# General Issues

**In this lecture. . .**

- Common machine learning jargon
- A quick reminder of some important mathematical tools we'll be needing
- You will have seen most of these tools in classical mathematics and statistics

## Introduction

There are some issues that are common to many branches of machine learning. In this lecture I put these all together for some brief discussion.

## Jargon And Notation

Every technical subject has its own jargon, and that includes machine learning. Important jargon to get us started is as follows.

- **Data (points, sets):** To train our machine-learning algorithm we will need lots of data to input. Each data point could also be called an 'example' or a 'sample.' In supervised learning each input (vector) has an associated output (vector), you can think of these as the independent and associated dependent variables.

- **Feature (vector):** A feature is a characteristic of the data. If you are from statistics then think of it as an explanatory variable.

  It might be numerical (height of tree 6.3m) or it might be descriptive (eye colour blue). Often if it is descriptive then you will still have to give it a numerical label in order to do mathematical manipulations.

  For example, you might want to group cats into different breeds, and the features might be length of fur (numerical) or whether or not it has a tail (descriptive but easily represented by value zero or one). (No tail? If you are wondering, the Manx cat is tailless.) Each data point will often be represented by a feature vector, each entry representing a feature.

- **Classification:** We will often be dividing our data into different classes or categories. In supervised learning this is done *a priori*. (Is the washing machine reliable or not?) In unsupervised learning it happens as part of the learning process.

- **Regression:** You want to output a numerical forecast? Then you need to do a regression. This contrasts with the above classification problems where the result of a algorithm is a class not a number.

## Notation

- I have tended to use the superscript $.^{(n)}$ to denote the $n^{\text{th}}$ out of a total of $N$ data points. That means I would have, say, $N$ flowers to categorize, and a general flower would have the label $n$. The class, output or value of the $n^{\text{th}}$ individual will be $y^{(n)}$.

- I use $m$ to denote the $m^{\text{th}}$ feature out of a total of $M$ features. A feature might be salary, or number of words in an email.

- When I have classes or clusters I will use $K$ to be number of classes, clusters, etc. and $k$ for one of the classes or clusters.

- Vectors are bold face, and are column vectors. The vector of features of the $n^{\text{th}}$ individual will be $\mathbf{x}^{(n)}$.

## Scaling

The first part of many algorithms that you'll be seeing is a transformation and a scaling of the data.

- We are often going to be measuring distances between data points so we must often make sure that the scales for each feature are roughly the same

Suppose we were characterising people according to their salary and number of pets. If we didn't scale then the salary numbers, being in the thousands, would outweigh the number of pets. Then the number of pets a person had would become useless, which would be silly if one was trying to predict expenditure on dog food.

This is easy to do.

With the original unscaled data take one of the features, say the top entry in the feature vector, measure the mean and standard deviation (or minimum and maximum) across all of the data. Then translate and rescale the first entries in all of the feature vectors to have a mean of zero and a standard deviation of one (or a minimum of zero and a maximum of one). Then do the same for the second entry, and so on.

And don't forget that when you have a new sample for prediction you must scale all its features first, using the in-sample scaling.

## Measuring Distances

We'll often be working with vectors and we will want to measure the distances between them. The shorter the distance between two vectors the closer in character are the two samples they represent. There are many ways to measure distance, and some of them are useful in machine learning.

**Euclidean distance:** This is the classic measurement, using Pythagoras, just square the differences between vector entries, sum and square root. This would be the default distance measure, 'As the crow flies.' It's the $L^2$ norm.

**Manhattan distance:** The Manhattan distance is the sum of the absolute values of the differences between entries in the vectors. The name derives from the distance one must travel along the roads in a city laid out in a grid pattern. This measure of distance can be preferable when dealing with data in high dimensions. This is the $L^1$ norm.

**Chebyshev distance:** Take the absolute differences between all the entries in the two vectors and the maximum of these numbers is the Chebyshev distance. This can be the preferred distance measure when you have many dimensions and most of them just aren't important in classification. This is the $L^{\infty}$ norm.

**Cosine similarity:** In some circumstances two vectors might be similar if they are pointing in the same direction even if they are of different lengths. A measure of distance for this situation is the cosine of the angle between the two vectors. Just take the dot product of the two vectors and divide by the two lengths. This measure is often used in Natural Language Processing, e.g. with Word2vec, mentioned briefly later.

## Curse Of Dimensionality

In any problem involving data in high dimensions in which one is measuring distances one has to be careful not to be struck by the curse of dimensionality. Suppose your data has $M$ features and you have $N$ data points.

Having a large number of data points is good. But what about number of features?

You might think that the more features you have the better because it means that you can be more precise in classification of data. Unfortunately it doesn't quite work like that.

Think how those $N$ data points might be distributed in $M$ dimensions.

Suppose that the numerical data for each feature is either zero or one, to keep things simple. There will therefore be $2^M$ possible combinations. If $N$ is less than this value then you run the risk of having every data point being on a different corner of the $M$-dimensional hypercube. The consequence of this is that the distances on which you base your analysis become meaningless.

## Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE) is a common method for estimating parameters in a statistical/probabilistic model. In words, you simply find the parameter (or parameters) that maximizes the likelihood of observing what actually happened. Let's see this in a few classical examples.

## Example: Taxi numbers

You arrive at the train station in a city you've never been to before. You go to the taxi rank so as to get to your final destination. There is one taxi, you take it. While discussing European politics with the taxi driver you notice that the cab's number is 1234. How many taxis are in that city?

To answer this we need some assumptions. Taxi numbers are positive integers, starting at 1, no gaps and no repeats. We'll need to assume that we are equally likely to get into any cab. And then we introduce the parameter $N$ as the number of taxis. What is the MLE for $N$?

Well, what is the probability of getting into taxi number 1234 when there are $N$ taxis? It is

$$\frac{1}{N} \text{ for } N \geq 1234 \text{ and zero otherwise.}$$

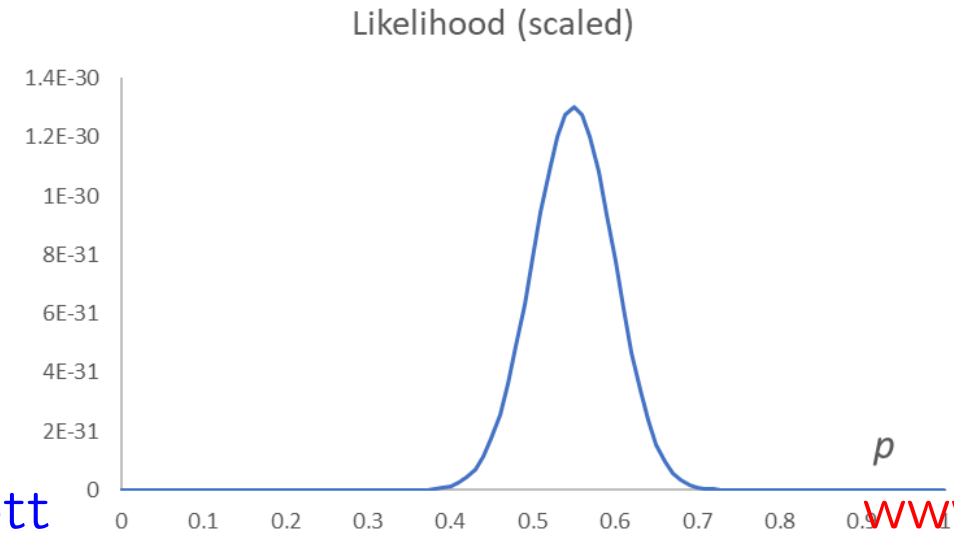
What value of $N$ maximizes this expression? Obviously it is $N = 1234$.

## Example: Coin tossing

Suppose you toss a coin $n$ times and get $h$ heads. Can we find the probability $p$ of tossing a head? The probability of getting $h$ heads out of $n$ tosses is

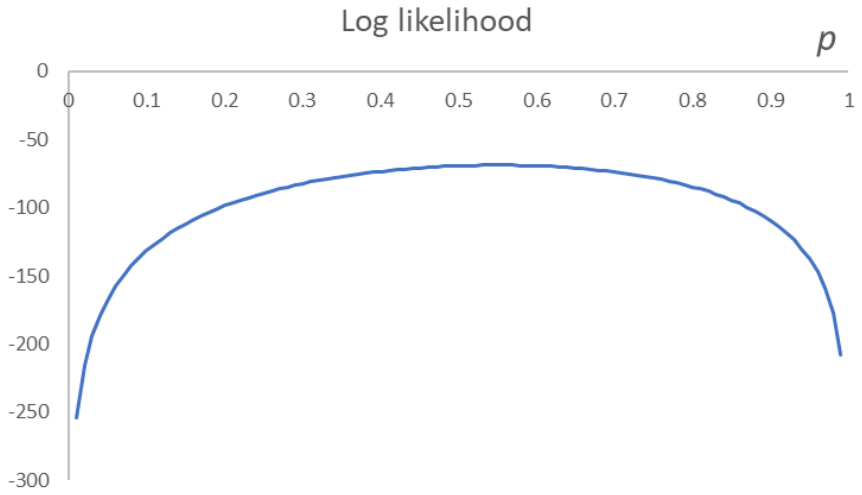
$$\frac{n!}{h!(n-h)!}p^h(1-p)^{n-h}$$

Applying MLE is the same as maximizing this expression with respect to $p$. For $n = 100$ and $h = 55$ the likelihood function is:



Often with MLE when multiplying probabilities, as here, you will take the logarithm of the likelihood and maximize that. This doesn't change the maximizing value but it does stop you from having to multiply many small numbers, which is going to be problematic with finite precision.



Since the first part of this expression is independent of $p$ we maximize

$$h \ln p + (n - h) \ln(1 - p) \tag{1}$$

with respect to $p$.

This just means differentiating with respect to $p$ and setting the derivative equal to zero. This results in

$$p = \frac{h}{n},$$

which seems eminently reasonable.

**Example: A continuous distribution and more than one parameter**

For the final example let's say we have a hat full of random numbers drawn from a normal distribution but with unknown mean and standard deviation, that's two parameters. The probability of drawing a number $x$ from this hat is

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right),$$

where $\mu$ is the mean and $\sigma$ the standard deviation which are both to be estimated. The log likelihood is then the logarithm of this, i.e.

$$\ln(p(x)) = -\frac{1}{2}\ln(2\pi) - \ln\sigma - \frac{1}{2\sigma^2}(x-\mu)^2.$$

If draws are independent then after $n$ draws, $x_i$, the likelihood is

$$p(x_1)p(x_2)\dots p(x_n) = \prod_{i=1}^{n} p(x_i).$$

And so the log-likelihood function is

$$\ln\left(\prod_{i=1}^{n} p(x_i)\right) = \sum_{i=1}^{n} \ln(p(x_i))$$

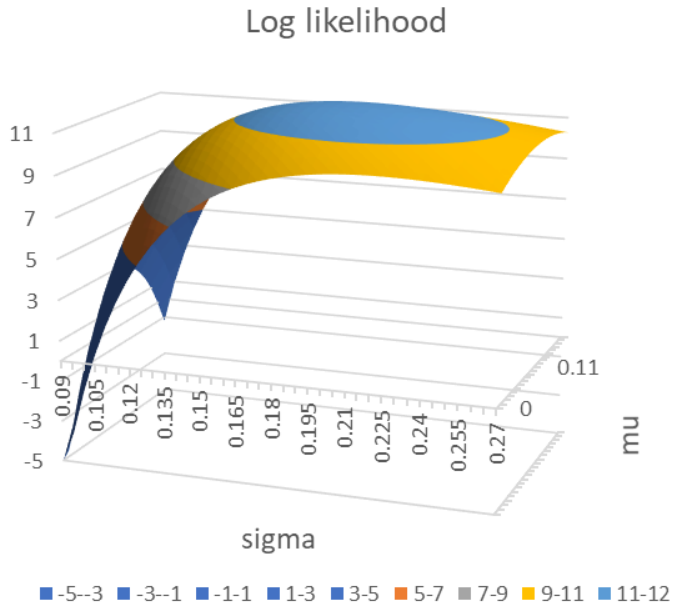
This gives us the nice log likelihood of

$$-n\ln\sigma - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2. \qquad (2)$$

(I've dropped the constant, it doesn't affect the location of the maximum.)

An example is plotted below. For this plot I generated ten $x$s from a normal distribution with mean of 0.1 and standard deviation of 0.2. These ten numbers go into the summation in expression (2), and I've plotted that expression against $\mu$ and $\sigma$. The maximum should be around $\mu = 0.1$ and $\sigma = 0.2$. (It will be at *exactly* whatever is the actual mean and standard deviation of the ten numbers I generated.)



To find the maximum likelihood estimate for $\mu$ you just differentiate with respect to $\mu$ and set this to zero. This gives

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

I.e. $\mu$ is the simple average. And differentiating with respect to $\sigma$ gives

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2}. \tag{3}$$

And this again looks as expected.

## Confusion Matrix

A confusion matrix is a simple way of understanding how well an algorithm is doing at classifying data. It is really just the idea of false positives and false negatives.

Let's look at a simple example.

An algorithm looks at pictures of fruit in order to classify them as apples, pears, oranges, etc. Most it gets right, some it gets wrong. Can we quantify this?

There are 100 items of fruit. Let's focus on how well our algorithm identifies apples.

Sixteen of the 100 are apples. The rest are a mix of pears, oranges, bananas etc. Our algorithm labels each item as whatever fruit it thinks it is. It thinks there are 20 apples. Unfortunately the algorithm had both identified as apples some that weren't and some that were apples it missed. Only 11 of the apples did it correctly identify. So it had some false positives and some false negatives.

| | | ***ACTUAL CLASS*** | |
|---|---|---|---|
| | | Apple | Non apple |
| ***PREDICTED CLASS*** | Apple | 11<br>TP | 9<br>FP |
| | Non apple | 5<br>FN | 75<br>TN |

In order to interpret these numbers they are often converted into several rates:

- Accuracy rate: $\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{Total}}$ where $\mathrm{Total} = \mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}$. This measures the fraction of times the classifier is correct

- Error rate: $1 - \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{Total}}$.

- True positive rate or Recall: $\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$

- False positive rate: $\frac{\mathrm{FP}}{\mathrm{TN}+\mathrm{FP}}$

Which of these measures is best depends on the context.

For example if you are testing for cancer then you might want a high true positive rate (a small false negative rate) and aren't too concerned about high false positives.

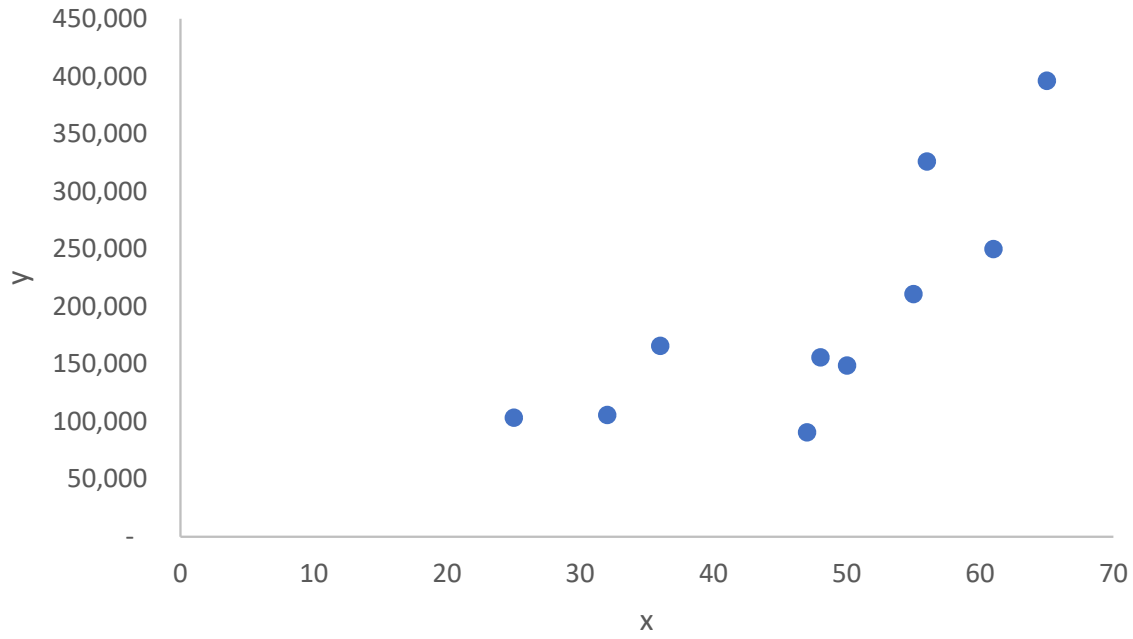
On the other hand you don't want to go around predicting everyone has cancer.

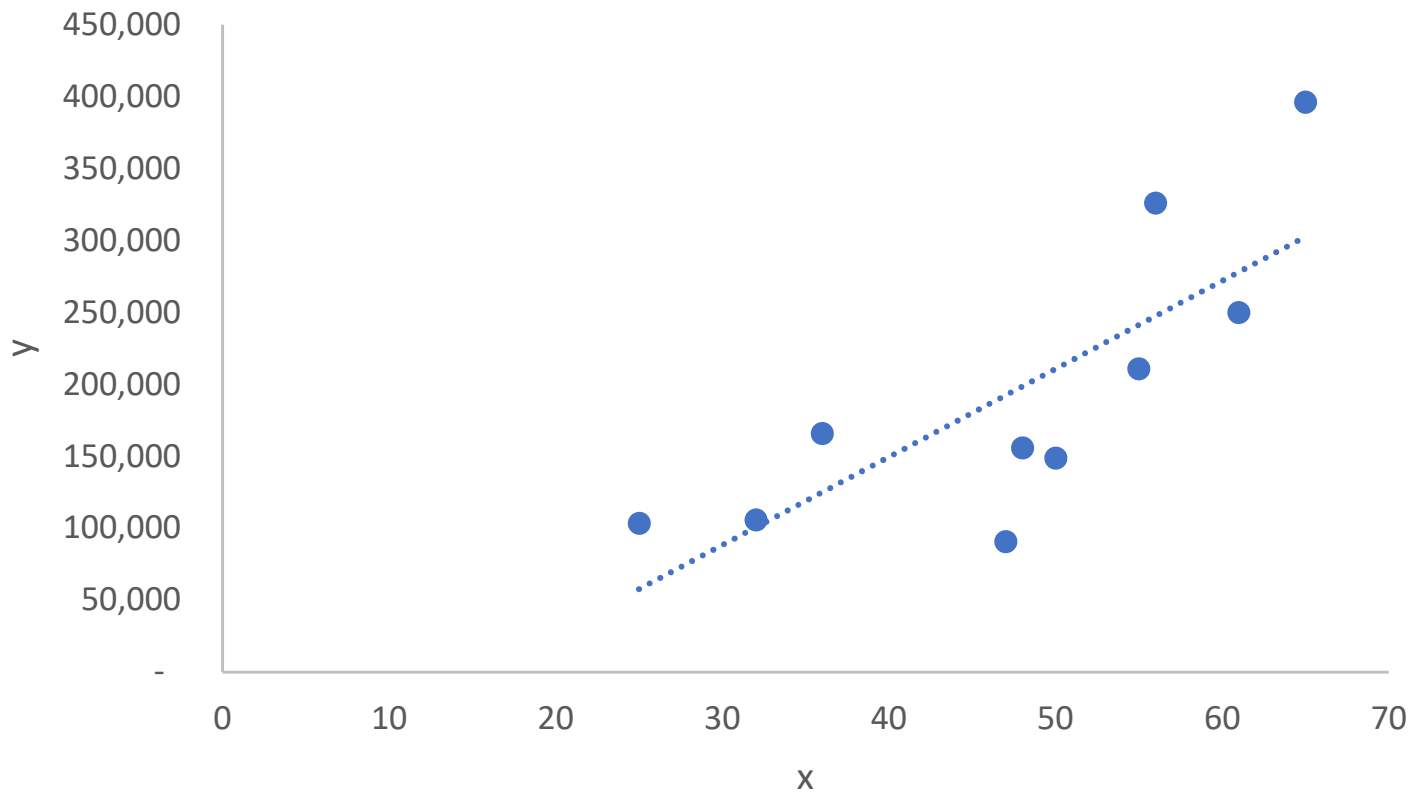
## Cost Functions

In machine learning a cost function or loss function is used to represent how far away a mathematical model is from the real data. One adjusts the mathematical model, perhaps by varying parameters within the model, so as to minimize the cost function. This is then interpreted as giving the best model, of its type, that fits the data.

Cost functions can take many forms. They are usually chosen to have properties that make some mathematical sense for the problem under investigation and also to have some tractability.

Let's look at an example. Suppose we have data for age and salary of a sample of lawyers then we might want to look for a linear relationship between the two. Old lawyers earn more perhaps. We might represent the $n^{\text{th}}$ lawyer's age by $x^{(n)}$ and their salary by $y^{(n)}$. And we'll have $N$ of them.





We want to find a relationship of the form

$$y = \theta_0 + \theta_1 x, \tag{4}$$

where the $\theta$s are the parameters that we want to find to give us the best fit to the data. (Shortly I'll use $\boldsymbol{\theta}$ to represent the vector with entries being these $\theta$s.) Call this linear function $h_\theta(x)$, to emphasise the dependence on both the variable $x$ and the two parameters, $\theta_0$ and $\theta_1$.

We want to measure how far away the data, the $y^{(n)}$s, are from the function $h_\theta(x)$. A common way to do this is via the quadratic cost function

$$J(\boldsymbol{\theta}) = \frac{1}{2N} \sum_{n=1}^{N} \left( h_\theta(x^{(n)}) - y^{(n)} \right)^2. \tag{5}$$

We want the parameters that minimize (5). This is called ordinary least squares (OLS).

This is a popular choice for the cost function because obviously it will be zero for a perfect, straight line, fit, but it also has a single minimum. This minimum is easily found analytically, simply differentiate (5) with respect to both $\theta$s and set the result to zero:

$$\sum_{n=1}^{N} \left(\theta_0 + \theta_1 x^{(n)} - y^{(n)}\right) = \sum_{n=1}^{N} x^{(n)} \left(\theta_0 + \theta_1 x^{(n)} - y^{(n)}\right) = 0.$$

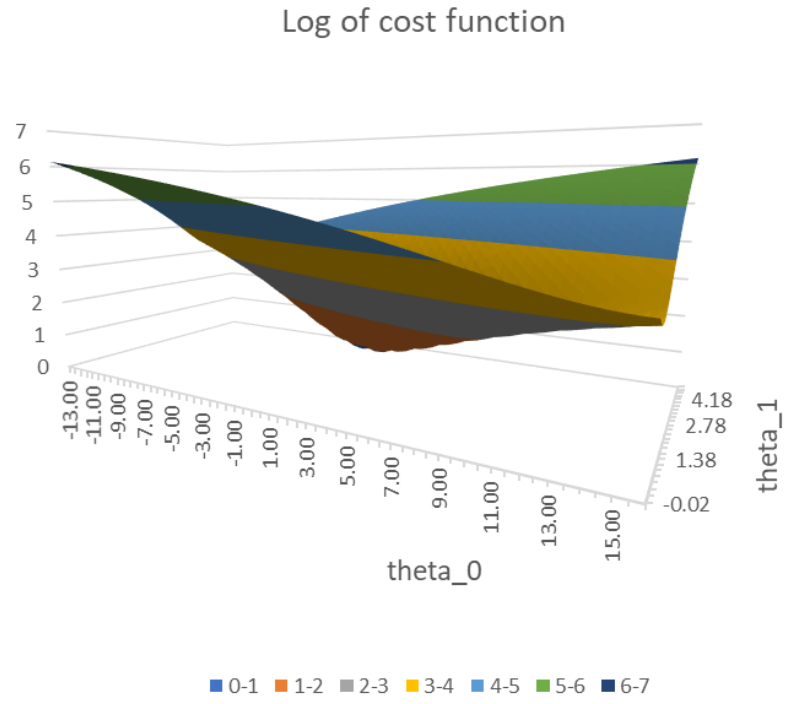
This can be trivially solved for the two $\theta$s:

$$\theta_0 = \frac{(\sum y)\left(\sum x^2\right) - (\sum x)(\sum xy)}{N\sum x^2 - (\sum x)^2}$$

and

$$\theta_1 = \frac{N(\sum xy) - (\sum x)(\sum y)}{N\sum x^2 - (\sum x)^2}.$$

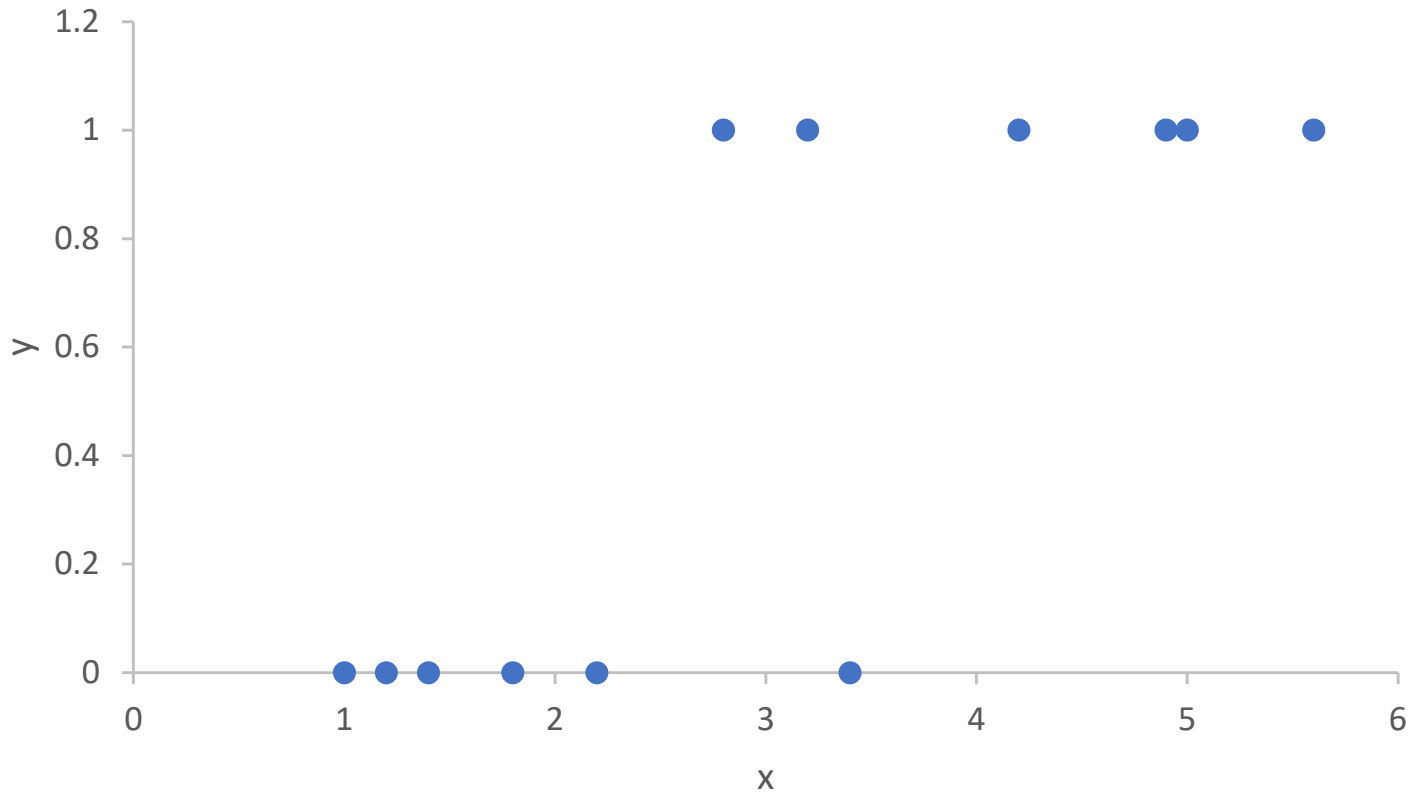
The cost function as a function of $\theta_0$ and $\theta_1$ is shown below. Actually it's the logarithm of the cost function. This makes it easier to see the minimum, without taking logs the 'wings' in this plot would dominate the picture.



Other cost functions are possible, depending on the type of problem you have. For example if we have a classification problem — "This is an apple" instead of "0.365" — then we won't be fitting with a linear function.

If it's a binary classification problem then apples would be 1, and non apples would be 0, say. And every unseen fruit that we want to classify is going to be a number between zero and one. The closer to one it is the more likely it is to be an apple.



In that situation we would typically fit an S-shaped sigmoidal function, one that takes values from zero to one also. An example would be

$$h_\theta(x) = \frac{1}{1 + e^{-\theta_0 - \theta_1 x}}.$$

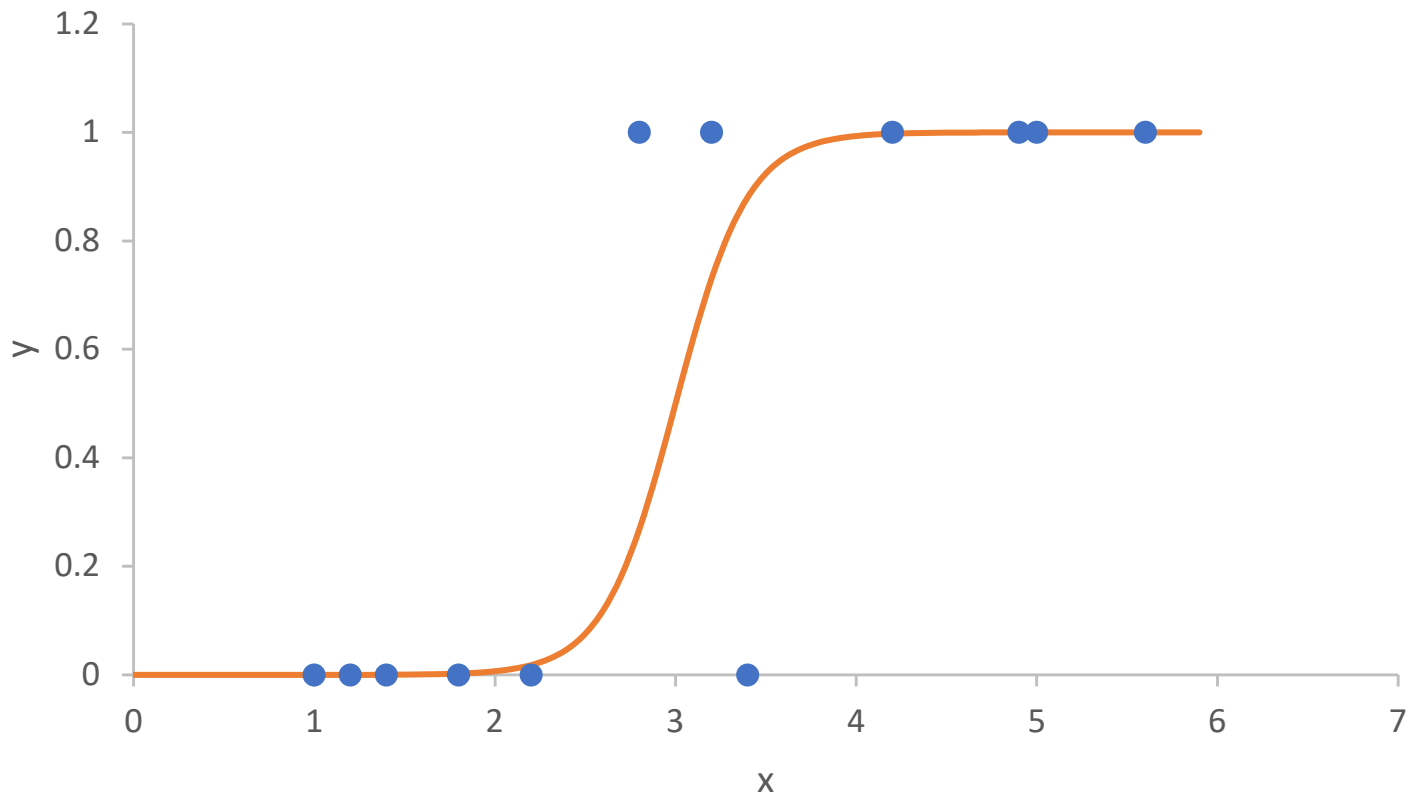
The quadratic cost function is not suitable in this situation. Instead we often see the following.

$$J(\boldsymbol{\theta}) = -\frac{1}{N} \sum_{n=1}^{N} \left( y^{(n)} \ln \left( h_\theta(x^{(n)}) \right) + (1 - y^{(n)}) \ln \left( 1 - h_\theta(x^{(n)}) \right) \right).$$

This looks a bit strange at first, it's not immediately clear where this has come from.

- But if you compare it with (1) then you'll start to see there might be a link between minimizing this cost function and maximizing the likelihood of something (there's a sign difference between the two). The analogy is simply that the $y$s represent the class of the original data (the head/tail or apple/non apple) and the $h_\theta$ is like a probability (for $y$ being 1 or 0).

This cost function is also called the **cross entropy** between the two probability distributions, one distribution being the $y$s, the empirical probabilities, and the other being $h_\theta$, the fitted function.



## Regularization

Sometimes one adds a regularization term to the cost function. Here's an example applied to OLS in a single variable:

$$J(\boldsymbol{\theta}) = \frac{1}{2N}\left(\sum_{n=1}^{N}\left(h_\theta(x^{(n)}) - y^{(n)}\right)^2 + \lambda\theta_1^2\right).$$

The addition of the second term on the right encourages the optimization to find simpler models. Note that it doesn't include the $\theta_0$ parameter.

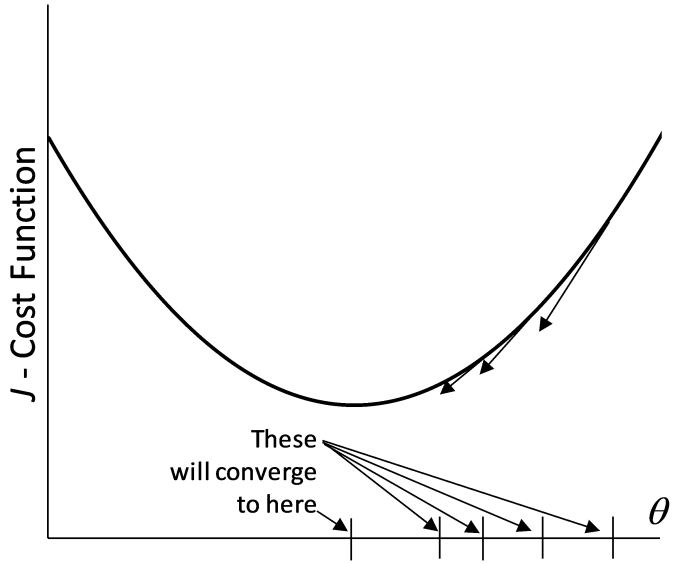
It's probably easiest to see this working if you think of fitting a polynomial to some data. The regularization terms reduces the size of all the coefficients, except for the first one representing a constant level, thus reducing possibly irrelevant wiggles.

## Gradient Descent

So you want to find the parameters that minimize a loss function. And almost always you are going to have to do this numerically.

If we have a nice convex function then there is a numerical method that will converge to the solution, it is called gradient descent. Even if it has local minima there are things we can do. The method is illustrated here.



The scheme works as follow. Start with an initial guess for each parameter $\theta_k$. Then move $\theta_k$ in the direction of the slope:

$$\text{New } \theta_k = \text{Old } \theta_k - \beta \, \partial J / \partial \theta_k.$$

Update all $\theta_k$ simultaneously, and repeat until convergence.

Here $\beta$ is a learning factor that governs how far you move. If $\beta$ is too small it will take a long time to converge. If too large it will overshoot and might not converge at all. It is clear that this method might converge to only a local minimum if there is more than one such.

The loss function $J$ is a function of all of the data points, i.e. their actual values and the fitted values. In the above description of gradient descent we have used all of the data points simultaneously. This is called 'batch' gradient descent.

But rather than use all of the data in the parameter updating we can use a technique called stochastic gradient descent. This is like gradient descent except that you only update using *one* of the data points each time. And that data point is chosen randomly. Hence the 'stochastic.'

Let me write the cost function as

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} J_n(\boldsymbol{\theta}),$$

with the obvious interpretation. So in the case of OLS, Equation (5), we would have

$$J_n(\boldsymbol{\theta}) = \frac{1}{2N}\left(h_\theta(x^{(n)}) - y^{(n)}\right)^2.$$

$J_n(\boldsymbol{\theta})$ is the contribution of data point $n$ to the cost function.

Stochastic gradient descent means pick an $n$ at random and then update according to

$$\text{New } \theta_k = \text{Old } \theta_k - \beta\, \partial J_n/\partial\theta_k.$$

Repeat, picking another data point at random, etc.

There are several reasons why we might want to use stochastic gradient descent instead of batch. For example:

- Since the data points used for updating are chosen at random the convergence will not be as smooth as batch gradient descent. But surprisingly this can be a good thing. If your loss function has local minima then the 'bouncing around' can bounce you past a local minimum and help you converge to the global minimum.

- It can be computationally more efficient. If you have a very large dataset it is likely that many of the data points are similar so you don't need to use all of them. If you did it would take longer to update the parameters without adding much in terms of convergence.

And then there's 'mini batch gradient descent' in which you use subsets of the full dataset, bigger than 1 and smaller than $n$, again chosen randomly.

Finally if you are using a stochastic version of gradient descent you could try smoothing out the bouncing around, if it's not being helpful, by taking an average of the new update and past updates, leading to an exponentially weighted updating. This is like having momentum in your updating and can speed up convergence. You'll need another parameter to control the amount of momentum.

## Training, Testing And Validation

Most machine-learning algorithms need to be trained. That is, you give them data and they look for patterns, or best fits, etc. They know they are doing well when perhaps a loss function has been minimized, or the rewards have been maximized. You'll see all of this later.

But you have to be careful that the training is not overfitting.

You don't want an algorithm that will give perfect results using the data you've got.

No, you want an algorithm that will do well when you give it data it hasn't seen before.

When doing the training stage of any algorithm you could use all of the data. But that doesn't tell you how robust the model is. To do this properly there are several things you should try.

- First of all, divide up the original data into training and test sets. Do this randomly, maybe use 75% for training and then the remaining 25% for testing. How well does the trained algorithm do on the test data?

Another thing you can do, if there is any time dependence in your data, maybe it was collected over many years, instead of taking 25% out at random you split the data into two halves, before and after a certain date. Now train these two data sets independently. Do you get similar results? If you do then that's great, the model is looking reliable. If you don't then there are three, at least, explanations.

One is that the model is fine but there are regions in one of the two halves that are just not reached in the other half.

Second, there might have been a regime shift. That is, the world has simply changed, and you need a time-dependent model.

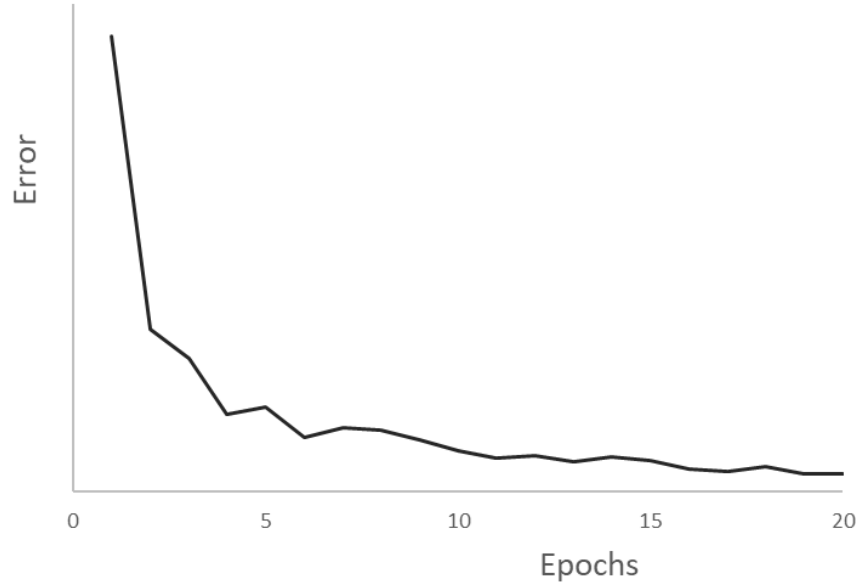
Or maybe you are simply barking up the wrong tree with whatever algorithm you are using.

## Epochs

In some machine-learning methods one uses the same training data many times, as the algorithm gradually converges, for example, in stochastic gradient descent. Each time the whole training set of data is used in the training that is called an epoch or iteration.

Typically you won't get decent results until convergence after many epochs.

One sees a decreasing error as the number of epochs increases, as shown below.

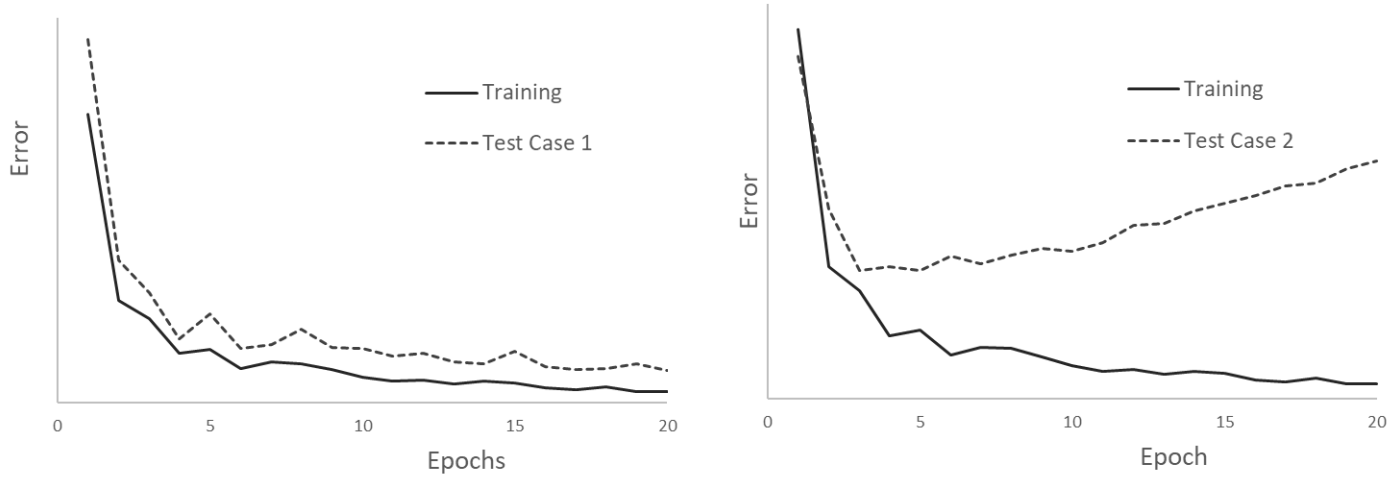



But that does not mean that your algorithm is getting better, it could easily mean that you are overfitting.

This can happen if the algorithm has seen the training data too many times, i.e. there have been too many epochs.

To test for this you introduce the test set of data, the data that you have held back. All being well you will get results on the left below. The test set is not as good as the training set, obviously, but both are heading in the right direction. You can stop training when the errors have levelled out consistently.




On the other hand if you get results like on the right above where the test-set error begins to rise again then you have overfitted.

To help avoid overfitting sometimes we divide up our original data into *three* sets.

The third data set is called the 'validation set.' We train the data on the training set as before. Then after a few epochs we use the validation set to see how the algorithm copes with out-of-sample data.

We keep doing this with more and more epochs. If we see the error for the validation set rising then we stop at that number of epochs. But we can't be sure that the algorithm hasn't learned to cope with new samples because our algorithm 'knows about' the validation set since we used it to decide when to stop. So now we bring in the test data and see how the algorithm copes with that.

## Bias And Variance

Suppose there is a relationship between an independent variable, $x$, and a dependent variable, $y$, given by

$$y = f(x) + \epsilon.$$

Here $\epsilon$ is an error term, $\epsilon$ has mean of zero (if it wasn't zero it could be absorbed into $f(x)$) and variance $\sigma^2$. The error term, whose variance could also be $x$ dependent, captures either genuine randomness in the data or noise due to measurement error.

And suppose we find, using one of our machine-learning techniques, a deterministic model for this relationship:

$$y = \tilde{f}(x).$$

Now $f$ and $\tilde{f}$ won't be the same. The function our algorithm finds, $\tilde{f}$, is going to be limited by the type of algorithm we use. And it will have been fitted using a lot of real training data.

And that fitting will probably be trying to achieve something like minimizing

$$\sum_{n=1}^{N} \left( \tilde{f}(x^{(n)}) - y^{(n)} \right)^2 .$$

The more complex, perhaps in terms of parameters, the model then the smaller this error will be. Smaller, that is, for the training set.

Now along comes a new data point, $x'$, not in the training set, and we want to predict the corresponding $y'$. We can easily see how much is the error in our prediction.

The error we will observe in our model at point $x'$ is going to be

$$\tilde{f}(x') - f(x') - \epsilon. \tag{6}$$

There is an important subtlety here. Expression (6) looks like it only has the error due to the $\epsilon$, because $f$ and $\tilde{f}$ are deterministic.

But it's also hiding another, possibly more important and definitely more interesting, error, and that is the error due to what is in our training data.

A robust model would give us the same prediction whatever data we used for training our model.

So let's look at the average error, the mean of (6), given by

$$E\left[\tilde{f}(x')\right] - f(x')$$

where the expectation $E[\cdot]$ is taken over random samples of training data (having the same distribution as the training data).

This is the definition of the bias,

$$\text{Bias}(\tilde{f}(x')) = E\left[\tilde{f}(x')\right] - f(x').$$

We can also look at the mean square error. And since $\tilde{f}(x')$ and $\epsilon$ are independent we easily find that

$$E\left[(\tilde{f}(x') - f(x') - \epsilon)^2\right] = \Big(\text{Bias}(\tilde{f}(x'))\Big)^2 + \text{Var}(\tilde{f}(x')) + \sigma^2, \quad (7)$$

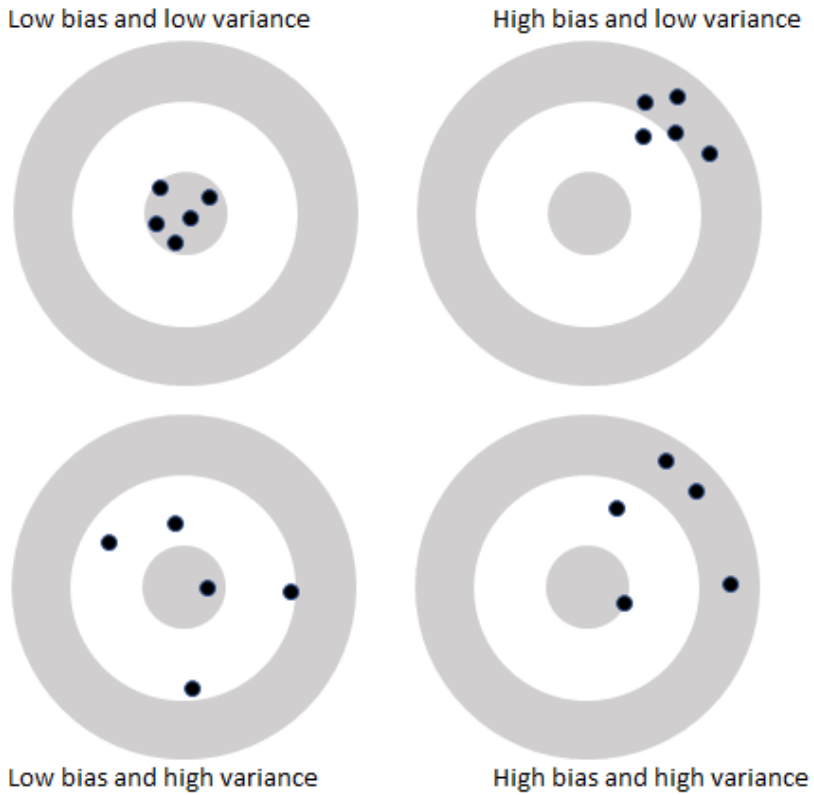
where

$$\text{Var}(\tilde{f}(x')) = E\left[\tilde{f}(x')^2\right] - E\left[\tilde{f}(x')\right]^2.$$

Expression (7) is the mean square error for our new prediction.

This shows us that there are two important quantities, the bias and the variance, that will affect our results and that we can control to some extent by tweaking our model. The squared error (7) is composed of three terms, the bias of the method, the variance of the method and a term that we are stuck with (the $\sigma^2$).

A good model would have low bias and low variance as illustrated here.



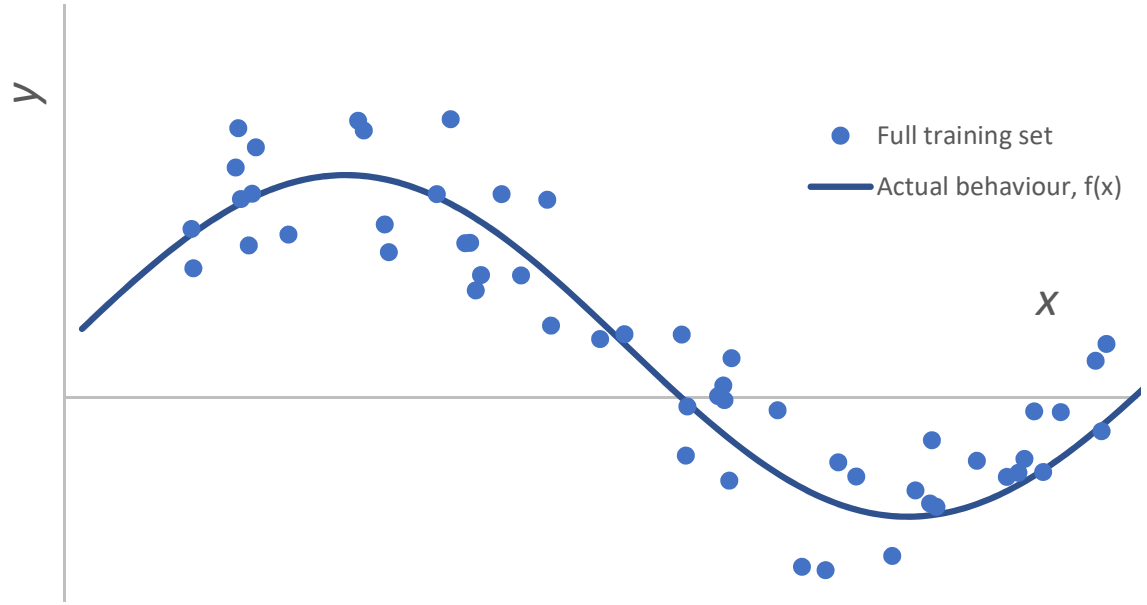
Bias is how far away the trained model is from the correct result on average. Where "on average" means over many goes at training the model, using different data. And variance is a measure of the magnitude of that error.

Unfortunately, we often find that there is a trade-off between bias and variance. As one is reduced, the other is increased. This is the matter of over- and underfitting.

Overfitting is when we train our algorithm too well on training data, perhaps having too many parameters. An analogy is the two-year old playing with his first jigsaw puzzle. After many attempts, and quite a few tears, he finally succeeds. Thereafter he finds it much easier to solve the puzzle. But has he really learned how to do jigsaw puzzles or has he just memorized the one picture?

Underfitting is when the algorithm is too simple or not sufficiently trained. It might work on average in some sense but fails to capture nuances of the data.

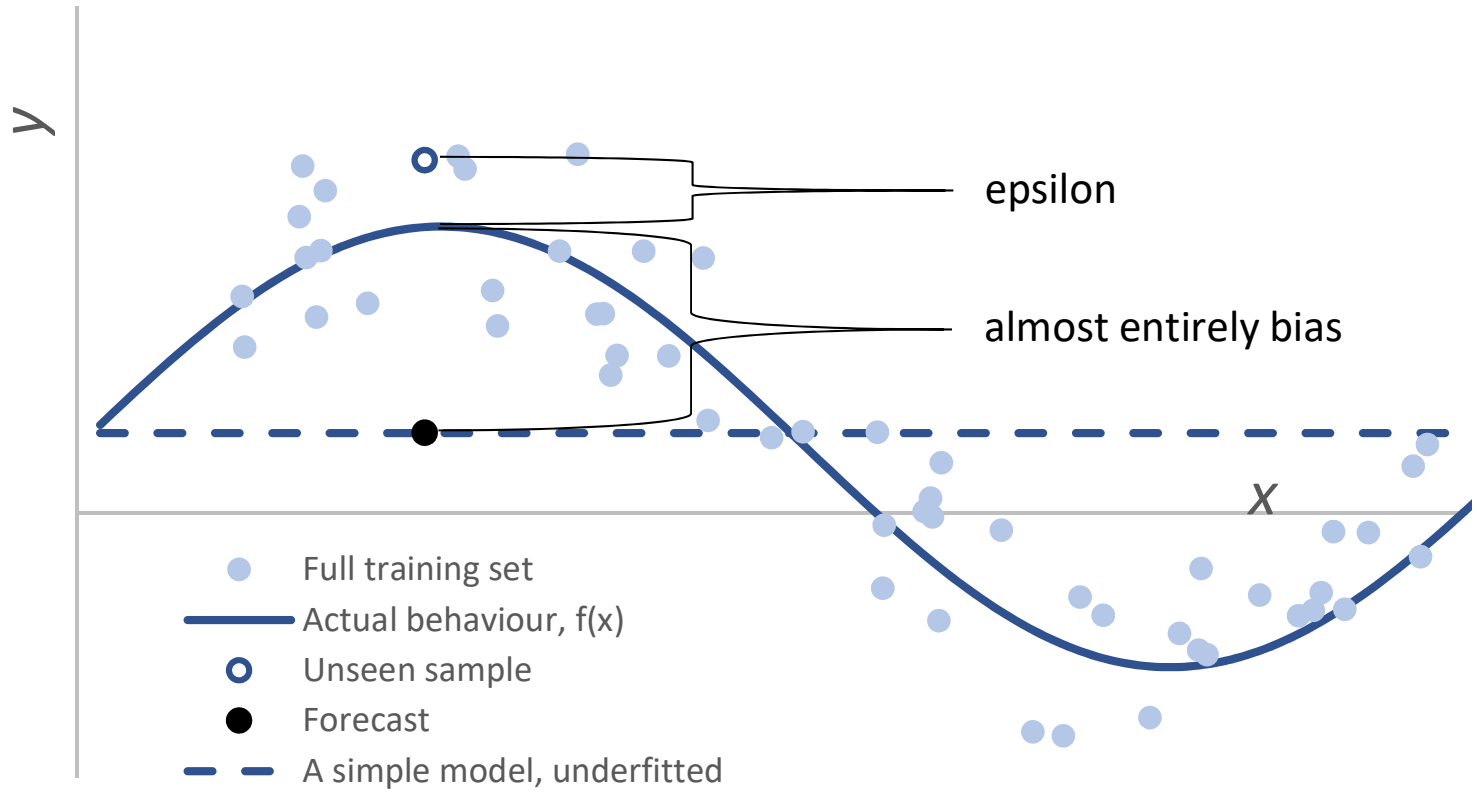
Let's see how this works with a simple example. The figure shows the relationship between $y$ and $x$, including the random component $\epsilon$. The function $f(x)$ is just a shifted sine wave, and there's a uniformly distributed random variable added to it. Each dot represents a sample, and the full training set is shown.



Let's start with a simple model for the relationship between $x$ and $y$, i.e. a simple $\tilde{f}(x)$.

It doesn't get any simpler than a constant. So in the next figure I have shown our first model, it is just a constant, the dashed line. I have chosen for this simple model the average $y$ value for a *subset of all the data*. It doesn't actually matter which subset of the data because if I changed the subset the average wouldn't change all that much. This is clearly very underfitted, to say the least.

Along comes an unseen sample, represented by the hollow dot in the figure. But our forecast is the solid dot. There are three sources of error, the error due to the $\epsilon$, the variance and the bias.




The simple 'model' has been fitted to *all* of the training data. **How would the model and the forecast change if I only used half of the data, randomly chosen, for training?**

The $\epsilon$ we are stuck with, we can't forecast that, although it is in the training data and will therefore be implicitly within the model. The variance (of the model) is tiny.

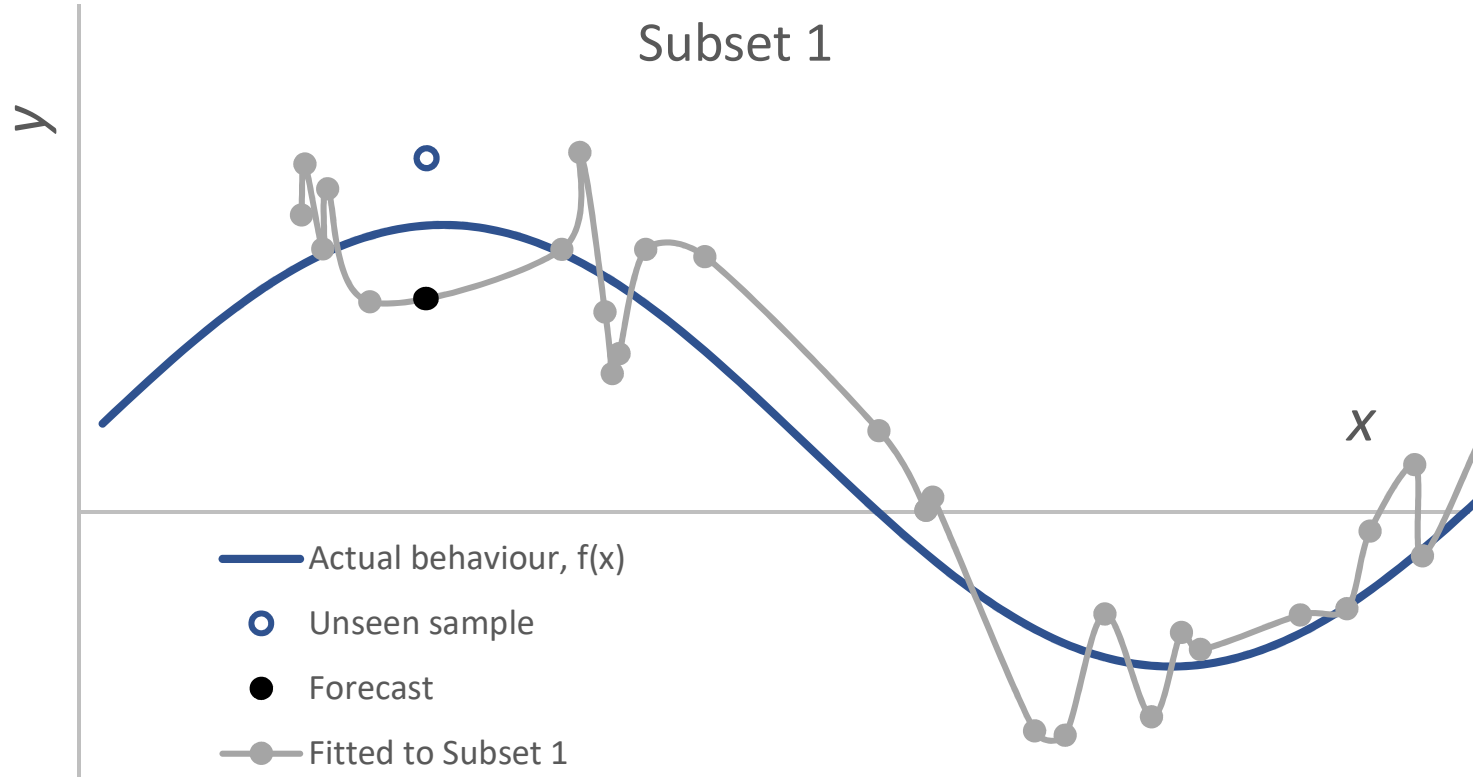
If I used a different subset of the training data then the model, here the average, would hardly change at all. So most of the model error is the bias.

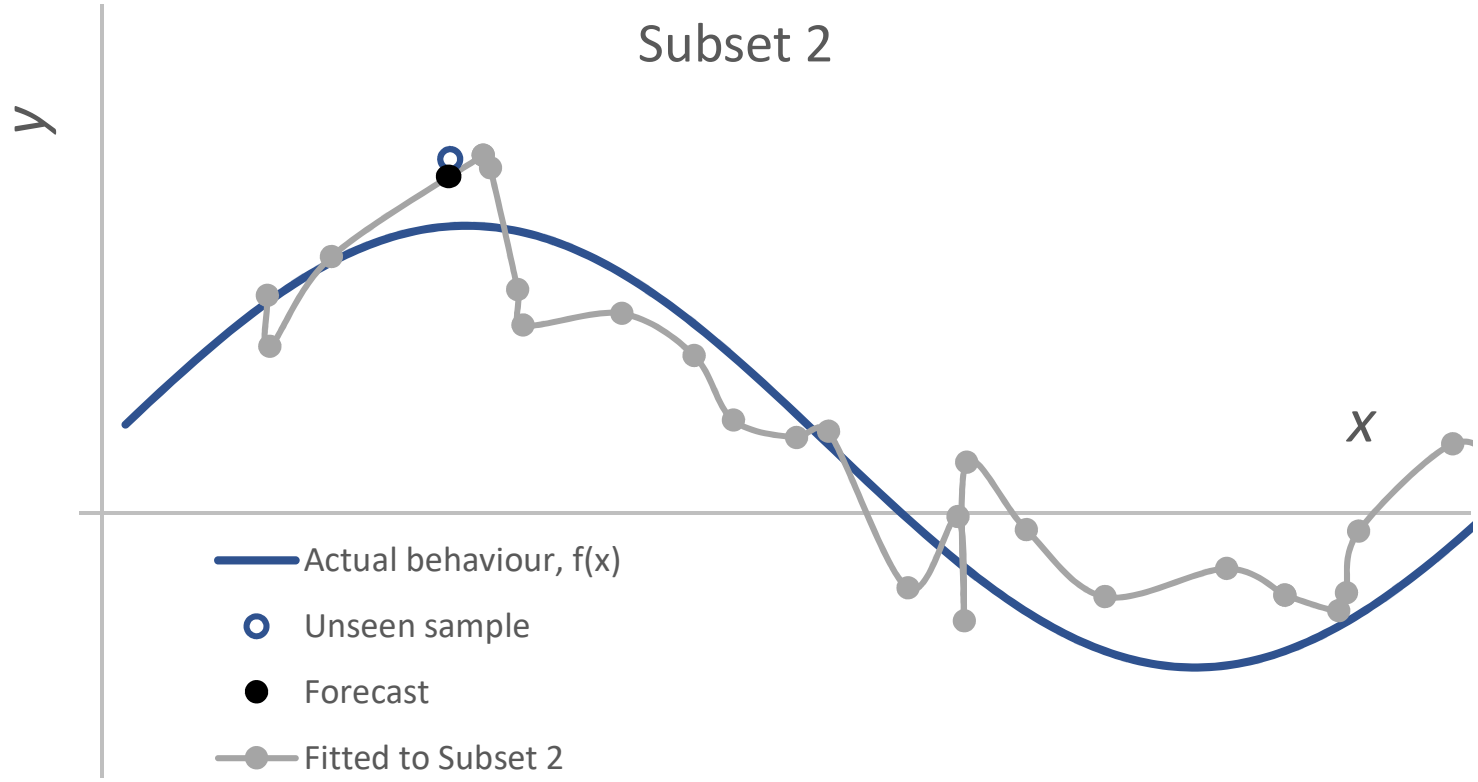
For an underfitted model the mean squared error will be dominated by the bias and the $\epsilon$.

In the next two figures we see what happens when we use a more complicated model and two different subsets of the full set of training data.

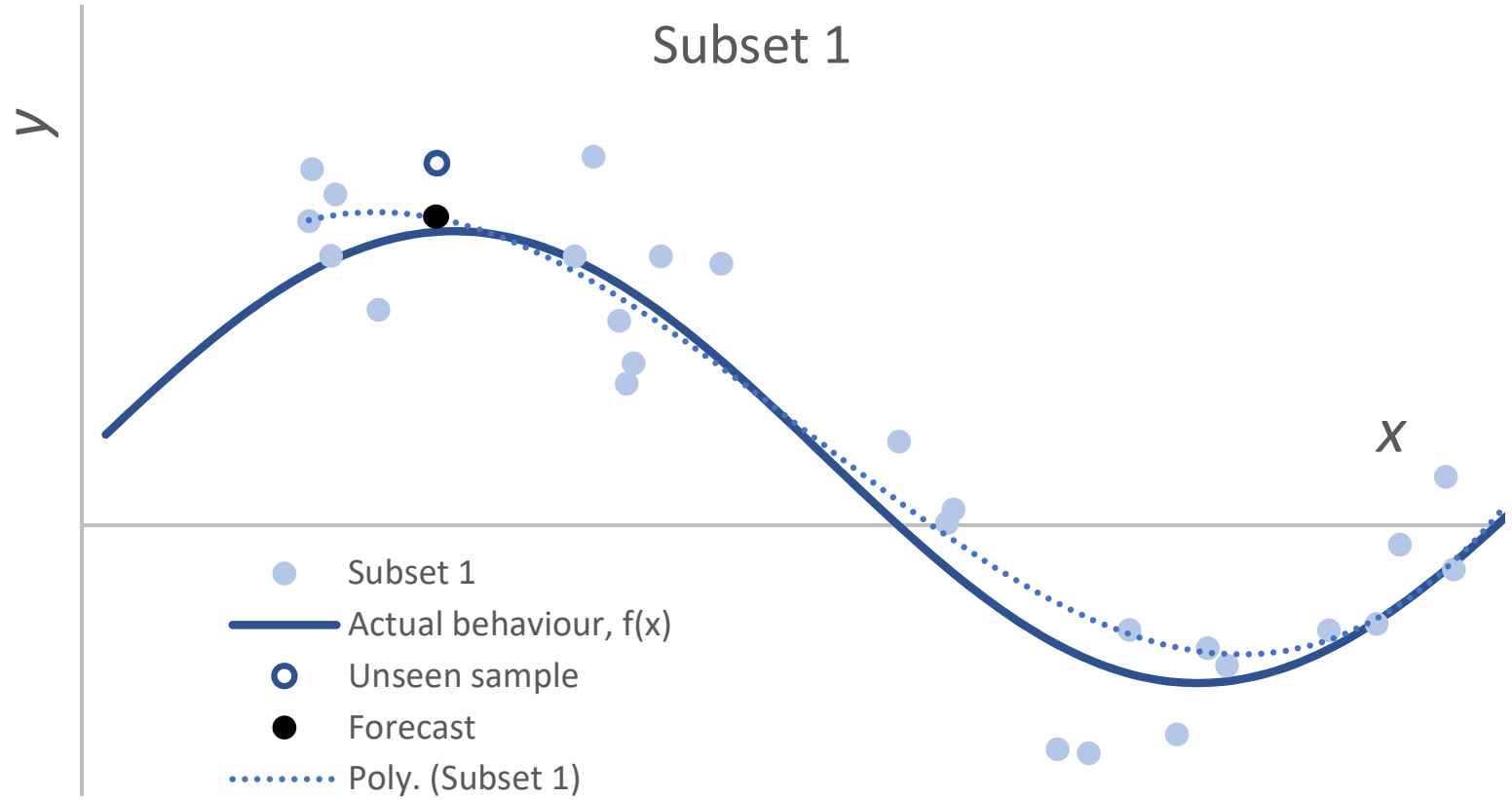
In both cases I have used Excel's smooth-fitting option with a subset of the data, so that's two different models and thus two different forecasts.

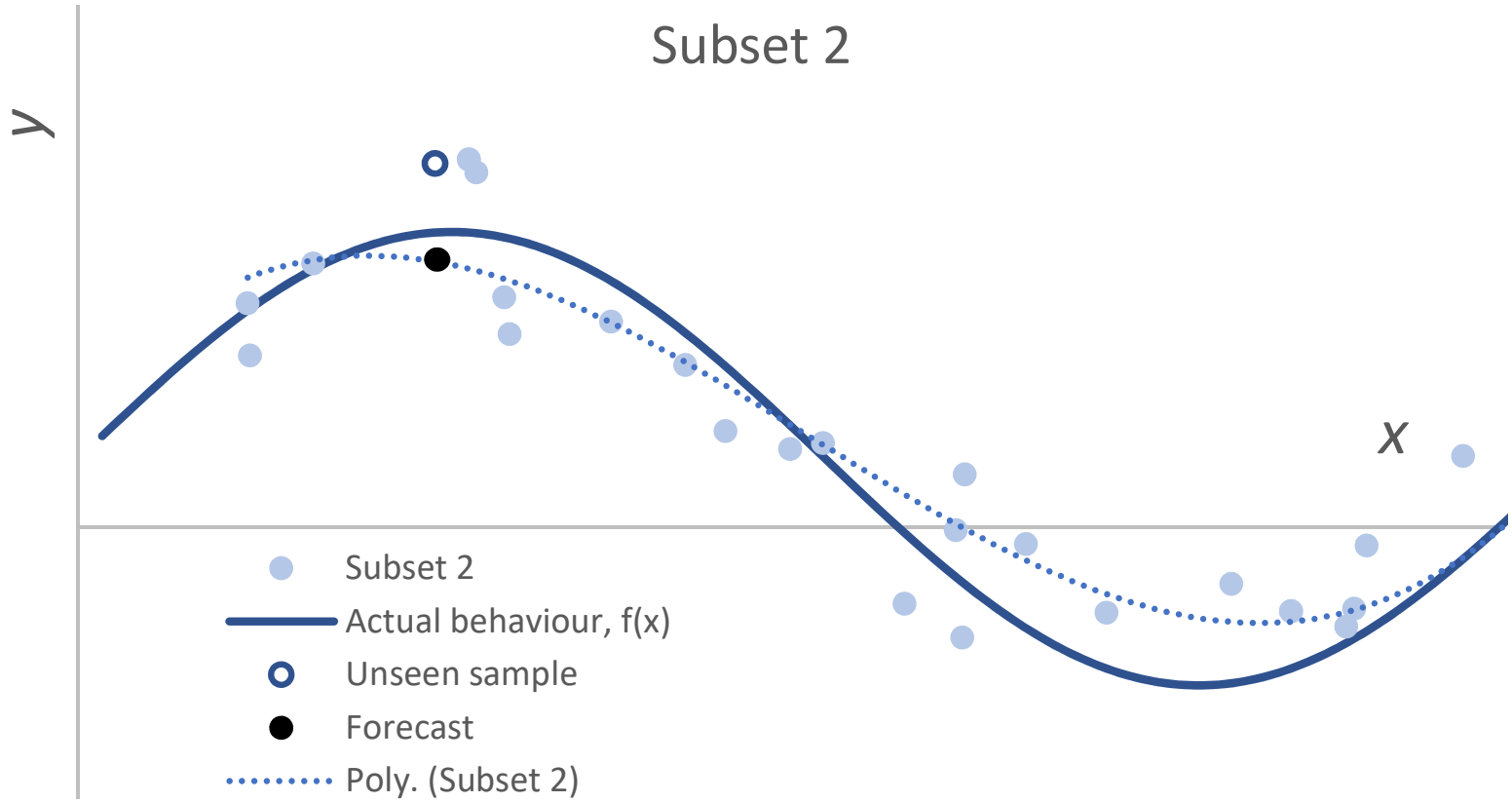
The forecasts move about because of both the bias and the variance in the model.



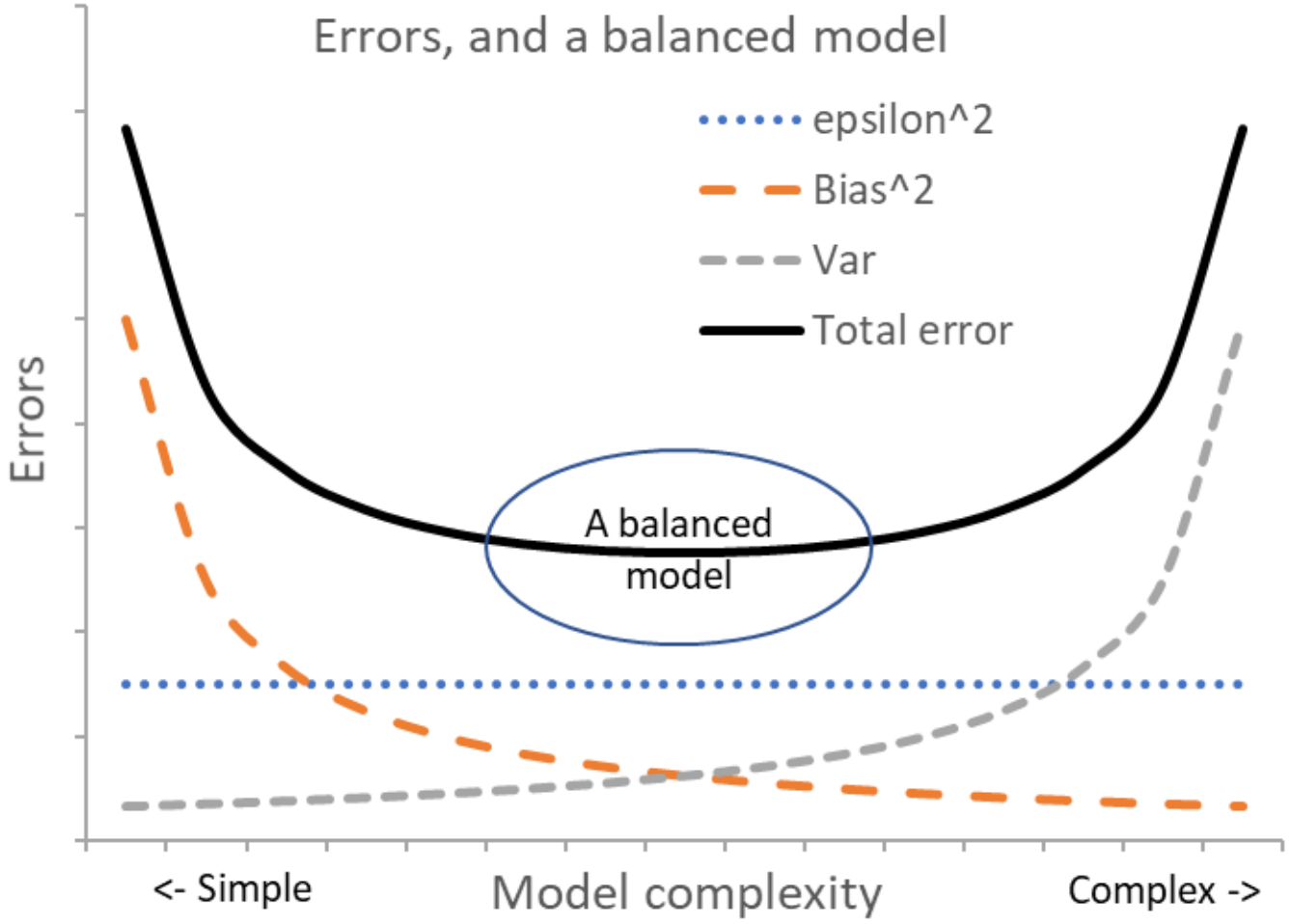


Finally, two more plots, fitted to the same subsets but using a simpler model, just a cubic polynomial. You see much less bias and variance now.





The trade off between bias and variance is shown here.



## Information Theory And Entropy

In its original form information theory concerned the transmission of messages.

If you have a crackly telephone line then messages might be corrupted between the transmitter and the receiver.

A simple Morse code message of dots and dashes could very easily be messed up as some dots turn to dashes and vice versa.

Part of information theory is the important concept of information entropy, or simply entropy.

If the sun rises in the East there is no information content, the sun always rises in the East. If you toss an unbiased coin then there is information in whether it lands heads or tails. If the coin is biased towards heads there is more information if it lands tails.

**Surprisal** associated with an event is the negative of the logarithm of the probability of the event

$$-\log_2(p).$$

You can use different bases for the logarithm, but it only makes a scaling difference. If you use base 2 then the units are the familiar 'bits.' If the event is certain, $p = 1$, the information associated with it is zero. The lower the probability of an event the higher the surprise, becoming infinity when the event is impossible.

But why logarithms?

The logarithm function occurs naturally in information theory. Consider for example the tossing of four coins. There are 16 possible states for the coins, HHHH, HHHT, . . . , TTTT. But only four bits of information are needed to describe the state. HTHH could be represented by 0100.

$$4 = \log_2(16) = -\log_2(1/16).$$

Going back to the biased coin, suppose that the probability of tossing heads is 3/4 and 1/4 of tossing tails. If I toss heads then that was almost expected, there's not that much information.

Technically it's $-\log_2(0.75) = 0.415$ bits. But if I toss tails then it is $-\log_2(0.25) = 2$ bits.

This leads naturally to looking at the average information, this is our entropy:

$$-\sum p \log_2(p),$$

where the sum is taken over all possible outcomes. (Note that when there are only two possible outcomes the formula for entropy must be the same when $p$ is replaced by $1-p$. And this is true here.)

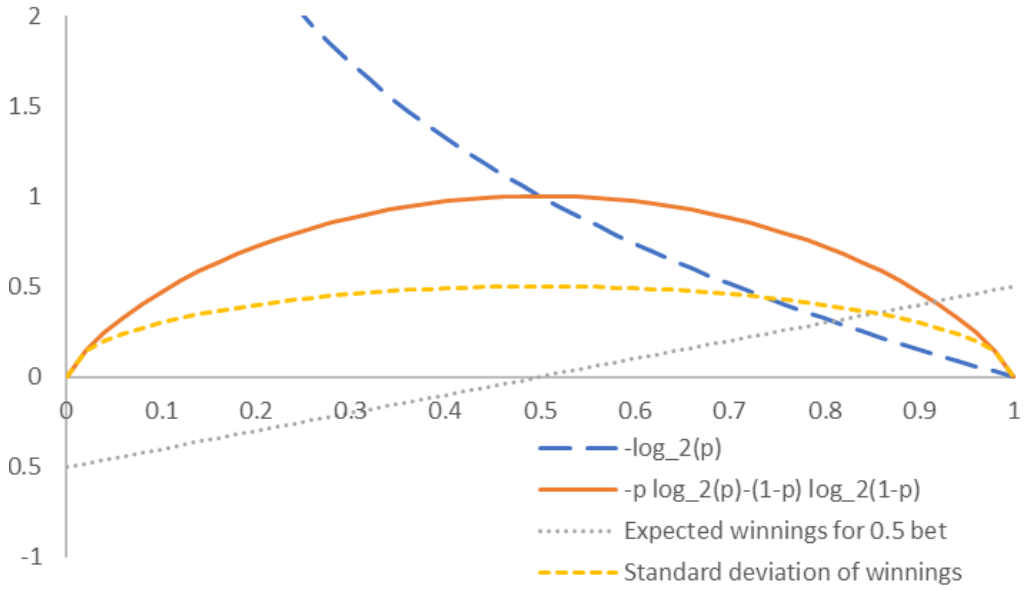
For an unbiased coin the entropy is easily found to be 1.

For a coin with zero probability of either heads or tails then the entropy is zero.

For the 75:25 coin it is 0.811.

Entropy is a measure of uncertainty, but uncertainty linked to information content rather than the uncertainty associated with betting on the outcome of the coin toss.

Below I plot four quantities for the coin tossing, all against $p$ the probability of heads. There is the information function, the $-\log_2(p)$, the entropy $-p\log_2(p) - (1-p)\log_2(1-p)$ and for comparison a couple of lines usually associated with coin tossing. One is the expected winnings for betting \$0.5 on heads, $p - 0.5$, and the standard deviation of the winnings $\sqrt{p(1-p)^2 + (1-p)p^2}$. You can see that the standard deviation, the risk, in the coin toss is qualitatively similar to the entropy.



Entropy is going to be important when we come to decision trees, it will tell us how to decide what order in which to ask questions so as to minimize entropy, or, as explained later, maximize information gain.

## Natural Language Processing

Some of the methods we will look at are often used for Natural Language Processing (NLP). NLP is about how an algorithm can understand, interpret, respond to or classify text or speech.

NLP is used for

- **Text/sentiment classification:** Spam emails versus non spam is the example often given. And is a movie review positive or negative?

- **Answering questions:** Determining what a question is about and then searching for an answer

- **Speech recognition:** "Tell us in a few words why you are contacting BT today"

Text is obviously different from numeric data and this brings with it some special problems. Text also contains nuances that a native speaker might understand intuitively but which are different to get across to an algorithm. No kidding?

Here are a few of the pre-processing techniques we might need to employ and issues we need to deal with when we have text.

- **Tokenize:** Break up text into words plus punctuation (commas, full stops, exclamation marks, question marks, ellipses, . . . )

- **Stop words:** Some words such as 'a,' 'the,' 'and,' etc. might not add any information and can be safely removed from the raw text

- **Stemming:** One often reduces words to their root form, i.e. cutting off their endings. Meaning might not be lost and analytical accuracy might be improved. For example, all of 'love,' 'loves,' 'loving,' 'loved,' 'lover,' . . . would become 'lov'

- **Lemmatize:** Lemmatizing is a more sophisticated version of stemming. It takes into account irregularity and context. So that 'eat,' 'eating,' 'ate,' . . . are treated as all being one word

- **Categorize:** Identify words as nouns, verbs, adjectives etc.

- **'N'-grams:** Group together multiple words into phrases such as 'red flag,' 'jump the shark,' etc.

- **Multiple meanings:** Sometimes one needs to look at surrounding words to get the correct meaning. 'The *will* of the people' or 'I *will* have pizza for lunch' or 'In his *will* he left all his money to his cat.' Some words might be spelled the same, and have the same root, but be subtly different, for example, 'read' and 'read'

## Word2vec

Word2vec are methods that assign vectors to words. This requires a large number of dimensions to capture a decent vocabulary.

Words that have something in common end up, after training, near each other.

Rather cleverly, combinations of words can be represented by simple vector operations, such as addition and subtraction. For example, 'fish' + 'chips' + 'mushy peas' might result in a vector close to the 'dinner' vector.

## Summary

Please take away the following important ideas

- Machine learning has its own jargon
- But most of the methods used as part of machine learning come from classical statistics, numerical analysis or mathematical methods