A Locally Weighted Learning Tutorial using Vizier 1.0

Jeff Schneider and Andrew W. Moore

February 1, 1997

Contents

1 Introduction **3**
- 1.1 The Vizier 1.0 User Interface . . . 3
- 1.2 The data opportunity . . . 3

2 Simple Solutions **4**
- 2.1 Linear regression . . . 4
- 2.2 Nearest neighbor . . . 5

3 Memory Based Learning **6**
- 3.1 A distance metric . . . 7
- 3.2 Near neighbors . . . 7
- 3.3 Weighting function . . . 8
- 3.4 How to fit the local points . . . 11
- 3.5 Multivariate Learning . . . 14
- 3.6 Classification . . . 18

4 Using Locally Weighted Learning for Modeling **19**
- 4.1 "Hands-off" non-parametric relation finding . . . 19
- 4.2 Low dimensional supervised learning . . . 19
- 4.3 Complex function of a subset of inputs . . . 20
- 4.4 Simple function of most inputs, but complex function of a few . . . 20
- 4.5 Complex function of a few features of many inputs . . . 20
- 4.6 Pros and cons vs. neural networks . . . 21
- 4.7 When should locally weighted learning be used? . . . 23

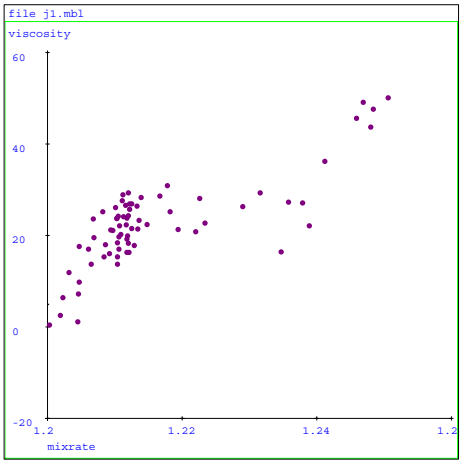
5 The Information Provided by a Learned Local Model **23**
- 5.1 Prediction distributions . . . 23
- 5.2 Noise estimate distributions . . . 24
- 5.3 Gradient estimate distributions . . . 25
- 5.4 Other things provided by local weighted models . . . 26
- 5.5 Why do we want all these estimates? . . . 27

6 Bayesian Locally Weighted Regression **27**

7 Efficient Data Storage and Retrieval **30**

8 Autonomous Modeling **31**
- 8.1 Judging Model Quality by Residuals . . . 31
- 8.2 Cross Validation . . . 32
- 8.3 Blackbox Model Selection . . . 33

9 Decisions with Locally Weighted Models **37**
- 9.1 Choosing Parameters to Achieve a Target . . . 37
- 9.2 Maximizing or Minimizing a Learned Model . . . 40
- 9.3 Using Locally Weighted Learning to Design Experiments . . . 42
- 9.4 Programming with the Vizier library . . . 43



Steel Temp	Line Speed	Slab Width	Temp Stage2	Cool SetPt	Cool Gain
-0.478	0.384	0.035	0.838	0.662	-0.951
0.347	-0.648	0.662	-0.508	0.684	-0.064
0.862	-0.536	-0.245	-0.143	-0.726	-0.611
-0.209	-0.597	0.061	-0.764	0.022	-0.762
-0.462	0.621	0.925	0.608	-0.873	-0.643
0.734	0.074	-0.265	-0.451	-0.454	0.575
0.423	-0.514	-0.073	0.038	-0.806	0.309
-0.126	0.810	-0.661	-0.076	-0.873	-0.336
0.033	0.775	0.771	-0.544	0.623	0.511
0.266	0.877	-0.569	-0.515	0.649	0.770
-0.568	0.845	0.566	-0.735	-0.339	0.588

Figure 1: A univariate (left) and a multi-variate (right) data set

1 Introduction

This tutorial is designed with two purposes. First, it teaches about Locally Weighted Learning (LWL) and what it can be used for. Second, it demonstrates the use of Vizier 1.0, a software package for doing Locally Weighted Learning. The reader may take at least two correspondingly different approaches to using this tutorial. The Vizier user may spend additional time trying out the various software options available each time interaction with the software is demonstrated, while glossing over the algorithmic details underlying the software. Alternatively, the Locally Weighted Learning student may choose to read this without using the software at all (although we believe that using Vizier is an excellent way to learn about LWL). Additional information and examples of LWL can be found in [1].

1.1 The Vizier 1.0 User Interface

Throughout the tutorial you will be requested to perform some operations using Vizier . These will be typeset in a different font as follows:

```
File -> Open -> j1.mbl
```

This is shorthand notation for: **Go to the File menu. Select the Open option. Choose the file** *j1.mbl*.

By the end of this tutorial you will have explored much of the Vizier user interface. There are several things that will not be covered explicitly, though. One of them is the shortcuts available for many of the operations. These come in two forms. First, many of the operations have their own distinct buttons on the toolbar. For example, a shortcut to the file opening operation above is to click on the button with a picture of an open folder on it. There are also keystroke shortcuts. For example, the file open operation can be invoked by typing: Control-o.

This tutorial is written with the Windows user interface in mind. There is also a command line interface which can be run from PC DOS windows or on Unix systems. If you run the command line interface and are in a PC Windows, or an X windows environment, the graphs will come up in separate windows. If not, you will not see the graphs.

1.2 The data opportunity

As computers and their networks become more prevalent throughout industry, more and more data is becoming available about all aspects of business. Examples of this data include process data from manufacturing or other control systems, inventory data, customer records, marketing and finance data, and product development test data. Locally Weighted Learning and Vizier are all about using this data to make better decisions.

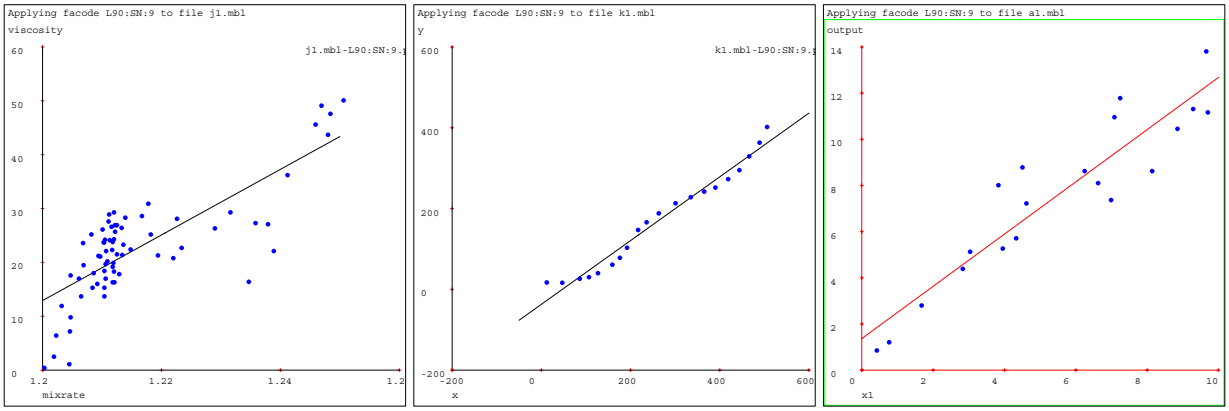


Figure 2: Global linear regression on some one dimensional data sets

Fig. 1 shows some examples of data. Much of this tutorial will focus on 1 dimensional data because it is easiest to visualize, but all of the software is designed to operate on, and is most useful on, multi-dimensional data. We assume that the relationships between the various attributes may be non-linear and that the data may have significant noise in it. There are many questions we might like to answer with the help of the data and LWL:

- What is the predicted result of choosing a particular control setting?
- How much noise is there in the data?
- What control setting will produce the best possible result?
- Which attributes are relevant to determining the results I'm interested in?
- If I want to acquire more data about my system, what experiment should I run next?

In this tutorial we show how to answer these questions and many others using LWL and some data. The tutorial covers three main topics. Sections 1 through 7 describe what a locally weighted model is and the tradeoffs involved in choosing various kinds of locally weighted models. Section 8 discusses methods of automatically finding a good locally weighted model. Section 9 gives examples of how a locally weighted model can be used to make better decisions and answer the questions listed above.

In this tutorial, we refer to a single data point, and a data set, as containing *input attributes* and *output attributes*. We are usually given a set of values for the input attributes and asked to estimate things about the corresponding output attributes, or given a desired property in the output attributes and asked to find the input attributes which are estimated to produce the desired output properties. We will assume that the attributes are already labeled as *input* or *output*, but there is no problem with changing the labels of attributes in order to investigate various possible relationships. In equations, the input attributes will often be referred to as x, and the output attributes as y.

2 Simple Solutions

Before describing Locally Weighted Learning, we'll look at some simple and common methods of building a model from data and see what kind of problems they have.

2.1 Linear regression

Linear regression is an old statistical method of determining relationships between variables. It finds the linear function (in the 1-d case, a straight line) which minimizes the sum of squared error between the function and all the data points.

We can use Vizier to see what happens when linear regression is applied to some sample data sets. All of the data sets in this tutorial can be found in the data subdirectory of the Vizier installation.

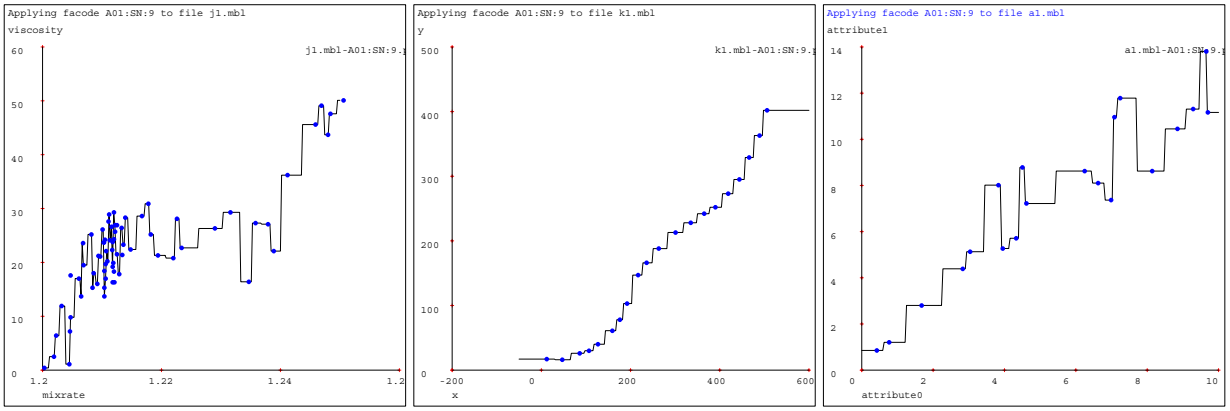


Figure 3: Nearest neighbor on some one dimensional data sets

```
File -> Open -> j1.mbl
Edit -> Metacode -> Regression   L: Linear
                    Localness    9: Global
                    Neighbors    0: No Nearest Neighbors
Model -> Graph -> Graph
```

In the previous operations, you did 3 separate things. First you loaded the data file named *j1.mbl*. Second, you specified that you wanted to do linear regression. Don't worry about the meaning of the various fields in the Metacode editor yet. They'll be described in more detail later. Finally, you drew a graph showing the data in the file and a fitted line from the linear regression. Again, don't worry about all the options available for graphing. They'll be described in more detail later. If you are not using Vizier to draw the graph, you can see what it looks like in fig. 2a. From the graph, it is evident that linear regression has captured a significant trend in the data, but has not accurately modeled the relationship.

Next, we look at another data set and linear regression applied to it.

```
File -> Open -> k1.mbl
Model -> Graph -> Graph
```

The resulting graph is shown in fig. 2b. Here there is no noise in the data and linear regression has a better fit. Unfortunately, it is also obvious that some of the relationship has been glossed over.

We'll look at one more data set and linear regression applied to it.

```
File -> Open -> a1.mbl
Model -> Graph -> Graph
```

The resulting graph is shown in fig. 2c. In this case linear regression appears to be a reasonable choice. There is significant noise in the data, but the underlying relationship seems mostly linear.

The models for the first two graphs suffer from undesirable *bias*. Bias refers to the underlying assumption made about the form of the relationship made by a particular function approximator. In these examples, the assumption is that the relationship is a straight line and any data not matching that assumption is poorly represented.

2.2 Nearest neighbor

Another obvious choice for a model is to look for the nearest point in the training data and predict whatever the output was for that point. This method is called nearest neighbor. We can see what it looks like using Vizier . We'll start with the *a1.mbl* file we've already loaded.

```
Edit -> Metacode -> Regression A: Averaging
                    Localness  0: No Local Weighting
```

```
                    Neighbors  1: T+1 Nearest
Model -> Graph -> Graph

File -> Open -> j1.mbl
Model -> Graph -> Graph

File -> Open -> k1.mbl
Model -> Graph -> Graph
```

The resulting graphs can be seen in fig. 3. The fit to *k1.mbl* is not bad, but the others are clearly fitting the noise. In some ways, nearest neighbor is at the opposite end of the spectrum from linear regression. It has little bias (assumptions about what the true function is), which allows it to fit non-linear functions without difficulty. Unfortunately, it suffers from high variance, which means noisy data causes it to make erratic predictions, as can be seen from the graphs. There are several drawbacks to the nearest neighbor method:

- Fitting the noise causes larger prediction errors than if the noise has been filtered out well.
- You can not make estimates of the gradients or the amount of noise since the approximated function is a step function passing through all the data.
- It does not interpolate smoothly between data points.

Generally, the way to overcome the problems listed above is to introduce a model with additional bias (such as assuming the function is relatively smooth). Adding bias can reduce variance which will make the model more robust to noisy data. However, as we saw with linear regression, bias which is based on incorrect assumptions can cause errors as well. The trick is to find a model with bias that will overcome noise and problems from little data (we will discuss this more later), while not causing problems from inability to properly fit the function. Unfortunately, there is no single model which performs well across all data sets. We will see later how Vizier can automatically find a good model for a given data set.

A note on the graphing options

The function approximated by nearest neighbor should pass through all the data points except cases where two data points have exactly the same input, but different outputs. If you look carefully at Vizier's nearest neighbor plot of *j1.mbl*, you'll notice that it doesn't really pass through all the points in the data set. The line drawn on the graph was generated by computing the value of the approximated function at a discrete number of points and drawing a line through those points. The default number of points is 60, which you can see in the graph dialogue box with the label "resolution". To see a more accurate graph of the function, increase this number to 500 and graph it again.

```
File -> Open -> j1.mbl
Model -> Graph -> Resolution  500
               -> Graph
```

You'll note that the new graph is more accurate, but takes significantly longer to compute. The default values are reasonable for many graphs, but you can always adjust them to trade off accuracy and compute time.

3 Memory Based Learning

Memory Based Learning (MBL) is a simple function approximation method whose roots go back at least to 1910. Training a memory based learner is an almost trivial operation: just store each data point in memory (or a database). Making a prediction about the output that will result from some input attributes based on the data is done by looking for similar points in memory, fitting a local model to those points, and then making a prediction based on the model. There are four components that define a memory based learner:

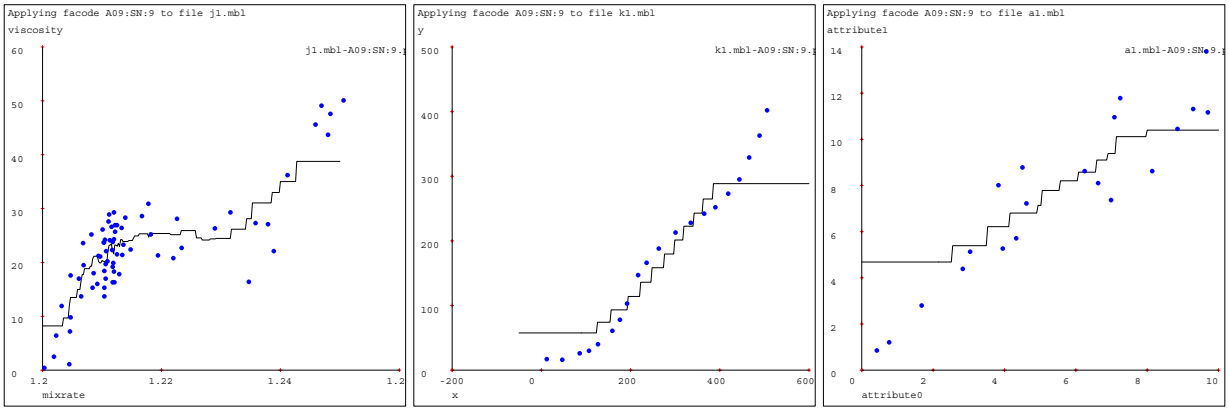


Figure 4: 9 nearest neighbor on some one dimensional data sets

a distance metric, the number of nearest neighbors, a weighting function, and a local model. Each will be described in turn in the next four subsections followed by a discussion of multivariate considerations and classification problems.

3.1 A distance metric

The set of input attributes, for which we want to make a prediction about the resulting output attributes, is called the *query*, or *query point*. The first step in making a prediction with MBL is to look through the database to find all the data points whose input attributes are similar to the query point. In order to do that, we have to define what is meant by *similar*. We need to define a distance metric that tells how close two points are.

Vizier uses a scaled Euclidean distance metric (L_2 norm). The distance between two points (between their input attributes) is defined by:

$$D(\mathbf{x}, \mathbf{x}') = \sqrt{(\mathbf{x} - \mathbf{x}')^T \Sigma (\mathbf{x} - \mathbf{x}')} \tag{1}$$

where Σ is a diagonal matrix and $\mathbf{x}$ refers to a vector of input attributes. Other distance metrics include L_1 norm (sometimes called Manhattan distance), L_∞ norm, and Mahalanobis distance (same as scaled Euclidean except that Σ is required to be symmetric, but not necessarily diagonal). Scaled Euclidean distance works well for most cases and we will not discuss the other metrics any further in this tutorial.

3.2 Near neighbors

By now, you may have realized that the nearest neighbor method described earlier is a kind of memory based learning. Look back at the graphs produced with the nearest neighbor model (fig. 3). This is also an opportunity to demonstrate another feature of Vizier . You can see the graphs from past pages of Vizier output using the following commands:

```
View -> Previous Page
     -> Next Page
```

The most distressing aspect of these graphs is that they fit noise, which yields less accurate predictions. One way to counteract this problem is to consider more than one nearest neighbor, which is called k-nearest neighbors. The prediction is the average output of the k points nearest to the query. We can use Vizier to see what happens when we try $k = 9$ on our three one dimensional data sets.

```
Edit -> Metacode -> Regression A: Averaging
                    Localness  0: No Local Weighting
                    Neighbors  1: T+9 Nearest
```

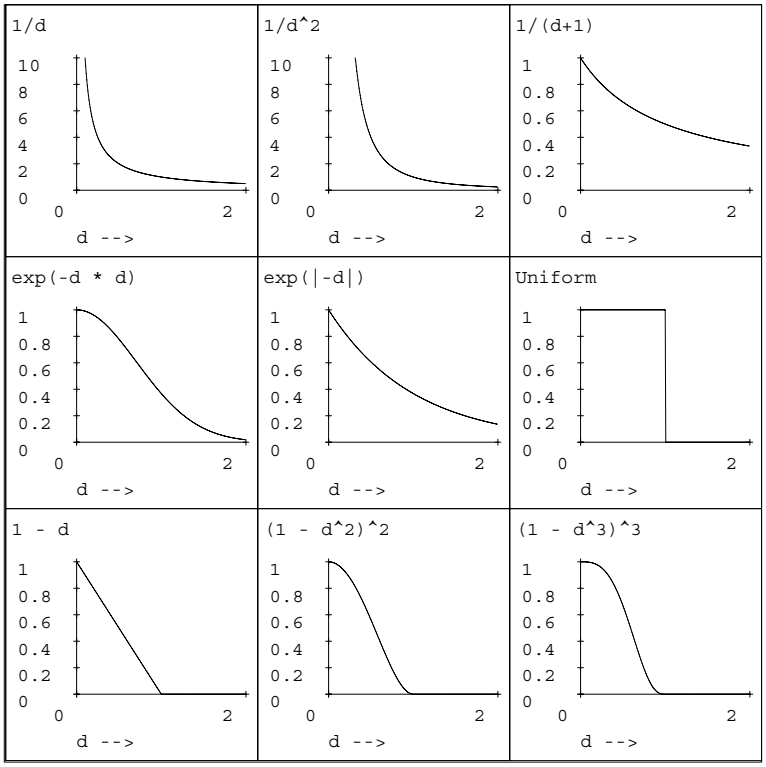


Figure 5: Nine different weighting functions. The Gaussian function used by Vizier is the leftmost function in the middle row.

```
File -> Open -> j1.mbl
Model -> Graph -> Graph

File -> Open -> k1.mbl
Model -> Graph -> Graph

File -> Open -> a1.mbl
Model -> Graph -> Graph
```

The resulting graphs can be seen in fig. 4. The graph of *j1.mbl* shows that 9 nearest neighbor has managed to average out the noise in the function and produce a better fit than 1 nearest neighbor. However, 9 nearest neighbor made the fit to *k1.mbl* worse. There was no noise in that data to begin with, and the averaging ended up smearing out some of the structure in the data. It is obvious that no single value of k works best for all data sets. On *a1.mbl* the noise is taken care of, but the other inadequacies of k-nearest neighbor are apparent. There is still no useful gradient information in the function, and the fit near the boundaries of the data at the left and right side of the graph is poor. In the next two sections we will see what can be done to remedy those problems, but even when using those remedies we may still want to choose some non-zero number of nearest neighbors to get the best fit to data.

3.3 Weighting function

With nearest neighbor, a prediction at any point is made from a simple average of a small subset of nearby points. All the other points in the data set are completely ignored. The result is a discontinuous step function. All queries made in a region where the same subset of points is the nearest neighbor get exactly the same prediction, and there are discontinuities at the boundaries of these regions. An alternative which

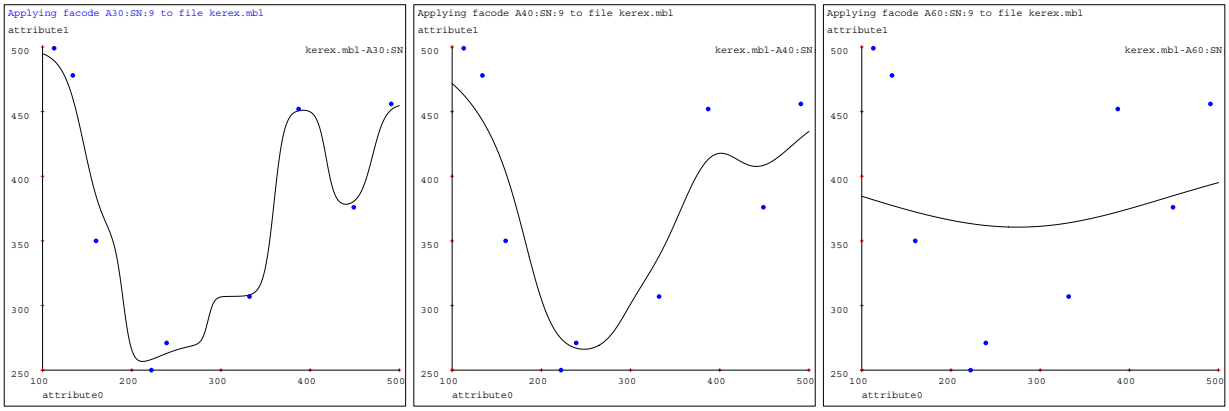


Figure 6: Kernel regression with different kernel widths. The graphs are of localness = 3, 4, and 6, respectively.

will smooth out the function is to use a *weighted* average instead. This is called *kernel regression*. Every point in the data set will receive a weight between 0.0 and 1.0 based on how close it is to the query. There are numerous different weighting functions that can be used. Fig. 5 shows nine common types. The first two at the top are undesirable because the weight goes to infinity as the distance goes to zero. This would defeat our attempts to average out noise in the data. The "Uniform" weighting is undesirable because it behaves much like k-nearest neighbor. Some subset of nearby points (all those within a fixed distance) get a weight of 1.0 while all the others get 0.0. This would cause the same discontinuities we saw with k-nearest neighbor. The functions on the bottom row are reasonable except that the weight drops completely to 0.0 after a certain distance. That's undesirable when a query is made far from the data. We would still like to give some weight to the nearest data points to give us some hint about what to predict. The functions in the center and the upper right have exactly the opposite problem. Their weight does not drop off fast enough and the result is that the numerous points far from the query collectively outweigh the points near the query and produce poor predictions. This leaves only the Gaussian function at the left of the middle row. Its tail drops off very quickly, but never becomes 0.0. We will focus only on the gaussian weighting function, which is what Vizier uses. A weight for each point is computed as follows:

$$w_i = \exp(\frac{-D(\mathbf{x_i}, query)^2}{K_w^2}) \tag{2}$$

Then a prediction is made with the weighted average:

$$\hat{y} = \frac{\sum w_i y_i}{\sum w_i} \tag{3}$$

Just as the choice of k in k-nearest neighbor is important for good predictions, the choice of K_w (known as the kernel width) is important for good predictions with kernel regression. As K_w gets large, the weight of all points approaches 1.0, and predictions become the global average of all the points. As K_w gets small, all but the very nearest points get a weight of 0.0, and the fitted function starts to look like nearest neighbor.

We can observe this behavior with Vizier . For this example we will use a data file with only 9 data points so it will be more obvious what is happening. These plots are all shown in fig. 6

```
File -> Open -> kerex.mbl
Edit -> Metacode -> Localness 3: Very very local
        Neighbors -> Neighbors 0: No nearest neighbors
Model -> Graph -> Graph
```

The kernel width in Vizier is set by the "localness" parameter of the metacode editor. A localness value of 3 means the kernel width is 1/16 of the range of the x axis. Especially near the bottom of the graph, the function flattens out at the values of the actual data points rather than interpolating smoothly between

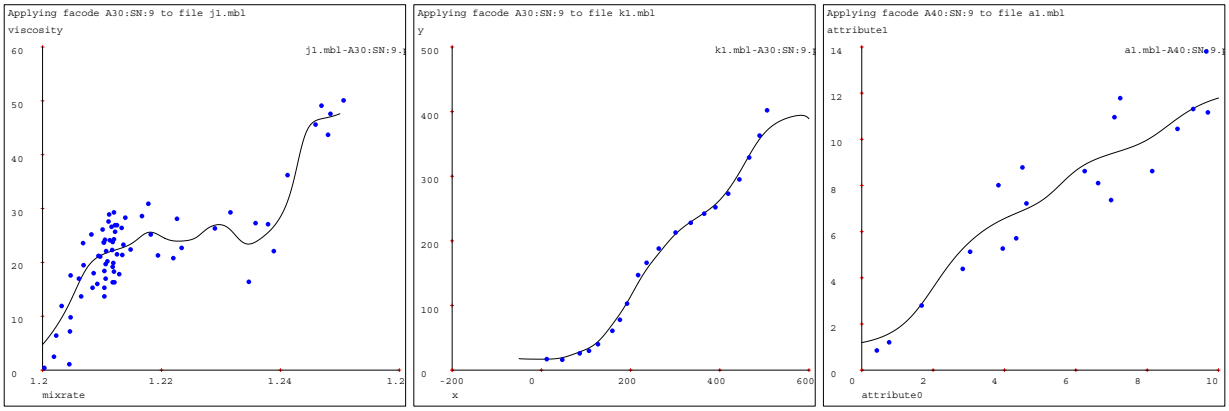


Figure 7: Kernel regression on three different data sets, with localness values of 3, 3, and 4, respectively.

them. This is reminiscent of the nearest neighbor plots. We can improve the model by increasing the kernel width.

```
Edit -> Metacode -> Localness 4: Very local
Model -> Graph -> Graph
```

This kernel width is 1/8 of the range of the x axis. The general relationship between the kernel width and the localness number is:

$$K_w = 2^{localness-7} * input\_range \tag{4}$$

The new graph is much better. It has smoothed out the noise and we might even trust its estimate of the function's gradient. Increasing the kernel width further will cause it to over-smooth the data.

```
Edit -> Metacode -> Localness 6: Somewhat local
Model -> Graph -> Graph
```

This kernel width is half the range of the x axis and we can see that it has almost smeared all the data together into one flat line which is a poor fit to the data. This is a good time to try the other localness values to see how they effect the fitted function. If the localness is set lower than 3, you will notice the function going to a value of 375 when there are no nearby data points even though the nearest data points do not have values around 375. This is an artifact of the Bayesian priors which are described later in this tutorial. You may also notice that the label written next to localness 9 is "global." 9 is a special case that does not follow eq. 4, but instead gives all the data a weight of 1.0.

Now we return to the three data sets we used earlier and see how kernel regression does with them. They are shown in fig. 7.

```
File -> Open -> j1.mbl
Edit -> Metacode -> Localness 3: Very very local
Model -> Graph -> Graph

File -> Open -> k1.mbl
Model -> Graph -> Graph

File -> Open -> a1.mbl
Edit -> Metacode -> Localness 4: Very local
Model -> Graph -> Graph
```

These three graphs look much nicer than the ones we got with linear regression or with k-nearest neighbor. They have smooth functions and do a decent job of fitting the data. The bumps in the graphs for *j1.mbl* and

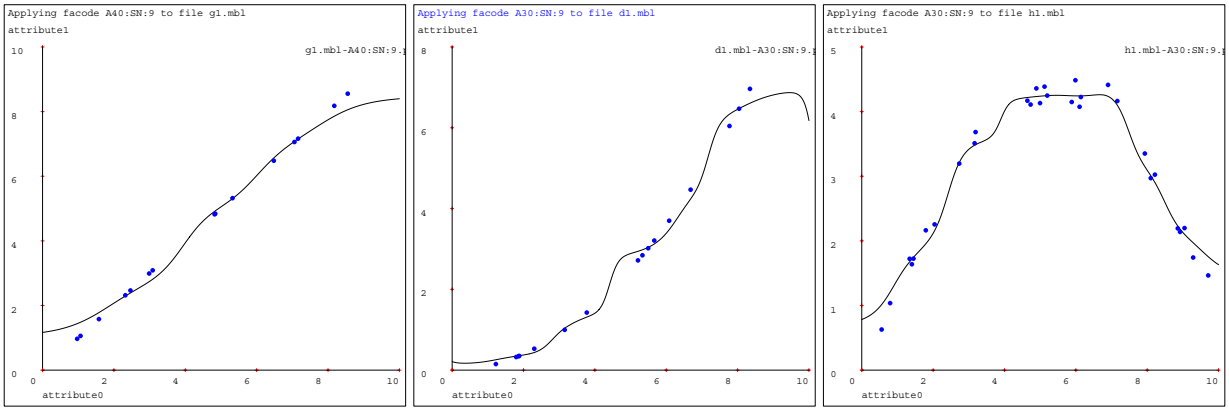


Figure 8: Kernel regression on three different data sets, with localness values of 4, 3, and 3, respectively.

a1.mbl are a little worrisome. In general, the way to smooth out bumps is by increasing the kernel width. Unfortunately, doing that in these cases will cause over-smoothing of the data and a poorer fit. The kernel widths for these data sets were hand chosen and they demonstrate how important it is to choose the right kernel width for each data set. Finally, it should be noted that k-nearest neighbor can be combined with kernel regression. In that case, the k nearest data points automatically receive a weight of 1.0, while all the others are weighted by the weighting function.

3.4 How to fit the local points

The fourth and last thing we need to define for a memory based learner is the function it will use to fit the local data. Kernel regression was an improvement over k-nearest neighbor on the data sets we've seen so far. However, it can still look fairly bad. The three data sets in fig. 8 show this.

```
File -> Open -> g1.mbl
Edit -> Metacode -> Localness 4: Very local
Model -> Graph -> Graph

File -> Open -> d1.mbl
Edit -> Metacode -> Localness 3: Very very local
Model -> Graph -> Graph

File -> Open -> h1.mbl
Model -> Graph -> Graph
```

Again, the kernel widths were chosen manually. Notice how the first two do an extremely poor job of extrapolating away from the data. The middle graph also has a strange bump where the gap in the data is. The data in *h1.mbl* was generated with a three segment piecewise linear function plus noise. Unfortunately, the gradient between the first two segments is not very trustworthy.

This problem can be addressed by changing the local model being fit. K-nearest neighbors and kernel regression use a constant, or averaging, local model. Sometimes, the fitted function can be improved by using a more sophisticated local model. Next, we consider a linear local model.

At the beginning of this tutorial we saw how global linear regression looks on some sample data sets. Now, we'll briefly summarize the math behind linear regression. It is fine to skim this section if looking at equations is not your thing. Assume that all of the data was generated from the following relationship:

$$y_k = \beta x_k + \varepsilon_k \tag{5}$$

where the k subscript is the data point number, x_k is the kth input, y_k is the kth output, β is the coefficient for the input, and ε is drawn from a Gaussian distribution with mean zero and standard deviation σ. y_k may

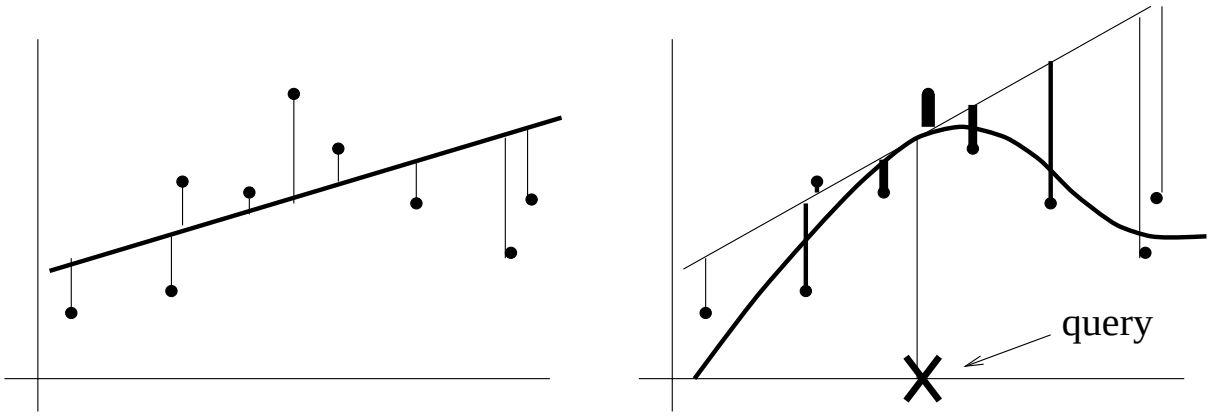


Figure 9: Comparison between global and local linear regression. a) Global linear regression minimizes the sum of squared distances between the data and the line. The result is a line fit to the data. b) Local linear regression fits a line at the query point. It minimizes the sum of weighted squared distances between the data and the line. The line widths reflect the relative strengths of the data points in influencing the fit at the indicated query. The final result is a nonlinear curve fit to the data.

be a vector of outputs, and x_k may be a vector of inputs (which would make ε_k a vector and β a matrix), but we leave them scalar here to simplify. The problem is to find the value $\hat{\beta}$ which makes the given data set most likely. The β which satisfies this condition turns out to be:

$$\hat{\beta} = {}^{\beta\, argmin^N} \sum_{k=1} (y_k - \beta x_k)^2 \tag{6}$$

In order to find the value for β which minimizes the right hand side of eq. 6, take the derivative of the right hand side with respect to β and set it to zero. Doing so gives the following:

$$\hat{\beta} = (\sum_k x_k^2)^{-1} \sum_k x_k y_k \tag{7}$$

Usually, it is preferable to write this equation in its matrix form:

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} (\mathbf{X}^T \mathbf{Y}) \tag{8}$$

where $\mathbf{X}$ is a matrix with one column for each input attribute and one row for each data point, $\mathbf{Y}$ is a matrix with one column for each output attribute and one row for each data point, and $\hat{\beta}$ is a matrix with the number of rows equal to the number of outputs and the number of columns equal to the number of inputs.

Eq. 5 assumes that the regression line passes through the origin. In order to remove that constraint, it is necessary to add an extra "input attribute" whose value is always 1 for every data point. Therefore, if a data set contains n input attributes, the $\mathbf{X}$ matrix has $n + 1$ columns.

Fig. 2 shows global linear regression on several data sets. We can improve on the fit to these by fitting the linear model locally, just as we computed weighted averages earlier. A weight for each data point is computed just as before. Rather than computing a weighted average of the outputs as done in kernel regression, the weight is used on the inputs and outputs of the the data point. In matrix form, the weights for the data points are composed into a diagonal matrix, $\mathbf{W}$, where the value of the ith diagonal element is the weight on the ith point. The regression equation becomes:

$$\hat{\beta} = ((\mathbf{WX})^T (\mathbf{WX}))^{-1} ((\mathbf{WX})^T (\mathbf{WY})) \tag{9}$$

Fig. 9 show a graphical comparison between the way global linear regression works and the way local linear regression works.

Fig. 2 shows what global linear regression does on some sample data files. Vizier can show us what locally weighted linear regression does on these same files.

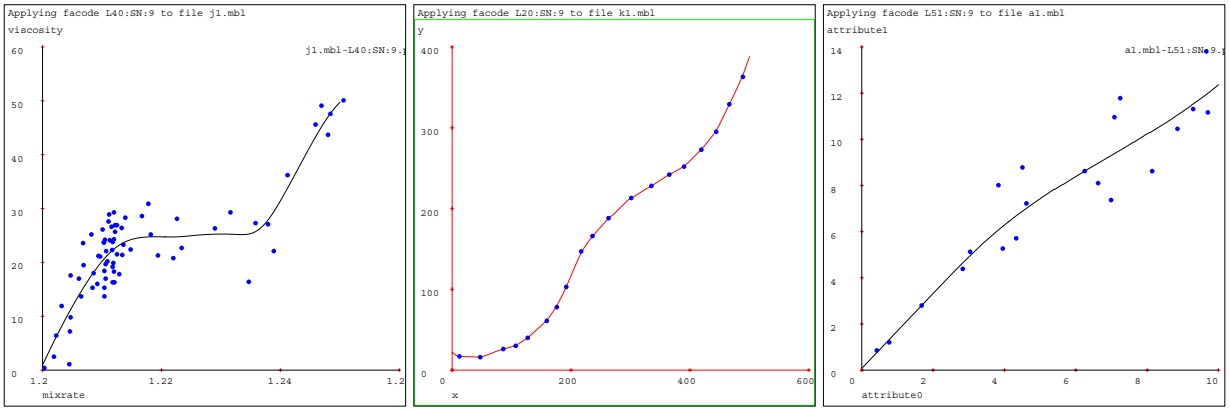


Figure 10: Local linear regression on three different data sets, with localness values of 4, 2, and 5, respectively.

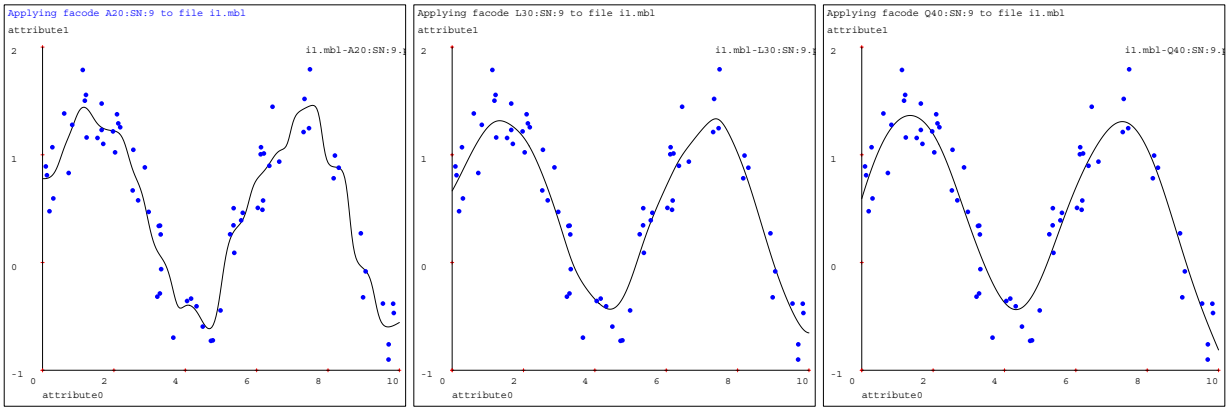


Figure 11: Kernel, linear, and quadratic regression on one data set. The best localness values are determined manually to be 2, 3, and 4, respectively.

```
File -> Open -> j1.mbl
Edit -> Metacode -> Localness   4: Very local
                    Regression  L: Linear
Model -> Graph -> Graph

File -> Open -> k1.mbl
Edit -> Metacode -> Localness   2: Ultra local
Model -> Graph -> Graph

File -> Open -> a1.mbl
Edit -> Metacode -> Localness   5: Fairly local
Model -> Graph -> Graph
```

These graphs are shown in fig. 10. The fits appear better than the global linear regression graphs and the local averaging in fig. 7. Linear regression is also preferable to averaging because it can be used to make estimates of gradients (there will be more about gradients later).

After observing that fitting a local linear model can be better than a local averaging model, we might ask whether there are other useful models. An obvious candidate is local quadratic regression. This is done by adding the quadratic terms to the $\mathbf{X}$ matrix. For example, the full $\mathbf{X}$ matrix for a two input quadratic regression would include separate columns for the terms: $1, x_1, x_2, x_1^2, x_1 x_2, x_2^2$. This increases the dimension of the resulting $\hat{\beta}$ matrix accordingly, but the equation for computing it is the same. In order to see the effects of quadratic regression, we can compare it against the average and linear models.

```
File -> Open -> i1.mbl
Edit -> Metacode -> Localness   2: Ultra local
                    Regression  A: Average
Model -> Graph -> Graph

Edit -> Metacode -> Localness   3: Very very local
                    Regression  L: Linear
Model -> Graph -> Graph

Edit -> Metacode -> Localness   4: Very local
                    Regression  Q: Quadratic
Model -> Graph -> Graph
```

Fig. 11 shows these graphs. The graphs get successively smoother as the regression goes from average to linear to quadratic. Also note that the best localness value increases. One way to get a smoother graph while using averaging is to increase its localness value, but this has the negative side effect of adding bias and making the fit worse (you can test this yourself with Vizier). One of the advantages of higher order polynomial regressions is that they allow you to use a wider smoothing kernel without causing bias problems. A wider kernel is advantageous because it makes use of more data in its fit. While linear regression has the advantage of giving gradient estimates, quadratic regression has the additional advantage of giving estimates of the local Hessian, which can be useful when doing optimization. The drawback of quadratic regression is that it has more parameters which means it requires more data to fit them well and it requires more computation to do the fit.

Vizier has two other models which are part way between linear and quadratic. The "Ellipses" model has all the pure quadratic terms (the x_i^2), but none of the cross terms ($x_i x_j$ for $i \neq j$). The "Circles" model is a linear model plus a single extra term for the sum of all the pure quadratic terms ($\sum_{i=1}^{D} x_i^2$). In some cases, one of these two models is a good compromise between linear and quadratic. It doesn't make sense to use them on the one dimensional data sets we've been using so far because one dimensional inputs have only one pure quadratic term and no cross terms.

At this point we have covered the four things that describe a locally weighted learning: the distance metric, the number of nearest neighbors, the weighting function, and the local model. In Vizier the choice of these four things is specified by the metacode. Using different metacode settings, you can create a variety of different locally weighted learners.

3.5 Multivariate Learning

All of the examples so far have been on one dimensional data sets, but all of the methods presented so far generalize easily to multivariate inputs and outputs. We can see the fits of two dimensional data with contour plots. *a2.mbl* is a data file containing 100 data points with two input dimensions and one output dimension. The data was generated with the following equation: $y = \sin(x_1) + \sin(x_2) + noise$

```
File -> Open -> a2.mbl
Edit -> Metacode -> Localness   4: Very local
                    Regression  Q: Quadratic
Model -> Graph -> Dimensions  2
                  Graph
```

The graph is shown in fig. 12. The contours show the approximated function and have the expected shape of smooth peaks mixed between smooth valleys. The data points are also shown. It is no longer possible, however, to evaluate how well the function is approximated visually. The distribution of the data points in input space can be seen, but the height, or output value, of each data point is not represented in the plot.

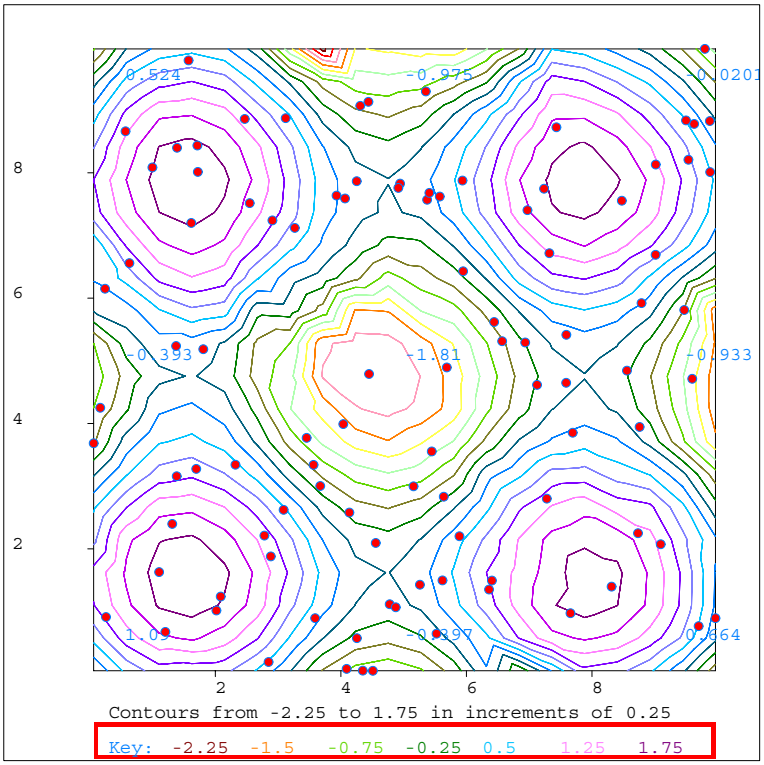


Figure 12: A two dimensional contour plot of $\sin(x_1) + \sin(x_2)$ with a quadratic local model

A note on graphing and the query point

Even though the data is multi-dimensional, it is often desirable to see one dimensional graphs or contour plots. Doing this causes several side effects that we will discuss now. First make a 1-d plot of the *a2.mbl* data.

```
Model -> Graph -> Dimensions  1
                  Graph
```

This graph plots the first input attribute against the output attribute while ignoring the second input attribute. Thus, it is now possible to see the output values of the data points. At first glance the fit to the data seems incorrect, but that is not the case. All of the data is presented in the graph regardless of what the value of its second input attribute is. The curve, however, shows the approximated function with the second input attribute held constant at its value in the query point. At the bottom of the output, the query is shown to be the point [5, 5]. If we back up and look at the page with the contour plot, we see that when the second attribute is held at a value of 5, the approximated function goes through the middle of the valley at the center and between the peaks at each side. This explains the 1-d graph with the curve near the bottom of the data. It is a graph of how the approximated function varies with the first input attribute while the second is held fixed. If we change the value of the second input attribute to 8 and repeat the graph, the curve will move near the top of the data.

```
Edit -> Query -> ``x2'' 8
Model -> Graph -> Graph
```

It is also possible to see how the function looks when the first input attribute is held fixed and the second is varied.

```
Model -> Graph -> x attribute -> ``x2''
                  Graph
```

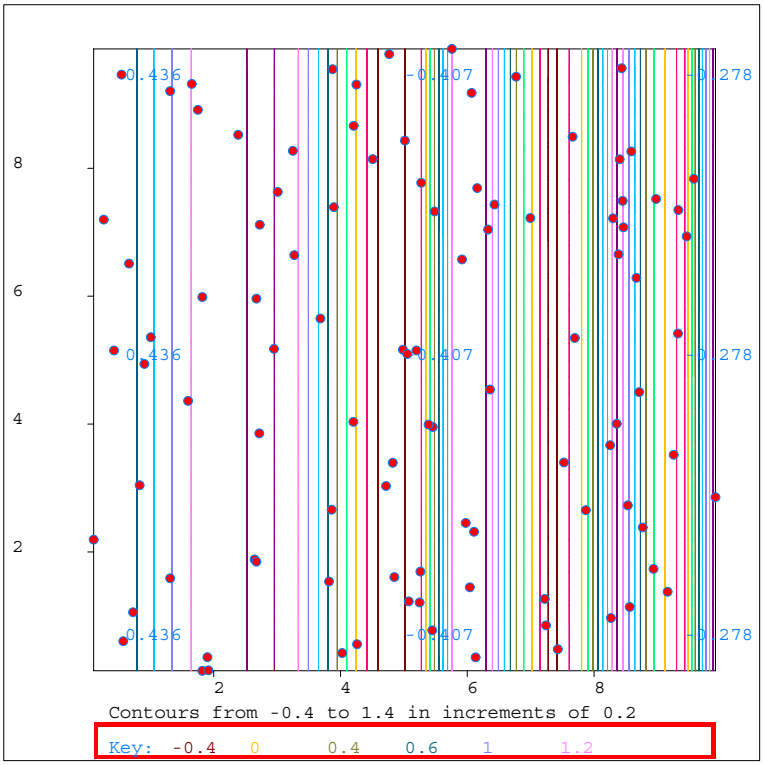


Figure 13: A two dimensional contour plot of $\sin(x_1^2)$ with an averaging local model

This graph looks very similar to the original graph using the "x1" input attribute. That is because the data was generated from a function symmetric with respect to the inputs. Finally, it is possible to generate multiple graphs at one time. Producing graphs for each input is done as follows:

```
Model -> Graph -> x attribute -> ``[All inputs]''
                  Graph
```

When the data has more than two inputs, it may be desirable to graph contour plots on pairs of the inputs, while leaving the other inputs fixed at their query point values. This can be done in a way similar to the 1-d graphs.

Input attribute weightings

When the number of input dimensions is greater than one, a new issue arises. It is possible that each input should be treated differently when modeling the data. Some of them may be totally irrelevant while others can be treated more globally than others.

In the last example, the two input attributes were equally important to the output (in fact, their effect was identical). Next, we'll look at *b2.mbl*, which has two input attributes but the second one is completely irrelevant. This file was generated from the following equation: $y = \sin(x_1^2) + noise$

```
File -> Open -> b2.mbl
Edit -> Metacode -> Regression  A: Average
                    Localness   1: Super ultra local
                    Input Weights -> ``x2''  -: Ignore completely
Model -> Graph -> Dimensions  2
                  x attribute -> ``x1''
                  y attribute -> ``x2''
                  Graph
```

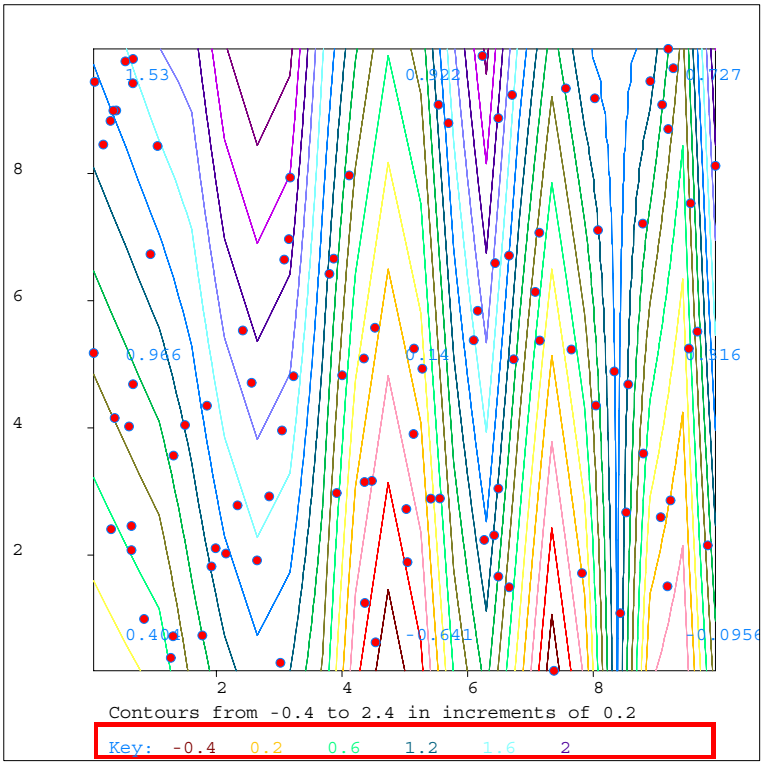


Figure 14: A two dimensional contour plot of $\sin(x_1^2) + x_2$ with a linear local model

The contour plot (shown in fig. 13) shows how the second input attribute is ignored (as we requested by editing the Metacode appropriately). Even though we know the second input attribute wasn't used to generate the data, we can verify its irrelevance visually by looking at the 1-d plots against each input as done in the last subsection on graphing. In this case, the 1-d curve does fit the data when graphed along the first input while leaving the second fixed. When doing the opposite, however, the curve is a flat line and the data appears scattered randomly everywhere. As before, we could change the place the flat line appears by adjusting the query point.

```
Model -> Graph -> Dimensions  1
                  x attribute -> ``[All inputs]''
                  Graph
```

Another common situation is that a certain input variable has a global effect on the output, rather than a complicated non-linear effect that must be modeled locally. This is the case with *c2.mbl*, which was generated by the following equation: $y = \sin(x^2) + y + noise$

```
File -> Open -> c2.mbl
Edit -> Metacode -> Regression  L: Linear
                    Localness   2: Ultra local
                    Input Weights -> ``x2''  0: Ignore in metric
Model -> Graph -> Dimensions  2
                  x attribute -> ``x1''
                  y attribute -> ``x2''
                  Graph
```

Using the label "Ignore in metric" means that input attribute will not be used when computing the distances between points (the same as saying the effect from that attribute is global), but it will be included in the regression equation. The contour plot (shown in fig. 14) shows how the fitted function varies non-linearly along the first attribute, but globally linearly along the second.

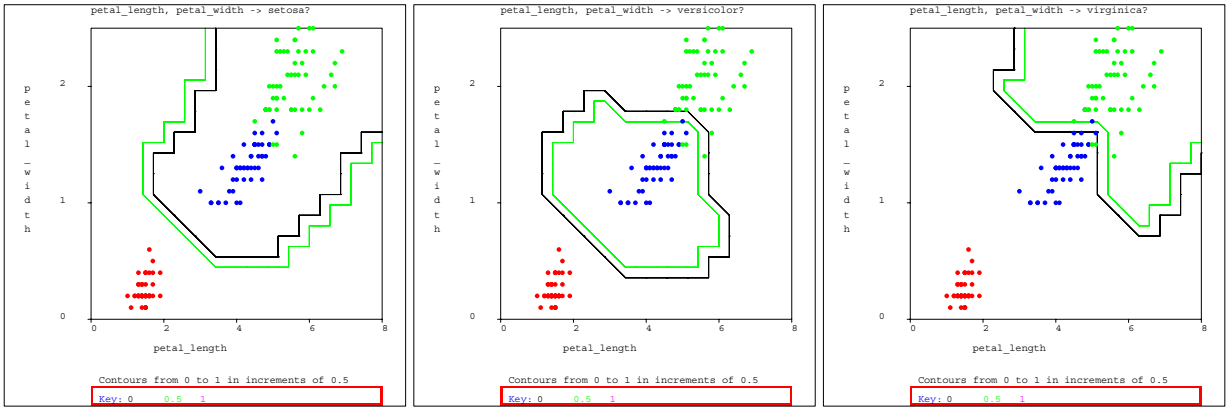


Figure 15: Classification on the iris data set. a) the setosa decision boundary, b) the versicolor decision boundary, c) the virginica decision boundary

3.6 Classification

Vizier is designed mainly for regression problems (predicting continuous valued outputs), but also does well on classification problems. In this section we'll see how it does classification and learn about some of the other features of Vizier . First, we'll look at the *format* and see how it can be changed. The format specifies which attributes in a data set are used for making predictions (the inputs), which are being predicted (the outputs), and which are unused. It is also specifies whether to do classification or regression.

```
File -> Open -> iris.mbl
Edit -> Format -> sepal_length -> Unused
                  sepal_width  -> Unused
```

While editing the format, note the check box specifying that this is a classification problem. The check was turned on automatically because the data file indicated that it contained class data. You may refer to the help on data file formats for more information on how this is done. The *extent* of each attribute is also available for editing in the format window. By default it is set near the high and low value of that attribute in the data set. The extent is used to scale the data for internal computations. Although it may be modified, in practice it is almost never useful or necessary to do so.

```
Edit -> Metacode -> Regression  A: Average
                    Localness   2: Ultra local
Model -> Graph -> Dimensions -> 2
                  x attribute -> petal_length
                  y attribute -> petal_width
                  z attribute -> setosa?
                  Graph
```

The resulting plot shows all the data points color coded according to their true class given in the data set. The contour lines show the decision boundary between the class *setosa* and the other classes. Often, it is desirable to see all the decision boundaries.

```
Model -> Graph -> z attribute -> [all outputs]
                  Graph
```

These plots are shown in fig. 15. Together they show the decision boundaries for all the classes. As with regression, the selection of an appropriate metacode is important for achieving accurate predictions. Experiment with other metacodes to see how they affect the decision boundaries for this data set.

HMO variables	Steel variables	Robot variables
Physician age	Line speed	Roll
Patient age	Line speed -10 mins	dRoll/dt
Charge/Day	Line speed -20 mins	ddRoll/dtt
Charge/Discharge	Slab width	Pitch
Discharges/1000	Slab height	dPitch/dt
ICD-9 diagnosis	Slab temp, stage 1	ddPitch/dtt
Market share	Slab temp, stage 2	Sonar height
Mortality/1000	Cool tunnel setpoint	Laser height
Patient ZIP	Cool tunnel temp	Flight time
Zip median age	ambient temp	Thrust rate

Figure 16: Examples of attributes in databases

4 Using Locally Weighted Learning for Modeling

At this point we take a break from examples using Vizier and discuss typical modeling applications where memory based learning can be useful. We will also discuss the pros and cons of memory based learning compared to neural networks, which are another commonly used function approximation tool.

4.1 "Hands-off" non-parametric relation finding

Suppose you run an HMO, or a steel tempering process, or a 7 degree-of-freedom dynamic robot arm. In each case, you have dozens of variables that may interact with each other as shown in fig. 16. You would like an intelligent assistant to spot patterns and regularities among pairs and triplets of the variables in your database. In particular, you would like to find more than just linear correlations. You would like to find significant non-linear relationships as well. Memory based learning is ideal for this application. Because it has no training phase (other than loading the data set), it can quickly check for relationships between many different subsets of variables in a database. Because it is locally weighted, it will even identify non-linear relationships in the data. These properties make it ideal for many data mining applications where the main goal is to extract previously unidentified, but interesting, relationships in data.

4.2 Low dimensional supervised learning

Suppose you have lots of data with a moderate number of input variables (less than 7, say) and you expect a very complex non-linear function of the data. Examples of this are:

- Skin thickness vs. τ, ϕ for a face scanner.
- A topographical map.
- Tumor density vs location (x,y,z).
- Amount of wasted aspirin vs (fill-target,mean-aspirin-weight, weight-std-dev,fill-rate) for an aspirin bottle filler.
- Object ball collision point vs position of object ball and angle of shot (x,y,θ) for a pool playing robot.

In each case, we already know exactly which features are relevant for predicting the outputs we're interested in, and there are a moderate number of them. We expect the relationship to be non-monotonic and non-linear, but we have plenty of data (although it's noisy) with which to determine the mapping. These are ideal applications for memory based learning because we can easily adjust the amount of localness in the approximator to best smooth out the noise in the data and model the desired relationship.

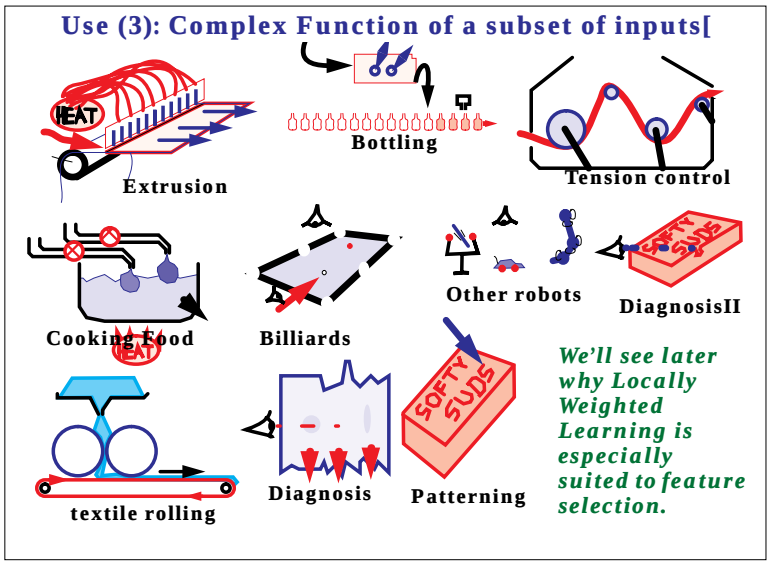


Figure 17: Example uses of locally weighted learning for modeling

4.3 Complex function of a subset of inputs

Fig. 17 shows a variety of applications where there are numerous measurements taken from a process, but the values we want to predict are a complex function of only a small subset of them. Later in this tutorial we'll show how Vizier does feature selection and why memory based learning is well suited to feature selection problems.

4.4 Simple function of most inputs, but complex function of a few

Another common situation in process control is that a particular output of interest is a function of a large number of variables, but the relationship to many of them is simple (globally linear, for example). We have already seen how Vizier can mix attributes that have globally linear effects with those that have complex non-linear effects. It is done by selecting input weights of 0 in the distance metric for the attributes that have globally linear effects, and non-zero weights for those with significant non-linearities. In fact, many of the applications shown in fig. 17 have this property.

4.5 Complex function of a few features of many inputs

Some types of data are very high dimensional. Examples of this are images, acoustic signals, and time series data. In each case, the data is often pre-processed to extract relevant features (as in fig. 18). Common feature extraction routines include principal components analysis, transformation to the frequency domain, and histogramming. The results of the feature extraction often yield a data set like those shown in section 4.3. Examples of this kind of application are:

- Mapping from acoustic signals to "probability of machine breakdown"
- Time series data analysis
- Mapping from images to classifications
- Product inspection, medical imagery, thin film imaging

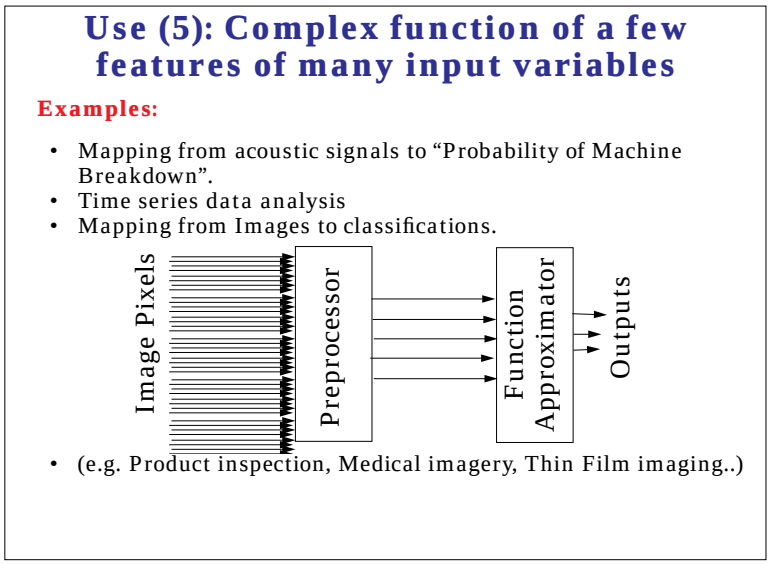


Figure 18: Examples of preprocessing high dimensional data for locally weighted learning

4.6 Pros and cons vs. neural networks

Neural networks have become a popular tool for function approximation and classification. Table 1 summarizes some of the tradeoffs between using them and using memory based learning. Some of these issues are described in further detail below, while others are addressed later in the tutorial.

Input dimensionality

Neural networks are generally more suitable for directly processing a large number of inputs. A typical example of this is learning from pixel values in images. A network usually has at least one node for each input variable. When there are many inputs, this results in a large number of additional parameters to determine during training. That means training may be slower and it can be more difficult to choose good learning rates, but neural nets have had success at processing images and other high dimensional input spaces.

Memory based learning with locally weighted regression is best suited for problems with a moderate number of input variables (40 or less). As the dimensionality increases, several things become problematic. The matrix inversion in the regression (see eq. 8) requires computation time that goes up as the cube of the number of dimensions. That can slow down the prediction speed significantly. The operation of weighting the data points and finding those nearest the query is also slowed down when the dimensionality increases. Even the efficient, tree-based data retrieval algorithms used by Vizier lose their zip as the dimensionality get high. The amount of data required for a good fit goes up with the dimensionality as well. Huge data sets are more problematic for memory based methods because they keep all the data, while neural nets discard the data once training is complete.

Despite these problems, memory based learning is still a good choice in many problems with high dimensional input. In some cases there are many input variables available, but the output is really only a function of a smaller number of them. Then the problem is to determine which inputs are relevant. Memory based learning is excellent for this kind of feature selection because it performs leave one out cross validation so cheaply (see the later sections on feature selection and cross validation).

Memory Based Learning	Neural Networks
Best suited for data with a moderate number of input variables	Can be trained directly on data with hundreds or thousands of inputs
Fits complex functions accurately	Can require considerable parameter tweaking and retraining to fit well
Confidence intervals, gradient estimates, and noise estimates are a natural part of predictions	Requires additional, inaccurate procedures to get confidences and other estimates
Training and adding new data is free	Training is computationally expensive
Predictions can be slow (although the methods used by Vizier alleviate this problem)	Once trained, predictions are fast
Does not forget old training data	Can suffer from "interference" in that new data can cause it to forget some of what it learned on old data
Cross validation is cheap	Validation is often done with a portion of the data held out from training

Table 1: Comparison between memory based learning and neural networks

Training and validation

A significant part of using a neural network is training it. Training is an iterative process where the data is repeatedly presented to the network and it incrementally improves its model to match the data more closely. Training is difficult because it can be computationally expensive and there are several parameters which must be adjusted in order to get it to learn well. The concept of training is somewhat foreign to memory based learning. Its training phase consists only of storing all the data its given.

The training required for neural nets presents an additional problem for them. In order to do feature selection, adjust and test different network architectures, and validate the learned model, it may be necessary to train several different networks on several different sets of data. The computation time required for training restricts the amount of testing that can be done with a neural net, while a memory based learner doesn't require any new training to test different subsets of data and different models.

Prediction speed

Neural networks make up for some of their slowness in training with fast prediction speed. As mentioned above, the prediction speed of memory based learners can suffer as the dimensionality increases. One solution to this problem is the efficient tree structure used by Vizier . It allows for fast finding of nearest neighbors and grouping of nearby data points for faster weighting and retrieval of information. These methods are described later in the section on kd-trees.

Interference and old data

Neural networks can suffer from a problem called *interference*. As mentioned earlier, a net is trained by repeatedly presenting data to it. The net then adjusts its parameters slightly to better represent the new data. The problem is that when a new set of data in one region of the input space is repeatedly presented to the net, it can forget the mapping it learned in other regions of the input space. This problem is referred to as interference. The forgetting behavior can be useful if the function being modeled is changing over time, because the new data will eventually erase the effects of the out-dated information.

Memory based learning does not suffer from interference. Since there is no training phase, there is no "loss of importance" of older data. Its all in memory. If the underlying function is changing through time, the confidence intervals provided with memory based learning can be used to explicitly determine when data should be discarded. Vizier does not do this automatically, but its confidence intervals can be used for this purpose (see the section on confidence intervals later).

4.7 When should locally weighted learning be used?

1. **Always!** We believe it is a good idea to approach all function approximation and data modeling first with locally weighted learning. It is a fast, efficient method of selecting features, validating models, estimating noise, and providing good insights into the relationships between variables.

2. **What if a global parametric model can be obtained?** Use the model locally and see *Rule 1!* Even if you have a global parametric model for your system, it often helps to fit it locally. Doing so can overcome inaccuracies in the global model caused by assumptions that only hold over a limited range of the state space.

3. **What if the data set is extremely large?** Buy more computer memory and processing and see *Rule 1!* The tree based algorithms used by Vizier make memory based learning efficient even in the face of large data sets. If the data set really is too big to keep around and process, it may be necessary to resort to other incremental learning algorithms that discard the data after training.

4. **What if predictions must be very fast?** Use caching methods and see *Rule 1!* When locally weighted learning produces accurate predictions, but faster speed is required, it is often useful to cache the predictions in a fast lookup scheme. The alternative is resorting to another function approximator that specializes in fast prediction times (neural networks can reach sub-millisecond prediction times).

5 The Information Provided by a Learned Local Model

We have seen how locally weighted learning can be used to build accurate models from data, and how predictions can be made from the models. However, predictions are not the only things we would like a model to give us. In many applications, we want estimates of gradients, and of noise, and confidence intervals or distributions for all predictions. As noted in table 1, it can be difficult to get these things from many function approximators, but it can be done easily with locally weighted learning methods. The local models fit by Vizier are well understood statistically, and most common regression analysis techniques can be converted to local forms in order to provide the information we want. In this section, we summarize how prediction, noise, and gradient distributions are done.

5.1 Prediction distributions

We often think of a prediction as a single scalar (or vector) value that the model predicts will be the output from a given input. In fact, the prediction from a regression model is a probability distribution on the values that could be output. The single prediction we are used to seeing is just the mean of that distribution. The distribution is a a t distribution (see a statistics text for a description of t distributions).

When regression is used to fit a model, the result is a multi-dimensional t distribution on all the coefficients of the model. The t distribution for the output is found by making a projection from the joint distribution on the coefficients. The projection is based on the attribute values of the input query.

Now, we'll use Vizier to see what confidence intervals look like on a small, 1-d data set:

```
File -> Open -> d4.mbl
Edit -> Metacode -> Regression   L: Linear
                    Localness    4: Very local
                    Neighbors    0: No Nearest Neighbors
Model -> Graph -> show_confidences ON
               -> Graph
```

Fig. 19a shows a graph similar to what Vizier produces (later we'll see how to make them look exactly alike). The middle curve is the predicted function given the data points. The other curves are the upper and lower confidence intervals on the function. That means that for each input point, we expect the true mean output to lie outside the range between the upper and lower curve only 5% of the time. In all cases, the predicted function lies exactly between the upper and lower bounds. There are several things that affect the width of these confidence intervals. On the left side, the confidence intervals are wider because the data

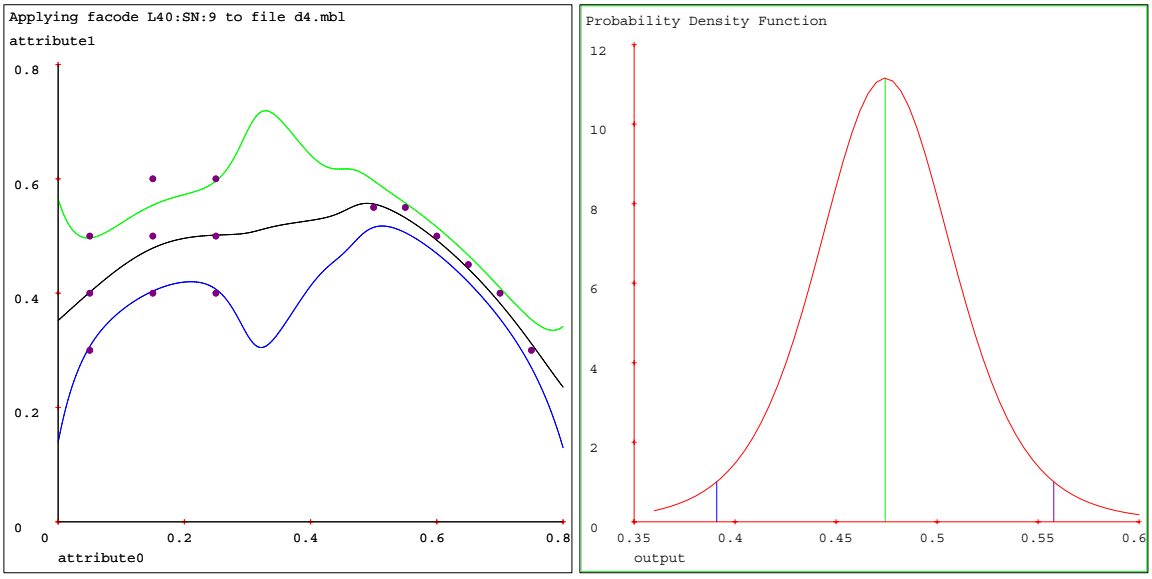


Figure 19: a) Fitted 1-d function with confidence intervals, b) Distribution of output variable at input query 0.15

points show widely varying outputs at particular input values. In the middle, the confidence intervals are even wider because there is no data in that vicinity. Farther right, they are narrow because all of the data lines up exactly and has no noise.

We can also ask Vizier to show what the distribution looks like at a particular input point:

```
Edit -> Query -> ``input'' 0.15
Model -> Graph -> Dimensions 0
                  Graph
```

Fig. 19b shows the graph. Compare the horizontal axis in fig. 19b with the vertical axis values of the curves in fig. 19a at the input point 0.15. The prediction and confidence interval values match. Notice that the shape of a t distribution is similar to a normal (Gaussian) distribution.

A note on graph scaling

The graph given by Vizier did not exactly match fig. 19a. We can make them the same by adjusting the axis scaling. Vizier is good at choosing default scales for the axes in its plots, but they can be manually adjusted:

```
Model -> Graph -> Dimensions  1
                  Axis x Auto  OFF
                  Axis x Low 0.0
                  Axis x High 0.8
                  Graph
```

5.2 Noise estimate distributions

The noise estimate from a regression is a gamma distribution on the inverse of the variance of the noise, $\frac{1}{\sigma^2}$. Vizier computes noise estimates with localized versions of statistics from classical regression analysis. Fig. 20a shows noise estimates for *d4.mbl*.

```
Model -> Graph -> Dimensions  1
                  Graph type  Noise
```

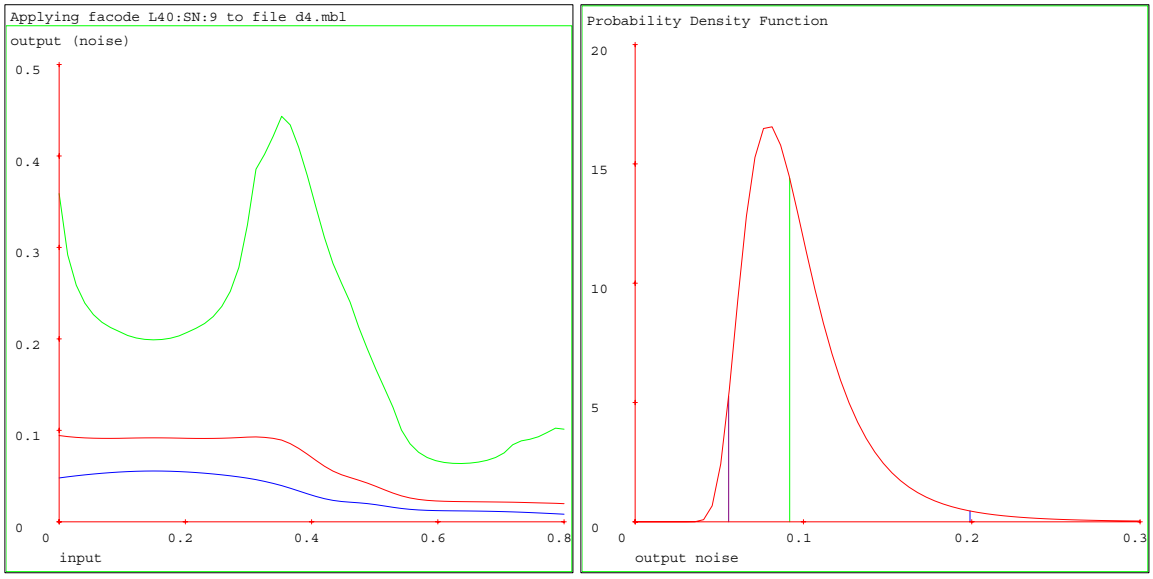


Figure 20: a) Noise estimates for a 1-d function with confidence intervals, b) Distribution of noise at input query 0.15

```
Show confidence  ON
Show datapoints  OFF
Graph
```

The Vizier graph may not match the figure exactly. In order to make them the same, adjust the x axis to a range of 0 to 0.8 and the z axis to a range of 0 to 0.5. Vizier is happy to plot the data points on this graph, but we turned it off because it doesn't make much sense. The data points would be plotted using the vertical axis as the output value of each point, while the noise estimate curves use the vertical axis as the estimated standard deviation at the input indicated by the horizontal axis.

The noise estimate graph shows the predicted, and high and low confidence intervals for the noise (standard deviation). The noise is higher near the left side of the graph where there was obvious noise in the data, and lower near the right side where the data appears noiseless. As before, the confidence intervals get wide where there is little data and near the edges.

Just as with the prediction, we can look at the distribution for the noise estimate at a single input query, as shown in fig. 20b.

```
Edit -> Query -> ``input'' 0.15
Model -> Graph -> Dimensions 0
```

In order to make the Vizier graph match the figure, the x axis should be scaled to a range of 0 to 0.3. As before, the predictions and confidence intervals in fig. 20b match those in fig. 20a at input point 0.15. Notice that this distribution is not symmetric. The upper confidence bound on the noise is usually farther above the predicted noise, than the lower confidence bound is below it.

5.3 Gradient estimate distributions

Gradient estimates are very useful for controllers and optimizers. Gradient descent is a popular optimization method that requires an estimate of the derivatives of the output with respect to each input at a certain point. It uses those derivative estimates to determine which direction in the input space to move in order to get a better output. Similarly, an intelligent controller tries to drive the output of a system toward some target by adjusting the control variables. Again, the derivative of the output with respect to each of the control variables can be used to decide which direction to move the controls.

Many common function approximators have difficulty providing gradient estimates, and even more difficulty putting confidence intervals on those estimates. Often they must resort to making several predictions in the vicinity of the query point. They can approximate the gradient by computing the difference in the predictions as the input is changed. That technique can be computationally expensive, and may also cause wild fluctuations in gradient estimates when the predictions are not smooth or the size of the region from which to make predictions is poorly chosen.

Using regression analysis we can get gradient estimates directly from the fitted model rather than requesting numerous additional predictions. An estimate of the derivative of the output with respect to one input is computed in a way similar to the prediction. A projection is done from the joint t distribution on the coefficients to a single dimensional t distribution for the gradient. A simple case of this is when we want to know the gradient of an output with respect to a particular input in a linear model. Then the distribution of the gradient is the same as the distribution for the coefficient on that input. If we have an averaging model, then the gradient estimate will always be 0. For a quadratic model, the computation of the gradient with respect to one input uses the coefficients on all the terms made from that input.

You may experiment with gradient graphs on your own. Notice that the graph dialog box asks you to specify both an x attribute and a gradient attribute. Often, they will both be the same input variable indicating that you would like to see the derivative of the output with respect to an input as that input varies. However, with multi-dimensional input data, it makes just as much sense to plot the derivative of the output with respect to input 1, as input 2 is varied.

5.4 Other things provided by local weighted models

There are many other useful things we can compute from local weighted models that are not available in version 1.0 of Vizier . In this section we discuss predicting the distribution of future responses, estimating the probability that a local minima exists within a region of interest, and estimating the probability that the steepest gradient lies within a specified solid angle. If you are reading this tutorial strictly to learn about Vizier , you may skip the rest of this section.

Look at the data points in fig. 19a. At least 6 of the data points lie at or outside the confidence intervals, which might seem to contradict the statement that we are 95% sure of the function being between the intervals. Actually, there is no contradiction. When there are lots of data points with lots of noise, the regression can be very confident about what the average response is for a particular input, even though the large amount of noise means that most of the data points are outside its confidence intervals. The mean of those points is still very likely to be inside the confidence intervals. Another question which is meaningful, though, is *what are the confidence intervals on where future data points will be?* These are specified such that we expect 95% of all future data points to lie within the intervals. Of course, in the high noise, many data points example, the future data points confidence intervals would be wide. These intervals would be useful if we wanted to build a controller that would operate a system safely, even in the presence of high noise.

Questions that may be of interest to an optimization routine are: *Is there a local optimum in this region of interest? With what confidence is there one?* These questions can be answered with locally weighted learning when quadratic models are used. For a quadratic model it is possible to determine its optimum in closed form (see a basic Calculus text to see how). In order to answer the first question above, it is only necessary to see whether an optimum lies in the region of interest. If we want a probabilistic estimate, we can use Monte Carlo sampling. We can look at the joint t distribution on the coefficients of the quadratic model, and then randomly choose coefficient values by sampling from that distribution. The whole algorithm is as follows:

1. Use locally weighted learning to determine the joint t distribution on the coefficients of a quadratic model in the center of the region of interest.
2. Choose a single value for each coefficient by sampling from the coefficient distribution.
3. Compute the local optimum given the chosen coefficients.
4. Determine whether it falls in the region of interest and increment a counter appropriately.

5. Repeat to step 2 a number of times.

6. Report the percent of times the optimum fell within the region of interest.

An optimization routine may also wish to estimate the probability that the steepest gradient lies within a certain solid angle, or that the gradient in a particular direction is greater than or less than a certain value. If the optimization routine is using a policy of collecting data in a certain region until it is confident about which direction it should shift the region of interest to in order to get better results (part of a technique called Response Surface Methodology), this is exactly the kind of information it needs to know. We have already seen that derivatives can be computed directly from a model. The algorithm for computing probabilistic estimates of those gradients is like the algorithm just given for estimating the probability of a local optimum.

5.5 Why do we want all these estimates?

In this section we have described numerous estimates that can be obtained by using locally weighted learning for function approximation. One question to ask is *what are all these estimates good for?* Although learning algorithms are almost always described in terms of their ability to make predictions, that is not their purpose. A good learning algorithm should help us *make decisions*. If a friend calls you up and says "I think this stock will double in price this year.", do you immediately go out to purchase the stock? Its more likely that you reply with "How sure are you that this will happen?" Making decisions based on the advice of learning algorithms is the same way.

We have already observed that optimization algorithms can make use of these estimates. Confidence intervals are also useful for computing the expected gains or losses of potential decisions. Some possible outputs may represent catastrophic consequences and any decision that has a significant chance of yielding those results should be avoided. Later on in this tutorial we will see more examples of confidence intervals being used in decision making.

6 Bayesian Locally Weighted Regression

Regression analysis can have a serious problem if there is not enough data present. The immediate result of insufficient data is that the matrix being inverted in eq. 8 becomes singular and the inversion fails. That means the data is insufficient to determine all the coefficients in the regression.

This problem shows up frequently when using locally weighted regression for decision making. The local weighting is part of the problem. It is possible to have a fairly large data set, but still make queries that fall well outside the regions covered by the data. In that case, the weighting function will weight all the data near zero and the situation will be just like the case where there really is no data. One scenario where this happens is when a controller is considering the potential affects of choosing a control decision that has never been tried before. Since it's never been tried before there is little or no data to estimate what will happen. Another scenario is when a decision is being made about what data to collect. Often, the best place to collect data is where there is not much data already and making queries on the model to find these places will cause the insufficient data problem to show up.

Bayesian regression can be used to avoid the insufficient data problem. It allows us to easily specify prior information about what values the coefficients should have when there is not enough data to determine them. To see how Bayesian regression works, we first review Bayes' rule with a simple example (see fig. 21). The important thing to understand about Bayes' rule is that it combines prior information (initially there is an equal chance of choosing from any bucket), with data (the chosen ball was white) to obtain a posterior estimate (that the probability of buckets A, B, and C is 0.44, 0.22, 0.33).

With Bayesian regression we specify a joint prior distribution on the coefficients and the noise called a normal-gamma distribution. The prior distribution on the inverse of the variance of the noise is a gamma distribution and the prior on the coefficients given a particular level of noise is a normal (Gaussian) distribution. Bayesian regression provides us with a way to combine those priors with the data to yield the posterior distributions on the coefficients and the noise described in the last section. The details are described in a text by DeGroot [3].

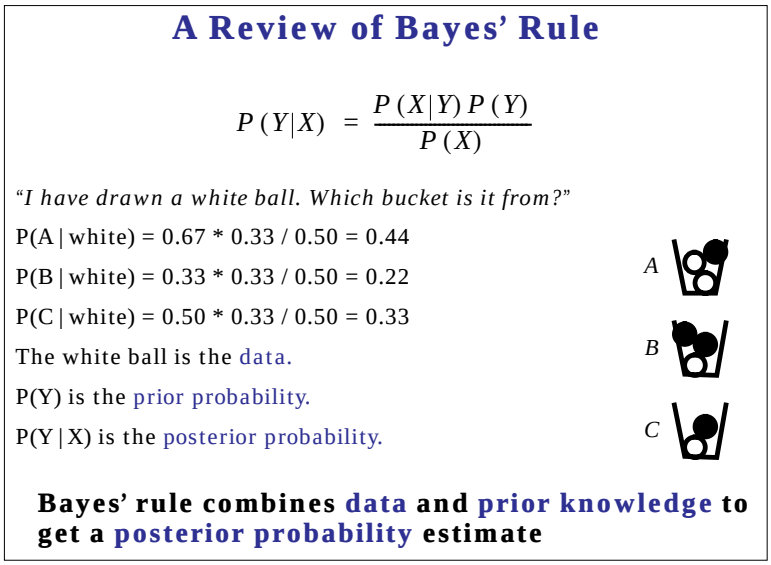


Figure 21: An illustration of Bayes' rule

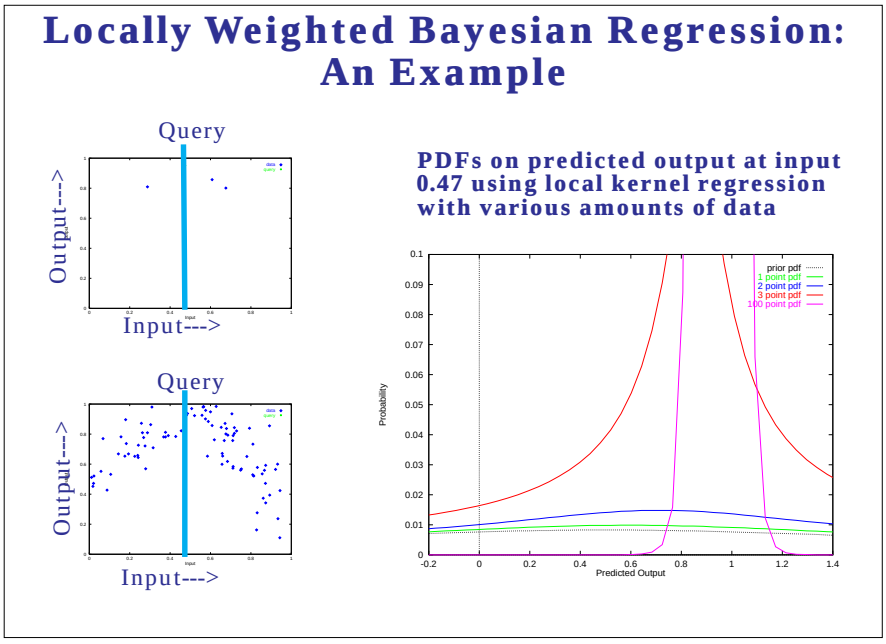


Figure 22: Example with Bayesian locally weighted regression. See text for explanation

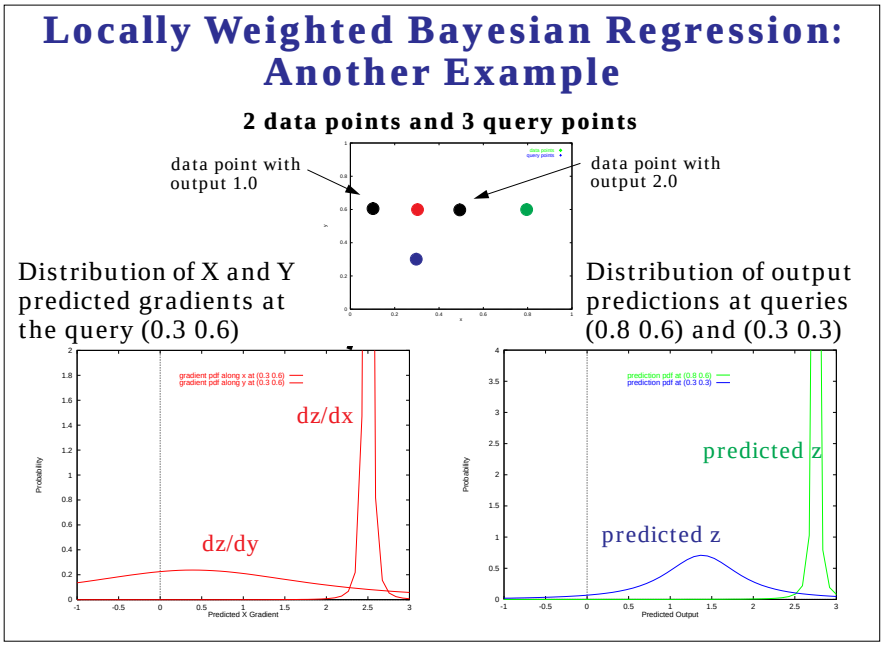


Figure 23: Example with Bayesian locally weighted regression. See text for explanation

Fig. 22 shows one example of what using Bayesian locally weighted regression does. The plot at the lower left shows a data set we are modeling. The plot above it shows the first few data points. The large plot on the right shows the distributions of the predicted outputs as more data points are included. The curves are like the 0-d prediction distributions we saw earlier. The lowest curve is the prior distribution for the output at a query of 0.47, which is what we get if we ask for a prediction at that point without giving it any data points. You can see what this looks like in Vizier by loading the data file *empty.mbl*. You'll have to rescale the axes to get a plot like the one in the figure. The next higher curve is obtained by giving it only one of the data points from the upper left hand plot. The next curve is from two data points. The distribution is gaining a slight hump around 0.8. Even with two data points, there is still not enough information to complete the regression using the basic regression method of eq. 8. It is the fact that we are using Bayesian methods, that allows the curves to be drawn. Finally, when the third data point is included, there is enough evidence to show some confidence in the output at input 0.47. Even with these three data points, though, the distribution shows that we shouldn't be surprised if the true mean output at input 0.47 is as low as 0.0 or as high as 2.0. Finally, the last curve shows what the distribution looks like after all 100 data points are included. The model is very confident that the true output at input 0.47 lies between 0.7 and 1.2.

Fig. 23 is another example demonstrating Bayesian regression in action. In this example, there are two input variables (labeled x and y) and one output variable (labeled z). There are only two data points as pointed out in the plot at the top. We want to fit a linear model to the data, but these two points are insufficient. Using Bayesian regression allows us to finish the computation and get reasonable confidence intervals anyway. Suppose we are interested in the gradient at the query half way between the two data points (we would like to know $\frac{dz}{dx}$ and $\frac{dz}{dy}$ at the point (0.3,0.6)). The plot at the lower left shows what these two distributions look like. Since the two data points are aligned horizontally, we can make a good estimate of $\frac{dz}{dx}$ and the distribution shows that it is probably between 2.0 and 3.0. There is no information in the y direction, though, and the distribution for $\frac{dz}{dy}$ show great uncertainty with only a slight preference for a derivative near zero. Now suppose we want to make a prediction of the output at the query (0.8,0.6). Since the query is aligned with the data points, we get a relatively confident prediction. However, if we query at (0.3,0.3), the prediction is unconfident with only a slight preference in the range of outputs from the two existing data points.

These two examples show some of the benefits of using Bayesian locally weighted regression. The main thing to remember about it is that we don't have to worry about the problem of insufficient data. The

computation completes without any numeric difficulties, and we get wide confidence intervals when we ask questions for which there isn't enough data to support a good answer.

7 Efficient Data Storage and Retrieval

When data sets get large, memory based learning can require a significant amount of computation just to accumulate all the relevant statistics needed to make its prediction. Vizier uses a data structure called a kd-tree to speed that process up tremendously. Kd-trees are data structures similar to binary trees, except that they are used to store continuous valued multi-dimensional data. This tutorial will not cover the details of kd-trees, which can be found in a good computer science algorithms text, or the specifics of the method used by Vizier which can be found in [4].

There is one aspect of the trees that is important to the Vizier user. Using the trees it is possible to trade off prediction speed and accuracy. When editing the metacode, you may have noticed a field labeled "kd-tree." Its choices are Slow, precise; Medium; Fast, approximate. This choice adjusts the speed and accuracy of predictions. The default is Slow, precise, which is often fine for moderate sized data sets. Medium is a safe choice that can speed predictions up considerably. Fast should not be used unless you want only approximate predictions.

A note on graphing and metacodes

```
File -> Open -> mpg.mbl
Edit -> Format -> american -> unused
                  european -> unused
                  asian    -> unused
Edit -> Metacode -> Metacode for Super Users:  L90:SN:{9}
```

All of the fields in the metacode editor can be expressed in one convenient shorthand string shown at the bottom of the editor. In the string "L90:SN:{9}", the 'L' means "Linear, the '9' is for localness, the '0' is for number of nearest neighbors, the 'S' is for Slow kd-tree, the 'N' is a place holder reserved for future use, and the '{9}' is short for "all the inputs receive weight 9. Alternatively, we could have specified it as "L90:SN:999999999", which lists the 9 for each input separately. Also, the middle part can be left out if "Slow" is the desired speed, as in "L90:{9}." From this point on in the tutorial, we will refer to all metacode settings according to this shorthand notation.

```
Model -> Graph -> Graph type   Predict
                  Dimensions   2
                  x attribute  weight
                  y attribute  displacement
                  Show prediction  ON
                  Show confidence  OFF
                  Show datapoints  ON
                  Graph
```

When the data set has several inputs or several outputs, we can select which ones to plot for any particular graph.

```
Model -> Graph -> x attribute [all inputs]
                  y attribute [all inputs]
                  Graph
```

The first message displayed by Vizier states that a total of 15 graphs will be drawn matching each input with each other input and showing the predicted output. Then the graphs are drawn one after the other. If you do not want to wait for them all to complete, you can hit 'q' to stop part way through.

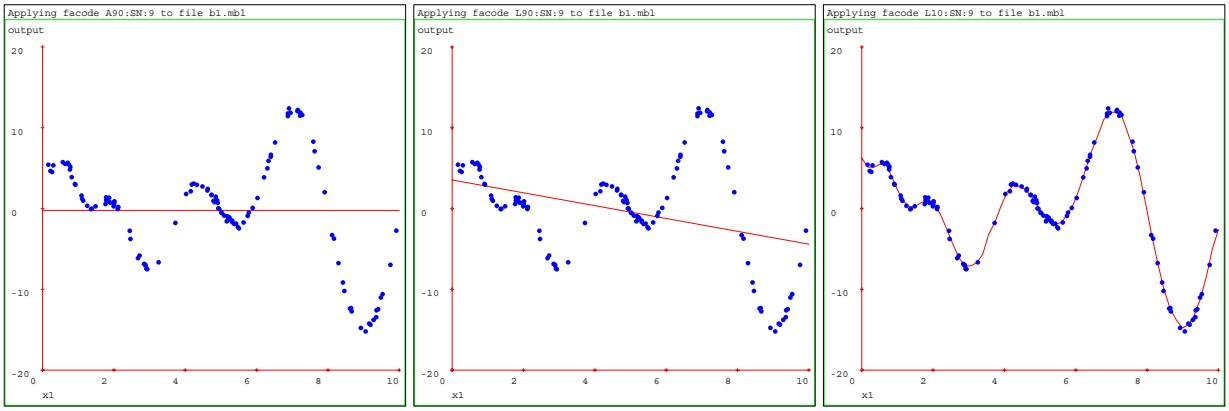


Figure 24: Approximating a one-dimensional data set with A90:9, L90:9, L10:9 metacodes. The residual error for each data point is the distance along a vertical line between it and the fitted line. The result is very large, large, and zero residual error, respectively.

8 Autonomous Modeling

We have been specifying what metacode to use for each data set. A question you may have been wondering is *How do I figure out what metacode to use for my data?* As we saw at the beginning, the metacode choice can have a big impact on how the predictions turn out. Fortunately, Vizier automates this choice for us. This section describes how that choice is made.

Before talking about the details, we'll first see how easy it is to automatically choose a good metacode.

```
File -> Open -> a1.mbl
Model -> Blackbox -> Launch!
```

Blackbox spends 60 seconds looking for a good metacode and reports what it has found when it is done. Depending on the speed of the machine you're using, it will choose something like "L19:MN:9." The metacode is automatically set to the choice it has made. You can use graph to see visually whether you think it has made a wise choice. There are several other ways to evaluate its choice which will be covered in the rest of the section.

8.1 Judging Model Quality by Residuals

One method of judging the quality of a particular model is by *residuals*. That means the model is fit using all the data points and the prediction for each data point is compared with its actual output. The absolute value of each error is taken and the mean of those values is computed to arrive at the mean absolute residual error. Models with lower values of this measure are deemed to be better.

Fig. 24 shows an example where choosing the model with the lowest residual error is a good idea (the data comes from *b1.mbl* if you want to load it into Vizier). The fit on the right is clearly the best fit of the three for the data and its mean absolute residual error is near zero. However, choosing models by residual error is a risky thing to do.

Fig. 25 shows an example (from *a1.mbl*) where residuals can lead us astray. Again, the residual error in the middle plot is moderate, and the residual error on the rightmost plot is near zero. The middle plot is a much better fit though. The reason is that the rightmost plot is fitting the noise in the data. This phenomenon is referred to as "overfitting" and is a common problem that must be avoided in learning systems. Overfitting in this example means that errors in predicting future data points from this curve will actually be higher than if we use the middle plot's fit instead. In general, it is preferable to use something more trustworthy than residual error to choose a good model and avoid overfitting.

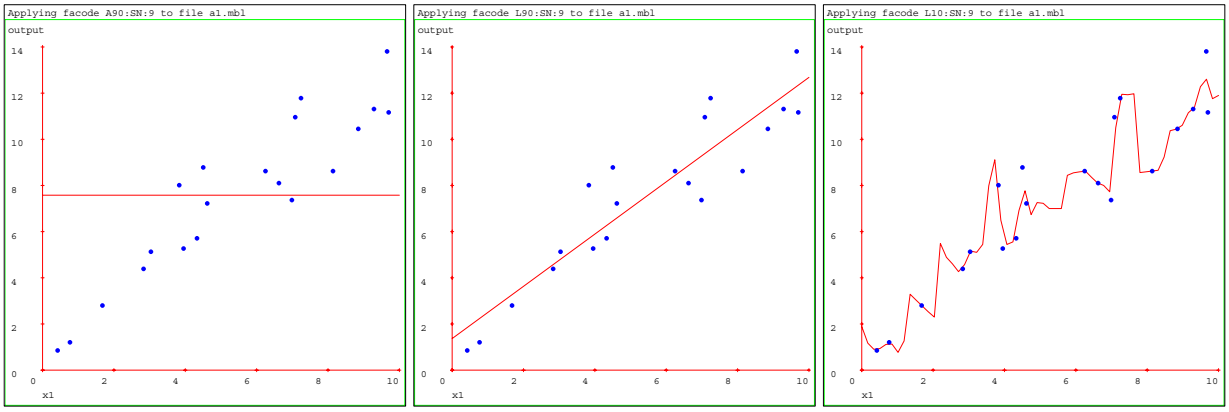


Figure 25: Approximating a one-dimensional data set, with A90:9, L90:9, L10:9 metacodes. The residual error for each data point is the distance along a vertical line between it and the fitted line. The result is very large, small, and near zero residual error, respectively.

8.2 Cross Validation

Cross validation is a model evaluation method that is better than residuals. The problem with residual evaluations is that they do not give an indication of how well the learner will do when it is asked to make new predictions for data it has not already seen. One way to overcome this problem is to not use the entire data set when training a learner. Some of the data is removed before training begins. Then when training is done, the data that was removed can be used to test the performance of the learned model on "new" data. This is the basic idea for a whole class of model evaluation methods called *cross validation*.

The **holdout method** is the simplest kind of cross validation. The data set is separated into two sets, called the training set and the testing set. The function approximator fits a function using the training set only. Then the function approximator is asked to predict the output values for the data in the testing set (it has never seen these output values before). The errors it makes are accumulated as before to give the mean absolute test set error, which is used to evaluate the model. The advantage of this method is that it is usually preferable to the residual method and takes no longer to compute. However, its evaluation can have a high variance. The evaluation may depend heavily on which data points end up in the training set and which end up in the test set, and thus the evaluation may be significantly different depending on how the division is made.

K-fold cross validation is one way to improve over the holdout method. The data set is divided into k subsets, and the holdout method is repeated k times. Each time, one of the k subsets is used as the test set and the other $k-1$ subsets are put together to form a training set. Then the average error across all k trials is computed. The advantage of this method is that it matters less how the data gets divided. Every data point gets to be in a test set exactly once, and gets to be in a training set $k-1$ times. The variance of the resulting estimate is reduced as k is increased. The disadvantage of this method is that the training algorithm has to be rerun from scratch k times, which means it takes k times as much computation to make an evaluation. A variant of this method is to randomly divide the data into a test and training set k different times. The advantage of doing this is that you can independently choose how large each test set is and how many trials you average over.

Leave-one-out cross validation is K-fold cross validation taken to its logical extreme, with K equal to N, the number of data points in the set. That means that N separate times, the function approximator is trained on all the data except for one point and a prediction is made for that point. As before the average error is computed and used to evaluate the model. The evaluation given by leave-one-out cross validation error (LOO-XVE) is good, but at first pass it seems very expensive to compute. Fortunately, locally weighted learners can make LOO predictions just as easily as they make regular predictions. That means computing the LOO-XVE takes no more time than computing the residual error and it is a much better way to evaluate models. We will see shortly that Vizier relies heavily on LOO-XVE to choose its metacodes.

Fig. 26 shows an example of cross validation performing better than residual error. The data set in the

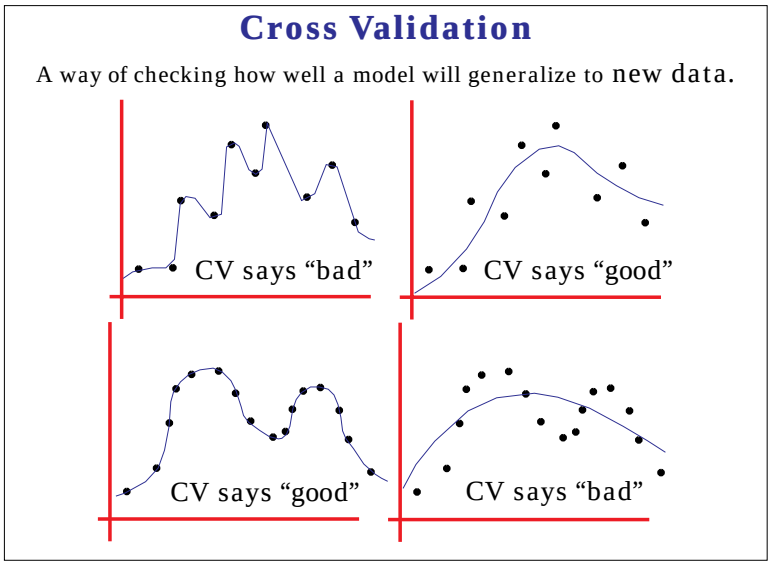


Figure 26: Cross validation checks how well a model generalizes to new data

top two graphs is a simple underlying function with significant noise. Cross validation tells us that broad smoothing is best. The data set in the bottom two graphs is a complex underlying function with no noise. Cross validation tells us that very little smoothing is best for this data set.

Now we return to the question of choosing a good metacode for data set *a1.mbl*:

```
File -> Open -> a1.mbl
Edit -> Metacode -> A90:9
Model -> LOOPredict
Edit -> Metacode -> L90:9
Model -> LOOPredict
Edit -> Metacode -> L10:9
Model -> LOOPredict
```

LOOPredict goes through the entire data set and makes LOO predictions for each point. At the bottom of the page it shows the summary statistics including Mean LOO error, RMS LOO error, and information about the data point with the largest error. The mean absolute LOO-XVEs for the three metacodes given above (the same three used to generate the graphs in fig. 25), are 2.98, 1.23, and 1.80. Those values show that global linear regression is the best metacode of those three, which agrees with our intuitive feeling from looking at the plots in fig. 25. If you repeat the above operation on data set *b1.mbl* you'll get the values 4.83, 4.45, and 0.39, which also agrees with our observations.

8.3 Blackbox Model Selection

Vizier's Blackbox uses LOO-XVE to determine which is the best model for a particular data set. A summary of the full Blackbox algorithm is as follows:

1. Load the relevant data for the *Cross Validation Expert*.
2. Load the relevant data for the *Cross Validation Police*.

3. Generate a new metacode according to a search algorithm that is guided by the *Cross Validation Expert*.

4. Test the new metacode with cross validation and record the results.

5. While there is still more time left, repeat to step 3.

6. Select the metacode with the best evaluation according to the *Cross Validation Police*.

Frequently, the *CV Expert* (also referred to as the "inner loop" since it guides the internal searching) is LOO-XVE. It is used to guide the generation of new metacodes to be considered. At each step, the chosen search algorithm will generate several new metacodes as modifications of the best metacodes seen so far according to the *CV Expert*. Even though LOO-XVE is a great way to avoid overfitting, even it can suffer from it occasionally. If enough metacodes are tried, it may find one that just got lucky and produced low LOO-XVE even though it will do poorly on future predictions. In order to add an extra level of protection, Blackbox checks the *CV Police* (also called the "outer loop" since it controls the overall final selection) and finds the metacode that does best according to it as its final choice. The default setting for the *CV Police* is a random 35% of the original data set that is kept out of the data used by the *CV Expert*. This extra measure of protection against overfitting is usually enough to assure a good metacode is chosen, but we will see an example later where a different kind of *CV Expert* and *CV Police* should be used.

You may have noticed when you ran Blackbox on *a1.mbl* earlier, that the *CV Police* section of the dialogue box was set to file "None." That indicates that there will be no separate *CV Police* evaluation. Blackbox sets it default that way because *a1.mbl* has so few data points. If a small data set is further subdivided, there won't be many points for the *CV Expert* to run on and there won't be enough for the *CV Police* to make a good check either.

The easiest thing to do when you get a new data file is just run Blackbox on it as we did on *a1.mbl*. However, Blackbox is quite a versatile tool and we will use the rest of the section to go over some of the options available. We'll also show some of the problems that can arise with automatic model selection via cross validation and what can be done to avoid them.

Searching specific sets of metacodes

Earlier, we manually tried different metacodes and asked Vizier to evaluate them with LOO-XVE so we could use the best one. Its much easier, though, to give Blackbox a list of metacodes and have it automatically go through them and report back the best one. In this example, we'll use Blackbox to determine the best number of nearest neighbors to use in a kernel regression.

```
File -> Open -> c1.mbl
Edit -> Metacode -> A01:9
Model -> Blackbox -> Search Algorithm         Best Neighbors
                     Cross Validation Police  Use Testset
                     Testset:                 None
                     Reset                    ON
                     Launch!
(Repeat for files k1.mbl and i1.mbl)
```

First, we set the metacode to one that we'd like the search to begin from. Then we choose which search algorithm to use. The default is a totally autonomous search. "Best Neighbors" means that the search will try all the different number of nearest neighbors in conjunction with the other settings specified by the metacode we just chose. The *CV Police* is set to use a separate test set. This feature allows you to separate your data manually into training and testing data. The filename "None" is a special case which means do not use anything for the *CV Police*. We chose that in this example because we want to choose a metacode based strictly on LOO-XVE. The reset feature means we want to clear all previous Blackbox work. If we had run Blackbox before on this data set, the search would begin from the best previous metacode found. Since we want it to start from the metacode just specified, we tell Blackbox to erase any previous results first. Launch begins the search. Depending on the speed of your computer, you will see the metacodes being tried

and their performance in the status bar at the bottom of Vizier's window. At the end, Blackbox displays its choice as the best metacode.

Similarly, we can use Blackbox to find the best kernel width for a local linear regression. In order to do that, we would start by setting the metacode to "L10:9", and then choosing the "Best Smoothing" search algorithm in the Blackbox dialogue box.

It is also possible to use the Blackbox dialogue box to eliminate some kinds of metacodes from consideration. This is done with the array of check boxes. For example, suppose we have a high dimensional data set and we know it will be impractical to fit quadratic models to it. Quadratic models can be eliminated from the search by unchecking the "Quadratic" check box.

Fast feature selection with LOO-XVE

Feature selection is an important part of modeling when the dimensionality of a data set becomes large. Even when there are many attributes, it is common that only a small subset are needed for good predictions. Finding that subset is something Blackbox is very good at because of the speed at which LOO-XVE can be done.

In order to demonstrate the feature selection capability of Blackbox, we'll give it a data file with 8 inputs and 1 output where only 4 of the inputs are relevant to the output. The data file was synthetically generated as 800 points from the function: $y = x_2 + x_4 + 4\sin(0.3x_6 + 0.3x_8) + noise$. The function is globally linear in x_2 and x_4 and nonlinear in x_6 and x_8. We'll use Blackbox to find the best metacode for this function.

```
File -> Open -> 4of8.mbl
Blackbox -> Seconds  300
            Launch!
```

The length of time required to find the best metacode will vary depending on the machine, but 300 seconds should work on many PCs. You can watch its progress during the search. It almost immediately finds the four relevant features and observes that a very local kernel regression does well (metacode A30:-9-9-9-9). Next, it finds that linear regression is better than kernel regression with these features (metacode L30:-9-9-9-9). This is no surprise since the function is globally linear in two of the features. After further exploration, it finds the two globally linear features (metacode L30:-0-0-9-9). Finally, it observes that when the two globally linear features are handled globally, it can make the smoothing even more local for the best model of L20:-0-0-9-9.

Time series data

Data points that are acquired as a series of readings over time can cause additional modeling difficulties. The problem is subtle, but common when the data is taken from a process that does not change quickly compared to the rate at which the data is taken. In that situation, the corresponding inputs and outputs of data points are very close to the inputs and outputs of the adjacent data points in time. That means the output of one data point can be predicted accurately by just finding its near neighbors in time and predicting the average of their outputs. How can a learner trick itself into doing that? Because the inputs of the near neighbors in time are similar, if it uses an approximation method that is very local in the input space, it will effectively identify the near neighbors in time. Why is this bad? When this kind of model is used to predict new data in the future it will fail because it no longer has the near neighbors in time to look at. In fact, some of the near neighbors in time are further into the future than the point being predicted so there's no hope of them being available. Note that when LOO-XVE is used, or when a random subset of points are taken out (as is the default for the *CV Police*), the near neighbors in time, including those occurring after the query point, are usually available in the training data.

We will use Vizier to observe an instance of this problem, and see how it can be avoided. The data file for this example is from the semiconductor industry. The sixteen input attributes are of various sensor readings available during an etching process. The output variable is a measure of quality that only becomes available much later in the processing. The factory managers would like to estimate the quality from the sensor readings. Then they can discard the bad wafers immediately rather than spending precious resources to finish the processing, only to find out at the end that they are defective. The data file has the time series

problem because the readings are taken from a series of wafers as they go through production and the sensor readings and quality of successive wafers are similar.

```
File -> Open -> semitrain.mbl
Edit -> Metacode -> A31:{9}
Model -> Blackbox -> Launch!
```

The actual metacode chosen by Blackbox will vary depending on the speed of your computer (because Blackbox runs for a chosen number of seconds). The following description is based on one particular run, but your results should be qualitatively similar. Blackbox chooses the metacode A30:FN:99—-99—–999-. It reports that simply choosing the global average output would result in an error of 1.387. The chosen metacode gets an error of 0.535, which is a 59.8% improvement. At this point a factory manager might feel that he can do a good job of predicting wafer quality from his sensors.

In order to see the problems he will cause, we have a separate data file taken from the same process, but much later in time. We can use the batchpredict command to see how well the quality for these future wafers can be predicted using the original data set and the metacode found by Blackbox.

```
Edit -> Metacode -> A30:FN:99----99-----999-
Model -> Batchpredict -> Show_errors  ON
                         Testfile:    semitest.mbl
                         Batch Predict
```

It turns out that the error for predicting this data set is 1.667, which is even worse than always predicting the average output! Exactly the problem described above has struck here. If the factory manager had implemented his plan, he would have caught very few of the defective wafers and discarded many good ones.

There is a way to safeguard against this when modeling time series data. The data set should be manually separated such that a test set is made from a single section of time, or a second data set can be taken at a completely different time. Then the *CV Police* can be set to use the other data set rather than randomly subsampling from the training data set. Although we don't do it here, we would get the most efficient searching by separating out a third data set and giving to the *CV Expert*.

```
Edit -> Metacode -> A31:{9}
Model -> Blackbox -> Reset      ON
                     CV Police  Use testset
                     Testset    semitest.mbl
                     Launch!
```

This time Blackbox chooses the metacode L90:9, which is global linear regression on all the attributes. It now reports that it can make only a 19.8% reduction in the error over just guessing the average every time. This is very disappointing, but it is also a correct analysis. As it turns out, the sensor readings in this data are not good indicators of the final wafer quality and a learner that claimed otherwise would be misleading its user.

How long should Blackbox run?

It isn't obvious how long to run Blackbox when looking at a new data file. In general, it will need to run longer when there is more data and more attributes. However, it also depends greatly on the underlying relationships in the data which is usually the one thing you don't know when beginning. The *4of8.mbl* file has some interesting relationships and takes a moderate amount of time to run. The semiconductor data is different. Once we give Blackbox a separate test set to check itself with, it only takes a short amount of time to find global linear regression (not much of a "find"). We would probably run Blackbox for much longer, though, in hopes that it would turn up something better. It's usually best to set Blackbox running for a long time and then watch its results. When it appears to be making no more progress, hit 'q' to stop the searching.

Blackbox caches the results of its searches, so it can be restarted several times to continue a search it was doing. Its important to set the reset check box appropriately when doing this. When it is checked, all previous search results will be wiped out and the search begins from the current metacode. When it is unchecked, it keeps all previous search results and continues from there.

General	Billiards Robot	Plasma Etch
Given a desired target	Given a desired pocket	Given a desired etch rate
Choose an action that, under the current observed state	Choose a cue angle that, under the current observed ball position	Choose a plasma gas mix that, under the current observed chamber state
Is predicted to achieve the target	Is predicted to sink the ball	Is predicted to achieve the etch rate

Figure 27: Examples of control to target problems

Autonomous model selection pros and cons

Parameter tweaking is a tedious operation not always performed well by people, so automating it is an obvious plus. There are several other benefits from it too. Automation makes it possible for non-experts to apply machine learning. Automation also makes it possible to put machine learning in embedded systems where humans don't have the access required to do the tweaking.

There are several things to be careful of with autonomous model selection. It can be computationally expensive, which is an issue if a human expert would have been able to find the model more quickly than a machine doing a comparatively brute-force approach. It is necessary to guard against overfitting when doing lots of aggressive cross validation. Vizier protects itself from this by holding out an additional data set (*CV Police*). Cross validation minimizes the prediction error, but there are scenarios where minimizing the prediction error is not what is needed. For example, when data is sparse, it is desirable to choose broad kernel widths in order to get the most information from the available data, even though doing so may degrade cross validation prediction accuracy.

We believe the pros for autonomous model selection strongly outweigh the cons, and thus have made it central to Vizier's operation. However, Vizier also offers the flexibility and the tools to completely take over the model selection process for those experts that prefer to.

9 Decisions with Locally Weighted Models

In this tutorial we have seen how locally weighted learning works, and how to automatically select good locally weighted models. We have also seen how to make predictions and get confidence intervals from the models, and how to graph the results. As we have stated before, though, making predictions is not what learning is about. *It is about making decisions.* In this section we will demonstrate how LWL and Vizier are used to make decisions. For more information on using LWL for control see [5].

9.1 Choosing Parameters to Achieve a Target

Building a controller is one common use for a learned model. Suppose there is data showing the mapping from current states and control variable settings to resulting outputs from a system, and suppose that data has been used to build a model. What we would like to do then, is to give the learner the current state and the desired result, and have it choose an action that will achieve that result. Fig. 27 shows examples of control to target problems.

There are two kinds of models that can be used to build controllers, called *forward* and *inverse* models. Suppose we have a system with one control variable, and one output variable, and we want to choose a control (or action) to achieve a desired output (behavior). An inverse model is one that maps outputs onto the controls that cause them, while a forward model maps controls onto resulting outputs. Examples of the two kinds of models are shown in fig. 28. You can create these models with Vizier as follows:

```
File -> Open -> control.mbl
```

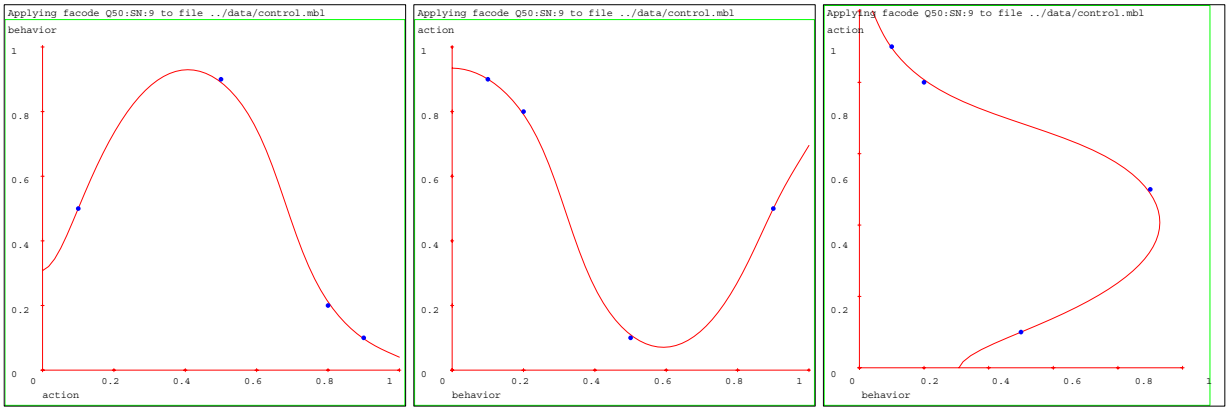


Figure 28: a) Forward model for a controller, b) Inverse model for a controller, c) what a correct inverse model would look like for this data.

```
Edit -> Format -> action  output
                  behavior  input
Edit -> Metacode ->  Q50:9
Model -> Graph -> Graph
Edit -> Format -> action input
                  behavior output
Edit -> Metacode ->  Q50:9
Model -> Graph -> Graph
```

Inverse models

The advantage of an inverse model is that it can be used directly to build a controller. The desired behavior is treated as an input variable in the model, and the action is treated as an output variable. When a new desired behavior is given, the controller just asks the model to predict the action needed. It works just like any other learner prediction. The problem that can arise with inverse models is that the mapping from actions to behaviors is often non-invertible. This happens when more than one action results in the same behavior. Notice that the graph in fig. 28a has exactly that property. When the data from a system with this property is used to construct an inverse model (something that isn't really possible without extra constraints to resolve the ambiguity), the inverse model is often poor, or just plain wrong. If you look carefully at fig. 28b, you'll see that the function fit to the data for the inverse model is incorrect. We can see the problem by requesting the inverse model to find us an action that will produce behavior 0.4.

```
Edit -> Format -> action  output
                  behavior  input
Edit -> Metacode ->  Q50:9
Edit -> Query -> 0.4
Model -> Predict
Model -> Analysis
```

The inverse model suggests action 0.25. This selection can be seen by looking at fig. 28b. To ask Vizier for a prediction only, use the Predict command. The analysis command provides gradients, noise estimates, and confidence intervals as well. We'll come back to the information from the analysis command shortly. Now, use the forward model to predict what behavior will result from this action.

```
Edit -> Format -> action  input
                  behavior  output
Edit -> Metacode ->  Q50:9
Edit -> Query -> 0.25
Model -> Predict
```

The forward model predicts a resulting behavior of 0.82, as can be seen in fig. 28a. This is far from the intended behavior of 0.4. In fact, the problem is even worse than this. It is common for a model to have some inaccuracies. Because of that, it is often a good idea to make a closed loop controller that looks at derivatives and adjusts its controls to hone in on the desired behavior. If we wanted to use the inverse model for this purpose, we could look at the derivatives it gives us. Go back to the results of the analysis command on the inverse model (or look at the slope of the curve in fig. 28b). It says the derivative of the action with respect to the behavior is -1.62. We know that the behavior we got from the forward model was higher than what we wanted and thus we would like to lower it. The negative derivative given to us by the inverse model says that if we want to decrease the behavior value, we should increase the action value. So a closed loop controller might increase the action from 0.25 to 0.3. Now check the forward model to see what it predicts for action 0.3. It predicts 0.88, which is even further from the behavior we wanted! You can run the analysis command on the forward model to see that the predicted derivative between actions and behaviors is positive, which confirms the problem. So we see that not only is the inverse model inaccurate, but attempts to put it in a closed loop controller could make things even worse.

The problem is further illustrated by considering fig. 28c. It shows what the correct inverse function would be if you could exactly swap the axes and mirror the forward function appropriately. That graph was not produced by Vizier since the curve in it is not a function. It has more than one output for some of the inputs.

Forward models

Inverse models are a good way to build controllers if the system can be inverted properly. If inversion is a problem, then using forward models is a much safer method. Forward models come with a caveat though. We are given a desired behavior. However, the behavior is an output in a forward model. We can not just set the query to the desired behavior and ask for a prediction. It is necessary to search through the space of possible actions to find the one that predicts the behavior we want.

Fortunately, Vizier can do exactly that. At this point, we will introduce the Optex command, which can do a variety of model inversion, optimization, and experiment design tasks. It has the flexibility to do many things, but we'll start with a simple example.

```
Optex -> New
         Project name            forward
         Use current data        ON
         Continue
         Minimize expression
         OUTPUT Variable weight  0.0
         OUTPUT Target value     0.4
         OUTPUT Target weight    1.0
         OK
         Choose
```

The default Optex search algorithm is called *PMAX*. It looks for the input that will produce the best output as defined in the *Taskspec*. In this example, "goodness" was defined as minimizing the squared difference between the desired behavior and the actual behavior. Optex suggested action 0.06. You can use Predict to verify that this action does produce a behavior near 0.4. Notice that there is more than one action that will produce the desired behavior. 0.72 will also work. Optex has chosen between the two of these arbitrarily. We can constrain Vizier's search so that it only considers actions above 0.5, and ask it to choose again. Then it selects 0.72.

Optex makes a dialogue box that stays on the screen after Optex commands have been completed. Subsequent Optex commands can then be run from that dialogue box. The dialogue box can be removed by selecting "Quit." It can then be recalled by selecting "Resume" from the Optex menu.

```
Optex -> Edit -> Taskspec -> INPUT Low  0.5
                             OK
         Choose
```

Note: *If you have done other operations that have invoked Vizier's random number generator, Optex may select 0.72 the first time. If that happens, you can see it choose 0.06 by constraining it to choose actions below 0.5.*

9.2 Maximizing or Minimizing a Learned Model

We have seen how to search a forward model to get a target tracking controller, but that is just one of many things Optex can do with a memory based model. Suppose that instead of tracking a target, we'd like to maximize the output of a certain system. This would be the case if the action was a temperature control on a chemical fermentation process, and the behavior was the process yield. To do this, we modify the Taskspec to describe the new problem.

```
Optex -> Edit -> Taskspec -> Maximize expression
                             OUTPUT Variable weight 1.0
                             OUTPUT Target weight   0.0
                             INPUT Low              0.0
                             OK
         Choose
```

Optex suggests action 0.4, and predicts that it will yield behavior 0.93. You can ask Vizier for a prediction, or check fig. 28 to verify the Optex selection.

IESAFE

PMAX seems to be a good choice for the *control.mbl* file. There are other cases where PMAX might be dangerous though.

```
File -> Open -> d5.mbl
Edit -> Metacode -> Q40:9
Graph -> show_confidences ON
         GRAPH
Optex -> New
         Project name            forward
         Use current data        ON
         Continue
         Maximize expression
         OK
         Choose
```

The graph is shown in fig. 29. With a goal of maximizing the output, Optex chooses input 0.50, which has a predicted output of 0.61. There is something troubling about the confidence intervals for that choice of inputs though. The confidence intervals say that we are 95% certain that the output will lie between -0.53 and 1.76. That's not very certain! If the model represents a real production process, we may not be able to afford any disasters such as producing far less than planned.

Fortunately, we can use the confidence intervals from the LWL model to make a much safer selection. Instead of choosing the input whose predicted output is best, it may be safer to choose the input whose lower confidence interval is best. That way, disasters can be avoided. Optex does this with the *IESAFE* algorithm. IESAFE searches the lower (upper) confidence interval when doing maximization (minimization) and chooses the point where it is highest (lowest).

```
Optex -> Edit -> Settings -> opt_type  IESAFE
         Choose
```

Now, Optex chooses input 0.60. The predicted output is slightly lower, 0.59, but the confidence intervals are much tighter, 0.53 to 0.64. Therefore, we can feel much safer about using this input to control a real physical process.

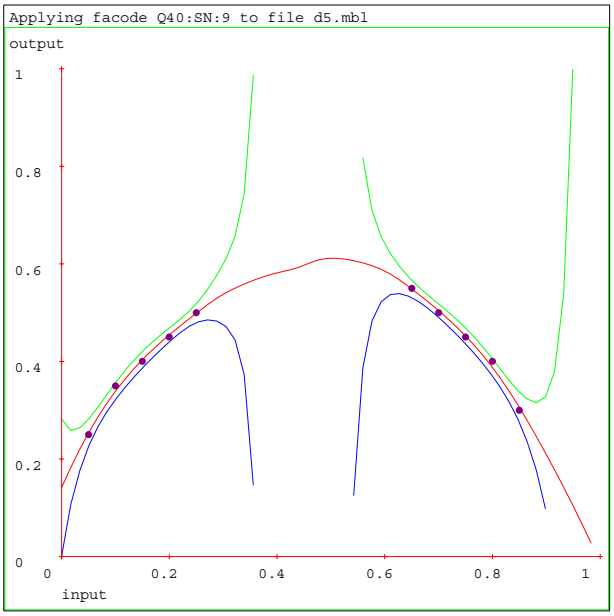


Figure 29: The file d5.mbl with metacode Q40:9

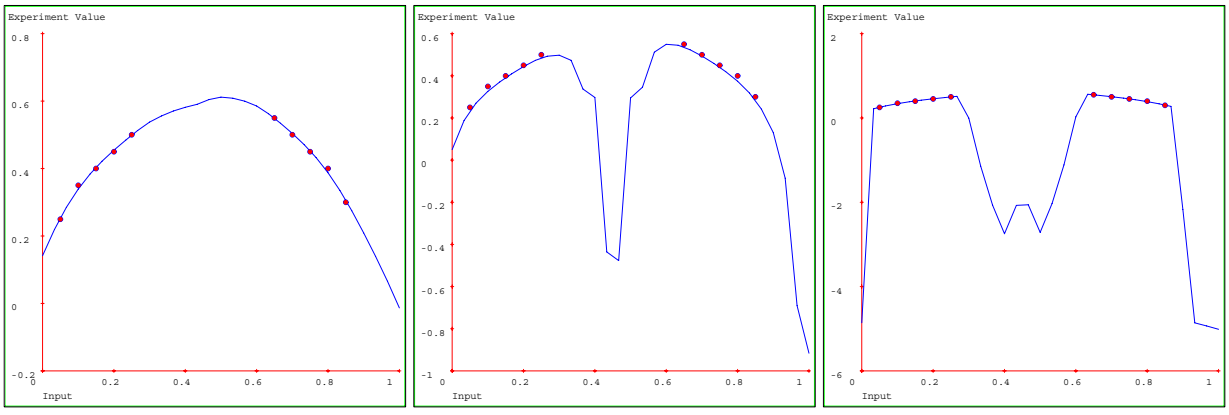


Figure 30: The experiment valuation function for *d5.mbl* using a) PMAX, b) IESAFE, c) EMAX

In order to make its choice, Optex defines an experiment value function that specifies how good each parameter setting is. Then it searches that function to find the setting with the best value. The definition of the value function depends both on the task defined by the user in the Taskspec, and on the Optex algorithm being used. When the data file has only one or two inputs, Optex plots its experiment value function along with its recommendation. Looking at these functions is sometimes helpful in understanding how the different algorithms work. Fig. 30 shows the value functions for PMAX, IESAFE, and EMAX (next section). Notice that the PMAX value function is exactly the same as the predicted output in fig. 29. The IESAFE value function is exactly the same as the lower confidence interval on the output in fig. 29.

EMAX

IESAFE is a good way to make safe choices about parameter settings. Sometimes it is not a disaster to produce less than expected, but there is a certain threshold below which the process must be shut down and restarted. Suppose that the cost of a shut down is 10.0. Then we would like to choose an action that explicitly considers this cost and the probability of incurring it. *EMAX* can be used for this purpose. First, update the Taskspec to indicate the addition to the cost function:

```
Optex -> Edit -> Taskspec -> Extra expression:  if (yield<0.0) then -10 else 0
         Choose
```

We could use PMAX to choose an action while considering this extra cost. However, PMAX will still choose 0.50. That's because the predicted output for temperature 0.50 will not incur the penalty. PMAX does not consider the fact that the confidence intervals are wide and the actual output may incur the penalty. In contrast, EMAX looks over the whole range of possible outputs and considers the cost of each one as well as the probability of getting it. Therefore, EMAX selects temperature 0.60 because it has a high expected yield and a low probability of incurring the penalty of 10.0. The experiment value function shown in fig. 30c shows how the probability of a yield below 0.0 drastically lowers the value of parameters settings where the confidence intervals reach below 0.0.

9.3 Using Locally Weighted Learning to Design Experiments

We have seen how Optex can search a model built from existing data to find desirable parameter settings. Another common problem is determining what experiments to run while collecting data to build a model. This is known as "Experiment Design." One way to do this is something created by statisticians known as *Response Surface Methodology*, or *RSM*[2]. It involves performing several experiments in a particular region of interest, and then analyzing the resulting data to determine whether to take additional experiments in that region, or to move the region to a more promising area. RSM is widely used and fairly successful. Its drawback is that much of it must be done manually, and often by a person with a strong statistics background. For more examples on using LWL for experiment design see [6].

We can use the information from LWL models to build an automatic experiment design system. The Optex dialogue box allows us to enter new data points as experiments are run. Then we can iterate over the sequence: 1) ask Optex for an experiment to run, 2) run the experiment, 3) enter the results of the experiment. This is a departure from the traditional RSM method which suggests a whole batch of experiments, and then evaluates them to suggest a new batch. Because LWL can perform evaluations quickly and choose new experiments online, it is able to respond to interesting data immediately rather than having to finish an entire batch of experiments first.

IEMAX

One approach to experiment design is that we would like to search a particular region and be sure that we have not overlooked any settings in the region that are better than those we found. At the same time, we'd like to concentrate the experiments in promising areas to hone in on the best ones. *IEMAX* operates in a way similar to IESAFE, except that it chooses the best upper confidence interval rather than the best lower confidence interval when doing maximization. This gives exactly the effect we are looking for. Initially, when there is no data, the upper confidence interval is very high everywhere. As data is collected, the

upper confidence interval gets pushed down near the data points. When there is enough data to give gross estimates throughout the region, IEMAX starts concentrating on the most promising areas since their upper confidence intervals will still be higher than the other areas.

To demonstrate this process, we'll show what IEMAX does on the trivial task of minimizing the function $y = (x - 0.7)^2$ without noise. Note that when minimizing, IEMAX will be choosing the point with the lowest lower confidence interval, rather than the highest upper confidence interval.

```
Optex -> New
         Project name      experiment
         Use current data  OFF
         Number of inputs  1
         Number of outputs 1
         Continue
         OK
         Minimize expression
         OK
Edit -> Metacode -> L40:9
Optex -> Edit -> Settings -> opt_type  IEMAX
         Choose (chooses 0.5)
         Output  0.04
         Observe
         Choose (chooses 1.0)
         Output  0.09
         Observe
(continue repeating these three steps)
```

You may continue asking for new experiments and manually computing what the output will be ($(x - 0.7)^2$). Fig. 31 shows what the predictions and confidence intervals look like from the second data point to the seventh. Notice that in each graph, the data point is taken at the point where the lower confidence interval was lowest in the previous graph. If you continue asking for more experiment choices after the seventh point, Optex will continue honing in on the best setting in the region of 0.7.

Using the other Optex methods for experiment design

The other Optex algorithms, PMAX, IESAFE, EMAX, can all be used for experiment design too. Generally they can't be trusted as well to explore an entire space. They sometimes settle on a good point and don't bother searching other areas. However, this can be a good feature if your search space is high dimensional and searching all of it is impractical. Also, IESAFE and EMAX can be useful if you already have some process data and prefer not to run any wild experiments with unpredictable, and likely poor, results.

Noise and dimensionality

The Optex examples done here are noiseless. In fact, this is a misapplication of Optex. It is built on locally weighted learning methods that are made to handle noisy data, and it is most useful with noisy data.

The Optex examples here are one dimensional. These examples were chosen specifically because it is easiest to understand the methods on one dimensional problems. In particular, it is difficult to see confidence intervals for higher dimensional models. Optex works well in higher dimensional problems. It is probably more useful in those problems since they're harder to visualize and optimize by hand. In general, Optex will take longer to make its choices as the dimensionality and the number of data points goes up.

9.4 Programming with the Vizier library

In this tutorial we have seen how locally weighted learning and Vizier are used to help make decisions. All of the examples here involved a user operating the software. This is a great way to explore new data sets and learn about LWL. There are many applications, though, where it is necessary to incorporate the functionality

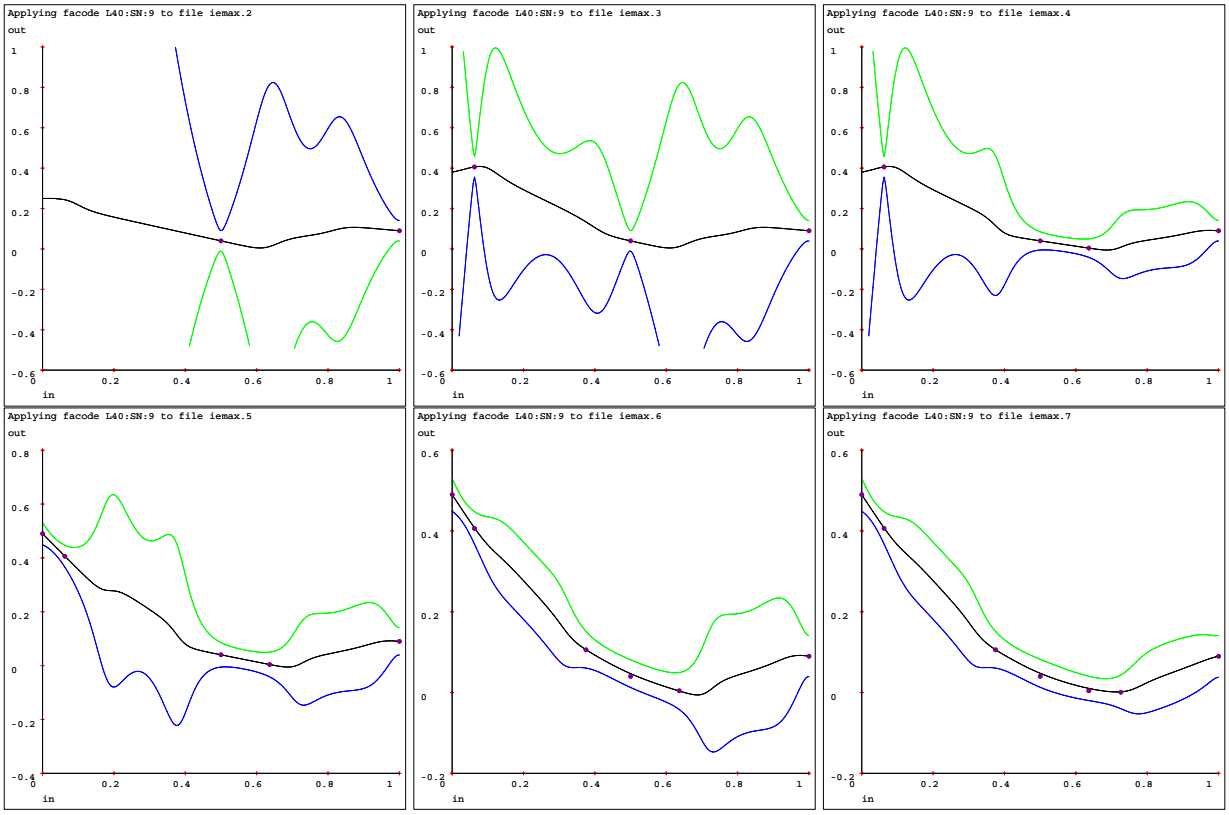


Figure 31: Experiment selections using IEMAX on a minimization problem

of Vizier into other software. A controller running online in a factory is one example of an application where Vizier software is already being used. For this purpose the Vizier software package includes a library in C. Use of the library is documented in the header file for it, *interface.h*.

References

[1] C. Atkeson, S. Schaal, and A. Moore. Locally weighted learning. *AI Review*, 1995.

[2] G. Box and N. Draper. *Empirical Model Building and Response Surfaces*. Wiley, 1987.

[3] M. DeGroot. *Optimal Statistical Decisions*. McGraw-Hill, 1970.

[4] K. Deng and A. Moore. Multiresolution instance-based learning. In *International Joint Conference on Artificial Intelligence*, 1995.

[5] A. Moore, C. Atkeson, and S. Schaal. Locally weighted learning for control. *AI Review*, 1995.

[6] A. Moore and J. Schneider. Memory based stochastic optimization. In *Advances in Neural Information Processing Systems (NIPS-8)*, 1995.