lab03a_neural_partial

February 13, 2018

1 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. 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

1.1 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

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. 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 [3]: #TODO
        print(mat_dict.keys())
```

```
dict_keys(['__header__', '__version__', '__globals__', 'Publication', 'timeBase', 'spikes', 'ti
```

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 time bins $\times$ number of neurons. For the `handPos` data, we take the second component which is the position of monkey's hand.

```
In [4]: X0 = mat_dict['spikes'].T
        y0 = mat_dict['handPos'][0,:]
```

The `spikes` matrix will be a `nt x nneuron` matrix where `nt` is the number of time bins and `nneuron` 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 `nt` and `nneuron` and print the values.

```
In [5]: # TODO
        (nt, nneuron) = X0.shape
        print("\'nt\' is %d." % X0.shape[0])
        print("\'nneuron\' is %d." % X0.shape[1])
```

```
'nt' is 15536.
'nneuron' is 196.
```

Now extract the `time` variable from the `mat_dict` dictionary. Reshape this to a 1D array with `nt` 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 [6]: # TODO
        time = mat_dict['time'].reshape(-1, 1)
        tsamp = np.mean(np.diff(time, axis=0))
        ttotal = time[-1] - time[0]
        print(time)
        print("\'tsamp\' is %0.2f." % tsamp)
        print("\'ttotal\' is %0.2f." % ttotal)
```

```
[[ 12.591]
 [ 12.641]
 [ 12.691]
 ...
 [789.241]
 [789.291]
 [789.341]]
'tsamp' is 0.05.
'ttotal' is 776.75.
```

1.2 Linear fitting on all the neurons

First divide the data into training and test with approximately half the samples in each. Let `Xtr` and `ytr` denote the training data and `Xts` and `yts` denote the test data.

```
In [7]: # TODO
        ntr = nt // 2
        nts = nt - ntr

        Xtr = X0[:ntr]
        ytr = y0[:ntr]
        Xts = X0[ntr:]
        yts = y0[ntr:]
```

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 [8]: import sklearn.linear_model

        # TODO
        regr = sklearn.linear_model.LinearRegression()
        regr.fit(Xtr, ytr)
```

```
/usr/local/lib/python3.6/site-packages/scipy/linalg/basic.py:1226: RuntimeWarning: internal gel
  warnings.warn(mesg, RuntimeWarning)
```

```
Out[8]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Measure and print the normalized RSS on the test data.

```
In [9]: # TODO
        y_pred = regr.predict(Xts)
        RSS = np.mean((y_pred - yts)**2)/np.std(yts)**2
        print("Normalized RSS is %f." % RSS)
```

```
Normalized RSS is 0.465803.
```

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

1.3 Linear Fitting with Heuristic Model Selection

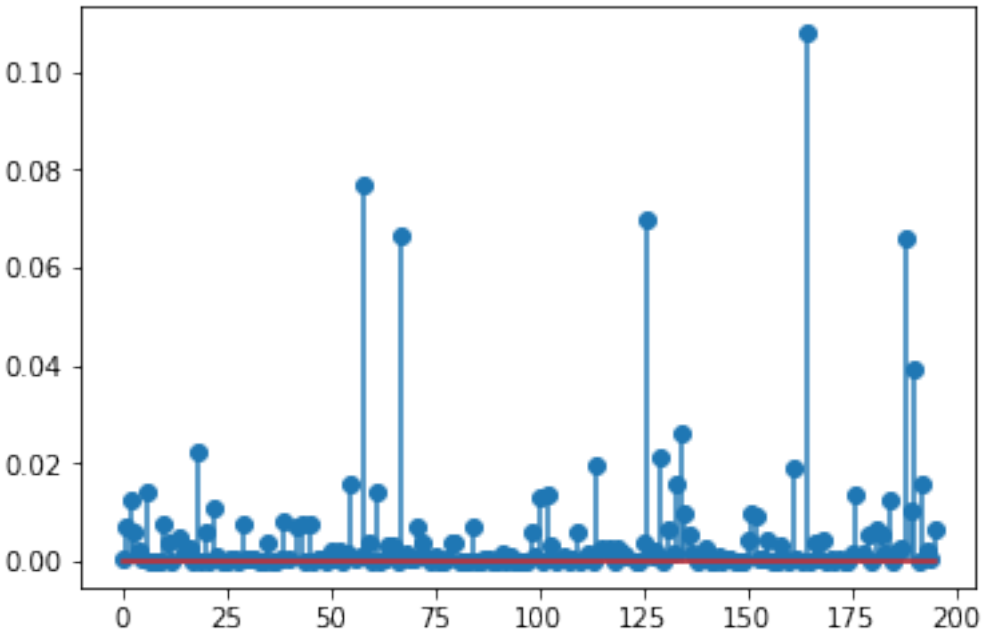
The above shows that we need a way to reduce the model complexity. One simple idea is to select only the neurons that individually have a high correlation with the output.

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.

```
In [10]: # TODO
         ym = np.mean(ytr)
         Xm = np.mean(Xtr, axis=0)
         syy = np.mean((ytr-ym)**2)
         sxx = np.mean((Xtr-Xm)**2, axis=0)
         sxy = np.mean((Xtr-Xm)*(ytr-ym)[:,None], axis=0)
         Rsq = sxy**2/sxx/syy
         plt.stem(Rsq.T)
         plt.show()
```

```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:7: RuntimeWarning: invalid value e
  import sys
```



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 `Isel`. Print the indices of the neurons with the 10 highest correlations.

```
In [11]: d = 50  # Number of neurons to use

         # TODO
         Isel = np.argsort(Rsq)[-d:]
         print("The neurons with the ten highest R^2 values are", Isel[-10:])
```

```
The neurons with the ten highest R^2 values are [ 67 126  58 164  41 105 122 177  13 139]
```

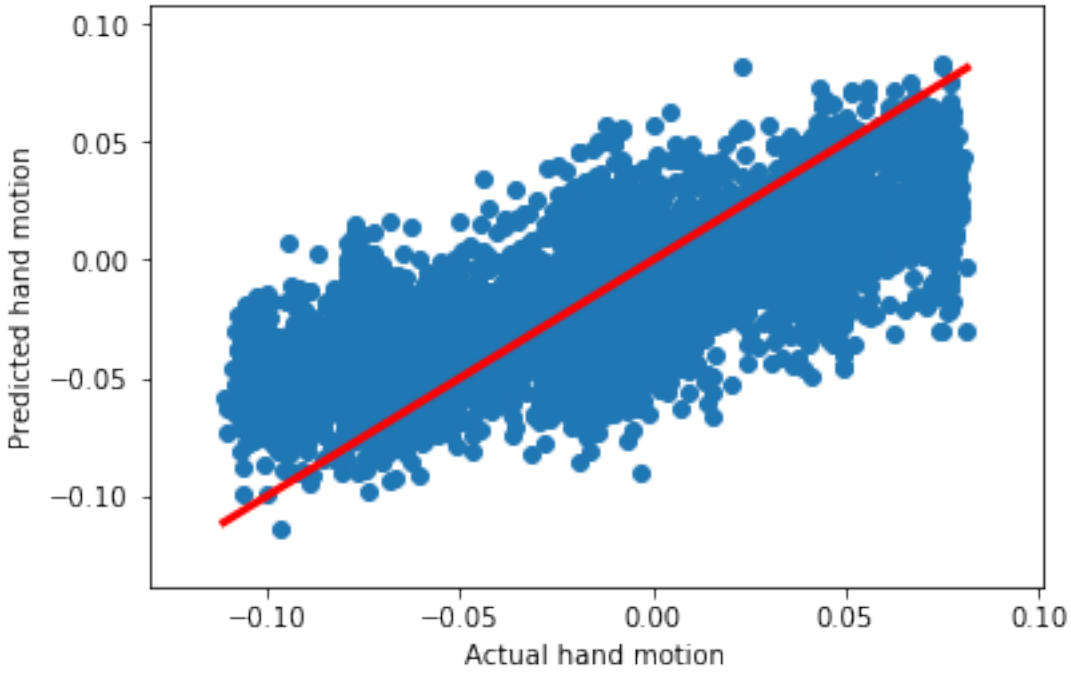
Fit a model using only the `d` neurons selected in the previous step and print both the test RSS per sample and the normalized test RSS.

```
In [12]: # TODO
         regr_d = sklearn.linear_model.LinearRegression()
         regr_d.fit(Xtr[:, Isel], ytr)
         y_pred_d = regr_d.predict(Xts[:, Isel])
         RSS_ps_d = np.mean((y_pred_d - yts)**2)
         RSS_d = RSS_ps_d/np.std(yts)**2
         print("Test RSS per sample is %f." % RSS_ps_d)
         print("Normalized test RSS is %f." % RSS_d)
```

```
Test RSS per sample is 0.000980.
Normalized test RSS is 0.504026.
```

Create a scatter plot of the predicted vs. actual hand motion on the test data. On the same plot, plot the line where `yts_hat = yts`.

```
In [13]: # TODO
         plt.scatter(yts, y_pred_d)
         plt.xlabel('Actual hand motion')
         plt.ylabel('Predicted hand motion')
         ymin = np.min(yts)
         ymax = np.max(yts)
         plt.plot([ymin, ymax], [ymin, ymax], 'r-', linewidth=3)
         plt.show()
```



1.4 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,...,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 [14]:
```
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.
from IPython.display import clear_output

RSSts = np.zeros((nd,nfold))

# Loop over the folds
for isplit, Ind in enumerate(kf.split(X0)):
```

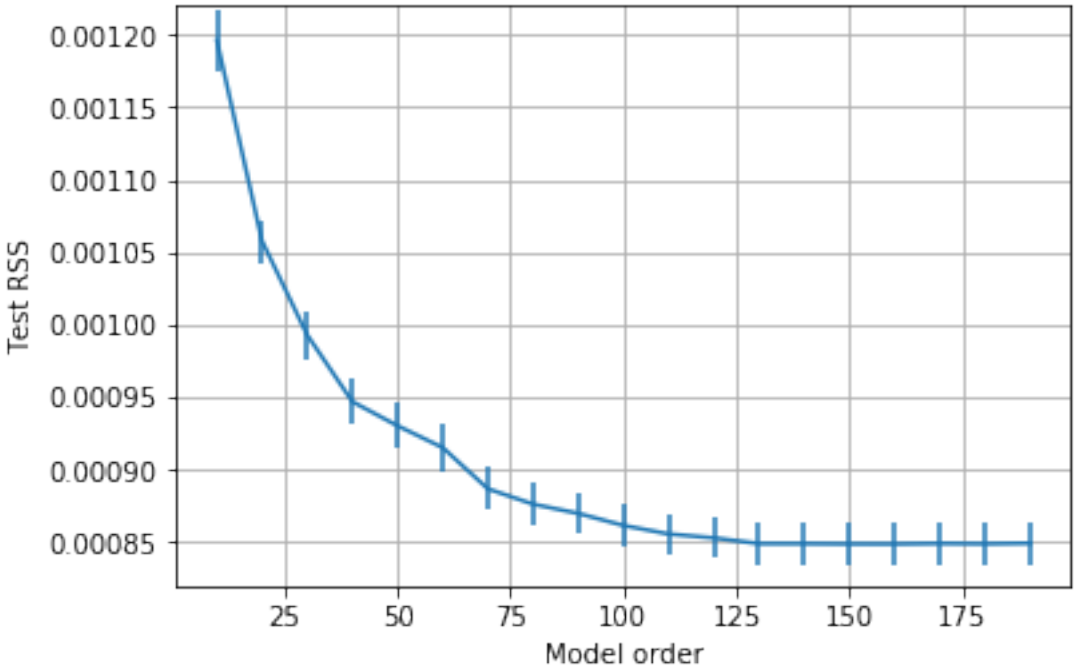
```
            i = round(isplit / nfold * 100)
            load_str = '>' * (i // 2) + ' ' * ((99 - i) // 2)

            clear_output(wait=True)
            print('\r' + load_str + '[%s%%]' % i)

            # Get the training data in the split
            Itr, Its = Ind
            #kf.split( ) returns Ind, which contains the indices to the training and testing
            Xtr = X0[Itr]
            ytr = y0[Itr]
            Xts = X0[Its]
            yts = y0[Its]

            # Loop over the model order
            for it, d in enumerate(dtest):
                ym = np.mean(ytr)
                Xm = np.mean(Xtr)
                syy = np.mean((ytr-ym)**2)
                sxx = np.mean((Xtr-Xm)**2, axis=0)
                sxy = np.mean((Xtr-Xm)*(ytr-ym)[:,None], axis=0)
                Rsq = sxy**2/sxx/syy
                Isel = np.argsort(Rsq)[-d:]

                # Fit data on training data
                regr_it = sklearn.linear_model.LinearRegression()
                regr_it.fit(Xtr[:, Isel], ytr)

                # Measure RSS on test data
                yhat = regr_it.predict(Xts[:,Isel])
                RSSts[it, isplit] = np.mean((yhat-yts)**2)

        clear_output()
        print("Done!")
```

```
Done!
```

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 [15]: # TODO
        RSS_mean = np.mean(RSSts, axis=1)
        RSS_std = np.std(RSSts, axis=1) / np.sqrt(nfold - 1)
        plt.errorbar(dtest, RSS_mean, yerr=RSS_std, fmt='-')
        plt.ylim([0.00082, 0.00122])
        plt.xlabel('Model order')
        plt.ylabel('Test RSS')
```

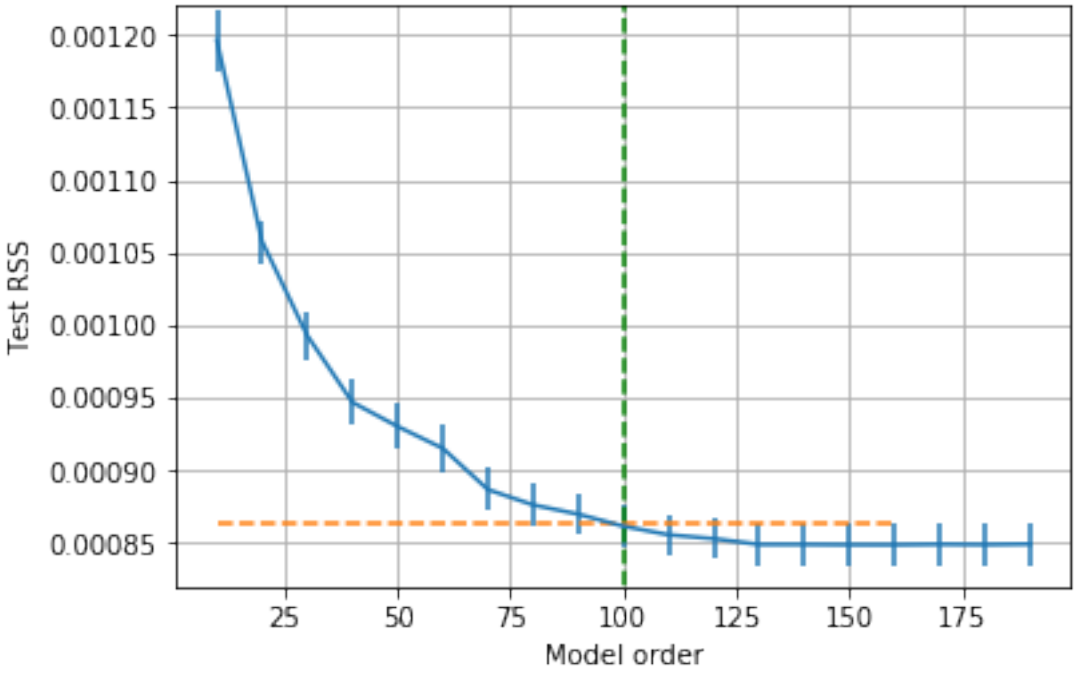
```
plt.grid()
plt.show()
```



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 [16]: # TODO
         # Find the minimum RSS target
         imin = np.argmin(RSS_mean)
         RSS_tgt = RSS_mean[imin] + RSS_std[imin]

         # Find the lowest model order below the target
         I = np.where(RSS_mean <= RSS_tgt)[0]
         iopt = I[0]
         dopt = dtest[iopt]

         plt.errorbar(dtest, RSS_mean, yerr=RSS_std, fmt='-')

         # Plot the line at the RSS target
         plt.plot([dtest[0],dtest[imin]], [RSS_tgt, RSS_tgt], '--')

         # Plot the line at the optimal model order
         plt.plot([dopt,dopt], [0,0.5], 'g--')

         plt.ylim([0.00082, 0.00122])
         plt.xlabel('Model order')
```

```
plt.ylabel('Test RSS')
plt.grid()
plt.show()

# Print results
print("The estimated model order is %d" % dopt)
```




```
The estimated model order is 100
```

1.5 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.