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(['startBins', 'handPos', 'targets', 'time', 'handVel', 'timeBase',
 '__globals__', 'spikes', 'Publication', '__header__', 'startBinned', 'targe
t', '__version__'])
```

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,:]
```

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("Num neurons = {0:d}".format(nneuron))
print("Num time = {0:d}".format(nt))
```

```
Num neurons = 196
Num time = 15536
```

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']
time = time.ravel()
tsamp = time[1]-time[0]
print("Sampling period (secs)= {0:f}".format(tsamp))
print("Total time (secs) =     {0:f}".format(tsamp*nt))
```

```
Sampling period (secs)= 0.050000
Total time (secs) =     776.800000
```

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 = ...
# ytr = ...
# Xts = ...
# yts = ...

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

# Create linear regression object
regr = sklearn.linear_model.LinearRegression()

# Fit the model
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]:
```
yts_hat = regr.predict(Xts)

RSS = np.mean((yts_hat-yts)**2)/(np.std(yts)**2)
print(RSS)
```

```
1.88937984291e+21
```

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

Linear Fitting with Heuristic Model Selection

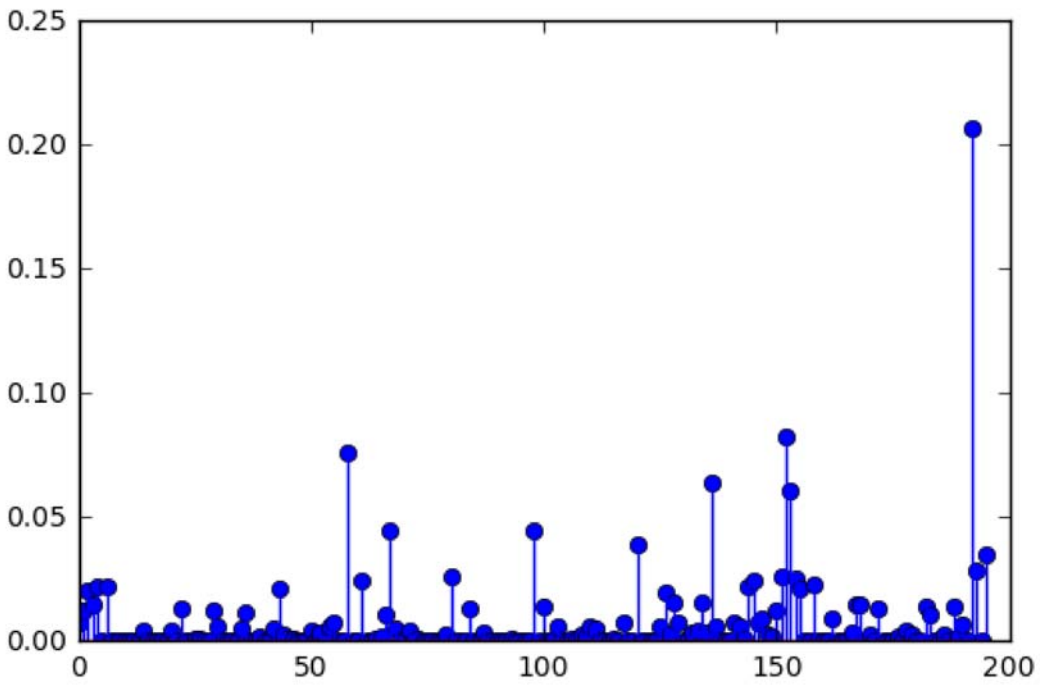
The above shows that we need a way to reduce the model complexity. One simple idea is to select only the neurons that individually have a high correlation with the output.

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(...)

# This is a solution with python broadcasting to avoid the for loops

# Compute the means
xm = np.mean(Xtr,axis=0)
ym = np.mean(ytr)

# Compute the differences
X1 = Xtr-xm[None,:]
y1 = ytr-ym

# Compute the variances and covariances
syy = np.mean(y1**2)
sxy = np.mean(X1*y1[:,None], axis=0)
sxx = np.mean(X1**2, axis=0)

# Compute the coefficient of determinations
Rsq = sxy**2/(1e-8+sxx)/syy

# Plot the values
plt.stem(Rsq)
```

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_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 = 100  # Number of neurons to use

# TODO
# Isel = ...
# print("The neurons with the ten highest R^2 values = ...)
I = np.argsort(Rsq)[::-1]
Isel = I[:d]

print("The neurons with the ten highest R^2 values = "+str(Isel[:10]))
```

```
The neurons with the ten highest R^2 values = [192 152  58 136 153  67  98 12
0 195 193]
```

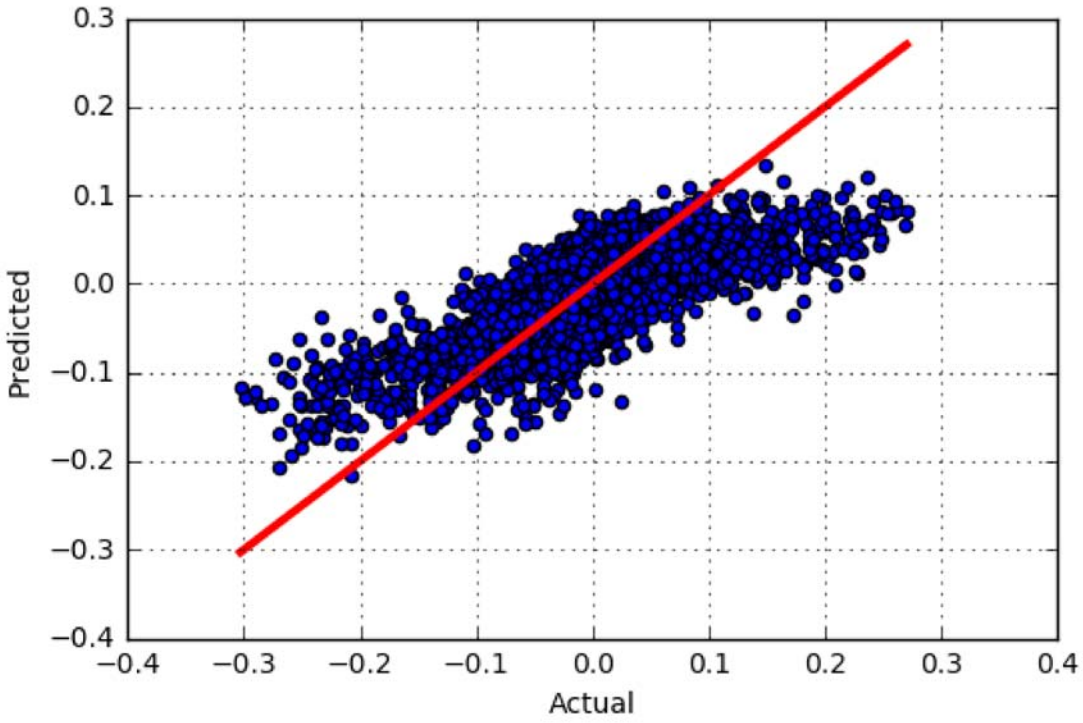
Fit a model using only the `d` neurons selected in the previous step and print both the test RSS per sample and the normalized test RSS.

In [12]:
```
regr.fit(Xtr[:,Isel], ytr)
yts_hat = regr.predict(Xts[:,Isel])

RSS_samp = np.mean((yts_hat-yts)**2)
RSS_norm = np.mean((yts_hat-yts)**2)/(np.std(yts)**2)
print("Test RSS per sample = %f" % RSS_samp)
print("Normalized test RSS = %f" % RSS_norm)
```

```
Test RSS per sample = 0.001629
Normalized test RSS = 0.514133
```

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]:
```
ymin = np.min(yts)
ymax = np.max(yts)
plt.scatter(yts, yts_hat)
plt.plot([ymin, ymax], [ymin, ymax], 'r-', linewidth=3)
plt.grid()
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.show()
```



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.

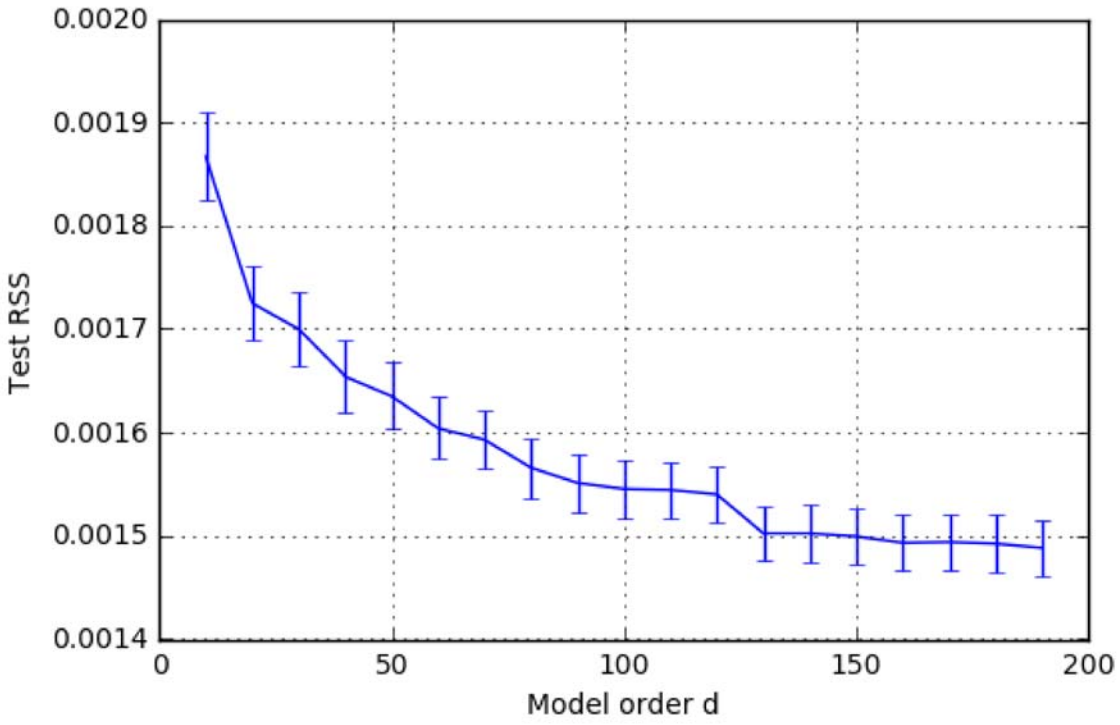
In [14]:
```
import  sklearn.model_selection

# 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 = ...

# Loop over the folds
RSSts = np.zeros((nd,nfold))
for isplit, Ind in enumerate(kf.split(X0)):

    print("fold = %d " % isplit)

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

    for it, d in enumerate(dtest):
        Isel = I[:d]
        X1 = X0[:,Isel]

        Xtr = X1[Itr,:]
        ytr = y0[Itr]
        Xts = X1[Its,:]
        yts = y0[Its]

        # Fit data on training data
        regr.fit(Xtr,ytr)

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

```
fold = 0
fold = 1
fold = 2
fold = 3
fold = 4
fold = 5
fold = 6
fold = 7
fold = 8
fold = 9
```

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)
plt.grid()
plt.xlabel('Model order d')
plt.ylabel('Test RSS')
```

Out[15]: <matplotlib.text.Text at 0x26dadbee630>

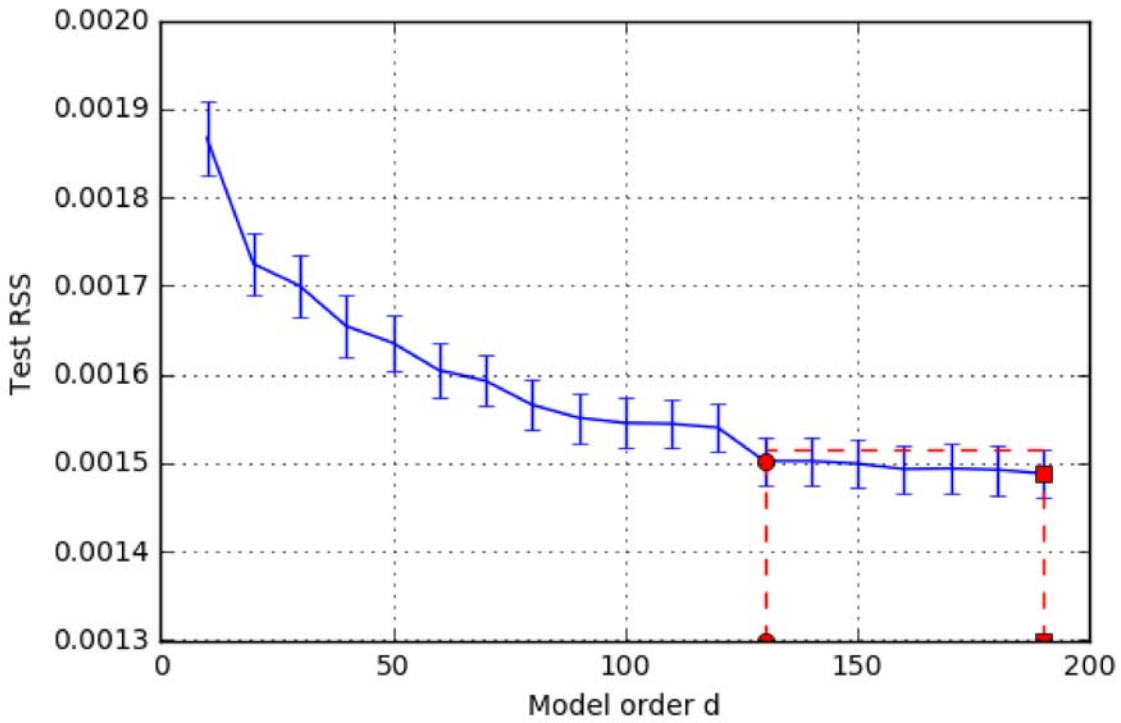


Find the optimal order using the one standard error rule. Print the optimal value of d and the mean test RSS per sample at the optimal d.

In [17]:
```
imin = np.argmin(RSS_mean)
dmin = dtest[imin]
RSS_tgt = RSS_mean[imin] + RSS_std[imin]
iopt = np.where(RSS_mean <= RSS_tgt)[0][0]
dopt = dtest[iopt]

print("The optimal d= %d" % dopt)
print("Test RSS per sample = %f" % RSS_mean[iopt])


# Plot the solution.  This is not necessary
RSS_min_plt = 0.0013
RSS_max_plt = 0.0020
plt.errorbar(dtest, RSS_mean, yerr=RSS_std)
plt.plot([dmin, dmin], [RSS_min_plt, RSS_mean[imin]], 'rs--')
plt.plot([dmin, dopt], [RSS_tgt, RSS_tgt], 'r--')
plt.plot([dopt, dopt], [RSS_min_plt, RSS_mean[iopt]], 'ro--')
plt.grid()
plt.xlabel('Model order d')
plt.ylabel('Test RSS')
plt.ylim([RSS_min_plt, RSS_max_plt])
```

```
The optimal d= 130
Test RSS per sample = 0.001502
```

Out[17]: (0.0013, 0.002)



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 [18]:
```
from sklearn import preprocessing

# TODO
Xs = preprocessing.scale(X0)
```

```
C:\Anaconda3\lib\site-packages\sklearn\utils\validation.py:429: DataConversio
nWarning: Data with input dtype uint8 was converted to float64 by the scale f
unction.
  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.

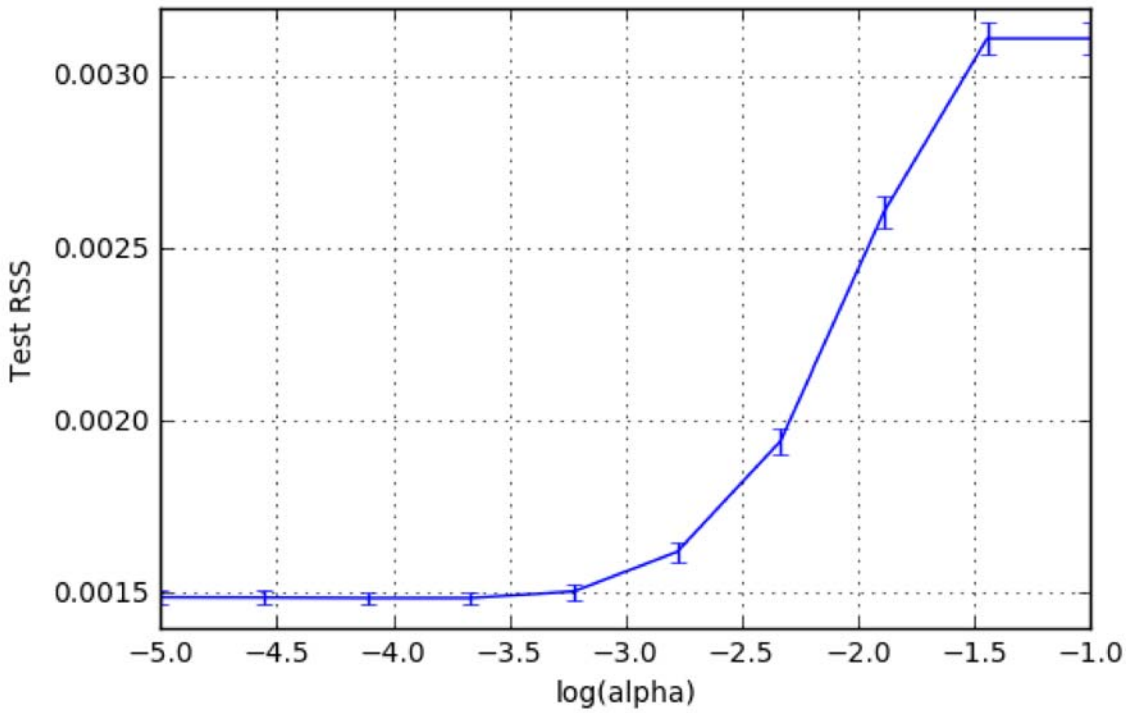
In [20]:
```
# Create a k-fold object
nfold = 10
kf = sklearn.model_selection.KFold(n_splits=nfold,shuffle=True)

# alpha values to test
nalpha = 10
alpha_test = np.logspace(-5,-1,nalpha)

# TODO

# Construct the LASSO estimator
model = sklearn.linear_model.Lasso(warm_start=True)

# Loop over the folds
RSSts = np.zeros((nalpha,nfold))
for isplit, Ind in enumerate(kf.split(X0)):

    print("fold = %d " % isplit)

    # Get the training data in the split
    Itr, Its = Ind
    Xtr = Xs[Itr,:]
    ytr = y0[Itr]
    Xts = Xs[Its,:]
    yts = y0[Its]

    for it, a in enumerate(alpha_test):

        # Fit data on training data
        model.alpha = a
        model.fit(Xtr,ytr)

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

```
fold = 0
fold = 1
fold = 2
fold = 3
fold = 4
fold = 5
fold = 6
fold = 7
fold = 8
fold = 9
```

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

In [21]:
```
RSS_mean = np.mean(RSSts, axis=1)
RSS_std = np.std(RSSts, axis=1)/np.sqrt(nfold)

plt.errorbar(np.log10(alpha_test), RSS_mean, yerr=RSS_std)
plt.grid()
plt.xlabel('log(alpha)')
plt.ylabel('Test RSS')
```

Out[21]:
```
<matplotlib.text.Text at 0x26dad000438>
```



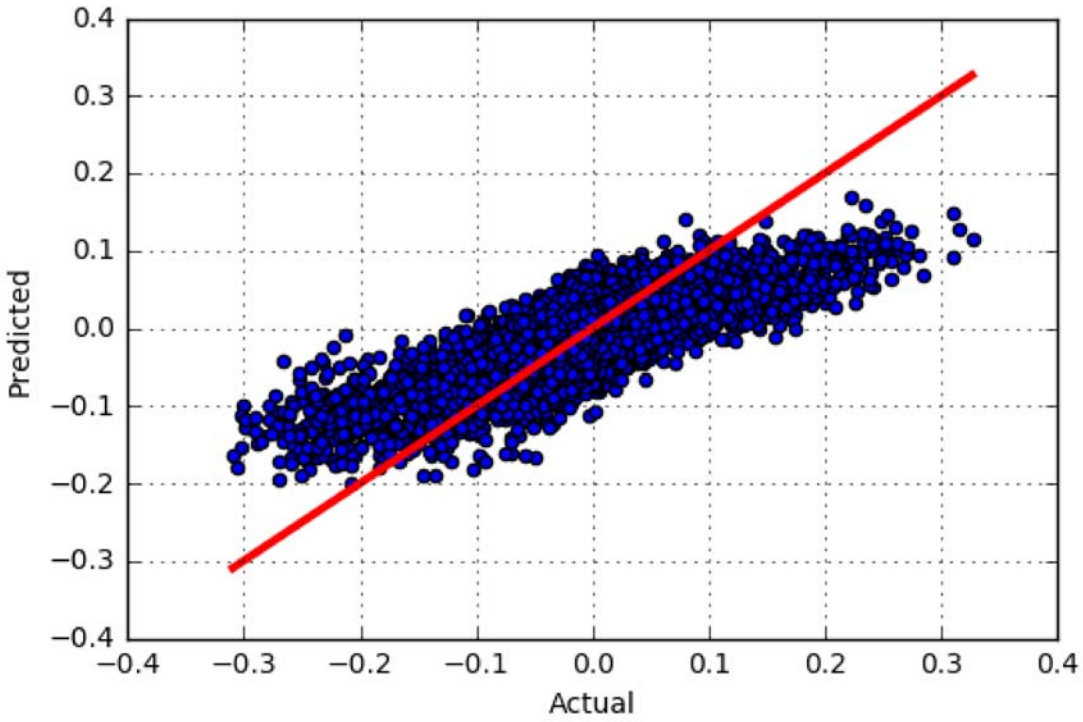
Find the optimal `alpha` and mean test RSS using the one standard error rule.

In [22]:
```
imin = np.argmin(RSS_mean)
alpha_min = alpha_test[imin]
RSS_tgt = RSS_mean[imin] + RSS_std[imin]
iopt = np.where(RSS_mean <= RSS_tgt)[0][-1]
alpha_opt = alpha_test[iopt]

print("The optimal alpha= %12.4e" % alpha_opt)
print("Mean test RSS = %f" % RSS_mean[iopt])
```

```
The optimal alpha=   2.1544e-04
Mean test RSS = 0.001485
```

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

In [23]:
```
model.alpha = alpha_opt
model.fit(X0, y0)
yhat = model.predict(X0)

ymin = np.min(y0)
ymax = np.max(y0)
plt.scatter(y0, yhat)
plt.plot([ymin, ymax], [ymin, ymax], 'r-', linewidth=3)
plt.grid()
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.show()
```



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 [ ]:
```
```