Introduction to Neural Networks
U. Minn. Psy 5038

Bias/variance

Initialize

■ **Read in Add-in packages:**

```
Off[General::"spell1"];
<< "ErrorBarPlots`";
<< "MultivariateStatistics`";
```

Make sure the SVM package is downloaded in in the default directory

```
<< MathSVMv7`
```

SVMs and Kernel methods

As in Lecture 27 (see discriminant functions), we assume a simple perceptron TLU to classify vector data x into one of two classes depending on the sign of g(x):

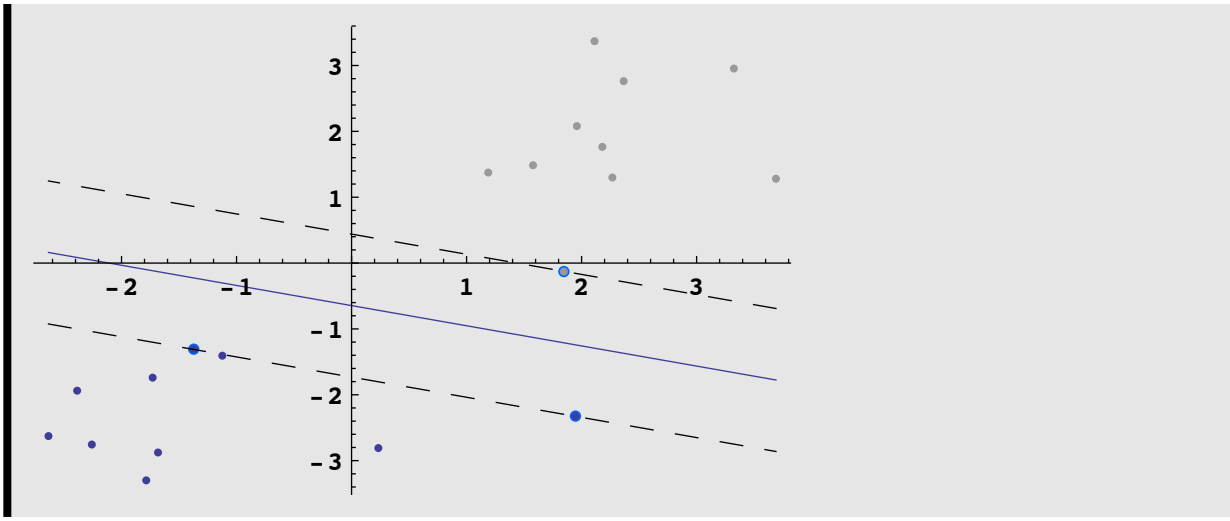
$$\text{decision}(x) = \text{sign}(w.x + b).$$

Given g〔**x**〕= **w.x** + b, recall that g(x)/||w|| is the distance of a data point x from a plane defined by g(x) = 0. In support vector machines, the goal is to find the separating plane (i.e. find w and b) that is as far as possible from any of the data points. The intuition is that this will minimize the probability of making wrong classifications when given new data at some future point. Formally, we want to solve"

$$\max_{(w,b)} \left(\min_i d\left(\Pi_{w,b}, x_i\right)\right), \tag{1}$$

where d$\left(\Pi_{w,b}, x_i\right)$ = g〔**x**〕/ || **w** ||, i.e.

$$d\left(\Pi_{w,b}, x_i\right) = g(x)/\|w\| = |w.x_i + b| / \|w\|$$



Two-class data (black and grey dots), their optimal separating hyperplane (continuous line), and support vectors (circled in blue). This is an example output of the `SVMPlot` function in *MathSVM*. The width of the "corridor" defined by the two dotted lines connecting the support vectors is the margin of the optimal separating hyperplane. (From Nilsson et al., 2006)

■ The Primal Problem

It can be shown that the optimal separating hyperplane solving (1) can be found as the solution to the equivalent optimization problem

$$\begin{aligned} &\min_{w,b} \frac{1}{2} \| w \|^2 \\ &\text{subject to } y_i\left(w^T x_i + b\right) \geq 1, \end{aligned} \tag{2}$$

Typically, equality will hold for a relatively small number of the data vectors. These data are termed *support vectors*. The solution (w, b) depends only on these specific points, and in effect contain all the information for the decision rule. The "dual problem".

A Simple linear SVM Example (from Nilsson et al. 2006)

Here's a demo of a simple SVM problem. It uses the add on package *MathSVMv7 written by Nilsson et al*.

```
len = 20;
X = Join[
    RandomReal[NormalDistribution[-2, 1], {len / 2, 2}],
    RandomReal[NormalDistribution[2, 1], {len / 2, 2}]];
y = Join[Table[1, {len / 2}], Table[-1, {len / 2}]];
```

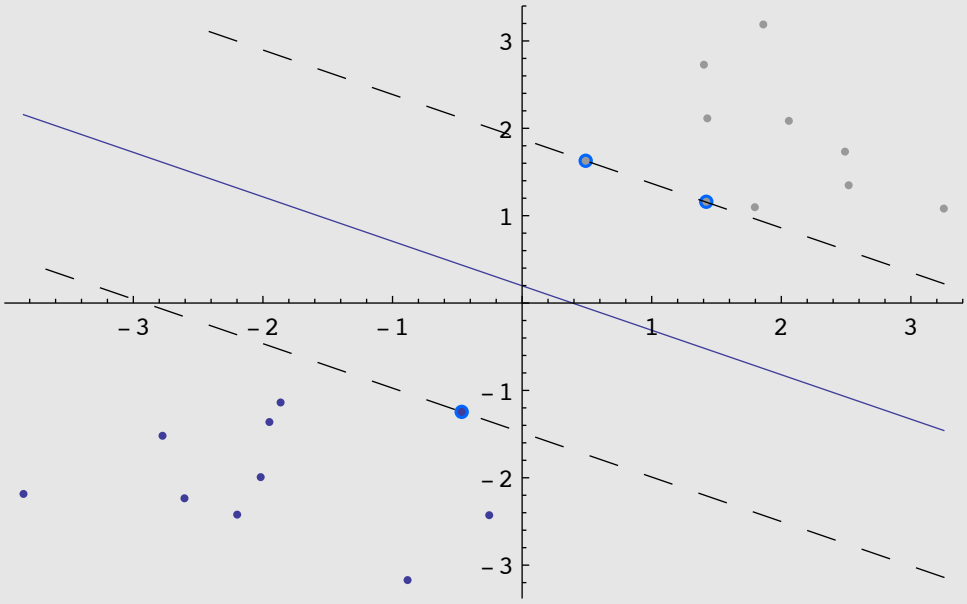
We use the simple SVM formulation provided in *MathSVM* by the `SeparableSVM` function.

```
τ = 0.01;
α = SeparableSVM[X, y, τ]
```

```
{0, 0.222806, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0.126342, 0.0964638, 0, 0, 0, 0, 0, 0, 0}
```

In the output figure below, the solid line marks the optimal hyperplane, and dotted lines mark the width of the corridor that joins support vectors (highlighted in blue).

```
SVMPlot[α, X, y]
```



A Nonlinear Example: Using Kernels (from Nilsson et al. 2006)

What if the data are not linearly separable? The essential idea is to map the data (through some non-linear mapping, e.g. polynomial) to a higher-dimensional "feature" space to find the optimal hyperplane separating the data. The dot product gets replaced by a non-linear kernel function. For example, the polynomial kernel is given by:

$$k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j})^d$$

If d = 1, we have the standard dot product, but for d = 2, 3, etc.. we have polynomial functions of the elements of the vectors x. See Nilsson et al, and paper by Jäkel (2009) for more information on kernels.

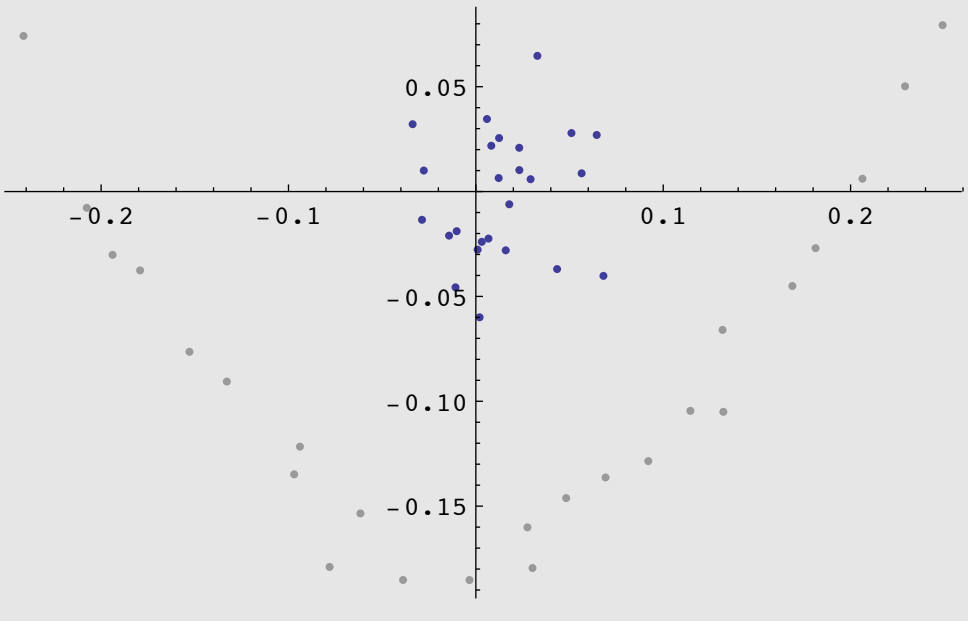
Here is a demo for an application for nonlinear classification . We'll use second-degree kernel:

```
PolynomialKernel[x, y, 2]
```

```
(1 + x.{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1})^2
```

Some synthetic data which is not linearly separable.

```
len = 50;
X = Join[
    RandomReal[NormalDistribution[0, 0.03], {len / 2, 2}],
    Table[
     {RandomReal[NormalDistribution[i / len - 1 / 4, 0.01]],
      Random[NormalDistribution[(2 i / len - 1 / 2)^2 - 1 / 6, 0.01]]},
     {i, len / 2}]];
y = Join[Table[1, {len / 2}], Table[-1, {len / 2}]];
SVMDataPlot[X, y, PlotRange → All]
```



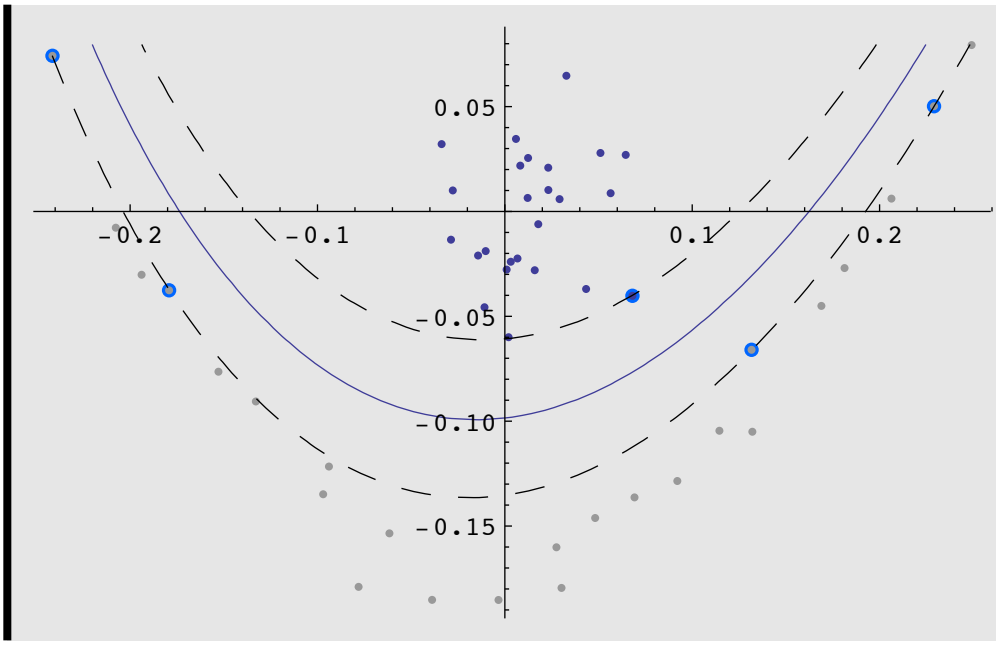
We use the `KernelFunction` to specify the kernel type and run **SeparableSVM[]**.

```
τ = 0.01;
pk = PolynomialKernel[#1, #2, 2] &;
α = SeparableSVM[X, y, τ, KernelFunction → pk]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4397.44, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 238.421, 0, 0, 736.733, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3022.5, 0, 0, 0, 399.786, 0}
```

When visualizing the results, SVMPlot can use the kernel functions to draw any nonlinear decision curves.

```
SVMPlot[α, X, y, KernelFunction → pk]
```




```
Clear[len, X, y, α, pk]
```

■ More information

The wiki entry for SVMs has a fairly good introduction (As of 12/14/2009,).

To go to the source see Vapnik (1995)

http : // svm.first.gmd.de/

Demo links:

http://svm.dcs.rhbnc.ac.uk/pagesnew/GPat.shtml

And for applications of kernel methods more generally, to cognitive and neuroscience see reviews by Jäkel et al. (2006; 2009). The links to the pdfs are in the course syllabus.

Statistical learning, model selection & the bias/variance dilemma

In Lecture 26, we summarized optimal rules for minimizing risk in Bayesian statistical decision theory, assuming that we know the distributions of the generative model.

But what if we don't? Consider the regression problem, fitting data that may be a complex function of the input.

The problem in general is how to choose the function that both remembers the relationship between **x** and **y**, and generalizes with new values of **x**. At first one might think that it should be as general as possible to allow all kinds of maps.

For example, if one is fitting a curve, you might wish to use a very high-order polynomial, or a back-prop network with lots of hidden units. There is a drawback, however, to the flexibility afforded by extra degrees of freedom in fitting the data. We can get drastically different fits for different sets of data that are randomly drawn from the same underlying process. The fact that we get different fit parameters (e.g. slope of a regression line) each time means that although we may exactly fit the data each time, we introduce variation between the average fit (over all data sets) and the fits over the ensemble of data sets. We could get around this problem with a huge amount of data, but the problem is that the amount of required data can grow exponentially with the order of the fit--an example of the so-called "curse of dimensionality".

On the other hand, if the function is restrictive, (e.g. straight lines through the origin), then we will get similar fits for different data sets, because all we have to adjust is one parameter--the slope. The problem here, is that the fit is only good if the underlying process is in fact a straight line through the origin. If it isn't a straight line for instance, there will be a fixed error or **bias** that will never go away, no matter how much data we collect. Statisticians refer to the trade-off between simple but biased fits, and complex but data-dependent variation, as the *bias/variance* dilemma.

To sum up, lots of parameter flexibility (or lots of hidden units) has the benefit of fitting anything, but at the cost of sensitivity to variability in the data set--there is *variance* introduced by the fits found over multiple training sets (e.g. of a small fixed size).

A fit with very few parameters is not as sensitive to the inevitable variability in the training set, but can give large constant errors or *bias* if the data do not match the underyling model.

There is no general way of getting around this problem, and neural networks are no exception. We generalized linear regression to non-linear fits using error back-propagation. Because back-propagation models can have lots of hidden layers with many units and weights, they form a class of very flexible approximators and can fit almost any function. But these models can show high variability in their fits from one data set to the next, even when the data comes from the same underlying process. Lots of hidden units can mean low bias, but at a high cost in variance.

Demonstration of bias/variance for regression

Suppose we have an unknown, underlying generative model given by: p(y|x)p(x). From this we obtain a set of samples $\{x_i, y_i\}$, i=1...N. We posit some estimator to fit the data: y_i~f(x_i). In general, there will be some cost assigned to errors in f's ability to predict the y's. E.g. the expected value of the squared difference between the fits and the true expected value of y, call it $\hat{y}$. We can write this cost as the sum of two terms:

$$E[(f - \hat{y})^2] = (E[f] - \hat{y})^2 + E[(f - E[f])^2]$$

The first term on the right is the bias (squared), and the second term the variance. The bias is the "constant error" which tells us how far off we'll be--could be non-zero even with an unlimited supply of data. The second term, the "variance", tells us how much variation we have in the ensemble of fits f about the average of all the fit, E[f].

Let's represent the expectation of f, E[f], by $\tilde{f}$:

$$E[(f-\hat{y})^2] = (\tilde{f}-\hat{y})^2 + E[(f-\tilde{f})^2]$$

A more general formulation of the bias/variance trade-off, for a risk function R() is:

$$\int R(\alpha_D; x) P(\alpha_D) d\alpha = \int (y-\hat{y}(x))^2 P(y|x) dy + \int (\tilde{f}(x)-f(x,\alpha))^2 P(\alpha_D) d\alpha_D + (\hat{y}(x)-\tilde{f}(x))^2$$

Use the following link for the notes:

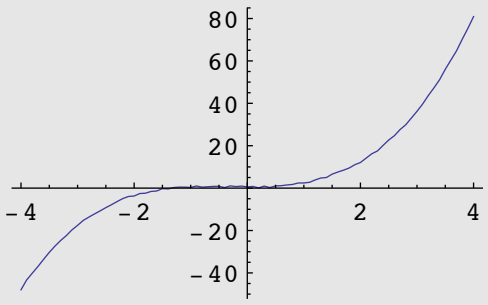
http://gandalf.psych.umn.edu/~kersten/kersten-lab/courses/Psy5038WF2003/MathematicaNotebooks/Lect_27_BiasVariance/biasvarianceNotes.pdf

Let's demonstrate the effects of the bias/variance trade-off by estimating the values of $\tilde{f}$ and $\hat{y}$, given a generative model.

■ *Mathematica*'s regression package

Go to Help, and find the Linear Regression package. Look up **LinearModelFit[]**. We are going to use Regress as our learning model. We could have used our errro-back prop network, or other learning algorithms that produce a set of fit parameters. The principles would be the same.

```
ff[x_, α_] := α.{1, x, x^2, x^3} + RandomReal[];
α = {0, 0, 1, 1};
xd = Table[{x, ff[x, α]}, {x, -4, 4, 0.1}];
ListPlot[xd, AxesOrigin → {0, 0}, Joined → True, ImageSize → Small]
lm = LinearModelFit[xd, {1, x, x^2, x^3}, x];
lm[{"ParameterTable", "RSquared"}]
```



	Estimate	Standard Error	t Statistic	P-Value
1	0.561972	0.0453611	12.3888	5.14194×10^{-20}
x	−0.0322744	0.0323481	−0.997723	0.321539
x^2	0.990781	0.00618542	160.18	6.0653×10^{-99}
x^3	1.00634	0.00301387	333.902	1.80769×10^{-123}

, 0.999893

Learning from one data set

- **True model**

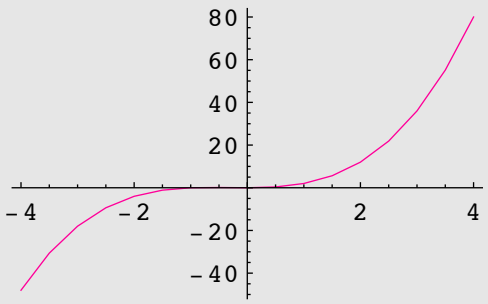
```
ff[x_, α_] := α.{1, x, x^2, x^3};
```

- **Generative data process: true plus some noise**

```
noise = 15;
ffn[x_, α_] := ff[x, α] + 1.5 * RandomReal[{-noise, noise}];
```

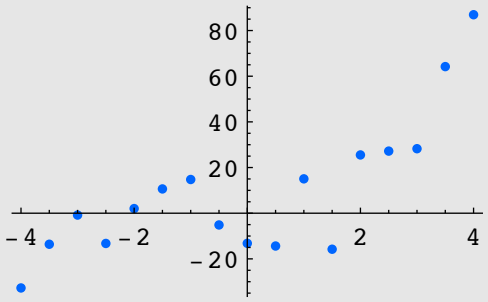
- **Choose domain and true model parameters. Calculate a set of samples from true model ffp, evaluated at xp.**

```
xp = Table[x, {x, -4, 4, 0.5}];
α = {0, 0, 1, 1};
ffp = (ff[#1, α] &) /@ xp;
gffp = ListPlot[Transpose[{xp, ffp}],
  PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True, ImageSize → Small]
```



■ **Run one experiment to collect y values for data process:**

```
y = (ffn[#1, α] &) /@ xp;
gy = ListPlot[Transpose[{xp, y}], PlotStyle → {PointSize[0.02], Hue[0.6]},
  ImageSize → Small]
```

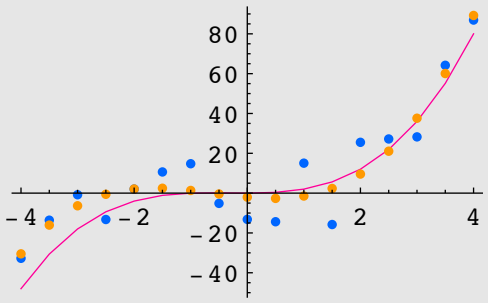


■ **Estimate model parameters using polynomial regression. Return model parameters, and predicted responses, fsquiggle**

```
xd = Table[{x, ff[x, α]}, {x, -4, 4, 0.1}];
```

```
αD = LinearModelFit[Transpose[{xp, y}], {1, x, x^2, x^3}, x];
fsquiggle = Table[{x, αD[x]}, {x, -4, 4, 0.5}];
gfsquiggle = ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]},
   ImageSize → Small];
```

```
Show[gffp, gy, gfsquiggle]
```



Repeat the above, and notice how the model parameters and the fit changes. Try changing the basis functions used for fitting.

Comparing learning from multiple data sets

We'd like some idea of how learning generalizes depending on model complexity

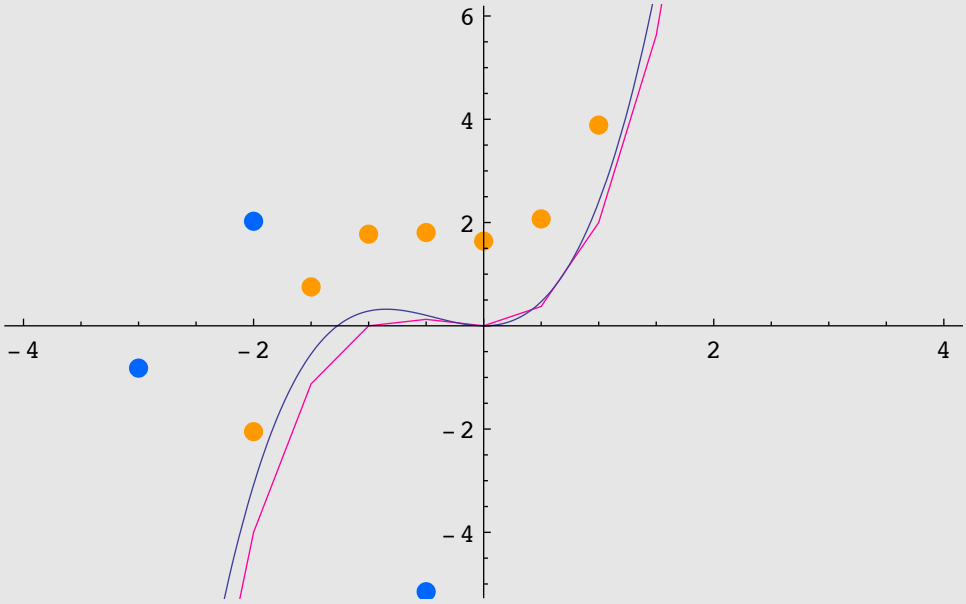
■ The "right" model

First, assume by some incredibly lucky guess, we've chosen the right model {x^2, x^3}, and want to find the parameters.

```
y = ffn[#1, α] & /@ xp;
```

```
αD2 = LinearModelFit[Transpose[{xp, y}], {x^2, x^3}, x];
fsquiggle = Table[{x, αD2[x]}, {x, -4, 4, 0.5}];
gfsquiggle = ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]},
   ImageSize → Small];
```

```
gffp = ListPlot[Transpose[{xp, ffp}],
    PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True];
gsmooth = Plot[Fit[Transpose[{xp, y}], {x^2, x^3}, x] /. x → x2,
    {x2, -4, 4}];
Show[gffp, gy, gfsquiggle, gsmooth]
```



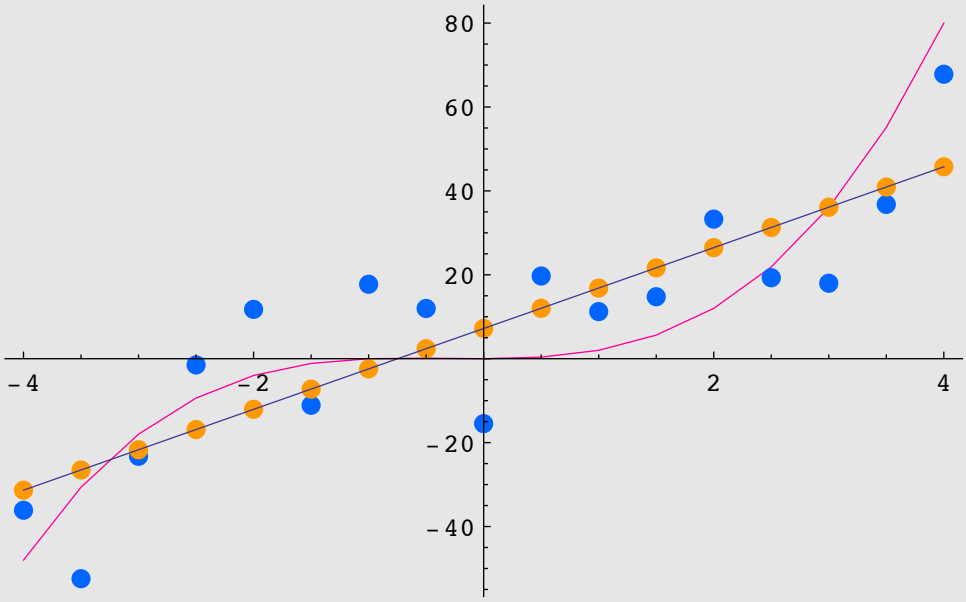
■ A simple, but wrong model

But now suppose that we are trying to fit the data with an inappropriate model. In particular, suppose that it is weak, say a linear model:

```
y = ffn[#1, α] & /@ xp;

αD3 = LinearModelFit[Transpose[{xp, y}], {1, x}, x];
fsquiggle = Table[{x, αD3[x]}, {x, -4, 4, 0.5}];
gfsquiggle = ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]},
    ImageSize → Small];
```

```
gffp = ListPlot[Transpose[{xp, ffp}],
    PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True];
gy = ListPlot[Transpose[{xp, y}],
    PlotStyle → {PointSize[0.02], Hue[0.6]}];
gsmooth = Plot[Fit[Transpose[{xp, y}], {1, x}, x] /. x → x2, {x2, -4, 4}];
Show[gffp, gy, gfsquiggle, gsmooth]
```



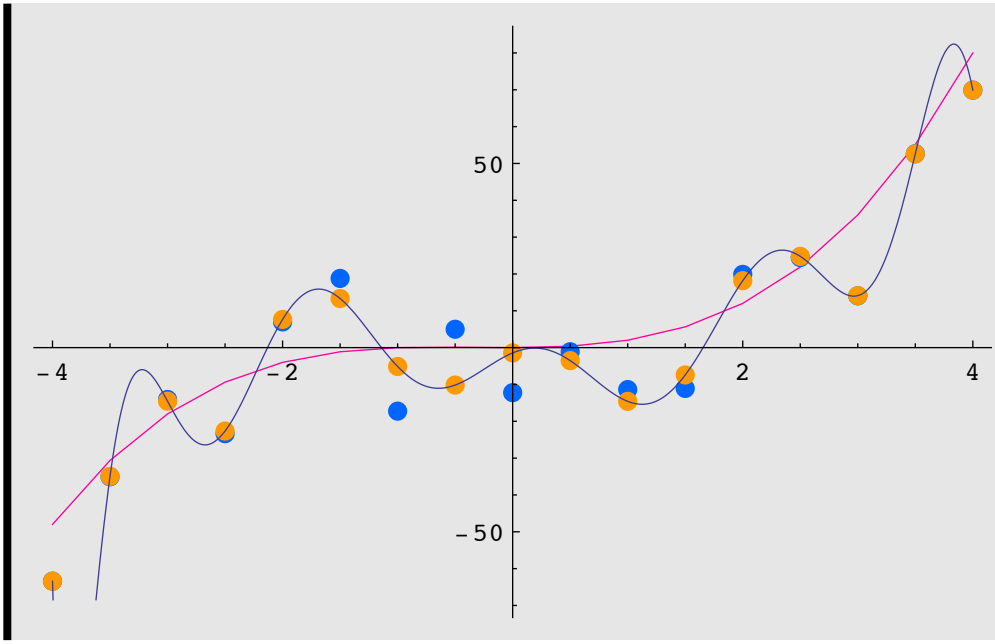
The bias is the squared difference between the average y values (blue) and the model fits (orange). The variance is the squared difference between the predicted responses (blue) and the true (red line). So we can see that the bias is high. The variance (represented by the square root of the variance, black points in the graph) is not zero. But how does it compare with a model with lots of parameters?

■ A complex model, with too many parameters

Now let's try over-fitting. (Analogous to having lots of hidden units and/or layers in a non-linear feedforward network).

```
y = ffn[#1, α] & /@ xp;
αD4 = LinearModelFit[Transpose[{xp, y}],
    {1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^10, x^11, x^12}, x];
fsquiggle = Table[{x, αD4[x]}, {x, -4, 4, 0.5}];
gfsquiggle = ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]},
    ImageSize → Small];
```

```
gffp = ListPlot[Transpose[{xp, ffp}],
    PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True];
gsmooth =
  Plot[
   Fit[Transpose[{xp, y}], {1, x, x^2, x^3, x^4, x^5, x^6, x^7,
      x^8, x^9, x^10, x^11, x^12}, x] /. x → x2, {x2, -4, 4}];
gy = ListPlot[Transpose[{xp, y}],
    PlotStyle → {PointSize[0.02], Hue[0.6]}];
Show[gffp, gy, gfsquiggle, gsmooth]
```



Note how the bias (discrepancy between the orange and blue) is lower than with too few parameters. But we have higher variance than with the "right model" family.

Model selection

Bayesian model selection

MacKay (1992)

Cross-validation

References

Duda, R. O., & Hart, P. E. (1973). Pattern classification and scene analysis . New York.: John Wiley & Sons.

Duda, R. O., Hart, P. E., & Stork, D. G. (2001). *Pattern classification* (2nd ed.). New York: Wiley.

(Amazon.com)

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. Neural Computation, 4(1), 1-58.

Jakel, F., Scholkopf, B., & Wichmann, F. A. (2009). Does cognitive science need kernels? Trends Cogn Sci, 13(9), 381-388. http://linkinghub.elsevier.com/retrieve/pii/S1364661309001430

Kersten, D., & Yuille, A. (2003). Bayesian models of object perception. *Current Opinion in Neurobiology, 13*(2), 1-9.

Kersten, D., Mamassian, P., & Yuille, A. (2004). Object perception as Bayesian Inference. *Annual Review of Psychology, 55*.

MacKay, D. J. C. (1992). Bayesian interpolation. *Neural Computation, 4*(3), 415-447.

Roland Nilsson, Johan Björkegren, Jesper Tegnér (2006) A Flexible Implementation for Support Vector Machines, The *Mathematica* Journal. (journal link) (pdf)

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge, UK: Cambridge University Press.

Vapnik, V. N. (1995). *The nature of statistical learning*. New York: Springer-Verlag.

http://neuron.eng.wayne.edu/software.html