

June 2002

Barcelona Short Course

Daniel Kaplan, Macalester College

Fitting and Predicting with Locally Linear Models

Objectives

This lab will introduce some concepts from function estimation. The emphasis will be on how to construct dynamical models from data and how to use those models to generate synthesized data, make predictions, and measure the determinism of data.

There are, of course, many different types of models. This lab will focus on one simple type: the locally linear model. This is not the best type of model for all purposes, but it is generally very good for most purposes. It has the great advantage of having only one parameter that needs to be set: the number of neighbors to use in fitting the locally linear model. It is easy to find reasonable values of this parameter. Of course, you may also need to embed a time series in order to generate the model, so the embedding parameters are needed as well.

Whatever the technical architecture of a model, the abstract structure is the same. Given an input, the model produces an output that corresponds to the input. There are two basic steps:

Training In the training stage, often called *fitting*, we provide some inputs and corresponding outputs. The model “learns” from this training data how to translate from an input to an output.

Evaluation Using the trained model, we can now provide an input. The model, having already been trained, translates this input into an output.

This lab focusses on how to set up the training data, and various ways of evaluating the model.

The Training Data

We will be modeling dynamical systems. In a dynamical system, the present state determines the future state. Our data, when suitably processed and embedded, provide a measurement of the present state. We will call these measurements the “pre-image.” The system’s dynamics translate the pre-image into the future, which we’ll call the “post-image.”

In order to train the model, we need to provide the pre-images and the corresponding post-images. As a simple example, consider some data from the quadratic map:

```
>> load quad.dat
```

Plotting the first few points versus time shows the data look irregular

```
>> plot(quad(1:100))
```

but plotting $x(t+1)$ vs $x(t)$

```
>> scatterplot(quad)
```

shows how the past value, $x(t)$ determines the future value, $x(t+1)$. In this case, the state is a scalar and an appropriate state space is one dimensional.

Although the `quad` data is 1000 points long, let’s take just the first 500 points as the training data. First, we need to take the pre-image data:

```
>> pre = quad([1:500]);
```

This¹ gives the state of the system at each time 1 to 500; there is one row for each time point.

¹Of course I could have written the command as

```
>> pre = quad(1:500);
```

Writing `[1:500]` instead of `1:500` gives exactly the same result. However, later I’ll be adding 1 to each of the points in `1:500`.

Next, we need to take the post-images that correspond to each of the 500 pre-images. At each time, t , the post-image is $x(t+1)$. So, our post-images are²

```
>> post = quad(1+[1:500]);
```

But, you object. The post-images are the same data as the pre-images. This is true. But each row in the post-image is different than the corresponding row in the pre-image. It is the nature of dynamics that the post-image at time t is the pre-image at time $t+1$.

Let's do an example more generally. Suppose we take the x, y, z signals from the Lorenz system. Then the state of the system as it evolves over time can be represented by the matrix `state = [x y z]`. Each row in this matrix gives the state at a single time. Suppose we wish to take as our training data the first 999 points of this state:

```
>> pre = state([1:999],:);
```

The corresponding post-image is the same set of states, but offset by one time step³

```
>> post = state(1+[1:999], :);
```

Similarly, we might have a state reconstructed via an embedding of a measured signal. The embedding is a matrix and we construct the pre- and post-images for the training data in the same way as above.

Do keep in mind that if you have N rows in your state matrix, you will not be able to use all N of them as pre-images. The reason is that you need to have a post-image for each pre-image, and so you could use at most $N-1$ of the rows as pre-images. Things can be even more restrictive, since we will often want to reserve some of our data for evaluation of the model.

Tools

The main program is `linpredict`. This is a very general program. The program takes at least 4 arguments:

pre The training pre-images.

post The training post-images.

k The number of neighbors to use in constructing the local linear model. This is the only parameter of the model, and it is quite easy to find a reasonable value for it. Here are the general principles:

- **k** must be larger than the dimension of the input space.
- **k** must be smaller than the length of the training set.
- **k** The smaller **k** is, the more local the model and the more jagged the model can be. The larger **k** is, the smoother the model.
- **k** If **k** is similar in size to the number of training data points, the model is essentially globally linear.

newpre another set of pre-images; the ones that you want to evaluate the model at.

The first three arguments set up the training of the model. The fourth argument evaluates the trained model at the given data points. This is potentially confusing, so let me emphasize the point: `linpredict` trains and evaluates the model all in one go.

Here's an example using the quadratic data.

```
>> pre = quad([1:500]);
```

```
>> post = quad(1+[1:500]);
```

Since the state space is one dimensional — there's just one column in `pre` — we need $k \geq 2$. We'll take

²Following up on the last footnote, note that `1+[1:500]` in the command below has the same meaning as `2:501`. Writing `1+1:500` would be equivalent to `2:500` which is not what we need.

³Actually, the offset doesn't need to be 1. It could be any integer, positive or negative. We'll explore this later.

```
>> k=10;
```

The three variables `pre`, `post`, and `k` provide all the information needed to train the model. But we also need to tell `linpredict` where to evaluate the model. We'll going to specify another set of pre-images, and `linpredict` will return to us the model's corresponding post-images. Since the pre-image space is one-dimensional, we can in this case make a graph of post-image vs pre-image. For the data themselves this is

```
>> plot(pre,post,'.')
```

which sketches out the shape of the quadratic function. Let's make a similar graph for the model, by asking `linpredict` to evaluate the model at a series of points from -0.5 to 1.5 :

```
>> newpre = [(-.5):.1:1.5];
```

(N.B.: `newpre` should have one column for each dimension in the state space. Since in our example the state space has dimension 1, `newpre` has one column. `linpredict` would complain if you made `newpre` a row vector.) Now, we simultaneously train and evaluate the model

```
>> modelvals = linpredict(pre,post,k,newpre);
```

The variable `modelvals` now contains the post-images that correpond to each of the points in `newpre`. We can make a plot comparing the data with the model as follows:

```
>> plot(pre,post,'.', newpre, modelvals, 'x');
```

Note that the locally linear model follows the data pretty closely, but it also extends the model beyond the domain of the training data. This extrapolation is linear, because for the extrapolation, no matter how far out we go, the "locale" in "locally linear" is the same points at the extreme of the training data.

In general, the pre-images and post-images will not be one-dimensional, and we will not be able to make plots like the above. Instead, we will need to be able to use other techniques to judge whether the fitted function is reasonable and to use the fitted function to study the dynamics of the experimental system we are interested in.

Evaluating the Fitted Function

What are we to do with this fitted function? The function comprises our model of the dynamics of the system; although it isn't in the form of an algebraic expression, it plays the same role in the dynamics:

$$x_{t+1} = f(x_t)$$

the fitted function is the $f(\cdot)$ that we estimate from the dynamics. We can therefore use it to synthesize data, just as we could if we had an algebraic form for $f(\cdot)$.

Here's an example based on data from the Lorenz system:

```
>> [t,x,y,z] = makelorenz(100, [1 0 10], .1);
```

```
>> state = [x y z];
```

```
>> pre = state([1:500],:);
```

```
>> post = state(1+[1:500],:);
```

```
>> k = 50;
```

Now we have all the information for a model.

Let's evaluate the model at $(0, 0, 0)$; we know that the real Lorenz system has a fixed point at the origin, and so the next state should be the same, $(0, 0, 0)$.

```
>> linpredict(pre,post,k,[0 0 0]) => ans: 0.0608 0.2234 4.9179
```

So, the model isn't exactly right. But perhaps it captures the essential dynamics of Lorenz even if the details aren't exact. One way to find out is to continue the dynamics starting at the new point we just calculated:

```
>> linpredict(pre,post,k,[0.0608 0.2234 4.9179]) => ans: 0.3559 0.8088 7.4691
```

We could continue on in this way indefinitely, generating a trajectory of the model dynamics. This is, admittedly, quite tedious. To save typing, the function `freerun` does the job for us:

```
>> traj = freerun(pre,post,k,[0 0 0],1000);
```

The last argument says to run the model for 1000 steps. We can look at the trajectory in the usual way:

```
>> plot3(traj(:,1), traj(:,2), traj(:,3))
```

You might imagine using free running of the model in the following way: construct the model and free run it from the first point in the data. If the model is right, then the model trajectory should be close to the data's trajectory. However, one serious shortcoming of this approach is that it uses the same data for constructing the model and for evaluating it. This is called “in-sample” testing and tends to overestimate the quality of the model.

One way to evaluate the similarity of the model dynamics to the real dynamics is to divide your data into two sets, a training set and a test set. This is “out-of-sample” testing. Use the training set to create your training pre- and post-images. Then free run the model from the first point in your test data. Ideally, the model should follow the same trajectory as the test data. By comparing the two trajectories, you can get some idea of how good the model is.

Note that “comparing the two trajectories” and “get some idea” are rather vague. One problem for free-running is that if the real system is chaotic, even if the model is almost exactly right the chaotic sensitive dependence on initial conditions will tend to pull the model data away from the real data. For chaotic data, comparing the two trajectories means something like comparing the shapes and locations of the two attractors, and not expecting the model predictions to stay close to reality for long prediction times.

Even for chaotic data, the predictions may be good for short forecasts. This suggests another strategy, make many short forecasts from the points in your testing set. That is, make a short forecast starting from the first point in the testing set and compare it to the real value. Then do this again starting from the second point in the testing set, and then the third, and so on.

Leave-one-out cross-validation is a way to use all of our data for training, without running into the problems of in-sample testing. The idea is to make many models. For each model, we leave out one point. We use that point to test the model. As a result, we can use all of our data for training even while reserving all of the data for testing!

The function `L00prediction` runs `linpredict` in a way that implements leave-one-out (LOO) cross validation. To illustrate, let's pick up on the Lorenz example:

```
>> [preds,actual] = L00prediction(pre,post,50);
```

The returned variable `preds` contains the predictions from each of the models. The variable `actual` contains the corresponding actual value. Insofar as they are similar, the model is good.

Let's plot the first column of `preds` versus the first column of `actual`: if they are similar, the graph should lie on the diagonal line.

```
>> plot(preds(:,1), actual(:,1), '.');
```

The residuals are the difference between the predictions and the actual values

```
>> resids = preds(:,1) - actual(:,1);
```

The smaller the residuals, the better the fit of the model. (We take the first column in the above to translate back from the the embedding to a scalar signal.) A standard way to measure the quality of the model is to take the variance of the residuals: `var(resids)`. Using this, you can calculate an r^2 statistic for the model by taking

```
>> rsq = 1 - var(resids)/var(actual(:,1))
```

You can also do the standard sorts of regression residual diagnostic plots

```
>> plot(resids)
```

```
>> plot(resids, actual(:,1), 'x')
```

Any structure in these plots suggests that the model is not capturing all of the structure in the data, and that you should try a smaller `k`. (However, a smaller `k` may possibly lead to larger residuals, since fewer data points are being averaged together to produce each prediction.)

Although the measurement of the size of residuals in terms of their variance is almost universal in the

context of linear regression, we are performing a nonlinear regression and may not have a good reason to think that the residuals are normally distributed. Nonlinear models sometimes produce very good estimates for most points and terrible estimates for some points (such as the ones at the edge of the cloud of points). Some estimates of the size of the residuals that is robust to outliers are the median square value of the residual, and the mean of the logarithm of the absolute value of the residual:

```
>> median( resid.^2)

>> mean( log( abs( resid ) ) )
```

Although I have been using the word “prediction,” for purposes such as evaluating the determinism of a model it is not clear that one wants to do predictions. For example, it might be preferable, acknowledging the role of sensitive dependence on initial conditions in chaotic systems, to make “post-dictions”: predict the past. We might even prefer to do something even funnier, “dura-diction”: use the past and the future to predict the present. There is nothing mathematically odd about this.

To implement pre- or dura-diction, we can use `lagembed` to create a matrix that is the series of columns of the signal delayed by different amounts. Pull off one of these columns to be the “post-image” and use the rest of them as the pre-image (making sure to delete the post-image column).

```
>> state = lagembed(x,4,3);

>> dura = state(:,3);
>> pre = state(:,[1 2 4]);
>> [preds,actual] = L00prediction(pre,dura,50);
```

(Note that `linpredict`, and by extension `L00prediction`, can use a “post-image” that has a different number of columns than the pre-image.)

Questions

- Free-run the Lorenz data with different values of k . Do the results depend strongly on k ? When k is very large (a large fraction of the number of data points in the pre-image set), the global dynamics are approximately linear because the “locale” is practically the whole set. How does this global linearity appear in the trajectories generated by the model?
- Create a free-running model of the Baltimore measles data. Try an embedding dimension of $p = 3$, a lag of $\tau = 4$ and $k = 30$. Run the model for 100 steps starting at `[1 1 1]`. Do the results resemble the original data? (Notice the period of the model as opposed to the original data.) What happens if you run the model further forward in time?
- Use leave-one-out cross validation to assess the modelability of the Lorenz or Rossler data. See if you can use the residuals to help you in choosing an optimal k .
- Assess the modelability of one of the real-world data sets. Note that for data that are very rapidly sampled, prediction over the short term may be very easy just because the data don’t change very much over the long term. You can make long-term predictions by changing the relationship between the `pre` and `post` vectors. For example:


```
>> pre = state([1:500],:);

>> post = state(100+[1:500],:);
```

 will build a model that makes predictions 100 time steps ahead. (Such a model isn’t suitable for free running, but it may be quite suitable for assessing determinism.)
- One idea for detecting nonlinear dynamics in a model is to try modeling for various k , from large to small. If the predictions are better at small k , that is evidence for nonlinearity. If the predictions are better for k very large — similar to the total number of data points — that is evidence against nonlinearity.