

# Lab: Model Order 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 model 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:

- Represent neural time-series data in arrays
- Load data from a pickle file
- Describe and fit memoryless linear models
- Describe and fit linear time-series models with delays
- Fit linear models with multiple target outputs
- Select the optimal delay 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 motion from the neural signals from the motor cortex.

We first load the key packages.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pickle

from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
```

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. However, the raw data files can be quite large. To make the lab easier, the [Kording lab](http://kordinglab.com/) (<http://kordinglab.com/>) at UPenn has put together an excellent [repository](https://github.com/KordingLab/Neural_Decoding) ([https://github.com/KordingLab/Neural\\_Decoding](https://github.com/KordingLab/Neural_Decoding)) where they have created simple pre-processed versions of the data. You can download the file `example_data_s1.pickle` from the [Dropbox link](https://www.dropbox.com/sh/n4924ipcfjqc0t6/AAD0v9JYMUBK1tlg9P71gSSra/example_data_s1.pickle?dl=0) ([https://www.dropbox.com/sh/n4924ipcfjqc0t6/AAD0v9JYMUBK1tlg9P71gSSra/example\\_data\\_s1.pickle?dl=0](https://www.dropbox.com/sh/n4924ipcfjqc0t6/AAD0v9JYMUBK1tlg9P71gSSra/example_data_s1.pickle?dl=0)). Alternatively, you can directly run the following code. This may take a little while to download since the file is 26 MB.

In [2]:

```
fn_src = 'https://www.dropbox.com/sh/n4924ipcfjqc0t6/AAD0v9JYMUBK1tlg9P71gSSra/example_data_s1.pickle?dl=1'
fn_dst = 'example_data_s1.pickle'

import os
from six.moves import urllib

if os.path.isfile(fn_dst):
    print('File %s is already downloaded' % fn_dst)
else:
    urllib.request.urlretrieve(fn_src, fn_dst)
```

The file is a *pickle* data structure, which is a package to serialize python objects into data files. Once you have downloaded the file, you can run the following command to retrieve the data from the pickle file.

In [3]:

```
with open('example_data_s1.pickle', 'rb') as fp:
    X,y = pickle.load(fp)
```

The matrix  $X$  is matrix of spike counts where  $X[i, j]$  is the number of spikes from neuron  $j$  in time bin  $i$ . The matrix  $y$  has two columns:

- $y[i, 0]$  = velocity of the monkey's hand in the x-direction
- $y[i, 1]$  = velocity of the monkey's hand in the y-direction Our goal will be to predict  $y$  from  $X$ .

Each time bin represent  $tsamp=0.05$  seconds of time. Using  $X.shape$  and  $y.shape$  compute and print:

- $nt$  = the total number of time bins
- $nneuron$  = the total number of neurons
- $nout$  = the total number of output variables to track = number of columns in  $y$
- $ttotal$  = total time of the experiment is seconds.

In [5]:

```
tsamp = 0.05 # sampling time in seconds

# TODO
nt = X.shape[0]
nneuron = X.shape[1]
nout = y.shape[1]
ttotal = nt*tsamp

print('nt = {}, and nneuron = {}, nout = {}, ttotal = {}'.format(nt, nneuron, nout, ttotal))

nt = 61339, and nneuron = 52, nout = 2, ttotal = 3066.95000
00000003
```

## Fitting a Memoryless Linear Model

Let's first try a simple linear regression model to fit the data.

First, use the `train_test_split` function to split the data into training and test. Let  $X_{tr}, y_{tr}$  be the training data set and  $X_{ts}, y_{ts}$  be the test data set. Use `test_size=0.33` so 1/3 of the data is used for test.

In [6]:

```
from sklearn.model_selection import train_test_split

# TODO
Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=0.33)
```

Now, fit a linear model using  $X_{tr}, y_{tr}$ . Make a prediction  $y_{hat}$  using  $X_{ts}$ . Compare  $y_{hat}$  to  $y_{ts}$  to measure  $rsq$ , the  $R^2$ . You can use the `r2_score` method. Print the  $rsq$  value. You should get  $rsq$  of around 0.45.

In [13]:

```
# TODO
naive_model = LinearRegression().fit(Xtr, ytr)
yhat = naive_model.predict(Xts)
# yhat = ...
ymean = np.mean(yts, axis = 0)
SS_tot = np.sum((yts - ymean.reshape(1, ymean.shape[0]))**2)
SS_res = np.sum((yts - yhat)**2)
rsq = 1 - SS_res/SS_tot
print('r2 score is = {}'.format(rsq))
```

r2 score is = 0.4679776344660658

It is useful to plot the predicted vs. true values. Since we have two outputs, create two subplots using the `plt.subplot()` command. In plot `i=0,1`, plot `yhat[:,i]` vs. `yts[:,i]` with a scatter plot. Label the axes of the plots. You may also use the command:

```
plt.figure(figsize=(10,5))
```

to make the figures a little larger.

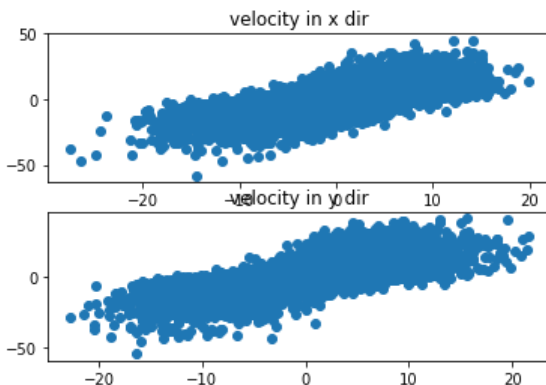
In [16]:

```
# TODO
plt.figure(figsize=(10,5))
f, ax = plt.subplots(2, 1)
i = [0,1]
ax[0].scatter(yhat[:,0], yts[:,0])
ax[0].set_title('velocity in x dir')
ax[1].scatter(yhat[:,1], yts[:,1])
ax[1].set_title('velocity in y dir')
```

Out[16]:

Text(0.5,1,'velocity in y dir')

<Figure size 720x360 with 0 Axes>



## Fitting Models with Delay

One way we can improve the model accuracy is to use a delayed version of the features. Specifically, the model we used above mapped the features

$$\hat{y}[i,k] = \sum_{j=0}^{p-1} X[i,j] * w[j,k] + b[k]$$

where  $p$  is the number of features and  $w[j,k]$  is a matrix of coefficients. In this model,  $\hat{y}[i,:]$  at time  $i$  was only dependent on the inputs  $X[i,:]$  at time  $i$ . In signal processing, this is called a *memoryless* model. However, in many physical systems, such as those that arise in neuroscience, there is a delay between the inputs  $X[i,:]$  and the outputs  $y[i]$ . For such cases, we can use a model of the form,

$$\hat{y}[i+d,k] = \sum_{k=0}^d \sum_{j=0}^{p-1} \sum_{m=0}^d X[i+m,j] * w[j,m,k] + b[k]$$

where  $W$  is a 3-dim array of coefficients where:

$w[j,m,k]$  is the influence of the input  $X[i+m,j]$  onto output  $y[i+d,k]$

In signal processing, this model is called an *FIR* filter and  $w[j,:,k]$  is the *impulse response* from the  $j$ -th input to the  $k$ -th output. The point is that the output at time  $i+d$  depends on the inputs at times  $i, i+1, \dots, i+d$ . Hence, it depends on the last  $d+1$  time steps, not just the most recent time.

To translate this into a linear regression problem, complete the following function that creates a new feature and target matrix where:

```
Xdly[i,:] has the rows X[i,:], X[i+1,:], ..., X[i+dly,:]
ydly[i,:] = y[i+dly,:]
```

Thus,  $Xdly[i,:]$  contains all the delayed features for the target  $\hat{y}$ . Note that if  $X$  is  $n \times p$  then  $Xdly$  will be  $n-dly \times (dly+1)*p$ .

In [44]:

```
def create_dly_data(X,y,dly):
    """
    Create delayed data
    p is number of features
    n is number of data
    """
    Xdly = np.zeros([nt - dly, (dly+1)*nneuron])
    ydly = np.zeros([nt - dly, nout])
    for i in range (nt - dly):
        Xdly[i,:] = X[i:i+dly+1,:].reshape(-1)
        ydly[i,:] = y[i+dly,:]

    return Xdly, ydly
```

Now fit an linear delayed model with dly=6 additional delay lags. That is,

- Create delayed data Xdly,ydly=create\_dly\_data(X,y,dly=6)
- Split the data into training and test as before
- Fit the model on the training data
- Measure the  $R^2$  score on the test data

If you did this correctly, you should get a new  $R^2$  score around 0.69. This is significantly better than the memoryless models.

In [45]:

```
# TODO
Xdly, ydly = create_dly_data(X, y, 6)

Xtr, Xts, ytr, yts = train_test_split(Xdly, ydly, test_size=0.33)

dly_model = LinearRegression().fit(Xtr, ytr)
yhat = dly_model.predict(Xts)
# yhat = ...

ymean = np.mean(yts, axis = 0)
SS_tot = np.sum((yts - ymean.reshape(1, ymean.shape[0]))**2)
SS_res = np.sum((yts - yhat)**2)
rsq = 1 - SS_res/SS_tot
print('r2 score is = {}'.format(rsq))

r2 score is = 0.687405233105947
```

Plot the predicted vs. true values as before. You should visually see a better fit.

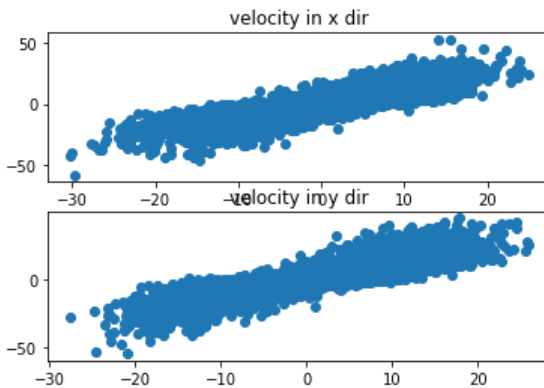
In [36]:

```
# TODO
# TODO
plt.figure(figsize=(10,5))
f, ax = plt.subplots(2, 1)
i = [0,1]
ax[0].scatter(yhat[:,0], yts[:,0])
ax[0].set_title('velocity in x dir')
ax[1].scatter(yhat[:,1], yts[:,1])
ax[1].set_title('velocity in y dir')
```

Out[36]:

Text(0.5,1,'velocity in y dir')

<Figure size 720x360 with 0 Axes>



*Note:* Fitting an FIR model with the above method is very inefficient when the number of delays,  $dly$  is large. In the above method, the number of columns of  $X$  grows from  $p$  to  $(dly+1)*p$  and the computations become expensive with  $dly$  is large. We will describe a much faster way to fit such models using gradient descent when we talk about convolutional neural networks.

## Selecting the Optimal Delay via Model Order Selection

In the previous example, we fixed  $dly=6$ . We can now select the optimal delay using model order selection. Since we have a large number of data samples, it turns out that the optimal model order uses a very high delay. Using the above fitting method, the computations take too long. So, to simplify the lab, we will first just pretend that we have a very limited data set.

Compute  $X_{red}$  and  $y_{red}$  by taking the first  $n_{red}=6000$  samples of the data  $X$  and  $y$ . This is about 10% of the overall data.

In [39]:

```
nred = 6000
```

```
# TODO
```

```
Xred = X[:nred, :]
```

```
yred = y[:nred, :]
```

We will look at model orders up to  $d_{\max}=15$ . Create a delayed matrix data,  $X_{dly}, y_{dly}$  from the reduced data,  $X_{red}, y_{red}$  using `create_dly_data` with `dly=dmax`.

In [46]:

```
dmax = 15
```

```
# TODO
```

```
Xdly, ydly = create_dly_data(X, y, dmax)
```

Complete the following code to implement K-fold cross validation with `nfold=5` and values of delays `dtest = [0, 1, ..., dmax]`.



In [64]:

```
import sklearn.model_selection

# Number of folds
nfold = 5

# TODO: Create a k-fold object
kf = sklearn.model_selection.KFold(nfold)

# TODO: Model orders to be tested
nd = dmax + 1
dtest = np.arange(nd) #what the hell is vector

# TODO.
# Initialize a matrix Rsq to hold values of the  $R^2$  across the model orders and folds.
Rsq = np.zeros([nfold, nd])

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

    print("fold = %d " % isplit)

    # Get the training data in the split
    Itr, Its = Ind

    for it, d in enumerate(dtest):
        # TODO:
        # Xdly1 = Xdly with the columns corresponding to only the `d+1`
        # most recent times.
        Xdly1 = Xdly[:, -(d+1)*nneuron:]

        # TODO
        # Split the data (Xdly1,ydly) into training and test
        Xtr = Xdly1[Itr]
        ytr = ydly[Itr]
        Xts = Xdly1[Its]
        yts = ydly[Its]

        # TODO: Fit data on training data
        dly_model = LinearRegression().fit(Xtr, ytr)
        yhat = dly_model.predict(Xts)

        ymean = np.mean(yts, axis = 0)
        SS_tot = np.sum((yts - ymean.reshape(1, ymean.shape[0]))**2)
        SS_res = np.sum((yts - yhat)**2)
        rsq = 1 - SS_res/SS_tot

        Rsq[isplit,it] = rsq

    # TODO: Measure the  $R^2$  value on test data and store in the matrix Rsq
```

```
fold = 0
fold = 1
fold = 2
fold = 3
fold = 4
```

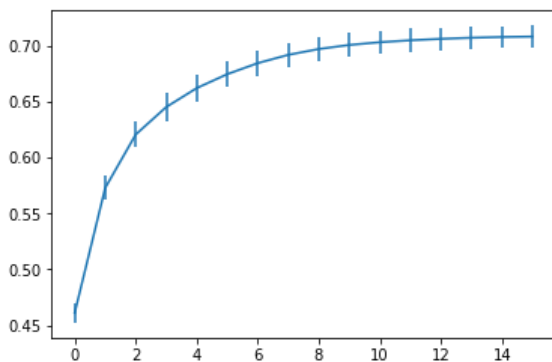
Compute the mean and standard error of the  $R^2$  values as a function of the model order  $d$ . Use a `plt.errorbar` plot. Label your axes.

In [69]:

```
# TODO
MeanRsqr = np.mean(Rsq, axis = 0)
StdRsqr = np.std(Rsq, axis = 0)
plt.errorbar(dtest, MeanRsqr, StdRsqr)
```

Out[69]:

<ErrorbarContainer object of 3 artists>



Find the optimal order  $d$  with the normal rule (i.e. highest test  $R^2$ )

In [71]:

```
# TODO
opt_d = np.argmax(MeanRsqr)
opt_std = StdRsqr[opt_d]
print(opt_d)
```

15

Now find the optimal model order via the one SE rule (i.e. highest test  $R^2$  within one SE)

In [74]:

```
# TODO
possible_idx = list(filter(lambda x: MeanRsq[x] >= MeanRsq[opt_d]- opt_s
td, np.arange(nd)))
choose_d = np.min(possible_idx)
print("best model order is {}".format(choose_d))
```

best model order is 9