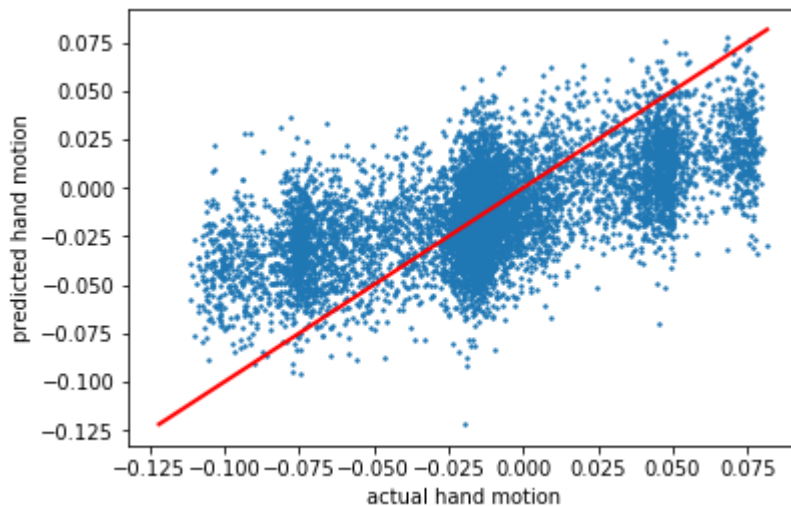
print('RRS persample = {}, and normalized RRS on test data = {}'.format(s_neuron_RSS/d, s_neuron_NRSS))

RRS persample = 0.17667182211720966, and normalized RRS on test data = 0.6098
317117945402
```

Create a scatter plot of the predicted vs. actual hand motion on the test data. On the same plot, plot the line where  $y_{ts\_hat} = y_{ts}$ .

```
In [128]: # TODO
x_y = np.linspace(min(min(ypredS),min(ytsS)),max(max(ypredS),max(ytsS)),100)
plt.plot(x_y,x_y,'r-',linewidth=2)
plt.xlabel('actual hand motion')
plt.ylabel('predicted hand motion')
plt.scatter(ytsS,ypredS, s = 2)
```

Out[128]: <matplotlib.collections.PathCollection at 0x1a1b8a7fa20>



## Using K-fold cross validation for the optimal number of neurons

In the above, we fixed  $d=50$ . We can use cross validation to try to determine the best number of neurons to use. Try model orders with  $d=10, 20, \dots, 190$ . For each value of  $d$ , use K-fold validation with 10 folds to estimate the test RSS. For a data set this size, each fold will take a few seconds to compute, so it may be useful to print the progress.

```

In [161]: import sklearn.model_selection

# Create a k-fold object
nfold = 10
kf = sklearn.model_selection.KFold(n_splits=nfold,shuffle=True)

# Model orders to be tested
dtest = np.arange(10,200,10)
nd = len(dtest)

# TODO.
RSS = np.zeros([nd, nfold])

for i in range(0, nd):
    X_d = X0[:,np.argsort(R_2)[-dtest[i]:]]
    cur_fold = 0

    for train_index, test_index in kf.split(X_d, y0):
        xtr ,ytr= X_d[train_index], y0[train_index]
        xts, yts = X_d[test_index], y0[test_index]
        temp_model = LinearRegression().fit(xtr, ytr)

        ypred = temp_model.predict(xts)
        RSS[i][cur_fold] += np.dot((yts - ypred).T, (yts - ypred))
        cur_fold += 1
    print("{} percent of mission complete".format(((i+1)/nd) * 100))

```

```

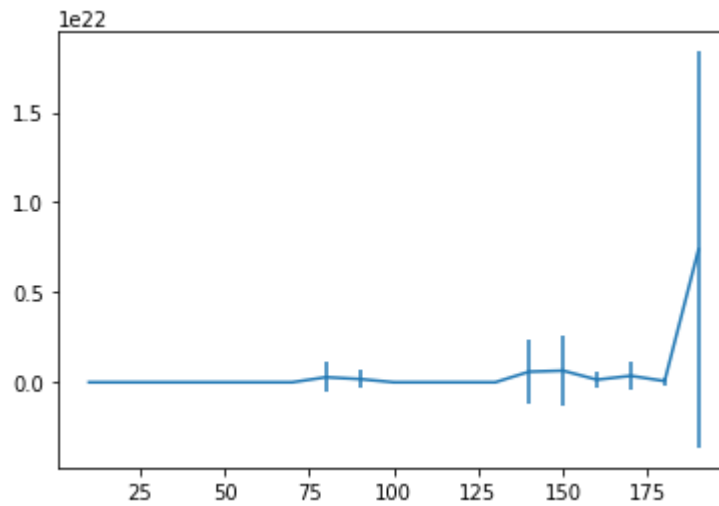
5.263157894736842 percent of mission complete
10.526315789473683 percent of mission complete
15.789473684210526 percent of mission complete
21.052631578947366 percent of mission complete
26.31578947368421 percent of mission complete
31.57894736842105 percent of mission complete
36.84210526315789 percent of mission complete
42.10526315789473 percent of mission complete
47.368421052631575 percent of mission complete
52.63157894736842 percent of mission complete
57.89473684210527 percent of mission complete
63.1578947368421 percent of mission complete
68.42105263157895 percent of mission complete
73.68421052631578 percent of mission complete
78.94736842105263 percent of mission complete
84.21052631578947 percent of mission complete
89.47368421052632 percent of mission complete
94.73684210526315 percent of mission complete
100.0 percent of mission complete

```

Compute the RSS test mean and standard error and plot them as a function of the model order  $d$  using the `plt.errorbar()` method.

```
In [166]: # TODO
RSS_mean = np.mean(RSS, axis = 1)
RSS_std = np.std(RSS, axis = 1)
plt.errorbar(dtest, RSS_mean, RSS_std)
```

Out[166]: <ErrorbarContainer object of 3 artists>



Find the optimal order using the one standard error rule. Print the optimal value of d and the mean test RSS per sample at the optimal d.

```
In [170]: # TODO
min_idx = np.argsort(RSS_mean)
min_mean = RSS_mean[min_idx[0]]
min_std = RSS_std[min_idx[0]]
possible_idx = list(filter(lambda x: RSS_mean[x] <= min_mean + min_std, min_idx
))

d_choose = np.min(possible_idx)

print("{} are the optimal model order chosen, and the mean test RRS = {}".format(
dtest[d_choose], RSS_mean[d_choose]/(y0.shape[0]/nfold)))
```

130 are the optimal model order chosen, and the mean test RRS = 0.00092413123  
28492335



## More Fun

You can play around with this and many other neural data sets. Two things that one can do to further improve the quality of fit are:

- Use more time lags in the data. Instead of predicting the hand motion from the spikes in the previous time, use the spikes in the last few delays.
- Add a nonlinearity. You should see that the predicted hand motion differs from the actual for high values of the actual. You can improve the fit by adding a nonlinearity on the output. A polynomial fit would work well here.

You do not need to do these, but you can try them if you like.

```

In [182]: '''
The std is too big as d increase
Check what's happened
'''

max_std_idx = np.argsort(RSS_std)
print(max_std_idx)
print(dtest[max_std_idx[-1]])

X_weird = X0[:,np.argsort(R_2)[-190:]]
Xtr, Xts, ytr, yts = train_test_split(X_weird, y0, test_size=0.5)

weird_model = LinearRegression().fit(Xtr, ytr)

ypred = weird_model.predict(Xts)

print(yts, ypred)

plt.xlabel('actual hand motion')
plt.ylabel('weird predicted hand motion')
plt.scatter(yts, ypred, s = 20)

'''
it seems that no matter what the actual result is, the prediction value is alw
ays zero
the model fits into a constant line, since the irrevelant information out numb
ers revelant information
'''

[ 5  9  6 12  2  4 11 10  3  0  1 17 15  8 16  7 13 14 18]
190
[-0.07589915 -0.05383698 -0.01675803 ... -0.09022686  0.04669881
 -0.00775485] [-0.07158467 -0.08567646  0.00985824 ... -0.07583984 -0.0003237
 3
 0.00844374]

```

```

Out[182]: '\nit seems that no matter what the actual result is, the prediction value is
always zero\n'

```

