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 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:

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

The last stage of the lab uses LASSO estimation for model selection. If you are doing this part of the lab, you should review the concepts in LASSO demonstration (./prostate.ipynb) on the prostate cancer dataset.

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

Out[3]:
```
dict_keys(['__header__', '__version__', '__globals__', 'Publication', 'timeBas
e', 'spikes', 'time', 'handVel', 'handPos', 'target', 'startBins', 'targets',
 'startBinned'])
```

We extract two variables, `spikes` and `handVel`, from the dictionary `mat_dict`, which represent the recorded spikes per neuron and the hand velocity. We take the transpose of the spikes data so that it is in the form time bins $\times$ number of neurons. For the `handVel` data, we take the first component which is the motion in the x-direction.

In [4]:
```
X0 = mat_dict['spikes'].T
y0 = mat_dict['handVel'][0,:]
print(X0)
```

```
[[1 0 2 ..., 2 0 2]
 [3 1 1 ..., 0 0 0]
 [1 0 1 ..., 0 0 3]
 ...,
 [0 2 0 ..., 0 0 3]
 [0 3 0 ..., 0 0 1]
 [0 2 0 ..., 0 0 2]]
```

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=X0.shape[0]
nneuron=X0.shape[1]
print(nt)
print(nneuron)
```

```
15536
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'][0,:]
print(time)
tsamp=time[1]-time[0]
print(tsamp)
ttotal=time[-1]-time[0]
print(ttotal)
print(ttotal/tsamp+1)
```

```
[  12.591   12.641   12.691 ...,  789.241  789.291  789.341]
0.05
776.75
15536.0
```

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
# Xtr = ...
Xtr=X0[:int(len(X0)/2)]
# ytr = ...
ytr=y0[:int(len(y0)/2)]
# Xts = ...
Xts=X0[int(len(X0)/2):]
# yts = ...
yts=y0[int(len(y0)/2):]
print(Xtr.shape)
print(Xts.shape)
print(7768*2)
print(Xtr[7767])
```

```
(7768, 196)
(7768, 196)
15536
[0 0 0 0 6 0 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 2 2 2 0 0 0 0 0 1 4 0 0 0 0 1 3
 0 0 0 0 0 1 4 1 1 0 0 0 0 0 0 1 0 0 1 0 0 2 1 0 3 0 0 1 4 1 2 0 0 0 6 0 0
 0 1 0 2 0 0 0 0 0 0 0 0 0 2 0 0 1 0 0 1 0 0 0 3 6 1 0 0 1 2 0 0 0 0 3 0 0
 0 1 0 0 0 0 3 0 0 4 0 0 0 0 2 1 0 1 4 0 0 1 0 1 2 3 0 0 0 3 5 0 0 1 2 0 2
 2 1 0 1 3 2 1 1 0 0 4 1 0 2 1 0 2 0 0 2 3 0 1 0 7 0 0 0 1 0 1 4 0 0 4 0 2
 0 1 2 6 0 3 0 0 0 0 2]
```

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)
```

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
yts_pred = regr.predict(Xts)
print(yts_pred)
RSS_ts = np.mean((yts_pred-yts)**2)/(np.std(yts)**2)
print("RSS per sample = {0:f}".format(RSS_ts))
print(Xts.shape)
print(Xts[105])
```

```
[ 0.04627061  0.06296143  0.08662423 ...,  0.00919423  0.03065438
  0.00755976]
RSS per sample = 152649046839368068628 48.000000
(7768, 196)
[0 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 4 0 0 1 0 0 0 0 0 2 2 0 0 0 0 1 1
 0 0 0 0 0 0 2 3 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 3 1 0 0 0 0 0 5 1 0
 0 0 0 0 1 2 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 6 0 0 0 1 1 0 0 1 0 0 0 0
 0 1 0 1 0 1 3 0 0 2 0 0 0 0 2 2 0 1 0 0 0 2 1 0 2 2 0 0 0 3 4 0 0 0 2 0 2
 2 0 0 2 0 4 1 1 0 0 2 0 0 1 1 0 0 0 2 4 2 0 2 0 3 0 0 1 3 0 3 0 0 1 1 1 2
 0 2 0 5 1 2 0 1 1 0 1]
```

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.

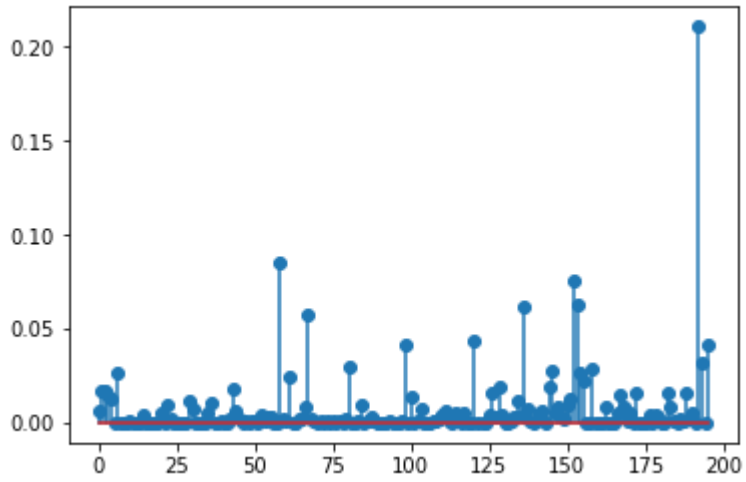
Write code which computes the coefficient of determination, R^2_k, for each neuron k. Plot the R^2_k values.

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 [10]:
```
# TODO
# Rsq = ...
# plt.stem(...)
nsamp, natt = X0.shape
ym = np.mean(y0)
syy = np.mean((y0-ym)**2)
Rsq = np.zeros(natt)
#beta0 = np.zeros(natt)
#beta1 = np.zeros(natt)
for k in range(natt):
    xm = np.mean(X0[:,k])
    sxy = np.mean((X0[:,k]-xm)*(y0-ym))
    sxx = np.mean((X0[:,k]-xm)**2)
#    beta1[k] = sxy/sxx
#    beta0[k] = ym - beta1[k]*xm
    Rsq[k] = (sxy)**2/sxx/syy

#    print("{0:2d}  Rsq={1:f}".format(k,Rsq[k]))
plt .stem(Rsq)
```

```
/home/ky935/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:16: Run
timeWarning: invalid value encountered in double_scalars
  app.launch_new_instance()
```

Out[10]: <Container 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=100` neurons with the highest R^2_k value. Put the `d` indices into an array `Isel`. Print the indices of the neurons with the 10 highest correlations.

In [11]:
```
d = 100  # Number of neurons to use

# TODO
# Isel = ...
# print("The neurons with the ten highest R^2 values = ...)
for i in range(natt):
    if np.isnan(Rsq[i]):
        Rsq[i]=0
#Rsq=Rsq[~np.isnan(Rsq)]
Isel=np.argsort(Rsq)
print("The neurons with the ten highest R^2 values =")
for i in range(10):
#    print(Rsq[Isel[-(i+1)]])
    print(Isel[-(i+1)])
```

```
The neurons with the ten highest R^2 values =
192
58
152
153
136
67
120
195
98
193
```

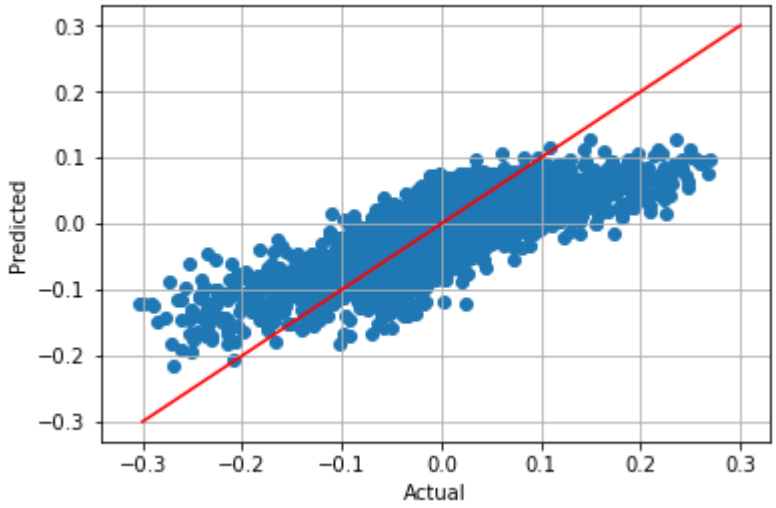
Fit a model using only the d neurons selected in the previous step and print both the test RSS per sample and the normalized test RSS.

In [12]:
```
# TODO
Isel=Isel[::-1]
#print(Isel)
#print(Xtr[:,(Isel[:d])])
Xtr[:,(Isel[:d])].shape
regr.fit(Xtr[:,(Isel[:d])],ytr)
yts_predd = regr.predict(Xts[:,(Isel[:d])])
RSS_tsd = np.mean((yts_predd-yts)**2)/(np.std(yts)**2)
print("RSS per sample = {0:f}".format(RSS_tsd))
```

```
RSS per sample = 0.496102
```

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,yts_predd)
plt.plot([-0.3,0.3],[-0.3,0.3],'r')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.grid()
```



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

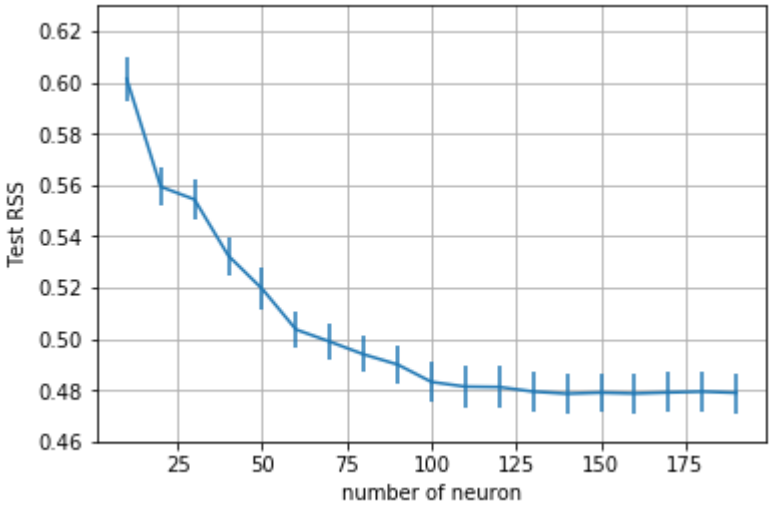
In the above, we fixed `d=100`. We can use cross validation to try to determine the best number of neurons to use. Try model orders with `d=10,20,...,190`. For each value of `d`, use K-fold validation with 10 folds to estimate the test RSS. For a data set this size, each fold will take a few seconds 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.
RSSts = np.zeros((nd,nfold))
for isplit, Ind in enumerate(kf.split(X0)):

    # Get the training data in the split
    Itr, Its = Ind
    x_tr = X0[Itr]
    y_tr = y0[Itr]
    x_ts = X0[Its]
    y_ts = y0[Its]
#    print(x_tr.shape)
#    print(x_ts.shape)
    for it, d in enumerate(dtest):

        # Fit data on training data
        beta_hat = regr.fit(x_tr[:,(Isel[:d])],y_tr)

        # Measure RSS on test data
        yhat = regr.predict(x_ts[:,(Isel[:d])])
        RSSts[it,isplit] =  np.mean((yhat-y_ts)**2)/(np.std(y_ts)**2)
#        print(RSSts[it,isplit])
#        print(yhat-y_ts)
```

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(0.46,0.63)
plt.xlabel('number of neuron')
plt.ylabel('Test RSS')
plt.grid()
print(RSS_mean)
```

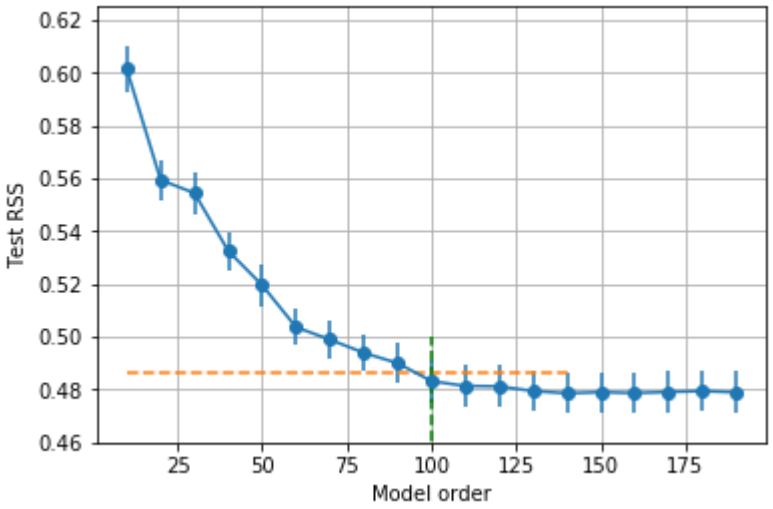
```
[ 0.601422    0.55942319  0.55442488  0.5324841   0.51953615  0.50365569
  0.49893299  0.49396337  0.49001066  0.48321247  0.48138864  0.48122873
  0.47946627  0.47861823  0.47903814  0.47870765  0.47911536  0.47946147
  0.47897225]
```



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
imin = np.argmin(RSS_mean)


# 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='o-')

# 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(0.46,0.625)
plt.xlabel('Model order')
plt.ylabel('Test RSS')
plt.grid()

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

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



Using LASSO regression

Instead of using the above heuristic to select the variables, we can use LASSO regression.

First use the `preprocessing.scale` method to standardize the data matrix `X0`. Store the standardized values in `Xs`. You do not need to standardize the response. For this data, the `scale` routine may throw a warning that you are converting data types. That is fine.

In [17]:
```
from sklearn import preprocessing

# TODO
Xs = sklearn.preprocessing.scale(X0)
ys = sklearn.preprocessing.scale(y0)
```

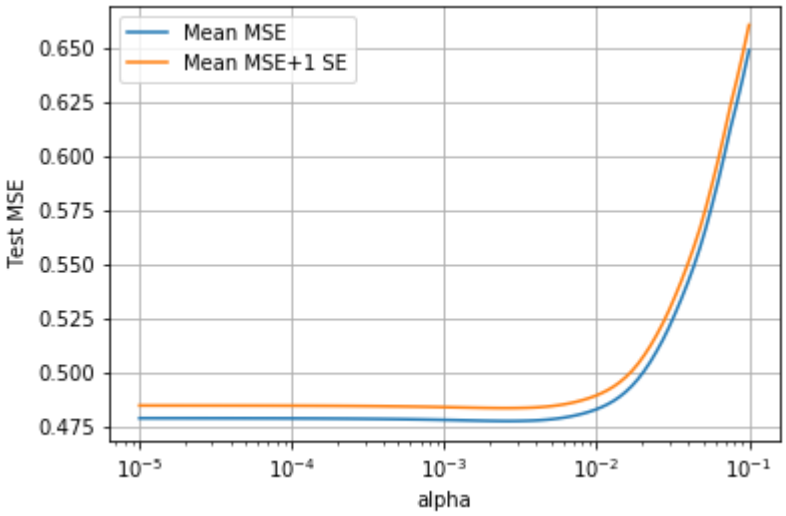
```
/home/ky935/anaconda3/lib/python3.6/site-packages/sklearn/utils/validation.py:4
29: DataConversionWarning: Data with input dtype uint8 was converted to float64
by the scale function.
  warnings.warn(msg, _DataConversionWarning)
```

Now, use the LASSO method to fit a model. Use cross validation to select the regularization level `alpha`. Use `alpha` values logarithmically spaced from `1e-5` to `0.1`, and use 10 fold cross validation.

In [18]:
```
# TODO
from sklearn import linear_model
# Create a k-fold cross validation object
nfold = 10
kf = sklearn.model_selection.KFold(n_splits=nfold,shuffle=True)

# Create the LASSO model.  We use the `warm start` parameter so that the fit will
# This speeds up the fitting.
model = linear_model.Lasso(warm_start=True)

# Regularization values to test
nalpha = 100
alphas = np.logspace(-5,-1,nalpha)

# MSE for each alpha and fold value
mse = np.zeros((nalpha,nfold))
for ifold, ind in enumerate(kf.split(X0)):


    # Get the training data in the split
    Itr,Its = ind
    X_tr = Xs[Itr,:]
    y_tr = ys[Itr]
    X_ts = Xs[Its,:]
    y_ts = ys[Its]

    # Compute the lasso path for the split
    for ia, a in enumerate(alphas):

        # Fit the model on the training data
        model.alpha = a
        model.fit(X_tr,y_tr)

        # Compute the prediction error on the test data
        y_ts_pred = model.predict(X_ts)
        mse[ia,ifold] = np.mean((y_ts_pred-y_ts)**2)
```

Plot the mean test RSS and test RSS standard error with the `plt.errorbar` plot.

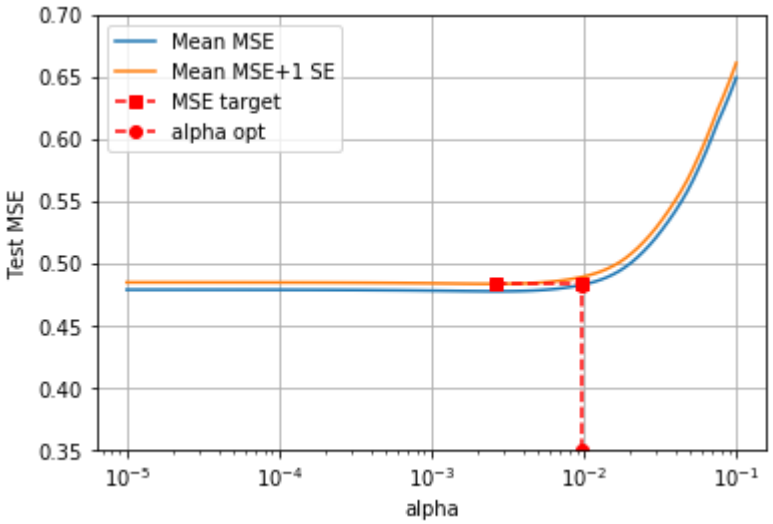
In [19]:
```
# TODO
# Compute the mean and standard deviation over the different folds.
mse_mean = np.mean(mse,axis=1)
mse_std = np.std(mse,axis=1) / np.sqrt(nfold-1)

# Plot the mean MSE and the mean MSE + 1 std dev
plt.semilogx(alphas, mse_mean)
plt.semilogx(alphas, mse_mean+mse_std)
plt.legend(['Mean MSE', 'Mean MSE+1 SE'],loc='upper left')
plt.xlabel('alpha')
plt.ylabel('Test MSE')
plt.grid()
plt.show()
```



Find the optimal `alpha` and mean test RSS using the one standard error rule.

In [20]:
```
# TODO
# Find the minimum MSE and MSE target
imin = np.argmin(mse_mean)
mse_tgt = mse_mean[imin] + mse_std[imin]
alpha_min = alphas[imin]

# Find the least complex model with mse_mean < mse_tgt
I = np.where(mse_mean < mse_tgt)[0]
iopt = I[-1]
alpha_opt = alphas[iopt]
print("Optimal alpha = %f" % alpha_opt)

# Plot the mean MSE and the mean MSE + 1 std dev
plt.semilogx(alphas, mse_mean)
plt.semilogx(alphas, mse_mean+mse_std)

# Plot the MSE target
plt.semilogx([alpha_min,alpha_opt], [mse_tgt,mse_tgt], 'rs--')

# Plot the optimal alpha line
plt.semilogx([alpha_opt,alpha_opt], [0.35,mse_mean[iopt]], 'ro--')

plt.legend(['Mean MSE', 'Mean MSE+1 SE', 'MSE target','alpha opt'],loc='upper lef
plt.xlabel('alpha')
plt.ylabel('Test MSE')
plt.ylim([0.35,0.7])
plt.grid()
plt.show()
```

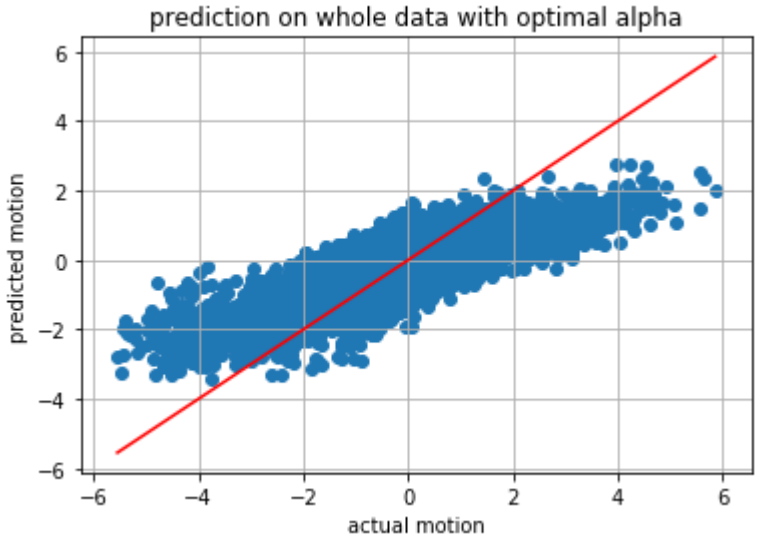
```
Optimal alpha = 0.009770
```



Using the optimal alpha, recompute the predicted response variable on the whole data. Plot the predicted vs. actual values.

In [21]:
```
# TODO
model.alpha = alpha_opt
model.fit(Xs,ys)
y_pred = model.predict(Xs)
ymean = np.mean(ys)
RSS_normalized = (np.mean((y_pred-ys)**2)/(np.mean((ymean-ys)**2)))
print('Normalized RSS for LASSO model is %f' %RSS_normalized)

line = np.linspace(np.min(ys), np.max(ys), 1000)
plt.plot(line, line, 'r')
plt.scatter(ys, y_pred)
plt.title('prediction on whole data with optimal alpha')
plt.xlabel('actual motion')
plt.ylabel('predicted motion')
plt.grid()
plt.show()
```

```
Normalized RSS for LASSO model is 0.475431
```



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 [ ]:
```
```