COMP4702/7703 - Machine Learning

Lecture Notes

Marcus Gallagher

Module 1: Introduction

Chapter 1 of our text gives an easy-to-read introduction and overall picture of what machine learning is all about. We have a few important concepts and terminology, but the main thing is to discuss some practical examples of tasks that machine learning is well-suited to and has been successfully applied.

Lindholm et al. begin by explaining machine learning in terms of data analysis, where this analysis is (at least partly) automated. As a result, we end up with a computer program that was (at least partly) learnt from a set of data. Generally speaking, we have some kind of mathematical model inside this program. Given some data, the parameters of the model are adjusted such that the model captures (to a good extent) the stucture of the data. This model can then be used to understand the variables present in the data, make predictions related to those variables and more.

Important terms:

- **Training data**
- **Mathematical model**
- **Learning algorithm**
- **Parameters**

Example 1.1: Automatically diagnosing heart abnormalities

This is a somewhat complicated example to start with. The data is ECG recordings of the electrical activity of a person's heart. This data is a signal: it is collected over some time period, sampled at some frequency. In this case, 12 electrodes are used, so we actually have multiple time series data! Nevertheless, the idea is that this data can provide information that is useful for diagnonsing the health of a person's heart. If you have a heart problem, the ECG will look different compared to a normal healthy heart. In the paper they mention, machine learning was used to classify/categorise ECG recordings and was found to perform better or comparably to "human experts".

- **Supervised learning**: the model was trained using training data where experts had labelled examples of each class of ECG signal.
- **Classification**: the output of the model is one of a fixed number of categories.

Example 1.2: Formation energy of crystals

Materials science investigates the development of new possible materials by studying the structural properties of hypothetical new materials. Something they are interested in specifically is calculating the formation energy of crystals. There is a "classical" way of calculating this, but it is computationally very expensive. If machine learning models could be trained (using existing/known data) to predict the energy of new crystals, many more materials could be investigated.

- This is a **regression** problem: the target variable (that we want to predict) is a numerical value rather than a category.
- The "correct outputs" for the training set come from an (expensive) simulation rather than a human expert.

Lindholm et al. make a very important point here that we will revist during the course. Machine Learning is often about predicting unknown values based on observed data for other values. Therefore, **uncertainty** is a fundamental part of machine learning. We often want to be able to quantify this uncertainty rather than just spitting out a prediction for an input. **This is why statistics and probability are so important in machine learning**.

Example 1.3: Probability of scoring a goal in soccer

Try and build a model to predict whether or not a shot results in a goal, based on the angle of the player relative to the goal posts. Clearly we cannot predict the outcome for certain, but historical data can be used to build a model that predicts the probability that the goal will be scored.

Example 1.4: Pixel-wise class prediction

This is an example where the output is more complex than a simple classification or regression problem: classify every pixel in an input image. This is also known as (a type of) image segmentation problem.

Example 1.5: Estimating air pollution levels across London

Predict NO_2 levels over an area (latitude, longitude) based on spatial and time-varying data.

Module 2: Supervised Learning

Chapter 2 of the text gives an introduction to supervised learning, including two very popular types of techniques: nearest-neighbour methods and decision trees. The math notation can be a bit of a shock at first, but don't stress! We're not going to delve deeply into machine learning theory or do things like mathematical proofs in this course. However, mathematical notation is very useful to neatly describe a lot of the models and algorithms in machine learning.

The training set can be written as:

$$T = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$$

So we often assume that there are multiple inputs (aka features) as a vector for each data point:

$$\mathbf{x} = [x_1 \;\; x_2 \;\; \cdots \;\; x_p]^\top$$

(The vectors are written as rows to look nice in a book: the transpose then makes matrix operations work!)

There is a single (scalar) output value. Multiple outputs is also possible.

We can have numerical and/or categorical variables in our data (see, e.g. Table 2.1). If y is categorical, we have a classification problem. If y is continuous, it's a regression problem.

The number of classes in a classification problem is denoted by M. We will often discuss binary ($M = 2$), but extensions to more classes is usually possible.

Example 2.1: Classifying songs

It is worth noting that the classes fundamentally overlap in this (2D) feature space. Any classifier that gets 100% performance on this training set is almost certainly doing something a bit weird and may well perform badly on data not included in the training set.

Example 2.2: Car stopping distances

We can see that the data follows some rough trend. Regression could give us a model that captures this trend, but we can also see, e.g. reasonably different distance values seem to occur at almost the same value for speed.

We don't usually care too much about training set performance in machine learning. What matters is the ability of the model to **generalise** to data that was not used for testing.

k-Nearest Neighbours (k-NN)

k-NN is a simple and intuitive model for supervised learning. Start by looking at **Example 2.3**. The heart of this model is a distance metric: the output/prediction is determined by looking at the labels/true values of nearby points in the training set.

Example 2.4 shows how we can visualise the prediction that the model makes over the entire feature space. This reveals the **decision boundaries** of k-NN.

Next, look at the **pseudocode** (and math notation) for k-NN in Method 2.1. k-NN is sometimes called a lazy learner, because training really just means storing the dataset. It's also called nonparametric, because it doesn't really have any clear "parameters" that are changed when the model is trained (unlike other models that we will see soon). How do we determine the output for an input data point? Majority vote for classification (how do we break ties? Well...), averaging for regression.

The model will depend on the value of k and this has to be chosen by the user when applying k-NN. This is an example of a hyperparameter. Figure 2.5 illustrates the effect of varying k.

When $k = 1$, the model will tend to be highly sensitive to local variations in the labels, possibly leading to poor performance because of **overfitting**. So how do we choose the value of k? This question takes us to perhaps the central challenge in machine learning! We will have lots more to say about this in the rest of the course.

Lindholm et al. also note that k-NN is one method that is sensitive to the scale of variables in the data. Normalisation or standardisation is advised.

Decision Trees

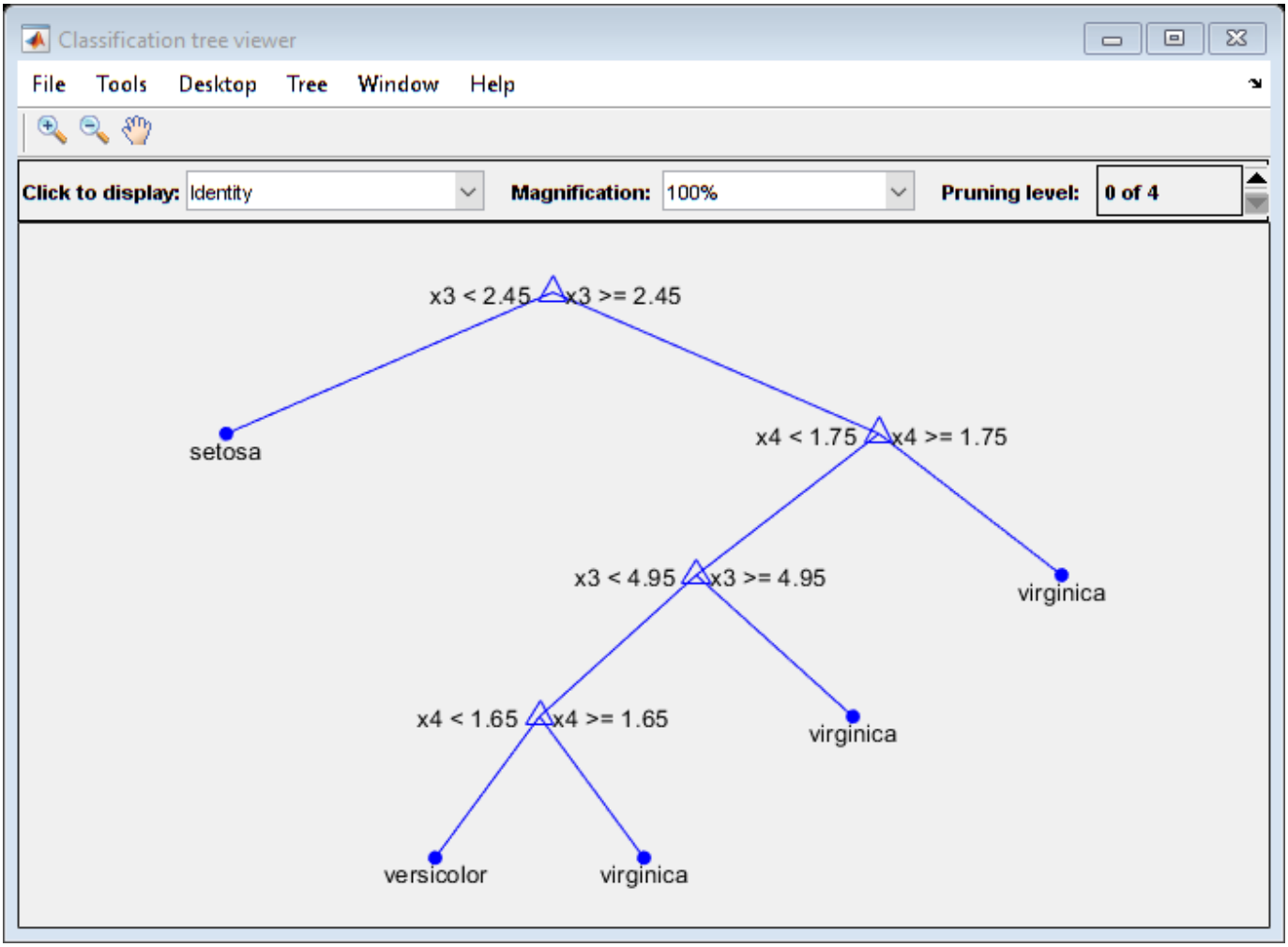
Decision trees are a very popular machine learning model. Perhaps their main advantage is their **transparency** and **interpretability**.

Let's focus on (binary, univariate) decision trees for classification. **Example 2.5** is a nice introduction.

```
%% View Decision Tree
% This example shows how to view a classification or regression tree.
% There are two ways to view a tree: |view(tree)| returns a text
% description and |view(tree,'mode','graph')| returns a graphic description
% of the tree.
%%
% Create and view a classification tree.
load fisheriris % load the sample data
ctree = fitctree(meas,species); % create classification tree
view(ctree) % text description
```

```
Decision tree for classification
1  if x3<2.45 then node 2 elseif x3>=2.45 then node 3 else setosa
2  class = setosa
3  if x4<1.75 then node 4 elseif x4>=1.75 then node 5 else versicolor
4  if x3<4.95 then node 6 elseif x3>=4.95 then node 7 else versicolor
5  class = virginica
6  if x4<1.65 then node 8 elseif x4>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

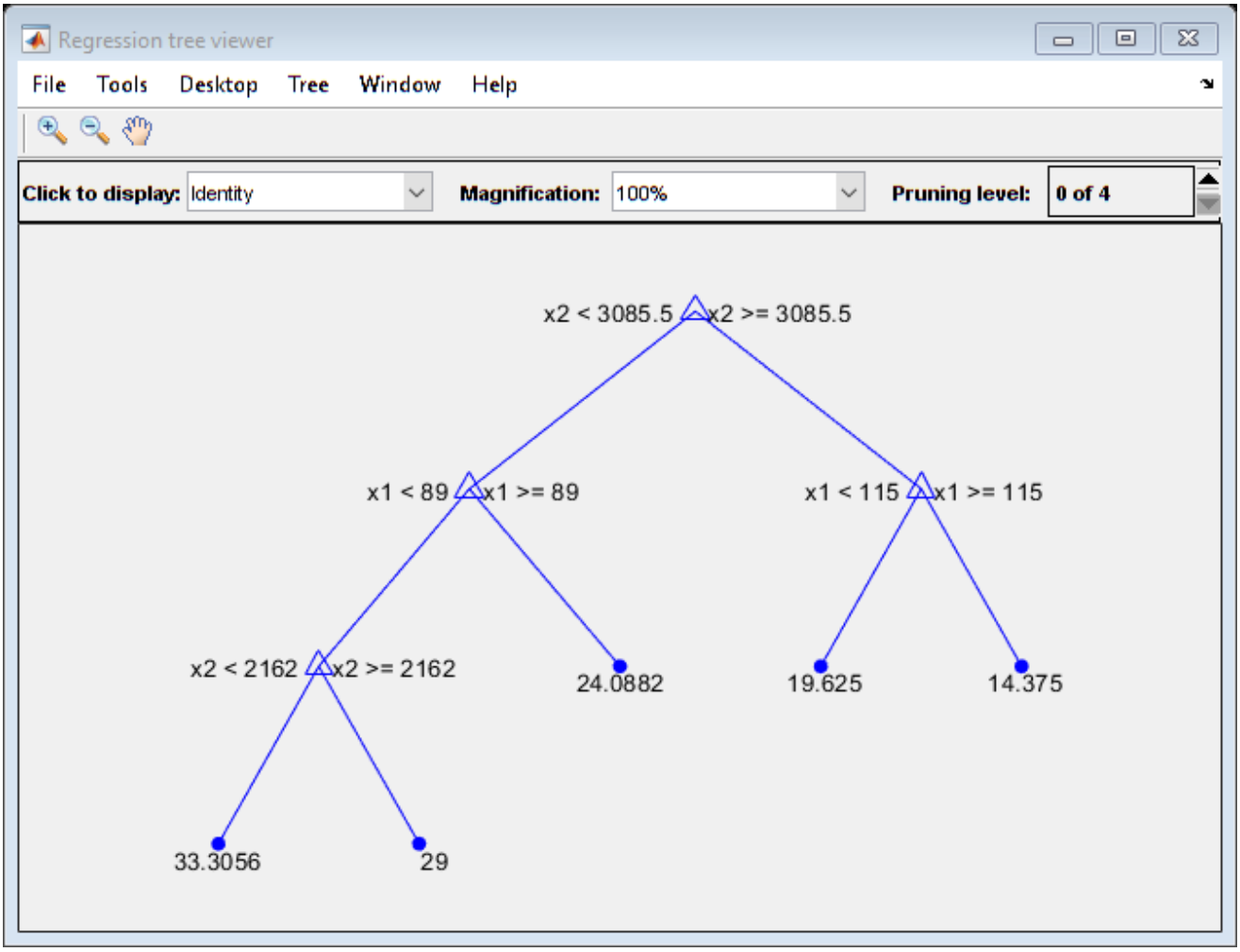
```
%%
view(ctree,'mode','graph') % graphic description
```




```
%%
% Now, create and view a regression tree.
load carsmall % load the sample data, contains Horsepower, Weight, MPG
X = [Horsepower Weight];
rtree = fitrtree(X,MPG,'MinParent',30); % create classification tree
view(rtree) % text description
```

```
Decision tree for regression
1  if x2<3085.5 then node 2 elseif x2>=3085.5 then node 3 else 23.7181
2  if x1<89 then node 4 elseif x1>=89 then node 5 else 28.7931
3  if x1<115 then node 6 elseif x1>=115 then node 7 else 15.5417
4  if x2<2162 then node 8 elseif x2>=2162 then node 9 else 30.9375
5  fit = 24.0882
6  fit = 19.625
7  fit = 14.375
8  fit = 33.3056
9  fit = 29
```

```
%%
view(rtree,'mode','graph') % graphic description
```



We have a binary tree which leads to a classification at the leaf nodes. At non-leaf nodes, a decision (split) is made using a single feature from the dataset which can be written as an "if...then...else" rule. So, for a test point, it's pretty easy to explain the decisions that lead the classifier to the prediction it makes.

A tree like this produces decision boundaries that partition the space into axis-aligned rectangles. From [Hastie09]:

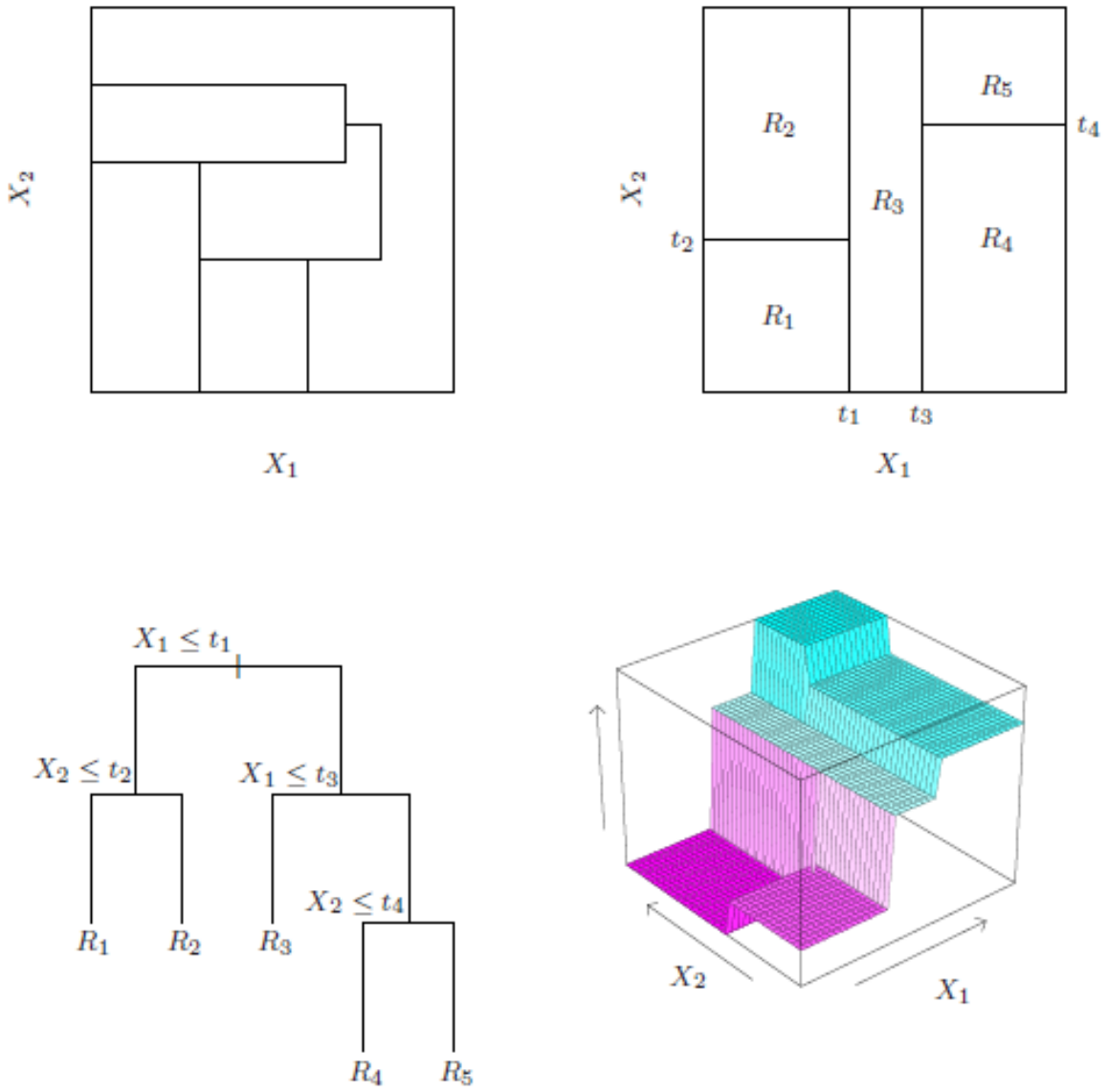


FIGURE 9.2. *Partitions and CART. Top right panel shows a partition of a two-dimensional feature space by recursive binary splitting, as used in CART, applied to some fake data. Top left panel shows a general partition that cannot be obtained from recursive binary splitting. Bottom left panel shows the tree corresponding to the partition in the top right panel, and a perspective plot of the prediction surface appears in the bottom right panel.*

Learning Decision Trees

Given a dataset, how is the tree constructed? Specifically, we need to determine the threshold values for the rules (i.e. the splits in the tree) to optimise the prediction. If $\mathbf{x} \in \Re^p$, there are an infinite number of (sets of) values. But the only time a prediction (on the training set) will change is when a threshold crosses a data point. Even so, the **search space** is intractable. The standard algorithm used to learn the tree is recursive binary partitioning. It is a greedy algorithm.

We can grow a tree until there is only a single data point in each region defined by each leaf. However this might result in very large trees! In practice, some stopping criterion is used, such as a limit on the depth of the tree or a minimum on the number of data points associated with each leaf node. This will lead to a hyperparameter that needs to be specified.

For regression, learning tries to minimise the sum of squared errors, with the prediction given by the average of the data points in a region.

For classification, we clearly want to minimise something related to whether or not the tree is getting the prediction correct for each (training) data point. There are several choices here, the most common being:

- misclassification rate
- Gini index
- entropy

Example 2.6 goes through an example to show how the learning algorithm works.

Lindholm et al. discuss the potential difference between misclassification rate, Gini index and entropy criteria in terms of the purity of the nodes produced. A node is more **pure** if it assigns more of the same class to every data point that "comes its way". More purity can lead to fewer splits in total. Gini index and entropy tend to be better than misclassification rate.

(Note that we're essentially talking about heuristics here! So the advice is typically good, but it depends on the data so is not guaranteed.)

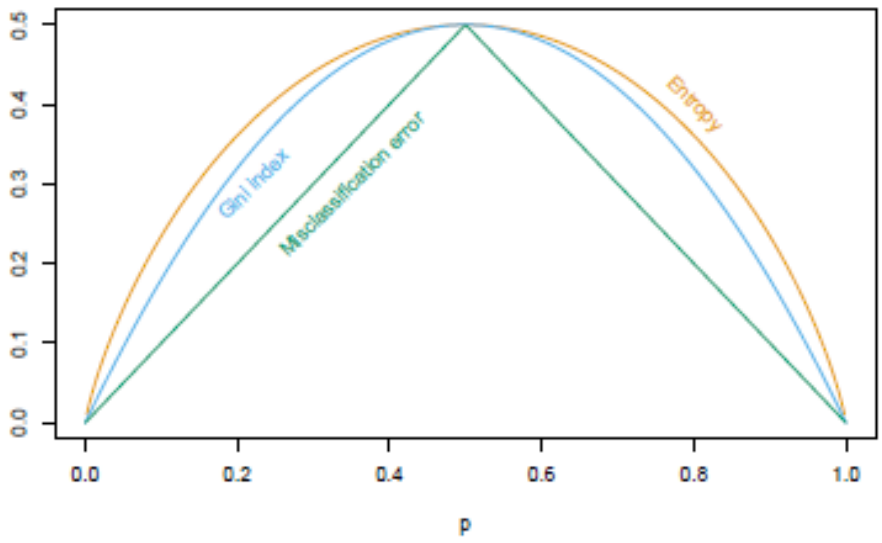


FIGURE 9.3. *Node impurity measures for two-class classification, as a function of the proportion p in class 2. Cross-entropy has been scaled to pass through* $(0.5, 0.5)$.

(from Hastie09)

Fig. 2.11 illustrates the effect on varying the depth of the tree. The actual learning and prediction algorithms are shown in **Method 2.2**.

Well-known (similar) algorithms are "Classification and Regression Trees" (CART), ID3 and C4.5.

Module 3: Basic Parametric Models and a Statistical Perspective on Learning

In Chapter 3, Lindholm et al. cover the two oldest and most popular parametric models that can be applied to supervised learning: linear regression (regression) and logistic regression (classification, despite the name!). What is also important here is that we are building a statistical framework around these models that will be useful in understanding much of the remaining course material. Statistics gives us a way of handling uncertainty! Also, while you have probably seen a 2D linear equation before, we want to be able to apply our models even when the data large (many features and many observations).

Linear Regression

Start by having a quick look at **Fig. 3.3**. Fundamentally we are doing line/curve fitting. The hope is that the model will capture the main/general trend of the data, but we probably don't expect it to perfectly fit the data for any real problem. To handle this, we declare some uncertainty and put something in to handle this: additive noise (ε):

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + \varepsilon$$

Look at **Fig. 3.1** and **Fig. 3.2**.

The math notation gets a little hectic here (vector-matrix, the "column of 1's", hats on learned parameters/predictions), but it will get easier once you get used to it!

In the end all the math turns into numbers in a trained model (**Example 3.1**).

The Maximum Likelihood Perspective

Linear algebra gives us a way to fit our linear model to the data, but what are we really doing and how does that fit into a sensible statistical framework? There are two things to understand here:

1. If our data and model parameters are instances of random variables, **the likelihood function is something that is a sensible thing to optimise** (maximize) if we want to build a good model for the data.
2. For linear regression, if we assume Gaussian (Normal)-distributed noise, then **minimising the squared error loss is equivalent to maximising the (log) likelihood**.

Categorical Input Variables

Lindholm et al. make a little aside here to discuss encoding categorical variables (which then allows those variables to be used in many machine learning models). This includes the (so-called) **one-hot encoding**, that is important in models like deep neural networks.

Logistic Regression

Don't be put off by the math notation - logistic is just linear regression where the output (target) variable is categorical (e.g. 0 or 1). So we need to add a squashing (sigmoidal) function to constrain the output to lie in $[0, 1]$. Look at **Fig. 3.5**, then **Fig. 3.4**.

Two key issues to understand from this Section:

1. To understand that **logistic regression fits nicely within the maximum likelihood framework** (hence the math notation!).
2. To understand that **training the model requires an iterative/numerical optimisation algorithm** (unlike linear regression where linear algebra provides a **closed-form solution**).

Understanding (1) is very useful because it allows us to use the output of the model ($g(\mathbf{x})$) as a probabilistic prediction of the class label for an input - see Example 3.3.

Predictions and Decision Boundaries

Probabilistic predictions for classification are nice because they **convey the uncertainty of our prediction**. If we are uncertain, perhaps the best output is "I don't know".

If we need to output a hard decision, we need a **rule** for doing so. It seems sensible to **choose the most probable class** as our prediction (and with some assumptions, it is). This corresponds to a **threshold value**, $r = 0.5$. Now we have defined the **decision boundary** for our classifier, partitioning the feature space up into regions where a certain class will be predicted by the model. **Logistic regression creates linear decision boundaries** (Fig. 3.5), in contrast to k-NN. Since we often work with multidimensional data, the term **hyperplane** is often used to describe linear functions such as a decision boundary.

Regarding issue (2) above, what the math shows is that for logistic regression, training the model via maximum likelihood means that the objective function for our optimisation problem is something called **cross-entropy loss**. We hand this function over to a numerical optimisation algorithm and it **finds model parameter values that minimise this loss function**.

Logistic Regression for More Than Two Classes

Things get a bit more complex when we have more than two classes. As we see in **Fig. 3.6**, we need a model that can create three decision boundaries/regions for three classes. The softmax function squashes the output with a sigmoidal shape and (**importantly**) normalises the outputs so that they all sum to one for any given input vector.

The model is still just a linear sum of inputs multiplied by model parameters, squashed through a sigmoid. Alternatively, the linear sum is a dot product of an input vector with the model parameter vector. This is called the **logit**, z (see **Eqn 3.28, 3.41, 3.42**).

A useful way to think of this is that we have a logistic regression model for each of the Mclasses in the data, but they are tied together with the softmax and also with the (multiclass cross-entropy) loss function that we need to train the model (to maximise the likelihood). **Example 3.4** doesn't do a lot for me, but I think the authors are trying to show that, given data and model parameters, all of the math notation really just comes down to calculating groups of numbers that are conveniently arranged in vectors and matrices.

Polynomial Regression and Regularisation

Lindholm et al. describe polynomial regression in a rather unusual way. More commonly, polynomial regression is thought of in terms of **curve fitting**. If we have a 1-D regression problem, $y = f(x)$, we can choose a mathematical function of whatever form we please as a model of the relationship between x and y. Polynomials are one class of functions, with a linear function being a special case (polynomial of order 1). See **Example 3.5**. As we increase the order of the polynomial, we can see that the model can become increasingly non-linear.

The book offers a different view. The idea is that we have some **non-linear transformations (basis functions)** that we use as a first step to transform/pre-process the input features into some other **representation of the data**. After that, the model is just the linear model again. This fits well with the previous notation and also with things we will discuss later, in particular kernel methods and support vector machines.

More complex models are potentially more "powerful", but when we move from fitting the "general trend" of the data to interpolating the training data, we very likely end up with eccentric models that overfit the data (and would actually perform worse on test data than simpler models). Dealing with model complexity is one of the most important issues in machine learning. One way of doing this is to perform regularisation. The most common way of doing this is introduced here: L^2regularisation. This will come up again later in the course.

Generalised Linear Models

We will leave this as optional reading. You would likely encounter such models in some STAT courses.

Module 4: Understanding, Evaluating, and Improving Performance

The book takes an in-depth look at these topics, probably deeper than I've seen in any other machine learning book. We will follow their terminology, but note that it is not universal.

4.1 Expected New Data Error E_{new}: Performance in Production

An **Error function**, $E(\hat{y}, y)$ compares the predicted value with the true value, with smaller being better. There are many possibilities but the default for regression is **squared error** and for classification it is **misclassification** (1 - **accuracy**) (eqn. 4.1). This could be the same as the loss function, but not nescessarily. The loss is used for training, the error is for evaluating a trained model.

In machine learning our ultimate goal is to build models that can be deployed to predict new, unseen data forever in the wild. It is very useful to think about the world as a "data generator". We can imagine that this data follows some unknown distribution, $p(\mathbf{x}, y)$. The default assumption is that this distribution is stationary (not shifting over time) and that our data points are drawn independently from this distribution (i.i.d). Then, we can say that **our goal is to minimise the expected new data error**, E_{new}, i.e. the expectation over all possible test data points with respect to $p(\mathbf{x}, y)$ (**Eqn. 4.2, 4.3**).

The error on the training set is E_{train} (**Eqn. 4.4**).

Lindholm et al. make a statement that might seem strange: "...but in general (E_{train}) gives no information on how well the method will perform for new unseen data points. We often hope that, in the end, getting good training

performance DOES mean good test performance and a lot of the time it does, if the data is good and plentiful and if the complexity of the model is sensible. But these things are not easy to guarantee.

As they say, minimising E_{train} might not be the best thing to do to minimise E_{new} (e.g. it might be a loss function that is related to E_{train} but has other terms).

4.2 Estimating E_{new}

Reasons to care about E_{new}:

- Is the performance of our model good enough (worth nothing that the problem ultimately determines how much we might be able to improve performance, e.g. by collecting more data or changing the model).
- **Model selection**
- **Model configuration**
- Reporting expected performance to the end-user

Unfortunately, we can't compute E_{new} so we have to think about estimating it.

A good example to see why $E_{train} \approx E_{new}$ is to think about using a "lookup table" as a supervised learning model. We can store the training data in a table and then always get $E_{train} = 0$, but this isn't going to help us predict new data. In practice, we often (but not always) find that $E_{train} < E_{test}$.

Instead, we can set some (n_v) of the data points aside (randomly) and use these to estimate E_{new}. This is called a **hold-out (or validation)** dataset ($E_{hold-out}$) (Fig. 4.1).

Intuitively, as n_v increases, it seems like our estimate of E_{new} should become more accurate. But there is a trade-off because if we are taking the data out of T (n decreases) then our model will have less data to train on. As a result, E_{train} and E_{new} itself are likely to increase! This is not cool unless we have a large amount of data. There isn't really a "standard" percentage split - people use 70/30, 80/20, etc.

An alternative method for estimating E_{new} is known as **(k-fold) cross-validation**.

How it works (**Fig. 4.2**): Randomly partition your data into l equally sized subsets. Put one subset aside, train on the other $l - 1$ subsets, and test on the "held out" subset. Repeat this k-fold (times), holding out subsets in turn. The final estimate of generalisation error is calculated as the average of the k hold out error values.

$k = 10$ is most common. If we increase k we spend more computation time but make the most of the data we have. In leave-one-out cross-validation, a single data point is held out and $k = N$, the number of training points!

Lindholm et al. talk a little bit here about our estimates of E_{new}; whether they are (un)biased, high/low variance, etc. But they don't really explain what these things mean until Section 4.4.

Cross-Validation is an overloaded term in machine learning. It is also used in the context of **model selection** (i.e. choosing model hyperparameters or even selecting which model to use). For example, we try to tune k-NN to find the best value of k (i.e. that might minimize E_{new}) via hold-out or k-fold. But this **invalidates its use** as an estimator of E_{new}! This means that we often want to have a test set as well, to provide an estimate of E_{new} after we've finalised the model.

4.3 The Training Error-Generalisation Gap Decomposition of E_{new}

We often see statments being made about machine learning techniques in terms of "it doesn't work", "performance is better than human", etc. But these statements are usually too vague to have very much meaning. To really understand the performance of an ML model, it is important to think about the **generalisation gap** (the difference between the new data error and training data error). We want to talk generally, so we don't want it to depend on a single, specific training set (from this point of view, E_{train} is also a random variable). If we had multiple different training sets, we could carry out our training multiple times and average the results to improve our estimates (Eqns. 4.8). This gives us $\bar{E}_{train}$ and $\bar{E}_{new}$:

$$\text{generalisation gap} \triangleq \bar{E}_{new} - \bar{E}_{train}$$

Fig. 4.3 shows us the idealised relationship between model complexity and error in ML. Both **underfitting** and **overfitting** are not desirable, but one difficulty is that this depends on the problem. **Example 4.1** illustrates this by creating artificial data, so that we know what E_{new} is.

The size of the dataset (n) also effects the generalisation gap (Fig.4.6): more training data usually means smaller generalisation gap, though $\bar{E}_{train}$ probably also increases!

Lindholm et al. talk about trying to have small E_{train} and small generalisation gap and end up with this practical advice:

- If $E_{hold-out} \approx E_{train}$, you would think the generalisation gap is small, but you could be underfitting. To try and improve results, maybe increase model flexibility.
- If E_{train} is close to zero but $E_{hold-out}$ is not, you could be overfitting. To try and improve results, maybe decrease model flexibility.

This is pretty good advice, all things considered. You can bet that many people applying ML to real-world problems all the time have not thought in this way and are not following this advice!

Model complexity is a tricky business for most ML models (e.g. that have multiple hyperparameters). **Example 4.2** uses an artificial problem to give a taste of this (we can't even do this for real data!).

4.4 The Bias-Variance Decomposition of E_{new}

To get better insight into the performance on an ML model and it's complexity, it is very useful to decompose the expression for error into different components. These are the (statistical) **bias** and **variance** of our estimates. For example, we use a GPS to measure our location. Our true location is z_0, but our GPS gives us noisy measurements (random variable z). If we read the GPS several times, we make observations of z. The mean is $\bar{z} = \mathbb{E}[z]$.

Bias: $\bar{z} - z_0$

Variance: $\mathbb{E}[(z - \bar{z})^2] = \mathbb{E}[z^2] - \bar{z}^2$

The variance describes variability we get in our observations (e.g. from noise in the GPS measurements). The bias is some systematic error (e.g. the GPS for some reason always gives measurements a bit to the left of where they should be). Eqn 4.13 shows that:

$$\text{expected squared error} = \text{variance} + \text{bias}^2$$

This is similar when we consider training an ML model: z_0 corresponds to the true relationship between the input and output features, z is the trained model (random variable because the training set is drawn from some distribution over the input and output features). Lindholm et al. describe the story again (**Eqns. 4.14 - 4.19**) and it is worth reading this. It is an example of how the **mathematical notation in ML can quickly become daunting, but no proofs and very little mathematical analysis/operations are actually happening. Rather, it is a way for us to precisely write important concepts down**.

Fig. 4.8 shows the idealised relationship between error and model complexity in terms of the components of the error (bias, variance and irreducible error). If we had a model (maybe a simple one) with no hyperparameters, this would end up as a **single point** on the x-axis (I would call this a model instance). If a model has hyperparameters, it is likely that the complexity of the model can be changed by varying the values of the hyperparameters. But **this relationship is usually not simple**, as we shall see throughout the course.

The bias is mainly of a property of the model. The variance also depends on the model, but is strongly affected by the data, in particular the number of data points, n (**Fig.4.9**). Calling n the "size of training data" is a bit dangerous because a data set has rows (observations, data points) **and** columns (features, dimensions).

Example 4.3 shows the actual curves, using artificial data and a polynomial regression model. L^2 regularisation is added to the loss function which varies the model complexity, so that becomes the (flipped!) x-axis. Have a think about what it would take to write some code to reproduce the results in this plot.

In practice, we don't know what the exact values of bias and variance are for a particular application. But knowing what is going on "underneath" can be very helpful to make things work in practice. Example 4.4 gives a couple more nice pictures. Lindholm et al. touch on a few things that haven't been covered yet, so this could be good to come back and look at in revision.

4.5 Additional Tools for Evaluating Binary Classifiers

Classification problems occur in many different areas, in particular two-class (binary) problems. Often, we want to know more detail than just the overall performance of a model. One very good reason is that in many real-world problems, the training data might be **imbalanced** (many more examples for one class compared to another class).

A **confusion matrix** is a table that breaks down for us a comparison of the class predicted versus the true class for the training data. Lindholm et al show the binary case (but this is easy extended to more classes). We then have a lot of terminology, starting with the four possible outcomes for predicting a data point (**true (T)/false (F), positive (P)/negative (N)**), for example precision and recall. It's good to look at these but it's not a list you have to memorize!

Using a classifier often involves applying some (adjustable) threshold value to make the decision about predicting positive or negative (e.g. logistic regression). If we change this threshold, we are likely to see the classification performance change, but what if we lower our FP rate but at the same time increase our FN rate?

A common way of evaluating a (binary) classifier is to produce a "receiver operating characteristics" (**ROC**) curve by looking at the performance for all values of the threshold (see **Fig.4.13a**). To summarize this picture in a single number, the area under the ROC curve (**AUC**) is often then calculated.

In addition to class imbalance, it is common to have classification problems where we care more about predicting one class correctly than the other (e.g. medical diagnosis). Lindholm et al. call this **asymmetric** (though I'm not sure that term is used generally). Situations like this (making decisions based on probabilities, with different costs for different errors) have been explored deeply in statistics and are also prominent in areas such as economics and operations research (e.g. expected risk, utility theory).

The F_1 score is commonly used as a metric for classification, and is appropriate if the problem is imbalanced with the negative class being most common (as long as it is symmetric!). In this case, a **precision-recall curve** is better than the ROC curve (**Fig.4.13b**).

An example of using some of these measures is given in **Example 4.5**.

Module 5: Learning Parametric Models

Chapter 5 of Lindholm et al. is focussed on the "learning" in ML. This often comes down to **solving an optimisation problem**.

5.1 Principles of Parametric Modelling

A general expression for a regression model is: $y = f_\theta(\mathbf{x}) + \varepsilon$. We are trying to model the relationship between the inputs and the output with a function ($f()$) that has parameters (collected into a vector, θ). We also assume additive noise (ε). In the case of linear regression, we saw that:

- the model output is a linear function of the parameters
- the model can be trained in one step, with a closed-form solution using least squares
- Assuming Gaussian noise, minimizing SSE is equivalent to maximum likelihood.

But in ML we want to train non-linear models, using various loss functions and (perhaps) with different noise assumptions.

Non-linear models are often used in science and engineering (e.g. Example 5.1), where the model parameters have a particular meaning. However in ML this is not usually the case.

Conceptually, we can replace the linear function in linear regression and the logit in logistic regression with some other nonlinear function $f_\theta(\mathbf{x})$ and things still fit into the maximum likelihood framework. To learn the model, we then need to **solve an optimisation problem (Eqn. 5.4) as a proxy for thing we really want to optimise (Eqn. 5.5)**. So it's not just a matter of solving a numerical optimisation problem:

- Optimising on the training set beyond a certain point is not going to help, because the training set is a finite sample (statistical accuracy, variance, overfitting, etc.).
- The evaluation function (error) might be different from the loss function used for training. This gives us some ability to do things like control model complexity (e.g. explicit regularisation).

- Early stopping is one way of achieving good generalisation, by not allowing the optimisation to progress all of the way to a solution.

5.2 Loss Functions and Likelihood-Based Models

This section of the book is fairly detailed. The essential points are:

- There are a variety of different loss functions used in machine learning, some more common than others.
- They have some different properties that may be useful in certain situations.
- Given the specific application, some combinations of model and loss function make more sense than others.

For **regression**, we have seen squared error (aka L_2) loss and we know it corresponds to maximum likelihood with a Gaussian noise assumption. Absolute error (L_1) is more robust to outliers and can help with model complexity, but it might require different optimisation techniques. Boutique options include **Huber loss** and ϵ**-insensitive loss** (Fig. 5.1).

For **classification**, the **misclassification loss** (Eqn. 5.1) is intuitive but often not used directly. One reason is that it results in a cost function that is piecewise constant and so not very friendly to numerical optimisation algorithms that use gradient information (see below). If we have a binary classifier that predicts conditional class probabilities $p(y = 1|\mathbf{x})$ in terms of a function $g(\mathbf{x})$, then maximum likelihood leads us to optimise **cross-entropy loss** (Eqn. 5.11).

One idea that has led to some powerful ML methods is to work with **a measure of how close the (training) data points are to the decision boundary** (aka the **margin**). If we can bring that information into the loss function, it allows us to train a model (for example) to try and **maximize the margin** (see, e.g. support vector machines later in the course). For example, if we think about logistic regression then we end up making a (hard) class prediction by thresholding the logit:

$$\hat{y}(\mathbf{x}) = \text{sign}\{f(\mathbf{x})\}$$
$$\hat{y}(\mathbf{x}) = \text{sign}\{\boldsymbol{\theta}^T\mathbf{x}\}$$

Ignoring the case where a point falls exactly on the decision boundary, the prediction is either -1 or +1. So **the margin of a classifier for a data point** $(\mathbf{x}, y)$ **is** $y \cdot f(\mathbf{x})$. There are several loss functions that include the margin. Some allow a probabilistic interpretation of the prediction, some don't (Fig. 5.2).

For **multiclass classification**, cross-entropy is straightforward, but loss functions that use the margin are not. In practice, there are two common ways to tackle these problems:

- One-versus-rest: train M binary classifiers, each one predicting one of the classes versus all of the other classes. To predict a point, evaluate each classifier and use the prediction with the greatest margin.
- One-versus-one: train a classifier to discriminate between every pair of classes ($\frac{1}{2}M(M-1)$ pairs). To predict a point, evaluate each classifier and take the majority vote, using the margin to break a tie.

[Note: in the interest of time we will leave the next two subsections as optional reading (Likelihood-Based Models and the Maximum Likelihood Approach; Strictly Proper Loss Functions and Asymptotic Minimisers]

5.3 Regularisation

Regularisation means some strategy for **controlling model complexity**, in particular avoiding models that are too complex. The book says that this translates to preferring/encouraging model parameters to be small (absolute value), with respect to "parametric models" like linear and logistic regression. Regularisation can be either **explicit** or **implicit** (see below).

Examples of explicit regularisation (**Eqn. 5.26**):

- L^2 regularisation adds a penalty term to the cost function that is the squared L^2-norm of the model parameters, with a hyperparameter $\lambda \geq 0$ to weight the influence of this term (e.g. **Eqn. 5.22**). In linear regression, this is called **ridge regression**. In neural networks it is called **weight decay**!
- L^1 regularisation adds the L^1-norm instead (usually called Manhattan distance). This is more complex to optimise, but has the advantage that it tends to favour **sparse solutions**, where some of the model parameters become (exactly) zero. This is like getting **feature selection** for free (see **Example 5.2**)!

Implicit regularisation doesn't add an (explicit) term to the cost function, but tries to control model complexity some other way. We will see a few in the course:

- **Early stopping**: if training means running an iterative optimisation algorithm, monitor $E_{\text{hold-out}}$ and stop training when it achieves a minimum value.
- Dropout (neural nets)
- Data augmentation
- Splitting criterion in decision trees? Hmm. You could say that model hyperparameters can also have a regularisation effect. Here I think they are trying to focus on the training algorithm.

5.4 Parameter Optimisation

Optimisation is at the heart of machine learning. Optimisation itself is a huge field and not all of it is used in ML, but there are certain parts of it that are very heavily used. The most important of these is definitely numerical optimisation techniques that use gradient information to try and find a good solution. But there is plenty of stuff that the book does not mention (e.g. derivative-free methods, metaheuristics such as evolutionary algorithms, linear programming, integer programming, etc.). They highlight two types of optimisation problems:

1. Training a model
2. Hyperparameter optimisation (tuning)

Example 5.3 gives an illustration of some of the potential difficulties of a numerical optimisation problem. Recall that for the linear regression problem, we could find the solution to the training optimisation problem in one step (closed-form expression). This is partly because the objective function is convex (but note that not all convex functions have closed-form solutions - e.g. logistic regression). However many ML models give us a non-convex optimisation problem to solve.

Gradient descent iterates by taking a step in the direction of the negative gradient (**Algorithm 5.1**). The step-size parameter controls the magnitude of the step. **Fig. 5.7** shows that the behaviour of the algorithm will depend on the value of the step-size. Sometimes people set up an "internal" optimisation problem (to find

the optimal step-size) and solve it on every iteration of the algorithm (line search). **Example 5.5** gives us an insight into how the learning rate and the initialisation of the algorithm might effect the result when the objective function has multiple critical points.

Second order gradient methods (e.g. Newton's method) use second derivative (curvature) information to try and improve the optimisation performance, but they add considerable complexity and don't always produce a good result. We won't discuss them in detail, but **Example 5.7** is still useful because it shows how early stopping can work and reminds us that we probably don't actually want the best solution (global optimum) of our training problem!

5.5 Optimisation with Large Datasets

When we train a ML model, the objective function usually contains a term that is a **sum over the data points** in the training set. An extremely important idea in modern ML has been to approximate this sum with a subsample of the data points. This lead us to **stochastic gradient descent** and the idea of calculating gradients with respect to a subsample (**mini-batch**) of the data. As well as computational efficiency, it is possible that the "noise" might actually help the algorithm to escape bad solutions during training. If each loop of gradient descent updates on a mini-batch, then once we have worked through the entire dataset it is called an epoch (E in **Algorithm 5.3** - look at **Eqns. 5.36** and **5.37** first).

In neural networks/deep learning, there have been many different variations of stochastic gradient descent algorithms for training, often including different techniques for adapting the step size (learning rate) during training. The ADAM optimiser is the most widely-used optimiser in deep learning. It combines gradient and learning rate values from previous time steps and updates the search direction and learning rate using this information. Some hyperparameters and assumptions are involved!

5.6 Hyperparameter Optimisation

The hyperparameters of a ML model are typically a collection of different terms that are probably a mixture of discrete and continuous values. Gradient descent usually cannot be applied. Trying all combinations of several different values for each hyperparameter (**grid search**) is common but expensive if we have a lot of hyperparameters (**Example 5.9**, **Algorithm 5.4**). This is a place where black-box/metaheuristic optimisation algorithms have been successfully applied (e.g. AutoML.org).

Module 6: Neural Networks and Deep Learning

6.1 The Neural Network Model

A "neuron" in an (artificial) neural network is a highly simplified model of what goes on in a biological neuron. An artificial neuron is an processing unit that takes in several inputs via weighted connections. The neuron computes a function of the sum of these weighted inputs and produces an output signal (**Fig. 6,1(b)**). We can see from **Eqn. 6.2** (RHS) that this input is the same function as linear regression, replacing β_0 with b and $\beta_1, \dots, \beta_p$ with $W_1, \dots, W_p$. We have the constant (offset) input value of 1 (also often called a bias). For a linear neuron that's it! But we can also have neurons that compute some other function, $h()$, of the weighted sum of inputs. This is known as an **activation function** (in an analogy to a biological neuron "firing" and producing

an electrical output if it has enough stimulus on it's inputs). The two most common activation functions are the logistic sigmoid and the ReLU (shown in **Fig. 6.2**).

A linear neuron and a neuron with a logistic activation are mathematically identical to the linear and logistic regression models we have already seen. We build neural networks by creating layers of these neurons/units and connecting them together. The simplest case (skipped over in the book) is to have several neurons, all connected to the same inputs but having separate (scalar) outputs. If you draw a box around that whole thing and call it your "model" then:

- For linear units, this (single-layer) network can be trained to perform **multiple linear regression** (i.e. a regression problem with multiple outputs).
- For sigmoidal units, this (single-layer) network is close to the **multiclass logistic regression model** from Chapter 3. Using softmax instead of logistic sigmoids will make it identical.

Note that we're not yet talking about how to train these models.

Two-Layer Neural Network

If we have a single-layer model then the notation q_k is the output of the kth neuron, $k = 1, \dots, U$. These outputs are connected to a second layer of neurons which provide the final output of the model, $\widehat{y}$ (so the first layer is called a hidden layer). Every connection has a weight (parameter), so we have notation like $W_{73}^{(1)}$ representing the weight from input 3 to neuron 7 in layer 1! There is also a bias/offset signal/weight for the hidden layer (**Fig. 6.3**).

Vectorisation over Units

In **Eqns. 6.7 - 6.9**, we see that the function that is a two-layer (**feed-forward**) neural network (aka **multilayer perceptron**) can be written more compactly in terms of matrices and vectors.

Deep Neural Network

We can extend the model to any number of layers, creating a deep neural network. The notation $l \in \{1, \dots, L\}$ indexes the layers and the number of units in the lth layer is U_l. Each unit is **fully connected** to the input/units/outputs of in the previous/next layer (**Fig. 6.4**).

Vectorisation over Data Points

In the early days of feedfoward neural networks (1980s), these models were seen as examples of "parallel distributed processing". If we think about an input signal propagating from the inputs through the layers, it is clear that the computation across the units in any given layer is independent and so could be parallelised. Indeed, if we want to compute the output for a whole set of data points, the computation required for each data point is independent. So there is more than mathematical notation to motivate representing things as vectors (this is at the heart of why GPUs are so useful for deep learning). Stacking the data points results in even more compact notation for the network (**Eqns 6.12 - 6.14**).

Neural Networks for Classification

For regression, we just have linear activation on the output units. For classification, we use the same building blocks from Chapter 3. For binary classification, a single output unit with a logistic sigmoid can do the job. For

$M > 2$ classes, we have Moutput units and use softmax activation functions, ensuring (as in Chapter 3) that $\sum_{m=1}^{M} g_m = 1$, where g_m is an output value that estimates $p(y_i = m|\mathbf{x}_i)$. **Example 6.1** puts some numbers onto all of this for a very well-known dataset, the MNIST hand-written digits dataset.

6.2 Training a Neural Network

Training is an optimisation problem over the parameters of the model. If we want to refer to all of the model parameters, we collect all of the weights (and bias weights) and arrange them in a big vector, θ(**Eqn. 6.17**). The loss function that is a sum over the losses for each data point in the training set (**Eqn. 6.18**). Gradient descent (Chapter 5) is the basis of most commonly-used training algorithms. So the key question is: how can we get the gradient for a multi-layer neural network with nonlinear activation functions?

Backpropagation

Lindholm et al. use the term "backpropagation" to mean computing the gradient, although it is often used when referring to the gradient descent algorithm as well. To get the gradient for this model, the chain rule of calculus is critical. The mathematical expressions in the book can be interpreted as follows:

- We need the partial derivative of $J(\theta)$ with respect to each weight and bias weight, in each layer of the network (**Eqn. 6.22**).
- We use these to do our update for gradient descent (**Eqn. 6.23**)

The gradient for multiple data points is just the sum of gradients for a single data point, so they start by writing the math for a single data point, $(\mathbf{x}, y)$.

- The first thing we have to do is calculate the network output for $\mathbf{x}$ and then the objective/loss function using y (**Eqn. 6.24**). This is called forward propagation.
- More notation: the gradient with respect to the (total weighted sum of) input signals to the units in a layer, $d\mathbf{z}^{(l)}$ and the output signals of units in a layer, $d\mathbf{q}^{(l)}$ (**Eqn 6.25**).
- Then, we start calculating the gradients at the output layer, using the derivative of the activation function and J(**Eqn. 6.26**).
- The gradients for the weights and bias weights in that layer can be computed, and the gradient signals in the current layer are used to compute the gradients for the previous layer (**Eqn. 6.27**).
- **Algorithm 6.1** includes multiple data points (otherwise we would have nested for loops!).

Initialisation

There is a lot more that could be said here, but the authors make the essential point: weights are typically initialised to small random values at the start of (gradient descent-based) training.

Example 6.2 compares training a logistic regression (one layer) model v's a two-layer model on the MNIST data.

This is also fun to play with: https://playground.tensorflow.org/

6.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a big part of the reason why deep learning has become such a hugely popular part of the surge of interest in AI/ML. The fundamental idea is to **take into account structure in the data, specifically in terms of the relationship between different inputs in the data**. CNNs were originally applied to image data so we talk about them in this context, but they have been applied to many other types of data with spatial and/or temporal structure.

Data Representation of an Image

Fig. 6.9 shows the representation of a (6x6) greyscale image. We want to preserve this structure, so we need to think of the input and layers of the neural network in terms of matrices rather than vectors. If we have 36 hidden units for this 36-D input, a fully connected (dense) layer would require 1296 weights (plus biases). Convolutional layers have far fewer parameters and do so by enforcing some useful structure:

- **Sparse Interactions**: Lining up the input matrix with the hidden unit matrix, only a limited local region of inputs are connected to each hidden unit (or think of the other weights as being fixed at zero) (**Fig. 6.10**). Borders are handled with (typically) zero-padding (**Fig. 6.11**).
- **Parameter Sharing**: rather than having separate weights for each hidden unit, how about we force them to all share the same weight values! The weights are learned in every position of the local input window and this set of weights is called a **filter**. This isn't quite the mathematical definition of a **convolution**, but still uses the name (**Eqn. 6.29**). We can have more than one filter in a layer, with each filter getting it's own weight parameters (see Multiple Channels below).

Important: these properties make the CNN relatively invariant to translations of objects in images.

Convolutional Layer with Strides

Another technique to reduce computation is to move the filter (i.e. the input window) around by more than one pixel at a time when computing an output (e.g. Fig. **6.12**). Now we're thinking about the hidden units as different activation values that we get from moving the filter around. The number of hidden units then is determined by the filter size and the **stride**. The edges can get messy as well depending on what sizes are chosen!

Pooling Layer

A pooling layer has no trainable parameters but is used as an additional component to reduce the size of the model, e.g. by computing the **average** or **maximum** of values over a given filter size (**Fig. 6.13**).

Important: pooling can make the model more invariant to small translations of the input.

Multiple Channels

With a single filter we still produce this matrix of "hidden units", so if we use multiple filters we get a "stack" of these matrices. These are called channels. Now the layer of hidden units is a **tensor** (rows x columns x channels) (Fig. 6.14). When a CNN is trained on colour images, it is typical to have one (or more) separate channels for the Red, Green and Blue intensity values of each pixel in the image.

Full CNN Architecture

All of the above ideas are typically applied multiple times and followed by a final densely-connected layer to get final outputs (e.g. for a classification problem). **Figure 6.14** and **Example 6.3** shows an example with training results. This may seem like a complex type of model, but in fact state-of-the-art CNNs now typically have tens

or hundreds of convolutional layers! For reasons that are not fully understood, these models have continued to achieve remarkable performance in many real-world classification tasks.

6.4 Dropout

Lindholm et al. get a bit ahead of themselves here by talking about bagging and ensembles (Chapter 7!). Let's simplify this a little for now and revisit when we get to those topics.

The basic idea of dropout is to try and reduce model complexity by randomly choosing a fraction of the hidden units in the network and removing/dropping them. During training, we then compute the gradient only with respect to the remaining (sub)network and only update the weights in that sub-network. This process then iterates. During testing, all the units are brought back into the (full) network and used together (**Example 6.4**).

Module 7: Ensemble Methods: Bagging and Boosting

Ensemble (aka committee) methods create a predictor by building a collection of (base) models and combining them together. This sounds like it could produce a complex model, but it turns out that ensemble methods have some favourable properties that mean they often achieve excellent performance.

7.1 Bagging

Recall the bias-variance tradeoff: complex models are potentially good for solving challenging problems in ML and they typically have low bias, but also have the potential to overfit the training data (because of high variance). Bagging (bootstrap aggregating) is a simple **resampling technique** that (provably) **reduces the variance** of a model without increasing the bias. Example 7.1 shows the basic idea for a regression problem.

The Bootstrap

Bootstrapping is a more general idea from computational statistics where it is used, e.g. to quantify uncertainties in statistical estimators. Given a dataset $\mathcal{T}$, create multiple datasets $\mathcal{T}^{(1)}, \ldots, \mathcal{T}^{(B)}$ by **sampling with replacement** from $\mathcal{T}$. The bootstrapped datasets will have multiple copies of some data points and may be missing others, but many statistical properties are preserved. Assume for simplicity that the size of every bootstrapped dataset is equal to the size of the original dataset. **Algorithm 7.1** and **Example 7.2** step through the process.

Bagging (Method 7.1) creates B bootstrapped datasets and trains B (base) models, one on each dataset. For prediction, the predictions of these base models are combined together (ensemble). For regression or for class probabilities, the **output is the average** of the base model predictions (Example 7.1). For hard classification labels, we can take a **majority vote**.

To see how bagging (the ensemble) reduces the variance (of a base model), consider a collection of random variables $z_1, \ldots, z_B$. Let the mean and variance of these variables be μ and σ^2 and the average correlation between pairs of them be ρ. The mean and the variance of the average of these variables (e.g. the ensemble output) is given by Eqns. 7.2a and 7.2b. We can see that:

1. The mean is unaffected.

2. The variance decreases as B increases.
3. The less correlated the variables are, the smaller the variance.

Typically then, **the (test) error of the ensemble will decrease as the size of the ensemble, B is increased** (see Example 7.3, but I wonder if they used a random test point at each step to calculate these error values?! They do say the curve is an average of multiple runs).

Possible issues in practice:

- We can't directly control the correlation between base models, but we might try to encourage them somehow to be less correlated. The randomness from resampling will produce some differences between the base models.
- There could still be overfitting from the base models.
- Compared to using the original dataset, the bias of the base models might increase because of the bootstrapping.

Out-of-Bag Error Estimation

Bagging is another example of a technique that spends more computational effort to get improvements in models (or estimation): recall cross-validation as a way to get an estimate of E_{new} . If we produce an ensemble via bagging, each base model will have seen (on average) about 63% of the data points. We can use the other $\sim \frac{1}{3}$ of the points to build up an estimate of E_{new}, called the **out-of-bag error** E_{OOB} . Each data point gets used as a test point for $\sim \frac{B}{3}$ base models and we average those estimates over all of the data points. The (training) computation was already done, so no extra effort needed compared to cross-validation.

7.2 Random Forests

Random forests are bagged decision trees, but with an extra trick to try and make the base model less correlated (see point above). At each step when considering splitting a node, we only consider a (random) subset of $q < p$ variables to split on. This might increase the variance of each base model, but if the decrease in ρ is larger then we will still get a benefit. In practice people find that this is often the case. See Example 7.4. q is a hyperparameter, but rules of thumb are to use $\lfloor q \rfloor = \sqrt{p}$ for classification and $\lfloor q \rfloor = \frac{p}{3}$ for regression.

7.3 Boosting and AdaBoost

Boosting is another ensemble technique, but the focus is more on combining simple (i.e. likely high bias) base models together to **reduce the bias** of the ensemble.

The training of models in bagging is (embarassingly) parallel. Boosting on the other hand, constructs an ensemble sequentially. Each model is encouraged to try and focus on the mistakes made by the previous model(s). This is done by weighting the data points, during resampling and then during prediction (**Example 7.5**).

AdaBoost

To see how the general idea of boosting works, let's look at the first algorithm of this type (and assume binary classification): AdaBoost. Looking at the pseudocode (**Method 7.2**), we can see that:

- Each data point is given a weight parameter w_i, set initially to be equal.
- Another parameter, $\alpha^{(b)}$ is calculated on each iteration, using the training error (**line 5**). This parameter is then used to modify the weight values to be used in the next iteration (**line 6**). The weights are also renormalized (**line 7**).
- The $\alpha^{(b)}$ values are used to calculate the predictions of the ensemble. They can be viewed as the degree of confidence in the predictions made by each base model and are used during prediction.

Where do the expressions on lines 5 and 6 come from?

- Adaboost trains the ensemble by **minimising an exponential loss function** of the boosted classifier at each iteration (**Eqn. 7.5**). This loss function makes the maths work out nicely! The book steps through the derivation of this, but let's leave this as optional reading.
- Part of the derivation shows that we can do the optimisation using the weighted misclassification loss at each iteration (line 4).

Bagging and boosting are compared in **Example 7.6** (perhaps a bit unfairly because very simple models are used).

Design Choices for AdaBoost

The book gives a few bits of advice for choosing the base classifier and the number of iterations (B) for AdaBoost:

- Good idea to use a simple base model that's fast to train (e.g. decision stump or small decision tree), because boosting reduces bias efficiently.
- Overfitting is possible if B gets too large, so could use early stopping.

7.4 Gradient Boosting

AdaBoost uses an **exponential loss function** that can be sensitive to outliers, noise in the data, etc. One way to address this is to use a different loss function. But to get there, we need to think about the model a bit differently. If we take a general view of a model (aka function approximator) as an weighted sum combining together some other functions, we have an **additive model**:

$$f^{(B)}(\mathbf{x}) = \sum_{b=1}^{B} \alpha^{(b)} f^{(b)}(\mathbf{x})$$

Statisticians have explored such models for many decades. Boosting is clearly an example of this form of model, but linear regression, polynomial regression, two-layer neural networks and other models also fit.

In boosting:

- Each base model/basis function is itself a machine learning model, learnt from data.
- The overall model is learnt sequentially (B) iteration.

Training an additive model is an optimisation problem over $\{\alpha^{(b)}, f^{(b)}(\mathbf{x})\}_{b=1}^{B}$ to minimise **Eqn. 7.15**. This is done greedily at each step for the exponential loss function in AdaBoost. Alternatively, any method that will improve $J()$ for us would be fine (**Eqns. 7.16, 7.17**). How this is done is a little bit complex (so optional reading for the course), but in summary:

- The gradient of $J()$ with respect to the current base models $(b-1)$ is taken as the gradient of the loss function for the base models on the data (**Eqn. 7.18**).
- The b^{th} model is trained to minimise a difference between its predictions and the negative gradient. Then, adding this base model into the ensemble should move things in the direction of negative gradient and reduce $J()$. Since $\alpha^{(b)}$ is like a weight on this model, it is like a step size parameter (determined by a line search).

Gradient boosting is summarised in **Method 7.3** and an example shown in **Fig. 7.9**. Practical implementations of gradient boosting (using decision trees as base models) are often found to give state of the art performance (e.g. winning kaggle competitions), with implementations such as XGBoost and LightGBM.

Module 8: Non-linear Input Transformations and Kernels

8.1 Creating Features by Non-linear Input Transformations

Lindholm et al. recall an idea here that is very important for this Chapter: prior to learning our ML model, we can think about using one or more **basis functions** $\boldsymbol{\phi}(\mathbf{x})$ to our data. If these basis functions are non-linear, then we end up with a model that is a non-linear function of x, even if the ML model is linear with respect to it's parameters. Their example is with polynomial (in x!!!) regression (**Eqn. 8.2, 8.3** and **Figure 8.1**). Note that $\boldsymbol{\phi}(\mathbf{x})$ is a vector of basis functions. The number and type of basis functions is then something we have to choose.

Different books would describe this as **a mapping from the input space to some other feature space**. See Example 8.1. The labels on the axes of the right figure would then be e.g., z_1, z_2, z_3.

```
x = rand(20,1);
y = cos(x);
y = y + 0.1*randn(20,1);
figure;
plot(x,y,'.')
p1 = polyfit(x,y,1)
```

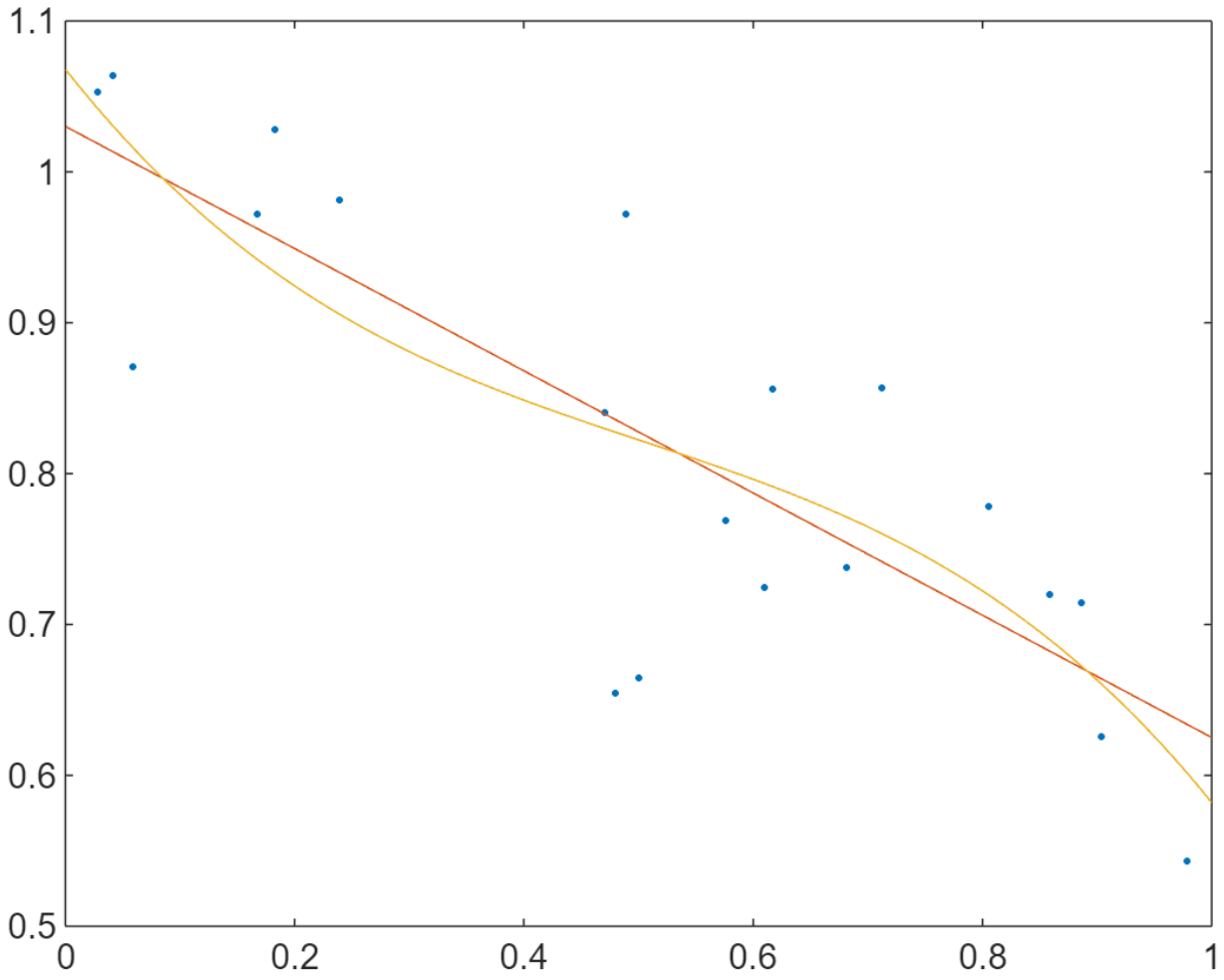
```
p1 = 1×2
   -0.4050    1.0302
```

```
x1 = linspace(0,1);
y1 = polyval(p1,x1);
hold on
plot(x1,y1)
xsqd = x.^2;
xcub = x.^3;
z = [x xsqd xcub];
%Doh! polyfit can only do regression with 1 input
```

```
b = ones(20,1);
z = [b z];
p3 = inv(z'*z)*z'*y;
p3
```

```
p3 = 4×1
    1.0680
   -0.9626
    1.4085
   -0.9319
```

```
y3 = zeros(1,100);
for i=1:100
y3(i) = p3(1) + p3(2)*x1(i) + p3(3)*(x1(i).^2) + p3(4)*(x1(i).^3);
end
plot(x1,y3);
```



8.2 Kernel Ridge Regression

The most important idea in this Chapter is the use of a so-called **kernel function**, $\kappa(\mathbf{x}, \mathbf{x}')$. The word "kernel" is generic and gets used in other areas of statistics and ML, but it's use in "kernel machines" is more specific.

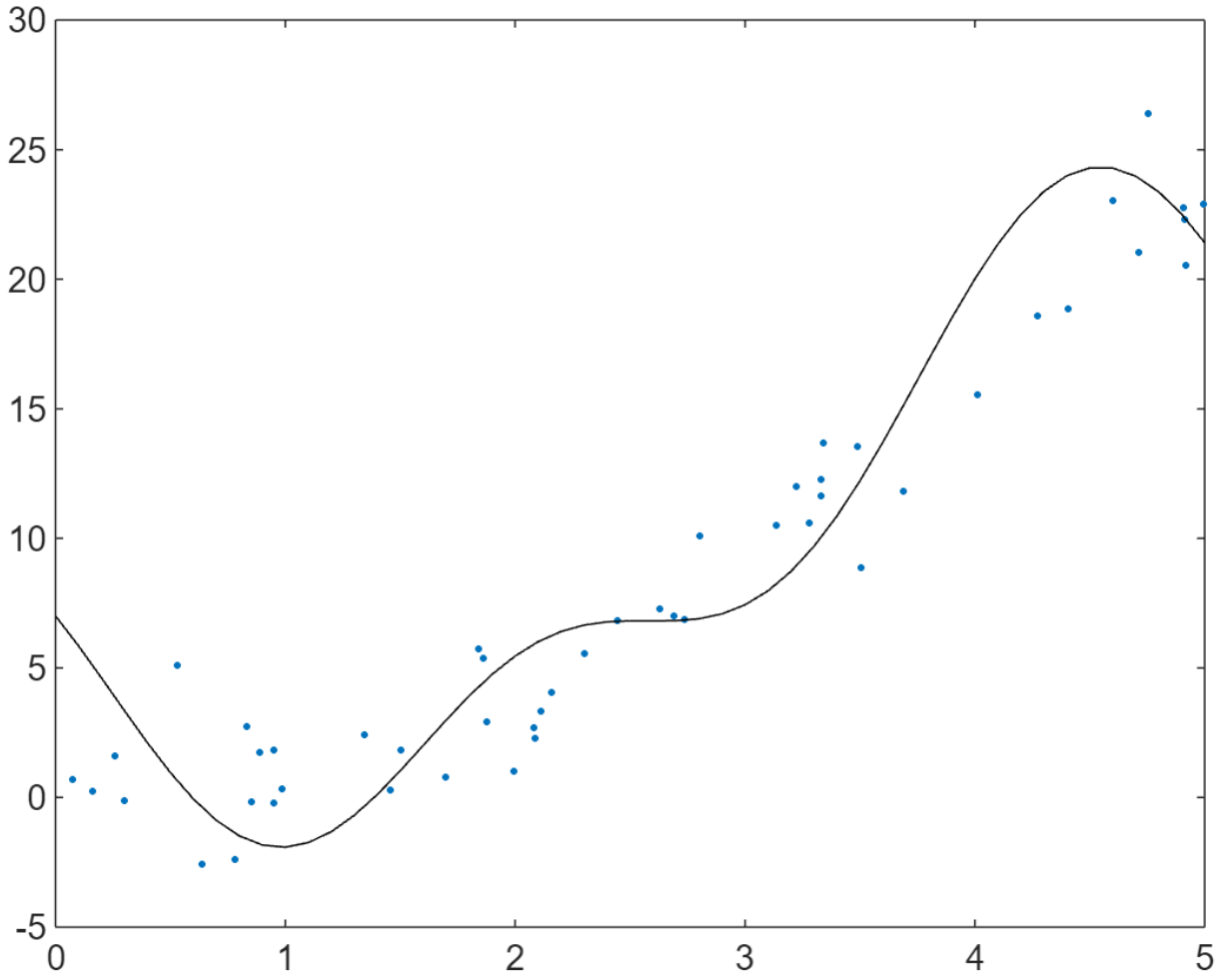
If we have a set of (non-linear) basis functions $\boldsymbol{\phi}(\mathbf{x})$, then a very useful kernel function is the **inner product between these for some pair of data points**:

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^\top \boldsymbol{\phi}(\mathbf{x}')$$

For a training set $\mathbf{X}$, the matrix $\mathbf{K}(\mathbf{X}, \mathbf{X})$ is the kernel function applied to all pairs of data points and $\mathbf{K}(\mathbf{X}, \mathbf{x}_\star)$ is a vector where we calculate the kernel function between a test point $\mathbf{x}_\star$ and every point in the training set. This bring us to kernel ridge regression, where our predictions are given by Eqn. 8.14b, with Eqn. 8.14a.

Let's try and implement kernel ridge regression directly from these equations!

```
clear *;
clf;
x = 5*rand(50,1);
y = x.^2;
y = y + 2*randn(50,1);
plot(x,y,'.')
n = 50;
lambda = 0.01;
l = 1;
d = pdist(x);
k = exp(-(d.^2)/(2*l^2));
K = squareform(k);
alpha = y'*inv(K + n*lambda*eye(50,50));
%Prediction
xtest = 0:0.1:5;
dtest = pdist2(x,xtest');
ktest = exp(-(dtest.^2)/(2*l^2));
ypred = alpha*ktest;
hold on;
plot(xtest,ypred,'k')
```



Ok! So kernel ridge regression looks like it predicts based on the value of the kernel function between a test point and each of the training points, weighted by the $\widehat{\alpha}$ terms from Eqn. 8.14a. We can then see that we get $\hat{\boldsymbol{\alpha}}$ by adding "a bit of" identity matrix to $\mathbf{K}(\mathbf{X}, \mathbf{X})$, inverting that and multiplying by the target values from the training set, $\mathbf{y}^\top$. So that's do-able. But now, why does this make sense and where does it come from? This is what Lindholm et al. are explaining on p.192-193:

- Start by writing down the loss function (and it's solution, the prediction function) for linear regression (with L^2 regularisation) in terms of the transformed data $\boldsymbol{\phi}(\mathbf{x})$
- Use some matrix algebra to realise that we can rewrite the solution in such a way that $\boldsymbol{\phi}(\mathbf{x})$ only appears as inner products $\boldsymbol{\phi}(\mathbf{x})^\top \boldsymbol{\phi}(\mathbf{x}')$.
- Oh, but that's our kernel function! So we can use the kernel function and we will be using the same model, loss function and prediction function.

Note that when we write linear regression in the previous form (Eqn. 8.5), we have a d-dimensional parameter vector, $\widehat{\theta}$, no matter how big n is (size of the training set). But now, we compute the kernel function for all pairs

of data points, so the size of our "model" is in terms of n, not d. In fact, we can use kernels that allow $d \to \infty$. Oh boy.

Example 8.2 is a tidier version of the code above (I used the squared exponential kernel function). A kernel function usually has one or more hyperparmeters - this example compares the **polynomial kernel** with the **squared exponential** with different hyperparameter values.

8.3 Support Vector Regression

The is some interesting foundational stuff here but we can skip over this from the point of view of using the techniques. In that spirit:

- Support Vector Regression means changing the loss function from that used in kernel ridge regression to the ϵ-insensitive loss (**Eqn. 8.17**). This no longer has a closed-form solution, so we need a numerical optimiser. However the problem is a convex, constrained optimisation problem so there are likely to be fewer issues then, e.g. with neural network training.
- Optimising this loss function leads to (many) of the α_i values **becoming (exactly) zero**. This means that in the end, the trained model **only depends on a small number of the data points** to make it's predictions. See also **Example 8.3**.

8.4 Kernel Theory

It turns out that several ML models can be rewritten (kernalised?) as kernel methods. Lindholm et al. show this for k-NN by writing the (squared) Euclidean distance in terms of a linear kernel, $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$.

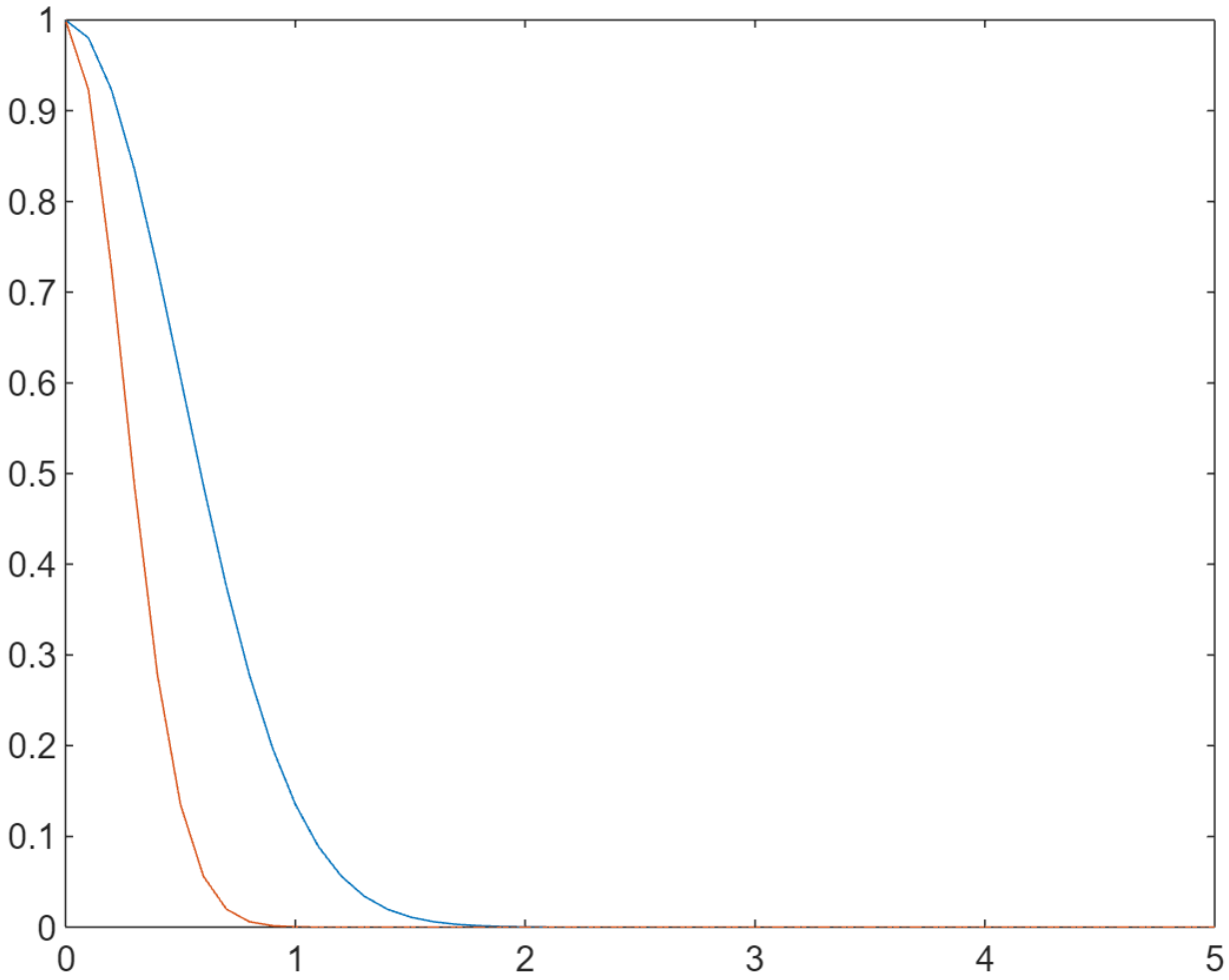
One of the main reasons that kernel methods became widely-used in machine learning is that they allow you to apply ML techniques to data where **Euclidean distance cannot be defined** - e.g. text snippets (Example 8.4).

The Meaning of a Kernel

The **three bullet points** that Lindholm et al. make here are worth reading a couple of times.

There is a lot of freedom in the choice of a kernel function. Many kernel functions that are commonly used are positive semidefinite, but for practical purposes this doesn't matter (except for kernel ridge regression). A number of examples functions are discussed in the book.

```
ddd = 0:0.1:5;
l = 0.5;
kddd = exp(-(ddd.^2)/(2*l^2));
figure;
plot(ddd,kddd);
hold on
l = 0.25;
kddd = exp(-(ddd.^2)/(2*l^2));
plot(ddd,kddd)
```



8.5 Support Vector Classification

Again, let's focus on understanding from the practical point of view:

- For binary classification, we saw in Chapter 5 the concept of a margin for logistic regression.
- If we use the hinge loss function for logistic regression (with transformed inputs), we end up with a training problem that is a convex constrained optimisation problem. When this is solved, the solution again tends to be sparse (many $\alpha_i = 0$) .
- In fact, when the margin for a data point i is > 1 (typo in the book?), then $\alpha_i = 0$. So **these data points (in the end) don't define the decision boundary**.
- When the margin for data point i is ≤ 1, the point is **inside the margin or on the wrong side of the decision boundary**.

Note that $\lambda(> 0)$ is a **regularization parameter** that penalizes points that are inside the margin. In ML libraries you will often find this parameter is there as $C = \frac{1}{2\lambda}$ or possibly even ν. In any case, it's a hyperparameter that needs to be determined (e.g. via cross-validation).

Example 8.5 compares different kernels and values of λ. **Figure 8.7** doesn't really bring out the power of SVMs - try playing with this demo instead: https://cs.stanford.edu/people/karpathy/svmjs/demo/

Module 9: The Bayesian Approach and Gaussian Processes

9.1 The Bayesian Idea

"The statistical methodology of Bayesian learning is distiguished by its use of probability to express all forms of uncertainty. Learning and other forms of inference can then be performed by what are in theory simple applications of the rules of probability. The results of Bayesian learning are expressed in terms of a probability distribution over all unknown quantities. In general these probabilities can be interpreted only as expressions of our degree of belief in the various possibilities." -Radford Neal.

Our core problem in ML: given some data, build a model of the unknown entity that generated the data.

Assume our model has parameters θ. We have seen that maximum likelihood estimation provides a principled way to learn θ from data:

$$L(\theta|\mathcal{X}) \propto p(\mathcal{X}|\theta) = \prod_{i=1}^{n} p(\mathbf{x}_i|\theta)$$

But why should we be **absolutely certain about our model parameters**?

Bayesian learning works with a **probability distribution over model parameters** that expresses our beliefs regarding how likely different parameter values are.

We start by defining a **prior** distribution, $p(\theta)$, expressing our belief about the model parameters before we have seen any data.

After seeing some data, we update our prior to a **posterior** distribution using Bayes rule:

$$p(\theta|\mathcal{X}) = \frac{p(x|\theta)p(\theta)}{p(\mathcal{X})} \propto L(\theta|\mathcal{X})p(\theta)$$

That is, posterior $\propto$ likelihood $\times$ prior.

This is the proper way to combine our prior knowledge with the information obtained from the data!

So then to make predictions about a new data point, x_* a Bayesian (ideally) uses

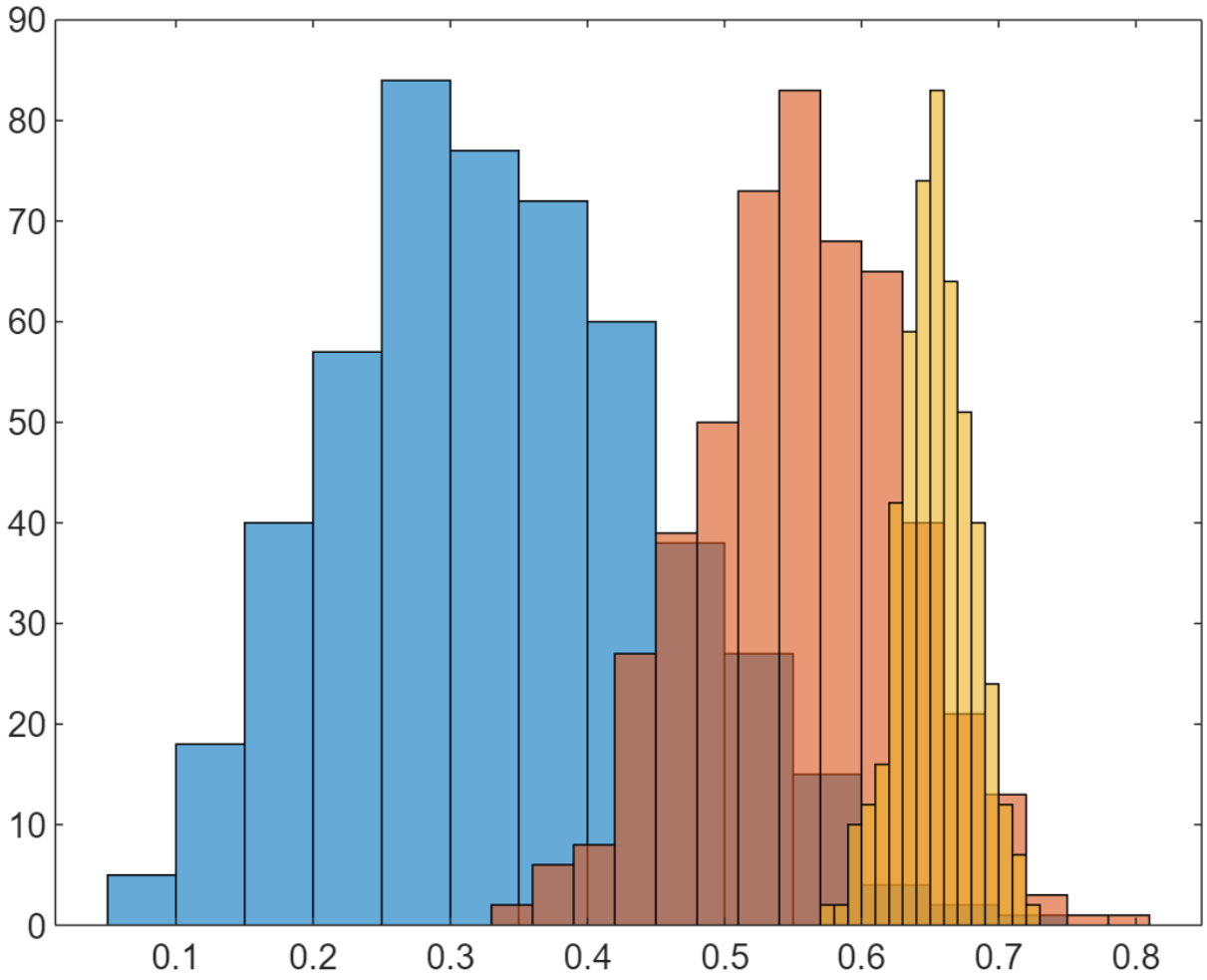
$$p(x_*|\mathcal{X}) = \int p(x_*|\theta)p(\theta|\mathcal{X})d\theta$$

the predictive distribution for x_*. So we **use all θ weighted by our model posterior** (i.e. we average over our uncertainty in estimating θ).

In practice the integral above might not be easy to solve. If you are fully Bayesian, you would use, e.g. a Markov-Chain Monte Carlo (MCMC) method to sample from this. Alternatively, the **maximum a posteriori (MAP)** estimate is often used, which is the mode of the posterior. If the prior is uniform over all θ, then the MAP estimate is equal to the maximum likelihood estimate.

Example (from Alpaydin, 16.2.2): you observe a number of coin tosses (aka a binary outcome) and want to use Bayesian inference. You naturally use a Bernoulli distribution, so you need to infer the parameter, q (i.e. probability of "H" or 1 - probability of "T"). Alpaydin shows that both the prior and posterior are Beta distributions with different parameter values.

```
clf
%Specify the prior
alpha = 5;
beta = 10;
%Check it out via a sample of 500 points
xprior = betarnd(alpha,beta,500,1);
histogram(xprior)
hold on
%Specify a posterior
A1 = 20;
N1 = 30;
%Check it out via a sample of 500 points
xpost1 = betarnd(A1+alpha-1,N1-A1+beta-1,500,1);
histogram(xpost1)
%Specify a different posterior
A2 = 200;
N2 = 300;
%Check it out via a sample of 500 points
xpost2 = betarnd(A2+alpha-1,N2-A2+beta-1,500,1);
histogram(xpost2)
```



9.2 Bayesian Linear Regression

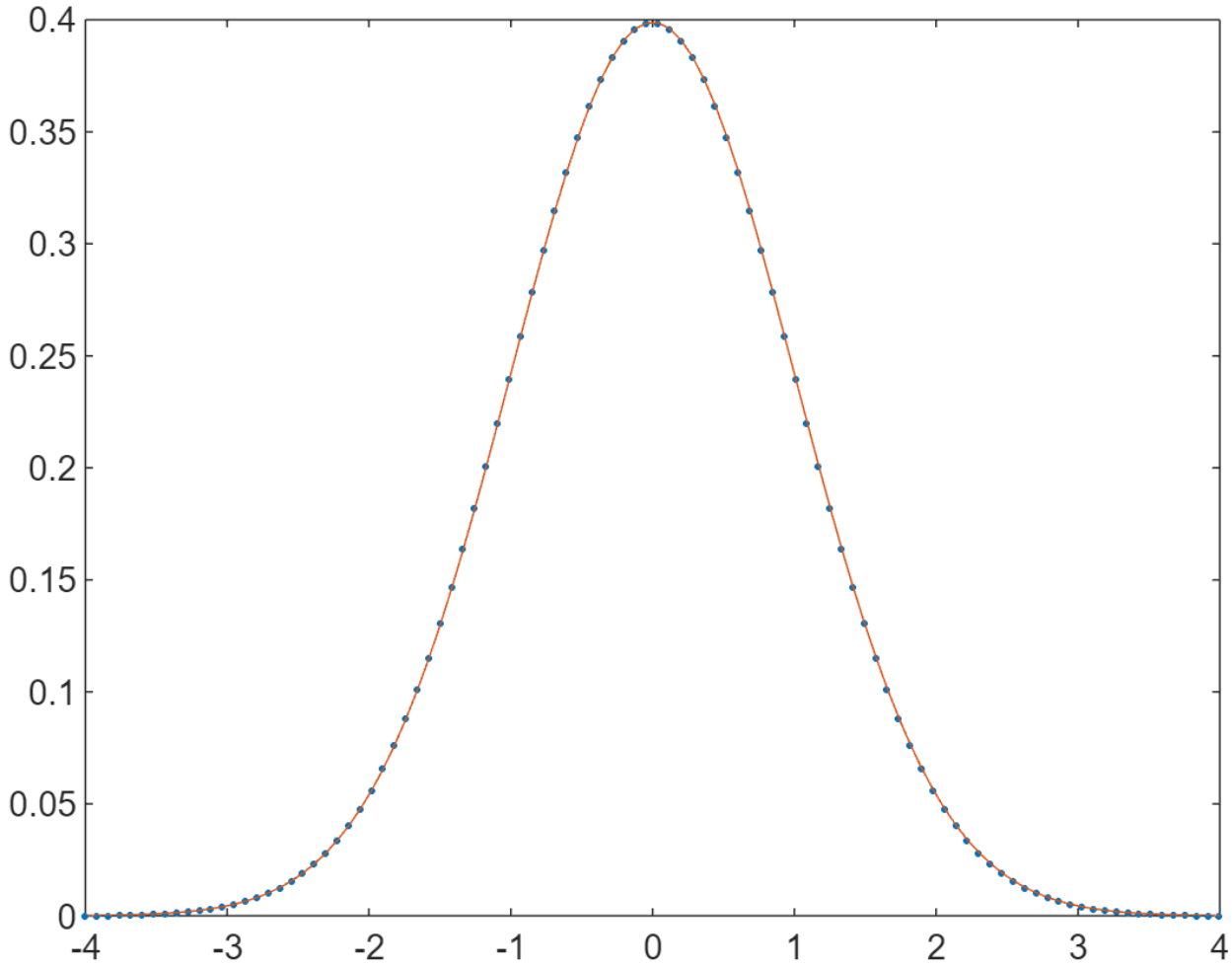
First, let's have a look at how Bayesian inference works for linear regression (where the posterior is simple and easy to calculate).

The Multivariate Gaussian Distribution

We need multivariate probability distributions, but we can pretty much get by with one type: a multivariate Gaussian (aka Normal). The univariate Gaussian is given by:

$$p(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{z-\mu}{\sigma}\right)^2}$$

```
figure
x = linspace(-4,4,100);
y = normpdf(x);
plot(x,y,'.')
hold on
plot(x,y)
```



If $\mathbf{z}$ is a q-dimensional random vector, then a multivariate Gaussian is parameterised by a mean vector, $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ (see book for notation before and up to **Eqn. 9.5**). Gaussian distributions have lots of useful mathematical properties and are a good model for real-world (continuous) data in lots of situations (for one reason, because of the central limit theorem).

Linear Regression with the Bayesian Approach

Recall (**Chapter 3**) that our setup for linear regression included additive (Gaussian) noise (**Eqn. 9.6**). So we can write the predictive distribution as a Gaussian, for a single point (**Eqn. 9.7**) or for multiple points (**Eqn. 9.8**). The i.i.d assumption means that this multivariate Gaussian has a **diagonal covariance matrix** (covariances are all zero).

We want to use **Bayesian inference**, so we need to start by specifying a prior for our model parameters, $p(\theta)$. Let's choose this to be a Gaussian as well! (**Eqn. 9.9**). Bayes' rule can then be used to calculate the posterior distribution for the model parameters (**Eqns. 9.10**). From this, the posterior predictive distribution is given in **Eqns. 9.11** and (adding the noise) **Eqn. 9.12**.

The key thing you should understand here is that we can calculate all of the things in these equations (given some data) and therefore we can do linear regression (Method 9.1).

The additive noise is on the target values of the data and it's standard deviation is σ. The variance of the prior distribution is σ_0. These (different!) hyperparameters can be tuned by maximizing (an optimization problem) the marginal likelihood $p(\mathbf{y})$ (**Eqn. 9.12**).

See **Examples 9.1 and 9.2**.

One of the nice things about Bayesian inference is that we get a predictive distribution, that captures precisely the uncertainty of our predictions, given the prior and model.

Connection to Regularised Linear Regression

Previously we look at adding L^2 regularisation to linear regression. When we train by optimising this loss function, we get a point estimate for the model parameters, $\widehat{\boldsymbol{\theta}}_{L^2}$. For the Bayesian linear regression model we just looked at (with prior **Eqn. 9.9**), it turns out that $\widehat{\boldsymbol{\theta}}_{L^2} = \widehat{\boldsymbol{\theta}}_{MAP}$, which is **the peak of the posterior distribution**: **the maximum a posteriori (MAP) solution**. This is interesting because it shows that the Bayesian approach is doing something like regularisation (making it less prone to overfitting). From the other direction, you could say that adding regularisation to a model is like implicitly choosing a prior distribution, but you don't get the posterior distribution unless you "go Bayesian"! :)

9.3 The Gaussian Process

The book focuses first on showing why the mathematics underlying Gaussian Processes works. Let's simplify and instead try to focus on the practical aspects of GPs (but the Figures are still useful to look at!).

Gaussian Processes (GPs) have a nice Bayesian interpretation. Consider again the regression problem. A GP **specifies a prior distribution directly over the function space and does inference with it given some (supervised) training data**.

Stochastic Process: a collection of random variables $\{Y(x), x \in X\}$ indexed by a set X.

A stochastic process is specified by giving the joint probability distribution for every finite subset of variables $Y(x_1, \ldots, x_k)$ in a consistent manner.

For a GP, every subset of data points has a multivariate Gaussian distribution. To achieve this, we need a mean function and a covariance function, which fully specify our GP. The covariance function $\kappa(\mathbf{x}, \mathbf{x}')$ represents our prior belief about how we think the value of the function at $\mathbf{x}$ is affected by the value of the function at $\mathbf{x}'$ (and vice-versa).

Figures 9.3 - 9.5 show how the marginal distribution for one variable is determined by a value observed for another variable. The GP is a tractable way of extending this idea of slicing up Gaussians (one for each data point) to an infinite number of points (using a function).

We end up with a posterior predictive distribution which is (of course) Gaussian.

Again, the key thing you should understand here is that we can calculate the things in Equations 9.25 (given some data) and therefore we implement a Gaussian Process (Method 9.2).

We start with a prior over the function space. Once some data is observed, we update our prior to get a posterior. Our belief about the possible functions has changed! The predictive variance has reduced considerably, especially close to where we observe data points (**Figure. 9.9**).

Drawing Samples from a Gaussian Process

The posterior predictive distribution for a single point is a univariate Gaussian. But over the whole input space, it is a distribution over functions. One common way of showing this is to plot a few sample functions from this distribution (e.g. **Figure 9.10**). This isn't hard to do: take a (test) set of data points, calculate the resulting mean vector and covariance matrix (using $\kappa()$). A sample of n points is equivalent to an n-dimensional multivariate Gaussian - we then "slice" this up and plot the values "side by side".

9.4 Practical Aspects of the Gaussian Process

To use a GP, we have to choose a covariance/kernel function. This will have a big impact on model, resulting preduction and performance. Kernel functions will have their own hyperparmeters (See **Figure 9.11** for a comparison in 1D). ML researchers tend to use a couple of general-purpose covariance functions, but there have been lots of variations developed for specific applications and other data settings (e.g. nonstationary kernels that are dependent on the specific position in the input space).

It is possible to optimise the hyperparameters of a covariance function in a principled way by **maximizing the (marginal) likelihood** (**Eqn. 9.27**). **Figures 9.12** and **9.13** show some examples of this. This is **non-convex optimisation problem but an expression for the gradient is available, so gradient-based methods are typically used**. See **Example 9.3** for fitting a GP to the car stopping data.

Lindholm et al. have a link to a cool GP demo in their book: http://smlbook.org/GP/index.html . This one is also very cool: http://chifeng.scripts.mit.edu/stuff/gp-demo/ .

Module 10: Generative Models and Learning from Unlabelled Data

There is quite a lot going on in Chapter 10 of Lindholm et al.! Fortunately most of the techniques are fairly easy to get a working understanding of. There is a mountain of knowledge underneath them that we won't go into in the course, but the book gives some references.

Generative models are all the rage right now in ML/AI (e.g. generative images, large language models).

Unsupervised learning is big enough that an entire course could be devoted to it.

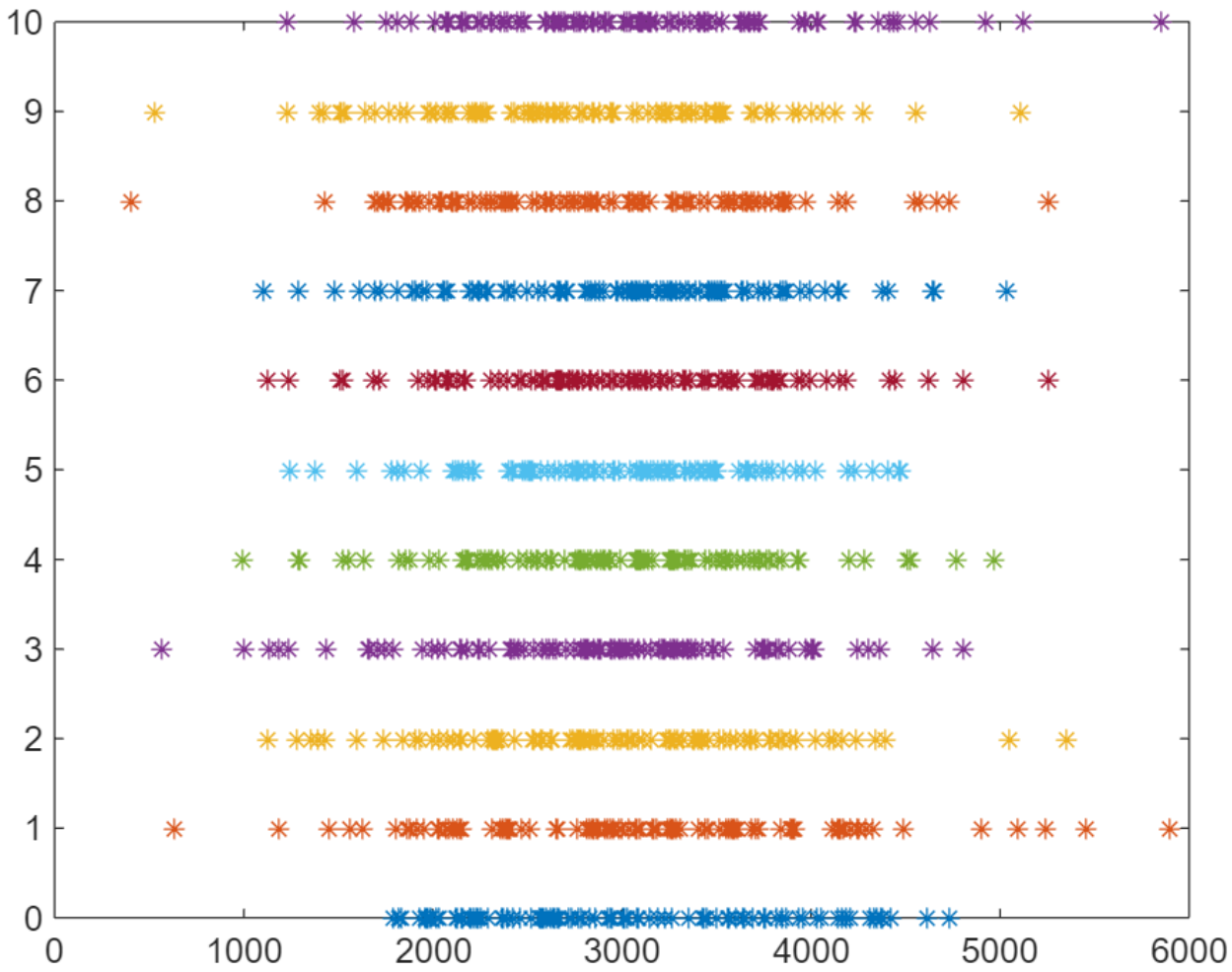
Previous chapters have been mostly about supervised learning. Probabilistically, these are models of $p(y|\mathbf{x})$. Generative models build a model of $p(\mathbf{x}, y)$ (which is short for $p(\mathbf{x}, y, \boldsymbol{\theta})$. You could also drop the y and then you would have unsupervised learning!

Generative models sound fancy but probabilistic models have always had this property. This is an important component of things like computer simulation models and lots more!

So we can use **probability density estimation** and think of this as a foundation for unsupervised learning and generative models.

```
%Use some example data in Matlab
```

```
load carsmall
%One of the features is weight - build a Gaussian model of p(x) using MLE!
m1 = mean(Weight);
s1 = std(Weight);
%Plot the data
plot(Weight,zeros(100,1),'*');
hold on
%Use our generative model to generate new data from p(x)
for i=1:10
w1 = m1 + s1*randn(100,1);
plot(w1,i*ones(100,1),'*');
end
```



10.1 The Gaussian Mixture Model and Discriminant Analysis

Let's step back from the book for a moment and focus on what a mixture model is.

Mixture Densities

A **mixture model** is a weighted sum of **component** densities:

$$p(\mathbf{x}) = \sum_{i-1}^{k} p(\mathbf{x}|G_i)P(G_i)$$

where G_i is the ith component (aka group or cluster) assumed to be in the data, $P(G_i)$ is the prior/mixture coefficient/weight of the ith component and $p(\mathbf{x}|G_i)$ is the ith component pdf. Note that $0 \leq P(G_i) \leq 1, \forall i$ and $\sum_{i=1}^{k} P(G_i) = 1$.

We will look at Gaussian mixture models (GMMs), so $p(\mathbf{x}|G_i) = \mathcal{N}(\boldsymbol{\mu}_i, \Sigma_i)$.

The parameters of our GMM are $\boldsymbol{\mu}_i, \Sigma_i, P(G_i), i = 1, \ldots, k$. So the LHS of the above equation is more properly written as $p(\mathbf{x}|\theta) \equiv p(\mathbf{x}|\boldsymbol{\mu}_i, \Sigma_i, P(G_i))$

Pics from [Bishop]:

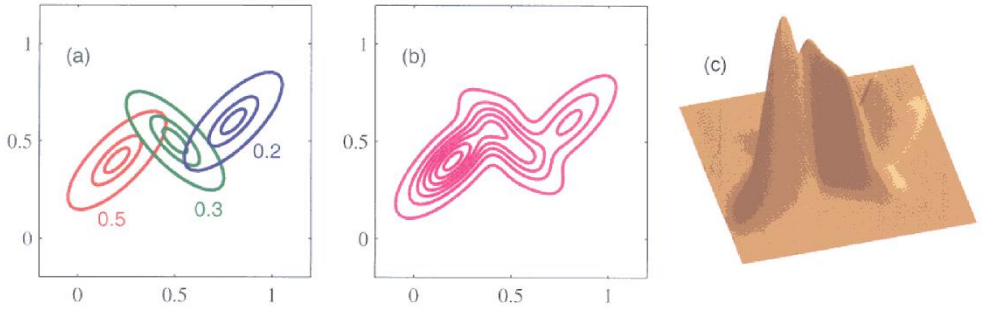


Figure 2.23 Illustration of a mixture of 3 Gaussians in a two-dimensional space. (a) Contours of constant density for each of the mixture components, in which the 3 components are denoted red, blue and green, and the values of the mixing coefficients are shown below each component. (b) Contours of the marginal probability density $p(\mathbf{x})$ of the mixture distribution. (c) A surface plot of the distribution $p(\mathbf{x})$.

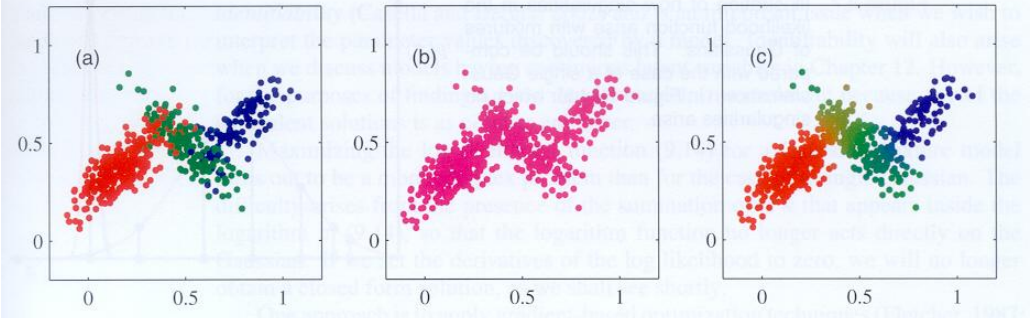


Figure 9.5 Example of 500 points drawn from the mixture of 3 Gaussians shown in Figure 2.23. (a) Samples from the joint distribution $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ in which the three states of $\mathbf{z}$, corresponding to the three components of the mixture, are depicted in red, green, and blue, and (b) the corresponding samples from the marginal distribution $p(\mathbf{x})$, which is obtained by simply ignoring the values of $\mathbf{z}$ and just plotting the $\mathbf{x}$ values. The data set in (a) is said to be *complete*, whereas that in (b) is *incomplete*. (c) The same samples in which the colours represent the value of the responsibilities $\gamma(z_{nk})$ associated with data point $\mathbf{x}_n$, obtained by plotting the corresponding point using proportions of red, blue, and green ink given by $\gamma(z_{nk})$ for $k = 1, 2, 3$, respectively

What happens next is really all about whether or not you know *apriori* whether your data comes from some groups/categories/classes and if you have those "labels" for your data.

Lindholm et al. start by assuming you know the labels - e.g. **Figure 10.1** where $y \in \{red, blue\}$. We would then use a discrete (categorical) distribution for y .

Supervised Learning of the Gaussian Mixture Model

They make things sound rather complicate here but it's easy: if we have labelled data, we fit a Gaussian distribution to the data from each class (MLE using sample mean and covariance matrix). The priors/mixture coefficients are estimated from the frequencies of each class in the data (again the MLE).

Predicting Output Labels for New Inputs: Discriminant Analysis

This section shows that you can use a GMM to do classification, because you can use your model of $p(\mathbf{x}, y)$ to get $p(y|\mathbf{x})$. This is a nice illustration for multiclass classification, but you could just use a single Gaussian distribution for the binary classification problem and you be close to logistic regression. Key differences:

- Priors
- Different parameterisations of the covariance matrix give rise to linear discriminant analysis (LDA - overloaded acronym!!!) v's quadratic discriminant analysis (QDA).

The equations should be understandable and **Figures 10.2-4** are informative.

Semi-supervised Learning of the Gaussian Mixture Model

Semi-supervised learning has received a lot of attention in the recent ML literature. This is where we have some data this is labelled but also (probably a lot more) that is unlabelled. We would like to make use of the unlabelled data as well as the labelled data.

We would like to maximize the likelihood but we can't do this directly using the unlabelled data (Eqn. 10.9), however we can take the following approach (**Method 10.2**):

1. Learn the GMM from the labelled data
2. Use the GMM to predict $p(y = m|\mathbf{x}_i, \widehat{\boldsymbol{\theta}})$ (as a proxy for) the labels for the unlabelled data
3. Update the GMM's parameters including the data with predicted labels

This works (**Figure 10.6**) and isn't just some random idea - it's a version of the **Expectation Maximisation (EM) algorithm** (see more below).

10.2 Cluster Analysis

If we do not have a defined target variable (labels or numerical values), we are in the realm of unsupervised learning. The data consists of observations of features collected from some problem domain, and presumably contains **structure and information** that is of interest. One way to learn about that stucture is to **build a model of the distribution of the data - i.e., probability density estimation**. A GMM is a flexible probability density estimator, and the peaks of the density give us an indication of clustering in the data (places in the feature space where data occurs with high probability).

The EM algorithm can be used to fit a GMM to data for $\mathbf{x}$ (in fact this is most common usage). Effectively, y is **marginalised** out of the estimate of $p(\mathbf{x}, y)$ and becomes a **latent variable** for the model. The algorithm is shown in **Method 10.3**. Key steps:

- E step (line 3): calculate the "**responsibility** values". How likely is it that a given data point could have been generated by each of the components in the mixture model? (Note: the notation here is a bit confusing: y is actually a vector of values for each mixture component.
- M step (line 4): update the model parameters $\boldsymbol{\theta}$ using the responsibility values computed in the E step. Lindholm refer to the responsibility values as **weights**, $w_i(m)$.

Maximising the likelihood for a GMM is a non-convex optimisation problem, and the EM algorithm performs local optimisation, hopefully converging towards a stationary point. This works pretty well most of the time, though there is also a degeneracy in the problem: undesirable solutions where the likelihood diverges towards infinity!

The result in **Figure 10.7** (unsupervised) looks almost identical to **Figure 10.6** (semi-supervised) and **Figure 10.1** (supervised), but this would not always be the case.

k-Means Clustering

We can understand the k-means clustering algorithm as a **simplified version of fitting a GMM using the EM algorithm**. Refer to steps (i)-(iii) on p.265. The simplifications are:

1. Rather than calculating (**soft**, probabilistic) responsibility values, k-means calculates "**hard**" responsibility values based on the distances between a point and a set of k cluster centers. It's a bit funny how Lindholm et al. insist on using M for k to keep their notation matching!
2. The distances in the GMM use the covariance information, i.e. the Mahalanobis distance between each data point and each component of the GMM.

Choosing the Number of Clusters

This is the number 1 question on the FAQ list! Unfortunately it doesn't have a simple answer. Sometimes problem domain knowledge can guide the choice of k. Lindholm discuss some ways that k might be chosen automatically (see, e.g. **Figure 10.9**).

10.3 Deep Generative Models

Invertible non-Gaussian Models and Normalising Flows

[This subsection optional reading!]

Generative Adversarial Networks

Consider the task of modelling a complex, high-dimensional probability distribution, $p(\mathbf{x})$. The approach here is to build an ML model (e.g. a deep neural network) that can transform a data point $\mathbf{z}$ into $\mathbf{x}$. If $\mathbf{z}$ corresponds to a sample from a simple distribution like a Gaussian, the network could then be used to transform this into a sample from the complex probability distribution $p(\mathbf{x}|\boldsymbol{\theta})$. We could then use this approach to generate new data for some interesting problem (e.g. images). So, how can we learn such a network from data?

The basic idea of Generative Adversarial Networks (GANs) is to compare the training data with synthetic samples generated from the model. If we can iteratively modify the model so that the synthetic images are more similar to the training data, we will get closer to our goal. This is done via steps of a "game" (i)-(iii) on p.273. We need a **critic**, who's job it is to determine whether or not a data point is real or synthetic. The learning in the generative model then aims to try and confuse the critic!

The critic is another ML model (a classifier), which takes as input a data point and predicts the probability that it is synthetic. The labels come from the game above and the critic is trained to **minimise** the loss function.

Finally, the parameters of the generator, θ are updated, but to **maximise** the loss function (Eqn. 10.28). The two updates alternate in a **minimax**, **adversarial** type of algorithm (**Method 10.4**).

Some demos/examples:

https://poloclub.github.io/ganlab/

https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/

10.4 Representation Learning and Dimensionality Reduction

Consider the so-called Swiss roll data:

https://scikit-learn.org/stable/auto_examples/manifold/plot_swissroll.html#sphx-glr-auto-examples-manifold-plot-swissroll-py

The data is 3D, but we can see that it actually lies on a 2D manifold that is embedded in this 3D space. If we knew (or could learn) the structure of the manifold, this would be valuable information that could be used for problems related to this data. For example, consider trying to predict the colour value of data points. If we could "unroll" the data, linear regression (in the 2D space) could do this easily.

Dimensionality reduction in ML is all about trying to learn a useful representation of high-dimensional data. A nice way to think about this is learning structure among the columns (features) of a dataset, compared to clustering which learns about structure among the rows (data points).

Autoencoders

An autoencoder is a neural network that can be trained to perform dimensionality reduction, from unsupervised data, but using a supervised learning algorithm (**Figure 10.10**). We can see from the structure that the network tries to learn an identity mapping, so the target output for some input vector is the input vector itself! This would be trivial, except that the hidden layers of the network have different numbers of units, typically fewer units than the number of inputs. This creates a **bottleneck**. For the network to perform well, it has to **learn a good internal/latent representation of the data in a space of lower dimensionality**.

Autoencoders typically have this symmetrical structure and can be trained with backpropagation/gradient descent. The first half of the network is called the **encoder** and the second half the **decoder**. The decoder can be used as a generative model and this idea has been used to develop several popular techniques in deep generative models.

Demo:

https://cs.stanford.edu/people/karpathy/convnetjs/demo/autoencoder.html

Principal Component Analysis

PCA is surely the most commonly-used technique for dimensionality reduction. PCA can be thought of as performing a **rotation of the coordinate system of the data space, to align the coordinate axes with the directions of greatest variance in the data**. Dimensionality reduction can then be done by **projecting the**

data onto a subspace of the new coordinate system (**Figure 10.11**). The solution of PCA gives us the basis vectors of the new coordinate system, together with a measure of the amount of variance (aka the **principal values**) that is captured by each of the basis vectors (aka the **principal components**).

Lindholm et al. take the tougher road here, explaining how PCA can be done via a **singular value decomposition** of the data matrix. This is indeed the most efficient way, however a simpler approach is to calculate the **eigenvectors and eigenvalues of the covariance matrix of the data**, which are the PCs and PVs. Either way, the learning/optimisation problem here is simple - closed-form solution, no iterative algorithm required.

Each PC is a linear combination of the original features, so PCA is **highly interpretable** in terms of what it is doing. The PVs also provide useful information - it is common to plot them to see how much (variance) information is being lost by choosing to reduce the dimensionality (**scree plot**). An example (from Alpaydin, 2014):

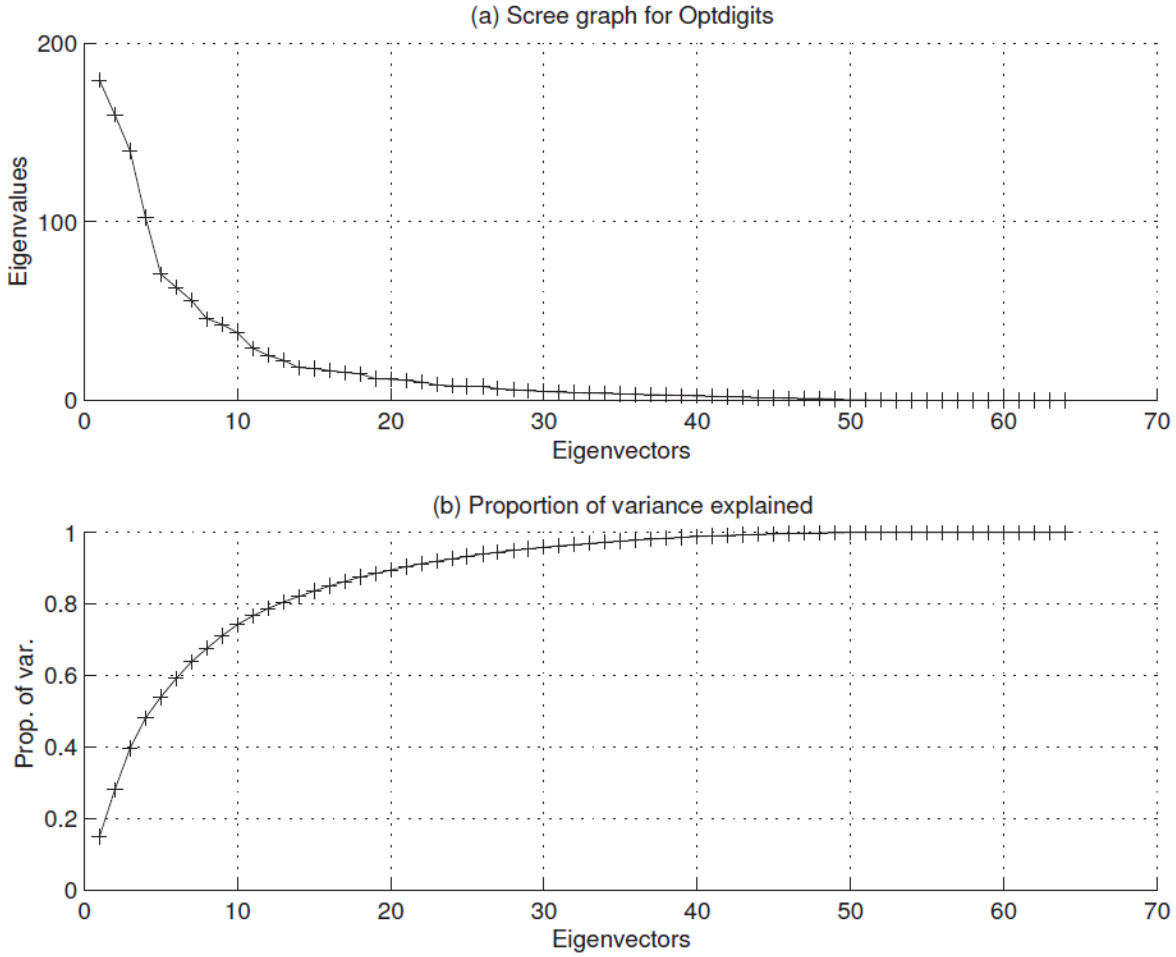


Figure 6.4 (a) Scree graph. (b) Proportion of variance explained is given for the Optdigits dataset from the UCI Repository. This is a handwritten digit dataset with ten classes and sixty-four dimensional inputs. The first twenty eigenvectors explain 90 percent of the variance.

Note: we need to **center the data** (subtract the mean from each column) before performing PCA. Routines from software libraries are likely to do this automatically.

Module 11: User Aspects of Machine Learning

This chapter gives some general practical advice about the application (engineering) of machine learning. It's a mostly non-technical read and it is very useful stuff, but we will be fairly brief in covering it.

Key Points:

- It is important to **handle data carefully** (training/validation/test splits), unintended relationships in datasets, variations over collection/observation time, etc.
- Trying to summarize performance in **a single number is at best a huge loss of information** (cf. compression).
- Establishing **baselines and bounding possible performance** is incredibly important (but often not done).
- Try **simple** things first (Occam's razor).
- Debugging can be tricky.
- Training error/generalisation gap stuff (recall Chap.4)
- Error Analysis (this is a great subsection!): evaluating an ML model should not be just calculating an accuracy rate. **We should look at which data points it is getting incorrect**. Is it something about the data (e.g. lighting in images)? In a real world application, working with a domain expert, this would be an iterative, interactive process.
- Getting more data is always good, but if we can't: maybe add some slightly different data; **data augmentation**; **transfer learning**; learning from unlabelled data.
- **Outliers** are somehow unusual and they might be useless (e.g. noise, incorrect measurement), but they could also be important to the problem. Generic outlier removal is a dangerous thing to do: you are changing the distribution of the data in an unprincipled way.
- **Missing data**: discarding rows, imputation.
- **Feature selection**: connection to L1 regularisation, correlations, PCA.
- Can I trust my ML model? Understanding why a prediction was made, transparency.
- Worst case guarantees: individual bad predictions for a class, etc.

Module 12: Ethics in Machine Learning

With the dramatic progress in AI and ML in recent years, ethical issues have (rightly) become a major concern. People are worried about the potential for misuse and harm from the use of AI. As a computer scientist, I am acutely aware of my lack of expertise in this area! I think **we are going to need cross-disciplinary efforts to work on these issues. We don't have all the answers!**

This Chapter of the book does not try and cover all ethical issues but gives some examples. Again, highly recommended reading. Here are a few of my thoughts on this Chapter:

- If your data has different groups in it (e.g. of people), then your ML model can give different performance on these groups. Things like false positive and false negative become serious when we start talking about serious things, like decision about how people are treated.
- The author shows that unless your data is perfectly balanced (in every way that matters!), it is impossible to create ML models that fulfil all desirable fairness criteria.
- **Misleading claims about performance** are a huge problem in ML. Research (especially when done by companies) becomes advertising and there are forces that push things towards overstating positive results and hiding negative ones.
- There is a nice subsection about explaining models in an understandable way. It focusses on AUC/ROC and makes the point that results should always be described in terms relevant to the specific application.
- The stochastic parrot analogy: predictions from an ML model can be viewed as repeating back the training data, plus some noise due to the model limitations (perhaps biases?); the ML model does not understand the problem and so it can't know when it is repeating something incorrect, out of context, or socially inappropriate.
- There is a bit of discussion here about language models (note the ref. to GPT-3). This was written before ChatGPT (3.5) and so already feels a bit dated.
- I think the stochastic parrot analogy is useful, but eventually we get to some fundamental issues. What actually is creativity? What does it mean to say that you understand something?