Chapter 3

Machines finding functions

The opening pages of machine learning textbooks often warn or enthuse about the profusion of techniques. 'The literature on machine learning is vast, as is the overlap with the relevant areas of statistics and engineering' writes David Barber in *Bayesian Reasoning and Machine Learning*(Barber 2011,4); 'statistical learning refers to a vast set of tools for understanding data' writes James and co-authors in an *Introduction to Statistical Learning with R* (James et al. 2013,1); or Matthew Kirk in *Thoughtful Machine Learning* comments:

> Machine learning is an amazing application of computation because it tackles problems that are straight out of science fiction. These algorithms can solve voice recognition, mapping, recommendations, and disease detection. The applications are endless, which is what makes machine learning so fascinating. This flexibility is also what makes machine learning daunting. It can solve many problems, but how do we know whether we're solving the right problem, or actually solving it in the first place? (Kirk 2014, ix)

From an *almost* purely technical standpoint, machine learning can be understood as function finding, that is, finding a mathematical expression that approximates to the process that generated the data in question. The prefatory comments from Barber, James and Kirk suggest, however, that machine learning comprises a rampant abundance of techniques. Much machine learning work, at least for many practitioners, concerns not so much implementation of particular techniques (neural network, decision tree, support vector machine, logistic

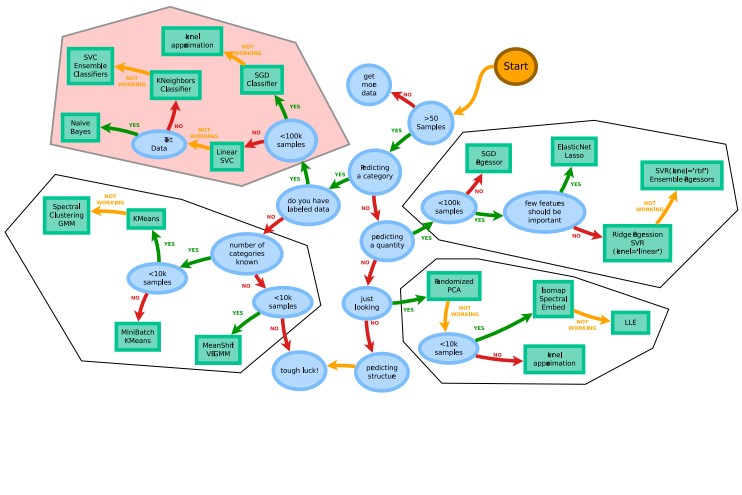


Figure 1: 'scikit-learn' map of machine learning techniques [TBA: ref to diagram]

regression, etc.), but rather navigating the maze of methods and variations that might be relevant to a particular situation.

'Functions' abound not just in the mathematical sense, but in the sense of operational units of code (the `print` function in `Python` writes to the screen, for instance). Finding the right function amidst the variety of functions affects any learning of machine learning. The map of machine learning techniques shown in Figure 1 comes from a particular software library written in `Python`, `scikit-learn` (Pedregosa et al. 2011). It is widely used in industry, research and commerce. In contrast to the pedagogical expositions, theoretical accounts or guides on implementation, code libraries such as `scikit-learn` tend to order the range of techniques by offering recipes and maps for the use of the functions they supply. The branching paths in the figure lay down some ways of negotiating the profusion of techniques. Less technically, machine learning itself has itself come to operate as a powerful diagram for infrastructural, operational, financial, scientific, governmental and marketing processes. Here, the machines

– the classifiers and the predictive models – find functions as they are inserted into infrastructures, experiments, and organisations. How do these different functions – the mathematical mapping, the unit-of-code, and the diagram of power - function together? What generality or positivity concatenates them?

The architecture of these software libraries itself presents a classificatory ordering of the techniques of machine learning. So, `scikit-learn` for instance comprises a number of sub-packages:

```
import sklearn
from sklearn import *
modules = dir(sklearn)
modules_clean = [m for m in modules if not m.startswith('_')]
print(modules_clean)
```

Traceback (most recent call last): File “”, line 1, in ImportError: No module named sklearn

Here functional modules such as `lda` (linear discriminant analysis), `svm` (support vector machine) or `neighbors` (k nearest neighbours) point to prominent models, but `cross-validation` or `feature_selection` refer to ways of testing models or transforming data respectively. While these divisions, maps and classifications help order the techniques, and such orderings are replicated across the content pages, course outlines and how-to tutorials for machine learning, data mining and ‘big data’ more generally, they do not in themselves reveal the problematic that first gave rise to the profusion that needs to be ordered. I propose in this chapter to understand that profusion in terms of a slippage between the two main sense of the function: function as mathematical relation and function as concrete machinic operation. The relationality of the mathematical function and the machinic function are both diagrammatic (as discussed in

previous chapters), but the *diagonals* that run between them are not always easy to see.

Which sense of function?

The mathematical sense of function is writ large in nearly all machine learning literature. Indeed, we might say that machine learning is nothing other than a machinic version of the functions that have long interested and occupied mathematicians. The technical devices of the field, for instance, classifiers, are often identified directly with functions:

> A classifier or classification rule is a function $d(\mathbf{x})$ defined on $\mathcal{X}$ so that for every $\mathbf{x}$, $d(\mathbf{x})$ is equal to one of the numbers $1, 2, ..., J$ (Breiman et al. 1984, 4)

Writing in the 1980s, the statistician Leo Breiman describes classifiers – perhaps the key technical achievement of machine learning – in terms of functions. A classifier *is* a function. The equation of classifiers to functions is quite pervasive. The learning, the predictions, and the classifications produced by machine learning depend on functions. So we need an account of what goes on in this equation of function and classification if we are to make sense of the broader problematisation associated with machine learning. The identification of machine learning with functions appears in the first pages of most machine learning textbooks. Learning in machine learning means finding a function that can identify or predict patterns in the data. As *Elements of Statistical Learning* puts it,

> our goal is to find a useful approximation $\hat{f}(x)$ to the function $f(x)$ that underlies the predictive relationship between input and output (Hastie, Tibshirani, and Friedman 2009, 28).

In a highly compressed form, this statement of goals already re-iterates the triple play on function referred to above. It contains *the* function that generated the data as a foundation, it refers to 'finding … $\hat{f}(x)$', where the '^' indicates an approximation produced by an algorithmic implementations, and it avers to 'use'. Similar formulations pile up in the literature. Perhaps more importantly, *learning* here is understood as function finding. A leading theorist of learning theory Vladimir Vapnik puts it this way: 'learning is a problem of *function estimation* on the basis of empirical data' (Vapnik 1999, 291). (Vapnik is said to have invented the support vector machine, one of the most heavily used machine learning technique of recent years on the basis of his theory of computational learning.) The use of the term 'learning' in machine learning displays affiliations to the field of artificial intelligence, but the attempt to find a 'useful approximation' – the 'function-fitting paradigm' as (Hastie, Tibshirani, and Friedman 2009, 29) terms it – stems mainly from statistics. Not all accounts of machine learning frame the techniques in terms of function fitting. Some retain the language of intelligent machines (see for example, (Alpaydin 2010, xxxvi) who writes: 'we do not need to come up with new algorithms if machines can learn themselves'). Despite any differences in the framing of the techniques, all accounts of machine learning, even those such as *Machine Learning for Hackers* (Conway and White 2012) that eschew any explicit recourse to mathematical formula, rely on the formalism and modes of thought associated with mathematical functions. Whether they are seen as forms of artificial intelligence or statistical models, the formalisms are directed to build 'a good and useful approximation to the desired output' (Alpaydin 2010, 41), or, put more statistically, 'to use the sample to find the function from the set of admissable functions that minimizes the probability of error' (Vapnik 1999, 31). Functions then are important anchor points in machine learning, and if we are make sense

of the transformations of media, or science, or security, then the investment in the notion of a function should interest us.

We have seen some functions already in the previous chapter: the linear regression model that fits a line to a set of points is just such a useful approximation to the actual function that generated the data. The kind of visual pattern it identifies is really elementary – a straight line – but the lengths to which machine learning is prepared to go to fit lines to situations is, as we will see, quite extraordinary. The linear model undergoes some drastic deformations as lines stretch and fold into planes, hyperplanes, and various curved and fitted surfaces, but it remains a function in the mathematical sense of a mapping between input or X values and output or Y values.

As is often the case in working with a massive technical literature, the first problem in making sense of what is happening with function in machine learning concerns sheer abundance. The pages of (Hastie, Tibshirani, and Friedman 2009) are marked with score of references to 'functions': quadratic function, likelihood function, sigmoid function, loss function, regression function, basis function, activation function, penalty functions, additive functions, kernel functions,step function, error function, constraint function, discriminant function, probability density function, weight function, coordinate function, neighborhood function, and the list goes on. Clearly we cannot expect to understand the functioning of all these functions in any great detail. However, even a glance through this prickly list of terms begins to suggest that not only is there quite a heavy reliance on functions in this field (as perhaps in many other science and engineering disciplines), but that the proliferation of functions might itself be a way to map some important operations occurring in and around machine learning. We can also already see in this list that the qualifiers of the term function are diverse. Sometimes, the qualifier refers to a mathematical form – 'quadratic,'

'coordinate', 'basis' or 'kernel'; sometimes it refers to modelling or statistical considerations – 'likelihood', 'regression', 'error,' or 'probability density'; and sometimes it refers to some other concern that might relate to a particular modelling device or diagram – 'activation,' 'weight', 'loss,' 'constraint,' or 'discriminant.' These multiple modes of functions matter, since they support and configure the different senses of mathematical, machinic and diagrammatic function-finding in machine learning.

Implicit too in the formal descriptions of machine learning as function finding cited above (for instance, Vapnik's or Hastie's formulation) is the core mathematical definition of a function. The primary mathematical sense refers to a relation between sets of values or variables. (A variable is a symbol that can stand for a set of numbers or other values.) A function is one-to-one relation between two sets of values. It maps a set of arguments (inputs) to a set of values (outputs, or to use slightly more technical language, it maps between a *domain* and a *co-domain.*) As we have already seen, mathematical functions are often written in formulae of varying degrees of complexity. They are of various genres, provenances, textures and shapes: polynomial functions, trigonometric functions, exponential functions, differential equations, series functions, algebraic or topological functions, etc. Various fields of mathematics have pursued the invention of functions. In machine learning and information retrieval, important functions would include the logistic function (discussed below), probability density functions (PDF) for different probability distributions (Gaussian, Bernoulli, Binomial, Beta, Gamma, etc.;[1]).

Functions appear in machine learning in several different ways: as statements or utterances, as formula-diagrams, as graphic forms and in technical imple-

[1]I will discuss these in greater depth in Chapter 5-6), as well as cost functions and Langrangian functions, etc. Not all functions take numbers as inputs or outputs. Letters, words or almost any other symbol can be values in a function.

mentations as code. Any account of machine learning as function finding needs to map the concatenation of these different elements, none of which alone can anchor the 'learning' that goes on in machine learning.

While functions are often diagrammatically written in formulae, they can often appear in different graphic or operational forms. Many of the functions listed above concern curves, and different ways of generating, moving along, finding or differentiating curves. The historical invention of the term 'function' by G.W. Leibniz in the 17th century relates to curves and their description. Functions for Leibniz describe variations in curves such as their slope, and these variations in curves still underpin key aspects of the function-finding done in machine learning. In contrast to the table, or even the generalized common vector space that encompasses, the curve and the many graphic genres that seek to show curves in different ways, relate to variations. If for instance, we look through the several hundred colour graphic plots in (Hastie, Tibshirani, and Friedman 2009)[2], a striking mixture of network diagrams, scatterplots, barcharts, histograms, heatmaps, boxplots, maps, contour plots, dendrograms and 3D plots appear there. Many of these graphic forms are common in statistics (histograms and boxplots), but some relate specifically to data mining and statistical learning (for instance, ROC – Receiver Operating Curve – or regularization path plots). Only a small proportion of these graphics show data. Nearly all of them either show the results of some modelling technique, contrast the operations of different models, or diagram something of the way that models relate to data. Viewed in terms of their visual composition, the diagrams in the book are dominated not by perspective, or by line (although the framing device of the X-Y axes is ubiquitous), but by curves. Curving lines outnumber all other graphic forms, whether the curves are the 'decision boundaries,' the plots of 'training

[2]A montage of all the graphics in *Elements of Statistical Learning* can be found at [TBC]

errors,' the 'contours of constant density' (Hastie, Tibshirani, and Friedman 2009, 109), or the 'regularization path' for the South African heart data (Hastie, Tibshirani, and Friedman 2009, 126). There is a tension in these graphic forms. Consonant with the vectoral ideal of a straight line that either connects or cuts the data, many of the graphics in the book (and others like it, especially the books written by computer scientists) show straight lines. But these straight lines are deflected and bent in various directions by variations and uncertainties of various kinds. The curves that proliferate in the graphic devices of machine learning often try to the soften the rigidity of the lines or find paths for lines through irregular terrain. Viewed very naively, the contrast between lines and curves in the graphic diagrams of a book such as *Elements of Statistical Learning* suggests that different forces are at work around the data, sometimes aligning and straightening relations (for instance, the many linear models) and sometimes tracing much more non-linear paths through the data (for instance, as in convolutional neural networks). Viewed less naively, the curves and their functions operate as the critical paths that connect data to worlds.

Diagramming with curves: the logistic function

How do curves get into the data? One way they approach the data is through *fitting*. Take the example of *sigmoid* functions. These quite simple functions underpin many classifiers and animate many of the operations of neural network, including their recent re-incarnations in 'deep learning' (Hinton and Salakhutdinov 2006; Mohamed et al. 2011). A well-known example of a sigmoid function, the logistic function, can be written as:

$$f(x) = 1/(1 + e^{-kx}) \tag{1}$$

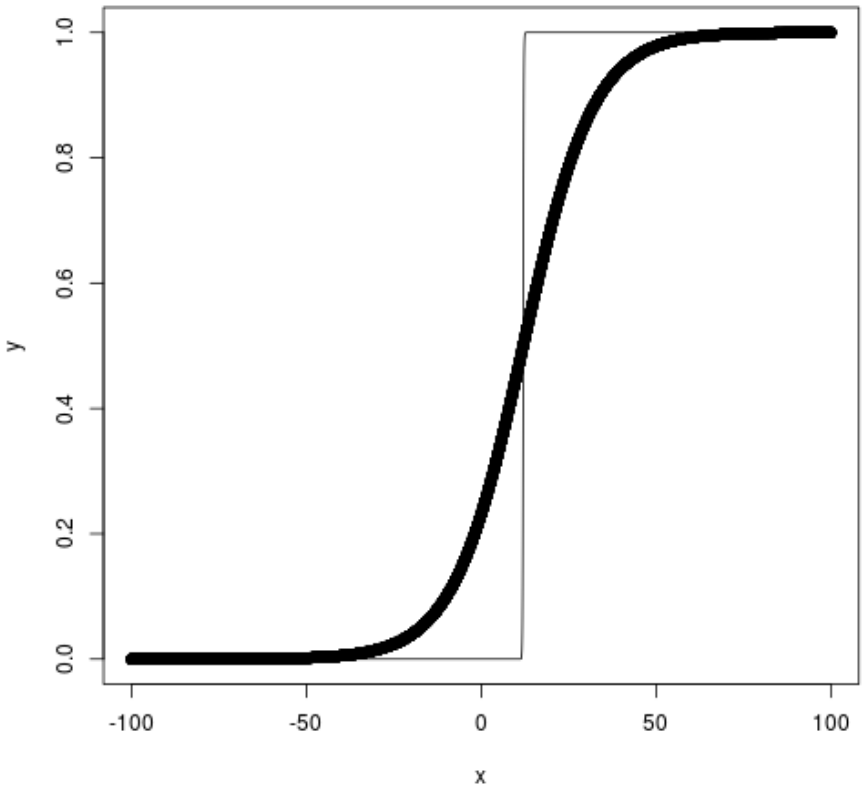
The function 1 can be graphed as:

```
#TBA: need to add plot labels for k
x = seq(-100, 100, 0.01)
k = 0.1
y = 1/(1+exp(-k*(x-12)))
k = 12
y2 = 1/(1+exp(-k*(x-12)))
plot(x,y)
lines(x, y2)
```

The logistic function (shown as Equation 1 and as two curves in Figure **??**), as we will see, is very important in many classification and decision settings precisely because of its *non*linear_ shape and its constrained movement within a limited range of values (0 to 1). How does a function such as the sigmoid function do anything? Here the curve itself and even the name 'sigmoid' is the best guide. The S-shape of the sigmoid curve guides the operation of many of the techniques and is a good example of the character of the operations associated with curves. The logistic function has quite a long history in statistics (Stigler 1986), since that curve suggests growth and change in various ways. As the historian of statistics J.S. Cramer writes:

> The logistic function was invented in the nineteenth century for the description of the growth of organisms and populations and for the course of autocatalytic chemical reactions (Cramer 2004, 614).

The Belgian mathematician Pierre-François Verhulst designated the sigmoid function the 'logistic curve' in the 1830-40s (Cramer 2004, 616). It was independently designated the 'autocatalytic function' by the German chemist Wilhelm


y
0.0
0.2
0.4
0.6
0.8
1.0
-100
-50
0
50
100
x

Ostwald in the 1880s, and then re-invented under various names by biologists, physiologists and demographers during 1900-1930s (617). In nearly all of these cases, the function was used to fit a curve to data on the growth of something: populations, tumours, tadpoles tails, oats and embryos. The term 'logistic' returns to visibility in the 1920s, and has continued in use as a way of describing the growth of something that reaches a limit. The reference of the curve to growth comes from its changing slope. Growth starts slowly, increases rapidly and then slows down again. This pattern can be seen in organisms, in chemical reactions and in populations. Hence the logistic function has a well-established biopolitical resonance. In the second half of the twentieth century, it was widely used in economics. In all these settings and usages, the curve was a way of summarising and predicting growth. Census data, clinical or laboratory measurements supplied the actual values of $f(x)$ at particular times, the x values. The task of the demographer, physiologist or economist was to work out the values of parameters such as k that controlled the shape of the curve.

Note that the curves showing in Figure **??** plot the same data ($\mathbf{X}$ and y values), but differ in their profile. This graphical variation derives from the parameter k, which discreetly appears in the equation 1 next to x. Such parameters are vital control points. Varying these parameters and optimising their values is the basis of 'useful approximation' in machine learning. Sometimes these parameters can be varied so much as to suggest entirely different functions. In **??** for instance, $k = 12$ produces a much sharper curve, a curve that actually looks more like a qualitative change, range than a smooth transition from 0 to 1. The sharp shape of the logistic function when the scaling parameter k is larger suggests another important transformation, somewhat orthogonal to the description of rates of growth under limits. The mathematical function $f(x) = 1/(1 + e^{-x})$ can be treated as a way of mapping continuously varying numbers (the x values)

and discrete values. Because $f(x)$ tends very quickly to converge on values of 1 or 0, it can be coded as 'yes'/'no'; 'survived/deceased', or any other binary difference. The transformation between the x values sliding continuously and the binary difference pivots on the combination of the exponential function (e^{-x}), which rapidly tends towards zero as x increases and rapidly tends towards inf as x decreases, and the $1/(1 + ...)$, which converts high value denominators to almost zero, and low value demominators to one. This constrained path between variations in x and their mapping to the value of the function $f(x)$ is mathematically elementary, but typical of the relaying of references that allows functions to intersect with and constitute matters of fact and states of affairs. This realisation – that a continuously varying sigmoid function could also map discrete outcomes – forms the basis of many machine learning classifiers. So, a contemporary biostatistical machine learning textbook can write: we can use logistic regression to 'estimate the probability that a critically-ill lupus patient will not survive the first 72 hours of an initial emergency hospital visit' (J. D. Malley, Malley, and Pajevic 2011, 5). If the same curve – the logistic curve – can describe quite different situations (the growth of a population, the probability that someone will die), then we can begin to see that sigmoid functions, and the logistic curve in particular, might be useful devices as approximations to the 'underlying function that generated the data.'

The curve as classifier

How does this take place practically? As I have already mentioned, the logistic function appears frequently in machine learning literature, prominently as part of perhaps the most classical learning machine, the logistic regression model, but also as a component in other techniques such as neural networks. Descriptions of logistic regression models appear in nearly all machine learning tutorials,

textbooks and training courses. Logistic regression models are heavily used in biomedical research, where, as 'logistic regression is the default "simple" model for predicting a subject's group status' (J. D. Malley, Malley, and Pajevic 2011, 43). As Malley et.al. suggest, 'it can be applied after a more complex learning machine has done the heavy lifting of identifying an important set of predictors given a very large list of candidate predictors' (43). Especially in comparison to more complicated models, logistic regression models are relatively easy to interpret because they are based on the linear model that we have been discussing already. As Hastie et.al write: 'the logistic regression model arises from the desire to model the posterior probabilities of the K classes via linear functions in x, while at the same time ensuring that they sum to one and remain in $[0, 1]$' (Hastie, Tibshirani, and Friedman 2009, 119). Paraphrased somewhat loosely, this says that the logistic regression model predicts what class or category a particular instance is likely to belong to, but 'via linear functions in x.' We see something of this predictive desire from the basic mathematical expression for logistic regression in a situation where there are binary responses or $K = 2$:

$$Pr(G = K|X = x) = \frac{1}{1 + \sum_{l=1}^{K-1} exp(\beta_{l0} + \beta_l^T x)} \tag{2}$$

(Hastie, Tibshirani, and Friedman 2009, 119)

In equation 2, the logistic function now encapsulates lines in curves. That is, the linear model (the model that fits a line to a scattering of points in vector space – see previous chapter) appears as $\beta_l 0 + \beta_l^T x$ (where as usual β refers to the parameters of the model and x to the matrix of input values). The linear model has, however, now been put inside the sigmoid function so that its output values no longer increase and decrease linearly. Instead they follow the sigmoid curve of the logistic function, and range between a minimum of 0 and a maximum

of 1. As usual, small typographic conventions express some of this shift. In equation 2, some new characters appears: G and K. Previously, the response variable, the variable the model is trying to predict, appeared as Y. Y refers to a continuous value whereas G refers to membership of a category or class (e.g. survival vs. death; male vs female; etc.).

What does this wrapping of the linear model in the curve of the sigmoid logistic curve do in terms of finding a function? Note that the shape of this curve has no intrinsic connection or origin in the data. Although it comes from elsewhere, this curvilinear variation on the linear model allows the left hand side of the expression to move into a different register. The left hand side of the expression is now a probability function, and defines the probability (Pr) that a given response value (G) belongs to one of the pre-defined classes ($k = 1, ..., K-1$)[3]. In this case, there are two classes ('yes/no'), so $K = 2$. Unlike linear models, that predict continuous y values for a given set of x inputs, the logistic regression model produces a probability that the instance represented by a given set of x values belongs to a particular class. When logistic regression is used for classification, values great than 0.5 are usually read as class predictions of 'yes', 'true' or `1`. As a result, drawing lines through the common vector space can effectively become a way of classifying things. Note that this increase in flexibility comes at the cost of a loss of direct connection between the data or features in the generalized vector space, and the output, response or predicted variables. They are now connected by a mapping that passes through the somewhat more mobile and dynamic operations of exponentiation exp and with some new difficulties in estimating the all important weighting parameters β in equation 2.

[3] I leave aside any further discussion of probability in machine learning here. It is the topic of the next chapter.

The cost of curves in machine learning

If the logistic function is a useful approximation to the process that generated the data, we should bear in mind that its use combines an attention to the common vector space, and the difficulties of drawing straight lines through that vector space. Let us investigate the cost of this flexibility that bends lines through volumes of data by continuing with logistic regression. Other techniques – neural networks or support vector machines – offer different kinds of flexibility, but they share some common intuitions with logistic regression in the way that they fit functions to data. The way in which we have 'learned' the logistic function by taking a textbook formula expression of it, and plotting the function associated with it is not the way that machine learners typically 'learns' the function that maps between the input data and the output variables (the so-called 'response variable'). Finding a function in practice means optimising parameters on the basis of the data. This is not a matter of mathematical analysis, but of algorithmic repetition.

As soon as we move from the more theoretical or expository accounts of function-finding into the domain of practice, instruction and learning of machine learning, a second sense of function comes to the fore. The second sense of function comes from programming and computer science. A function there is a part of the code of a program that performs some operation, 'a self-contained unit of code,' as Derek Robinson puts it (**???**). The three lines of R code written to produce the plot of the logistic function are almost too trivial to implement as a function in this sense, but they show something of the transformations that occur when mathematical functions are operationalised in algorithmic form. The function is wrapped in a set of references. First, the domain of x values is made much more specific. The formulaic expression $f(x) = 1/(1 + e^{-x})$ says nothing explicitly about the x values. They are implicitly real numbers (that is, $x \in \mathbb{R}$) in this

formula but in the algorithmic expression of the function they become a sequence of 20001 generated by the code. Second, the function itself is flattened into a single line of characters in code, whereas the typographically the mathematical formula had spanned 2-3 lines. Third, a key component of the function $e^{-}x$ itself refers to Euler's number e, which is perhaps the number most widely used in contemporary sciences due to its connection to patterns of growth and decay (as in the exponential function e^x where $e = 2.718282$ approximately). This number, because it is 'irrational,' has to be computed approximately in any algorithmic implementation.

If we turn just to the diagrammatic forms associated with logistic regression in *Elements of Statistical Learning*, something quite different and much more complicated than calculating the values of a known function is going on. For instance, in their analysis of the South African coronary heart disease data[4], Hastie and co-authors repeatedly model the risk of heart disease using logistic regression. They first apply logistic regression fitted by 'maximum likelihood', and then by 'L1 regularized logistic regression' (Hastie, Tibshirani, and Friedman 2009, 126). The results of this function-finding work appear as tables of coefficients or as 'regularization plots.'

```
library(ElemStatLearn)
library(lars)
```

```
## Loaded lars 1.2
```

```
data(SAheart)
```

```
sah = as.matrix(SAheart[,c('sbp', 'ldl', 'adiposity', 'typea', 'obesity', 'alcohol', 'ag
```

[4] I discussed the use of this dataset in Chapter 1. It is a typical biomedical data set.

```
SAheart_model = lars(y=SAheart$chd, x = sah)
plot(SAheart_model)
```

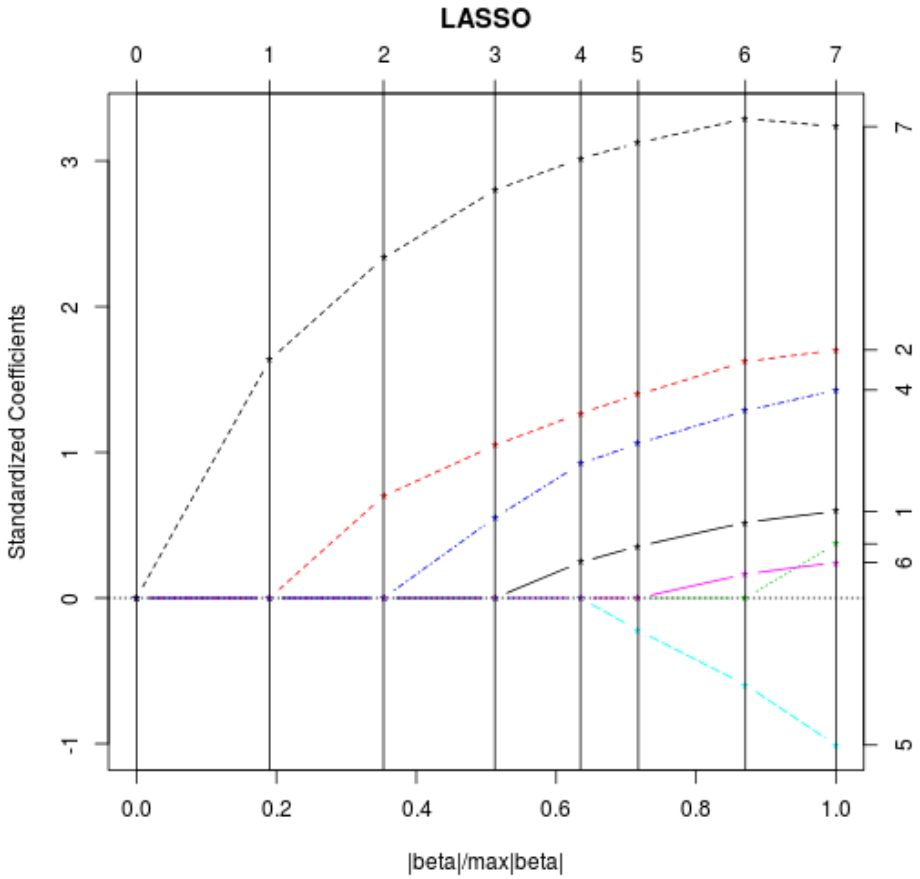


Figure 2: plot of chunk heart_model

```
tab1 = summary(SAheart_model)
print(tab1)

## LARS/LASSO
## Call: lars(x = sah, y = SAheart$chd)
##   Df     Rss      Cp
```

```
## 0  1 104.589 99.2762
## 1  2  94.796 48.9096
## 2  3  89.978 25.1483
## 3  4  87.233 12.4659
## 4  5  86.028  8.0255
## 5  6  85.537  7.3968
## 6  7  85.018  6.6222
## 7  8  84.901  8.0000
```

The function finding that goes on in these different techniques does not involve finding radically different curves that might fit the data. As we saw in the previous chapter, the production of new tables of numbers that list the parameters of models that seek to combine or superimpose different dimensions of the data is a core operation of machine learning. Many of the plots and tables found in machine learning texts, practice and code offer nothing else but measurements of how the model weights or values different dimensions of the vector space. In the case of logistic regression, the shape of the curve is determined using maximum likelihood. For our purposes, the statistical significance of this procedure is less important than the algorithmic implementation. This is the opposite to what might appear in a typical statistics textbook where the implementation of maximum likelihood would normally be quickly passed over.[5]

Learning as optimising

This is not the way that machine learners typically learn functions. Much machine learning practice takes the form of re-shaping the parameters of the func-

[5]For instance, in *An Introduction to Statistical Learning with R*, a textbook focused on using `R` to implement machine learning techniques, the authors write: 'we do not need to concern ourselves with the details of the maximum likelihood fitting procedure' (James et al. 2013, 133).

tion within a set of constraints or limits. We saw in the case of the linear regression model that the classic statistical model of linear regression fits lines to the data through the method of ordinary least squares (see Chapter 2, equation **??**). In mathematical practice more generally, problems are posed in terms of finding solutions to functions. Mathematics textbooks are replete with demonstrations of this problem-solving activity, and learning mathematics is in large part becoming practiced in solving problems by finding either values or functions that solve problems. Even in machine learning, some function-finding through solving systems of equations occurs. For instance, the closed form solution of the least sum of squares problem for linear regression is given by $\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$. As we saw in the previous chapter, this expression provides a very quick way to calculate the parameters of a linear model given a matrix of input and output values. This formula itself is derived by solving a set of equations for the values $\hat{\beta}$, the estimated parameters of the model.

As we saw earlier, in his formulation of the 'learning problem', Vladimir Vapnik speaks of choosing a function that approximates to the data, yet minimises the 'probability of error' (Vapnik 1999, 31). Unique 'closed form' solutions are quite unusual in machine learning. In machine learning, functions are normally learned through a process of approximation and optimisation that has little resemblance to the deductive solving of equations. Even the apparently simplest data modelling procedure of fitting a line to a set of points is usually implemented differently in machine learning settings. This is a point where machine learning differs substantially from conventional statistical techniques. The use of approximation, optimisation and heuristic learning approaches is much more prevalent for reasons that have to do with the situations in which machine learning is put work. For instance, describing the application of machine learning to biomedical and clinical research, James Malley, Karen Malley and Sinisa Pajevic

contrast it to more conventional statistical approaches:

> working with statistical learning machines can push us to think about novel structures and functions in our data. This awareness is often counterintuitive, and familiar methods such as simple correlations, or slightly more evolved partial correlations, are often not sufficient to pin down these deeper connections. (J. D. Malley, Malley, and Pajevic 2011, 5–6)

The novel structures and functions in 'our data' are precisely the functions that machine learning technique seek to learn. As we will see, these structures and functions take various forms and shapes (lines, trees, curves, peaks, valleys, forests, boundaries, neighbourhoods, etc.), and they can identify 'deeper connections' than the correlations we have saw in the house price or iris datasets.

But how do we know whether a model is a good one, or that the function that a model proffers to us fits the functions in our data. (If it is not already obvious, in the world of machine learning it is simply taken as read that functions are in the world; 'Nature' has been thoroughly mathematised here.) One problem with closed-form or analytical solutions typical of mathematical problem-solving is precisely their closed-form. To continue with key example of the closed form solution for linear regression

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-}1\mathbf{X}^T\mathbf{y} \tag{3}$$

We see that it estimates the parameters of the linear model by carrying out a series of operations on matrices of the data. These operations include matrix transpose, several matrix multiplications (so-called 'inner product') and matrix inversion (the process of finding a matrix that when multiplied by the input

matrix yields the identity matrix, a matrix with 1 along the diagonal, and 0 for all other values). When the dataset has a hundred or a thousand rows, these operations can be implemented and executed easily. But as soon as datasets become much larger, it is not easy to actually carry out these matrix operations even on fast computers. For instance, a dataset with a million rows and several dozen columns is hardly unusual today. Although linear algebra libraries are carefully crafted and tested for speed and efficiency, there is no way that they can quickly carry out matrix multiplication (inner products) on million row datasets in a reasonable time. The closed form solution, even for the simplest possible structures in the data, begins to break down in this situation. If, for instance, instead of working with the San Francisco house price dataset used in the previous chapter, we used much older Boston house price dataset available from the University of California Irvine machine learning data repository (http://archive.ics.uci.edu/ml/datasets/Housing), and tried to calculate the parameters of the model using the closed form solution, problems of scale arise immediately. With only 500 rows of data, and 12 variables, calculation of the parameters for a linear model to predict house prices is still workable, but with 5000 rows, my laptop starts to struggle for minutes at a time. With hundreds of variables and millions of rows, the closed form solution becomes increasingly unworkeable.

```
import sklearn.datasets
import numpy as np
boston = sklearn.datasets.load_boston()
boston.data.shape
x = boston.data[:,0:10]
y = boston.data[:,11]
```

```
beta_hat = np.linalg.pinv(x.dot(x.transpose())).dot(x.transpose().dot(y))

Traceback (most recent call last):
  File "<string>", line 2, in <module>
ImportError: No module named sklearn.datasets
```

There is another problem with the closed form approximation. The closed form solution is run once, and the model it produces is subject to no further computation, elaboration or variation. That is, the closed form solution based on the so-called 'normal equations' delivers a fixed solution. The parameters $\hat{beta}$ define the line of best fit for the datasets on which they are based. A more typical machine learning approach is to find a way of modelling how well the model has dealt with the data. It replaces the exactitude and precision of mathematically-deduced closed-form solutions with algorithms that search for a good solution, not 'the' solution. The combination of all the variables can be imagined as a surface whose contours include peaks and valleys. As we have seen, a linear model tries to find a line or plane or hyperplane (a higher dimensional plane) that fits this topography. Many different planes more or less fit the contours, but how do we choose the best one? If we can't produce an exact answer to this problem, we could spend time trying out different parameters, varying some, and keeping the others the same until we find a set of parameters that seems to fit well.This is a classic machine learning scenario, where the machine is meant to learn something that programmers, engineers, scientists or statisticians cannot. How would the machine learning approach the line/plane of best fit?

In many machine learning techniques , and especially in the techniques of 'supervised learning', the search for an approximation to the function that generated the data is guided by another function called the 'cost function' (also known as the 'objective function' or the 'loss function'; both terms are somewhat evoca-

tive). There are various ways of learning functions, but cost functions are an essential component of many machine learning models. Deciding on the cost function means thinking about how the predictions relate to the values in the data set. Every cost function implicitly has some measure of the difference or distance between the prediction and the values actually measured. In supervised learning, cost functions work with the known values. Cost functions stage 'the act of fitting a model to data as an optimization problem' (Conway and White 2012, 183). Having defined a cost function, the learning algorithm can fit many models to the data, and use the cost function to decide which fits best. The cost functions themselves only provide a way of testing whether a given model performs better than another model in terms of changes in the parameters. So, the kind of model or type of prediction it performs does not change radically. If there is learning here, it is not some enigmatic form or a higher form of scientific reason. Just the opposite, the cost function does something more like create a place from which variations in models can be viewed.

A typical and widely cost function associated with regression is defined as:

$$J(\beta) := min_(\beta)\frac{1}{m}\sum^{m}(h_\beta(x^{(i)} - y^{(i)}))^2$$

Again, the reading this kind of formula expression of a function is not easy. But several key terms stand out. First, the cost function $J(\beta)$ is a function of all the parameters of the model. Second, the function is defined in terms of the goal of minimizing the overall value of the expression. The *min* describes the results of the repeated application of the function. Third, the heart of the function is a kind of average: it adds ($\sum$) all the differences between the values of y predicted by the model and the known values of y, and divides them by the number of values of y ($\frac{1}{m}$). This is the so-called 'mean squared error' measure

of prediction, a core measure of prediction in machine learning, and the basis of many standard statistical procedures. Mean squared error (MSE) is so taken for granted that machine learning textbooks such as *The Elements of Statistical Learning* (Hastie, Tibshirani, and Friedman 2009) often rely on it without any further explanation.

Importantly, the cost function itself does not say anything about how the values of the parameters are to be found. They could be generated randomly, or perhaps could be just a range of values (e.g. 0 to 100).

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data
boston_house =read.delim(url, sep='', header=FALSE)
X = as.matrix(boston_house[, 1:13])
Y = as.matrix(boston_house[,14])

X_norm =cbind(X[,1], apply(X[,2:3], MARGIN=2, function(X) {(X-mean(X))/sd(X)}))
Y_norm = (Y - mean(Y))/sd(Y)

X_means = colMeans(X)
X_sd = apply(X=X,MARGIN=2,FUN=sd )

Y_mean = mean(Y)
Y_sd = sd(Y)

beta = matrix(0, nrow = ncol(X_norm))

computeCost <- function(X,Y, beta) {
    m = nrow(X)
    h = X %*% beta
```

```
    cost = 0.5 /m * sum( (h-Y)^2)
    return(cost)
}
```

In the code fragment above, the R function `computeCost` takes as inputs the matrix of X , the vector of Y values (the outputs), and a vector of current values of β, the model parameters. It predicts the Y values by multiplying X by $/beta$ (this is a linear model in which predictions are generated by multiplying variables by their respective parameters and then adding the result). In the cost line of code, it calculates the differences between all the predictions and known actual values, adds them, and averages them to yield the 'cost' of these particular values of the parameters. In practice, the cost function will be calculated for each change in the model's parameters, and the calculated cost or loss will be checked to see if is lower or higher than previous values. If lower, then the new model parameters are producing better predictions than previous values of the parameters. This is good because it means that the approximation to the function that generated the data is better. Measures of error, or closeness and distance are crucial to machine learning. If we can learn functions non-deductively and if functions can learn functions, it is partly by virtue of the kinds of view on variation set up in the cost functions.

[HERE] - want to go to say that the process of fitting the curve to the data is sometimes stated just straight out, as it is obvious. But in many ways not. Various techniques try to deal with this as a problem of optimisation. Means we need to look more at the function from a different more operational angle – the function in code.

Gradient descent and the search for a partial observer

'Science brings to light partial observers in relation to functions within systems of reference' wrote Gilles Deleuze and Feliz Guattari in their account of scientific functions (Deleuze and Guattari 1994, 129). I'm not sure whether Deleuze and Guattari were aware of the extensive work done on problems of mathematical optimization during the 1950-1960s, but their strong interest in the mathematics of differential calculus somewhat unexpectedly makes their account of functions highly relevant to machine learning. Many of the techniques of optimisation underlying machine learning techniques rely on differential calculus.

How then are a stream values of the parameters generated for the cost function to be minimized? This is a key problem since lowering the MSE or any cost/loss/risk function implies a way to find better values. If the values of the model parameters were generated randomly, there would be little guarantee that any model was going to be better than the last one. Perhaps after trying enough random values, a good fit might appear. Or perhaps a good model might never appear. A more ordered approach might be to try a 'grid search' (Conway and White 2012, 185), in which all the possible combinations of values for the model parameters are tested successively, just like a search party systematically combing some terrain for a missing person. But with more than a few parameters, the matrix of values quickly becomes vast (100 different values for 10 parameters means calculating and comparing 100^10 or 10^20 values). It is impossible to move through large fields of values without missing some important features. Both random and grid search are likely to fail because they do not actually take into account any structure in the data. If implemented, they might minimize the cost function or they might not. Hence, a third decision has to be made in any machine learning setting. Having chosen a type of model and a cost function, it is also necessary to choose an optimisation method.

One widely used optimisation algorithm called 'gradient descent' is quite easy to grasp intuitively and it illustrates the process of searching for an approximation to the function that produced the data. As in many formulations of machine learning techniques, the framing of the problem is finding the parameters of the model/function that best approximates to the function that generated the data. It calculates the parameters of a model in short. The algorithm can be written using calculus style notation as:

Repeat until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_j)$$

This version of the algorithm is called 'batch gradient descent.' As always, in presenting these mathematical formula, I don't expect readers to read and understand them directly. Actually reading these formal expressions, and being able to follow the chain of references, and indexical signs that lead away from them in various directions is quite challenging. Many people who directly use machine learning techniques in industry and science would not often if ever need to make use of such expressions as they build models. They would mostly take them for granted, and simply execute them. My purpose here, however, is a bit different. Rather than explaining these formulations, my interest is following the threads and systems of reference that wind through them, and to identify the points of transition, friction or slippage that both allow them to work and also not be everything they claim to be.

Given that this expression encapsulates the heart of a major optimisation technique, we might first of all be struck by its brevity. This is not an elaborate or convoluted algorithm. As Malley, Mally and Pajevic observe, 'most of the [machine learning] procedures … are (often) nearly trivial to implement' (J. D.

Malley, Malley, and Pajevic 2011, 6). Below, I provide an implementation in a few lines of R code. Note that this expression of the algorithm, taken from the class notes for Lecture 4 of Andrew Ng's 'Machine Learning' course on Coursera, mixes an algorithmic set of operations with function notation. We see this in several respects: the formulation includes the imperative 'repeat until convergence'; it also uses the so-called 'assignment operator' := rather than the equality operator =. The latter specifies that two values or expressions are equal, whereas the former specifies that the values on the right hand side of the expression should be assigned to the left. Both algorithmic forms – repeat until convergence, and assign/update values – owe more to techniques of computation than to mathematical abstraction. In this respect more generally, by looking at implementations we begin to see how systems of references are put together. In this case, the specification for the gradient descent algorithm starts to bring us to the scene where data is actually reshaped according by the more abstract mathematical functions we have been describing (mainly using the example of the linear regression and its classic algebraic form $y = \theta_0 + \theta_1 x_1 + ...\theta_n x_n$).

At the heart of this reshaping lies a different mathematical formalism: the partial derivative, $\frac{\partial}{\partial \theta_j} J(\theta_j)$. Like all derivatives in calculus, this expression can be interpreted as a rate; that is as the rate at which the cost function $J(\theta)$ changes with respect to the values of θ.[6] If there is any learning in machine learning, it will take something like this form. But how could a partial derivative learn? Deleuze and Guattari again are useful on this point. They write that 'science brings to light partial observers in relation to functions within systems of reference' (Deleuze and Guattari 1994,129). Again, it is not clear that they are referring to partial derivatives in particular, but the partial derivatives in gradient descent comprehensively demonstrate what they describe. The partial

[6] The derivative $\frac{\partial}{\partial \theta_j} J(\theta_j)$ is *partial* because θ is a vector $\theta_0, \theta_1 ... \theta_j$.

derivative in the gradient descent algorithm is a kind of observer moving through a function (the best approximation to the function that generated the data) within a system of reference. What is the partial derivative partially observing? Remember that cost function $J(\theta)$ was defined as the average of the differences between predicted values and actual values squared ('mean squared error'). This function then itself has a particular curved form – a parabola – and this curve has a minimum value at the bottom (see the figure).

```
x = seq(-100,100,0.5)
y = x*x
plot(x,y)
```

As always, the diagram or plot shows one, two or at most three dimensions of a curve that may in actual cases have hundreds or thousands of dimensions. We have to imagine that this curve is a hyperdimensional surface, but even in many dimensions, it is still parabolic and has a minimum value at the bottom of the dish. An observer that can manage to traverse this curve or hypercurve carries with it the possibility of minimising the cost function $J(\theta)$ and thereby optimising the approximation of a function that fits the data.

How to move around on a surface in a way that tends to take you towards lower ground and into the valley? This problem hardly presents itself as a problem to us as we walk over hill and dale. But then maybe it would be more of a problem if we didn't have paths to follow and we were walking on a moonless night. Indeed the problem is perhaps more like crossing sand dunes at night than hiking in the mountains. The only sense of the way down is a feeling underfoot of which way the ground is sloping. This is the intuition that gradient descent abstractly follows in the data. On the one hand, the datasets in which gradient descent moves have no paths or roads in them. The algorithm has no gods' eye viewpoint

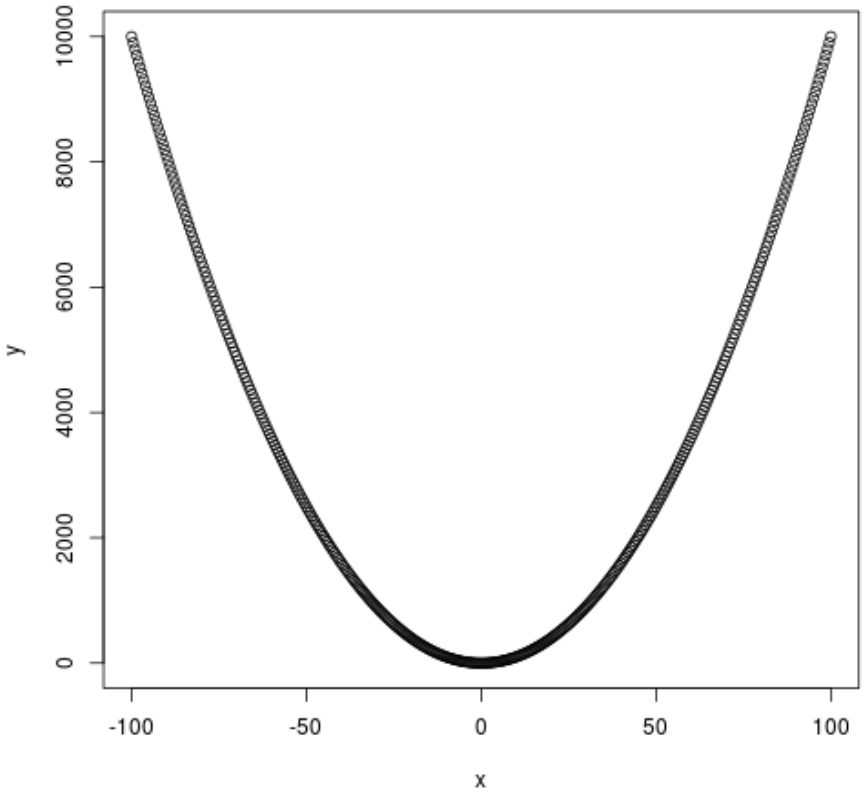


Figure 3: plot of chunk cost_function

on the data, even though it can move up and down the matrices as many times as it wants. Datasets are in some respects highly organised spaces in that the data has been cut into dimensions, and grids by the rows and columns, but these dimensions have no particular value in themselves. We don't know which dimension is more important. On the contrary, the whole point of modelling the dataset is to give us some sense of what dimensions matter most in the states of affairs they pertain to. But on the other hand, the parabolic curve/surface embodied in the cost function $J(\theta)$ already shapes the space traversed by the algorithm. A parabola, as mentioned above, is a kind of bowl (although this bowl might be in many dimensions). It is much easier to reach the bottom of a bowl, even a hyper-dimensional bowl, than to reach the bottom of a valley in hilly terrain. That is, there is little chance of getting caught in some hollow halfway up the hillside.

```
gradientDescentMulti <- function(X,Y, theta, alpha, iterations) {

    m = nrow(X)
    par(mfrow = c(1,2))
    theta = as.matrix(theta, ncol=1)
    theta_temp = c()
    J = c()
    theta_all = matrix(nrow=iterations, ncol=length(theta))

    for (i in 1:iterations) {
        # compute theta
        h = X%*%theta
        theta_temp = theta - alpha/m *sum ( apply(X, MARGIN=2, FUN = function(x) {x*
```

```
        # update theta
        theta = theta_temp
        cat('theta current: ', theta, '\n')
        cost = computeCost(X,Y, theta)
        cat('cost: ',cost, '\n')
        J[i] = cost
        theta_all[i,]= theta
    }
    return(list(theta=theta_all,J=J))
}
```

In the code snippet above, the implementation of gradient descent is quite brief. It occurs in one line: `theta_temp = theta - alpha/m *sum ( apply(X, MARGIN=2, FUN = function(x) {x*(h-Y)}))`. The brevity of this code again demonstrates the simplicity of the partial observer in many machine learning functions. refers to the direction and steepness of values on this slope. The only structure that gradient descent takes into account is the value of the cost function, $J(\theta)$. It explores this surface by always trying to move downwards and given that the cost function is a parabolic curve, it is hard to move in the wrong direction. But the algorithm can move too quickly. It can overshoot the minimum value – the base of the bowl - and start climbing the other side, heading for even higher ground. The value of the key parameter of the gradient descent algorithm itself α matters here. If α is too high, the algorithm might stride over the minimum value of $J(\theta)$. If α is too low, the algorithm might crawl too slowly down the gradient, and still be somewhere on the side of the parabolic curve before the algorithm ends. When does the algorithm end? In the simple implementation shown here, the algorithm ends after a certain number of iterations, typically several thousand times around the gradient descent step.

Gradient descent is a low-level algorithm in the sense that it does not predict or classify anything directly. It only minimises the value of a function, and does so according to a relatively simple strategy of always stepping in the most steeply sloping direction in the expectation of reaching the lowest point sooner or later. The effect of this search is to gradually adjust values of the linear model $\hat{\beta}$ so the differences between the actual values and the predicted values is reduced. Effectively, gradient descent starts by drawing a line/plane at random through the points, and then changes some of the parameters controlling slope and position of the line in order to reduce the difference between the predicted values and the actual values. But everything here depends on the cost function, and its regular geometry. Without that, gradient descent would very likely be wandering the dunes for ever.

[[want to get to the point where I can say the power of the machine learning to learn, and hence appetite for data, for infrastructural transformations, etc, can all be understood in terms of this form of movement that pulls lines around the curve – a kind of constraint; this is an antidote to the mythologised power of algorithms]

This description of the differences between functions in a mathematical sense as a mapping and functions in an algorithmic sense as an implementation of some repeated operations that might express a mathematical function is meant to highlight a key issue in learning functions. As we move from the mathematical formula to the three lines of R code that produces a plot of the function what has happened? This is a kind of implementation of a function, and perhaps we learn something about the logistic function, that for instance, that the y values change decisively between 0 and 1 across a very brief interval of x values. Unlike the linear functions we saw in the house-price models (see previous chapter), logistic functions approximate a switch between 1 and 0, or other values such as `yes`

and `no`. This rapid change in value will, as we see, proves incredibly useful in machine learning: it opens up the possibility of using continuous-value function to approximate states of affairs where differences are much more heavily marked. That is, the logistic function can be used to classify or decide.

There is deep disturbance in the practice of machine learning around the problem of knowing whether it works or not. While the field is almost despotically pragmatic in its concerns with classification and prediction (although in certain ways, curiously idealistic too), it is troubled by the persistence of two broadly different kinds of *learning*: supervised and unsupervised (as well as hybrid kinds such as semi-supervised learning). Writing around 2000, Hastie et. al. state:

> With supervised learning there is a clear measure of success or lack thereof, that can be used to judge adequacy in particular situations and to compare the effectiveness of different methods over various situations. Lack of success is directly measured by expected loss over the joint distribution $Pr(X, Y)$. This can be estimated in a variety of ways including cross-validation. In the context of unsupervised learning, there is no such direct measure of success. … This uncomfortable situation has led to heavy proliferation of proposed methods, since effectiveness is a matter of opinion and cannot be verified directly. (Hastie, Tibshirani, and Friedman 2009, 486–7)

Curves lie at

Alpaydin, E. 2010. *Introduction to Machine Learning.* Cambridge, Massachusetts; London: The MIT Press.

Barber, David. 2011. *Bayesian Reasoning and Machine Learning.* Cambridge; New York: Cambridge University Press.

Breiman, Leo, Jerome Friedman, Richard Olshen, Charles Stone, D. Steinberg, and P. Colla. 1984. "CART: Classification and Regression Trees." *Wadsworth: Belmont, CA* 156.

Conway, Drew, and John Myles White. 2012. *Machine Learning for Hackers*. Sebastopol, CA: O'Reilly. http://search.ebscohost.com/login.aspx?direct=true/&scope=site/&db=nlebk/&db=nlabk/&AN=436647.

Cramer, J. S. 2004. "The Early Origins of the Logit Model." *Studies in History and Philosophy of Science Part C: Studies in History and Philosophy of Biological and Biomedical Sciences* 35 (4): 613–626.

Deleuze, Gilles, and Félix Guattari. 1994. *What Is Philosophy?* European Perspectives. New York ; Chichester: Columbia University Press.

Hastie, Trevor, Robert Tibshirani, and Jerome H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer.

Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. 2006. "Reducing the Dimensionality of Data with Neural Networks." *Science* 313 (5786): 504–507. http://www.sciencemag.org/content/313/5786/504.short.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Springer. http://link.springer.com/content/pdf/10.1007/978-1-4614-7138-7.pdf.

Kirk, Matthew. 2014. *Thoughtful Machine Learning: A Test-Driven Approach*. 1 edition. Sebastopol, Calif.: O'Reilly Media.

Malley, James D., Karen G. Malley, and Sinisa Pajevic. 2011. *Statistical Learning for Biomedical Data*. 1st ed. Cambridge University Press.

Mohamed, Abdel-rahman, Tara N. Sainath, George Dahl, Bhuvana

Ramabhadran, Geoffrey E. Hinton, and Michael A. Picheny. 2011. "Deep Belief Networks Using Discriminative Features for Phone Recognition." In 5060–5063. IEEE. doi:10.1109/ICASSP.2011.5947494. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5947494.

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. 2011. "Scikit-Learn: Machine Learning in Python." *The Journal of Machine Learning Research* 12: 2825–2830. http://dl.acm.org/citation.cfm?id=2078195.

Stigler, Stephen M. 1986. *The History of Statistics: the Measurement of Uncertainty Before 1900*. Cambridge, Mass.: Harvard University Press.

Vapnik, Vladimir. 1999. *The Nature of Statistical Learning Theory*. 2nd ed. 2000. Springer.