Machine Learning

CSC_2S004_EP (previously: CSE204)

Jesse Read

jesse.read@polytechnique.edu

Version: May 11, 2025

These Lecture Notes are still a work in progress and **will be updated and adapted throughout the course** (expect updates each week). There is approximately one chapter per topic/week. These notes are *not* intended as a complete reference of all course material; they are designed to be read to complement (not to substitute) lectures. Neither are these notes uniquely required for a full understanding of course material. Machine Learning is by now a very well-covered topic, you will find many alternative (often, open-source) references.

The main objective of these notes is to introduce and develop theoretical concepts which are presented in the lectures. Practical machine learning is also an important component of the course. Practical aspects will be discussed in lectures, but mainly covered in the Tutorial/Lab sessions.

How to read the colours in these notes: Updates to earlier chapters (e.g., to address questions/comments raised in class, during the current iteration of the course); keywords (important concepts/key terms); various highlights (aspects worth emphasising, and 'takeaway' messages); hyperlinks (click on them to jump within and without of the document); and out-of-scope material (either not graded but perhaps of interest to some).

Quick Link: Table of Contents

Acknowledgements: To all CSE204 students who inspired content by asking insightful questions (and pointed out all the bugs).

About CSC_2S004_EP: Knowledge You will Obtain from This Course

Briefly: You should finish the course having gained sufficient knowledge to complete a typical machine learning pipeline/project from start to end: from formulating the problem and preparing data, to selecting an appropriate model and learning algorithm, and being able to understand and explain the performance and the limitations of such a selection on such a problem.

The course is divided very roughly into two parts: first we lay the foundations, and some important formal notions. This allows us, in the second half of the course, to take a tour through a number of contemporary machine learning methods and 'recipes', understanding both the practical implications and the theoretical insights behind each one. At the end of the course you will have enough knowledge to take on a range of machine learning problems, as well as the ability to move into advanced topics, such as those covered in post-graduate studies.

A rough outline of what you're expected to know/be able to do by the end of week 5

Theoretical knowledge: You should

- Have notions about the main terms (these are highlighted as keywords where introduced), including features, labels, training vs test set, model, prediction, irreducible error vs residual error vs loss
- Be able to follow a derivation of Ordinary Least Squares (OLS) and Logistic Regression. In doing so you should be able to identify and motivate: *data types* (what is x, what is y), choice of *performance/error/loss metric*, *optimization method*, and *model representation*.
- Understand the main assumptions of OLS (and how/when these assumptions could be incorrect; and some strategies to take in those cases).
- Understand Gradient Descent (GD): when/why use it; how to derive gradient update from loss metric; will it converge
- Be able to discuss the main differences (and advantages vs disadvantages) of the parametric models mentioned above wrt non-parametric k-Nearest-Neighbors (kNN).
- Be clear on how different k affects the performance of kNN and polynomial regression (having k basis functions), in particular with regard to overfitting
- Understand uncertainty in machine learning and how this relates to the bias-variance trade-off;
- What is overfitting and how it can be mitigated with regularization.

Practical skills: You should be able to

- Use the PANDAS library for loading, inspecting, and manipulating data sets
- Be familiar with the NUMPY library for matrix algebra and probability, e.g., be able to translate something like $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_2 0.1)$ or $(\mathbf{w}^\top \mathbf{X} - \mathbf{y})^\top (\mathbf{w}^\top \mathbf{X} - \mathbf{y})$ into one line of Python code.
- Interpret (from a theoretical and practical perspective) the meaning of the parameters/weights/coefficients
- and the output of the sigmoid function $\sigma(\cdot)$ wrt classification and confidence
- Design and implement basis functions ϕ, and use them to obtain a non-linear decision boundary/surface
- Cast a relatively simple ML problem (given as a textual description) into an ML abstraction (`X`, `y`, etc.)
- Implement (at least in a basic form) methods such as: OLS, kNN, Logistic Regression with Gradient Descent
- Understand practical implications of, e.g., using OLS vs kNN regression
- Instantiate ML models from the SKLEARN library, and use them to solve regression and classification tasks
- Interpret learning curves (on training data, and test data)
- Be able to tune hyper-parameters (k, α, λ, ...) to values that work for a given [practical] problem
- Be able to produce an intuitive and concise visualisation of your results (e.g., tables, or plots)

- Be aware of what is at stake wrt *overfitting* vs *underfitting* (bias vs variance) for every method (linear-, polynomial regression, kNN, ...), with regard to a practical/hands-on problem; and be able to choose and employ techniques (especially: ridge penalty, and cross validation) to ensure a reasonable tradeoff

Updates to expect to material in 2025

The field of Machine Learning is expanding and developing at a rapid pace. In the present iteration of `CSC_2S004_EP` (2025), respect with previous years, there will be relatively more emphasis on Probabilistic Machine Learning (and uncertainty analysis) and Generative Modelling. Kernel Methods such as Support Vector Machines will remain phased out. That does *not* mean that this material is no longer relevant in machine learning. We retain a study of many 'traditional' ML techniques such as Linear and Logistic Regression, k-Means Clustering, not for tradition's sake; but because they are effective tools on their own as well as serving as excellent foundation to more advanced (including, but not only, modern) methods. In terms of notation, we make a particular effort in 2025 to distinguish between loss metric ℓ of which the expectation is minimised by a machine learning algorithm (and other kinds of error).

Contents

Part I

Foundations

Chapter 1

Introduction to Machine Learning

We will introduce some notation and important terms and concepts that will be used throughout much of the course; a general overview of machine learning; and some practical advice for taking on a machine learning project (such as the course project).

Main Concepts

You should learn what this course is about (what you are expected to learn), and also the following concepts:

- data, [test] instance, features, labels, training set, predictions
- model and error (vs residuals), ground truth
- loss metric and notions of distance
- optimization as learning
- tasks: what is supervised vs unsupervised learning; regression vs classification, classification vs clustering; . . .
- setting up a machine learning problem
- issues arising when dealing with data: missing values, bias, scaling, size, anomalies, avoiding typical pitfalls . . .
- what is machine learning (and with respect to statistics, optimisation, computer science, . . .)

1.1 Representing Data

We can typically represent data as a matrix,

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,d} \end{bmatrix} \tag{1.1}$$

with columns being features (or attributes), and each row is an instance of data (or data point). Let the i-th row (instance) be

$$\boldsymbol{x}_i = [x_{i,1}, \ldots, x_{i,d}]$$

with $x_{i,j} \Leftrightarrow x_j^{(i)}$ (alternative notation) being the j-th attribute of the i-th instance.

The data can also be represented as a set of n instances:

$$\{\boldsymbol{x}_i\}_{i=1}^n \Leftrightarrow \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$$

We often have labels available:

$$\boldsymbol{y} = [y_1, \ldots, y_n]^\top = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{1.2}$$

such that label y_i (also called the target) is associated with each i-th instance $\boldsymbol{x}_i$. Putting these together as tuples/examples, we have a data set:

$$\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n \Leftrightarrow \{\mathbf{X}, \boldsymbol{y}\}$$

We learn to *predict* y from $\boldsymbol{x}$, i.e., map $\boldsymbol{x}$ to y. The domain $\mathcal{Y}$ in which each y_i lives ($y_i \in \mathcal{Y}$), and whether we have y_i available or not, determines the type of machine learning task. Of course, the domain $\mathcal{X}$ in which each $\boldsymbol{x}_i$ lives, also has a significant effect on how we approach a problem.

Often we will borrow notation from statistics, where upper case $\boldsymbol{X} := X_1, \ldots, X_d$ and Y denote random variables (describing/denoting features, i.e., symbols attached to columns) and lower case $\boldsymbol{x} = [x_1, \ldots, x_d]$ and y (as above) as actual data points, i.e., *instances*/realisations of those random variables, containing actual values. If the row-subscript is absent, ($\boldsymbol{x}$, y), it indicates a generic instance, often a test instance along side hypothetical label, i.e., one that was not necessarily observed previously in the training set (from which our machine learning algorithms derive knowledge). This is an important distinction of machine learning: the data processed by a learning algorithm is usually not the resulting model (produced by an algorithm) will operate on.

Machine learning is at the crossroads of different fields (computer science, statistics, applied mathematics, signal processing, ...); each field brings its own terms and notation – so be ready to see different notation and different terms that mean the same thing!

1.2 Model and Error

Any relation we are interested in modeling using the data may be described by the following:

$$y = f_\star(\boldsymbol{x}) + \epsilon \tag{1.3}$$

where $f_\star$ is the true ('ground truth' or 'generating') model – a hypothetical concept/never observed in practice; and ϵ is its irreducible error. The true model is the best one we could obtain given infinite data. Why does it have error?

Let's consider a toy problem, of Example 1.2. Some made-up data for this toy example is plotted in Figure 1.1. The black points represent the training examples we have observed (actual people that would exist if this were real data). These points were 'generated' from some true *concept*. Formally we represent a concept via probability distributions, for example $P(Y, X) = P(Y|X)P(X)$; in the figure, the variance of conditional distribution $P(Y \mid \boldsymbol{x})$ is shown, and can be interpreted as an indication of confidence. Specifically, in this case, $\epsilon \sim \mathcal{N}(0, \sigma^2)$ and $y \sim \mathcal{N}(f(x), \sigma^2)$.

Example 1.2.1- GPA vs Early career earnings

Suppose we want to model the relationship of grade point averages (GPAs) vs their early-career earnings (in k€/month). For example, a financial institution may wish to know this is order make a decision regarding a loan to final-year students, or we might simply be conducting analytics/a study involving this question in order to formulate advice for students.

The 'true' (unknown) model $f_\star$ will provide the best possible estimate $f_\star(x)$ for any given x, generated from the underlying concept. The data we consider here (Fig. 1.1) is made up, so the 'true' concept refers to the one that generated *this* data. It indicates that higher GPA correlates *in general* with higher earnings. But there are cases such that as hypothetical Fred who obtained a GPA of 1.0 and then went to work in his father's business earning a nice monthly sum, and hypothetical Susan (not shown) who got a GPA of 4.3 but went into research, embarking on a PhD, and earning barely above the minimum wage.

We only have a finite set of training points $(\boldsymbol{x}_i, y_i)$ with which to build our model ($\widehat{f}$; in the Fig. 1.1 has been constructed via OLS – which will be covered later in Section 2.1). If all our assumptions are correct and we have infinite training data, the best we can hope for is a fit and an error similar to that incurred by $f_\star$.

Why is there always error? Yes, we could add features (GPA from which university, intelligence, family wealth, etc.) to our data (thus representing a more detailed concept), but still we would never expect 0 error in a real-world problem. We can never *know* the future with complete certainty.

Say you are aiming for a GPA of 3.0 ($x = 3$) and want to have an idea of your future earnings, the prediction of the model $\widehat{y} = \widehat{f}(x)$ is shown as a horizontal line in magenta. The true expected earnings (according to this example) is slightly lower. What are the actual earnings (what is the actual error of this prediction)? We will have to wait and see; if we knew that already, we wouldn't need to build a model; and hence the point of machine learning.

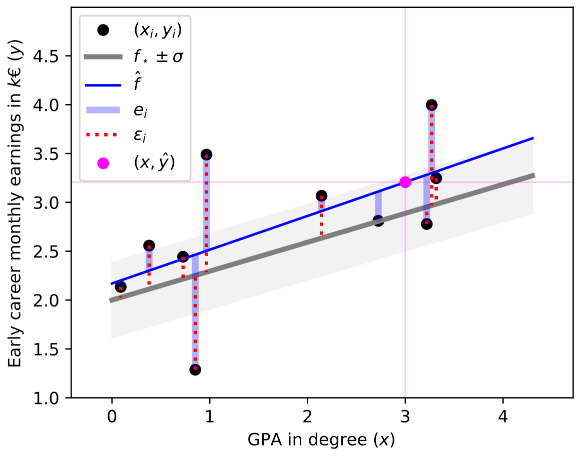


Figure 1.1: Fictitious data $\{(x_i, y_i)\}_{i=1}^n$, model $\widehat{f} \approx f_\star$, irreducible error $\epsilon = y - f_\star(x)$, prediction $\widehat{y} = \widehat{f}(x)$, and residual error e_i which is input to the loss metric $\ell_i = \ell(y_i, \widehat{y}_i) = \ell(e_i)$. The relationship between x and y does not need to be linear for these concepts to hold.

1.3 Machine Learning

Machine learning is about obtaining model $\widehat{f} \approx f_\star$ from data $\mathcal{D}$. We could instead ask a domain expert to design $\widehat{f}$ for us from expert intuition (or $\widehat{f}$ may just *be* a human expert), but in machine learning we use a learning algorithm to *learn* $\widehat{f}$. How do we learn $\widehat{f}$? Which $\widehat{f}$ is good enough for our task?

The criteria we are optimizing is our performance metric. A performance metric can be a payoff function like accuracy (higher is better) or, inversely, a loss metric (lower is better; an accuracy of 100% equals loss of 0). Let us denote loss metric (or loss function)

$$\ell(\widehat{y}, y)$$

This function should give $\ell(\widehat{y}, y) = 0$ when $\widehat{y} = y$ (when our prediction $\widehat{y}$ is precisely accurate, i.e., zero error) and some number $\ell(\widehat{y}, y) > 0$ when our prediction is incorrect to some degree; where the value $\ell(\widehat{y}, y)$ is proportional to the 'incorrectness'. This function ℓ does not need to be symmetrical with regard to its arguments; underestimating y may produce a larger error than overestimating by the same quantity. The choice and/or design of the performance metric should come under extremely careful consideration, as it answers the question from above *which $\widehat{f}$ is a good one*: the one that provides predictions $\widehat{y} = \widehat{f}(\boldsymbol{x})$ such that error $\ell(\widehat{y}, y)$ is, in general, as low as possible. (We emphasise, and will do throughout, the *in general*, as opposed to for each specific training case).

1.3.1 Error vs Residual vs Loss: Sorting it out

Although there is clear conceptual overlap, we nevertheless strive to distinguish among these terms, because often it is important: the irreducible error *variable* ϵ is not the same as the loss *function* $\ell(\cdot, \cdot)$, although our *expected loss* (coming soon, Eq. (1.4)) will contain/inherit the irreducible error. The residual error e is empirically observable (for a specific data point); and ℓ is usually a function of e. Observe these relations:

$$\begin{aligned}
\widehat{y} &= y + e && \triangleright \text{ A prediction} \\
e &= y - \widehat{y} && \triangleright \text{ Its residual error} \\
\ell(y, \widehat{y}) &= \ell(e) && \triangleright \text{ The loss function, a function of the residual} \\
\widehat{y} &= f_\star(x) + \epsilon + e && \triangleright \text{ Plugged Eq. (1.3) into the top equation}
\end{aligned}$$

thereby we see both the irreducible and reducible error are involved in a prediction for a given input.

A common example of a loss metric in regression is the squared error:

$$\ell(\widehat{y}_i, y_i) = e_i^2 = (\widehat{y}_i - y_i)^2$$

which is also equivalent to classification error (the inverse of classification accuracy) when $y \in \{0, 1\}$. The terms *error* and *loss* can sometimes be used interchangeably, but there is actually a conceptual difference: sometimes in machine learning (and life, in general), *we incur loss even when we do not commit any error*.

So now there are several meanings behind the term 'error'; which error should we reduce, from a machine-learning perspective? The residual error is only tangible in the training phase, since to measure it requires both a training label and a prediction, for the same instance; we do not the true label when a model is deployed (otherwise there would be no point to deploy it). And in this context it is easy to reduce the residual error to 0 simply by predicting $\widehat{y}$ such that $\widehat{y} = y$. Anyway, in the general sense it does not make sense to seek lower e because negative values are just as bad as positive. The irreducible error is irreducible, so neither does it make sense to target that (even when $\widehat{f} = f_\star$, and hence why, as a rule of thumb, we should never expect 100% accuracy). We can aim to reduce loss ℓ, where lower is always better (and 0 is best); but, on the other hand, achieving $\ell(y_i, \widehat{y}_i) = 0$ is simple, by setting $\widehat{y}_i = f(x_i) = y_i$. Simple, and pointless.

So the question: In machine learning, what error/loss are we most interested in? Here is the answer: we do not aim to minimise loss $\ell(\widehat{y}_i, y_i)$, rather we aim to minimise the expected loss:

$$\widehat{y} = \operatorname*{argmin}_f \mathbb{E}_{Y \sim P(Y|x)}[\ell(f(x), Y)] \tag{1.4}$$

This equation expresses concisely the goal of machine learning: build model f whose predictions $\widehat{y}$ for a given instance x can be *expected* to be 'good' (i.e., small *loss*).

This is worth emphasising, as it highlights the main particularity of machine learning (as opposed to, say, pure optimisation or basic statistics): we aim that $\ell(\widehat{y}, y)$ be low on instances we have *not seen* before; i.e., it should be such that when we *deploy* $\widehat{f}$ on new input (test instance) x which has never been seen before, we obtain prediction $\widehat{y} = \widehat{f}(x)$ where $\ell(\widehat{y}, y)$ is low.

There can be many loss metrics, error metrics, accuracy metrics, and so on, to evaluate a model. But we strive to distinguish: loss ℓ is the one that *our* model $\widehat{f}$ has been trained to minimise, in expectation.

The expectation is with respect to Y since this (the true label) is the quality we do not know[1]; i.e., we *do not know* the value of the label (y) associated with the observation (x), and we *do not know* distribution P either. Only x (the data point) and ℓ (the error metric) may be given to us (and if they are not, we have to obtain it, and decide on it, respectively, before proceeding).

Statistics and probability can help us estimate (i.e., to *model*) P, and *optimisation algorithms* can help us with the argmin. In the real-world, many practical aspects (data wrangling, *computer science* and programming, ...) are required to implement, and to complete the pipeline; and ensure successful deployment. In deployed form, the solution to Eq. (1.4) is commonly called (in the popular media) 'the AI'.

The following steps are necessary (though not necessarily in this order) to approach a machine learning problem:

1. Obtain and curate data (preprocessing) $\mathcal{D}$
2. Choose/design your performance metric (accuracy, or error/loss function ℓ)
3. Decide on a model/representation (for $\widehat{f}$)
4. Employ a learning algorithm (optimization method) to produce $\widehat{f}$

Table 1.1: Major steps involved in a machine learning problem

By now steps 1. and 2. should be becoming clear. We will look at steps 3. and 4. later, in Section 2.

1.4 Types of Machine Learning

Machine learning is always about minimizing expected error (or equivalently), but the different domain of inputs ($\mathcal{X}$), outputs ($\mathcal{Y}$), data available to us and underlying concept, and different loss functions, all contributes to a vast diversity of tasks which can be categorised by many different aspects. –

Firstly, if we have some labels $y_1, \ldots, y_n$ in our training set then we can approach a supervised machine learning task.

If the domain of the labels is continuous (real-valued numbers, e.g., $\mathcal{Y} = \mathbb{R}$), then we have a regression task. If the domain of the labels is discrete, e.g., $\mathcal{Y} = \{0, 1\}$ or, more generally, $\mathcal{Y} = \{1, \ldots, k\}$, then we have a classification task (binary classification, and multi-class/k-class classification, respectively). There is a close overlap with regression if we are looking for probabilistic output, e.g., $P(y = 1|\boldsymbol{x})$ (the probability that instance $\boldsymbol{x}$ belongs to class 1) because $P \in [0, 1] \subset \mathbb{R}$ provides a continuous value. Both regression and

[1]We use upper case Y sometimes to explicitly point out that this is a random variable, in the statistical sense, *not* a concrete value in a dataset

classification encompass an enormous number of tasks, often going under other names such as: forecasting, filtering, smoothing, interpolation, extrapolation, recommender systems, predictive maintenance, and so on.

If we do *not* have labels y_i associated with instances $\boldsymbol{x}_i$ in the training set, we may still build $\widehat{f}$ to assign labels to $\boldsymbol{x}$s anyway. This is called unsupervised machine learning. If we assign discrete values, we may be doing clustering analysis. If we consider continuous values as labels, we can approach tasks such as dimensionality reduction, representation learning, or density estimation (where $\widehat{f}(\boldsymbol{x})$ is the relative probability of observing $\boldsymbol{x}$ in the distribution $p(\boldsymbol{x})$).

In reinforcement learning, we learn an agent/policy to produce outputs which are actions to take in an environment, and receive inputs which are states (or observations of states in that environment), rather than a data set. This is a complex context, because with our actions we can affect future observations (future inputs) received by our agent; essentially the model generates its own training data by interacting with the environment.

1.5 Practical Advice for Tackling a Machine Learning Problem

In practice (machine learning in the real world), it is normal to find the most challenging aspect of machine learning to be understanding/setting up/formulating a problem (What are the features? What is the label? What is the appropriate error metric?). It is normal if the most time consuming part of the machine-learning pipeline is preprocessing (tabulating, checking and cleaning data, dealing with missing values, outliers, feature selection, feature removal, feature extraction, . . .) and it is normal that this task never seems to finish, even in advanced stages of a project.

Once you have defined your problem, it would be unusual if you find this class of problem has not been tackled before; i.e., you can usually expect that there is an off-the-shelf method to solve the problem. Often, in practice, machine learning is almost only about defining a problem then choosing, and parametrizing a method (rarely do we have to design a new one from scratch). Although 'choosing a method' sounds quick, a good theoretical understanding of machine learning, and the different algorithms, is essential to guide this choice in terms of which method and its parameters, or otherwise decide that a modification/novel method is required.

Warning signs: You get 100% accuracy. A 'dumb'/baseline model (e.g., predicts 0 for everything) or a simple statistic (e.g., predict always the mean, or majority class label without considering the input) performs as well or almost as well as your model. Even worse: you forgot to use a baseline at all. You cannot explain your results clearly and intuitively to a machine learning expert. You cannot explain your results clearly and intuitively to a domain expert[2] who is *not* an expert in machine learning. You are unsure of the limitations of your model or about the uncertainty associated with its predictions. You went straight for deep learning but can't justify why that was the necessary/best choice (as opposed to a classical [and usually more understood and reliable] method). You're not having fun.

[2]A 'domain expert' not necessarily in the academic sense. If you are training a computer vision system to recognise pictures of cats, a domain expert is someone who knows what a cat looks like

Chapter 2

Regression

So far we have covered some fundamental terminology, and discussed machine learning from a high-level and abstract point of view; as an *algorithm* using *data* to build a *model* with which we can make predictions that minimise expected *loss/error*. And we covered some general limitations and danger-points to be aware of (potential bias, measures of confidence on predictions, a need for explainability, . . .).

We are now going to commit to some concrete decisions (make some specific assumptions) regarding the *data-error-model-algorithm* combination. Namely, in the context of regression.

We begin by looking at ordinary least squares (OLS) regression, i.e., linear regression. Then, non-linear regression via basis functions (e.g., polynomial regression). Finally, we look at least squares regression via gradient descent, which offers a more robust optimization ('learning') than OLS.

Main Concepts

By the end of this week you should know

- How to recognise (and set up) a regression problem
- With respect to ordinary least squares (OLS), what is (cf. Table 1.1):
  - the data looking like (in matrix/vector form)
  - the error/loss metric
  - the representation of the model
  - the optimization routine/algorithm

  i.e., the derivation of OLS from a machine learning point of view (the *why* of each equation is most important than the *how*
- Why OLS may be insufficient, i.e., when the associated assumptions may fail; and
- . . . which of these assumptions may be overcome by polynomial regression
- The main notions of gradient descent (GD); why/when to prefer GD over OLS; how is it not a closed form solution; will it converge, how do we know that, what does that mean in practical terms?

You'll need some matrix algebra and some matrix calculus to work through the material in this chapter.

See the Section A.2 and Section A.3 in the Appendix if you are not familiar or need a refresher.

2.1 Ordinary Least Squares (OLS)

The method of ordinary least squares (OLS) is well known to many areas of mathematics and science and lays a base for almost all methods we will look at in this course. We will approach it here from a machine learning point of view.

2.1.1 Data

We assume training data in the form of inputs $\mathbf{X}$ and output labels $\mathbf{y}$ (as elaborated above in Eq. (1.1) and Eq. (1.2), respectively). In regression, we assume output labels in the real domain, i.e., $y \in \mathbb{R}$.

Example 2.1.1- House Pricing (Toy Example) – Data

Consider the Price of a house (y, in €) given its floor Area (x_1, in m^2) and Distance to the town centre (x_2, in km). Our data could look as follows:

	Area (m^2)	Dist. (km)	Price (×1000 €)
i	x_1	x_2	y
1	58	1.2	260
2	65	1	280
3	80	1.5	420
4	120	8	320
n	250	20	270

2.1.2 Model

We consider the following model:

$$\begin{aligned}
\mathbf{w} &= [w_1, \ldots, w_d]^\top \\
\widehat{y}_i &= \mathbf{x}_i \mathbf{w} = \widehat{f}(\mathbf{x}_i) = \sum_{j=1}^{d} x_{i,j} \cdot w_j = x_{i,1} w_1 + \cdots + x_{i,d} w_d \\
\widehat{\mathbf{y}} &= \mathbf{X}\mathbf{w} = [\widehat{f}(\mathbf{x}_1), \ldots, \widehat{f}(\mathbf{x}_n)]^\top
\end{aligned} \tag{2.1}$$

i.e., the model is represented by a vector of weights $\mathbf{w}$ (sometimes equivalently called *parameters* and denoted $\boldsymbol{\theta}$ or *coefficients* and denoted $\boldsymbol{\beta}$); specifically one weight for each feature. Note how a prediction is obtained by a linear combination of these weights.

Note: For convenience of notation, we mostly avoid the explicit specification of an intercept (w_0, also known as the bias term). This simplification is perfectly valid, under any of a number of assumptions, for example: $x_1 = 1$ always, or y has mean 0 such that no intercept is needed (only slope).

2.1.3 Metric

We consider the squared error (Fig. 2.1) as the loss metric, such that

$$\ell(y_i, \widehat{y}_i) = (y_i - \widehat{y}_i)^2 = e_i^2$$

is the loss wrt the i-th training example and its residual error e_i. When we sum this error over all examples in the training set, the error can be viewed as the residual sum of squares:

$$E(\mathbf{w}) = \sum_{i=1}^{n} \ell(y_i, \widehat{y}_i) = \sum_{i=1}^{n} e_i^2 \tag{2.2}$$

and thus written as a function of weights $\mathbf{w}$.

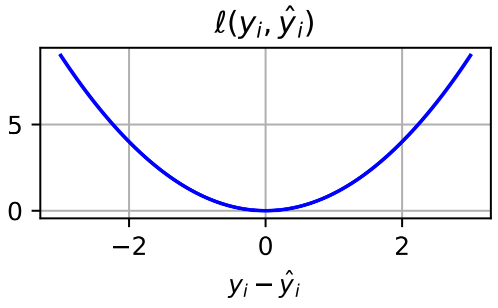


Figure 2.1: Squared Error

We may express all the residual errors in vector form, thus:

$$\begin{aligned}
\mathbf{e} &= [e_1, \ldots, e_n]^\top \\
&= \mathbf{y} - \widehat{\mathbf{y}} \\
&= \mathbf{y} - \mathbf{X}\mathbf{w} \quad \triangleright \text{ From Eq. (2.1)} \\
E(\mathbf{w}) = \mathbf{e}^\top\mathbf{e} &= (\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) \qquad (2.3)
\end{aligned}$$

i.e., which is equivalent to Eq. (2.2). Have a look at Eq. (A.1) if the development so far is not obvious.

2.1.4 Optimization (learning algorithm)

First, here's a warm-up/refresher: how to minimize a function (x, w, and y are all scalars) $E(w) = (y - wx)^2$ wrt w? We set the derivative to 0 and solve for w:

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}w}E(w) &= \frac{\mathrm{d}}{\mathrm{d}w}(y - xw)^2 \quad \triangleright \text{ where } xw = \widehat{y} \\
0 &= 2(y - xw)\frac{\mathrm{d}}{\mathrm{d}w}(-xw) \\
0 &= -(2y - 2xw)x \\
0 &= -2yx + 2xwx \\
2yx &= 2xwx \\
xw &= y \\
w &= y/x
\end{aligned}$$

This is just a *slope* of best fit!

Now again, but for for Eq. (2.3) wrt $\mathbf{w}$, setting to 0, and solving for $\mathbf{w}$:

$$\begin{aligned}
\nabla_{\mathbf{w}} E(\mathbf{w}) &= \nabla_{\mathbf{w}} \mathbf{e}^\top \mathbf{e} \\
&= \nabla_{\mathbf{w}} (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) && \triangleright \text{ From Eq. (2.3)} \\
&= \nabla_{\mathbf{w}} (\mathbf{y}^\top - (\mathbf{X}\mathbf{w})^\top)(\mathbf{y} - \mathbf{X}\mathbf{w}) && \triangleright \text{ Vector/matrix transpose, See Eq. (A.5)} \\
&= \nabla_{\mathbf{w}} \left(\mathbf{y}^\top\mathbf{y} - \mathbf{y}^\top\mathbf{X}\mathbf{w} - (\mathbf{X}\mathbf{w})^\top\mathbf{y} + (\mathbf{X}\mathbf{w})^\top(\mathbf{X}\mathbf{w})\right) \\
&= \nabla_{\mathbf{w}} \left(\mathbf{y}^\top\mathbf{y} - 2(\mathbf{X}\mathbf{w})^\top\mathbf{y} + \mathbf{w}^\top\mathbf{X}^\top(\mathbf{X}\mathbf{w})\right) && \triangleright \text{ Scalar `trick'; See Eq. (A.6)} \\
&= \nabla_{\mathbf{w}} \left(\mathbf{y}^\top\mathbf{y} - 2(\mathbf{X}\mathbf{w})^\top\mathbf{y} + \mathbf{w}^\top(\mathbf{X}^\top\mathbf{X})\mathbf{w}\right) && \triangleright \text{ Association; See Eq. (A.3)} \\
&= \nabla_{\mathbf{w}} \left(\mathbf{y}^\top\mathbf{y} - 2\mathbf{w}^\top\mathbf{X}^\top\mathbf{y} + \mathbf{w}^\top(\mathbf{X}^\top\mathbf{X})\mathbf{w}\right) && \triangleright \text{ Transposes; See Eq. (A.4)} \\
0 &= -2\mathbf{X}^\top\mathbf{y} + 2(\mathbf{X}^\top\mathbf{X})\mathbf{w} && \triangleright \text{ Derivative of quadratic; Eq. (A.10). Set it to 0.} \quad (2.4) \\
\mathbf{X}^\top\mathbf{y} &= (\mathbf{X}^\top\mathbf{X})\mathbf{w} && \triangleright \text{ Now solve for } \mathbf{w}\ldots \\
\mathbf{w} &= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} && (2.5)
\end{aligned}$$

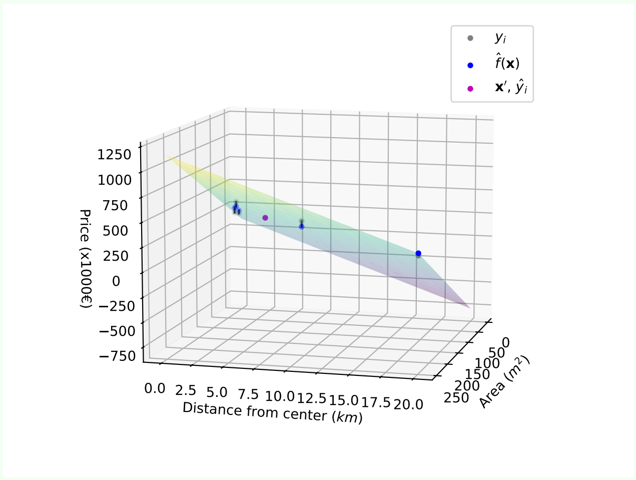
and we walk away with $\mathbf{w}$ – it is our 'trained' model. Machine learning! At this point, we might denote this vector $\widehat{\mathbf{w}}$ to specifically show that we will deploy this version of the weights for making predictions. To be more specific, we could denote $\widehat{\mathbf{w}}_{\mathsf{OLS}}$ to show that it is the weights provided by OLS in particular – not necessarily the same ones discovered by some other optimisation method. The point is to distinguish this solution of particular weight-values from the 'generic' or 'abstract' $\mathbf{w}$ in the derivation above.

Example 2.1.2- House Pricing – Model (continued from Ex. 2.1.1)

According to OLS assumptions (namely, of linearity) if $w_1 = 4.21$, then adding 10 of m^2-floor-area adds €$42k$ to the price:

$$\begin{aligned}
y = f(\mathbf{x}) &= w_0 \cdot 1 + w_1 \cdot x_1 + w_2 \cdot x_2 = \mathbf{x}\mathbf{w} && \triangleright \text{ where } x_0 = 1 \\
&= 99.8 + 4.2x_1 - 43.1x_2
\end{aligned}$$

However, this increase may *only be true when close to the centre* (i.e., feature-feature interaction). In addition, it may make a difference if we are adding 50 m^2 to a small house vs adding to an already-large house (non-linearity, non-additivity).



Remark: We can check that the result Eq. (2.5) is a minimum by taking the derivative of Eq. (2.4) again. We find:

$$
\begin{aligned}
&= \nabla_{\mathbf{w}}(\mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y}) \\
&= \nabla_{\mathbf{w}}\mathbf{X}^\top \mathbf{X}\mathbf{w} \\
&= \mathbf{X}^\top \mathbf{X}
\end{aligned}
$$

which is positive semi-definite (equivalent to finding a positive scalar number when working in a single dimension) and thus – yes – it is a minimum.

Sometimes closed-form solutions like this do not 'feel like' machine learning, but we should take this option whenever we believe the necessary assumptions match our problem. One might argue that least squares is a basic tool of statistics, not a machine learning algorithm. The difference (if there is to be one) is only how we see the outcome. From the machine learning point of view: we have a predictive function, of parameters $\widehat{\mathbf{w}}$, that we can deploy and will provide us with predictions $\widehat{y}$ for *any* new future instances $\mathbf{x}$.

The main message from this section: OLS makes particular choices regarding the main components of machine learning (data format, error metric, model representation, and optimization routine; listed in Table 1.1) – it should be clear which ones.

2.2 Issues with OLS

Or, more precisely: issues with the *assumptions* behind OLS. As stated above, OLS was built on the main assumption of a linear function of attributes. For example, when $d = 3$ features, we assume that the *true* relationship between $\mathbf{x}$ and y is

$$y = \underbrace{w_1^\star x_1 + w_2^\star x_2 + w_3^\star x_3}_{f_\star(\mathbf{x}) = \mathbf{x}\mathbf{w}_\star} + \epsilon \quad \text{where irreducible error } \epsilon \sim \mathcal{N}(0, \sigma^2)$$

for some particular $\mathbf{w}_\star$. This assumption entails, more precisely:

1. linearity: the relationship between $\mathbf{x}$ and y is linear

2. no collinearity among features $x_1, \ldots, x_d$

3. data points have been sampled identically and independently (iid) with respect to the generating (ground-truth) distribution; i.e., $(x_i, y_i) \sim P_\star(X_i, Y_i)$ where $P_\star(X_i, Y_i)$ some real-world concept

4. homoskedasticity, i.e., σ is fixed; so ϵ is also always distributed the same way for any $\mathbf{x}$

5. the irreducible error has mean 0, and is Gaussian-distributed; hence $\epsilon \sim \mathcal{N}(0, \sigma^2)$

We will overcome assumptions 1 and 2 in Sections 2.3 and 2.4–2.5. We continue to hold on the iid assumption; see Section 2.7 for a brief exploration of this topic. Meanwhile we discuss the final assumptions in Sections 2.6 and Section 2.8; note that these do not affect the prediction itself but rather the error term – our uncertainty around the prediction.

Example 2.2.1- Forecasting career earnings; Version II (Interpretation of parameters).

Let's come back to the example based on predicting earnings (Example 1.2), but this time with two different features: *gender* and *height*. The 'gender pay gap' is widely acknowledged (about 13% in the EU[a]; i.e., being of gender female is associated with a 13% decrease in salary). Some studies[b] further suggest that each additional centimetre of height is associated with a 1% increase in annual income. These are linear relationships. We might imagine something like:

$$\mathsf{income} = f(\mathbf{x}) = 40000 - 5200 \cdot \mathsf{female} + 400 \cdot \mathsf{cm\_above\_average} + \epsilon$$

where $(\mathsf{female} = 0) \Leftrightarrow$ male. Note: 5200 is 13% of 40000.

If this were the true $f_\star$, OLS would be a perfect choice to model this relationship from data. However, the reality is certainly more complex: it could be that the relationship between height and income is non-linear: the height effect moving from 210cm to 211cm could be less than from 169cm to 170cm (feature non-linearity). Furthermore, it could be that height contributes as a different factor for men and women (feature interaction). Or, it even might be that women are paid less on account of their height not their gender (feature correlation).

These considerations bring out the limitations of OLS: even with this two-variable problem, we have just broken several of the assumptions listed just above.

[a]according to [1]

[b]e.g., [3, 8], though to emphasise: we are only looking for a thought-provoking example sufficiently complex to highlight the limitations of OLS; not a serious investigation into the subject matter – which we leave to the domain experts for now!

2.3 Basis Functions and Polynomial Regression (for Non-Linearity)

To fix the issues suggested in Example 2.2 we need to remove the assumption of linearity; i.e., non-linear regression. For this, we can introduce the concept of a basis function $\phi(\mathbf{x})$ to be used as a feature.

For example (feature interaction) if an expert suggests that each additional cm of height above average (feature x_2) may provide a different penalty/gain for men ($x_1 \neq 1$) than for women ($x_1 = 1$), then we might use extra features

$$\phi_1(\mathbf{x}) = (1 - x_1)x_2 \quad \text{and} \quad \phi_2(\mathbf{x}) = x_1 x_2$$

to model that possibility.

When non-linear relationships are suspected, the basis function can provide a non linearity, e.g.,

$$\phi_3(\mathbf{x}) = x_2^2$$

suggests that the influence of feature x_2 on y is quadratic wrt its value. A domain experts can help choose such functions for us.

When we have no domain insight to work from, we can turn to polynomial regression which is a special case of non-linear regression where the basis functions are attributes and/or combinations of attributes raised powers $1, \ldots, k$ (e.g., $\boldsymbol{\phi}(\mathbf{x}) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2, (x_1x_2)^2, \ldots, x_1^k, x_2^k, (x_1x_2)^k]$) and hope that the learning algorithm will calculate reasonable weights for them.

Non-linear regression in this context is as simple as fitting linear regression (e.g., OLS) to these features $\boldsymbol{\phi}$. So OLS provides $\boldsymbol{\phi}\mathbf{w}$; i.e., a linear combination, and indeed, the resulting hyperplane is linear in the space where $\boldsymbol{\phi}$ lives. This means that we assume a linear relationship between $\boldsymbol{\phi}$ and y, whereas ϕ-functions are non-linear, thus there has been a non-linear transformation from $\mathbf{x}$ to $\widehat{y}$. That is why, if we plot $\mathbf{x}$ vs $\widehat{y}$ we will see a non-linear decision surface.

2.4 Dealing with Collinearity and too Many Features

In OLS we need to invert $(\mathbf{X}^\top \mathbf{X})$ (or, in the context of non-linear regression via basis functions: $(\mathbf{\Phi}^\top \mathbf{\Phi})$) but this is not possible if variables are collinear! (because the matrix is not full rank; and determinant is zero). As a consequence, we cannot find a unique solution $\widehat{\mathbf{w}}$.

Consider when x_2 is deterministically dependent on x_1, as in $x_2 = 2.2x_1$:

$$\begin{aligned}\widehat{y} &= \widehat{w}_1 x_1 + \widehat{w}_2 x_2 \\ &= \widehat{w}_1 x_1 + \widehat{w}_2 (2.2 x_1) \\ &= (\widehat{w}_1 + 2.2\widehat{w}_2) x_1\end{aligned}$$

There are an infinite number of possibilities for $\widehat{\mathbf{w}} = [\widehat{w}_1, \widehat{w}_2]$! We have this problem whenever we have more columns/features than rows/examples; the matrix cannot have linearly-independent columns. Having more features than examples is very likely when using many polynomial basis functions!

2.5 Gradient Descent (GD)

With gradient descent (GD) we find $\widehat{\mathbf{w}}$ in a different way than in Eq. (2.5) (the OLS solution). Essentially we stop the derivation at Eq. (2.4),

$$\begin{aligned} 0 = \mathbf{g} &= -2\mathbf{X}^\top \mathbf{y} + 2(\mathbf{X}^\top \mathbf{X})\mathbf{w} \quad \triangleright \text{ from Eq. (2.4)} \\ &= -\mathbf{X}^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) \quad \triangleright \text{ \ldots and simplified a bit} \end{aligned} \tag{2.6}$$

thus obtaining vector $\mathbf{g}$ of the same dimensions as $\mathbf{w}$. But also note that the formula *contains* $\mathbf{w}$, thus it is *not* a closed-form solution; in fact, not a solution of any kind yet. This $\mathbf{g}$ is our gradient. It points in the direction of higher error. So we descend away from that direction (move in the other direction), i.e., towards a solution. How much to descend? We introduce a learning rate α to determine that. We start with some (any, random) $\mathbf{w}$, choose a sensible α (more on this later), and repeat the process

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \mathbf{g}$$

until we are satisfied (ideally, until convergence). Good news: if the function is convex (and in least squares it is!) and for a sensible choice of learning rate (it should not be too large, and not be 0) the solution will converge to $\min_{\mathbf{w}} E(\mathbf{w})$. It means, in practice, we will certainly find $\mathbf{w}$ which achieves the smallest possible error on our training data. Even better: the theory of universal approximation (e.g., [6]) tells us with the right basis functions, we can reduce this error to 0, i.e., obtain perfect performance (essentially, by interpolating the data points with our function)! Bad news: perfect performance only on our training data. In machine learning we care about the error on our test data that we haven't seen yet. More bad news: the theory of universal approximation does not tell us which are the right basis functions.

Gradient descent [and its variants] is a very well-used (and very successful) algorithm in machine learning today. It is the base of practically all deep neural networks, and deep learning is essentially not more than an automatic way to learn the ϕ that we discussed earlier.

2.6 Dealing with Heteroskedasticity

So far we have overcome the issue of non-linearity (in Section 2.3) and co-linearity (Section 2.4; via GD – Section 2.5). Let's now look at the issue of heteroskedasticity, which is when the underlying distribution is a

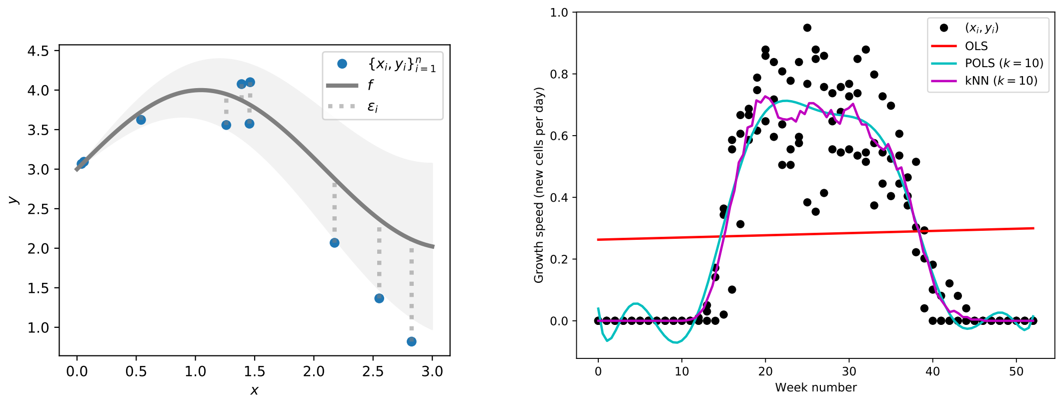


Figure 2.2: Synthetic ground-truth $f_\star(\mathbf{x}) \pm \sigma(\mathbf{x})$ (left) showing heteroskedasticity. And a real-world example (right): celular growth (y radial cells) in Scots pine tree vs week of the year ($x \in \{1, \ldots, 52\}$) in a region in Southern Finland. Although polynomial least squares (POLS; as per Section 2.3) takes care of non-linearity, it does not take care of heteroskedasticity: note that $\sigma(x)$ is near zero for certain ranges of x, and significantly larger for other ranges (there is almost no variance/error over the coldest/darkest winter months, since the tree does not grow at this time). On the other hand, k-nearest neighbours (kNN) makes no assumption on distribution of errors.

different shape for different locations (points) $\mathbf{x}$. Figure 2.2 shows an example where both the assumptions of linearity and homoskedasticity are broken, and $\sigma(\mathbf{x})$ is now a function of $\mathbf{x}$ (rather than constant σ).

We easily overcome the assumption of non-linearity with polynomial basis functions (denoted POLS in the figure) but heteroskedasticity is still an issue: note that the fit around ranges $x = 1, \ldots, 10$ and $x = 42, \ldots, 52$ is a bit 'off'.

2.7 A Word on the IID Assumption

The iid assumption is essential to machine learning.

If $\mathbf{x}_{i+1}$ is strongly correlated to $\mathbf{x}_i$ (or y_{i+1} strongly correlated to y_i), we cannot strictly consider error terms E_i and E_{i+1} separately; rather we should model them together (notice that ϵ_{i+1} and ϵ_i are now interdependent), which creates a major headache: we could not then reduce $E(\mathbf{w})$ to a decomposable sum over individual instances as we have been doing until now ($E(\mathbf{w}) = \sum_{i=1}^n E_i$).

If the presence of minor correlation among instances, it may be OK to proceed anyway. However, the presence of significant dependence suggests a time series/signal processing problem, and we will need to model the dependence and/or decorrelate the data (break the dependence) before proceeding with off-the-shelf machine learning methods (such as gradient descent).

2.8 Gaussian Errors

The assumption of Gaussian errors, refers to

$$\begin{aligned} \epsilon &\sim \mathcal{N}(0, \sigma^2) \\ y &\sim \mathcal{N}(\mathbf{x}\mathbf{w}, \sigma^2) \end{aligned} \tag{2.7}$$

This is a reasonable assumption most of the time.

If residual errors e_i don't look like they are Gaussian noise, it's usually a sign that other assumptions have failed. Normally, the errors should be void of clear patterns that might aid a prediction (such patterns should be captured by f instead!) Fig. 4.1 shows such a case.

Later (Week 3.2, and beyond) we will consider the loss metric as the negative log likelihood which provides us a probabilistic interpretation:

$$\begin{aligned}
E(\mathbf{w}) = -\log p(\mathbf{y};\mathbf{w}) &= -\log \prod_{i=1}^{n} p(y_i|\mathbf{x}_i) && \triangleright \text{ negative log likelihood}\\
&= -\sum_{i=1}^{n} \log \mathcal{N}(y_i|\mathbf{x}_i\mathbf{w},\sigma^2) && \triangleright \text{ log Gaussian, from Eq. (2.7)}\\
&= -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mathbf{x}_i\mathbf{w}) - \frac{n}{2}\log(2\pi\sigma^2) && \triangleright \text{ some math}\\
0 &= \nabla\Big[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i-\mathbf{x}_i\mathbf{w}) - \frac{n}{2}\log(2\pi\sigma^2)\Big] && \triangleright \text{ set derivative to 0}\\
&= -\sum_{i=1}^{n}(y_i-\mathbf{x}_i\mathbf{w})\mathbf{x}_i && \triangleright \text{ got rid of terms not a function of } \mathbf{w}\\
&= -\mathbf{X}^\top(\mathbf{y}-\mathbf{X}\mathbf{w}) && \triangleright \text{ the gradient}\\
\mathbf{w} &= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} && \triangleright \text{ closed form solution as per OLS}
\end{aligned}$$

which is exactly the result we obtained in Eq. (2.5).

2.9 Conclusion

With the right basis functions we can model *any* regression function to any desired degree of accuracy. Since the error function is convex, we can be guaranteed to find the minimum (i.e., solve the problem!). However, we cannot know beforehand which basis functions to use, nor even how many of them will be required. Even if we could 'calculate' this, we are only guaranteed to obtain this desired accuracy on the training data. Hence, our journey into machine learning must continue! However, we have already some really important ingredients. A linear combination of weights and attributes plus a non-linearity plus gradient descent-based learning comprises a huge part (we might say, most of) modern machine learning and AI.

Chapter 3

Classification

This week we commit to a different sort of model, namely classification models (or classifiers), which can be used to make *qualitative* decisions, where the *class* label is chosen from a set of categories, e.g., $y \in \{0, 1\}$. We concentrate mainly on the method of logistic regression, which builds on many of the same concepts as in linear regression. We also look at the method of k-nearest neighbours. Both these approaches provide the main building blocks of more advanced methods that we will look at later.

Main Concepts

By the end of this week you should know

- How to recognise (and formulate) a classification problem
- Why is logistic regression a *discriminative* classifier
- With respect to logistic regression: what is (here, referring to the list in Table 1.1)
  - the data looking like (in tabular/matrix/vector form)
  - the error/loss metric
  - the representation of the model
  - the optimization routine/algorithm

  i.e., the derivation of logistic regression (again, the *why* is more important than the details of the *how*)
- A probabilistic view of machine learning (and logistic regression, in particular)
- How k-nearest neighbours works; why is it a *non-parametric* method
- Various considerations of logistic regression vs k-nearest neighbours

Earlier (Section 1.2) we discussed a toy example regression problem of modeling earnings ($y \in \mathbb{R}_+$ in k€), suggesting that such a prediction could be used to make a decision on providing a loan [or not] to an applicant specified by instance $\boldsymbol{x}$. We could also model this decision directly as a classification problem:

$$\widehat{y} = h(\boldsymbol{x}) \quad \text{where} \quad y \in \begin{cases} 0 & \Rightarrow \text{Deny applicant } \boldsymbol{x} \text{ the loan} \Leftrightarrow \text{predict applicant will default,} \\ 1 & \Rightarrow \text{Give applicant } \boldsymbol{x} \text{ the loan} \Leftrightarrow \text{predict applicant will not default} \end{cases}$$

where, in classification, we usually denote h (for $h$ypothesis) instead of f (for regression $f$unction); but the concepts introduced so far (model, error/loss, weights, ...) hold also in this case. In fact, classification can be (and often is) approached as a regression problem:

$$f(\boldsymbol{x}) = P(Y = y \mid \boldsymbol{X} = \boldsymbol{x}) \tag{3.1}$$

where (wrt the loan example) $P(Y = 0 \mid \boldsymbol{X} = \boldsymbol{x}) = 0.9$ implies that *our model believes* that the client specified by $\boldsymbol{x}$ would likely default (with probability 0.9) on the loan, i.e., the loan should be denied. Since output $0.9 \in \mathbb{R}$, this is regression.

And since $0.9 \in [0, 1]$ (it is a probability) then, we can make a classification (hard, 0–or–1 decision), by passing it over a threshold (usually 0.5), as

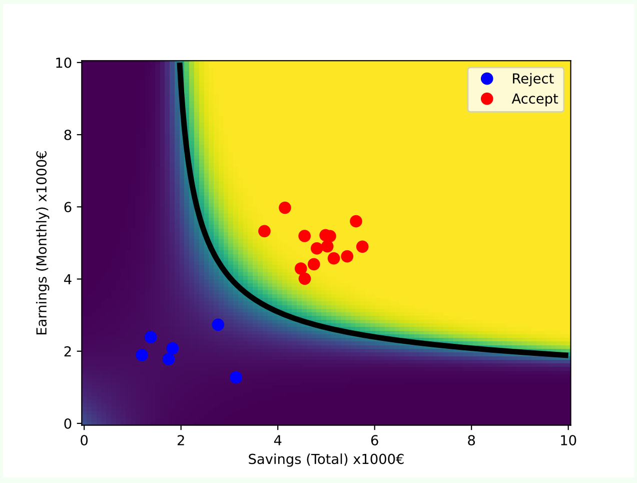
$$\begin{aligned}\widehat{y} = h(\boldsymbol{x}) &= \underset{y \in \{0,1\}}{\operatorname{argmax}} P(Y = y \mid \boldsymbol{X} = \boldsymbol{x}) && (3.2)\\ &= [\![f(\boldsymbol{x}) \geq 0.5]\!] && (3.3)\\ &= \begin{cases} 0 & f(\boldsymbol{x}) < 0.5 \\ 1 & f(\boldsymbol{x}) \geq 0.5 \end{cases}\end{aligned}$$

where $[\![\cdot]\!]$ be an indicator function, such that $[\![A]\!] = 1$ iff condition A holds.

In this chapter we focus on discriminative classifiers: we want to learn to discriminate (distinguish) between class 0 and class 1, when making a decision. Later we will look at generative classifiers.

Example 3.0.1- A Toy Example: Obtaining a Loan

Suppose several clients go to the bank and ask for a 100k loan. The bank considers their monthly earnings and their savings and makes a decision. A logistic regression model, as we consider in this chapter, might look like the following (note, the data and example is completely fictitious):



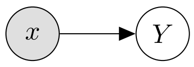
3.1 Discriminative vs Generative Classifiers

First, recall Bayes rule:

$$p(y \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid y)p(y)}{p(\boldsymbol{x})}$$

and associated concepts, presented in Appendix A.4.

In discriminative models, we only want to discriminate between one class $y = 0$ and another $y = 1$ (i.e., make a prediction or *classify* in instance). As a directed probabilistic graphical model (Bayesian Network):



where x is shaded to indicate that it has been observed, and the arrow represents conditional probability.

Discriminative classifiers are a function of distribution $p(Y|x)$. The 'function' we are referring to is h; the decision rule of a classifier or regression model:

$$\widehat{y} = h(\boldsymbol{x}) = \operatorname*{argmax}_{y\in\{0,1\}} p(y \mid \boldsymbol{x}) = \operatorname*{argmax}_{y\in\{0,1\}} p(y, \boldsymbol{x}) = \operatorname*{argmax}_{y\in\{0,1\}} p(\boldsymbol{x} \mid y)p(y) \tag{3.4}$$

The fact that each term is equal does not mean that all p are the same; they refer to different distributions which stem from different modelling decisions. In this chapter we are concerned with the first one (discriminative classifier):

$$\widehat{y} = h(\boldsymbol{x}) = \operatorname*{argmax}_{y\in\{0,1\}} p(y \mid \boldsymbol{x})$$

Later, in Chapter 11, we will look at generative classifiers where we instead model

$$\widehat{y} = h(\boldsymbol{x}) = \operatorname*{argmax}_{y\in\{0,1\}} p(\boldsymbol{x} \mid y)p(y) \tag{3.5}$$

It means that in this Chapter we need to model $P(Y \mid \boldsymbol{x})$. This is exactly what is represented Eq. (3.1).

3.1.1 0/1 Loss in Classification

In discriminative classification we frequently refer to our goal as

$$\widehat{y} = \operatorname*{argmax}_{y\in\{0,1\}} P(Y = y|X = x)$$

(as in Eq. (3.2)). What follows (in this subsection) is a derivation that explains why this makes sense, with respect to classification accuracy. First note that, equivalently (in the inverse sense, of an error/loss metric) the 0/1 loss

$$E_{0/1}(y, \widehat{y}) = [\![y \neq \widehat{y}]\!] = 1 - [\![y = \widehat{y}]\!] = \begin{cases} 0 & \text{if } y = \widehat{y} \\ 1 & \text{otherwise} \end{cases} \tag{3.6}$$

is equivalent to classification accuracy (maximizing accuracy, is the same as minimizing 0/1-loss).

As in regression, we write out the expected conditional expected error, and try to minimize it. It's a non-differentiable function, but we can simplify:

$$\begin{aligned}
\mathbb{E}[E_{0/1}(Y, y) \mid x] &= \sum_{y\in\{0,1\}} E(y, \widehat{y}) \cdot P(Y = y|X = x) \\
&= \sum_{y\in\{0,1\}} (1 - [\![y = \widehat{y}]\!]) \cdot P(Y = y|X = x) \quad \triangleright \text{ from Eq. (3.6)} \\
&= 1 - P(Y = y|X = x) \\
\widehat{y} &= \operatorname*{argmin}_{y\in\{0,1\}} \Big[1 - P(Y = y|X = x)\Big] \quad \triangleright \text{ minimize this} \\
\widehat{y} &= \operatorname*{argmax}_{y\in\{0,1\}} P(Y = y|X = x) \quad \triangleright \text{ \ldots, i.e., maximize this}
\end{aligned} \tag{3.7}$$

This is a MAP estimate (maximum a-posteriori). Where do we get $P(Y|X = x)$ from? We should learn it; and logistic regression is one way to do it.

3.2 Logistic Regression

In logistic regression, we approach the problem similarly to linear regression: 1) assume a linear combination of attributes and weights, 2) identify an error function, 3) obtain the gradient on the error function wrt some parameters $\boldsymbol{w}$, and then 4) apply gradient descent. We do the same here with one additional ingredient[1]: we wrap a non-linearity around the linear combination such that

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}) = p(y = 1 \mid \boldsymbol{x}) \tag{3.8}$$

The $f(\boldsymbol{x})$ here is the same one as Eq. (3.1), and the non-linearity refers to the sigmoid function (also known as the logistic function) denoted σ, such that

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \tag{3.9}$$

which provides a convenient mechanism to 'squish' any value $a \in [-\infty, +\infty]$ into range $\sigma(a) \in [0, 1]$, and thus in turn allowing us to treat this as a probability (the right-hand-side of Eq. (3.1)). We pass this number over a threshold to get a classification $\widehat{y} \in \{0, 1\}$ as described by Eq. (3.3).

Generally we are interested in evaluating a classifier (*decision function*, h) under classification accuracy: $\frac{1}{n}\sum_{i=1}^{n} [\![\widehat{y}_i = y_i]\!]$ (+1 for each correct classification). However this function is non-differentiable because of the indicator function $[\![\cdot]\!]$. Instead, we usually use as a surrogate loss metric (recall: we would typically distinguish the *loss* as the metric optimised by the algorithm, as possibly different from the *evaluation metric* that will tell us how well a classifier solves the problem it is deployed on). A loss metric well suited to our needs, of having a smooth and differentiable function, is log loss:

$$E(\boldsymbol{w}) = \sum_{i=1}^{n} \ell(y_i, \sigma_i) = -\sum_{i=1}^{n} \{y_i \log(\sigma_i) + (1 - y_i)\log(1 - \sigma_i)\} \tag{3.10}$$

where $\sigma_i \equiv \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i) = p(y_i = 1 \mid \boldsymbol{x}_i)$ is the sigmoid function expressed in Eq. (3.9).

Once you are convinced that this loss makes sense, the next task is to minimise it wrt our parameters. As you know from the previous chapter, this means having an expression for the gradient of $E(\boldsymbol{w})$ wrt $\boldsymbol{w}$:

$$\begin{aligned}
\nabla_{\boldsymbol{w}} E(\boldsymbol{w}) &= \nabla_{\boldsymbol{w}} \left\{ -\sum_{i=1}^{n} \{y_i \log(\sigma_i) + (1 - y_i)\log(1 - \sigma_i)\} \right\} \quad \triangleright \text{ Plugged in from Eq. (3.10)} \\
&= -\sum_{i=1}^{n} \left(\frac{y_i \nabla_{\boldsymbol{w}} \sigma_i}{\sigma_i} + \frac{\nabla_{\boldsymbol{w}}(1 - \sigma_i)}{1 - \sigma_i} - \frac{y \nabla_{\boldsymbol{w}}(1 - \sigma_i)}{1 - \sigma_i} \right) \quad \triangleright \text{ Derivative of log} \\
&= -\sum_{i=1}^{n} \left(\frac{y_i(1 - \sigma_i)\cancel{\sigma_i} \nabla_{\boldsymbol{w}} \boldsymbol{w}^\top \boldsymbol{x}_i}{\cancel{\sigma_i}} - \frac{\cancel{(1 - \sigma_i)}\sigma_i \nabla_{\boldsymbol{w}} \boldsymbol{w}^\top \boldsymbol{x}_i}{\cancel{1 - \sigma_i}} + \frac{y\cancel{(1 - \sigma_i)}\sigma_i \nabla_{\boldsymbol{w}} \boldsymbol{w}^\top \boldsymbol{x}_i}{\cancel{1 - \sigma_i}} \right) \quad \triangleright \text{ \ldots of } \sigma \\
&= -\sum_{i=1}^{n} (y_i(1 - \sigma_i)\boldsymbol{x}_i - \sigma_i \boldsymbol{x}_i + y_i \sigma_i \boldsymbol{x}_i) \\
&= -\sum_{i=1}^{n} (y_i \boldsymbol{x}_i \cancel{- y_i \sigma_i \boldsymbol{x}_i} - \sigma_i \boldsymbol{x}_i \cancel{+ y_i \sigma_i \boldsymbol{x}_i}) \quad \triangleright \text{ cancel} \\
&= \sum_{i=1}^{n} \boldsymbol{x}_i(\sigma_i - y_i) \quad \triangleright \text{ cleanup} \\
&= \mathbf{X}^\top(\boldsymbol{\sigma} - \boldsymbol{y}) \quad \triangleright \text{ vector/matrix notation (recall: } \mathbf{X} \in \mathbb{R}^{n \times d}\text{; see Eq. (A.2) – where } e_i = \sigma_i - y_i)
\end{aligned}$$

[1] and one change notation/convention: $\boldsymbol{x}$ is now assumed to be a *column vector*, and hence we write $\boldsymbol{w}^\top \boldsymbol{x}$ rather than $\boldsymbol{x}\boldsymbol{w}$ as before. We still assume that $\mathbf{X} \in \mathbb{R}^{n \times d}$

That wasn't too crazy; only a lot of simple rules to recall at the right time. Such a derivation only needs to be be done once for the development of any machine learning method. We only need the last line (the gradient)! And it looks similar to Eq. (2.6), apart from the presence of $\boldsymbol{\sigma}$ (and the transpose, due to the use of column vectors rather than row vectors $\boldsymbol{x}$, here).

Now we can apply gradient descent (recall: Section 2.5).

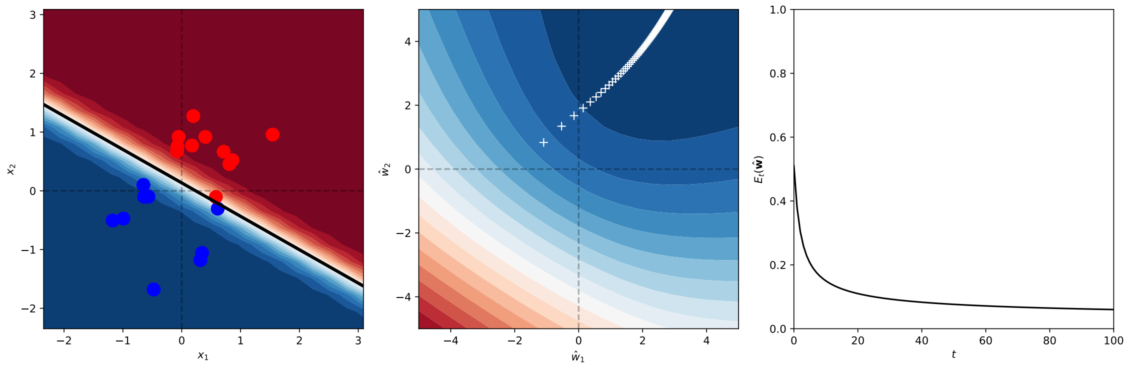


Figure 3.1: Gradient descent for logistic regression. It is important to distinguish between the data space (left), weight space (centre), error signal (right).

At this point we are only a small step from the deep neural networks which have revolutionised so many domains in the recent decade.

3.3 k-Nearest Neighbours (kNN)

The k-nearest neighbours method (kNN) is lazy and non-parametric. We do *not* learn parameters $\boldsymbol{w}$. To make a prediction for test point $\boldsymbol{x}$, we simply find the k points in the training set which are closest to it, then:

$$P(Y = y \mid \boldsymbol{X} = \boldsymbol{x}) = \frac{1}{k} \sum_{\kappa=1}^{k} y_\kappa \tag{3.11}$$

where $y_1, \ldots, y_k$ are the labels of the k closest instances (the labels belonging to the $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\} \subset \{\boldsymbol{x}_i\}_{i=1}^n$ that are most similar to test instance $\boldsymbol{x}$) we found; known as the neighbourhood of $\boldsymbol{x}$. Getting from here (Eq. (3.11)) to a classification $\widehat{y} \in \{0, 1\}$ is as simple as Eq. (3.1)—Eq. (3.3).

kNN does not have a error function *explicitly* minimized during training because there *is no training*. It is nevertheless possible to express the error function that kNN minimizes (implicitly); but we will look into this later.

The kNN method can be applied to regression also (kNN regression). This is the same: our prediction $\widehat{y} = f(\boldsymbol{x})$ is simply the right hand side of Eq. (3.11). There is no assumption required about the underlying distribution; the resulting *fit* can be highly non-linear; indeed sometimes *too* non-linear (we discuss the implications in more depth later).

In Fig. 2.2 we saw that kNN's decisions can be too local, making it susceptible to overfitting. Could we combine kNN and parametric linear regression?

A method resulting from the combination of linear regression and kNN is local regression, or locally-weighted least squares (LWLS).

The best of both worlds, and the worst of both worlds. This worst case complexity becomes cubic for a given $(\boldsymbol{x}, \boldsymbol{x}_i)$-pair, at test time. In other words, we suffer the curse of dimensionality already well-known to kNN.

Consider (even in only two dimensions) Figure 3.2. This is an increasing problem as the dimensionality d rises.

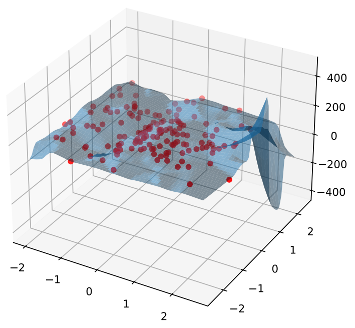


Figure 3.2: GPS coordinates (standardized, to mean $\mathbf{0}$) $\boldsymbol{x} = [x_1, x_2]$ vs elevation (in metres) y around the *Plateau de Saclay*. Despite the relative smoothness of local regression (as depicted) it is still at risk where data is sparse: see, e.g., the right-hand side providing an estimate of $-400m$ below sea level.

How relevant is kNN in the age of modern deep neural architectures? Actually, kNN can still be remarkably effective. It can be used in conjunction with neural networks (for an improved/compacted feature space representation, i.e., mitigate the curse of dimensionality). It can even be itself expressed as a neural network. Finally, and perhaps most importantly, kNN provides us insight into core concepts of machine learning which are omnipresent in AI. Even in modern generative AI models like ChatGPT, there is a legitimate debate about how much is learning and generation, and how much is kNN-style memorisation.

Chapter 4

Overfitting and Regularization

This is arguably the most important chapter. Machine learning is essentially about minimising expected error (or loss). Given that we have such powerful tools available to us, it is overfitting; lack of generalisation; that is by far the most common pitfall in machine learning. It results from the fact that we use our training data as an approximation to the expectation (of expected error), therefor we overfit the training data. Regularization is the solution (mitigation) to overfitting.

By now, some questions have begun arising: *Should I use k-nearest neighbours or logistic regression? How to know if I should use basis functions? How many basis functions should I use? Which basis functions?* There is a precise answer to this question: you should use exactly the basis functions **which allow your model to best minimise expected loss**. What is the expected loss? Unfortunately, there is no precise answer to *this* question; not in practical terms, at least. But in this chapter we move closer to being able to estimate it, and understand its composition.

Main Concepts

By the end of this week you have a good idea of

- How is minimizing expected loss machine learning?
- What is bias and variance in the context of expected loss?
- What is overfitting and what is it caused by
- How can we mitigate it (i.e., with regularization), particularly via
  - Ridge regression
  - Cross-validation
- How might underfitting occur and manifest itself.

Have a look at Section A.4 and Section A.5 if you need refresher on random variables and expectations; we will use them a lot in this chapter.

4.1 A Discussion on Rational Agents and Uncertainty

A *rational agent* (i.e., intelligent agent/what the popular media would refer to as 'an AI') is one that minimizes expected loss; or, equivalently, that acts optimally based on its **beliefs** and according to observed **evidence** and clear **preferences**. It should model any **uncertainty**. As in artificial intelligence in general, in machine learning we aim that the model we produce performs as a rational agent, where

- **beliefs** = model f (and, more generally, $P(y|\boldsymbol{x})$, of which f is a function),
- **evidence** = observations $\boldsymbol{x}$,
- **preferences** = loss function ℓ; and where
- **uncertainty** is modeled by an expectation $\mathbb{E}$.

Normally, humans are remarkably good at avoiding overfitting in their learning and decision making processes. Hence, it it almost a paradox that we must train ourselves carefully to avoid it when training models in a machine learning context.

But sometimes, even as intelligent agents, we also fall into modelling problems. The following is a thought exercise to explore the topic (inspired vaguely by this article).

Let $y = 1$ denote *it is necessary to act urgently to prevent a catastrophic outcome from climate change* and let $y = 0$ be the opposite (it is *not* necessary to act). And so $\widehat{y} = 0$ is a prediction[1] that *it is not necessary to act urgently to prevent climate change* (implying we either believe climate change does not exist or that it may exist but does not require any action to address); and of course $\widehat{y} = 1$ is the prediction to the contrary.

As an exercise: complete the example (i.e., plug in your own numbers) by specifying these elements yourself: decide on a number for $P(Y = 1|\boldsymbol{x})$ the probability you believe in catastrophic results from climate change if we do not act urgently, according to the evidence $\boldsymbol{x}$ you have observed so far (in the most general sense: scientific literature, news, social media posts, anecdotal discussion from your entourage, ...); noting that this gives you automatically $P(Y = 0|\boldsymbol{x}) = 1 - P(Y = 0|\boldsymbol{x})$. And fill in the values for $\ell(y, \widehat{y})$ for all $y \in \{0, 1\}$ and $\widehat{y} \in \{0, 1\}$; noting that it could be that $\ell(0, 0) = 0$ (no loss/cost involved); yet $\ell(1, 1) > 0$ (cost involved; even though prediction is correct).

Your $0 < P(Y = y|\boldsymbol{x}) < 1$ numbers should express **uncertainty** (we do not live in a fully-observed deterministic world). The only ingredient left to being a rational agent is make a decision (prediction) taking into account this uncertainty. We deal with that next.

How do you know if you have the 'right answer'? When you minimize *your* expected loss. This may not align with everyone else's! This is due to a different loss function, different evidence, or holding different beliefs (interpretations of evidence). If you have the same loss function, evidence, and beliefs as someone else, yet your conclusions differ, one of you is behaving as an *irrational* agent.

[1]Or better, *prescription*. Often, in machine learning, the output of the model implies an action

4.2 Conditional Expected Loss

Recall: In machine learning we want to build $\widehat{f}$. Given some instance $\boldsymbol{x}$, it should return a prediction

$$\widehat{y} = \widehat{f}(\boldsymbol{x})$$

What value should $\widehat{y}$ take, optimally speaking? The answer to this question is precisely the value which minimizes the loss;

$$\min_{\widehat{y}} \ell(y, \widehat{y})$$

When we have y, this is a trivial optimization problem: the minimum of $\ell(y, \widehat{y})$ can be obtained simply by setting $\widehat{y} = y$. But of course, in general (in practice/under deployment) we don't have y (otherwise there would be no point in building a model to predict it in the first place). The fact that we do not know what y is represents a source of uncertainty. What should we do? Which $\widehat{y}$ should we choose? The answer: choose $\widehat{y}$ which minimizes conditional expected loss:

$$\mathbb{E}_{Y \sim P(Y|\boldsymbol{x})}[\ell(Y, \widehat{y})] = \sum_{y \in \{0,1\}} \ell(y, \widehat{y}) P(y|\boldsymbol{x}) \tag{4.1}$$

where Y is discrete (with probability mass function P, i.e., a classification problem), or, in the continuous case,

$$\mathbb{E}_{Y \sim P(Y|\boldsymbol{x})}[\ell(Y, \widehat{y})] = \int \ell(y, \widehat{y}) p(y|\boldsymbol{x}) \, \mathrm{d}y$$

with probability density function p (i.e., a regression problem).

This is simply the definition of a conditional expectation of a function (conditioned on $\boldsymbol{x}$, of function $\ell(Y, \widehat{y})$).

Notice that we take the expectation wrt Y, whereas $\widehat{y}$ is fixed (because we fixed it; it is our prediction), and also $\boldsymbol{x}$ is fixed (it was given to us as observed evidence, let us not tamper with the evidence). It is essentially a weighted sum; a sum over $\ell(y, \widehat{y})$, weighted by $P(y|\boldsymbol{x})$.

Recall we wanted to minimize this, therefore (taking the example of discrete-valued Y):

$$\widehat{y}^* = \operatorname*{argmin}_{\widehat{y} \in \{0,1\}} \left[\sum_{y \in \{0,1\}} \ell(y, \widehat{y}) P(y|\boldsymbol{x}) \right] \tag{4.2}$$

where $\widehat{y}^*$ corresponds to our best prediction, under this minimization.

Before looking further – check you have understood so far. For example, recall your P and ℓ from the exercise in Section 4.1. Now solve Eq. (4.2). Do you agree that the result $\widehat{y}^*$ seems *rational*?

Just as we have expressed the uncertainty wrt y in terms of P, now we must : we do not have P!

We only have training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ which has been generated or drawn from P.

4.3 Minimizing Empirical Loss

Because we do not have access to the *true* $P(Y|x)$, we cannot directly minimize the expected loss. Another option is to instead minimize the empirical loss, using our training data:

$$\ell(\boldsymbol{w}) = \ell(\{y_i, \widehat{f}(x_i)\}_{i=1}^n) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, \widehat{f}(x_i)) \tag{4.3}$$

(where we have explicitly replaced $\widehat{y}$ with $\widehat{f}(x)$ here, so that we can remember that $\widehat{f}$ is represented by $\boldsymbol{w}$; and since our training data is fixed, then ℓ is a function of $\boldsymbol{w}$). Notice that P does not appear here! The data (which has been sampled from true P) represents P instead. The *law of large numbers* tells us, with infinite samples, minimizing Eq. (4.3) is equivalent to minimizing Eq. (4.1). Since we are given each y_i and x_i in the training data, the loss function is a function of $\boldsymbol{w}$ only (everything else is fixed/given as part of the problem). Therefore we just minimize Eq. (4.3) wrt $\boldsymbol{w}$:

$$\widehat{\boldsymbol{w}} = \operatorname*{argmin}_{\boldsymbol{w}} \ell(\boldsymbol{w})$$

and produce $\widehat{f}$ (represented by [model representation] $\widehat{\boldsymbol{w}}$). The problem is: *we do not have infinite samples*, so the equivalence doesn't hold; i.e., we are minimizing the wrong error function. And that is where the problem of overfitting comes in (we overfit the training data quantity and fail to generalize to all possible data that our model might see).

Over-fitting the training data is like concluding that there will be no catastrophic effects from human-induced climate change, because this did not happen in the past, and we have no evidence of this ever happening[2]. In other words, a potential failure to model uncertainty about the future.

4.4 Bias-Variance Decomposition

We have so far been discussing uncertainty regarding our prediction, under a fixed model (producing that prediction). Yet, what about the uncertainty regarding which model to start with? We need to think about what we *do and do not* know (i.e., our uncertainty). We *do* have ℓ (let's suppose squared error[3]), we *do* have data $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$. But, we *do not* have the true P from which the data was sourced/generated.

Our uncertainty is around Y and $\widehat{F}$. $\widehat{F}$? But isn't $\widehat{f}$ determined deterministically, when we train our model? Indeed, $\widehat{f}$ is fit on the training data which is sourced from an unknown distribution; i.e., a random variable; so the modelling decision inherits this uncertainty, and becomes a random variable itself; $\widehat{F}$. Intuitively: whatever model we choose (let's say, linear regression), we will get a different $\widehat{f}$ for a different training set drawn from the same distribution.

Then,

$$\begin{aligned}
\mathbb{E}_{Y,\widehat{F}}[\ell(Y,\widehat{F})|X=x] &= \mathbb{E}[(Y-\widehat{F})^2|X=x] \quad \triangleright \text{ let } \widehat{F} \text{ be shorthand for } \widehat{F}(\boldsymbol{x}) \qquad (4.4)\\
&= \mathbb{E}[Y^2 - 2Y\widehat{F} + \widehat{F}^2]\\
&= \mathbb{E}[Y^2] - \mathbb{E}[2Y\widehat{F}] + \mathbb{E}[\widehat{F}^2]\\
&= \mathbb{E}[Y^2] + \mathbb{E}[\widehat{F}^2] - \mathbb{E}[2Y\widehat{F}] \quad \triangleright \text{ reorder terms}\\
&= \underbrace{\mathbb{V}[Y] + (\mathbb{E}[Y])^2}_{\mathbb{E}[Y^2]} + \underbrace{\mathbb{V}[\widehat{F}] + (\mathbb{E}[\widehat{F}])^2}_{\mathbb{E}[\widehat{F}^2]} - \mathbb{E}[2Y\widehat{F}] \quad \triangleright\ \mathbb{V}[Y] = \mathbb{E}[Y^2] - \mathbb{E}[Y]^2\\
&= \mathbb{V}[Y] + \mathbb{V}[\widehat{F}] + (\mathbb{E}[Y])^2 + (\mathbb{E}[\widehat{F}])^2 - 2\mathbb{E}[Y\widehat{F}] \quad \triangleright \text{ reorder; and } \mathbb{E}[2] = 2\\
&= \mathbb{V}[Y] + \mathbb{V}[\widehat{F}] + f^2 + (\mathbb{E}[\widehat{F}])^2 - 2f\mathbb{E}[\widehat{F}] \quad \triangleright\ \mathbb{E}[Y] = \mathbb{E}[f+\epsilon] = f, \text{ and } \mathbb{E}[\epsilon] = 0\\
&= \sigma^2 + \mathbb{V}[\widehat{F}] + \left\{f^2 + (\mathbb{E}[\widehat{F}])^2 - 2f\mathbb{E}[\widehat{F}]\right\} \quad \triangleright\ Y \sim \mathcal{N}(f,\sigma^2), \text{ so } \mathbb{V}[Y] = \sigma^2\\
&= \sigma^2 + \mathbb{V}[\widehat{F}] + (f - \mathbb{E}[\widehat{F}])^2\\
&= \sigma^2 + \mathbb{V}[\widehat{F}] + (\mathbb{E}[f - \widehat{F}])^2 \quad \triangleright\ \mathbb{E}[f] = f\\
&= \underbrace{\sigma^2}_{\text{irreducible}} + \underbrace{(\mathbb{E}[f-\widehat{F}])^2}_{\text{bias}^2} + \underbrace{\mathbb{V}[\widehat{F}]}_{\text{variance}}
\end{aligned}$$

[2]Catastrophic effects implying effects worse than presently

[3]If we don't have a loss function, we should go decide/design one quickly and then come back; see earlier chapters

This is the expression of the bias-variance trade-off (plus the irreducible error – see Section 1.2 if you have forgotten what that means).

The '*reducible* error' (bias + variance) embodies the trade-off, caused by uncertainty in the data set which represents only a random finite sample of infinite possible datasets from P. With an infinite data set we can hope to reduce bias and variance 0 (because there is no more uncertainty, after infinite data we have 'seen it all'). But in all other cases (including all real-world problems), we have to make appropriate model choices. The no free lunch theorem (this is a real thing, see [13]) tells us that no model can be best for every problem setting.

4.5 Unbiased Estimators

Here is another question to check your understanding: is the ordinary least squares estimator an unbiased estimator? In other words, is the bias of ordinary least squares regression 0? Here we prove this (denoting the model by its parameters):

$$\begin{aligned}
\mathbb{E}[\mathbf{w} - \widehat{\mathbf{w}}] &= \mathbf{0} \\
\mathbf{w} &= \mathbb{E}[\widehat{\mathbf{w}}] \\
&= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}\mathbb{E}[\boldsymbol{Y}] && \triangleright \text{ where } \boldsymbol{Y} \text{ contains the uncertainty; not the same as } \mathbf{y} \\
&= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}(\mathbf{w}\mathbf{X} + \mathbb{E}[\boldsymbol{\epsilon}]) && \triangleright\ \boldsymbol{Y} = \mathbf{f} + \boldsymbol{\epsilon} \text{ where } \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}\sigma) \\
&= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}(\mathbf{w}\mathbf{X}) && \triangleright \text{ and so, therefore, } \mathbb{E}[\boldsymbol{\epsilon}] = \mathbf{0} \\
&= (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{X})\mathbf{w} \\
&= \mathbf{w}
\end{aligned}$$

4.6 Bias vs Consistent Estimators

An estimator is consistent if $\widehat{\mathbf{w}}$ *converges* in probability to $\mathbf{w}_\star$ as $n \to \infty$ (i.e., optimality). An estimator is unbiased if $\mu = \mathbb{E}[\widehat{\mathbf{w}}] = \mathbf{w}_\star$ which is true for all sample sizes. It is possible to be unbiased but not converge to any value. It is also possible to be bias, but still converge as $n \to \infty$. Example: The estimator $\hat{\mu} = \bar{x} + \frac{1}{n}$ for $\mathbb{E}[X]$. We note that $\mathbb{E}[\bar{x} + \frac{1}{n}] \neq \mu$ for finite n (so it is a biased estimator). But, as $n \to \infty$, the $\frac{1}{n}$ term vanishes, and $\hat{\mu} \to \mathbb{E}[X]$ converges to true mean, so it is a consistent estimator.

4.7 Bias and Variance in Human Judgement

Despite our many flaws as human beings, we excel at *not* overfitting. In other words, we are excellent when it comes to generalisation to new settings. When presented with uncertainty due to a relatively small amount of data relative to the complexity of a problem, we tend to reduce our expected error by employing massive bias. We say *bias* here as a neutral term. But of course in the social sense it can, sometimes, indeed be bad. And this can be a problem as we rely on bias to reduce variance, i.e., make consistent decisions that largely reduce our overall error (at the cost of some particular errors). Example: you have to translate 'nurse' into French (as in, "*the patient called for the nurse*" without any additional context). If you provide $\widehat{y}$ = 'infirmière' = f('nurse') you reduce your expected error, since most nurses in the world are female, but you risk propagating social bias. It is argued, e.g., [4], that humans excel at generalisation to new tasks precisely because we trade off bias against variance; this is not always negative. This does not mean we should be proud of our social bias. Indeed, on the one hand there are obvious negative consequences from bias. And, working together as a group, humans are able to reduce bias without increasing variance (error); an analogy to ensemble methods in machine learning – we look at those later.

4.8 Ridge Regression and Lasso Regression

The techniques of ridge regression and lasso regression are forms of regularization which basically means mitigating overfitting. We noted that bias can help us guard against overfitting. Therefore it should not be a surprise to know that the resulting models from both these techniques are *biased* estimators. Both methods involve restricting the magnitude (length, or norm) of the weight vector $\mathbf{w}$. Which norm? Exactly, that is the difference between the two methods.

Consider a simple example of fitting a line to a set of data points $\{(x_i, y_i)\}_{i=1}^n$; each x_i is a single dimension (e.g., as in Figure 1.1), thus we only regularize (restrict) the slope w of the line (a single parameter). Ridge regression adds a penalty λw^2 to the loss function. Whereas Lasso regression adds penalty $\lambda|w|$. It means we want a slope such that the line fits the training data well, but also at a trade-off that the norm of w is not too large;

- in Lasso: the absolute-value norm, also known as the L_1-norm[4], $\|w\|_1 = |w|$; and
- in Ridge: the Euclidean norm, $\|w\|_2 = w^2$). (also known as the L_2 norm).

With ridge regression we have the advantage that w^2 has a straightforward derivative (thus a closed form solution is possible). Lasso regression has the advantage that it may set coefficients to 0 and thus performs feature selection (if $w_j = 0$, then the j-th feature x_j is worthless as the model only ever sees $w_3 \cdot x_3 = 0 \cdot x_3 = 0$, and thus the column x can be discarded from the data). Why does this not happen with Ridge?

Note that if w is already small, then w^2 will be *really* small (thus incurring tiny penalty). Suppose we have a budget to reduce the magnitude of some parameter w by 0.1. Note:

$$\begin{aligned} 100^2 - 99.9^2 &= 20 \quad \triangleright \text{ Large } w;\ \|w\|_2 \text{ norm (Ridge)} \\ |100| - |99.9| &= 0.1 \quad \triangleright \text{ Large } w;\ \|w\|_1 \text{ norm (Lasso)} \\ 0.1^2 - 0^2 &= 0.01 \quad \triangleright \text{ Small } w;\ \|w\|_2 \text{ norm (Ridge)} \\ |0.1| - |0| &= 0.1 \quad \triangleright \text{ Small } w;\ \|w\|_1 \text{ norm (Lasso)} \end{aligned}$$

Given a choice, Ridge will eliminate much more penalty by reducing the magnitude of large parameters. To Lasso it makes no difference if the parameter being reduced is large or small (the penalty reduction is the same), therefore it will may reduce parameters all the way to 0.

Recall that we are talking about the penalty function which is added to the loss function ℓ.

The ridge and lasso techniques can be combined into what is known as the elastic net method.

4.9 Validation Sets and Cross Validation

In machine learning we run intro trouble (with overfitting) when we target our training data instead of test data, as we never have access to our 'real' test data (which our deployed model will see). But we can simulate test data, by cutting off a bit of the training set. We call this a validation set. For example, we fit our model on 60% of the training set, and use the remaining 40% to check model performance.

[4]Though we avoid the L_p notation here to avoid confusion with loss metric ℓ; even though there is a connection to the loss metric, it is not an equivalence here, since we are discussing a penalty which is *added* to the loss metric

60%? More, less? How much of our data can we spare for a validation set (knowing that *we cannot use this same data for training*, thus we reduce the size of our training set)? The technique of k-fold[5] cross validation *allows us to use all of it.* Suppose a dataset of $n = 100$ points, if $k = 5$ we can train on instances $1, \dots, 80$, then test ('validate') on instances $81, \dots, 100$; also train on instances $1, \dots, 20, 41, \dots, 100$ and test on instances $21, \dots, 40$; and so on for $k = 5$ sets. Since we can measure mean and variance over multiple sets, this is a great way to gauge uncertainty.

In deep learning (when training sets are typically huge) it is typical to only have a single validation set.On small datasets we can consider leave-one-out cross validation where $k = n$.

In any case, it is fundamental that no test point should ever have been seen at training time.

4.9.1 Cross validation and hyper-parameter selection

Many methods have hyper-parameters, such as the λ in ridge (or lasso) regression, number (and type) of basis functions, or the k in k-nearest neighbours (quick check: it should be clear the difference from 'regular' *trainable* parameters such as w_j). Which λ works best? More accurately, our question is: Which λ provides the best bias-variance trade-off (i.e., reduces expected loss the most) on the unseen test data of our problem setting.

It is a question involving Eq. (4.4) since we have uncertainty about the test label (it is not available to us) as well as uncertainty around the model (which involves λ). Let $\ell_l(\lambda)$ be the empirical loss of model $\widehat{f}_\lambda$ on the l-th test fold, where this $\widehat{f}_\lambda$ has been trained on the l-th training fold. Suppose we want to estimate which of values $\lambda \in \{0.1, 1, 10\}$ is best for our problem. We can choose via the results k-fold cross validation:

$$\hat{\lambda} = \operatorname*{argmin}_{\lambda \in \{0.1,1,10\}} \frac{1}{k} \sum_{l=1}^{k} \ell_l(\lambda)$$

(recall: on the test folds!).

4.9.2 Cross validation and preprocessing

Often we need to standardize our data, or some other kind of preprocessing. Should we standardize the original training set and *then* do cross validation, or should we standardize each train and validation set independently.

Answer: We should always try to adhere as closely as possible to the principal of treating validation sets as true test data; we *should not look at it or touch* it at training time. Therefore we should standardize on the training fold (this will involve calculating means and variances for each feature) and then *use the same* means/variances to standardize the validation set for that fold.

4.10 Underfitting

Due to the massive amounts of power and capacity at our fingertips these days (deep learning frameworks), and our temptation to use it, we are unlikely to experience underfitting on a simple problem. On the other hand, there are plenty of complex problems out there for which we do not yet have good models. And an unsuited model is akin to the problem of failed assumptions. Consider the examples Fig. 2.2 and Fig. 2.1.1: the assumption of linearity is not a good one, when approaching these problems. Fig. 4.1 shows how this relates to least squares and its assumptions; namely via the detection of non-Gaussian errors. This issue would be rapidly remedied with a few basis functions, but then we are back to our original question

[5]Beware: *not* the same k as in kNN

(almost literally, the first question of this chapter): what is the right amount/degree of polynomial of basis functions?

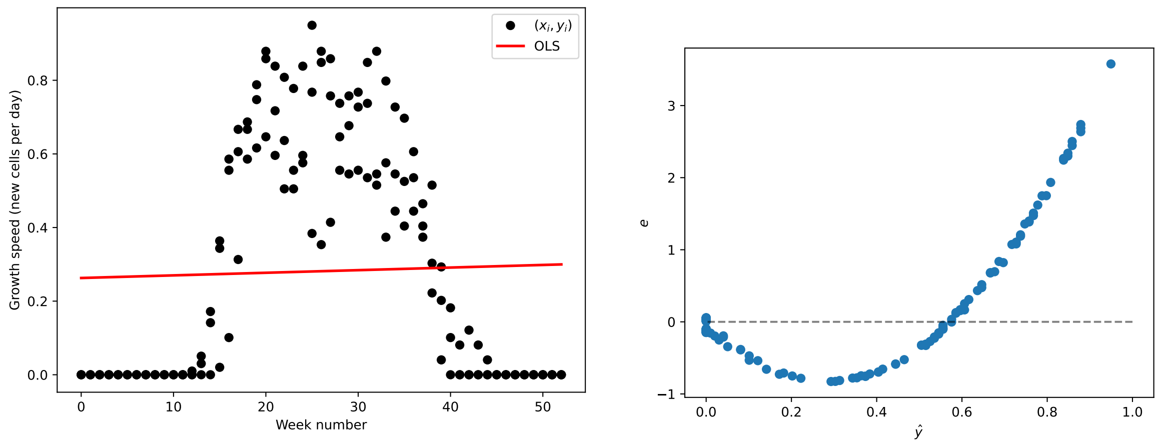


Figure 4.1: Diagnosing a weak model with a residuals plot. OLS (left) and its residual plot (right). Even if unable to view the fit (e.g., due to high-dimensional $\mathbf{x}$), we might suspect there is an under-fitting issue because the [standardized] residual errors are not approximately Gaussian distributed; $\sim \mathcal{N}(0, 1)$.

Part II

Building on Foundations

Chapter 5

Neural Networks: An Introduction

Artificial neural networks play a major role in modern machine learning and artificial intelligence. They are not a new technology, but important developments in the previous decade have lead to impressive performance in a number of relevant application domains.

Main Concepts

By the end of this week (Introduction to Neural Networks) you have a good idea of

- The historical context of artificial neural networks
- What is an artificial neuron, its activation function, and how it relates to methods that you have already learned (linear, and logistic regression)
- What are hidden (latent) neurons (units)/hidden layers and when/why are they needed
- Forward propagation
- Some notions of backpropagation (at least: what is the general idea, why is it needed)

We can formalize several of the methods we've seen so far (e.g., linear regression, logistic regression, polynomial regression)[1] into the framework of neural networks. Or, more specifically, a single neuron ('node' or 'unit'); as exemplified in Figure 5.1; with weights $\mathbf{w}$ and an activation function f.

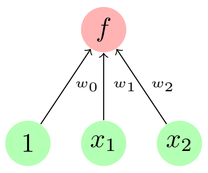


Figure 5.1: An artificial neuron. Each edge represents a weight w_j (here: $\mathbf{w} = [w_0, w_1, w_2]$); the output, or activation, of this neuron is $f(\mathbf{w}^\top \mathbf{x})$.

What is the difference between a *model* f and an *activation function* f? In the case of a single neuron,

[1]Actually, almost any model, including k-nearest neighbours can be put into the framework of a neural network

and in terms of output, there is no difference; we can still write $f(\mathbf{w}^\top \mathbf{x})$ as we have done earlier, to represent the model. However when we have a neural *network* of several neurons (such as the one in Figure 5.2 below), we must distinguish: each neuron has its own activation function, and own set of inputs, $z_1 = f_1(\mathbf{w}_1^\top \mathbf{x})$, $z_2 = f_2(\mathbf{w}_2^\top \mathbf{x})$, $\ldots$, $\sigma = \sigma(\mathbf{u}^\top \mathbf{z})$, and so on. This means that the *model* is represented by multiple activation functions (and correspondingly multiple sets of weights). In neural networks and in particular deep learning, the *model* is often called the representation or architecture, or simply the network.

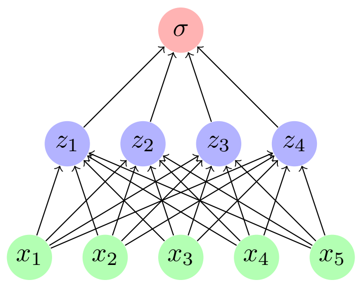


Figure 5.2: A vanilla two-layer multi-layer neural network, often called a multi-layer perceptron (despite the fact that it make use arbitrary activation functions, not just the step function used by the perceptron). It is a deep neural network in the most minimal sense, as it is comprised of more than one layer. In this case, we have used sigmoid activation functions in both layers, including the outer layer.

With reference to Fig. 5.1, we can see the difference between several methods seen until now as simply a different choice of

1. activation function f (e.g., the identity function $f(a) = a$ for linear regression; or the sigmoid activation $f(a) = \sigma(a)$ for logistic regression),
2. error/loss function $E(\mathbf{w})$ (e.g., squared error for OLS, plus a Euclidean norm penalty for ridge, log loss for logistic regression, and so on), and
3. optimization method (e.g., closed-form optimization, or gradient descent) for learning weights

as summarised in Table 5.1.

Table 5.1: Here is a summary of some of the methods seen so far, plus a new one – the perceptron. Note that more variations are possible, e.g., there is a closed form available for ridge regression. The activation functions can be viewed in Fig. 5.3.

Optimization	Error/Loss function	Activation function	Name
Closed form	squared error	identity	ordinary least squares
Gradient descent	squared error	identity	linear regression
Gradient descent	squared error + penalty	identity	ridge regression
Gradient descent	logistic loss	sigmoid	logistic regression
Perceptron algorithm	accuracy	threshold linear unit	perceptron classifier

Recall the no free lunch theorem (we mentioned in Section 4.4), which tells us that the average performance of all possible methods on all possible problems will be equal. In the context of this course, it means: we are not building towards a single best machine learning method; such a method does not exist; we are rather concerned with producing different classes of model which tend to better suited to different types of problem.

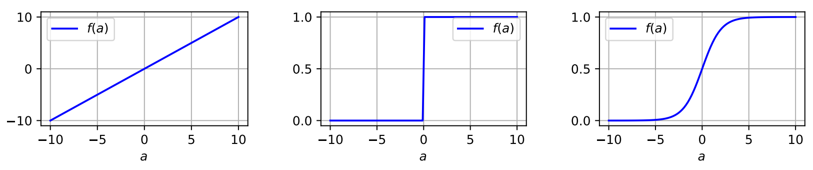


Figure 5.3: Activation functions: identity function, threshold linear unit, and sigmoid function.

And (we mentioned in Section 2.9) the theorem of universal approximation tells us that a neural network of one hidden layer, with an appropriate (non-linear) activation function, can approximate *any* $f(x)$ to *any* desired degree of precision. However, it does not tell us how to train the network; i.e., the theorem also applies to polynomial basis function expansion and 'hard-wired'/'hand-coded' networks. A visual demonstration is provided in Fig. 5.4 and Fig. 5.5.

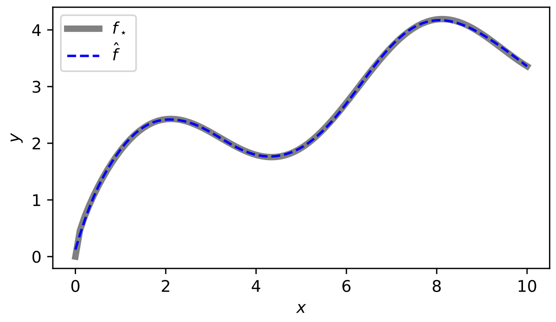


Figure 5.4: A visual hint at universal approximation: in this case only 30 hidden nodes (sigmoid activation function) are used to approximate this function: $y = f(x) = 0.8 \cdot \sin(x) + 1.2\sqrt{x}$. We could (according to the theorem) approximate arbitrarily complex functions to any desired degree of accuracy, with a finite number of hidden nodes. But beware: the theorem does not guarantee how to train such a network (and indeed, here we have generated an unrealistic density of training points).

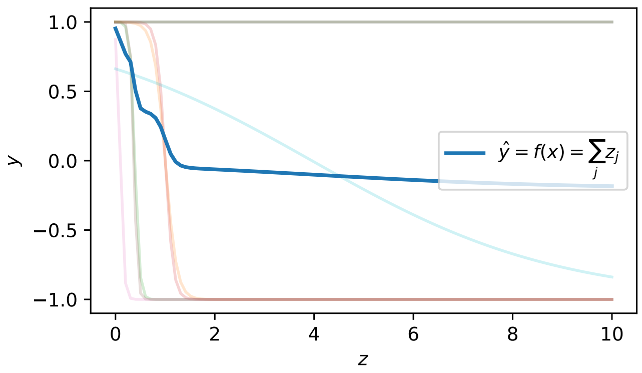


Figure 5.5: Only 10 simple random (untrained) activation functions, shown faintly in the background, have been averaged together to produce the blue curve. Much of what we do in machine learning boils down to weighted averages; where learning is about adjusting weights.

Chapter 6

Deep Learning (Backpropagation)

We have already motivated neural networks, and in particular deep neural networks (containing hidden nodes between the input and output). Now we just need to train such models, i.e., deep learning.

Deep learning (fitting neural networks of arbitrary depth) is performed via error backpropagation, i.e., fitting all learnable/trainable parameters[1]). Learning via error backpropagation is 'just gradient descent' but there are some important qualitative differences, requiring additional study and consideration, when we *go deeper*.

Main Concepts

These are some important concepts that you should understand in the context of deep learning:

- Backpropagation (how it works, how it is implemented; how it is represented as a computational graph).
- Some forms of regularization typically used in deep neural networks; in particular, dropout and early stopping
- Issues with depth, e.g., exploding and vanishing gradients, dead neurons
- An intuition of the abstraction performed by deep neural networks, moving from raw feature inputs up towards high-level class concepts
- A rough idea of how big a network should/can be, and which activation functions to consider and why (e.g., why go beyond the sigmoid activation); why is the rectified linear unit (ReLU) commonly used.
- How to deal with $k > 2$ classes (multi-class classification) and multi-output prediction (in the general case where $\mathbf{y} \in \mathbb{R}^m$)

Back propagation has been used since the 1980s. However, major interest in deep neural networks started to pick up from around 2006, pushed by a number of conceptually small yet qualitatively important details, including:

- Layer-wise pre-training

[1]We do *not* learn the activation functions; the class of activation function is usually considered a hyper-parameter

- Larger quantities of labelled data
- More powerful computers and, particularly, the use of Graphical Processing Units (GPUs) for matrix computations
- Software infrastructures (e.g., TensorFlow, PyTorch)
- Small algorithmic changes, e.g., the use of cross entropy instead of MSE

6.1 A Worked Example: Setting Up

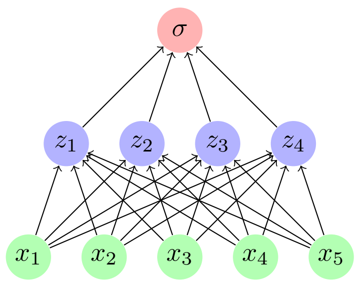
Let's set up a machine learning problem in terms of the four main ingredients (recall: Table 1.1).

6.1.1 Data

Let's suppose dataset $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i=1}^n$ where inputs $\mathbf{x}_i = [x_1, \ldots, x_5]$ (i.e., input space $\mathbf{x} \in \mathbb{R}^5$) are associated with labels $y_i \in \{0, 1\}$.

6.1.2 Model

Let's suppose the network depicted in Figure 5.2 (reproduced here):



The network (model) is specified by weight matrices $\mathbf{W} \in \mathbb{R}^{5\times 4}$ (layer 1, note $d = 5$) and $\mathbf{u} \in \mathbb{R}^{4\times 1}$ (layer 2); with activation function for both layers being the sigmoid function: $f_1 \equiv f_2 \equiv \sigma$ with 2 indicating the top layer, $\sigma \equiv f_2(\mathbf{z})$.

The output of the network, σ, can be converted to a predicted label $\widehat{y}$ via a threshold.

6.1.3 Error metric

Let us use the squared error loss metric:

$$E_i = \ell(y_i, \sigma_i) = \| \underbrace{y_i - \sigma_i}_{e_i} \|_2^2 = e_i^2$$

Squared error is not typically associated with classification, but it has been in the past, and there is no reason why we cannot use it.

6.1.4 Optimisation

We're just missing the component of optimization; i.e., the actual learning. We'll do that with stochastic gradient descent; and for this we need to obtain the gradients via error backpropagation through the network. That is what this chapter is about.

6.2 Forward Propagation

Backpropagation can be called more fully 'error backpropagation'. We need to start by calculating an error, which involves a forward pass/forward propagation *of the input*. We will consider learning via stochastic gradient descent on mini-batches (details in Section 6.5). In the special case of batch size 1, we may train on (update by) one example $\mathbf{x}_i \in \mathbb{R}^{1\times 5}$ at a time.

The forward pass looks like this:

$$\begin{aligned}
\mathbf{x} &\leftarrow \mathbf{x}_i \quad \triangleright \text{ Select/load/impute training instance}\\
\mathbf{z} &= f_1(\underbrace{\mathbf{xW}}_{\mathbf{a}_1}) \quad \triangleright \text{ i.e., } \mathbf{z} \in \mathbb{R}^{1\times 4}\\
\sigma &= f_2(\underbrace{\mathbf{zu}}_{a_2}) \quad \triangleright \text{ i.e., } \sigma \in \mathbb{R}\\
\widehat{y} &= [\![\sigma \geq 0.5]\!] \quad \triangleright \text{ We only do this when network is deployed}\\
y &\leftarrow y_i \quad \triangleright \text{ Load/impute training label, corresponding to input } \mathbf{x}_i\\
E(y,\sigma) &= \|y-\sigma\|_2^2 \quad \triangleright \text{ i.e., } E \in \mathbb{R}_+
\end{aligned}$$

Note that the training error is evaluated on σ and *not* $\widehat{y}$; i.e., $\widehat{y}$ is not involved in the learning process; that is why it is greyed-out.

Note (see Section A.2 if this is note clear):

$$\mathbf{xW} = [x_1, x_2, x_3, x_4, x_5]\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4}\\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4}\\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4}\\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4}\\ w_{5,1} & w_{5,2} & w_{5,3} & w_{5,4}\end{bmatrix} = [\mathbf{xw}_1, \mathbf{xw}_2, \mathbf{xw}_3, \mathbf{xw}_4] = [a_1, a_2, a_3, a_4] = \mathbf{a}_1$$

where $\mathbf{a}_1$ is the activation, which is provided to the activation function:

$$\mathbf{z} = f_1(\mathbf{a}_1) = f_1(\mathbf{xW}) = [f_1(a_1), \dots, f_1(a_4)] \tag{6.1}$$

It should be clear that different dimensions will work similarly (if multiplied/transposed correctly), producing σ. And that the top layer works similarly:

$$\mathbf{zu} = [z_1, z_2, z_3, z_4]\begin{bmatrix} u_1\\ u_2\\ u_3\\ u_4\end{bmatrix} = \sum_{j=1}^{k} z_j u_j = a_2$$

and

$$\sigma = f_2(\mathbf{zu}) = f_2(a_2)$$

where $\sigma \in [0, 1]$.

6.3 Backpropagation

With regard to input $\mathbf{x}$, forward propagation has produced for us $\mathbf{z}$, σ, and *most importantly* (required for the following), the error E (indeed, note that we propagate from input to error and back, not only between input and output). We want to propagate this error back down through the network; by providing a gradient to all trainable parameters.

We already know how to calculate gradients. Note that, using the standard techniques (that we have seen already), gradient $\nabla_{\mathbf{u}} E$ can be obtained as follows (the $\frac{1}{2}$ is for convenience):

$$\begin{aligned}
\mathbf{g}_2 &= \nabla_{\mathbf{u}} E \\
&= \nabla_{\mathbf{u}} \frac{1}{2}(y-\sigma)^2 \\
&= (y-\sigma)\nabla_{\mathbf{u}}(y-\sigma) \\
&= (\sigma-y)\nabla_{\mathbf{u}}\sigma \\
&= \underbrace{(\sigma-y)}_{e} \cdot \underbrace{\sigma(1-\sigma)}_{f_2'} \nabla_{\mathbf{u}}\mathbf{z}\mathbf{u} \\
&= \underbrace{(\sigma-y)\cdot f_2'}_{\delta_2} \mathbf{z} \\
&= \mathbf{z}\cdot\delta_2
\end{aligned}$$

We have gradient $\mathbf{g}_2 \in \mathbb{R}^{4\times 1}$, with which we can update weights $\mathbf{u}$!

But to be doing deep learning, we need to update weights under the surface layer – how to update $\mathbf{W}$? Again, we need the gradient; this time with respect to $\mathbf{W}$. Continuation of backpropagation (for gradient $\mathbf{G}_1$, and noting that $\mathbf{z}$ is a function of $\mathbf{W}$) is as follows. Note, in particular, how information about the error, embodied in δ_2, is used (back-propagated).

$$\begin{aligned}
\mathbf{G}_1 = \nabla_{\mathbf{W}} E = \delta_2 \nabla_{\mathbf{W}}[\mathbf{u}\,\underbrace{f_1(\mathbf{xW})}_{\mathbf{z}}] & \quad \triangleright \text{ Expanding } \mathbf{z} \text{ from Eq. (6.1)} \\
= \delta_2 \mathbf{u}^\top \nabla_{\mathbf{W}}\,[f_1(\mathbf{xW})] & \quad \triangleright \text{ Product rule, } \nabla_{\mathbf{W}}\mathbf{uz} = (\nabla_{\mathbf{W}}\mathbf{u})\mathbf{z} + \mathbf{u}(\nabla_{\mathbf{W}}\mathbf{z}) \\
= \delta_2 \mathbf{u}^\top \odot \mathbf{f}_1' \nabla_{\mathbf{W}}[\mathbf{xW}] & \quad \triangleright \text{ Element-wise multiplication; see A.6} \\
= \mathbf{x}^\top \underbrace{\delta_2 \mathbf{u}^\top \odot f_1'(\mathbf{Wx})}_{\boldsymbol{\delta}_1} & \quad \triangleright \text{ See also, Eq. (A.11)} \\
= \mathbf{x}^\top \boldsymbol{\delta}_1 &
\end{aligned}$$

We have gradient $\mathbf{G}_1 \in \mathbb{R}^{5\times 4}$ (again, it's important to realise that this is exactly the dimensions of $\mathbf{W}$)!

Now we can update our parameters according to gradient descent, using gradients: $\mathbf{G}_1, \mathbf{g}_2$.

6.4 Computational Graphs

A computational graph tells how to compute something. They are sometimes called 'call graphs' or 'dependency graphs' in the context of functional programs. Indeed, computation graphs can be used for many things, like for separating the data from dynamics in video game design, but in the context of deep learning, we will use it to denote the forward and backward passes, for error back-propagation.

Fig. 6.1 shows the computational graph for a single-layer network with a sigmoid output; like logistic regression, except in this example, considering squared error (rather than log loss). When querying this graph for the relationship between w_2 and E we obtain the expression:

$$\begin{aligned}
\frac{\partial E}{\partial w_2} &= \frac{\partial E}{\partial f_2}\frac{\partial f_2}{\partial a_2}\frac{\partial a_2}{\partial w_2} \\
&= \delta_2 \frac{\partial a_2}{\partial w_2}
\end{aligned}$$

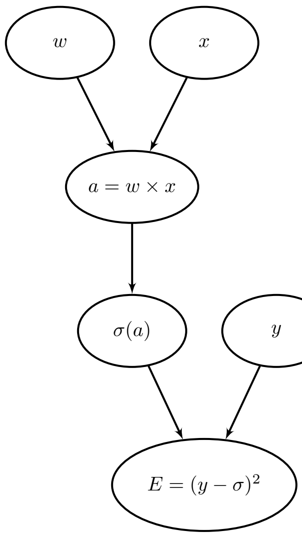


Figure 6.1: A computational graph. This is *not* a neural network; but it shows the computations involved in a neural network's forward pass – all the way to the error.

6.5 Stochastic Gradient Descent (SGD)

Methods of gradient descent is not strictly tied to backpropagation, but backpropagation provides the necessary gradients with which we can perform gradient descent. In the context of deep neural networks, for reasons of tractability, we need to look at stochastic gradient descent (SGD). The only difference (from 'standard', or batch gradient descent) is that we take random samples (we will call them mini-batches), so we descend the error function stochastically. In the case of a mini-batch of size 1, we take a single example $\mathbf{x}_i$ from the training data, and carry out gradient descent, by calculating the error against the label y_i, and the network's output. A larger number of iterations is required, wrt batch gradient descent, but this is by far worth the trade-off against the complexity of using the entire dataset on each iteration (when the dataset is large, as is often the case in deep learning applications). The optimal size of the mini-batch is another hyper-parameter. For the generalisation to minibatches of size > 1, see Sec. 6.9.

6.6 Suggested Practical Procedure for Training Neural Networks

The general strategy for learning via stochastic gradient descent in deep neural networks is the same for any network architecture; as long as we can define a derivative for each computation. A rough outline of a suggested procedure to follow is as follows:

1. Design the network architecture (from inputs to outputs, e.g., Figure 5.2, including the chosen activation functions)

2. Draw the computational graph for the network (including the error computation), e.g., Fig. 6.3 corresponding to Fig. 5.2 and use it to obtain the gradient computation with respect to all quantities of interest (trainable parameters/weights)

3. Apply SGD; which, for each each iteration:

   (a) Forward-propagate inputs $\mathbf{x}$ all the way through to the error E

(b) Back-propagate track error signals $\boldsymbol{\delta}$ recursively all the way down to the input layer

(c) Obtain gradients $\mathbf{G}$ and do SGD updates 'as usual'

Hint: Be very careful about matrix dimensions at each step.

6.7 Generalising to Multiple Outputs

In a multi-class problem for $m > 2$ classes, outputs should be one-hot-encoded into an m-length vector (m labels); e.g., $\mathbf{y} = [0, 1, 0]$ representing the second of three possible classes ('class number 2' of 3). We should use a softmax function,

$$\sigma_k = P(Y = k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_k^\top \mathbf{x})}{\sum_{k'=1}^m \exp(\mathbf{w}_{k'}^\top \mathbf{x})}$$

which is simply a generalisation of sigmoid functions ($\sigma = \sigma_1 = 1 - \sigma_2$), placed over the output to scale the outputs accordingly to represent a categorical distribution (σ_ks should sum to unity).

If we allow for multiple labels per input (the multi-label problem), we can just use a sigmoid activation plus a threshold per-label; since, e.g., $\mathbf{y} = [1, 1, 0]$ is a valid classification (more than one class-label activated simultaneously, for a single input vector).

Such a network is shown in Fig. 6.2.

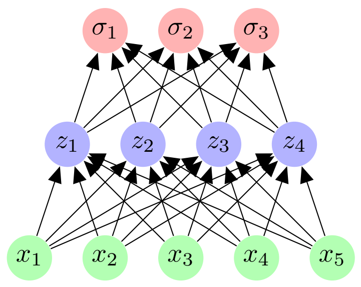


Figure 6.2: This network has three sigmoid outputs, it can be used for multi-label or multi-class classification, depending on the normalisation/thresholding on the outputs.

As we shall see, neither of these setups affects our back-propagation procedure, other than changing some dimensionality.In particular, note that (where $\mathbf{y}$ are the m-labels associated with some particular instance $\mathbf{x}$)

$$E(\mathbf{y}, \boldsymbol{\sigma}) = \| \underbrace{\mathbf{y} - \boldsymbol{\sigma}}_{\mathbf{e}} \|_2^2 = \mathbf{e}\mathbf{e}^\top \tag{6.2}$$

i.e., $\mathbf{e} \in \mathbb{R}^{1\times 3}$ (the residual errors for each output separately, respective of the input $\mathbf{x}$ and corresponding true target outputs $\mathbf{y}$), and $E \in \mathbb{R}_+$ (overall error associated with the network output for this input). This is because the labels are stored in vectors $\mathbf{y} \in \mathbb{R}^{1\times 3}$ and the outputs of the network in $\boldsymbol{\sigma} \in [0, 1]^{1\times 3}$.

6.8 Generalising to L-Layers

Let's do a worked example, on the network Fig. 6.2, with a further generalisation: $\{\mathbf{x}, \mathbf{z}, \boldsymbol{\sigma}\} \Leftrightarrow \{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2\}$, and $\{d, k, m\} \Leftrightarrow \{d_0, d_1, d_2\}$. This means that (for our forward pass) $\mathbf{x}_0 \in \mathbb{R}^{1\times d_0}$, $\mathbf{W}_1 \in \mathbb{R}^{d_0\times d_1}$, $\mathbf{W}_2 \in \mathbb{R}^{d_1\times d_2}$. For d_0 inputs, d_1 hidden units, d_2 outputs. The computational graph is shown in Fig. 6.3.

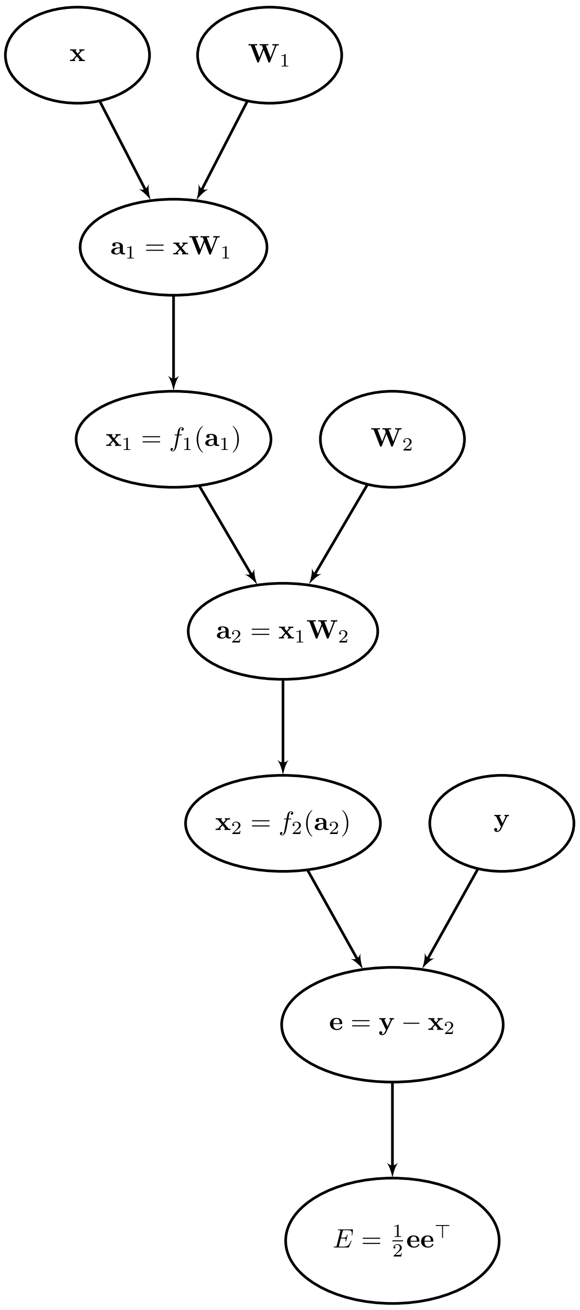

$\mathbf{x}$
$\mathbf{W}_1$
$\mathbf{a}_1 = \mathbf{x}\mathbf{W}_1$
$\mathbf{x}_1 = f_1(\mathbf{a}_1)$
$\mathbf{W}_2$
$\mathbf{a}_2 = \mathbf{x}_1\mathbf{W}_2$
$\mathbf{x}_2 = f_2(\mathbf{a}_2)$
$\mathbf{y}$
$\mathbf{e} = \mathbf{y} - \mathbf{x}_2$
$E = \frac{1}{2}\mathbf{e}\mathbf{e}^\top$

Figure 6.3: A computational graph for the 'standard' multi-layer perceptron of a single row of hidden units. We distinguish (different node for each of) the transfer function (producing $\mathbf{a}_\ell$) and the activation function (producing $\mathbf{x}_\ell$); a detailed reminder in Section 6.8.

The update for $\mathbf{W}_2$ can be read straight off the computational graph:

$$\begin{aligned}
\nabla_{\mathbf{W}_2} E &= \nabla_{\mathbf{e}} E \cdot \nabla_{\mathbf{x}_2}\mathbf{e} \cdot \nabla_{\mathbf{a}_2}\mathbf{x}_2 \cdot \nabla_{\mathbf{W}_2}\mathbf{a}_2 \\
&= \nabla_{\mathbf{e}}[\frac{1}{2}\mathbf{e}\mathbf{e}^\top] \cdot \nabla_{\mathbf{x}_2}[\mathbf{y} - \mathbf{x}_2] \cdot \nabla_{\mathbf{a}_2}[\sigma(\mathbf{a}_2)] \cdot \nabla_{\mathbf{W}_2}[\mathbf{x}_1\mathbf{W}_2] \\
&= \mathbf{e} \cdot -1 \odot f_2'(\mathbf{a}_2) \cdot \mathbf{x}_1^\top \\
&= -\mathbf{x}_1^\top \underbrace{(\mathbf{e} \odot f_2'(\mathbf{a}_2))}_{\boldsymbol{\delta}_2 \in \mathbb{R}^{1\times m}} \\
&= -\mathbf{x}_1^\top \boldsymbol{\delta}_2
\end{aligned}$$

And similarly for $\mathbf{W}_1$:

$$\begin{aligned}
\nabla_{\mathbf{W}_1} E &= \nabla_{\mathbf{e}} E \cdot \nabla_{\mathbf{x}_2}\mathbf{e} \cdot \nabla_{\mathbf{a}_2}\mathbf{x}_2 \cdot \nabla_{\mathbf{x}_1}\mathbf{a}_2 \cdot \nabla_{\mathbf{a}_1}\mathbf{x}_1 \cdot \nabla_{\mathbf{W}_1}\mathbf{a}_1 \\
&= \nabla_{\mathbf{e}}[\tfrac{1}{2}\mathbf{e}\mathbf{e}^\top] \cdot \nabla_{\mathbf{x}_2}[\mathbf{y} - \mathbf{x}_2] \cdot \nabla_{\mathbf{a}_2}[\sigma(\mathbf{a}_2)] \cdot \nabla_{\mathbf{x}_1}[\mathbf{x}_1\mathbf{W}_2] \cdot \nabla_{\mathbf{a}_1}[\mathbf{x}_1] \cdot \nabla_{\mathbf{W}_1}[\mathbf{x}_0\mathbf{W}_1] \\
&= \mathbf{e} \cdot -1 \odot f_2'(\mathbf{a}_2) \cdot \mathbf{W}_2^\top \odot f_1'(\mathbf{a}_1) \cdot \mathbf{x}_0^\top \quad \triangleright \text{ Recall here that } \mathbf{a}_1 = \mathbf{x}\mathbf{W}_1 \in \mathbb{R}^{k\times 1} \\
&= -\mathbf{x}_0^\top \underbrace{(\mathbf{e} \odot f_2'(\mathbf{a}_2) \cdot \mathbf{W}_2^\top \odot f_1'(\mathbf{a}_1))}_{\boldsymbol{\delta}_1 \in \mathbb{R}^{1\times d}} \\
&= -\mathbf{x}_1^\top \boldsymbol{\delta}_1
\end{aligned}$$

Now let ℓ be the ℓ-th layer of the network. A network goes from $\ell = 0, 1, \ldots, L$ where the 0th layer simply imputes the training sample as input, and the L-th layer provides the output:

$$(\mathbf{x}_0, \mathbf{x}_L) \leftarrow (\mathbf{x}_i, \mathbf{y}_i)$$

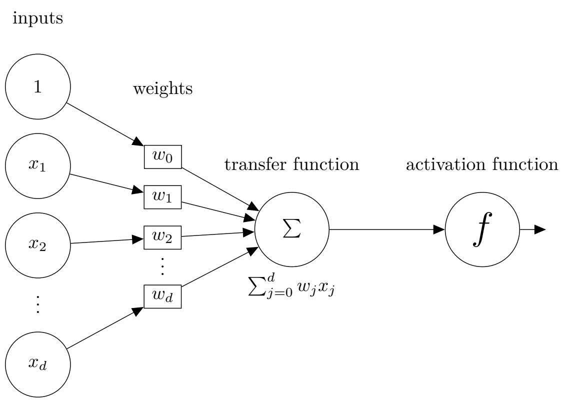
where $(\mathbf{x}_i, \mathbf{y}_i)$ a training pair from the training set (not part of the network definition).

In a dense network[2] (like the one exemplified above, where nodes in nodes are fully connected) we have weight matrices $\mathbf{W}_1, \ldots, \mathbf{W}_L$ and activation functions $f_1, \ldots, f_L$; we may now set $L > 1$ arbitrarily for a deep neural network.

The forward pass (including error calculation, vs true outputs $\mathbf{y}$) is as follows,

$$\boxed{\begin{aligned}\mathbf{x}_\ell &= f_\ell(\mathbf{x}_{\ell-1}\mathbf{W}_\ell) \\ E_L &= E(\mathbf{x}_L, \mathbf{y})\end{aligned}}$$

(we can also read this straight off the computational graph; Fig. 6.3) where, recalling a depiction of a neuron,



we may denote (as we have above, and in Fig. 6.3),

$$\begin{aligned}
\mathbf{a}_\ell &= \mathbf{x}_{\ell-1}\mathbf{W}_\ell \quad \triangleright \text{ Output of transfer function (the activation)} \\
\mathbf{x}_\ell &= f_\ell(\mathbf{a}_\ell) = \mathbf{f}_\ell \quad \triangleright \text{ Output of activation function}
\end{aligned}$$

(we distinguish between the output of the transfer function $\mathbf{a}_\ell = \mathbf{x}_{\ell-1}\mathbf{W}_\ell = [a_1, \ldots, a_{d_\ell}]$, and the output of the activation function $\mathbf{x}_\ell = f_\ell(\mathbf{a}_\ell) = [f_\ell(a_1), \ldots, f_\ell(a_{d_\ell})]$).

And backpropagation can be summarized as the following:

[2] We study non-dense layers in coming weeks

$$
\boxed{
\begin{aligned}
\boldsymbol{\delta}_L &= \mathbf{e} \odot f_L'(\mathbf{a}_L) \\
\boldsymbol{\delta}_\ell &= \boldsymbol{\delta}_{\ell+1} \mathbf{W}_{\ell+1}^\top \odot f_\ell'(\mathbf{a}_\ell) \\
\mathbf{G}_\ell = \nabla_{\mathbf{W}_\ell} E &= \mathbf{x}_{\ell-1}^\top \boldsymbol{\delta}_\ell
\end{aligned}
}
$$

(note that only the final layer L is different).

So we can see that the $\boldsymbol{\delta}_\ell$ vector is responsible for propagating the error information back through the network.

6.9 Generalising to Mini Batch Updates

In stochastic gradient descent, we do not obtain a gradient based on the entire data set at once because it can be too expensive to do so (for large n). Instead, we choose a random subset of instances; with the special case of a single instance $(\mathbf{x}_0, \mathbf{x}_L) \leftarrow (\mathbf{x}_i, \mathbf{y}_i)$ being dealt with already. Note that if we use mini-batch $\mathbf{X}_0$ and $\mathbf{Y}_L$ (containing n_b-instances, randomly selected from the data-set), noting in the algorithm/implementation needs to change! (supposing we are carefully considering the matrix dimensions; e.g., $\mathbf{X}_0 \in \mathbb{R}^{n_b \times d}$ to fit with the running example above).

Chapter 7

Convolutional Neural Networks (CNNs)

The family of deep learning architectures known as convolutional neural networks (CNNs) are one of the most important used today, particularly in the context of machine learning for images and sequences/signals (e.g., sensor measurements, audio, text, and medical signals and imagery). Like many deep architectures, they have existed already for several decades in some form (see in particular [9], inspired by [7]) and the concept of a convolution itself since at least a century ago, but have become widely used in recent years due to increases in available data and the computational resources to process that data (particularly GPUs), and associated software frameworks. CNNs are a powerful and versatile tool in the machine learning toolbox.

Main Concepts

To follow the material this week, you should already know

- How to carry out the forward pass in a neural network with dense/fully-connected layers (a MLP)
- How backpropagation works with gradient descent

This week your objective should be to learn:

- Why/when to use a CNN (rather than a MLP), including (but not only): images, text, signals
- How a discrete convolution works; you should be able to do a simple convolution 'by hand'
- The purpose of a pooling layer; average pool and max pool units, and how to back propagate through them
- What is a filter (kernel), and how is it updated via backpropagation
- What are the stride and padding hyper-parameters

In the lab-tutorial, you should learn how to implement and apply CNNs.

CNNs are a type of multi-layer neural network that share weights across inputs.

In this chapter we get into the itsy bitsy details of how CNNs work in order to gain a strong intuition about their functionality and their potential application to real-world problems. If you need a primer or longer reference introducing CNNs, a few references to consider: [5, 10, 2].

7.1 Convolution

Convolutional Neural Networks are so called due to their use of convolution[1] We are particularly interested in discrete convolutions (and to start with, in 1d), where signal (e.g., vector) $\mathbf{x}$ is convolved with filter $\mathbf{w}$ as follows:

$$\mathbf{z} = (\mathbf{x} * \mathbf{w})$$
$$z_t = \sum_{j=1}^{k} x_{t+j-1} w_j$$

i.e., $\mathbf{x}$ and $\mathbf{w}$ are convolved to produce $\mathbf{z}$; where the symbol $*$ is the convolution operator (*not* multiplication).

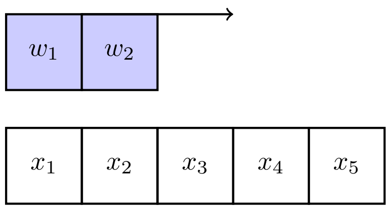


Figure 7.1: We can picture a 1d discrete convolution as vector $\mathbf{w}$ moving across vector $\mathbf{x}$, where we take the dot product of the overlap, at each step.

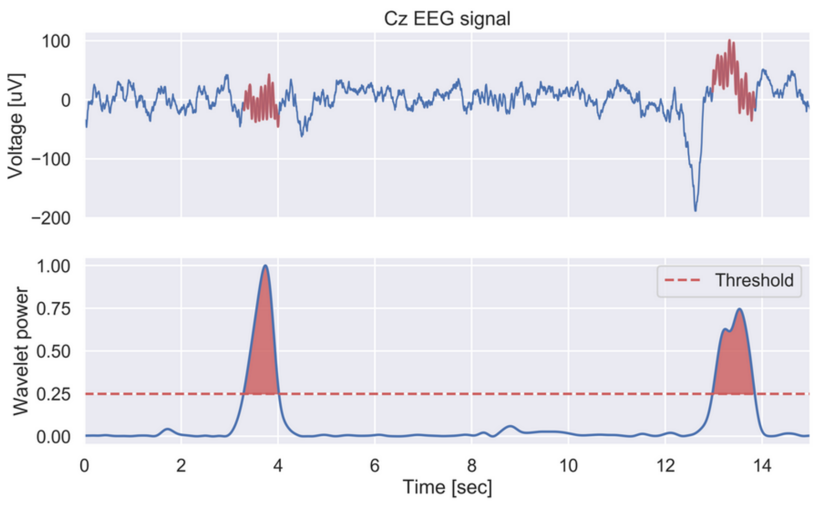


Figure 7.2: A specific wavelet (kernel/filter) is convolved with an EEG signal (top); such that the motifs known as spindles can be extracted using a threshold (bottom). Source: [11]. All that is needed is a pooling mechanism to collect the result, for example, to output the percent of the signal spent above the threshold.

[1]Technically we do a cross-correlation, not a convolution; a veteran of signal processing might point out the subtle difference: a convolution involves flipping the filter as a first step; but it becomes equivalent in our context of a CNN in machine learning. Just be aware that in NUMPY you would use `numpy.correlate` instead of `numpy.convolve` to reproduce the simple example of ‘convolutions’ given in the lecture

Fig. 7.2 illustrates the application of a convolution on a electroencephalographic (EEG) signal, used to detect particular patterns known as 'spindles'. But we can begin a detailed analysis on some simple toy examples. First, where signal $\mathbf{x} = [x_1, x_2]$ (1d signal of length 2) is convolved with filter (also called a kernel in the context of CNNs) $\mathbf{w} = [w]$, the result of the convolution is:

$$\mathbf{z} = [z_1, z_2] = \mathbf{x} * \mathbf{w} = [x_1 w, x_2 w] \tag{7.1}$$

So, if $\mathbf{x} = [2, 4]$ and $w = 0.5$, then $\mathbf{z} = [z_1, z_2] = [1, 2]$.

Note how the parameter/weight (the filter) $\mathbf{w}$ is shared among input $\mathbf{x}$ (even though $\mathbf{x}$ is *not* the same length as $\mathbf{w}$, unlike previous models we looked at).

Another example, where $\mathbf{x} = [1, 2, 3]$ and $\mathbf{w} = [0.4, 0.6]$:

$$\mathbf{z} = [z_1, z_2] = \mathbf{x} * \mathbf{w} = [x_1 w_1 + x_2 w_2, x_2 w_1 + x_3 w_2] \tag{7.2}$$

This result can be replicated in Python with `np.correlate(x,w,mode = 'valid'`[2]).

7.1.1 2d Convolutions

A discrete convolution can also be carried out in 2d (e.g., an image $\mathbf{X}$). A 2d convolution allows us to work with images without reshaping them; but the fundamentals are the same to a 1d convolution over a signal.

Again we 'slide' $\mathbf{W}$ over $\mathbf{X}$, and compute dot products on the overlap. We produce a new image $\mathbf{Z}$.

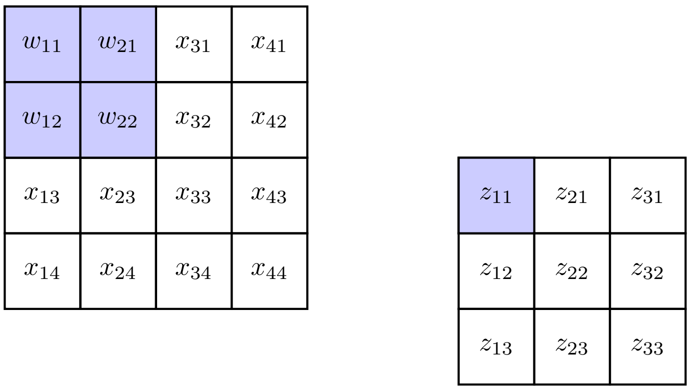

w_{11} w_{21} x_{31} x_{41}
w_{12} w_{22} x_{32} x_{42}
x_{13} x_{23} x_{33} x_{43}
x_{14} x_{24} x_{34} x_{44}
z_{11} z_{21} z_{31}
z_{12} z_{22} z_{32}
z_{13} z_{23} z_{33}

Figure 7.3: We can picture a 2d discrete convolution as matrix $\mathbf{W}$ moving across matrix $\mathbf{X}$, to produce $\mathbf{Z}$.

7.1.2 Padding and the Mode of a Convolution (`valid`, `full`, or `same`)

You can imagine a convolution (as we have studied) as running one fixed-length signal $\mathbf{w}$ towards and through another fixed-length signal $\mathbf{x}$, and taking a sum over products at each step of intersection. What do we consider the first step of intersection?

1. At first point of overlap between $\mathbf{x}$ and $\mathbf{w}$ (mode = `full`)

2. At first point where $\mathbf{x}$ and $\mathbf{w}$ are *fully* overlapping (mode = `valid`)

[2]This is a parameter of `numpy.convolve` and `numpy.correlate`; it is a non-negligible consideration in toy examples, but not so important in larger CNN architectures; see Section 7.1.2 for further elaboration

3. Such that the result $\mathbf{w} * \mathbf{x}$ is the same length as $\mathbf{x}$ (mode = `same`).

This does not really have too much effect on results in a large network.

7.1.3 Other Remarks

A convolution in time/space is equivalent to multiplication in the frequency domain (e.g., after a Fourier transform; after appropriate padding). From the signal processing point of view, an image can be seen as a 2D signal.

7.2 Pooling

Although called *convolutional*, CNNs employ pooling layers which are a crucial element. An alternative name for a CNN is a shift invariant artificial neural network, as shift/translation invariance is by far one of its most useful properties, and this is obtained using pooling. Backpropagation is carried out through both pooling and convolutional layers, but only the weights determining the convolution are updated.

You can think of pooling (also called subsampling) as a summary statistic of a convolution. Consider the average pool unit: an average is invariant to the position of inputs, e.g.,

$$\mathsf{avg}([1,2]) = \mathsf{avg}([2,1]) = 1.5$$

Another option is the max pool unit:

$$\mathsf{max}([1,2]) = \mathsf{max}([2,1]) = 2$$

Again we see observe translation invariance.

7.3 Activation Function

Like in other neural architectures, we include an activation function. In practice the activation function is be applied right after the convolution, then the pooling is applied (in theory, the activation could equivalently be applied after the pooling, depending on the activation/pool units). As for MLPs, the activation function is important for non-linearity; but it is not specific to CNNs – activation functions (and their derivatives, and back-propagating through them) have already been covered above (Chapter 6) with regard to dense layers – it's the only reason we don't make more of a fuss of them here.

7.4 Building a CNN / Architecture of a CNN

A CNN is typically composed of several convolutions and pooling operations in parallel, and several convolutions and pooling operations in series, followed by a flattening (when working with images) and concatenation, and dense layers, finishing very similarly to a standard MLP, producing output y. A simplified example is shown in Fig. 7.4.

7.5 Backpropagation and Training a CNN

In machine learning, we want to *learn* the weights (kernel) $\mathbf{W}$. To do so, again we turn to backpropagation.

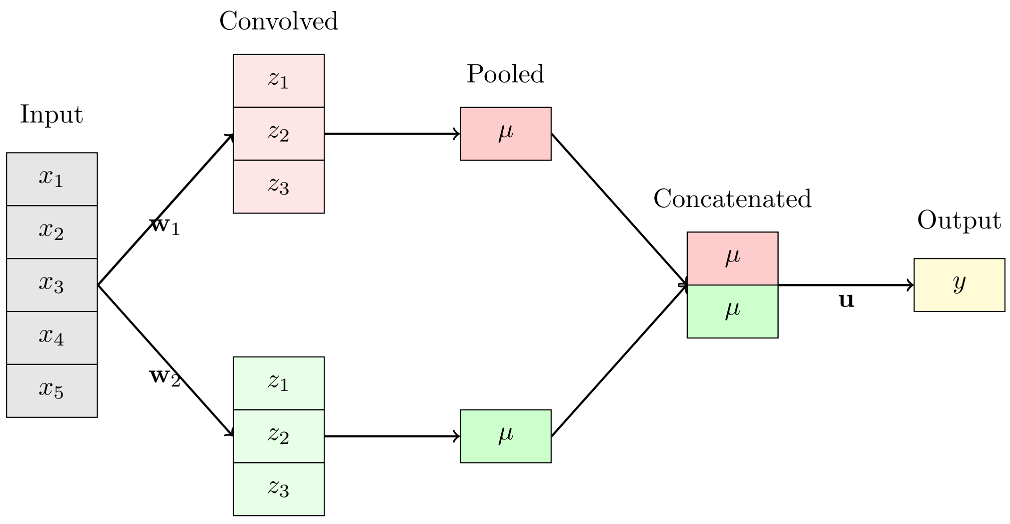
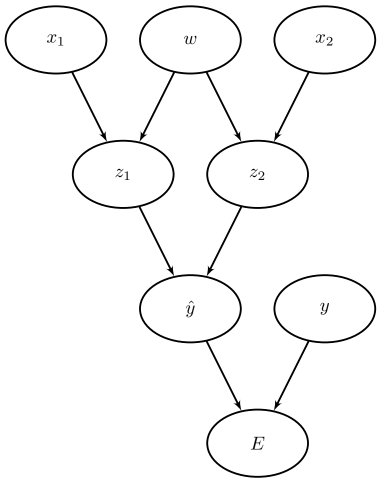


Figure 7.4: A minimalistic CNN operating on signal $\mathbf{x} \in \mathbb{R}^5$, producing classification $y \in \mathbb{R}$. Activations (behind z_j and y) not shown. There are two convolution operations involved (typically, there would be more; and also several layers of convolution and pooling, plus several dense layers following concatenation).

First, consider again the example from Eq. (7.1), for which the computational graph is drawn in Figure 7.5: a 1D filter $[w]$ on 2D input signal $[x_1, x_2]$. Reading off the computational graph, we see:

$$\begin{aligned}\frac{\partial E}{\partial w} &= \frac{\partial E}{\partial \widehat{y}}\frac{\partial \widehat{y}}{\partial z_1}\frac{\partial z_1}{\partial w} + \frac{\partial E}{\partial \widehat{y}}\frac{\partial \widehat{y}}{\partial z_2}\frac{\partial z_2}{\partial w} \\ \frac{\partial E}{\partial w} &= \frac{\partial E}{\partial \widehat{y}}(\frac{\partial \widehat{y}}{\partial z_1}\frac{\partial z_1}{\partial w} + \frac{\partial \widehat{y}}{\partial z_2}\frac{\partial z_2}{\partial w}) \\ &= E'\delta_1 x_1 + E'\delta_2 x_2\end{aligned} \tag{7.3}$$



$$\begin{tikzcd}
x_1 \arrow[dr] & w \arrow[d] \arrow[dr] & x_2 \arrow[d] \\
& z_1 \arrow[d] \quad z_2 \arrow[d] & \\
& \widehat{y} \arrow[dr] & y \arrow[d] \\
& & E
\end{tikzcd}$$

Figure 7.5: A computational graph for a simple convolution with 1D filter $[w]$ and 2D input signal $[x_1, x_2]$; the result $\mathbf{z}$ is pooled into $\widehat{y}$ which is used directly as a prediction, leading to error $E(\widehat{y}, y)$.

Now consider Eq. (7.2), for which the computational graph drawn in Figure 7.6: a 2D filter $\mathbf{w} = [w_1, w_2]$

on 3D input signal $[x_1, x_2, x_3]$.

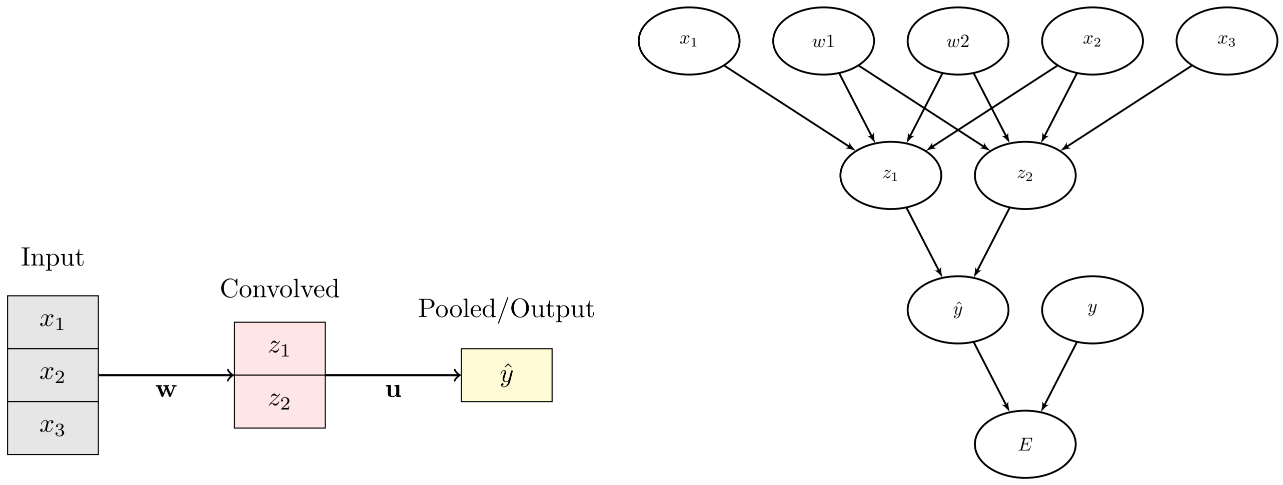


Figure 7.6: A depiction of the architecture (left) and associated computational graph (right) for a simple convolution with 2D filter $\mathbf{w} = [w_1, w_2]$ and 3D input signal $[x_1, x_2, x_3]$; the result $\mathbf{z}$ is pooled into $\widehat{y}$ which is used directly as a prediction, leading to error $E(\widehat{y}, y)$.

Since $\mathbf{w} = [w_1, w_2]$, there are two gradients to calculate. We follow the standard approach:

$$\begin{aligned}
\nabla_{\mathbf{w}} E(\mathbf{w}) &= \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}\right] && \triangleright \text{ Write out the error} \\
\frac{\partial E}{\partial w_1} &= \frac{\partial E}{\partial \widehat{y}} \left(\frac{\partial \widehat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1} + \frac{\partial \widehat{y}}{\partial z_2} \frac{\partial z_2}{\partial w_1} \right) && \triangleright \text{ Write out our query (wrt } w_1\text{)} \\
&= \frac{\partial E}{\partial \widehat{y}} \left(\frac{\partial \widehat{y}}{\partial z_1} \frac{\partial \mathbf{x}_{1:2}\mathbf{w}}{\partial w_1} + \frac{\partial \widehat{y}}{\partial z_2} \frac{\partial \mathbf{x}_{2:3}\mathbf{w}}{\partial w_1} \right) \\
&= \frac{\partial E}{\partial \widehat{y}} \left(\frac{\partial \widehat{y}}{\partial z_1} x_1 + \frac{\partial \widehat{y}}{\partial z_2} x_2 \right) \\
&= \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial z_1} x_1 + \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial z_2} x_2 \\
\frac{\partial E}{\partial w_2} &= \frac{\partial E}{\partial \widehat{y}} \left(\frac{\partial \widehat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_2} + \frac{\partial \widehat{y}}{\partial z_2} \frac{\partial z_2}{\partial w_2} \right) && \triangleright \text{ Write out our query (wrt } w_2\text{)} \\
&= \frac{\partial E}{\partial \widehat{y}} \left(\frac{\partial \widehat{y}}{\partial z_1} \frac{\partial \mathbf{x}_{1:2}\mathbf{w}}{\partial w_2} + \frac{\partial \widehat{y}}{\partial z_2} \frac{\partial \mathbf{x}_{2:3}\mathbf{w}}{\partial w_2} \right) \\
&= \frac{\partial E}{\partial \widehat{y}} \left(\frac{\partial \widehat{y}}{\partial z_1} x_2 + \frac{\partial \widehat{y}}{\partial z_2} x_3 \right) \\
&= \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial z_1} x_2 + \frac{\partial E}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial z_2} x_3 \\
\nabla_{\mathbf{w}} &= [\delta_1, \delta_2] * [x_1, x_2, x_3] = \boldsymbol{\delta} * \mathbf{x}
\end{aligned}$$

The forward pass is a convolution and the backward pass is also a convolution!

What about backpropagating through the pooling layer? In other words, what is $\frac{\partial \widehat{y}}{\partial z_1}$ and $\frac{\partial \widehat{y}}{\partial z_2}$ in Eq. (7.3) above? It depends on what pooling function we are using.

Sum pool Suppose a sum pool operation

$$\widehat{y} = z_1 + z_2$$

Then:

$$\frac{\partial \widehat{y}}{\partial z_j} = \frac{\partial (z_1 + z_2)}{\partial z_j} = (z_1 + z_2) = 1$$

(it's the same for $j = 1$ and $j = 2$).

Average pool Similarly, for the average pool operation,

$$\widehat{y} = \frac{1}{2}(z_1 + z_2)$$

for which the derivatives are:

$$\frac{\partial \widehat{y}}{\partial z_j} = \frac{\partial \frac{1}{2}(z_1 + z_2)}{\partial z_j} = 0.5 \tag{7.4}$$

(again, for $j = 1$ and $j = 2$; equivalently).

Max pool And for max-pool:

$$\widehat{y} = \mathtt{max}(z_1, z_2)$$

So if $z_1 = 1$ and $z_2 = 2$, then $\widehat{y} = z_2 = 2$. Let's do the derivatives:

$$\frac{\partial \widehat{y}}{\partial z_1} = 0 \quad \frac{\partial \widehat{y}}{\partial z_2} = 1$$

Intuitively: if we change z_2 by a little bit, y will change by the equivalent amount (i.e., 1 times the amount); but if we change z_1 by a little bit, there will be no (0) change to y. So, in back-propagation we pass δ back only along the branch which was maximal on the way up.

7.6 A Concrete Example

Let's get a bit more concrete: Assume squared error $E(w) = \frac{1}{2}(\widehat{y} - y)^2$, and *average pool* (Eq. (7.4)). Furthermore, filter $\mathbf{w} = [0.5]$, and signal $\mathbf{x} = [0.8, 0.4]$, with true label $y = 1$, all together as follows. The

Forward pass:

$$\begin{aligned} x_1 &\leftarrow 0.8 \\ x_2 &\leftarrow 0.4 \\ z_1 &= w \cdot 0.8 = 0.4 \\ z_2 &= w \cdot 0.4 = 0.2 \\ \hat{y} &= 0.5(z_1 + z_2) = 0.3 \\ E &= 0.5(\widehat{y} - y)^2 = 0.15 \end{aligned}$$

Backward pass:

$$\begin{aligned} g = \frac{\partial E}{\partial w} &= \frac{\partial E}{\partial \widehat{y}} \left(\frac{\partial \widehat{y}}{\partial z_1} \frac{\partial z_1}{\partial w} + \frac{\partial \widehat{y}}{\partial z_2} \frac{\partial z_2}{\partial w} \right) \\ &= (\hat{y} - y)(0.5 \cdot x_1 + 0.5 \cdot x_2) \\ &= (0.3 - 1) \cdot (0.5 \cdot 0.8 + 0.5 \cdot 0.4) \\ &= -0.42 \end{aligned}$$

Gradient descent update:

$$w \leftarrow w + \alpha 0.42$$

As earlier (in dense networks), here too, a weight w represents *influence* over a decision $\widehat{y}$ (and therefore also, the error produced by the network). And again, we are asking the question *how* does it influence the decision? And through *which nodes* does it gain influence? Under max pool one of the z-nodes (the non-maximal one) holds no responsibility for the decision, and therefore backpropagation should not pass through that node on the way back down the network. Under average pool the z-nodes have equal responsibility.

Chapter 8

Decision Trees (and Ensembles)

Learning with decision trees and ensemble methods are two separate concepts, but they combine well, and are often considered together; as we shall do here.

Even though decision trees are not closely related to other methods we've looked at so far, it is important to have understood already the concepts of overfitting and regularization (arguably, more than ever, with regard to trees) before proceeding.

Main Concepts

Some of the main points to understand regarding decision trees:

- Base concepts: entropy and information gain
- Algorithm to build (induce, from data; namely greedy recursive induction) a decision tree; and
- how to interpret such a decision tree
- The main advantages/disadvantages of decision tree models vs other models you have seen so far

And relating to ensemble methods, you should get:

- The main idea behind bagging and random forest ensembles (beyond the assumption that more trees = better accuracy), e.g., it should become clear to you
  - *why* decision trees in particular are so popular/effective in this context,
  - how ensemble methods can reduce variance of estimates.
- How is boosting different from bagging (e.g., regarding bias/variance trade-off, and more particularly with regard to the reduction of bias), and what is a 'weak learner'.

In decision tree learning, our model f takes the form of a decision tree. Making classifications (or regression) with a decision tree is usually intuitive, as it reflects human decision making. Our main question revolves around: how to learn (induce) such a tree. It requires some basics in information theory.

8.1 Information and Entropy

Consider random variable X; taking values $x \in \mathcal{X}$. In the context of information theory we can refer to x as a *message*.

The self information of a 'message' x,

$$\begin{aligned} I(x) &= -\log P(x) \qquad (8.1) \\ &= \log \frac{1}{P(x)} \end{aligned}$$

tells us a 'surprise value' of receiving message x (note: although it does *not* tell us about the *importance* of that message).

And entropy is simply the expected self information:

$$\begin{aligned} H(X) &= \mathbb{E}[I(X)] \qquad (8.2) \\ &= -\sum_{x \in \mathcal{X}} P(x) \log P(x) \qquad (8.3) \end{aligned}$$

which tells us the uncertainty around any given x. So, entropy is a measure of uncertainty. Note that entropy is a function of the random variable X, not a specific value x).

For simplicity, we will start with binary variables: X has a Bernoulli distribution parametrized by $\theta \in [0,1]$; a fair-coin has $\theta = 0.5$; and $\mathcal{X} = \{0,1\}$. See Fig. 8.1.

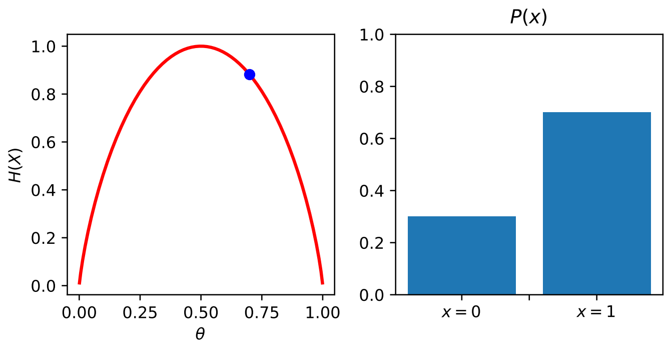


Figure 8.1: The relationship of a Bernoulli random variable X to its entropy $H(X)$; where $\theta = P(X=1)$, and $1-\theta = P(X=0)$ and (specific to this figure) $\theta = 0.7$.

Many sport and social events are at their 'most exciting' when the entropy of the result variable is maximal. For example, the recent trend in the U.S. of *gender reveal parties*[1].

[1] Where the gender of a baby is revealed to expectant parents, in a party context, in an over-exuberant and often dangerous fashion https://en.wikipedia.org/wiki/Gender_reveal_party

8.1.1 Conditional Entropy

We will also need conditional entropy, which expresses the uncertainty around Y given X:

$$
\begin{aligned}
H(Y \mid X) &= -\sum_{x\in\mathcal{X},y\in\mathcal{Y}} p(x,y)\log p(y \mid x) && \triangleright \text{ Entropy of } Y|X \quad (8.4)\\
&= -\sum_{x\in\mathcal{X},y\in\mathcal{Y}} p(y \mid x)p(x)\log p(y \mid x) && \triangleright \text{ Definition of joint probability}\\
&= -\sum_{x\in\mathcal{X}} p(x)\sum_{y\in\mathcal{Y}} p(y \mid x)\log p(y \mid x) && \triangleright\ p(x) \text{ is not dependent on } y\\
&= \sum_{x\in\mathcal{X}} p(x)\mathbb{E}_{Y\sim p(Y|x)}[-\log p(Y \mid x)] && \triangleright \text{ Expected self information} \quad (8.5)\\
&= \sum_{x\in\mathcal{X}} p(x)H(Y \mid x) && \triangleright \text{ cf. eqs. (8.2) and (8.3) (expected self information)} \quad (8.6)
\end{aligned}
$$

8.1.2 Information Gain

How much information do we gain by considering $P(Y|X)$ vs $P(Y)$? Another way to think about it: how much entropy do we reduce by splitting on X (entropy indicates uncertainty, so we want to *reduce* uncertainty)? We can define information gain as:

$$
\begin{aligned}
G(S,X) &= H(Y) - H(Y|X) && (8.7)\\
&= H(Y) - \sum_{x\in\mathcal{X}} p(x)H(Y|x) && (8.8)
\end{aligned}
$$

where $X \in \{X_1,\dots,X_d\}$ (it's one of the available features). But what if we do not have $P(Y)$, $P(Y|X)$, $P(X)$? What we do have is data, e.g., a set $S = \{y_1,\dots,y_n\}$ (where each $y_i \in \{0,1\}$ a label), or a subset $S_x \subset S$ (labels y_i where $x_i = x$ which are instances x_i matching on feature x). Empirically, $P(Y = 1) \approx \frac{1}{n}\sum_{i=1}^{n} y_i$. More generally, $P(Y = 1) := \frac{1}{|S|}\sum_{y_i\in S} y_i$ (allows for S to be subset of the full dataset). Therefore, we can define information gain on empirical sets S as:

$$
G(S,X) = H(S) - \sum_{x\in\mathcal{X}} \frac{|S_x|}{|S|}\cdot H(S_x) \qquad (8.9)
$$

where S_x the set of y_i where corresponding $x_i = x$.

8.2 A Worked Example

Again: information gain $G(S,X_j)$ measures the change in entropy from *not* splitting ($H(S)$) vs splitting ($H(S|X_j)$) on attribute X_j. We want to maximize this change wrt X_j (i.e., by choosing the right feature to split on.

To begin building a decision tree, we need to split branches from the root. A good split is one where the purity of labels is maximised at the leaves, which means that uncertainty is minimised, which means that entropy is minimised. A leaf containing three examples whose labels are $\{1,1,1\}$, is 100% pure; and if we are to understand that this represents $P(Y = 1 \mid \mathbf{x}) = 1$, then entropy is 0. Thus, in the absence of knowledge of P, but having a set, we treat the set as an empirical distribution, and calculate entropy on the set.

Let's take a toy example: Fig. 8.2. Imagine we want to build a decision tree to decide to address someone as vous ($y = 1$) or tu ($y = 0$) in French, given the available attributes $\mathbf{x}$.

Index i	Age X_1	Friend X_2	Gender X_3	Form Y
1	Adult	No	M	vous
2	Adult	No	F	vous
3	Adult	Yes	M	tu
4	Adult	Yes	F	tu
5	Child	No	M	tu
	Child	No	F	?

Figure 8.2: A toy example, showing a very small dataset on a highly simplified framing of the question of the formal vs informal 'you' in French; alongside the question (regarding the induction of a decision tree): which attribute to split on in order to construct an effective and efficient decision tree, providing accurate decision y, based on attributes $\mathbf{x}$?

To start building a decision tree we should split on some attribute. Which attribute X_j shall we split on? Let's try Gender (let vous $= 1$, and $S = \{1, 1, 0, 0, 0\}$ as per Fig. 8.2):

$$
\begin{aligned}
H(S) &= H(\{1,1,0,0,0\}) = 0.97 && \triangleright \text{ No split} \\
H(S \mid X = \texttt{M}) &= H(\{1,0,0\}) = 0.92 && \triangleright \text{ Left branch} \\
H(S \mid X = \texttt{F}) &= H(\{1,0\}) = 1 && \triangleright \text{ Right branch} \\
H(S \mid X_3) &= \sum_{x \in \{\texttt{M},\texttt{F}\}} P(X = x) \cdot H(S_x) && \triangleright \text{ Split on } X_3\text{; cf. Eq. (8.6)} \\
&= 0.95 \\
G(S, X_3) &= H(S) - H(S|X_3) && \triangleright \text{ Information Gain} \\
&= 0.02
\end{aligned}
$$

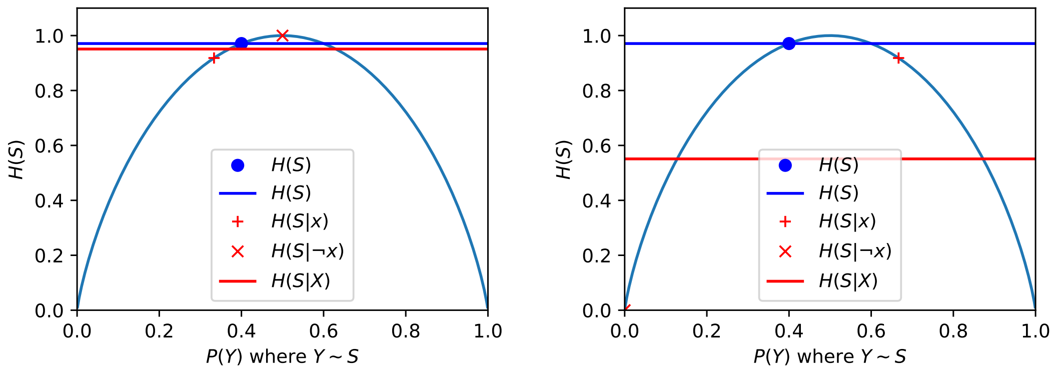


Figure 8.3: Splitting on Gender (left) vs Friend (right) attribute.

That's not very much gain! Let's try splitting on Friend:

$$\begin{aligned} H(S \mid \texttt{Yes}) &= H(\{0,0\}) = 0 \\ H(S \mid \texttt{No}) &= H(\{1,1,0\}) = 0.92 \\ H(S \mid X_3) &= 0.55 \\ G(S, X_2) &= 0.42 \end{aligned}$$

A much better choice ($0.42 > 0.02$)! Recall: We want to maximize purity/lower uncertainty at the leaves (push entropy towards 0). In fact this is the best possible split. Fig. 8.3 illustrates the result in information gain. The numbers correspond to Figure 8.4, which is the Scikit-Learn decision tree. The remaining branches in Fig. 8.4 are following the same process recursively, with appropriate stopping criteria.

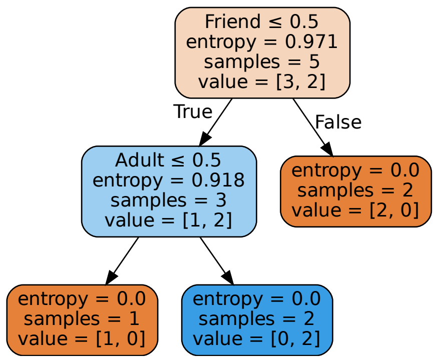


Figure 8.4: Scikit-Learn's depiction of the full decision tree (default parameters); where value = [1, 0] indicates that there is one sample of the 0-th indexed class (tu), and zero for the other. Orange represents the concentration of $Y = 0$ ('tu'), and blue represents the concentration of $Y = 1$ ('vous').

8.2.1 Information Gain with feature variables taking many values

You can see that in Fig. 8.4 there would not be any change even if we introduced categorical attributes of arbitrary cardinality $|\mathcal{X}|$, e.g., $x \in \{1, 2, 3, 4\}$. We still induce a binary-splitting decision tree, but there are more are more possible splits to decide from, e.g., $x \leq 1.5$, $x \leq 2.5$ and $x \leq 3.5$, for attribute X (in this example).

8.2.2 Information Gain with continuous feature variables

If $X \in \mathbb{R}$ (rather than $X \in \{0, 1\}$, as treated above), there could be infinite possible values, but with a finite data set $\{x_1, \ldots, x_n\}$ there are only finite split points (namely, $n - 1$). Therefore this is the same case as in Section 8.2.1, considering splits $X < x'$ (left branch), $X \geq x'$ (right branch), for split points x' in between data points.

In many cases, the use of continuous variables in decision trees may lead to less interpretability, and it might be advised (if interpretability is indeed the main focus) to discretise features first. For example, with regard to our toy data (Table 8.2), even though more information is available it could be less useful to consider a continuous 'Age' attribute and end up with a split of something like Age < 18.208 years; as compared to '= Child'.

8.3 Decision Tree Regression

What about when the *target* variable is continuous, $Y \in \mathbb{R}$? With continuous labels, $Y \in \mathbb{R}$ (i.e., decision trees for regression) we can use squared error instead of entropy to measure impurity; and continue with the above methodology. You would notice that the shape of squared error (i.e., parabola) is 'similar' to the logarithmic shapes shown for entropy (when viewed upside down), and is conceptually similar: error is a measure of variance, and both are maximal in Bernoulli variables if $P(Y = 1) = 0.5$. Consider Fig. 8.5.

But why use squared error instead of just entropy again? The formula for entropy requires us to evaluate pdf $p(y)$. How to evaluate this over a set, when $y \in \mathbb{R}$?

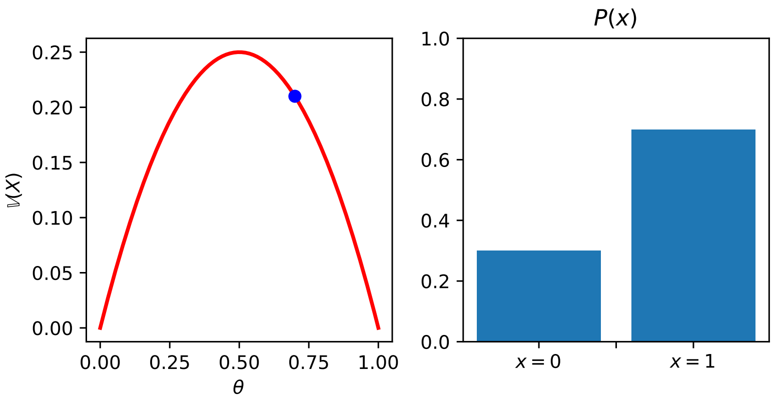


Figure 8.5: The relationship of a Bernoulli random variable X to its variance $\mathbb{V}(X)$; where $\theta = P(X = 1) = \mathbb{E}[X]$, and $1 - \theta = P(X = 0)$. This helps understand why squared error is a good option when $X \in \mathbb{R}$.

8.4 Gini Impurity

An alternative measure of impurity to entropy used for classification is Gini impurity (you'll see it offered as an option (i.e., hyper-parameter) in SCIKIT-LEARN):

$$g(Y) = \sum_{y \in \{0,1\}} P(Y = y)(1 - P(Y = y)) = 1 - \sum_{y \in \{0,1\}} P(Y = y)^2$$

There is a connection to *Gini coefficient* which is commonly used to measure income inequality (if the wealth of the world is *distributed* evenly across the population; vs if the [probability] mass of wealth lies with a single individual; the two extremes).

See Fig. 8.6; and note again the main point here: uncertainty is maximal and minimal in the same places (as entropy, and squared error).

8.5 Advanced Connections

At the beginning [of this chapter] we said decision trees are not closely related to other methods we've looked at so far. This is only superficially true. Indeed, the concepts behind decision tree induction (e.g., *entropy*), are very much an integral part of machine learning. Indeed, logistic regression is/was known as the 'maximum entropy method' in natural language processing. When we take Eq. (**??**) but consider two

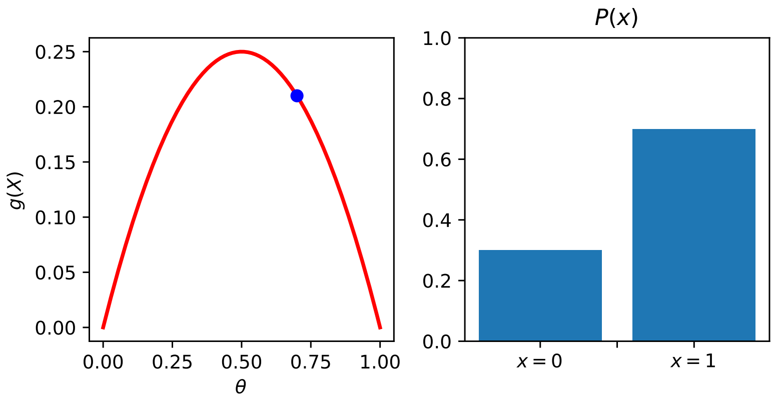


Figure 8.6: The relationship of a Bernoulli random variable X to its Gini impurity $g(X)$; where $\theta = P(X = 1)$, $1 - \theta = P(X = 0)$. Compare to Fig. 8.5 and Fig. 8.1.

different distributions, we arrive at the same *cross entropy* that we use as a loss function for in neural networks – as we essentially want to minimize the difference between our distribution $p(y \mid \mathbf{x})$ and the distribution producing the true label.

8.6 Knowing When to Stop

Important aspects of growing/inducing a decision tree are

1. knowing when to stop, and – a related item,
2. interpreting the result

(overly deep trees will both overfit *and* be difficult to interpret).

Decision trees, on their own, are not known for consistently topping the charts in predictive performance (although they can!), but when accuracy is a priority; we would usually consider *ensembles* of decision trees – as we do, next.

8.7 Ensemble Methods

Using an ensemble of models is a general approach that can be used with any class of model. But decision trees are often associated with this approach used in ensembles. A first objective is to understand why (and why not, ensembles of ridge regression, for example?).

An ensemble is simply using many instantiations of one type of method, which together make a *single* decision per instance $\mathbf{x}$. But beware, that the intuition of 'more is better' is only generalisable under certain conditions; namely when individual models are different from each other (their predictions should not be expected to always be the same, even for the same instance $\mathbf{x}$). Therefore most of the emphasis in ensembles is on achieving diversity among the models. Decision trees are thus a great candidate for ensembles because decision trees are unstable learners, meaning that they can vary significantly with only minor changes to the training data; diversity among trees is easy to obtain.

8.7.1 Bagging

Trees are unstable, but decision-tree induction is still a deterministic algorithm, and will render the same tree from the same training data set – so how to obtain diversity from a single training set? That's where bagging comes in.

In bootstrap aggregating (bagging) we sample from the original data set $\mathcal{D}_m \sim \mathcal{D}$[2] with replacement. That means, we select a random instance from the data set $\mathcal{D}$ and place it into $\mathcal{D}_m$, and then repeating that process *without ever removing* the instance from $\mathcal{D}$. An immediate implication is that we can select the same instance more than once (meaning that $\mathcal{D}_m$ may contain duplicate instances). It also means that:

- samples are independent of each other (an important assumption; recall Section 2.7);
- $\mathcal{D}_m$ can contain the same number of samples as $\mathcal{D}$ (or fewer, or more); and
- $\approx 63.2\%$ of samples in $\mathcal{D}_m$ will be unique, with others being duplicates.

8.7.2 Random Forest

There are other ways to increase diversity. For example, the method of random forest considers a random subset of features when making each split. This approach can be used in combination with bagging.

Taking this to the extreme we might consider a random split point; an approach known as using extra-randomized trees.

[2]Elsewhere the notation $\sim$ means *distributed according to*; here it means *sampled* or *drawn from* in the empirical sense (indeed, both concepts are related; we may write $\mathcal{D}_m \sim \mathcal{D} \sim P$)

Chapter 9

Unsupervised Learning I: Feature Extraction (Representation Learning)

The paradigm of unsupervised learning is inherently more subjective than supervised learning, but it is of great and growing importance in machine learning. It is much easier to gather data for unsupervised learning because we do not need to have labels attached to each instance. And, as such, it is closer to the way that humans learn.

Case in point: most modern large language models are trained mainly in an unsupervised nature (technically, *self-supervised*) in the sense that most learning is done without access to class labels. The applications involving labels, in most cases, only involve fine tuning for a specific task (such as classification).

In this setting we only have training instances $\{\mathbf{x}_i\}_{i=1}^n$, *without* output labels. Nevertheless, our task is still to produce a mapping to outputs! To avoid confusion, we will denote these outputs as $\mathbf{z}$, to make explicit that this is an unsupervised task (i.e., y_i is reserved for the *training* labels). Hence, we seek: $f : \mathbf{x} \mapsto \mathbf{z}$ where $\mathbf{z} \in \mathbb{R}^k$, that we have created (extracted) k features, to use as a representation. This is the task of representation learning or feature extraction or dimensionality reduction – different names, depending on the precise purpose for which we build this mapping.

Main Concepts

Some of the main points for this week:

- Feature selection vs feature extraction: What is the difference?
- The main types of feature selection: filter method, wrapper method, embedded method (and the advantages/disadvantages of each).
- Principal Components Analysis (PCA): how does it work and how to interpret results (what is a *principal component*)?
- Autoencoders: what are they, when to use them (vs PCA); and how can we regularize them what is the connection to PCA?
- A rough idea of related methods for unsupervised auto-encoding, and of advanced architectures and their applications.

9.1 Introduction to Principal Components Analysis (PCA)

Suppose: we are provided a data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$. Our goal is to associate each i-th instance $\mathbf{x}_i = [x_1, \ldots, x_d]$ with a vector of k *new* features (where $k < d$); i.e., producing data matrix. Unlike feature selection, we consider that the columns of $\mathbf{Z} \in \mathbb{R}^{n \times k}$ refer to potentially *different* features (than those of $\mathbf{x}_i$); i.e., we have 'extracted' features. For a given k, how can we decide upon the *best* k features to produce? Principal components analysis (PCA) provides an answer.

For convenience of notation and derivation (in this introduction): suppose that our data has already been centered before we operate on it, i.e., that $\mathbf{X}$ has a mean of $\mathbf{0}$. Furthermore, let's focus on *the* [single] best feature z (that is, let $k = 1$). Which one is it? Figure 9.1 shows PCA's answer to the problem on a toy dataset.

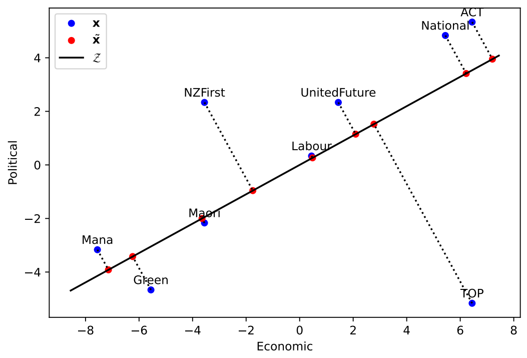


Figure 9.1: The points (source: `https://www.politicalcompass.org/nz2017`) indicate a possible placement of New Zealand's political parties in 2017, on two axes: economic-conservativeness (horizontal) and political-authoritarian (vertical) conservativeness. PCA has projected the points $\mathbf{x}_i \in \mathbb{R}^2$ to subspace $\mathcal{Z} = \mathbb{R}$ (which is a line, in this case, since $k = 1$). It has 'discovered' the traditional left-wing/right-wing political spectrum. We can use Z as our new feature (to *replace* the two original features). We can see $\tilde{\mathbf{x}}_i \in \mathbb{R}^2$ as the reconstruction of $\mathbf{x}_i$, and so the difference between these two as a reconstruction error.

What is PCA? PCA is a linear function, and can be seen as a linear orthogonal projection of points $\mathbf{x}_i$ onto corresponding points z_i which lie on the principal component:

$$z_i = f(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w} \tag{9.1}$$

Does $f(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w}$ look familiar? Hint: ordinary least squares. But how to do the learning (find $\mathbf{w}$?) in this case? As with machine learning methods we have looked at so far, we need to consider a loss metric/error function, $E(\mathbf{w})$. But we can't compare z_i to our true label y_i because we *don't have* such a label. In Sections 9.2 and 9.3 we introduce two equivalent loss metrics.

9.2 PCA as maximizing variance

We want component $\mathbf{w}$ such that the variance among z_i-points is maximal. Intuition: we want points to be as spread out as possible in $\mathcal{Z}$-space; the more spread out they are, the easier they are to distinguish from each other. Hence our error function (includes the definition of variance, on $\mathbf{0}$-centered data):

$$E(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n} z_i^2 = -\frac{1}{n}\sum_{i=1}^{n} (\mathbf{x}_i \mathbf{w})^\top (\mathbf{x}_i \mathbf{w}) \tag{9.2}$$

(we minimize negative variance).

The solution to Eq. (9.2): take the eigenvector corresponding to the largest eigenvalue; this becomes $\mathbf{w}$.

9.3 PCA as minimizing reconstruction error

We want component $\mathbf{w}$ such that $z_i = \mathbf{x}_i \mathbf{w}$ back into $\mathcal{X}$-space as a reconstruction $\tilde{\mathbf{x}}_i \approx \mathbf{x}_i$, then the squared difference (reconstruction error) between these two is minimal. These differences are shown as dashed lines in Figure 9.1. Realising that each reconstruction is formed via $\tilde{\mathbf{x}}_i = \mathbf{x}_i \mathbf{w}\mathbf{w}^\top$, our error function is:

$$E(\mathbf{w}) = \sum_{i=1}^{n} \|\mathbf{x}_i - \tilde{\mathbf{x}}_i\|_2^2 = \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{x}_i \mathbf{w}\mathbf{w}^\top\|_2^2 \tag{9.3}$$

The solution to Eq. (9.3): take the eigenvector corresponding to the largest eigenvalue; this becomes $\mathbf{w}$.

9.4 On the Connection Between PCA and Eigenvectors

So you knew already what an eigenvector and eigenvalue is, but – how does this connect to PCA?

Suppose we have data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$. Our sample-covariance matrix is (recall: given centered data)

$$\mathbf{S} = \frac{1}{n-1}\mathbf{X}^\top \mathbf{X} = \begin{bmatrix} \sigma_1^2 & \sigma_1\sigma_2 \\ \sigma_2\sigma_1 & \sigma_2^2 \end{bmatrix} \tag{9.4}$$

What does our data look like? Figure 9.2 shows an example. The length and width of the ellipse correspond to the eigenvalues of matrix $\mathbf{S}$, corresponding to data $\mathbf{X}$. The eigenvectors of $\mathbf{S}$ provide the directionality of the data.

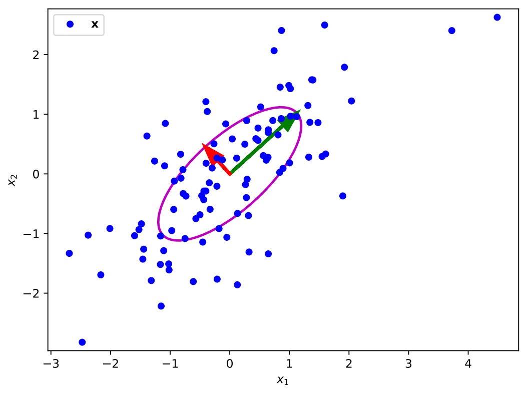


Figure 9.2: Eigenvectors tell us the direction of the data, (rotation), and the eigenvalues tell us the magnitude of those vectors (scale).

The covariance matrix $\mathbf{S}$ contains information about how variables X_1 and X_2 interact; how they vary together, and thus – how the data is spread in space. If X_1 varies strongly with X_2 (e.g., value x_1 is large when x_2 is large) then then $\sigma_1\sigma_2$ will be large. If X_1 varies a lot (on its own), then σ_1^2 will be large, and so on. The ellipse is basically a contour line around $\mathcal{N}(\mathbf{0}, \mathbf{S})$.

There are some interesting properties of co-variance matrices. We can treat it as a linear transformation, transforming vectors $\mathbf{x}$, i.e., $\mathbf{S}\mathbf{x}$. So what does transformation $\mathbf{S}$ do to this vector $\mathbf{x}$? It can *scale and rotate* it. In Figure 9.3, we see the data (left) has been scaled (mid) and then rotated (right). Eigen-decomposition separates exactly these two operations:

$$\mathbf{S} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1} \tag{9.5}$$

where diagonal matrix $\mathbf{\Lambda}$ of eigenvalues (scaling) and a stack of eigenvectors $\mathbf{U}$ (rotation);

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{bmatrix} \quad \text{and} \quad \mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_d]$$

($\mathbf{u}_l$ is the l-th eigenvector, associated to the l-th eigenvalue λ_l).

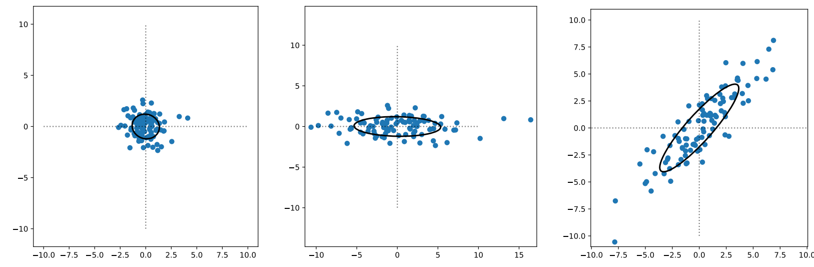


Figure 9.3: The left dataset is a scaled and rotated version of the right. Equivalently, we can say: the left one is a whitened version of the right. .

And what is the connection to PCA: We want to choose a component (or several, when $d > k > 1$) Z and project to it. We want the spread (variance) of the data to be maximized across this component. Looking at the image in Figure 9.3 (on the right), it should be clear that this corresponds to a line passing through the length of the ellipse. This is exactly the line pointing along the direction of $\mathbf{u}_1$ corresponding to largest eigenvalue (if we assume they are sorted by size, that is λ_1). Having chosen a direction, we no longer care about $\mathbf{\Lambda}$; eigenvector $\mathbf{u}_1$ provides what we need to proceed with an orthogonal projection onto the component.

Let's call this vector $\mathbf{u}_1 \equiv \mathbf{w}$ and get rid of the subscript for clarity (recall: we're first considering the simple case of only looking for a single best component; $k = 1$). We just want to project data points $\mathbf{x}$ into that line (it would be a plane when $k = 2$ and in general we can talk of a subspace) along which this vector points. The name of that line/subspace is $\mathcal{Z}$. We can use the formula for an orthogonal projection:

$$\begin{aligned} \mathsf{proj}_{\mathcal{Z}}(\mathbf{x}) &= \frac{\mathbf{x}\mathbf{w}}{\mathbf{w}^\top\mathbf{w}}\mathbf{w}^\top && \triangleright \text{ Project } \mathbf{x} \text{ orthogonally onto subspace } \mathcal{Z} && (9.6) \\ &= (\mathbf{x}\mathbf{w})\mathbf{w}^\top && \triangleright \text{ Let } \mathbf{w}^\top\mathbf{w} = 1\text{; see Section 9.5} && (9.7) \\ &= \mathbf{x}\mathbf{w}\mathbf{w}^\top && && (9.8) \end{aligned}$$

These orthogonal projections are clearly depicted in Fig. 9.1. And that is PCA!

Still not clear? The following special cases might help understand:

1. Suppose that there is *no rotation*, only scaling (i.e., $\sigma_1\sigma_2 = \sigma_2\sigma_1 = 0$, as in Figure 9.3 (middle)) – then we are already aligned with the standard axes, and just consider which of these axes to project to; it is the same as feature selection (selecting one of the two features). No need to do eigen-decomposition here (because there is no *composition to start with*; try eigen-decomposing $\mathbf{S}$ anyway in this case – what do you see?). This corresponds to feature selection.

2. Consider the extreme case of *perfectly correlated data*, where feature $X_1 = X_2$. This is actually an ideal candidate for application of PCA, we can reduce from $d = 2$ to $k = 1$ features without any loss of information. Try this on some toy data, and have a look at λ_1 vs λ_2; what do you notice?

A final question to check your understanding: Can you see why we center the data before we begin? Think about what happens if we don't do this step.

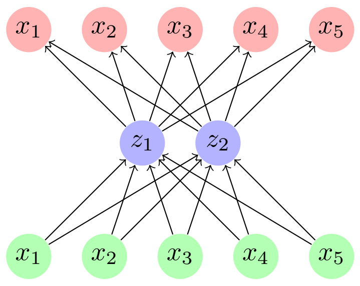
9.5 The Constraint

Whether we maximize variance or minimize reconstruction error, we must impose a constraint that $\mathbf{w}^\top \mathbf{w} = 1$. We see this to get from Eq. (9.6) to Eq. (9.7). Why? Think about doing this optimization without the constraint. Essentially, without such a constraint, the projection is no longer orthogonal and it's possible to 'cheat' (project out towards infinity) in order to get great variance in $\mathcal{Z}$-space.

9.6 Autoencoders

If PCA is the unsupervised analogy of ordinary least squares; then auto-encoding neural networks (Autoencoders) are the unsupervised version of neural networks. One might argue that they are in fact *self-supervised*[1].

Auto-encoders are simply neural networks, trained to predict their own inputs. Here is a visual impression: $d = 5$ features and $k = 2$ hidden layer units:



Auto-encoders provide one important advantage over PCA. It's the same advantage that MLPs provide over OLS: non-linearity.

Auto-encoders are a big deal, but this is a relatively short section because autoencoders are, indeed, essentially just 'regular' neural networks (which we have covered in quite some depth already), and they are trained in the same ways. So the only difference is their selection of output (being equal to the input). Like all neural networks (but arguable, especially in this case), we need to pay special attention to regularization (you can imagine the possibility to overfit $\mathbf{x} = f(\mathbf{x})$)!

Autoencoders are an integral part of the deep learning ecosystem. Like most of the other animals in this ecosystem of methods, they have been around in some form or other for decades, but have found relatively recent popularity (and success), and many different varieties have been spawned and evolved in the scientific literature in recent years.

9.7 Advanced Representations

It is only a small step from auto-encoders to word-embeddings, which have been a fundamental step forward in language models.

[1] *Eigen-*, *Auto-* and *Self-* are equivalent in German, Latin, and English, respectively

Even without considering recurrent neural networks or transformers (very interesting models, but we haven't covered them yet) you should have some ideas (using methods covered so far in the course) of how you might use of embeddings to train a model to predict the next word in a sentence, or the next pixel of an image, i.e., generative modelling; generating text or images.

Chapter 10

Unsupervised Learning II: Clustering

Clustering is the unsupervised analogue of classification: we wish to provide (learn) function that maps a given instance to a cluster label (group). We might also be interested in a 'soft' label, or *relevance* of the instance to any given cluster.

Formally, the problem can be set exactly like classification, except, we will use label z (rather than y) to specifically denote that this is not a training label; $z \in \{1, 2, \ldots, k\}$ and z_i to be assigned to the i-th instance (or $\mathbf{z}_i$ such that $z_{ij} = 1$ indicates instance i is to be assigned to the j-th cluster). We desire to learn function $f : \mathbf{x} \mapsto z$ and, similarly as in Logistic Regression, we might also be interested in $P(Z = z \mid \mathbf{X} = \mathbf{x})$, the relevance of $\mathbf{x}$ to a given label z, which can be interpreted as a probability. Since no z_i-labels are provided in the training data it is necessary to decide beforehand how many clusters k we will consider. As always in machine learning, there should be some explicit or implicit loss metric that allows us to define a good clustering function f.

Main Concepts

1. You should aim to understand k-means clustering; in particular have a good idea of
   - how to implement it,
   - how to choose a reasonable k,
   - applications where k-means is useful; and importantly,
   - when *not* to use k-means (what are its limitations)
2. You should be aware of Gaussian mixture models as a probabilistic solution to clustering (and how it relates to and overcomes disadvantages of k-means).
3. And you should understand how spectral clustering works (such that you could implement it), and in particular how it can overcome a main limitation of k-means clustering.

As always, be aware than the material in these lecture notes is a complement to, not a version of, the material in the lecture.

10.1 k-Means Clustering

The method of k-means clustering works by iteratively assigning each instance to one of k clusters based on the distance between the instance and the mean instance (we will also call a centroid) of each cluster.

Having chosen k, we aim to minimise the intra-cluster distance, while maximising the inter-cluster distance. This can be expressed formally (given n observations $\mathbf{x}_1, \ldots, \mathbf{x}_n$) as identifying k means $(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k)$ and n cluster assignments $(\mathbf{z}_1, \ldots, \mathbf{z}_n)$, according to the minimization of

$$E(\mathbf{z}_1, \ldots \mathbf{z}_n, \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k) = \sum_{i=1}^{n} \sum_{j=1}^{k} z_{ij} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2 \tag{10.1}$$

where (recall) $z_{ij} = 1$ indicates that instance $\mathbf{x}_i$ belongs to the j-th cluster (and $z_{ij} = 0$ that it does not; similarly to the multi-class notation mentioned in Section 6.7[1]).

What's tricky here (respective of minimisation of error functions for classification) is that there are two sets of unknown parameters: cluster labels $(\mathbf{z}_i)$, *and* the cluster means (centroids, $\boldsymbol{\mu}_j$). If we had one, we could compute the other (and in closed form), and vice versa. But *how to proceed* if we have neither?

To break this chicken-and-egg problem, we set one set of parameters randomly (cluster means $\boldsymbol{\mu}_j$), then solve for the other (cluster labels z_{ij}), and then proceed iteratively. So, we iterate between two steps corresponding to the optimisations for $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$ and $\{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ respectively, keeping one fixed as we optimise the other. Once iterated until convergence we can expect be at a local optimum.

For the random centroids, we can start by choosing k-points randomly from our dataset – and using these as the initial centroids. Algorithm 10.1 sketches out the algorithm for minimising Eq. (10.1) which implies minimising intra-cluster distance and maximising inter-cluster distance.

Algorithm 10.1.1- k-means algorithm

First, select randomly k centroids, then (iteratively, until convergence):

1. Assign each $\mathbf{x}_i$ (for $i = 1, \ldots, n$) to the closest cluster, i.e., setting $z_{ij} = 1$ for

$$j = \operatorname*{argmin}_{j' \in \{1,\ldots,k\}} \sum_{j'=1}^{k} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2$$

2. Compute the centroids $\boldsymbol{\mu}_j$ (for $j = 1, \ldots, k$) to maximize the fit:

$$\boldsymbol{\mu}_j \leftarrow \frac{\sum_{i=1}^{n} z_{ij} \mathbf{x}_i}{\sum_{i=1}^{n} z_{ij}}$$

[1]Just as for classification, it can be more intuitive to use notation such as $z_i \in \{1, \ldots, k\}$ as the i-th instance is assigned label k or $\mathbf{x}_i \in \mathcal{C}_j$ for '$\mathbf{x}_i$ belongs to the j-th cluster' but the $\mathbf{z}$-vector notation facilitates the derivation of more advanced models later, where $\mathbf{z}$ can be a richer feature space beyond a simple indicator

The second step of Algorithm 10.1 comes from the minimisation of Eq. (10.1) wrt $\boldsymbol{\mu}_j$:

$$\begin{aligned}
\nabla_{\boldsymbol{\mu}_j} E(\boldsymbol{\mu}_j) &= \nabla_{\boldsymbol{\mu}_j} \sum_{i=1}^{n} \sum_{j=1}^{k} z_{ij} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|^2 && \triangleright \text{ derivative of error function} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{k} z_{ij} \nabla_{\boldsymbol{\mu}_j} (\mathbf{x}_i - \boldsymbol{\mu}_j)^\top (\mathbf{x}_i - \boldsymbol{\mu}_j) \\
0 &= -\sum_{i=1}^{n} \sum_{j=1}^{k} z_{ij} 2(\mathbf{x}_i - \boldsymbol{\mu}_j)^\top && \triangleright \text{ set to 0} \\
\boldsymbol{\mu}_j &= \frac{\sum_{i=1}^{n} z_{ij} \mathbf{x}_i}{\sum_{i=1}^{n} z_{ij}} && \triangleright \text{ solved for } \boldsymbol{\mu}_j
\end{aligned}$$

It just means: compute the centroids (average instance for a cluster).

10.1.1 How to choose k?

Results will depend largely on the chosen k prior to learning. How to choose k?

Normally (earlier, when we have considered hyper-parameters) the response is simply: the one which makes the expected error E (on the test set) small.

But we have no training labels with which to create a validation set and simulate testing. Have a close look at Eq. (10.1) and note that the error E should decrease inversely to k (because clusters are smaller), with $k = n$ leading to a minimum (zero distance from each point to its cluster centre). Such a solution would be of no use to us. We can instead increase k gradually and look for a *sudden* drop in error, which hints towards a nice value of k. This will appear as an "elbow" in a plot of k vs E.

A commonly used technique is that of called silhouette analysis, which provides a score between -1 (worst possible clustering) and $+1$ (good clustering). Such measures are based on some kind of calculation of intra- vs inter-cluster distance (considering instances $\mathbf{x}_i$ and cluster means/centroids $\boldsymbol{\mu}_j$).

10.2 Gaussian Mixture Models: A Solution for Probabilistic Clustering

As in classification it is often very useful to have a gauge of confidence or relevance for assigning a label, especially with a probabilistic interpretation, $p(\mathbf{z} \mid \mathbf{x})$. In Fig. 10.1 (right) we see how this is obtained by including another set of parameters: $\boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_k$, which give the shape of each cluster. This provides us a Gaussian mixture model (GMM), i.e., a mixture or combination of Gaussian distributions. We will estimate these parameters alongside means $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$, but using a 'soft' version of the k-means algorithm, which is the very well known approach of expectation maximization (EM). The derivation will be found later, in Chapter 11.

10.3 Spectral Clustering: A Solution for Non-Linear Clustering

We can understand spectral clustering as k-means performed in a kind of feature space. Feature space should remind you of basis functions earlier (feature space is where $\boldsymbol{\phi}$-instances live). Except due to the way we obtain this space (via eigen-decomposition), we can call it instead spectral space. We will denote $\mathbf{z} \in \mathcal{Z}$ (feature space $\mathcal{Z} = \mathbb{R}^\kappa$) to make the connection with vectors $\mathbf{z}$ as above, seen in the above clustering methods; and be aware that $\mathbf{Z} \in \mathbb{R}^{n \times \kappa}$ (as used in the following).

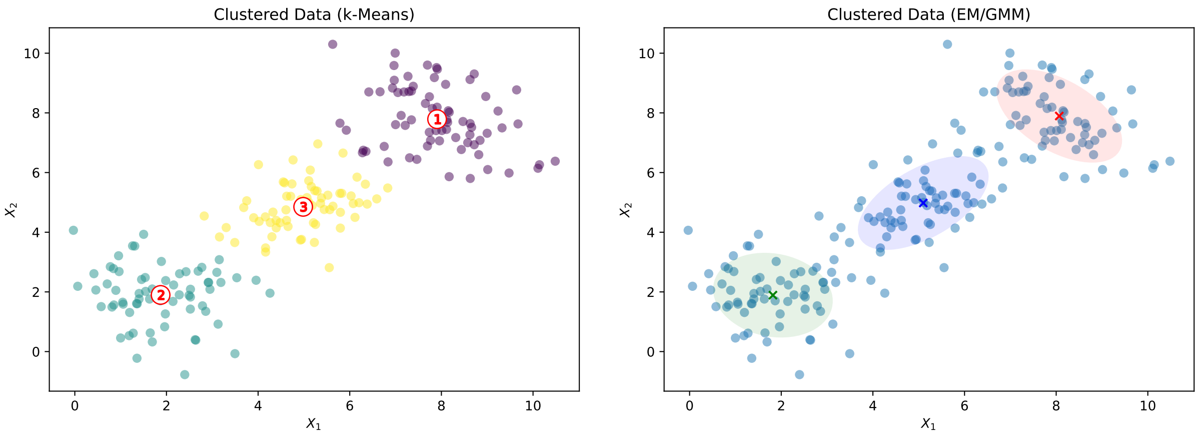


Figure 10.1: A clustering solution of k-Means (left) vs a GMM (right). Beyond just having centroids (means, $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$), a GMM also gives us a shape (covariance, $\boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_k$) for each cluster.

There is a strong relation to kernel methods (see Chapter B if you are further interested) and indeed, an equivalence if we consider a Gaussian kernel.

10.3.1 A Brief Outline

Spectral clustering can be seen as a graph partitioning problem. We define similarity function $K_{i,j} \equiv K(\mathbf{x}_i, \mathbf{x}_j)$ between each i-th and j-th instance; to derive an adjacency matrix $\mathbf{A}$ (where $A_{i,j} = 1$ if the instances should be connected; and $A_{i,j} = 0$ if not). From this we create the Laplacian matrix $\mathbf{L} = \mathbf{D} - \mathbf{A}$ (degree matrix $-$ adjacency matrix) and do eigenvalue decomposition on it:

$$\mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top = \mathbf{L}$$

We then order columns of $\mathbf{U}$ (the eigenvectors) $\propto$ eigenvalues, $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$, in order to take the top-κ components,

$$\mathbf{Z} = \left[\mathbf{u}_1 \mid \ldots \mid \mathbf{u}_\kappa\right]$$

and then apply k-means to $\mathbf{Z}$.

An example is shown in Fig. 10.2 where $\kappa = 2, k = 2$.

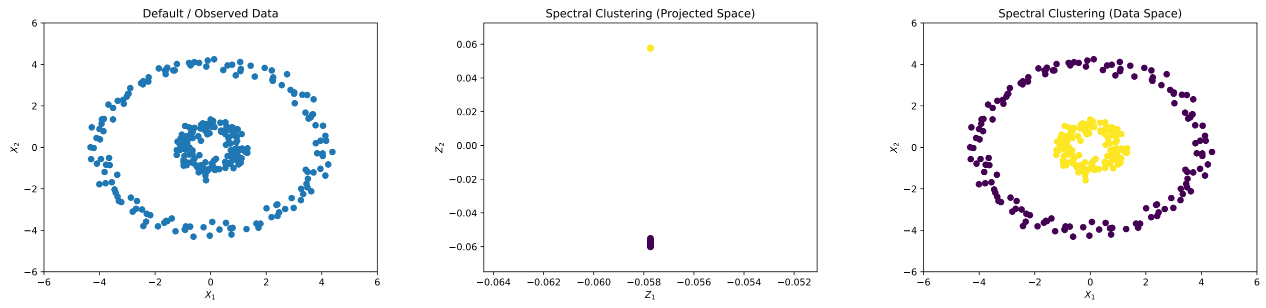


Figure 10.2: Spectral clustering, using ℓ_2 norm and $k = 2$-means clustering.

10.3.2 Why the Smallest Eigenvalues?

Spectral clustering is similar to our study of PCA in the sense of eigen-decomposition (recall: Section 9.4), except there we eigen-decompose the d-dimensional sample co-variance matrix, and were looking for the *largest* eigenvalues. In spectral clustering we eigen-decompose the n-dimensional Laplacian matrix and instead consider the *smallest* eigenvalues.

First, take note of some interesting properties of eigenvalues in $\mathbf{\Lambda}$ (having eigen-decomposed $\mathbf{L} = \mathbf{D} - \mathbf{A}$):

- The smallest eigenvalue is 0, i.e., $\lambda_1 = 0$
- The first k-smallest eigenvalues are 0 where k is the number of components

Next, to illustrate more clearly, consider a simple toy example ($n = 2$ examples):

$$\mathbf{X} = \begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix} = \begin{bmatrix} -1 \\ +1 \end{bmatrix}$$

And indeed, we can see that this holds: if we connect $x^{(1)}$ and $x^{(2)}$ (into a single component; i.e., $A_{1,2} = A_{2,1} = 1$), we get $\lambda_1 = 0$, $\lambda_2 = 2$. If we disconnect them; i.e., ($A_{1,2} = A_{2,1} = 0$) then $\lambda_1 = \lambda_2 = 0$ (two separate components).

After spectral clustering we end up with two points $\mathbf{z}^{(1)}$ and $\mathbf{z}^{(2)}$ which live in the spectral/feature space $\mathbf{Z} \in \mathbb{R}^{n,\kappa}$ ($n = 2, \kappa = 2$ in this case). A big λ_j will mean they are far apart in the corresponding j-th dimension.

In general, a small λ_j means that in the j-th dimension, items are far apart, and if $\lambda_j = 0$ then components are on top of each other in the j-th dimension. Exactly, we want connected components to be close together (such that they can easily be clustered by ordinary k-means)!

When the two points are connected: So let's take the dimension corresponding to the smallest eigenvalue[2], $\mathbf{u}_1$, and use it to go into the space of that dimension (project) and then come back (reconstruct):

$$\tilde{\mathbf{X}} = \mathbf{X}^\top \mathbf{u}_1 \mathbf{u}_1^\top = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

When the two points are *not* connected:

$$\tilde{\mathbf{X}} = \mathbf{X}^\top \mathbf{u}_1 \mathbf{u}_1^\top = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

That's precisely what we want to see: the points belonging to the same cluster are inseparable!

Of course, this depends on our threshold for connectivity.

A more in-depth discussion is given by [12]

10.4 A Comparative Look at Clustering Algorithms

[2]Assuming eigenvalues are ordered in size: $\lambda_1 < \lambda_2 < \cdots < \lambda_n$

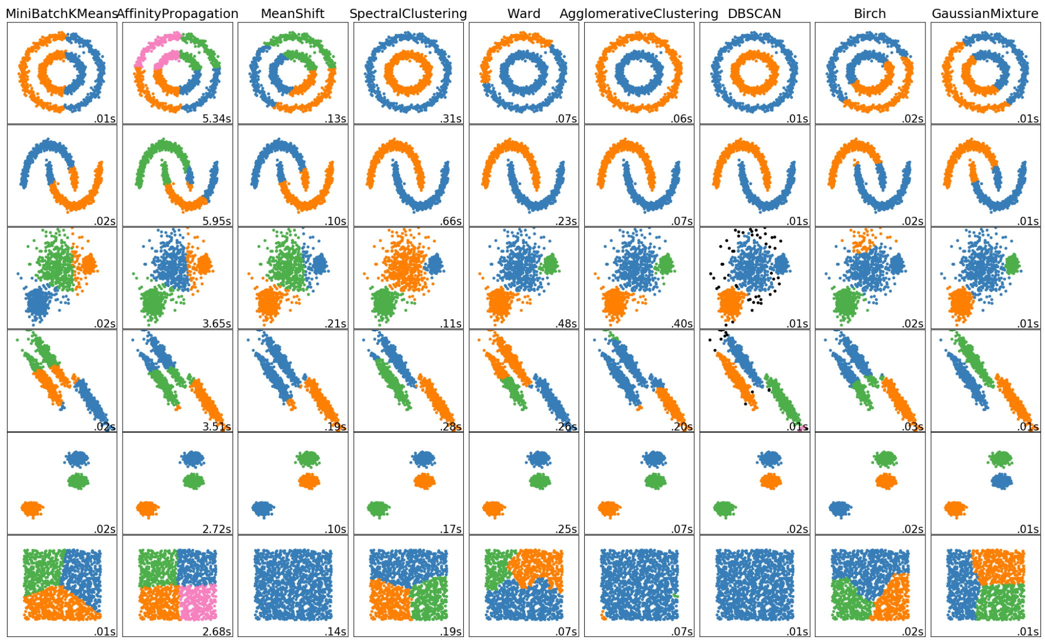


Figure 10.3: Source: https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html. In this course, we discuss k-means (first column), spectral clustering, and Gaussian mixture models (last column).

Chapter 11

Probabilistic Machine Learning and Generative Models

The focus of this chapter is to study and see in context together probabilistic models and generative models.

Generative models have been used in machine learning since its beginnings, but have become significantly more popular in recent years, in a large part due to improvements in large neural-network architectures, and successful application to image and natural-language generation.

Main Concepts

By the end of this topic you should

- Know what is a generative model, why you might want one, how to use it for generation *and* discrimination (classification)
- Be able to derive and implement a simple generative model for supervised classification: naive Bayes (NB); using maximum likelihood estimation (MLE)
- Be able to derive a generative model with latent variables: Gaussian mixture models (GMM); using expectation maximization (EM). By this stage you should also understand
  - Why you need EM (why not a 'standard approach' of MLE) in this context
  - Its potential use for probabilistic clustering, and its connection to k-means clustering
- Have a high-level view of contemporary generative models, e.g., the variational auto-encoder (VAE), generative adversarial network (GAN), and auto-regressive models for generation including recurrent neural networks (RNN, LSTM) and transformers.

By the end, you should feel that, given the right data, you could implement and train a language model or image-generation tool from scratch, or adapt some existing one.

11.1 Preliminaries

This chapter will only make sense if we recall Bayes rule:

$$p(y \mid x) = \frac{p(x \mid y)p(y)}{p(x)} \tag{11.1}$$

and conditional and joint probability,

$$p(x, y) = p(x \mid y)p(y) = p(y \mid x)p(x) \tag{11.2}$$

(note the chain rule), multiplying conditional probability and marginal probability. It will also be very helpful to remember that

$$p(x) = \sum_z p(x, z) \qquad \text{or, if } z \text{ is continuous,} \qquad p(x) = \int p(x, z)\, \mathrm{d}z$$

(i.e., marginalising out z).

In probabilistic graphical model notation (as a Bayesian network), the expressions of the joint distribution $p(x, y)$ given in Eq. (11.2) can be represented as

$$\begin{tikzcd} X \arrow[r] & Y \end{tikzcd}$$

which indicates $P(Y \mid X)P(X)$, or, *equivalently*, as

$$\begin{tikzcd} X & Y \arrow[l] \end{tikzcd}$$

which indicates $P(X \mid Y)P(Y)$; and is the main focus in this chapter. These graphical depictions always factorise as $P(\textsf{node} \mid \textsf{parents-of-node})$.

Another important concept for the following is that of likelihood:

$$\mathcal{L}(\theta) = P_\theta(\mathcal{D})$$

which tells us the probability of observing data $\mathcal{D}$, according to probability distribution P parametrized by θ. A graphical representation is shown in Fig. 11.1.

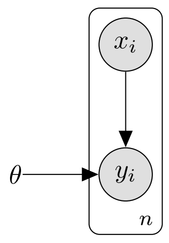


Figure 11.1: Likelihood of data $\mathcal{D} = \{x_i, y_i\}_{i=1}^n$ given parameter θ. This graphical model is in *plate notation*, showing $P_\theta(\mathcal{D}) = \prod_{i=1}^n p_\theta(y_i \mid x_i)$. Such a compact notation is only possible under the iid assumption (no dependence between data instances).

11.2 Discriminative Models – A Probabilistic Perspective

Earlier, in Section 3.1, we briefly discussed probabilistic vs generative models. And, in Chapter 3, we took a probabilistic perspective of classification, where the $\sigma(\boldsymbol{\theta}^\top \mathbf{x})$ function in logistic regression represents $p_\theta(y = 1 \mid \mathbf{x})$.

Let's write out the likelihood (see also Fig. 11.1):

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p_\theta(y_i \mid x_i) \tag{11.3}$$

In logistic regression, p_θ is a Bernoulli distribution, whose evaluation can be expressed as in the following:

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} \sigma_i^{y_i}(1 - \sigma_i)^{1-y_i}$$

where $\sigma_i = \sigma(\theta^\top x_i)$. Think about how this works out.

As we're accustomed, let's take the log for mathematical convenience (make the math easier on us, when we take the gradient later). Our log likelihood is:

$$\log \mathcal{L}(\theta) = \sum_{i=1}^{n} \{y_i \log(\sigma_i) + (1 - y_i) \log(1 - \sigma_i)\} \tag{11.4}$$

Putting a negative sign in front, we obtain exactly log loss (cf. Eq. (3.10)) as minimised in logistic regression. Therefore, maximising log likelihood for logistic regression is equivalent to minimising log loss.

What you have just learned: you already knew how to do maximum likelihood estimation.

11.3 Generative Models (for Discrimination): Naive Bayes Classifier

In Section 3.1 we covered already discriminative models, involving

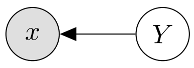
$$\begin{aligned}\widehat{y} &= \operatorname*{argmax}_{y \in \{0,1\}} p(x, y) \\ &= \operatorname*{argmax}_{y \in \{0,1\}} p(y|x)\end{aligned}$$

in the context of classification. We can also use generative models for discrimination:

$$\begin{aligned}\widehat{y} &= \operatorname*{argmax}_{y \in \{0,1\}} p(x, y) \\ &= \operatorname*{argmax}_{y \in \{0,1\}} p(x|y)p(y)\end{aligned}$$

(notice we cannot drop the $p(y)$ term because y is what is being arg-maximised).

We can see this as a kind of *query* (or *prompt* if we use the language of LLMs) $p(x, Y)$:



where the shaded x indicates that this instance has been observed and thus contains a known value; whereas the upper case Y indicates a *distribution* (and uncertainty), rather than a particular value[1].

If we can generalise to feature vector (instance) $\mathbf{x} \in \mathbb{R}^d$, we have ourselves a classifier.

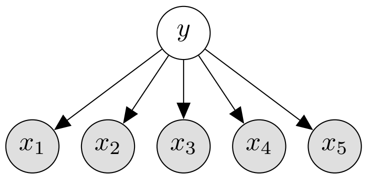


Figure 11.2: A probabilistic graphical model of Naive Bayes. Each edge represents $P(x_j|y)$; one for each attribute, $j = 1, \ldots, 5$.

The Naive Bayes classifier is the '`hello world`' of generative models. As a probabilistic graphical model, it is displayed in Fig. 11.2, as it is used for classification (that is to say, where $\mathbf{x} = [x_1, \ldots, x_d]$ has been observed, and a classification is pending).

It defines the joint distribution as follows:

$$p(\mathbf{x}, y) = p(\mathbf{x} \mid y)p(y) \tag{11.5}$$

$$= \prod_{j=1}^{d} p(x_j \mid y)p(y) \tag{11.6}$$

where the naivety is a shown as a simple product of features (indicating feature independence).

Putting an argmax in front is sufficient to create a classifier:

$$\widehat{y} = \underset{y}{\operatorname{argmax}}\, p(\mathbf{x} \mid y)p(y)$$

but how to train this thing?

11.3.1 Discrete case, $x \in \{0, 1\}$

The above indicates that we will need parameters to model terms $p(x_1 \mid y), \ldots, p(x_d \mid y)$ and $p(y)$, for each possible y.

First we consider the discrete binary case of a single feature $x \in \{0, 1\}$, and binary class $y \in \{0, 1\}$. We just need to two tables to completely specify the model:

$$\theta = \{\boldsymbol{\phi}, \boldsymbol{\pi}\} = \left\{ \underbrace{\begin{bmatrix} p(X=0|Y=0) & p(X=0|Y=1) \\ p(X=1|Y=0) & p(X=1|Y=1) \end{bmatrix}}_{\boldsymbol{\phi}}, \underbrace{\begin{bmatrix} P(Y=0) & P(Y=1) \end{bmatrix}}_{\boldsymbol{\pi}} \right\} = \left\{ \begin{bmatrix} \phi_{1,1} & \phi_{1,2} \\ \phi_{2,1} & \phi_{2,2} \end{bmatrix}, \begin{bmatrix} \pi_1, \pi_2 \end{bmatrix} \right\}$$

(actually, in the binary case here, half of each table is redundant, since, e.g., $P(Y = 0) = 1 - P(Y = 1)$). Obviously, when $d > 1$ (number of features) or $m > 1$ (number of classes), more values are needed for the sufficient statistics.

Naive Bayes is often rather good (at least, good for such a simple model) under a bag-of-words model ($x_j^{(i)} = 1$ when the j-th word in the dictionary appears in the i-th document-instance) at text classification, but it is not able to generate grammatically-sound sentences, or realistic images, or classifying many other types of data, all due to its naive assumption of independence among features.

[1]We can use lowercase y equivalently, when it is clear from context

11.3.2 Continuous case, $x \in \mathbb{R}$

When $x \in \mathbb{R}$ (again, let's first suppose a single feature; and again a binary class, $y \in \{0, 1\}$) we need to model $p(x|y)$ as a probability density function. We can use a Gaussian:

$$p(x \mid y) = \mathcal{N}(x \mid \mu_y, \sigma_y^2) \tag{11.7}$$

Note that we need one Gaussian (mean, variance) for each $y \in \{0, 1\}$. So at this point we have [to learn] parameters

$$\theta = \{\mu_y, \sigma_y, \pi_y\} \text{ for } y \in \{0, 1\} \tag{11.8}$$

to completely specify Eq. (11.7).

11.4 Maximum Likelihood Estimation (MLE): Standard Case

How to estimate these parameters θ? In other words, how to do the *learning*[2]?

As a reminder, the standard procedure for MLE is outlined here in Table 11.1.

Given training data, a MLE estimate of parameters $\boldsymbol{\theta}$ can be obtained via the following:

1. Write out the likelihood using parameters θ
2. (Take the log)
3. Take the derivative
4. Set to 0, solve for θ

Table 11.1: Standard procedure to Maximum Likelihood Estimation

11.4.1 MLE for estimating μ in the context of a Gaussian Model

Forget about Naive bayes for a minute, as we look at an even simpler problem. Suppose we are given training data $\{x_1, \ldots, x_n\}$, and taking the iid assumption[3] and that the distribution in question is Gaussian,

[2]Again, we can debate the nuances of *learning* vs *fitting* vs parameter *estimation* vs .... But in any case, this corresponds to a `.fit(X,y)` call in SCIKIT-LEARN

[3]An assumption we make throughout the course; an explanation was given in Section 2.7

$x_i \sim \mathcal{N}(x \mid \mu, \sigma)$. What should be our estimate for μ? We simply follow the steps outlined in Table 11.1:

$$\begin{aligned}
\mathcal{L}(\theta) &= \prod_{i=1}^{n} p(x_i) \quad \triangleright \text{ Write out the likelihood} \\
&= \prod_{i=1}^{n} \mathcal{N}(x_i \mid \mu, \sigma^2) \quad \triangleright \text{ \ldots which involves a Gaussian} \\
&= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(\frac{-(x_i-\mu)^2}{2\sigma^2}\right) \quad \triangleright \text{ expand } \mathcal{N} \\
\log \mathcal{L}(\theta) &= \sum_{i=1}^{n} \left(\log \frac{1}{\sqrt{2\pi\sigma^2}} + \left(\frac{-(x_i-\mu)^2}{2\sigma^2}\right)\right) \quad \triangleright \text{ Take the log} \\
&= \sum_{i=1}^{n} \frac{-(x_i-\mu)^2}{2\sigma^2} \quad \triangleright \text{ drop constant not dependent on } \mu \\
\nabla_\mu \log \mathcal{L}(\theta) &= \frac{1}{2\sigma^2} \sum_{i=1}^{n} 2(x_i - \mu) \quad \triangleright \text{ Take the derivative} \\
0 &= \sum_{i=1}^{n} (x_i - \mu) \quad \triangleright \text{ Set to 0} \\
\sum_{i=1}^{n} x_i &= n\mu \quad \triangleright \text{ Solve for } \mu \ldots \\
\hat{\mu} &= \frac{1}{n} \sum_{i=1}^{n} x_i
\end{aligned}$$

It's just the average! This should coincide with our intuition.

11.4.2 MLE for estimating μ_y in the context of Gaussian Naive Bayes

Now back to Naive Bayes, a model trained via supervised learning, under class labels y_i. Suppose we are given iid training data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, and we assume it can be modeled with a Gaussian distribution $\mathcal{N}(x|\mu_y, \sigma_y^2)$, as per Section 11.3.2. In the a binary classification problem, $y \in \{0, 1\}$, we have two distributions: $\mathcal{N}(x|\mu_0, \sigma_0^2)$ and $\mathcal{N}(x|\mu_1, \sigma_1^2)$; i.e., two sets of parameters, as per Eq. (11.8).

Let's try using MLE for estimating $\hat{\mu}_1$: Since we're deriving μ_1 in particular, only samples $x_i|y_i = 1$ are of interest to us:

$$\begin{aligned}
\mathcal{L}(\theta) &= \prod_{i=1}^{n} p(x_i, y_i) \quad \triangleright \text{ Write out the likelihood} \\
&= \prod_{i=1}^{n} \mathcal{N}(x_i|\mu_1, \sigma_1^2)^{y_i} \pi_1^{y_i} \cdot \mathcal{N}(x_i|\mu_0, \sigma_0^2)^{1-y_i} \pi_0^{1-y_i} \\
&= \prod_{x_i|y_i=1} \mathcal{N}(x_i|\mu_1, \sigma_1^2) \quad \triangleright \text{ Only over instances } x_i \text{ where } y_i = 1 \\
\nabla_{\mu_1} \mathcal{L}(\theta) &= \nabla_{\mu_1} \ldots \\
\ldots &= \ldots \\
\hat{\mu}_1 &= \frac{1}{n_1} \sum_{x_i|y_i=1} x_i
\end{aligned}$$

where n_1 is the number of examples (x_i, y_i) having label $y_1 = 1$. Apart from that, the derivation was not so different from $\hat{\mu}$ just above; this parameter of interest was only present in $\mathcal{N}(x_i|\mu_1, \sigma_1^2)$. Again, this should coincide with our intuition. The other parameters $\mu_0, \sigma_0, \sigma_1, \pi_0, \pi_1$ can be derived in a similar way.

11.4.3 MLE for estimating π_y in the context of Gaussian Naive Bayes

In estimating class probabilities $\pi_y = p(y)$, we are estimating the parameter of a Bernoulli distribution; by following the same procedure as above (likelihood $\mathcal{L}(\pi_1), \dots$),

$$\begin{aligned}
\mathcal{L}(\theta) &= \prod_{i=1}^{n} p(x_i, y_i) \quad \triangleright \text{ Write out the likelihood} \\
&= \prod_{i=1}^{n} p(x_i \mid y_i) p(y_i) \\
&= \prod_{i=1}^{n} \mathcal{N}(x_i|\mu_1, \sigma_1^2)^{y_i} \cdot \pi_1^{y_i} \cdot \mathcal{N}(x_i|\mu_0, \sigma_0^2)^{1-y_i} \cdot \pi_0^{1-y_i} \\
\log \mathcal{L}(\theta) &= \sum_{i=1}^{n} y_i \log \mathcal{N}(x_i|\mu_1, \sigma_1^2) + y_i \log \pi_1 + (1-y_i)\mathcal{N}(x_i|\mu_0, \sigma_0^2) + (1-y_i)\log \pi_0 \\
0 &= \nabla_{\pi_1} \sum_{x_i|y_i=1} \log y_i \mathcal{N}(x_i|\mu_1, \sigma_1^2) + y_i \log \pi_1 \\
\dots &= \dots \\
\hat{\pi}_1 &= \frac{1}{n} \sum_{i=1}^{n} y_i
\end{aligned}$$

we thus find that π_1 is simply the number of examples belonging to class 1 in the training set.

Particularly to the binary case, $\pi_0 = 1 - \pi_1$.

11.4.4 Generalisation to > 1 features

When we have d features, under the feature-independence assumption of Naive Bayes:

$$\begin{aligned}
\mathcal{L}(\theta) &= \prod_{i=1}^{n} \prod_{j=1}^{d} p(x_j^{(i)} \mid y^{(i)}) \\
&= \dots
\end{aligned}$$

so it requires estimating d means and co-variances separately *for each class*, such as to completely specify

$$\mathcal{N}(x \mid \mu_{y,1}, \sigma_{y,1}) \cdots \mathcal{N}(x \mid \mu_{y,d}, \sigma_{y,d}) \tag{11.9}$$

for both $y = 0$ and $y = 1$. But this is not more complicated than what we have already done so far, it just involves more [of the same kind of] parameters.

Note that the independence assumption of Naive Bayes implies that we can write

$$\mathbf{x} \sim \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_y, \mathbf{I}_y) \tag{11.10}$$

where $\mathbf{I}_y$ is a $d \times d$ identity matrix; i.e., zero covariance. So, $\boldsymbol{\mu}_y = \mu_{y,1}, \dots, \mu_{y,d}$ and $\text{diag}(\mathbf{I}_y) = \sigma_{y,1}, \dots, \sigma_{y,d}$ as per Eq. (11.9).

By now, it should be clear how we can use MLE to obtain estimates of all parameters

$$\theta = \{\mu_{y,j}, \sigma_{y,j}, \pi_y\} \text{ for } j \in \{1, \dots, d\} \text{ and } y \in \{1, \dots, k\}$$

and we note that they can all be estimated in closed form.

11.4.5 Generalisation to > 2 class labels

In the general multi-class case (when $y \in \{1, \dots, m\}$ for $m > 2$) it becomes convenient to use indicator variables $y_k^{(i)} \Leftrightarrow [\![\text{instance } i \text{ is assigned class } k]\!]$, hence with one-hot-encoded classes $\mathbf{y}_i$:

$$\begin{aligned}\mathcal{L} &= \prod_{i=1}^{n} p(\mathbf{x}_i|\mathbf{y}_i) \\ &= \prod_{i=1}^{n}\prod_{j=1}^{d}\prod_{k=1}^{m} \mathcal{N}(x_k^{(i)} \mid \mu_{k,j}, \sigma_{k,j}^2)^{y_k^{(i)}} \\ &= \dots\end{aligned}$$

Notice that taking the log will bring down the exponent as such:

$$y_k^{(i)} \log \mathcal{N}(x_k^{(i)} \mid \mu_{k,j}, \sigma_{k,j}^2)$$

where $y_k^{(i)}$ is acting like a binary switch on the Gaussian term; and all but one will be 'turned off' for any given i.

In the multi-class case we also need to bear in mind the constraint that $\sum_{k=1}^{m} \pi_k = 1$ (we can use lagrange multipliers for that). But still this can be done parameter-wise and in closed form, for all parameters.

11.5 Gaussian Bayes

We can overcome the independence assumption by including co-variance, so that instead of Eq. (11.10), we have, more generally,

$$\mathbf{x} \sim \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y) \tag{11.11}$$

This is known sometimes as Gaussian Bayes (as opposed to Gaussian *Naive* Bayes). MLE involves (alongside estimating the means as just above) estimating the sample co-variance matrix (as per Eq. (9.4)).

11.6 Generative AI

Of course, as the name suggests, we can use generative models for *generating* data:

$$y \sim p(y) \tag{11.12}$$

$$x \sim p(x \mid y) \tag{11.13}$$

For example, we might want to generate a picture:

$$\begin{aligned}\mathsf{cat\_label} &\sim p(y) \\ \mathsf{picture\_of\_a\_cat} &\sim p(x \mid \mathsf{cat\_label})\end{aligned}$$

We could also set directly $y = \mathsf{cat}$ if we knew already we wanted a cat. We might also generate a sentence of text describing a cat. This depends on a suitable architecture.

11.7 Gaussian Mixture Model (GMM)

If we mix Gaussian distributions together according to different proportions, the result would be a Gaussian mixture model (GMM):

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z}) \tag{11.14}$$

where, specifically,

$$p(\mathbf{z}) = \prod_{k=1}^{m} \pi_k^{z_k} \quad \text{and} \quad p(\mathbf{x} \mid \mathbf{z}) = \prod_{k=1}^{m} \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}$$

which by now should not see mysterious; as it is exactly what we had above. However, we have moved to the unsupervised setting; indeed we wouldn't usually mix together *classes* in this way, as in classification we precisely want to separate (discriminate between) them.

GMMs are very powerful, flexible and very widely used in machine learning. The first important thing to notice that we have expressed $p(\mathbf{x}, \mathbf{z})$ in a generative form. This is illustrated by the Bayesian network in Fig. 11.3.

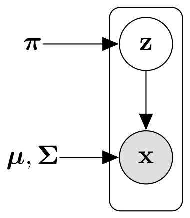


Figure 11.3: A GMM as a Bayesian network; where $\theta = \{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$; the same kind of parameters we considered for naive Bayes classifier.

For a given parameters θ we can sample from this using ancestral sampling ($\mathbf{z}$ is the 'ancestor' of $\mathbf{x}$ in Fig. 11.3):

$$\begin{aligned} &\mathbf{z} \sim p(\mathbf{z}) \quad && \triangleright \text{ Draw from Bernoulli/multinoulli parametrised by } \boldsymbol{\pi} \\ &\mathbf{x} \sim p(\mathbf{x} \mid \mathbf{z}) \quad && \triangleright \text{ Draw from Gaussian, parametrized by } \boldsymbol{\mu}, \boldsymbol{\Sigma} \end{aligned}$$

In this case, only $\mathbf{x}$ is of interest, and $\mathbf{z}$ is only a means to an end (greater modelling capacity). Thus, our marginal distribution becomes of interest:

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}) = \sum_{k=1}^{m} \{\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \cdot \pi_k\} \tag{11.15}$$

Note that we can add arbitrary expressiveness to $p(\mathbf{x})$ by varying the number of hidden compontents (dimensionality of $\mathbf{z}$, i.e., m, which we decide as part of our architecture). This is unlike previously, e.g., in Eq. (11.11), where we were limited to the shape of a single Gaussian distribution to express a class concept.

11.7.1 MLE for estimating parameters of a GMM

Our log-likelihood (over all instances and parameters) is:

$$\begin{aligned} \log \mathcal{L}(\theta) &= \log \prod_{i=1}^{n} \prod_{k=1}^{m} \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_{i,k}} \cdot \pi_k^{z_{i,k}} \qquad (11.16) \\ &= \sum_{i=1}^{n} \sum_{k=1}^{m} z_{i,k} \cdot \{\log \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) + \log \pi_k\} \end{aligned}$$

we can easily solve for any θ as above, but only if we have $\mathbf{z}$, *which we do not*, because they are latent variables! This is an unsupervised task.

The point of this subsection is to express the likelihood and then to realise: this is a hard problem! Without further development we are stuck; we cannot use the standard approach (as outlined above), because there are too many unknowns. You could try standard MLE and see where you 'get stuck'.

11.8 Expectation Maximization (EM) for GMMs

So, we can't solve directly for parameters θ because we have not yet any values $\mathbf{z}_i$. And we cannot estimate them without the parameters.

In this section we will solve this chicken-and-egg problem via the expectation maximization (EM) algorithm.

11.8.1 Initialization

To get around the problem of not having θ, we can just make up some random values first. So we can initialize all parameter values to some random numbers (still meeting certain constraints for certain parameters; for example $\sum_k \pi_k = 1$).

Understandably, using random numbers causes quite a bit of uncertainty as to which cluster (component) a given instance should belong to. When we speak of uncertainty, a natural reaction should be to model it with an expectation. This takes us to the E step of EM.

11.8.2 The $\mathbb{E}$ step (Expectation)

Having an idea of $\mathbf{z}$ (for any given $\mathbf{x}$) would help. We calculate its expectation,

$$\mathbb{E}[\mathbf{z}] = \sum_{\mathbf{z}} \mathbf{z} \cdot p(\mathbf{z} \mid \mathbf{x})$$

one at a time:

$$\begin{aligned}
\mathbb{E}[z_k] &= \sum_{z_k \in \{0,1\}} z_k \cdot p(z_k \mid \mathbf{x}) \\
&= p(z_k = 1 \mid \mathbf{x}) \\
&= \frac{p(\mathbf{x} \mid z_k = 1) p(z_k = 1)}{p(\mathbf{x})} \\
&= \frac{\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \cdot \pi_k}{\sum_{k=1}^{m} \{\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \cdot \pi_k\}} && \triangleright \text{ the numerator from Eq. (11.14) and } p(\mathbf{x}) \text{ from Eq. (11.15)} \\
&= r_k && \triangleright \text{ the } \textit{relevance} \text{ of } \mathbf{x} \text{ to the } k\text{-th component}
\end{aligned}$$

where the denominator is just to correctly normalise the conditional. So, essentially we just use $p(\mathbf{z}, \mathbf{x})$, which was well defined above.

And let $r_{i,k}$ concern the i-th instance $\mathbf{x}_i, \mathbf{z}_i$ in particular.

11.8.3 The M step (Maximisation)

Now that we have each $\mathbf{z}_i$, or at least an estimation thereof (associated uncertainty $r_{i,k} = \mathbb{E}[z_{i,k}]$), the problem proceeds almost in a supervised fashion.

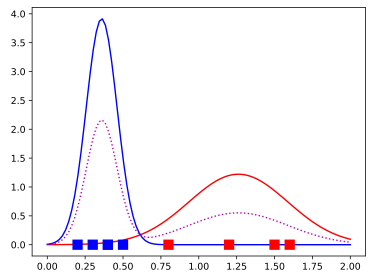


Figure 11.4: GMM-EM on the 'beetles' toy dataset; showing $p(\mathbf{x}|y=0)$, $p(\mathbf{x}|y=1)$, and $p(\mathbf{x})$.

The M in EM refers to Maximization, in in particular the M of MLE; for estimating $\hat{\theta}$. We take the likelihood from Eq. (11.16), but replacing $z_{i,k}$ with $r_{i,k}$, and perform MLE for the parameters. Let's proceed, one at a time, starting with

$$\hat{\pi}_k = \frac{1}{n}\sum_{i=1}^{n} r_{i,k}$$

and so on for all parameters (they will be solvable in closed form, as earlier, but this time plugging in $r_{i,k}$ (from the $\mathbb{E}$ step) instead of $z_{i,k}$.

11.8.4 The EM algorithm: summary

Until convergence (e.g., until there is almost no change from one step to another):

1. Initialize parameter values $\hat{\theta}$
2. E. Step: Compute all $\{r_{i,k}\}$ using $\hat{\theta}$
3. M. Step: Compute all $\hat{\theta}$ using $\{(\mathbf{x}_i, \mathbf{r}_i)\}_{i=1}^{n}$
4. Repeat from 2.

11.9 GMM-EM for Probabilistic Clustering

What did do we get from everything so far? What do we walk away with after convergence of EM on a GMM?

One thing we can do is probabilistic clustering. If we just want a cluster label for any point $\mathbf{x}$, then we have it:

$$\hat{z} = \underset{z\in\{1,\ldots,m\}}{\operatorname{argmax}}\ p(z,\mathbf{x}) = \underset{z\in\{1,\ldots,m\}}{\operatorname{argmax}}\ \pi_z \cdot \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)$$

But this models offers us much more, we also have a probability

$$r_{i,k} = P(z=k|\mathbf{x})$$

(that instance $\mathbf{x}$ belongs to k-th cluster) for any given $\mathbf{x}$.

And we have a description of the location $\boldsymbol{\mu}$ and *shape* $\boldsymbol{\Sigma}$ of all clusters, and their relative influence π. All together, we have an expression of the data; a Gaussian mixture *model of the data*.

11.10 k-Means Clustering, or 'Maximization Maximization'

What happens if instead of the $\mathbb{E}$ step we do a hard classification, i.e., just plug in

$$\hat{z}_i = \operatorname*{argmax}_{z \in \{1,\ldots,m\}} p(z, \mathbf{x}_i) \tag{11.17}$$

instead of $r_{i,k}$, and use those estimated labels to calculate the parameters?

Answer: The algorithm becomes 'Maximization Maximization' also known, in this context, as the k-Means Clustering algorithm. We covered this algorithm in Section 10.1.

11.11 Mixture Density Networks

We can implement GMMs with deep neural networks; such a network is called a mixture density network.

Part III

Appendices (Fundamentals)

Appendix A

Fundamentals

Material that appears here in the Appendix is intended to introduce terms and concepts that are be used in the course, but not necessarily to explain them (at least, not in great detail). In other words, material here is intended as a reminder/refresher or as a minimalistic introduction. If any of what you read in the following seems very unfamiliar, locate the relevant keyword in order to find a more complete reference.

A note about notation: Because machine learning spans different scientific areas, it is much more difficult than it might seem to come up with a single concise notation that can survive all topics without any changes. It is why it is important to try understand the base concepts, rather than relying on recognising certain symbols and formulae.

A.1 Vectors and Sets, and Main Notation

Required for topic 1 (Introduction) onwards

To follow Chapter 1, you need to know what a matrix, a vector and a set is; and how data can be represented in this way.

In line with common statistical notation, we use upper case (e.g., X or Y) to denote a random variable and, in relation to data, an abstract feature or label concept, for example 'square metres', which would correspond to the column headers in a table. And lower case lower (e.g., x or y) a specific data value, i.e., realisations of those random variables, e.g., 20.4. In the latter case, we usually append subscripts or superscripts specifying the indices so we know where exactly to find this value in the data, such that, in terms of a table, $x_{i,j}$ is the value of the *cell* where the i-th row intersects with the j-th column. For example, $x_{3,2} = 47$ the value of the 3-rd feature of the 2-nd instance.

The entire i-th row can be represented as $\mathbf{x}_i$. A data set can either be represented as a set of such vectors where $\mathbf{x}_i$ is the i-th element, or a matrix where i is the i-th row. The label y_i is associated with the i-th row, and so the vector $\mathbf{y} = [y_1, \ldots, y_n]^\top$ is typically assumed to be a column vector.

Table A.1 provides an initial set of notation to work with for Chapter 1.

Symbol	**Meaning**
f	a model; a function that maps d-dimensional input $\mathbf{x} = [x_1, \ldots, x_d]$ to output y
$f_\star$	specifically the ground-truth model (the one we never know in practice)
$\widehat{f}$	specifically 'our' model (what a machine learning algorithm produces)
ϵ	the irreducible error; such that from $y_i = f_\star(\mathbf{x}_i) + \epsilon$ we observe only $(\mathbf{x}_i, y_i)$
$\widehat{y}$	a prediction obtained from our model; $\widehat{y} = \widehat{f}(\mathbf{x})$
e_i	the residual error; obtained *for any training pair*, as $e_i = y_i - \widehat{y}_i$
ℓ	the loss metric

Table A.1: A summary of some notation introduced in Chapter 1.

A.2 Matrix Multiplication

Required for Topic 2 (Least Squares Regression) onwards

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a matrix of n rows and d columns, and let $\mathbf{W} \in \mathbb{R}^{d \times m}$ have d rows and m columns. We can multiply these matrices together (this is *only possible* when number of columns of $\mathbf{X}$ equals the number of rows in $\mathbf{W}$) as

$$\mathbf{Y} = \mathbf{XW}$$

where

$$y_{i,j} = (\mathbf{XW})_{i,j} = \mathbf{X}_{i,:}\mathbf{W}_{:,j} = \sum_{k=1}^{d} x_{i,k} w_{j,k}$$

is the multiplication of the i-th row(-vector) with the j-th column(-vector). We can also write:

$$y_{i,j} = [x_{i,1}, x_{i,2}, \ldots, x_{i,d}] \cdot \begin{bmatrix} w_{1,j} \\ w_{2,j} \\ \vdots \\ w_{d,j} \end{bmatrix}$$

Multiplying a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ with a vector $\mathbf{w} \in \mathbb{R}^{d \times 1}$ gives:

$$\mathbf{Xw} = \mathbf{y}$$

where $\mathbf{y} = [y_1, \ldots, y_n]^\top$ is a column vector, $\mathbf{y} \in \mathbb{R}^{n \times 1}$. For example ($n = 4, d = 3$):

$$\mathbf{Xw} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ x_{4,1} & x_{4,2} & x_{4,3} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1\mathbf{w} \\ \mathbf{x}_2\mathbf{w} \\ \mathbf{x}_3\mathbf{w} \\ \mathbf{x}_4\mathbf{w} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{d} x_{1,j} w_j \\ \sum_{j=1}^{d} x_{2,j} w_j \\ \sum_{j=1}^{d} x_{3,j} w_j \\ \sum_{j=1}^{d} x_{4,j} w_j \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \mathbf{y}$$

we see that $y_i = \sum_{j=1}^{d} x_{i,j} w_j$.

If $\boldsymbol{\epsilon} = [e_1, e_2, \ldots, e_n]^\top$ (a column vector) then the dot product:

$$\boldsymbol{\epsilon}^\top \boldsymbol{\epsilon} = [e_1, e_2, \ldots, e_n] \cdot \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix} = \sum_{i=1}^{n} e_i^2 \tag{A.1}$$

(the result is a scalar value, hence this operation is also known as the scalar product).

Another example:

$$\mathbf{X}^\top \mathbf{e} = \begin{bmatrix} x_{1,1} & x_{2,1} & x_{3,1} & x_{4,1} \\ x_{1,2} & x_{2,2} & x_{3,2} & x_{4,2} \\ x_{1,3} & x_{2,3} & x_{3,3} & x_{4,3} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} \mathbf{e} = \begin{bmatrix} \sum_{i=1}^{n} x_{i,1} e_i \\ \sum_{i=1}^{n} x_{i,2} e_i \\ \sum_{i=1}^{n} x_{i,3} e_i \end{bmatrix} = \sum_{i=1}^{n} \mathbf{x}_i e_i = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix} = \mathbf{g} \tag{A.2}$$

Some properties of matrix multiplication:

$$\begin{aligned}
\mathbf{A}(\mathbf{B}+\mathbf{C}) &= \mathbf{AB}+\mathbf{AC} && \triangleright \text{ It is distributive} \\
(\mathbf{B}+\mathbf{C})\mathbf{A} &= \mathbf{BA}+\mathbf{CA} \\
\mathbf{A}(\mathbf{BC}) &= (\mathbf{AB})\mathbf{C} && \triangleright \text{ It is associative} \\
\mathbf{AB} &\neq \mathbf{BA} && \triangleright \text{ It is not commutative}
\end{aligned} \tag{A.3}$$

Some matrix transpose properties and examples (in the context of multiplication):

$$\begin{aligned}
(\mathbf{A}^\top)^\top &= \mathbf{A} \\
(\mathbf{AB})^\top &= \mathbf{B}^\top\mathbf{A}^\top \quad \triangleright \text{ N.B. Terms reversed!} \\
(\mathbf{A}^\top\mathbf{w})^\top &= \mathbf{w}^\top\mathbf{A} \\
\mathbf{w}^\top\mathbf{A}\mathbf{x} &= (\mathbf{A}^\top\mathbf{w})^\top\mathbf{x}
\end{aligned} \tag{A.4}$$

$$\begin{aligned}
(c\mathbf{A})^\top &= c\mathbf{A}^\top \quad \triangleright \text{ Transpose with scalars (no effect)} \\
(\mathbf{y}-\mathbf{Xw})^\top &= \mathbf{y}^\top - (\mathbf{Xw})^\top \quad \triangleright \text{ Transpose respects addition}
\end{aligned} \tag{A.5}$$

Combining these rules, we can write

$$\begin{aligned}
(\mathbf{XW})^\top(\mathbf{XW}) &= \mathbf{W}^\top\mathbf{X}^\top(\mathbf{XW}) && \triangleright \text{ From Eq. (A.4)} \\
&= \mathbf{W}^\top(\mathbf{X}^\top\mathbf{X})\mathbf{W} && \triangleright \text{ From Eq. (A.3)}
\end{aligned}$$

If two different formulations result in a scalar, they may be combined:

$$\underbrace{\mathbf{y}^\top\mathbf{Xw}}_{\alpha} + \underbrace{\mathbf{w}^\top\mathbf{X}^\top\mathbf{y}}_{\alpha} = 2\alpha = 2\mathbf{w}^\top\mathbf{X}^\top\mathbf{y} = 2\mathbf{y}^\top\mathbf{Xw} \tag{A.6}$$

A.3 Gradients and Vector/Matrix Calculus

Required for Topic 2 (Gradient Descent) onwards

Let $\mathbf{w} = [w_1, \ldots, w_d]^\top$ (note: a column vector). The gradient of a function $E(\mathbf{w})$ wrt $\mathbf{w}$ is

$$\mathbf{g} = \nabla_{\mathbf{w}} E(\mathbf{w}) = \left[\frac{\partial}{\partial w_1} E(\mathbf{w}), \ldots, \frac{\partial}{\partial w_d} E(\mathbf{w})\right]^\top \tag{A.7}$$

i.e., a column vector of d partial derivatives $\mathbf{g} = [g_1, \ldots, g_d]^\top$ where

$$\begin{aligned} g_1 &= [\nabla_{\mathbf{w}} E(\mathbf{w})]_1 = \frac{\partial}{\partial w_1} E(\mathbf{w}) \\ g_2 &= [\nabla_{\mathbf{w}} E(\mathbf{w})]_2 = \frac{\partial}{\partial w_2} E(\mathbf{w}) \\ \vdots &= \vdots \\ g_d &= [\nabla_{\mathbf{w}} E(\mathbf{w})]_d = \frac{\partial}{\partial w_d} E(\mathbf{w}) \end{aligned}$$

Some handy rules for derivatives:

$$\begin{aligned} \nabla_{\mathbf{w}} \mathbf{w}^\top \mathbf{x} &= \mathbf{x} && \text{(A.8)} \\ \nabla_{\mathbf{w}} \mathbf{w}^\top \mathbf{w} &= 2\mathbf{w} && \text{(A.9)} \\ \nabla_{\mathbf{w}} \mathbf{w}^\top \boldsymbol{\Sigma} \mathbf{w} &= 2\boldsymbol{\Sigma}\mathbf{w} && \text{(A.10)} \\ \nabla_{\mathbf{w}} \mathbf{X}^\top &= (\nabla_{\mathbf{w}} \mathbf{X})^\top \end{aligned}$$

where $\mathbf{x}$ is a row vector of equal length to $\mathbf{w}$, and $\boldsymbol{\Sigma}$ is a symmetric matrix.

Note that when $\mathbf{w}^\top \mathbf{z} \in \mathbb{R}$ (i.e., a *scalar* product; $\mathbf{z}$ is a column-vector shaped like $\mathbf{w}$), then $\mathbf{w}^\top \mathbf{z} = \mathbf{z}^\top \mathbf{w}$, and thus

$$\nabla_{\mathbf{w}} \mathbf{w}^\top \mathbf{z} = \frac{\mathrm{d}\mathbf{w}^\top \mathbf{z}}{\mathrm{d}\mathbf{w}} = \frac{\mathrm{d}\mathbf{z}^\top \mathbf{w}}{\mathrm{d}\mathbf{w}} = \mathbf{z} = \mathbf{z}^\top \tag{A.11}$$

See also Eq. (A.6).

A.4 Random Variables and Probability Distributions

Required for Topic 3 (Logistic Regression) onwards

A random variable X has probability distribution $P(X)$ (or just P, when the context is clear). We can write this as:

$$X \sim P$$

meaning that X is distributed according to P.

A particular instance x, is a realisation of X, having domain (or support) $\mathcal{X}$, i.e.,

$$x \in \mathcal{X}$$

A probability mass function (pmf), when X is a discrete/nominal variable, provides, for all $x \in \mathcal{X}$,

$$P(X = x) \tag{A.12}$$

where $P(X = x) \in [0, 1]$ and $\sum_{x \in \mathcal{X}} P(X = x) = 1$. Often we will use shorthand $P(x) \equiv P(X = x)$ Sometimes we will also write $p(x)$, when there is no urgency to distinguish the fact that x takes a discrete distribution (i.e., is a pmf).

So $P(x)$, for a specific x, is *not* a distribution, rather is a *query* to the pmf, of the probability that X takes value x. The distribution shall be denoted $P(X)$ (as mentioned above).

Example A.4.1- Flipping a fair coin: Bernoulli distribution

Let X be a random variable representing the flip of a fair coin, so $\mathcal{X} = \{\mathsf{Heads}, \mathsf{Tails}\}$. We can represent this distribution as:

x	Heads	Tails
$P(X = x)$	0.50	0.50

This is a Bernoulli distribution, so we can denote this $\mathcal{B}(0.5)$ (the 0.5 is value of the parameter specifying $P(X = \mathsf{Heads}) = 0.5$. And thus $X \sim \mathcal{B}(0.5)$ indicates that random variable X of a Bernoulli distribution. We can denote a coin flip as $x \sim \mathcal{B}(0.5)$ (realization of the random variable, or a sample from the distribution). The result (value of x/flip of the coin) will either be Heads or Tails. We denote x_i as the i-th outcome.

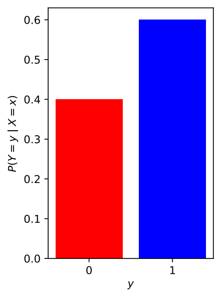


Figure A.1: Visualisation of an example Bernoulli distribution, for variable Y; in this case $\mathcal{B}(0.6) = P(Y|X = x)$.

Example A.4.2- Categorical distribution

Following on from Example A.4 we can introduce a random variable $Z = \sum_{i=1}^{2} [\![X_i = \mathsf{Heads}]\!]$ which represents the total number of Heads that have come up after 2 flips. Here, $\mathcal{Z} = \{0, 1, 2\}$. This distribution (and its pmf) can be represented as:

z	0	1	2
$P(Z = z)$	0.25	0.50	0.25

Notice how Z 'inherits' the randomness of the Xs. This distribution $P(Z)$ is shown also in Fig. A.2.

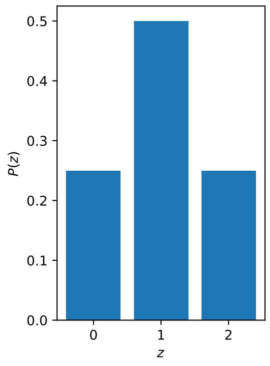


Figure A.2: A categorical distribution, $P(Z)$, where $z \in \{0, 1, 1\}$.

The conditional distribution

$$P(Y|X = x) \tag{A.13}$$

is interesting to us in machine learning, since we usually assume to have *observed* some instance x. From this we have the conditional pmf; i.e., $P(Y = y|X = x)$ for all $y \in \mathcal{Y}$, and for this x *in particular*.

Note that all these functions (queries[1])

$$P(Y = y|\boldsymbol{X} = \boldsymbol{x})$$
$$P(\boldsymbol{Y} = \boldsymbol{y})$$
$$P(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x})$$

return a single number (scalar), even though domains may be multi-dimensional (e.g., $\boldsymbol{y} \in \{0, 1\}^2$, and $\boldsymbol{x} \in \mathbb{R}^4$). Again, $P(\boldsymbol{Y} \mid \boldsymbol{x})$ denotes the distribution (describing $P(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x})$ *for all* possible $\boldsymbol{y}$).

Example A.4.3- Double coin flip – continued

An alternative way to represent the double-coin flip of Ex. A.4 (and Fig. A.2) considering instead a random vector instead: We flip two fair coins, Y_1 and Y_2, then (supposing a fair coin)

$$P(\boldsymbol{Y} = [\mathsf{Heads}, \mathsf{Heads}]) = 0.25$$

is the probability of getting Heads *on both* coins. Here, $\boldsymbol{y} \in \{\mathsf{Heads}, \mathsf{Tails}\}^2$.

[1] Again, possible shorthand: $P(\boldsymbol{y}|\boldsymbol{x}) \equiv P(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x}) \equiv \mathbb{P}(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x})$, etc.

In Example. (A.4), the outcome of both flips are independent of each other, hence the iid assumption (for $n = 2$):

$$P(\boldsymbol{y}) = \prod_{i=1}^{n} P(y_i) = 0.25$$

The joint probability

$$P(X, Y) = P(X|Y)P(Y) = P(Y|X)P(X) \tag{A.14}$$

is worth studying wrt the differences between discriminative and generative classifiers. In particular, note (from Eq. (A.14)) that:

$$\widehat{y} = \operatorname*{argmax}_y P(x|y)P(y) = \operatorname*{argmax}_y P(y|x)$$

(the fact that $P(x)$ is missing from the right side is *not* an error; it is because it is not a function of y – the variable over which the optimization is carried out).

Required for Topic 4 (Uncertainty and Bias-Variance Tradeoff) and onwards

When an instance lives in continuous space, e.g., $x \in \mathbb{R}$ or $\boldsymbol{x} \in \mathbb{R}^m$, we use a probability density function (pdf):

$$p(x)$$

which gives the *relative likelihood* at point x (relative to other parts in the domain). We use the pdf because $P(X = x) = 0$ for any given $x \in \mathbb{R}$, so this is not much use to us. Let's note the relationship

$$P(a < X \leq b) = \int_a^b p(x)\,\mathrm{d}x$$

As pmfs (in the discrete case), pdfs should also be non-negative everywhere and sum to 1: i.e., $\int_{-\infty}^{+\infty} p(x)\,\mathrm{d}x = 1$; but $p(x)$ itself may be more than 1 for some values of x.

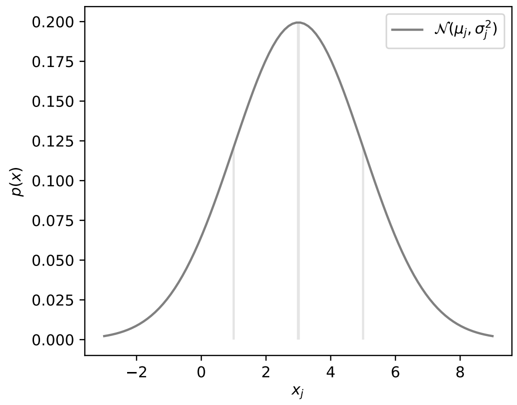


Figure A.3: The Gaussian distribution.

For example, (Fig. A.3 shows) the normal/Gaussian distribution, which we denote $P(X) = \mathcal{N}(\mu, \sigma^2)$. The pdf is denoted $p(x) = \mathcal{N}(x|\mu, \sigma)$. So $X \sim \mathcal{N}(\mu, \sigma^2)$ denotes that X is a Gaussian-distributed random variable. The random draw

$$x \sim \mathcal{N}(0, 1)$$

can be implemented in NUMPY as `x = np.random.randn() * 1 + 0`. It will produce a number; instantiation of the random variable X.

A.5 Expectations

The expected value of a variable X:

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \cdot P(X = x) \quad \triangleright \text{ when discrete, e.g., } \mathcal{X} = \{1, 2, \ldots, k\}$$
$$\mathbb{E}[X] = \int_{-\infty}^{+\infty} x \cdot p(x) \, \mathrm{d}x \quad \triangleright \text{ when continuous, e.g., } \mathcal{X} = \mathbb{R}$$

For example, for a fair 6-sided die ($\mathcal{X} = \{1, 2, \ldots, 6\}$): $\mathbb{E}[X] = 3.5$. For the standard Gaussian distribution (cf., Fig. A.3), $\mathbb{E}[X] = \mu = 0$.

The expectation of a function, e.g., $f(X) \in \mathbb{R}$:

$$\mathbb{E}[f(X)] = \int_{-\infty}^{+\infty} f(x) \cdot p(x) \, \mathrm{d}x$$

In all cases so far, it is implicit that the expectation is wrt $X \sim P(X)$.

The conditional expectation of a function ℓ taking variable Y *given* $\boldsymbol{x}$ (where $\boldsymbol{x}$ the realisation of random variable $\boldsymbol{X}$) is

$$\mathbb{E}_{Y \sim P(Y|\boldsymbol{x})}[\ell(\widehat{y}, Y)] = \sum_{y \in \mathcal{Y}} \ell(\widehat{y}, y) \cdot P(Y = y \mid \boldsymbol{X} = \boldsymbol{x})$$

We have explicitly denoted that the expectation is wrt Y only ($\boldsymbol{x}$ and $\widehat{y}$ are given here, considered fixed numbers/vectors). We suppose that $\mathcal{Y}$ is a set of discrete numbers (otherwise the sum would be an integral). In shorthand, we might denote this as $\mathbb{E}[Y|x]$.

Note how the function ℓ inherits the uncertainty of Y.

The expectation of a variable refers to the mean of its distribution $\mathbb{E}[X] = \mu$. We can also consider expected variance; variance is just a function of a variable. Variance represents uncertainty (think about how this makes sense). In shorthand, We can denote $\mathbb{V}[X]$ as the expected variance of the distribution of X:

$$\mathbb{V}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

A.6 Element-wise multiplication (in the context of back propagation)

Required for Topic 6 (back propagation)

Element-wise multiplication, also known as the Hadamard product, is denoted

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B}$$

where $c_{j,k} = a_{j,k} \cdot b_{j,k}$.

Note that it is commutative (unlike the standard matrix product $\mathbf{AB}$). It corresponds to matrix multiplication (matrix product) with a diagonal matrix.

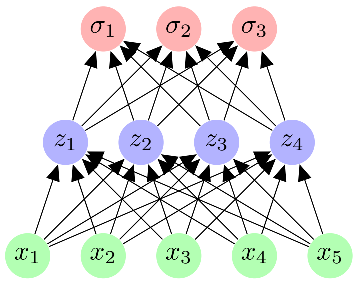
The following is an example of how it arises in back propagation. First, let's recall Eq. (6.2):

$$\begin{aligned} E(\mathbf{y}, \boldsymbol{\sigma}) &= \|\mathbf{y} - \boldsymbol{\sigma}\|_2^2 \\ &= (y_1 - \sigma_1)^2 + (y_2 - \sigma_2)^2 + (y_3 - \sigma_3)^2 \\ &= e_1^2 + e_2^2 + e_3^2 \end{aligned}$$

We will take the derivative. Recall (chain rule, where $\mathbf{a}$ is the activation; i.e., $\boldsymbol{\sigma} = \sigma(\mathbf{a})$):

$$\nabla_{\mathbf{a}} E = \nabla_{\boldsymbol{\sigma}} E \nabla_{\mathbf{a}} \boldsymbol{\sigma}$$

Here we reproduce the network of Fig. 5.2:



Also recall (from Eq. (A.7)) that:

$$\nabla_{\boldsymbol{\sigma}} E = \left[\frac{\partial E}{\partial \sigma_1}, \frac{\partial E}{\partial \sigma_2}, \frac{\partial E}{\partial \sigma_3}\right] \quad \text{and} \quad \nabla_{\mathbf{a}} \boldsymbol{\sigma} = \begin{bmatrix} \frac{\partial \sigma_1}{\partial a_1} & \frac{\partial \sigma_2}{\partial a_1} & \frac{\partial \sigma_3}{\partial a_1} \\ \frac{\partial \sigma_1}{\partial a_2} & \frac{\partial \sigma_2}{\partial a_2} & \frac{\partial \sigma_3}{\partial a_2} \\ \frac{\partial \sigma_1}{\partial a_3} & \frac{\partial \sigma_2}{\partial a_3} & \frac{\partial \sigma_3}{\partial a_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial \sigma_1}{\partial a_1} & 0 & 0 \\ 0 & \frac{\partial \sigma_2}{\partial a_2} & 0 \\ 0 & 0 & \frac{\partial \sigma_3}{\partial a_3} \end{bmatrix}$$

(because $\frac{\partial \sigma_2}{\partial a_1} = 0$ – the activation a_1 does not affect σ_2 in any way – etc).

So, finally, we see that (continuing the example of the sigmoid activation function):

$$\begin{aligned} \nabla_{\mathbf{a}} E &= \nabla_{\boldsymbol{\sigma}} E \nabla_{\mathbf{a}} \boldsymbol{\sigma} \\ &= \nabla_{\boldsymbol{\sigma}} E \odot (\boldsymbol{\sigma} \odot (\mathbf{1} - \boldsymbol{\sigma})) \\ &= \nabla_{\boldsymbol{\sigma}} E \odot \mathbf{f}' \end{aligned}$$

where $\mathbf{f}' = \mathsf{diag}(\nabla_{\mathbf{a}} \boldsymbol{\sigma})$.

Another way to see this is to consider the computational graphs (a layer with 2 outputs):

$$\begin{tikzcd}
a \arrow[d] & & a_1 \arrow[d] & & a_2 \arrow[d] \\
\sigma \arrow[d] & & \sigma_1 \arrow[dr] & & \sigma_2 \arrow[dl] \\
E & & & E &
\end{tikzcd}$$

The one on the left could be considered operations on matrices, with the default operation being matrix multiplication. In the graph on the right it is explicit the fact that only a_k is required to produce σ_k.

$$\begin{aligned}
\nabla_{\mathbf{a}} E &= \left[\frac{\partial E}{\partial a_1}, \frac{\partial E}{\partial a_2}\right] \\
&= \left[\frac{\partial E}{\partial \sigma_1}\frac{\partial \sigma_1}{\partial a_1}, \frac{\partial E}{\partial \sigma_2}\frac{\partial \sigma_2}{\partial a_2}\right] \\
&= \mathbf{e} \odot \boldsymbol{\sigma}'
\end{aligned}$$

and if there was a w and x in common to each a_k:

$$\begin{aligned}
\nabla_w E &= \left[\frac{\partial E}{\partial \sigma_1}\frac{\partial \sigma_1}{\partial a_1}\frac{\partial a_1}{\partial w} + \frac{\partial E}{\partial \sigma_2}\frac{\partial \sigma_2}{\partial a_2}\frac{\partial a_2}{\partial w}\right] \\
\nabla_w E &= \left[\frac{\partial E}{\partial \sigma_1}\frac{\partial \sigma_1}{\partial a_1}x + \frac{\partial E}{\partial \sigma_2}\frac{\partial \sigma_2}{\partial a_2}x\right] \\
&= (\mathbf{e} \odot \boldsymbol{\sigma}')x
\end{aligned}$$

Appendix B

Kernel Methods

Note to CSE204 2023–2024 students: You are not required to study the material in this chapter; it is here for historical reasons. Given one more week of the course, it would have been about this topic/chapter. The first paragraph in the following (written in previous years) still holds!

Despite the soaring popularity of deep neural network architectures in recent decades, kernel methods are still relevant to a plethora of modern application domains. And an understanding of the mathematical concepts behind such methods are of invaluable importance across the many modern domains of artificial intelligence. Although we have been hearing about the 'Big Data' era for some years now, and have certainly seen mammoth amounts of data being turned into deep models (particularly in the areas of computer vision and natural language), many modern data science problems can be tackled with kernel methods, and one can easily argue they are being overlooked.

Kernel methods give us the non-linearity of a feature-map (seen earlier, e.g., in the context of polynomial regression) but with the advantage of never having to explicitly calculate that feature map. Rather we only need to define a similarity function, known as a kernel. Kernel methods, which are kernelized versions of methods we have already seen (such as regression, PCA, k-means), can be applied to data with complex input spaces such as sequences, graphs, and text. They are also very well mathematically supported.

Main Concepts

In this topic, you should aim to develop a notion of

- Implicit vs explicit feature spaces
- The kernel trick
- The position held by kernel methods in modern machine learning

You should know the simple recipe of how to kernelize a method. And you should understand the method of Spectral Clustering, including how to implement it, and how (and when is best, vs e.g., standard k-means) to apply it.

B.1 Feature Maps and Feature Space; Explicit vs Implicit Mapping

Recall (e.g., from Section 2.3 in the context of polynomial regression) that we can use a feature map

$$\phi : \mathcal{X} \mapsto \mathcal{H} \tag{B.1}$$

to map any input $\mathbf{x}$ from input-data space $\mathcal{X}$ into feature space $\mathcal{H}$, hence the corresponding point in feature space is $\boldsymbol{\phi} \in \mathcal{H}$, obtained via $\boldsymbol{\phi} = \phi(\mathbf{x})$.

The family of methods known as kernel methods fit a linear decision surface (decision boundary in the case of classification), i.e., a hyperplane in $\mathcal{H}$; noting that the corresponding surface in the original space $\mathcal{X}$ may be non-linear.

This is what we did in, e.g., polynomial regression. However this is the major difference: kernel methods deal with an implicit feature space; and as doing such these methods never need to explicitly compute the map denoted in Eq. (B.1). This has huge practical implications.

B.2 Kernels and the Inner Product

A kernel is simply a comparison between two instances,

$$\boxed{\begin{aligned} K : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R} \\ K(\mathbf{x}, \tilde{\mathbf{x}}) \in \mathbb{R} \end{aligned}}$$

providing a scalar output which is a measure of comparison between the two inputs.

To explore kernels and kernel methods, we should first be convinced that the inner product (or dot product[1]) $\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle$ can provide us a useful comparison between $\mathbf{x}$ and $\tilde{\mathbf{x}}$, i.e.,

$$K(\mathbf{x}, \tilde{\mathbf{x}}) = \langle \mathbf{x}, \tilde{\mathbf{x}} \rangle$$

We note, for example, that it is symmetric, $\langle \tilde{\mathbf{x}}, \mathbf{x} \rangle = \langle \mathbf{x}, \tilde{\mathbf{x}} \rangle$. And there is a relationship with Euclidean distance; if two vectors $\mathbf{x}$ and $\tilde{\mathbf{x}}$ are both normalized (such that $\langle \mathbf{x}, \mathbf{x} \rangle = 1$ and $\langle \tilde{\mathbf{x}}, \tilde{\mathbf{x}} \rangle = 1$), then we find that

$$\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle = 1 - \frac{1}{2} \|\mathbf{x} - \tilde{\mathbf{x}}\|_2^2$$

We also know that the inner product of these vectors is the cosine of the angle θ between $\mathbf{x}$ and $\tilde{\mathbf{x}}$:

$$\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle = \cos \theta$$

The sign of the inner product tells us if two vectors are pointing in the same direction ($\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle > 0$) or not ($\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle < 0$). And the vectors are orthogonal to each other when $\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle = 0$.

Note also the connection to covariance: large numbers multiplied together give a large number. When normalization is done appropriately:

$$\langle \mathbf{x}, \tilde{\mathbf{x}} \rangle = \mathsf{cov}(\mathbf{x}, \tilde{\mathbf{x}})$$

[1]We can also talk of the *dot product* here, so-called because of the notation $\mathbf{x} \cdot \tilde{\mathbf{x}}$, or a *scalar product*, because it produces a scalar output $\in \mathbb{R}$. We can also write $\mathbf{x}^\top \tilde{\mathbf{x}}$ under the implication of column vectors. However, the *inner product* is a more general concept and more appropriate when dealing with kernels; as they can be defined on infinite-dimensional spaces

And indeed, another name for the kernel matrix is the covariance matrix (but note: variance among instances, not among features):

$$\mathbf{K} = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) & \dots & K(\mathbf{x}_1, \mathbf{x}_n) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) & \dots & K(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & & & \\ K(\mathbf{x}_n, \mathbf{x}_1) & K(\mathbf{x}_n, \mathbf{x}_2) & \dots & K(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}. \tag{B.2}$$

Consider the utility of the inner product in classification illustrated in Fig. B.1.

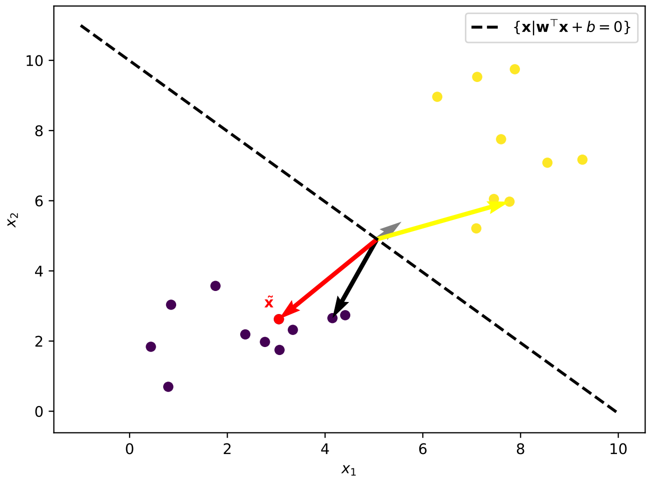


Figure B.1: A classification problem separating two classes $\mathcal{Y} = \{-1, +1\}$ of data instances; where black and yellow points form the training dataset (class -1 and $+1$ respectively), and $\tilde{\mathbf{x}}$ is a test point. A line has been fitted as a decision boundary, defined by $\mathbf{w}$ (a normalized version of this vector is shown in gray). There are numerous obvious uses of the inner product here: $\langle \tilde{\mathbf{x}}, \mathbf{x} \rangle > 0$ tells us that $\tilde{\mathbf{x}}$ is pointing in the same direction as $\mathbf{x}$ (and therefore might perhaps take the same class label). And we can compare to $\mathbf{w}$ in the same way to find out which side of the decision boundary the test point is on, leading to a classification $\widehat{y} = \mathsf{sign}\left(\langle \tilde{\mathbf{x}}, \mathbf{w} \rangle\right)$.

We additionally note that we can easily generalize to $d > 2$. Even if we can't visualize in many dimensions in $\mathcal{X}$, we can nevertheless visualize vectors plotted j vs x_j. Consider Fig. B.2

Are you now convinced that the inner product is a useful comparison? Then we already have our first kernel, the linear kernel:

$$K(\mathbf{x}, \tilde{\mathbf{x}}) = \langle \mathbf{x}, \tilde{\mathbf{x}} \rangle \tag{B.3}$$

We can also use many functions K to get many kinds of kernels: polynomial kernels, Gaussian kernels, sequence and string kernels, graph kernels, etc. In many cases, the nature of the kernel depends on the nature of $\mathcal{X}$ and, like explicit feature maps, we can choose random kernels or they can be designed by an expert.

What happened to our feature vector $\phi(\mathbf{x}) \in \mathcal{H}$ in all this? That's exactly the point. Whatever function we choose for K, we can imagine that in feature space

$$\boxed{K(\mathbf{x}, \tilde{\mathbf{x}}) = \langle \phi(\mathbf{x}), \phi(\tilde{\mathbf{x}}) \rangle \qquad \text{(B.4)}}$$

x_1	x_2	x_3	y
1	10	2	-1
4	10	3	-1
4	0	3	+1

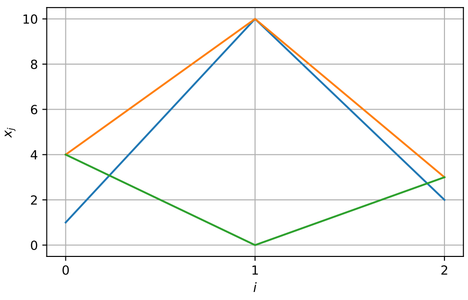


Figure B.2: The dot product between each pair of $\mathbf{x}$ reveals a pattern among the two visually similar instances (namely, $\mathbf{x}_1 = [1, 10, 2]$ and $\mathbf{x}_2 = [4, 10, 3]$); $\langle \mathbf{x}_1, \mathbf{x}_2 \rangle = 110$ whereas $\langle \mathbf{x}_1, \mathbf{x}_3 \rangle = 10$: bigger number = greater similarity. N.B. Centering the data is important for these comparisons to work properly.

but we just implement $K(\cdot, \cdot)$ and never need to implement $\phi(\cdot)$ or visualize any $\boldsymbol{\phi}$ or do any computations explicitly in its space $\mathcal{H}$.

So I can choose any function $K(\mathbf{x}, \tilde{\mathbf{x}})$ that I want? Well, although we never need to go into feature space $\mathcal{H}$ ourselves we do want to be assured that there is some inner product representation in that space[2], i.e., that $\langle \phi(\mathbf{x}), \phi(\tilde{\mathbf{x}}) \rangle$ from Eq. (B.4) *is valid* given our implementation of $K(\cdot, \cdot)$. To obtain these assurances, then kernels should be symmetric positive-definite (although some other kernels may work in practice). A symmetric function K is positive definite if:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) \geq 0$$

for any $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n]$, $\alpha_i \in \mathbb{R}$.

We can see that this is the case for the inner product, Eq. (B.3). Polynomial and Gaussian (RBF) kernels are also positive definite. Recall: the symmetry simply implies that $K(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_j, \mathbf{x}_i)$. And recall also that this also applies to covariance functions, which can thus be used as kernels.

B.3 Kernel Trick

We use the kernel trick to kernelize a method. To do this, one chooses a machine learning method and completes the following procedure:

1. Express the method in terms of inner products (every $\mathbf{x}$ should appear inside some $\langle \mathbf{x}, \mathbf{x}_i \rangle$ or $\langle \mathbf{x}_i, \mathbf{x} \rangle$).

2. Replace every $\langle \mathbf{x}_i, \mathbf{x} \rangle$ with $K(\mathbf{x}_i, \mathbf{x})$.

3. Implement $K(\cdot, \cdot)$.

[2]As an aside: that is why we have used notation $\mathcal{H}$ instead of $\mathcal{F}$ for the feature space so far; $\mathcal{H}$ is a Hilbert space; which is a kind of feature space which can generalize to infinite dimensions but is equipped with an inner product

B.4 Exemplified: Kernel Ridge Regression

We exemplify on ridge regression. The estimator for this method is derived (you may recall from earlier topics on regression) by minimizing the error function as follows:

$$\begin{aligned}
E(\mathbf{w}) &= (\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda\mathbf{w}^\top\mathbf{w} \\
&= \dots \quad \triangleright \text{ (take the derivative, set to 0, solve for } \mathbf{w}) \\
\widehat{\mathbf{w}} &= (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^\top\mathbf{y} \quad \triangleright \text{ where } \mathbf{I}_d \text{ is a } d\text{-dimensional identity matrix} && \text{(B.5)} \\
&= \mathbf{X}^\top \underbrace{(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I}_n)^{-1}\mathbf{y}}_{\boldsymbol{\alpha}} \quad \triangleright \text{ N.B. } \mathbf{I}_n \text{ is an } n\text{-dimensionaly identity matrix} && \text{(B.6)} \\
&= \mathbf{X}^\top\boldsymbol{\alpha} \quad \triangleright \text{ where } \boldsymbol{\alpha} \in \mathbb{R}^{n\times 1} && \text{(B.7)} \\
&= \sum_{i=1}^{n} \alpha_i \mathbf{x}_i \quad \triangleright \text{ (as a summation)}
\end{aligned}$$

The difference from Eq. (B.5) to Eq. (B.6) is a subtle but important one. The difference in terms of notation (since these two equations are equal to each other); notably the size of the matrix. Note that $\mathbf{S} = \mathbf{X}^\top\mathbf{X} \in \mathbb{R}^{d\times d}$ (the $\mathbf{S}$ is to recall, e.g., Eq. (9.4)) but $\mathbf{K} = \mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{n\times n}$ (specifically with connection to kernel matrix Eq. (B.2) – in the case of the linear kernel). Recall that

Whereas $\widehat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ lies in the column space of $\mathbf{X}$; the vector $\widehat{\mathbf{w}} = \mathbf{X}^\top\boldsymbol{\alpha}$ lies in the row space of $\mathbf{X}$;

$$\widehat{\mathbf{w}} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i = \mathbf{X}^\top\boldsymbol{\alpha}, \qquad \widehat{y} = \sum_{j=1}^{d} \widehat{w}_j x_j = \mathbf{x}\mathbf{w}$$

Recall: Our goal is to express the method in terms of inner products between $\mathbf{x}$s. To do so, we substitute $\widehat{\mathbf{w}}$ (from Eq. (B.7)) into $\widehat{y} = \mathbf{x}\widehat{\mathbf{w}}$, obtaining:

$$\begin{aligned}
\widehat{y} &= \mathbf{x}(\mathbf{X}^\top\boldsymbol{\alpha}) \\
&= (\boldsymbol{\alpha}^\top\mathbf{X})\mathbf{x}^\top \\
&= (((\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I}_n)^{-1}\mathbf{y})^\top\mathbf{X})\mathbf{x}^\top \\
&= ((\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I}_n)^{-1}\mathbf{y})^\top\mathbf{X}\mathbf{x}^\top \\
&= ((\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I}_n)^{-1}\mathbf{y})^\top(\mathbf{x}\mathbf{X}^\top)^\top \\
&= ((\mathbf{K} + \lambda\mathbf{I}_n)^{-1}\mathbf{y})^\top\boldsymbol{\kappa}^\top
\end{aligned}$$

where $\mathbf{K}$ is the $n \times n$ kernel matrix, of entries $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ and $\boldsymbol{\kappa} = [K(\mathbf{x}_1, \tilde{\mathbf{x}}), \dots, K(\mathbf{x}_n, \tilde{\mathbf{x}})]$. Done!

Now we choose some value for the ridge (say, $\lambda = 0.1$) and plug in a Gaussian kernel for K_{ij}. When we apply it to the Tadpole problem[3] in Fig. B.3.

Conclusion: Since ridge regression is a linear model, we know the regression decision survace is in fact a *line* (in *feature space*). The novelty is: we didn't have to go into that space; we didn't compute $\phi(\mathbf{x})$ at any point; no tadpoles were harmed or manipulated in any way – we just simply measured the similarity of one to another using some similarity function K.

[3]If you didn't attend the lecture: each point in the figure is a tadpole

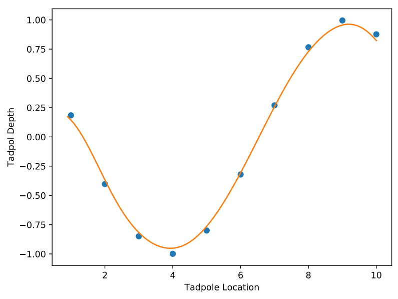


Figure B.3: The orange curve is in fact a line (in feature space) .

Bibliography

[1] The gender pay gap situation in the EU, Accessed 2024. https://commission.europa.eu/strategy-and-policy/policies/justice-and-fundamental-rights/gender-equality/equal-pay/gender-pay-gap-situation-eu_en.

[2] Christopher M Bishop and Hugh Bishop. *Deep Learning, Foundations and Concepts*. Springer, 2023.

[3] Anne Case and Christina Paxson. Stature and status: Height, ability, and labor market outcomes. *Journal of political Economy*, 116(3):499–532, 2008.

[4] Gerd Gigerenzer and Henry Brighton. Homo heuristicus: Why biased minds make better inferences. *Topics in cognitive science*, 1(1):107–143, 2009.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. https://www.deeplearningbook.org.

[6] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[7] David H Hubel and Torsten N Wiesel. Cortical and callosal connections concerned with the vertical meridian of visual fields in the cat. *Journal of neurophysiology*, 30(6):1561–1573, 1967.

[8] Timothy A Judge and Daniel M Cable. The effect of physical height on workplace success and income: preliminary test of a theoretical model. *Journal of Applied Psychology*, 89(3):428, 2004.

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[10] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep learning with PyTorch*. Manning Publications, 2020.

[11] Raphael Vallat. A simple and efficient sleep spindles detector, September 2018. https://raphaelvallat.com/spindles.html.

[12] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007. https://people.csail.mit.edu/dsontag/courses/ml14/notes/Luxburg07_tutorial_spectral_clustering.pdf.

[13] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.