1 Introduction

1.1 What is machine learning?

A popular definition of **machine learning** or **ML**, due to Tom Mitchell [Mit97], is as follows:

> A computer program is said to learn from experience E with respect to some class of tasks T, and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Thus there are many different kinds of machine learning, depending on the nature of the tasks T we wish the system to learn, the nature of the performance measure P we use to evaluate the system, and the nature of the training signal or experience E we give it.

In this book, we will cover the most common types of ML, but from a **probabilistic perspective**. Roughly speaking, this means that we treat all unknown quantities (e.g., predictions about the future value of some quantity of interest, such as tomorrow's temperature, or the parameters of some model) as **random variables**, that are endowed with **probability distributions** which describe a weighted set of possible values the variable may have. (See Chapter 2 for a quick refresher on the basics of probability, if necessary.)

There are two main reasons we adopt a probabilistic approach. First, it is the optimal approach to **decision making under uncertainty**, as we explain in Section 5.1. Second, probabilistic modeling is the language used by most other areas of science and engineering, and thus provides a unifying framework between these fields. As Shakir Mohamed, a researcher at DeepMind, put it:[1]

> Almost all of machine learning can be viewed in probabilistic terms, making probabilistic thinking fundamental. It is, of course, not the only view. But it is through this view that we can connect what we do in machine learning to every other computational science, whether that be in stochastic optimisation, control theory, operations research, econometrics, information theory, statistical physics or bio-statistics. For this reason alone, mastery of probabilistic thinking is essential.

1.2 Supervised learning

The most common form of ML is **supervised learning**. In this problem, the task T is to learn a mapping f from inputs $\boldsymbol{x} \in \mathcal{X}$ to outputs $\boldsymbol{y} \in \mathcal{Y}$. The inputs $\boldsymbol{x}$ are also called the **features**,

1. Source: Slide 2 of https://bit.ly/3pyHyPn

(a) (b) (c)

Figure 1.1: Three types of Iris flowers: Setosa, Versicolor and Virginica. Used with kind permission of Dennis Kramb and SIGNA.

index	sl	sw	pl	pw	label
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
	...				
50	7.0	3.2	4.7	1.4	Versicolor
	...				
149	5.9	3.0	5.1	1.8	Virginica

Table 1.1: A subset of the Iris design matrix. The features are: sepal length, sepal width, petal length, petal width. There are 50 examples of each class.

covariates, or **predictors**; this is often a fixed-dimensional vector of numbers, such as the height and weight of a person, or the pixels in an image. In this case, $\mathcal{X} = \mathbb{R}^D$, where D is the dimensionality of the vector (i.e., the number of input features). The output $\boldsymbol{y}$ is also known as the **label**, **target**, or **response**.[2] The experience E is given in the form of a set of N input-output pairs $\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_{n=1}^N$, known as the **training set**. (N is called the **sample size**.) The performance measure P depends on the type of output we are predicting, as we discuss below.

1.2.1 Classification

In **classification** problems, the output space is a set of C unordered and mutually exclusive labels known as **classes**, $\mathcal{Y} = \{1, 2, \ldots, C\}$. The problem of predicting the class label given an input is also called **pattern recognition**. (If there are just two classes, often denoted by $y \in \{0, 1\}$ or $y \in \{-1, +1\}$, it is called **binary classification**.)

1.2.1.1 Example: classifying Iris flowers

As an example, consider the problem of classifying Iris flowers into their 3 subspecies, Setosa, Versicolor and Virginica. Figure 1.1 shows one example of each of these classes.

2. Sometimes (e.g., in the statsmodels Python package) $\boldsymbol{x}$ are called the **exogenous variables** and $\boldsymbol{y}$ are called the **endogenous variables**.

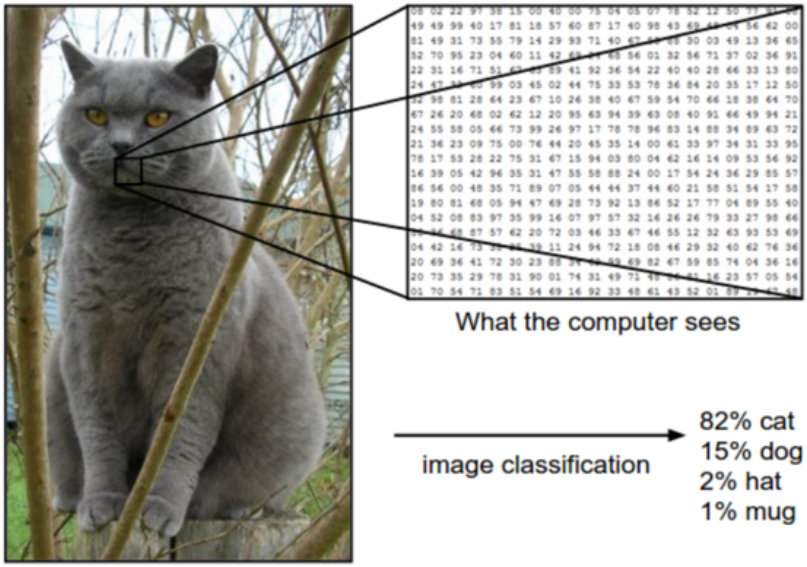


Figure 1.2: Illustration of the image classification problem. From https://cs231n.github.io/. Used with kind permission of Andrej Karpathy.

In **image classification**, the input space $\mathcal{X}$ is the set of images, which is a very high-dimensional space: for a color image with $C = 3$ channels (e.g., RGB) and $D_1 \times D_2$ pixels, we have $\mathcal{X} = \mathbb{R}^D$, where $D = C \times D_1 \times D_2$. (In practice we represent each pixel intensity with an integer, typically from the range $\{0, 1, \ldots, 255\}$, but we assume real valued inputs for notational simplicity.) Learning a mapping $f : \mathcal{X} \to \mathcal{Y}$ from images to labels is quite challenging, as illustrated in Figure 1.2. However, it can be tackled using certain kinds of functions, such as a **convolutional neural network** or **CNN**, which we discuss in Section 14.1.

Fortunately for us, some botanists have already identified 4 simple, but highly informative, numeric features — sepal length, sepal width, petal length, petal width — which can be used to distinguish the three kinds of Iris flowers. In this section, we will use this much lower-dimensional input space, $\mathcal{X} = \mathbb{R}^4$, for simplicity. The **Iris dataset** is a collection of 150 labeled examples of Iris flowers, 50 of each type, described by these 4 features. It is widely used as an example, because it is small and simple to understand. (We will discuss larger and more complex datasets later in the book.)

When we have small datasets of features, it is common to store them in an $N \times D$ matrix, in which each row represents an example, and each column represents a feature. This is known as a **design matrix**; see Table 1.1 for an example.[3]

The Iris dataset is an example of **tabular data**. When the inputs are of variable size (e.g., sequences of words, or social networks), rather than fixed-length vectors, the data is usually stored

3. This particular design matrix has $N = 150$ rows and $D = 4$ columns, and hence has a **tall and skinny** shape, since $N \gg D$. By contrast, some datasets (e.g., genomics) have more features than examples, $D \gg N$; their design matrices are **short and fat**. The term "**big data**" usually means that N is large, whereas the term "**wide data**" means that D is large (relative to N).

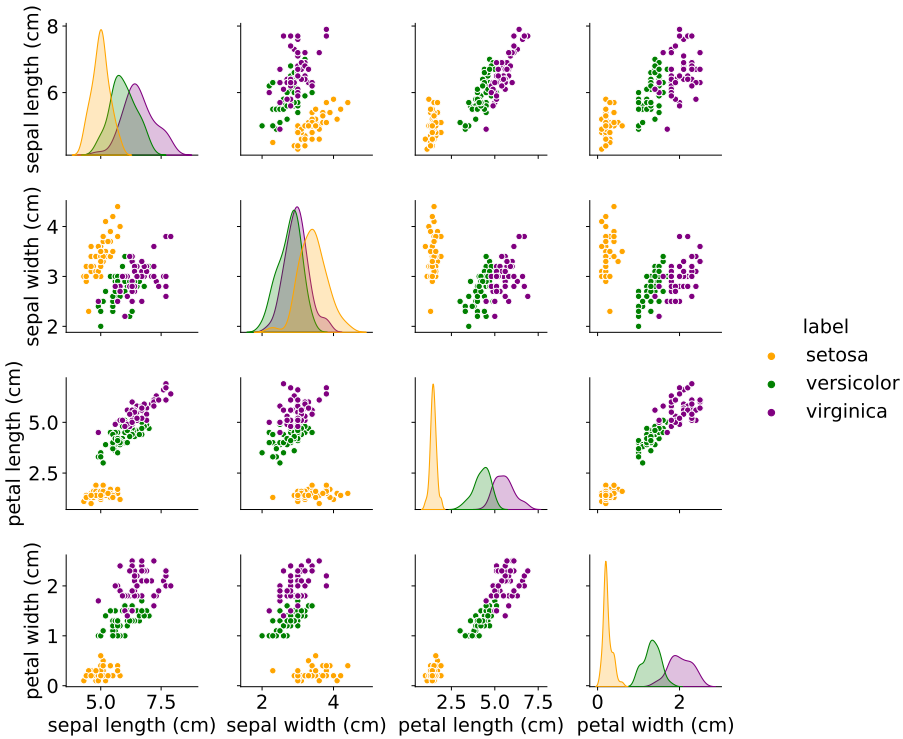


Figure 1.3: Visualization of the Iris data as a pairwise scatter plot. On the diagonal we plot the marginal distribution of each feature for each class. The off-diagonals contain scatterplots of all possible pairs of features. Generated by iris_plot.ipynb

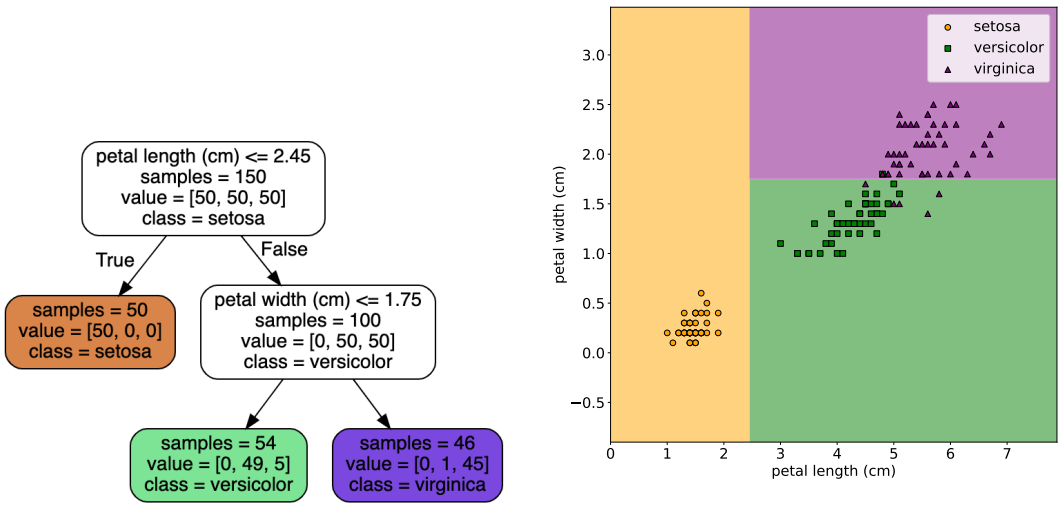
in some other format rather than in a design matrix. However, such data is often converted to a fixed-sized feature representation (a process known as **featurization**), thus implicitly creating a design matrix for further processing. We give an example of this in Section 1.5.4.1, where we discuss the "bag of words" representation for sequence data.

1.2.1.2 Exploratory data analysis

Before tackling a problem with ML, it is usually a good idea to perform **exploratory data analysis**, to see if there are any obvious patterns (which might give hints on what method to choose), or any obvious problems with the data (e.g., label noise or outliers).

For tabular data with a small number of features, it is common to make a **pair plot**, in which panel (i, j) shows a scatter plot of variables i and j, and the diagonal entries (i, i) show the marginal density of variable i; all plots are optionally color coded by class label — see Figure 1.3 for an example.

For higher-dimensional data, it is common to first perform **dimensionality reduction**, and then



(a) (b)

Figure 1.4: Example of a decision tree of depth 2 applied to the Iris data, using just the petal length and petal width features. Leaf nodes are color coded according to the predicted class. The number of training samples that pass from the root to a node is shown inside each box; we show how many values of each class fall into this node. This vector of counts can be normalized to get a distribution over class labels for each node. We can then pick the majority class. Adapted from Figures 6.1 and 6.2 of [Gér19]. Generated by iris_ dtree.ipynb.

to visualize the data in 2d or 3d. We discuss methods for dimensionality reduction in Chapter 20.

1.2.1.3 Learning a classifier

From Figure 1.3, we can see that the Setosa class is easy to distinguish from the other two classes. For example, suppose we create the following **decision rule**:

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \begin{cases} \text{Setosa if petal length} < 2.45 \\ \text{Versicolor or Virginica otherwise} \end{cases} \tag{1.1}$$

This is a very simple example of a classifier, in which we have partitioned the input space into two regions, defined by the one-dimensional (1d) **decision boundary** at $x_{\text{petal length}} = 2.45$. Points lying to the left of this boundary are classified as Setosa; points to the right are either Versicolor or Virginica.

We see that this rule perfectly classifies the Setosa examples, but not the Virginica and Versicolor ones. To improve performance, we can recursively partition the space, by splitting regions in which the classifier makes errors. For example, we can add another decision rule, to be applied to inputs that fail the first test, to check if the petal width is below 1.75cm (in which case we predict Versicolor) or above (in which case we predict Virginica). We can arrange these nested rules into a tree structure,

		Estimate		
		Setosa	Versicolor	Virginica
Truth	Setosa	0	1	1
	Versicolor	1	0	1
	Virginica	10	10	0

Table 1.2: Hypothetical asymmetric loss matrix for Iris classification.

called a **decision tree**, as shown in Figure 1.4a This induces the 2d **decision surface** shown in Figure 1.4b.

We can represent the tree by storing, for each internal node, the feature index that is used, as well as the corresponding threshold value. We denote all these **parameters** by $\boldsymbol{\theta}$. We discuss how to learn these parameters in Section 18.1.

1.2.1.4 Empirical risk minimization

The goal of supervised learning is to automatically come up with classification models such as the one shown in Figure 1.4a, so as to reliably predict the labels for any given input. A common way to measure performance on this task is in terms of the **misclassification rate** on the training set:

$$\mathcal{L}(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}\left(y_n \neq f(\boldsymbol{x}_n; \boldsymbol{\theta})\right) \tag{1.2}$$

where $\mathbb{I}(e)$ is the binary **indicator function**, which returns 1 iff (if and only if) the condition e is true, and returns 0 otherwise, i.e.,

$$\mathbb{I}(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{if } e \text{ is false} \end{cases} \tag{1.3}$$

This assumes all errors are equal. However it may be the case that some errors are more costly than others. For example, suppose we are foraging in the wilderness and we find some Iris flowers. Furthermore, suppose that Setosa and Versicolor are tasty, but Virginica is poisonous. In this case, we might use the asymmetric **loss function** $\ell(y, \hat{y})$ shown in Table 1.2.

We can then define **empirical risk** to be the average loss of the predictor on the training set:

$$\mathcal{L}(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n; \boldsymbol{\theta})) \tag{1.4}$$

We see that the misclassification rate Equation (1.2) is equal to the empirical risk when we use **zero-one loss** for comparing the true label with the prediction:

$$\ell_{01}(y, \hat{y}) = \mathbb{I}\left(y \neq \hat{y}\right) \tag{1.5}$$

See Section 5.1 for more details.

One way to define the problem of **model fitting** or **training** is to find a setting of the parameters that minimizes the empirical risk on the training set:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \mathcal{L}(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n; \boldsymbol{\theta})) \tag{1.6}$$

This is called **empirical risk minimization**.

However, our true goal is to minimize the expected loss on *future data* that we have not yet seen. That is, we want to **generalize**, rather than just do well on the training set. We discuss this important point in Section 1.2.3.

1.2.1.5 Uncertainty

> [We must avoid] false confidence bred from an ignorance of the probabilistic nature of the world, from a desire to see black and white where we should rightly see gray. — Immanuel Kant, as paraphrased by Maria Konnikova [Kon20].

In many cases, we will not be able to perfectly predict the exact output given the input, due to lack of knowledge of the input-output mapping (this is called **epistemic uncertainty** or **model uncertainty**), and/or due to intrinsic (irreducible) stochasticity in the mapping (this is called **aleatoric uncertainty** or **data uncertainty**).

Representing uncertainty in our prediction can be important for various applications. For example, let us return to our poisonous flower example, whose loss matrix is shown in Table 1.2. If we predict the flower is Virginica with high probability, then we should not eat the flower. Alternatively, we may be able to perform an **information gathering action**, such as performing a diagnostic test, to reduce our uncertainty. For more information about how to make optimal decisions in the presence of uncertainty, see Section 5.1.

We can capture our uncertainty using the following **conditional probability distribution**:

$$p(y = c|\boldsymbol{x}; \boldsymbol{\theta}) = f_c(\boldsymbol{x}; \boldsymbol{\theta}) \tag{1.7}$$

where $f : \mathcal{X} \to [0, 1]^C$ maps inputs to a probability distribution over the C possible output labels. Since $f_c(\boldsymbol{x}; \boldsymbol{\theta})$ returns the probability of class label c, we require $0 \leq f_c \leq 1$ for each c, and $\sum_{c=1}^{C} f_c = 1$. To avoid this restriction, it is common to instead require the model to return unnormalized log-probabilities. We can then convert these to probabilities using the **softmax function**, which is defined as follows

$$\operatorname{softmax}(\boldsymbol{a}) \triangleq \left[\frac{e^{a_1}}{\sum_{c'=1}^{C} e^{a_{c'}}}, \ldots, \frac{e^{a_C}}{\sum_{c'=1}^{C} e^{a_{c'}}} \right] \tag{1.8}$$

This maps $\mathbb{R}^C$ to $[0, 1]^C$, and satisfies the constraints that $0 \leq \operatorname{softmax}(\boldsymbol{a})_c \leq 1$ and $\sum_{c=1}^{C} \operatorname{softmax}(\boldsymbol{a})_c = 1$. The inputs to the softmax, $\boldsymbol{a} = f(\boldsymbol{x}; \boldsymbol{\theta})$, are called **logits**. See Section 2.5.2 for details. We thus define the overall model as follows:

$$p(y = c|\boldsymbol{x}; \boldsymbol{\theta}) = \operatorname{softmax}_c(f(\boldsymbol{x}; \boldsymbol{\theta})) \tag{1.9}$$

A common special case of this arises when f is an **affine function** of the form

$$f(\boldsymbol{x};\boldsymbol{\theta}) = b + \boldsymbol{w}^\mathsf{T}\boldsymbol{x} = b + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D \tag{1.10}$$

where $\boldsymbol{\theta} = (b, \boldsymbol{w})$ are the parameters of the model. This model is called **logistic regression**, and will be discussed in more detail in Chapter 10.

In statistics, the $\boldsymbol{w}$ parameters are usually called **regression coefficients** (and are typically denoted by $\boldsymbol{\beta}$) and b is called the **intercept**. In ML, the parameters $\boldsymbol{w}$ are called the **weights** and b is called the **bias**. This terminology arises from electrical engineering, where we view the function f as a circuit which takes in $\boldsymbol{x}$ and returns $f(\boldsymbol{x})$. Each input is fed to the circuit on "wires", which have weights $\boldsymbol{w}$. The circuit computes the weighted sum of its inputs, and adds a constant bias or offset term b. (This use of the term "bias" should not be confused with the statistical concept of bias discussed in Section 4.7.6.1.)

To reduce notational clutter, it is common to absorb the bias term b into the weights $\boldsymbol{w}$ by defining $\tilde{\boldsymbol{w}} = [b, w_1, \ldots, w_D]$ and defining $\tilde{\boldsymbol{x}} = [1, x_1, \ldots, x_D]$, so that

$$\tilde{\boldsymbol{w}}^\mathsf{T}\tilde{\boldsymbol{x}} = b + \boldsymbol{w}^\mathsf{T}\boldsymbol{x} \tag{1.11}$$

This converts the affine function into a **linear function**. We will usually assume that this has been done, so we can just write the prediction function as follows:

$$f(\boldsymbol{x};\boldsymbol{w}) = \boldsymbol{w}^\mathsf{T}\boldsymbol{x} \tag{1.12}$$

1.2.1.6 Maximum likelihood estimation

When fitting probabilistic models, it is common to use the negative log probability as our loss function:

$$\ell(y, f(\boldsymbol{x};\boldsymbol{\theta})) = -\log p(y|f(\boldsymbol{x};\boldsymbol{\theta})) \tag{1.13}$$

The reasons for this are explained in Section 5.1.6.1, but the intuition is that a good model (with low loss) is one that assigns a high probability to the true output y for each corresponding input $\boldsymbol{x}$. The average negative log probability of the training set is given by

$$\text{NLL}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N} \log p(y_n|f(\boldsymbol{x}_n;\boldsymbol{\theta})) \tag{1.14}$$

This is called the **negative log likelihood**. If we minimize this, we can compute the **maximum likelihood estimate** or **MLE**:

$$\hat{\boldsymbol{\theta}}_{\text{mle}} = \operatorname*{argmin}_{\boldsymbol{\theta}} \text{NLL}(\boldsymbol{\theta}) \tag{1.15}$$

This is a very common way to fit models to data, as we will see.

1.2.2 Regression

Now suppose that we want to predict a real-valued quantity $y \in \mathbb{R}$ instead of a class label $y \in \{1, \ldots, C\}$; this is known as **regression**. For example, in the case of Iris flowers, y might be the degree of toxicity if the flower is eaten, or the average height of the plant.

Regression is very similar to classification. However, since the output is real-valued, we need to use a different loss function. For regression, the most common choice is to use **quadratic loss**, or ℓ_2 **loss**:

$$\ell_2(y, \hat{y}) = (y - \hat{y})^2 \tag{1.16}$$

This penalizes large **residuals** $y - \hat{y}$ more than small ones.[4] The empirical risk when using quadratic loss is equal to the **mean squared error** or **MSE**:

$$\text{MSE}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(y_n - f(\boldsymbol{x}_n; \boldsymbol{\theta}))^2 \tag{1.17}$$

Based on the discussion in Section 1.2.1.5, we should also model the uncertainty in our prediction. In regression problems, it is common to assume the output distribution is a **Gaussian** or **normal**. As we explain in Section 2.6, this distribution is defined by

$$\mathcal{N}(y|\mu, \sigma^2) \triangleq \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y-\mu)^2} \tag{1.18}$$

where μ is the mean, σ^2 is the variance, and $\sqrt{2\pi\sigma^2}$ is the normalization constant needed to ensure the density integrates to 1. In the context of regression, we can make the mean depend on the inputs by defining $\mu = f(\boldsymbol{x}_n; \boldsymbol{\theta})$. We therefore get the following conditional probability distribution:

$$p(y|\boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y|f(\boldsymbol{x}; \boldsymbol{\theta}), \sigma^2) \tag{1.19}$$

If we assume that the variance σ^2 is fixed (for simplicity), the corresponding negative log likelihood becomes

$$\text{NLL}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{n=1}^{N_{\mathcal{D}}} \log\left[\left(\frac{1}{2\pi\sigma^2}\right)^{\frac{1}{2}} \exp\left(-\frac{1}{2\sigma^2}(y_n - f(\boldsymbol{x}_n; \boldsymbol{\theta}))^2\right)\right] \tag{1.20}$$

$$= \frac{1}{2\sigma^2}\text{MSE}(\boldsymbol{\theta}) + \text{const} \tag{1.21}$$

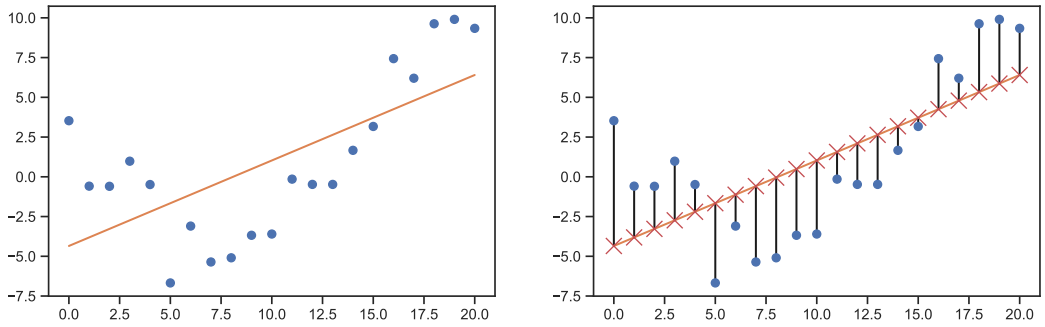
We see that the NLL is proportional to the MSE. Hence computing the maximum likelihood estimate of the parameters will result in minimizing the squared error, which seems like a sensible approach to model fitting.

1.2.2.1 Linear regression

As an example of a regression model, consider the 1d data in Figure 1.5a. We can fit this data using a **simple linear regression** model of the form

$$f(x; \boldsymbol{\theta}) = b + wx \tag{1.22}$$

4. If the data has outliers, the quadratic penalty can be too severe. In such cases, it can be better to use ℓ_1 loss instead, which is more **robust**. See Section 11.6 for details.



(a) *(b)*

Figure 1.5: (a) Linear regression on some 1d data. (b) The vertical lines denote the residuals between the observed output value for each input (blue circle) and its predicted value (red cross). The goal of least squares regression is to pick a line that minimizes the sum of squared residuals. Generated by linreg_residuals_plot.ipynb.

where w is the **slope**, b is the **offset**, and $\boldsymbol{\theta} = (w, b)$ are all the parameters of the model. By adjusting $\boldsymbol{\theta}$, we can minimize the sum of squared errors, shown by the vertical lines in Figure 1.5b. until we find the **least squares solution**

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \operatorname{MSE}(\boldsymbol{\theta}) \tag{1.23}$$

See Section 11.2.2.1 for details.

If we have multiple input features, we can write

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = b + w_1 x_1 + \cdots + w_D x_D = b + \boldsymbol{w}^\mathsf{T} \boldsymbol{x} \tag{1.24}$$

where $\boldsymbol{\theta} = (\boldsymbol{w}, b)$. This is called **multiple linear regression**.

For example, consider the task of predicting temperature as a function of 2d location in a room. Figure 1.6(a) plots the results of a linear model of the following form:

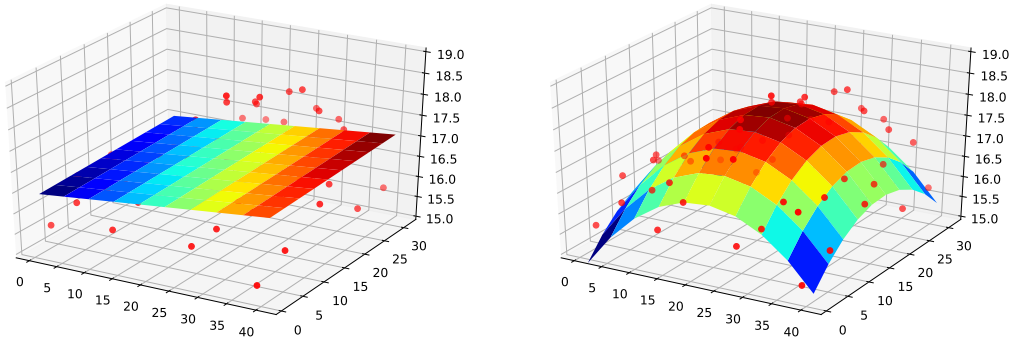
$$f(\boldsymbol{x}; \boldsymbol{\theta}) = b + w_1 x_1 + w_2 x_2 \tag{1.25}$$

We can extend this model to use $D > 2$ input features (such as time of day), but then it becomes harder to visualize.

1.2.2.2 Polynomial regression

The linear model in Figure 1.5a is obviously not a very good fit to the data. We can improve the fit by using a **polynomial regression** model of degree D. This has the form $f(x; \boldsymbol{w}) = \boldsymbol{w}^\mathsf{T} \boldsymbol{\phi}(x)$, where $\boldsymbol{\phi}(x)$ is a feature vector derived from the input, which has the following form:

$$\boldsymbol{\phi}(x) = [1, x, x^2, \ldots, x^D] \tag{1.26}$$



(a) (b)

Figure 1.6: Linear and polynomial regression applied to 2d data. Vertical axis is temperature, horizontal axes are location within a room. Data was collected by some remote sensing motes at Intel's lab in Berkeley, CA (data courtesy of Romain Thibaux). (a) The fitted plane has the form $\hat{f}(\boldsymbol{x}) = w_0 + w_1 x_1 + w_2 x_2$. *(b) Temperature data is fitted with a quadratic of the form* $\hat{f}(\boldsymbol{x}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2$. *Generated by linreg_2d_surface_demo.ipynb.*

This is a simple example of **feature preprocessing**, also called **feature engineering**.

In Figure 1.7a, we see that using $D = 2$ results in a much better fit. We can keep increasing D, and hence the number of parameters in the model, until $D = N - 1$; in this case, we have one parameter per data point, so we can perfectly **interpolate** the data. The resulting model will have 0 MSE, as shown in Figure 1.7c. However, intuitively the resulting function will not be a good predictor for future inputs, since it is too "wiggly". We discuss this in more detail in Section 1.2.3.

We can also apply polynomial regression to multi-dimensional inputs. For example, Figure 1.6(b) plots the predictions for the temperature model after performing a quadratic expansion of the inputs

$$f(\boldsymbol{x}; \boldsymbol{w}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 \tag{1.27}$$

The quadratic shape is a better fit to the data than the linear model in Figure 1.6(a), since it captures the fact that the middle of the room is hotter. We can also add cross terms, such as $x_1 x_2$, to capture interaction effects. See Section 1.5.3.2 for details.

Note that the above models still use a prediction function that is a linear function of the parameters $\boldsymbol{w}$, even though it is a nonlinear function of the original input $\boldsymbol{x}$. The reason this is important is that a linear model induces an MSE loss function $\text{MSE}(\boldsymbol{\theta})$ that has a unique global optimum, as we explain in Section 11.2.2.1.

1.2.2.3 Deep neural networks

In Section 1.2.2.2, we manually specified the transformation of the input features, namely polynomial expansion, $\boldsymbol{\phi}(\boldsymbol{x}) = [1, x_1, x_2, x_1^2, x_2^2, \ldots]$. We can create much more powerful models by learning to do such nonlinear **feature extraction** automatically. If we let $\boldsymbol{\phi}(\boldsymbol{x})$ have its own set of parameters, say $\mathbf{V}$, then the overall model has the form

$$f(\boldsymbol{x}; \boldsymbol{w}, \mathbf{V}) = \boldsymbol{w}^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{x}; \mathbf{V}) \tag{1.28}$$

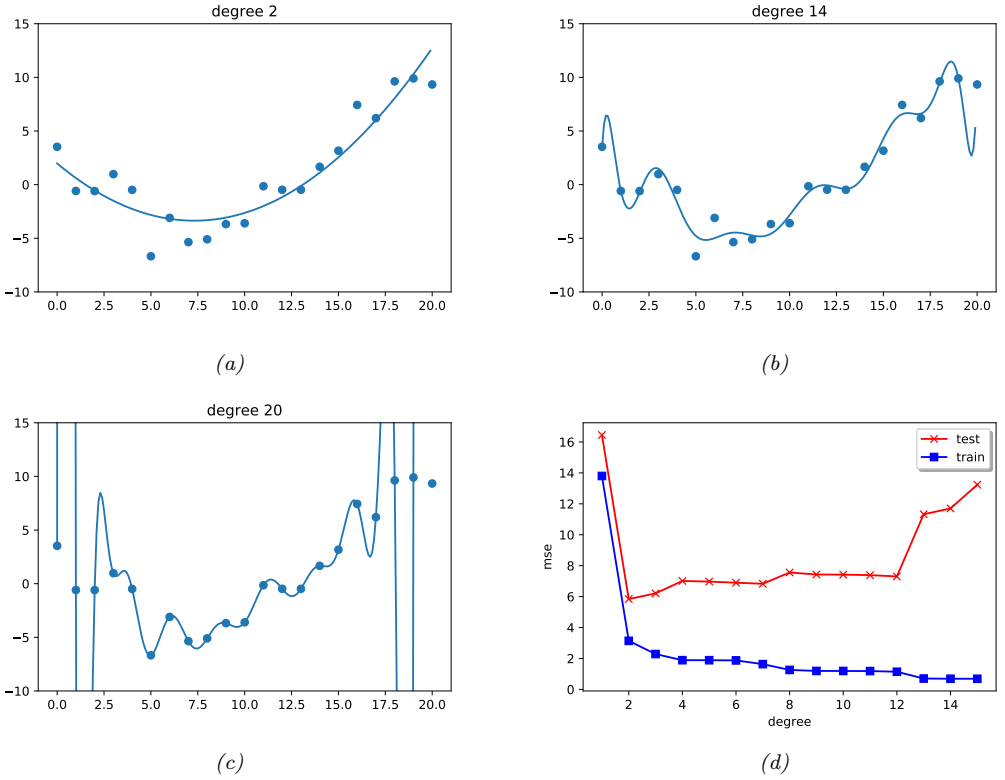


Figure 1.7: (a-c) Polynomials of degrees 2, 14 and 20 fit to 21 datapoints (the same data as in Figure 1.5). (d) MSE vs degree. Generated by linreg_poly_vs_degree.ipynb.

We can recursively decompose the feature extractor $\boldsymbol{\phi}(\boldsymbol{x}; \mathbf{V})$ into a composition of simpler functions. The resulting model then becomes a stack of L nested functions:

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = f_L(f_{L-1}(\cdots(f_1(\boldsymbol{x}))\cdots)) \tag{1.29}$$

where $f_\ell(\boldsymbol{x}) = f(\boldsymbol{x}; \boldsymbol{\theta}_\ell)$ is the function at layer ℓ. The final layer is linear and has the form $f_L(\boldsymbol{x}) = \boldsymbol{w}^\mathsf{T} f_{1:L-1}(\boldsymbol{x})$, where $f_{1:L-1}(\boldsymbol{x})$ is the learned feature extractor. This is the key idea behind **deep neural networks** or **DNN**s, which includes common variants such as **convolutional neural networks** (CNNs) for images, and **recurrent neural networks** (RNNs) for sequences. See Part III for details.

1.2.3 Overfitting and generalization

We can rewrite the empirical risk in Equation (1.4) in the following equivalent way:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_{\text{train}}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}_{\text{train}}} \ell(\boldsymbol{y}, f(\boldsymbol{x}; \boldsymbol{\theta})) \tag{1.30}$$

where $|\mathcal{D}_{\text{train}}|$ is the size of the training set $\mathcal{D}_{\text{train}}$. This formulation is useful because it makes explicit which dataset the loss is being evaluated on.

With a suitably flexible model, we can drive the training loss to zero (assuming no label noise), by simply memorizing the correct output for each input. For example, Figure 1.7(c) perfectly interpolates the training data (modulo the last point on the right). But what we care about is prediction accuracy on new data, which may not be part of the training set. A model that perfectly fits the training data, but which is too complex, is said to suffer from **overfitting**.

To detect if a model is overfitting, let us assume (for now) that we have access to the true (but unknown) distribution $p^*(\boldsymbol{x}, \boldsymbol{y})$ used to generate the training set. Then, instead of computing the empirical risk we compute the theoretical expected loss or **population risk**

$$\mathcal{L}(\boldsymbol{\theta}; p^*) \triangleq \mathbb{E}_{p^*(\boldsymbol{x},\boldsymbol{y})}\left[\ell(\boldsymbol{y}, f(\boldsymbol{x}; \boldsymbol{\theta}))\right] \tag{1.31}$$

The difference $\mathcal{L}(\boldsymbol{\theta}; p^*) - \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_{\text{train}})$ is called the **generalization gap**. If a model has a large generalization gap (i.e., low empirical risk but high population risk), it is a sign that it is overfitting.

In practice we don't know p^*. However, we can partition the data we do have into two subsets, known as the training set and the **test set**. Then we can approximate the population risk using the **test risk**:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_{\text{test}}) \triangleq \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in \mathcal{D}_{\text{test}}} \ell(y, f(\boldsymbol{x}; \boldsymbol{\theta})) \tag{1.32}$$

As an example, in Figure 1.7d, we plot the training error and test error for polynomial regression as a function of degree D. We see that the training error goes to 0 as the model becomes more complex. However, the test error has a characteristic **U-shaped curve**: on the left, where $D = 1$, the model is **underfitting**; on the right, where $D \gg 1$, the model is **overfitting**; and when $D = 2$, the model complexity is "just right".

How can we pick a model of the right complexity? If we use the training set to evaluate different models, we will always pick the most complex model, since that will have the most **degrees of freedom**, and hence will have minimum loss. So instead we should pick the model with minimum test loss.

In practice, we need to partition the data into three sets, namely the training set, the test set and a **validation set**; the latter is used for model selection, and we just use the test set to estimate future performance (the population risk), i.e., the test set is not used for model fitting or model selection. See Section 4.5.4 for further details.

1.2.4 No free lunch theorem

> All models are wrong, but some models are useful. — George Box [BD87, p424].[5]

Given the large variety of models in the literature, it is natural to wonder which one is best. Unfortunately, there is no single best model that works optimally for all kinds of problems — this is sometimes called the **no free lunch theorem** [Wol96]. The reason is that a set of assumptions (also called **inductive bias**) that works well in one domain may work poorly in another. The best

5. George Box is a retired statistics professor at the University of Wisconsin.

way to pick a suitable model is based on domain knowledge, and/or trial and error (i.e., using model selection techniques such as cross validation (Section 4.5.4) or Bayesian methods (Section 5.2.2 and Section 5.2.6). For this reason, it is important to have many models and algorithmic techniques in one's toolbox to choose from.

1.3 Unsupervised learning

In supervised learning, we assume that each input example $\boldsymbol{x}$ in the training set has an associated set of output targets $\boldsymbol{y}$, and our goal is to learn the input-output mapping. Although this is useful, and can be difficult, supervised learning is essentially just "glorified curve fitting" [Pea18].

An arguably much more interesting task is to try to "make sense of" data, as opposed to just learning a mapping. That is, we just get observed "inputs" $\mathcal{D} = \{\boldsymbol{x}_n : n = 1 : N\}$ without any corresponding "outputs" $\boldsymbol{y}_n$. This is called **unsupervised learning**.

From a probabilistic perspective, we can view the task of unsupervised learning as fitting an unconditional model of the form $p(\boldsymbol{x})$, which can generate new data $\boldsymbol{x}$, whereas supervised learning involves fitting a conditional model, $p(\boldsymbol{y}|\boldsymbol{x})$, which specifies (a distribution over) outputs given inputs.[6]

Unsupervised learning avoids the need to collect large labeled datasets for training, which can often be time consuming and expensive (think of asking doctors to label medical images).

Unsupervised learning also avoids the need to learn how to partition the world into often arbitrary categories. For example, consider the task of labeling when an action, such as "drinking" or "sipping", occurs in a video. Is it when the person picks up the glass, or when the glass first touches the mouth, or when the liquid pours out? What if they pour out some liquid, then pause, then pour again — is that two actions or one? Humans will often disagree on such issues [Idr+17], which means the task is not well defined. It is therefore not reasonable to expect machines to learn such mappings.[7]

Finally, unsupervised learning forces the model to "explain" the high-dimensional inputs, rather than just the low-dimensional outputs. This allows us to learn richer models of "how the world works". As Geoff Hinton, who is a famous professor of ML at the University of Toronto, has said:

> When we're learning to see, nobody's telling us what the right answers are — we just look. Every so often, your mother says "that's a dog", but that's very little information. You'd be lucky if you got a few bits of information — even one bit per second — that way. The brain's visual system has 10^{14} neural connections. And you only live for 10^9 seconds. So it's no use learning one bit per second. You need more like 10^5 bits per second. And there's only one place you can get that much information: from the input itself. — Geoffrey Hinton, 1996 (quoted in [Gor06]).

1.3.1 Clustering

A simple example of unsupervised learning is the problem of finding **clusters** in data. The goal is to partition the input into regions that contain "similar" points. As an example, consider a 2d version

6. In the statistics community, it is common to use $\boldsymbol{x}$ to denote exogenous variables that are not modeled, but are simply given as inputs. Therefore an unconditional model would be denoted $p(\boldsymbol{y})$ rather than $p(\boldsymbol{x})$.

7. A more reasonable approach is to try to capture the probability distribution over labels produced by a "crowd" of annotators (see e.g., [Dum+18; Aro+19]). This embraces the fact that there can be multiple "correct" labels for a given input due to the ambiguity of the task itself.