MA50290: Applied Machine Learning

Dr James Foster
Department of Mathematical Sciences
University of Bath

(based on previous notes by Dr Sergey Dolgov)

Contents

1 Commonalities of machine learning **3**
1.1 What is "Machine Learning" . . . 3
1.2 Problem and model selection . . . 5
1.2.1 Supervised or unsupervised learning . . . 5
1.2.2 Example: polynomial regression . . . 6
1.2.3 Underfitting and overfitting . . . 7
1.2.4 Training data and test data splitting . . . 8
1.2.5 Empirical risk minimisation . . . 8
1.3 Introduction to statistical learning theory . . . 9
1.3.1 Data distribution and expected risk . . . 11
1.3.2 Cross validation: estimation of the prediction error and model selection . 12
1.3.3 Bias-Variance tradeoff . . . 12
1.3.4 The No-Free-Lunch Theorem . . . 14
1.3.5 Fundamental theorem of statistical learning . . . 15

2 Data preparation and retrieval **17**
2.1 Numerical and non-numerical data . . . 17
2.1.1 Explicit numerical data in vectors and matrices . . . 17
2.1.2 One-dimensional data: time series and discretised functions . . . 18
2.1.3 Images: 2- and 3-dimensional data . . . 19
2.1.4 Non-numerical data: text and categories . . . 20
2.2 Information retrieval from text data (non-examinable) . . . 21
2.2.1 Vector space model of text . . . 21
2.2.2 Inverse document frequency weighting . . . 23
2.3 Metrics and scores on data (examinable) . . . 24
2.3.1 Vector distance . . . 24
2.3.2 Angle distance and cosine similarity score . . . 25
2.3.3 Cosine similarity scoring of documents (non-examinable) . . . 25

3 Unsupervised learning **28**
3.1 Clustering of data . . . 28
3.1.1 Clustering model . . . 29
3.1.2 Linkage clustering algorithms . . . 29

3.1.3 K-means loss and K-means algorithm . . . 31
3.1.4 Choosing the number of clusters . . . 33
3.1.5 Silhouette Coefficient: a score of clustering outliers . . . 34
3.1.6 Rand index: a similarity score of two clusterings . . . 35
3.2 Principal Component Analysis for dimensionality reduction . . . 36
3.3 Example: spectromicroscopy . . . 39

4 Supervised learning **42**
4.1 Simple prediction models . . . 42
4.1.1 Linear functions as prediction rules . . . 42
4.1.2 Linear regression . . . 43
4.1.3 Linear regression for Polynomial features . . . 43
4.1.4 Halfspaces binary classifier . . . 44
4.1.5 Logistic regression and maximum likelihood estimators . . . 45
4.1.6 Naive Bayes . . . 48
4.1.7 Multiclass classification . . . 49
4.2 Optimization algorithms . . . 49
4.2.1 First-order methods: gradient descent (GD) . . . 50
4.2.2 Convergence of gradient descent . . . 51
4.2.3 GD for empirical risk minimisation and linear regression (non-examinable) 58
4.2.4 Stochastic gradient descent (SGD) . . . 60
4.2.5 Convergence of SGD . . . 60
4.2.6 SGD for empirical risk minimisation . . . 63
4.2.7 Early stopping of GD and SGD based on test loss . . . 64
4.2.8 Variance reduction methods: mini-batching and stochastic average gradient 64
4.2.9 Derivative-free methods. Perceptron algorithm for halfspaces . . . 65
4.2.10 Second-order methods: Newton's method (non-examinable) . . . 68
4.3 Non-parametric prediction methods . . . 71
4.3.1 Decision trees . . . 71
4.3.2 K-nearest neighbours . . . 74

Outline of the course

This unit is aimed to introduce you to core mathematics and algorithms of machine learning. The mathematics part of the unit will be assessed in an exam, which contributes 60% of your total mark for this unit. The algorithmic part will be assessed in a coursework, which contributes 40% of your mark.

In most weeks, we will have two lectures or a lecture and a computer lab. During the computer labs, you will be able to ask me questions about problem sheets or more generally about the course. You can also leave questions for me on the Padlet forum, which can be found at `https://padlet.com/jmf68/ma50290`.

Problem sheets will contain both theoretical questions on mathematics of machine learning, and programming exercises in Python. Although the problem sheet marks do not contribute to the final grade directly, doing them is an **essential** part of the course. The remaining material, coursework and exam will be designed assuming that you have done the problem sheets.

We will use the Noteable Jupyter server `https://moodle.bath.ac.uk/mod/lti/view.php?id=1343867` as the default Python environment. Alternatively, you may install any Python environment (such as Anaconda or Visual Studio Code) on your laptop or use Google Colab. However, please note that we might be unable to support every personal Python environment.

If you are new to Python programming, then you may have to teach yourself the basics. Fortunately, there are plenty of online resources and I hope the "Reading on Python" section on the Moodle page will help. You may also ask me for further advice.

If you do not have a mathematical background, you may also find it helpful to look through the "Mathematics for Machine Learning" textbook – which can be found in the "Reading on mathematics of machine learning" section on Moodle.

1 Commonalities of machine learning

1.1 What is "Machine Learning"

There is no formal definition, but generally speaking, *Machine Learning covers anything related to design, application and analysis of algorithms mimicking intelligent behaviour.*

For example, an algorithm may need to be able to recognise objects on a video stream from cameras, label them, and predict their movement. A video can be seen as a time series of matrices of colour intensity values, each of which is measured with some noise due to changing light conditions and electrical noise in the camera. Thus, to describe this object recognition task mathematically, we need to draw upon

- Linear Algebra for operations with matrices and vectors, and
- Statistics to deal with random variables defining the noise (or data altogether).

Moreover, similar to human learning, machine learning usually involves improvement of the behaviour as more and more tasks are completed. To quantify "improvement" for a computer, we need to identify a suitable **reward** function, which takes the result of the algorithm as input (for example, a vector of labels), and produces a number (the value of the reward) as output. *Higher* values of the reward will correspond to a "better" behaviour of the algorithm, in whatever desired sense. Alternatively, we can define a **loss** function (such as the prediction error), *lower* values of which correspond to a better behaviour.

> Most machine learning problems can be formalised as **optimisation** of a certain function (reward or loss).

Suitable methods from numerical analysis (such as the gradient descent method) can be used to solve the resulting optimisation problem. However, distinguishing features of machine learning compared to traditional statistics or numerical analysis are **prediction** and **large data**.

A common task in statistics is *estimation* of the value of an unknown quantity from a *given* data. Physicists, for example, estimated the speed of light from a relatively *small* amount of interferometer measurements using techniques from statistics. In contrast, the machine learning task of object recognition mentioned above concerns *prediction* of object characteristics from *large* volumes of future videos rather than estimation of statistics of recordings taken in the past. In addition, *estimation problems* are often focused on the accuracy of the result, whereas *prediction problems* focus on the ability and stability of algorithms to predict new observations.

The success of tackling large data is twofold. The first component is new mathematical techniques (such as the stochastic gradient descent method) which can analyse large datasets faster. However, the so-called "big data" revolution hinges also on the exponential increase in computing power and memory, commonly known as the Moore's law. Modern computers enabled calculations with previously unthinkable prediction models and datasets of billions of numbers. Although the Moore's law on a *single* chip has now reached its physical limits, *parallel* computations using graphical cards (Graphical Processing Units, or GPUs) and more dedicated hardware (such as Google's Tensor Processing Units, or TPUs) allow us to expand the scale of machine learning problems ever further.

Machine learning can be roughly divided into three categories:

- **supervised learning**,
- **unsupervised learning**, and
- **reinforcement learning**.

Definition 1.1. ***Supervised learning*** *concerns drawing* **prediction** *models from* **data** *previously* ***labelled*** *by humans.*

A famous example is a collection of pictures labelled as containing a cat or not containing a cat. Common supervised learning tasks include classification and regression.

Definition 1.2. ***Unsupervised learning*** *is concerned with finding patterns and* ***structure in unlabelled*** *data.*

Typical unsupervised learning applications include clustering, dimensionality reduction, and generative modelling – for example, clustering of different materials in a specimen.

Finally, in reinforcement learning, an agent learns by interacting with an environment and changing its behaviour to maximize its reward. For example, a robot can be trained to navigate in a complex environment by assigning a high reward to actions that help the robot reach a desired destination.

While informative, the distinction between these three categories of machine learning is sometimes fuzzy and fluid, and many applications often combine them in novel and interesting ways. For example, the success of Google DeepMind in developing algorithms that excel at tasks such as playing Go and video games employ deep reinforcement learning – which combine reinforcement learning with supervised learning methods based on deep neural networks.

In our course, we limit our focus to supervised and unsupervised learning as more simple tasks. However, before we can perform any learning, we need to understand and quantify the corresponding data. Therefore, we will focus on three main topics:

- Data retrieval
- Unsupervised learning, and
- Supervised learning,

embodied into simplest regression and classification problems.

1.2 Problem and model selection

Given a task described in lay terms (e.g. "label images containing a cat"), one needs to select the correct machine learning treatment.

1.2.1 Supervised or unsupervised learning

The first step is to understand whether the dataset is *labelled* or *unlabelled.*

Definition 1.3. *A* ***unlabelled dataset*** *is a set* $\mathbf{D} = \mathbf{X} = \{\mathbf{x}_i\}_{i=1}^N$ *of some data objects* $\mathbf{x}_i \in \mathcal{X}$*, where* $\mathcal{X}$ *is a (potentially larger) set of admissible data.*

For example, $\mathbf{x}_i$ can be a vector of colour values of n pixels in the i-th image, in which case $\mathcal{X} = \mathbb{R}^n$. However, we can also *label* each $\mathbf{x}_i$ with an additional datum y_i. For example, a label y_i can be 1 if the image $\mathbf{x}_i$ contains a cat, and -1, otherwise.

Definition 1.4. *A* ***labelled dataset*** *is a set* $\mathbf{D} = (\mathbf{X}, \mathbf{y})$ *which contains both data objects* $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^N$*, and a vector of labels* $\mathbf{y} = \{y_i\}_{i=1}^N$*,* $y_i \in \mathcal{Y}$*, which* ***depend*** *on* $\mathbf{x}_i$*, that is, for each* $\mathbf{x}_i \in \mathbf{X}$ *there corresponds a* ***unique*** *label* y_i*,* $i = 1, \dots, N$*. The data objects* $\mathbf{x}_i$ *are also called* ***domain****, or* ***input*** *points, and the labels* y_i *are called* ***output*** *points.*

For computing convenience, labels are usually real numbers, $y_i \in \mathbb{R}$, although one can consider any set $\mathcal{Y}$ that labels can belong to. In the example above, the initial labels can be text, $\mathcal{Y} = \{$"cat", "not cat"$\}$, but one can formulate and solve the image classification problem using $\mathcal{Y} = \{1, -1\}$ instead.

Labelled data normally imply a **supervised** learning problem: given the known dataset $\mathbf{D} = (\mathbf{X}, \mathbf{y})$, and a *new* data point $\mathbf{x}_* \notin \mathbf{X}$, *predict* the label $y_* \in \mathcal{Y}$ corresponding to $\mathbf{x}_*$ following a rule defined implicitly by the dataset $\mathbf{D}$. Normally, we need to predict labels for many different $\mathbf{x}_*$, so instead of finding a single prediction label y_*, we look for the entire **model** of the (unknown) rule of assigning the labels y_i to $\mathbf{x}_i$ in $\mathbf{X}$, which is expected to hold also for $\mathbf{x}_* \in \mathcal{X} \backslash \mathbf{X}$. Mathematically, this can be formalised as finding a **function**. However, computers cannot just find an abstract function. In computational algorithms, we need to **parametrise** a function with additional numerical inputs, and search for optimal values of these parameters.

Definition 1.5. *A* ***prediction rule*** *(or* ***prediction model****) is a function* $h_{\boldsymbol{\theta}}(\mathbf{x}) : \mathcal{X} \to \mathcal{Y}$ *that for each* $\mathbf{x} \in \mathcal{X}$ *predicts a label* $y \in \mathcal{Y}$*, for any admissible vector of* ***tuneable parameters*** $\boldsymbol{\theta}$*.*

Unlabelled data can be used in a simple retrieval, comparison and metric problem, or in an unsupervised learning problem. Like labels, domain points $\mathbf{x}$ do not have to consist of any numerical data. These can be natural words, for example, $\mathbf{x} = ($"machine", "learning", "is", "cool"$)$. The domain of data can also be **categorical**, that is, data points take values only in an abstract fixed set, for example, geometrical shapes, $\mathbf{x} \in \{\text{triangle}, \text{circle}, \text{rectangle}\}$. However, computers can only operate with numbers eventually. Therefore, most (especially non-numerical) data requires the selection of a **distance** function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_+$ that is symmetric, satisfies $d(\mathbf{x}, \mathbf{x}) = 0$ for all $\mathbf{x} \in \mathcal{X}$, and often also satisfies the triangle inequality.

Remark 1.6. *In principle, any categorical data can be turned into numerical data by replacing each data point with its index in the set of categories. In the previous example of geometrical shapes, this would correspond to replacing each* $\mathbf{x} =$ *triangle with* $x = 0$*,* $\mathbf{x} =$ *circle with* $x = 1$*, and so on. However, a natural distance function (such as* $|x - y|$*) may lead to a poor accuracy of the prediction rule. Therefore, the freedom of choosing the distance function is essential.*

Unsupervised learning problems often concern finding an **output**, or **feature** function $\psi : \mathbf{X} \to \mathbf{Z}$. In general $\mathbf{Z}$ can be any other (numerical or non-numerical) set. Often $\mathbf{Z}$ is numerical though, such as a distance from some desired data point $\mathbf{x}_*$, or an index of the cluster the given point $\mathbf{x}$ belongs to. The output function $\psi(\mathbf{x})$ can also be called prediction rule. However, in contrast to supervised learning problems, no known samples of $\psi(\mathbf{x})$ are available, and one needs to determine the output function using the unlabelled dataset $\mathbf{X}$ only.

After the type of the machine learning problem is defined, the next crucial step is the selection of the *class* of prediction rules $h_{\boldsymbol{\theta}}(\mathbf{x})$ or $\psi(\mathbf{x})$.

1.2.2 Example: polynomial regression

Let us start with a simple supervised learning example to illustrate the difference between selecting the **class** of functions $h_{\boldsymbol{\theta}}(\mathbf{x})$, and **parameters** $\boldsymbol{\theta}$ within a particular class. Suppose we want to predict the temperature in Oxford in the future. A labelled dataset from the Met Office contains previous time points in months $\mathbf{X} = \{1, \ldots, 16\}$, where 1 corresponds to January 2022, and labels $\mathbf{y}$ denoting monthly average temperatures from the Oxford meteorological station, see Figure 1. In total there is $m = 16$ data points. Since the temperature varies smoothly, it

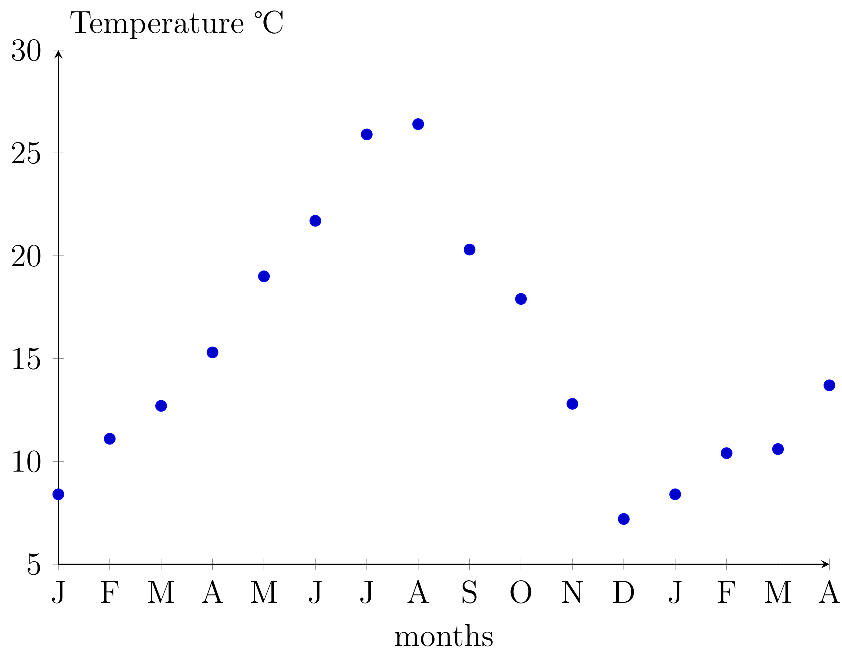


Figure 1: Monthly average temperature in Oxford since Jan 2022

looks reasonable to approximate it with a polynomial. Therefore, we take

$$h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \cdots + \theta_n x^n = \sum_{j=0}^{n} \theta_j x^j,$$

where $\boldsymbol{\theta} = (\theta_0, \ldots, \theta_n) \in \mathbb{R}^{n+1}$, $\theta_n \neq 0$, is a tuneable vector of coefficients and $x \in \mathbb{R}$ is the time in months. In the terms defined above, we select the **polynomial class** of models, and a *subclass* of polynomials of degree n. Now, we can find the **parameters** $\boldsymbol{\theta}$ such that the discrepancy between the actual temperature and that predicted by $h_{\boldsymbol{\theta}}(x)$ is minimal in the

months with known temperature. Thus, we can aim to minimize a *sum of squares* loss function,

$$L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(x_i) - y_i\right)^2, \qquad x_i \in \mathbf{X}, \quad y_i \in \mathbf{y}.$$

From Multivariable Calculus, we know that first-order conditions defining candidate minimizers $\boldsymbol{\theta}^*$ are that all partial derivatives of $L_{\mathbf{D}}(\boldsymbol{\theta}^*)$ are zero,

$$\frac{\partial L_{\mathbf{D}}(\boldsymbol{\theta}^*)}{\partial \theta_0} = \cdots = \frac{\partial L_{\mathbf{D}}(\boldsymbol{\theta}^*)}{\partial \theta_n} = 0. \tag{1.1}$$

For any $j = 0, \ldots, n$, we can calculate using the chain rule,

$$\begin{aligned} \frac{\partial L_{\mathbf{D}}(\boldsymbol{\theta})}{\partial \theta_j} &= \frac{1}{m} \sum_{i=1}^{m} 2\left(h_{\boldsymbol{\theta}}(x_i) - y_i\right) \frac{\partial h_{\boldsymbol{\theta}}(x_i)}{\partial \theta_j} \\ &= \frac{2}{m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x_i \cdots + \theta_n x_i^n - y_i\right) x_i^j \\ &= \frac{2}{m} \sum_{i=1}^{m} x_i^j \sum_{k=0}^{n} x_i^k \cdot \theta_k - \frac{2}{m} \sum_{i=1}^{m} x_i^j y_i. \end{aligned}$$

Introducing the *Vandermonde* matrix

$$V = \begin{bmatrix} 1 & x_1 & \cdots & x_1^n \\ 1 & x_2 & \cdots & x_2^n \\ \vdots & & & \vdots \\ 1 & x_m & \cdots & x_m^n \end{bmatrix} \in \mathbb{R}^{m \times (n+1)}$$

we can notice (see Problem Sheet 1, Task (a)) that the first-order conditions (1.1) can be written as the linear system of equations

$$A\boldsymbol{\theta} = \mathbf{b}, \qquad A = V^\top V, \quad \mathbf{b} = V^\top \mathbf{y}. \tag{1.2}$$

These equations can be solved using any method of Numerical Analysis, such as the Gaussian elimination[1] or the Gauss–Seidel method.

Considering also the eigenvalues of the Hessian of $L_{\mathbf{D}}(\boldsymbol{\theta})$, we can find out that the parameters $\boldsymbol{\theta}$ computed from (1.2) are the *global minimizer* of $L_{\mathbf{D}}(\boldsymbol{\theta})$.

However, what happens with the **future forecast** if we vary the **polynomial degree** n, even if for each n we compute the optimal $\boldsymbol{\theta}$ from (1.2)? Let's experiment with this in the Python part of Problem Sheet 1.

End of lecture 1

1.2.3 Underfitting and overfitting

This simple example highlights why machine learning is different from numerical analysis or statistics.

- Fitting is not predicting. Fitting existing data well is fundamentally different from making predictions about new data.

[1] Implemented for instance in the `numpy.linalg.solve` function in Python.

- Using a too simple model (e.g. too small n) can result in *underfitting.*

  Definition 1.7. *A prediction model is said to* ***underfit*** *(the data) if both the fitting error (loss on the known data) and the prediction error (loss on new data) are large.*

- Using a too complex model (e.g. too large n) can result in *overfitting.* Increasing the complexity (that is, the number of tuneable parameters) of the model will usually yield better results on the known data. However, when the amount of the known data is small, and/or this data is noisy, and/or the new data which we want to predict is too different from the known data, then the prediction can degrade significantly, even more so than in the underfitting scenario.

  Definition 1.8. *A prediction model is said to* ***overfit*** *(the data) if the fitting error is small, but the prediction error is large.*

1.2.4 Training data and test data splitting

The issues of underfitting and overfitting indicate that we need a reliable algorithm for estimating the prediction error (more generally, any desired prediction loss). Ideally, this algorithm should be independent of the particular model of the prediction rule. For example, we should not rely on the latest polynomial coefficient. In machine learning, the prediction loss is usually estimated (and optimized) by partitioning the data into **training** and **test** sets and using the **cross validation**.

Given the dataset $\mathbf{D}$ (with or without labels), the first step is to *randomly* divide it into two mutually exclusive subsets $\mathbf{D}_{train}$ and $\mathbf{D}_{test}$, called the training and test sets. Crucially, the test set should be excluded **before** performing any analysis (such as optimising parameters) – otherwise such analysis can lead to incorrect conclusions. Typically, the majority of the data (for example, 90%) are partitioned into the training set with the remainder going into the test set. Now the prediction rule is computed by performing an appropriate machine learning task on the training dataset **only**. In other words, the *learning* of the prediction rule is achieved by *training* it on the training set, hence the name.

After the "learning" (or "training") is completed, the prediction rule is **tested** by applying it to the yet unseen data from the testing set, and computing the corresponding loss function. The value of the loss function on the test set can now be compared across candidate models, since none of these models have been "spoiled" by the test data during the training phase.

Definition 1.9. ***Data splitting*** *is a random a priori partitioning of the given dataset* $\mathbf{D}$ *into a* ***training*** *dataset* $\mathbf{D}_{train}$ *and a* ***test*** *dataset* $\mathbf{D}_{test}$ *such that* $\mathbf{D} = \mathbf{D}_{train} \cup \mathbf{D}_{test}$, $\mathbf{D}_{train} \cap \mathbf{D}_{test} = \emptyset$.

When $\mathbf{D} = (\mathbf{X}, \mathbf{y})$ is labelled, it is most crucial to partition labels **together** with their corresponding domain points. That is, a **labelled** dataset is split as follows:

$$\mathbf{X} = \mathbf{X}_{train} \cup \mathbf{X}_{test}, \quad \mathbf{X}_{train} \cap \mathbf{X}_{test} = \emptyset, \qquad \mathbf{y} = \mathbf{y}_{train} \cup \mathbf{y}_{test}, \quad \mathbf{y}_{train} \cap \mathbf{y}_{test} = \emptyset.$$

1.2.5 Empirical risk minimisation

It's easier to illustrate training and testing on a supervised learning problem. To make the data splitting as in Def. 1.9 useful, we must assume that the (total) loss function admits

some matching partitioning too. For simplicity (since the selection of a machine learning model admits quite a freedom anyway), one often assumes that there exists a **pointwise loss function**

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

that takes the true label $y \in \mathcal{Y}$ and the prediction $\hat{y} \in \mathcal{Y}$ (for instance, $\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x})$) as input, and produces a real nonnegative number (the loss value) representing how much error we have made on this particular prediction. Now we can average these losses over given training data.

Definition 1.10. *Given a pointwise loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$, training domain points $\mathbf{X}_{train} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ and corresponding labels $\mathbf{y}_{train} = \{y_1, \ldots, y_m\}$, and a prediction rule $h_{\boldsymbol{\theta}}(\mathbf{x})$, the **empirical risk** (or **training loss**) is the average pointwise loss over $\mathbf{D}_{train}$:*

$$L_{\mathbf{D}_{train}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \ell(y_i, h_{\boldsymbol{\theta}}(\mathbf{x}_i)). \tag{1.3}$$

Now the optimal tuneable parameters can be computed.

Definition 1.11. *Given a pointwise loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$, training domain points $\mathbf{X}_{train} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ and corresponding labels $\mathbf{y}_{train} = \{y_1, \ldots, y_m\}$, and a prediction rule $h_{\boldsymbol{\theta}}(\mathbf{x})$ with $\boldsymbol{\theta} \in \mathbb{R}^n$, the **empirical risk minimizer** (ERM) is defined as*

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \mathbb{R}^n} L_{\mathbf{D}_{train}}(\boldsymbol{\theta}). \tag{1.4}$$

Remark 1.12. *The availability of the pointwise loss $\ell(y, \hat{y})$ is essential. Consider, for example, the following loss that depends on the entire dataset $\mathbf{D} = \{x_1, x_2, x_3\}$:*

$$L_{\mathbf{D}} = \begin{cases} x_1 + x_2, & x_3 \geq 0, \\ x_1 - x_2, & x_3 < 0. \end{cases}$$

In this case we cannot separate for example x_1, x_2 into the training set, since the value of $L_{\mathbf{D}}$ cannot be reduced to use only x_1 and x_2.

Once the empirical risk minimizer is computed, we can test its prediction quality on the previously excluded test set.

Definition 1.13. *Given a pointwise loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$, test domain points $\mathbf{X}_{test} = \{\mathbf{x}_{m+1}, \ldots, \mathbf{x}_N\}$ and corresponding labels $\mathbf{y}_{test} = \{y_{m+1}, \ldots, y_N\}$, and a prediction rule $h_{\boldsymbol{\theta}}(\mathbf{x})$, the **test loss** is the average pointwise loss over $\mathbf{D}_{test}$:*

$$L_{\mathbf{D}_{test}}(\boldsymbol{\theta}) = \frac{1}{N - m} \sum_{i=m+1}^{N} \ell(y_i, h_{\boldsymbol{\theta}}(\mathbf{x}_i)). \tag{1.5}$$

Another reason for the particular form (and name) of the empirical risk (1.3) comes from probability and statistics.

1.3 Introduction to statistical learning theory

Recall the following definitions from introductory probability theory[2], slightly extended to allow data points $\mathbf{x}$ to be vectors.

[2]At Bath, this is covered in the first-year units, MA10211 and MA10212, on Probability & Statistics

Definition 1.14. *The* sample space Ω *is the set of all possible (random) outcomes* $\omega \in \Omega$.

Definition 1.15. *A (random)* event *is any subset* $C \subseteq \Omega$.

Definition 1.16. *We denote the* set of all events, $C \subseteq \Omega$, *by* $\mathcal{F}$.

Definition 1.17. *A* probability *measure is a function* $\mathbb{P} : \mathcal{F} \to [0, 1]$ *satisfying* $\mathbb{P}(\Omega) = 1$ *and* $\mathbb{P}\big(\bigcup_{i\in I} C_i\big) = \sum_{i\in I} \mathbb{P}(C_i)$ *whenever* $\{C_i\}_{i\in I}$ *is a countable collection of disjoint events.*

Definition 1.18. *A* random variable *is a real-valued (possibly vector) function defined on the sample space,*

$$X : \Omega \to \mathbb{R}^n.$$

Any outcome $\omega \in \Omega$ *yields a* sample $X(\omega)$. *Therefore, any event* $C \subset \Omega$ *yields a real-valued domain* $D_C = \{X(\omega) : \omega \in C\} \subset \mathbb{R}^n$. *Similarly, we can extend the probability of event* $\mathbb{P}(C)$ *to the* probability distribution *of a random variable,* $\mathbb{P}(X \in D_C) = \mathbb{P}(C)$, *and write* $X(\omega) \sim \mathbb{P}$.

Definition 1.19. X *is a* continuous random variable *if there exists a piecewise continuous function* $f_X : \mathbb{R}^n \to [0, \infty)$ *(called* probability density function*) such that for any* $D \subset \mathbb{R}^n$,

$$\mathbb{P}(X \in D) = \int_D f_X(\mathbf{x})d\mathbf{x}.$$

The normalisation axiom implies $\mathbb{P}(\mathbb{R}^n) = \int_{\mathbb{R}^n} f_X(\mathbf{x})d\mathbf{x} = 1$. If $n = 1$ and $D = [a, b]$, $\int_D$ is a standard definite integral $\int_a^b$. If $n > 1$, $\int_D$ is a volume integral, and $f_X(\mathbf{x})$ is the *joint* probability density function. For example, if $D = [a, b] \times [\mu, \nu]$ in $n = 2$, $\int_D$ is a double integral $\int_a^b \int_\mu^\nu$. However since n can be arbitrary, we will always write a single integral sign, $\int_D$.

Definition 1.20. *The* expectation *of a continuous random variable* X *is*

$$\mathbb{E}[X] = \int_{\mathbb{R}^n} \mathbf{x} f_X(\mathbf{x})d\mathbf{x}$$

as long as

$$\int_{\mathbb{R}^n} \|\mathbf{x}\| f_X(\mathbf{x})d\mathbf{x} < \infty.$$

Here, $\|\mathbf{x}\|$ is any valid norm of vectors in $\mathbb{R}^n$. For example, we will mostly use the *euclidean* norm (also called *2-norm*), $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}$.

Theorem 1.21 (The law of the unconscious statistician). *For a continuous random variable* X *and function* $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^{n'}$,

$$\mathbb{E}[\mathbf{g}(X)] = \int_{\mathbb{R}^n} \mathbf{g}(\mathbf{x}) f_X(\mathbf{x})d\mathbf{x}$$

as long as

$$\int_{\mathbb{R}^n} \|\mathbf{g}(\mathbf{x})\| f_X(\mathbf{x})d\mathbf{x} < \infty.$$

Several random variables (vector or not) can be put together into one bigger random variable, such as $Z = (X, Y)$. Now one can consider the probability density function $f_Z(\mathbf{z}) \equiv f_{X,Y}(\mathbf{x}, \mathbf{y})$.

Definition 1.22. *The random variables* $X : \Omega \to \mathbb{R}^n$ *and* $Y : \Omega \to \mathbb{R}^{n'}$ *are* independent *if*

$$f_{X,Y}(\mathbf{x}, \mathbf{y}) = f_X(\mathbf{x}) f_Y(\mathbf{y}), \quad \forall \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^{n'}.$$

For independent random variables, one can split the expectation of a product,

$$\mathbb{E}[g(X)\cdot h(Y)] = \int_{\mathbb{R}^{n'}} \left(\int_{\mathbb{R}^n} g(\mathbf{x}) f_X(\mathbf{x}) d\mathbf{x}\right) h(\mathbf{y}) f_Y(\mathbf{y}) d\mathbf{y} = \mathbb{E}_X[g(X)]\cdot \mathbb{E}_Y[h(Y)],$$

where we denote by $\mathbb{E}_X, \mathbb{E}_Y$ expectations using *marginal* probability density functions f_X, f_Y.

Now suppose the function of a continuous random variable depends on a *parameter*, $g_{\boldsymbol{\theta}}(X)$. Since the expectation is just the integral, we can *differentiate* it over $\boldsymbol{\theta}$ (as long as $g_{\boldsymbol{\theta}}$ is differentiable and the integral exists),

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}[g_{\boldsymbol{\theta}}(X)] = \int_{\mathbb{R}^n} \nabla_{\boldsymbol{\theta}} g_{\boldsymbol{\theta}}(\mathbf{x}) f_X(\mathbf{x}) d\mathbf{x} = \mathbb{E}[\nabla_{\boldsymbol{\theta}} g_{\boldsymbol{\theta}}(X)].$$

1.3.1 Data distribution and expected risk

The so-called **statistical learning theory** considers data as random variables to formalise and quantify the selection of supervised learning models, success of training a particular class of models, or a lack thereof. It operates under the following **assumptions.**

1. *Data belong to some probability space* $(\Omega, \mathcal{F}, \mathbb{P})$.
   For instance, a labelled dataset $\mathbf{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$ consists of independent random samples from the same distribution, $(X(\omega), Y(\omega)) \sim \mathbb{P}$.
2. *There exists a* ***hypothesis class*** $\mathcal{H} = \{h_{\boldsymbol{\theta}}(\mathbf{x})\}$, *containing all prediction rules of interest.*
   In addition to just coefficients $\boldsymbol{\theta}$, this may include substantially different *families* of functions, such as polynomial, trigonometric, tabular and so on.

Note that as long as any data-label point $(X(\omega), Y(\omega))$ is a random variable, so is the pointwise loss of any prediction rule. We are interested in **statistics of prediction success**, such as:

- an expectation of the loss of the given prediction rule at **any** possible sample from $\mathbb{P}$;
- a probability that the loss is below some upper bound; or vice versa,
- an expectation or lower bound on the **size** of the training dataset that is needed to train a prediction rule with a desired expectation or upper bound on the loss.

The law of the unconscious statistician allows us to define such an expectation of the loss.

Definition 1.23. *Given a data probability space* $(\Omega, \mathcal{F}, \mathbb{P})$ *and a prediction rule* $h_{\boldsymbol{\theta}}(\mathbf{x})$, *the* ***expected risk*** *is defined as*

$$L(\boldsymbol{\theta}) = \mathbb{E}[\ell(Y(\omega), h_{\boldsymbol{\theta}}(X(\omega)))], \qquad (X(\omega), Y(\omega)) \sim \mathbb{P}, \tag{1.6}$$

Theoretically, one could still formulate a "Holy Grail" problem of optimising the expected risk $L(\boldsymbol{\theta})$ with respect to the tuneable parameters $\boldsymbol{\theta}$, which takes into account the entire probability space which can realise any possible training data. In practice, this problem is of course unsolvable. In the assumption of independent training data points, the empirical risk (1.3) can be seen as a **Monte Carlo** quadrature, **approximating** the exact expected risk (1.6). If the variance of the loss $\ell(Y(\omega), h_{\boldsymbol{\theta}}(X(\omega))$ is bounded, both the training loss (empirical risk) and the test loss converge to the expected risk,

$$\lim_{m\to\infty} L_{\mathbf{D}_{train}}(\boldsymbol{\theta}) = \lim_{(N-m)\to\infty} L_{\mathbf{D}_{test}}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}).$$

In turn, empirical risk minimisation can be seen as fitting a prediction rule to a finite number of known empirical samples from the desired distribution.

End of lecture 2

1.3.2 Cross validation: estimation of the prediction error and model selection

The test loss (1.5) can already be monitored to prevent overfitting. However, separating *precisely* the data points $m+1$ to N into the test set suffers from two issues. First, these particular points may incidentally be too good or too bad points in terms of the test error compared to actually new points the model will be evaluated on in routine practice. Moreover, by selecting the precise indices, we violate the assumption of independent sampling. Second, we would like to use as much of the data available to train the model as possible, but the amount of data is limited. This would require us to keep our test set $\mathbf{D}_{test}$ small, which then would lead to a noisy estimate of the test error with high variance.

One solution to these contradictory objectives (large training set, large test set) is to use **cross validation**. More precisely, the K-fold cross validation partitions the data into K chunks, $K-1$ of which form the training set $\mathbf{D}_{train}$, and the last chunk serves as the test set $\mathbf{D}_{test}$. Each chunk is composed of *randomly* selected data points, instead of just taking consecutive points from the original ordering of data. Moreover, the cross validation iterates through all assignments of chunks to $\mathbf{D}_{train}$ and $\mathbf{D}_{test}$, as illustrated in Figure 2 and Algorithm 1.

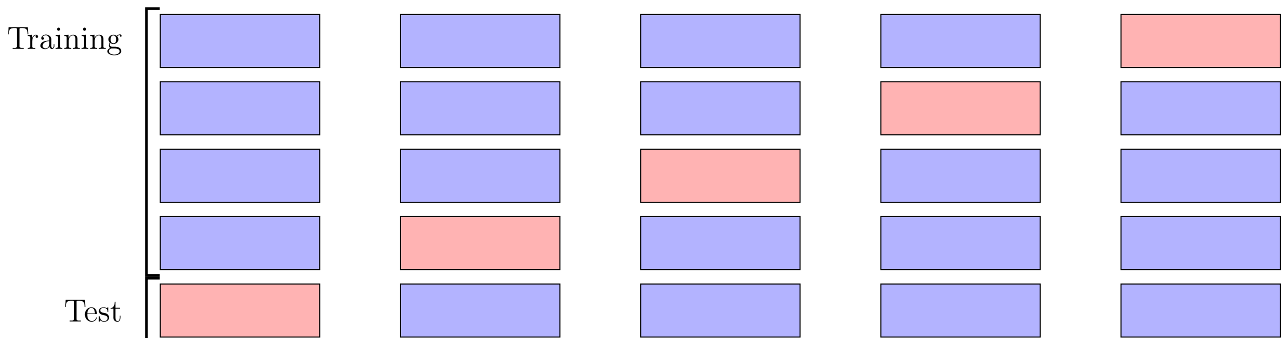


Figure 2: Data splitting in the 5-fold cross validation

The cross validation loss (1.7) can be computed for **different** classes $\mathcal{H}$ (for example, polynomials of degrees varying from 1 to 10), and the model which gives the minimal L_{cv} is selected as the ultimate prediction rule.

1.3.3 Bias-Variance tradeoff

We have already seen the data splitting and cross validation that can give some indication that a prediction rule that gives a small loss on the training dataset, $L_{\mathbf{D}_{train}}$, will give a small loss on a new test dataset, $L_{\mathbf{D}_{test}}$. Can we say something general about the expected risk (1.6)?

We can decompose it into two components

$$L(\boldsymbol{\theta}) = L_{bias} + L_{var}(\boldsymbol{\theta}), \quad \text{where} \quad L_{bias} = \min_{h_{\boldsymbol{\theta}} \in \mathcal{H}} L(\boldsymbol{\theta}), \quad L_{var}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) - L_{bias}. \tag{1.8}$$

These two components have the following meaning.

- L_{bias} is the **bias (or approximation) loss**. This is the minimum loss achievable by any rule in $\mathcal{H}$. This term measures how much loss we have because we restrict ourselves to a specific class $\mathcal{H}$, namely, how much *inductive bias* we have, hence the name. The bias does not depend on the size of the training set, and is determined only by the hypothesis

Algorithm 1 K-fold Cross validation

1: Shuffle $\mathbf{D}$ randomly, keeping data-label pairs. ▷ Using e.g. `numpy.random.shuffle`

2: Partition $\mathbf{D}$ into K chunks

$$\mathbf{D}_k = \{(\mathbf{x}_{1+N(k-1)/K}, y_{1+N(k-1)/K}), \dots, (\mathbf{x}_{Nk/K}, y_{Nk/K})\}, \qquad k = 1, \dots, K.$$

3: **for** $k = 1, \dots, K$ **do**

4: Assemble the k-th test and training sets ▷ "\" denotes the set subtraction

$$\mathbf{D}^{(k)}_{test} = \mathbf{D}_k, \qquad \mathbf{D}^{(k)}_{train} = \mathbf{D} \backslash \mathbf{D}_k.$$

5: Minimize the empirical risk over each training set and the **same class** $\mathcal{H}$ ▷ e.g. degree-n polynomials

$$\boldsymbol{\theta}^{(k)} = \arg\min_{\boldsymbol{\theta} \in \mathbb{R}^n} L_{\mathbf{D}^{(k)}_{train}}(\boldsymbol{\theta}).$$

6: **end for**

7: Compute the **cross validation loss**

$$L_{cv}(\mathcal{H}) = \frac{1}{K} \sum_{k=1}^{K} L_{\mathbf{D}^{(k)}_{test}}(\boldsymbol{\theta}^{(k)}). \tag{1.7}$$

class chosen. Enlarging the hypothesis class can decrease the bias (see the blue dashed line in Figure 3). If the class $\mathcal{H}$ can predict any possible relation between $\mathbf{x}$ and y under the distribution $\mathbb{P}$, L_{bias} is zero. However, if the class $\mathcal{H}$ is small, the bias is large, and the loss on the actual test set can never be smaller. For example, linear polynomials would have a high bias on a general data, such as the temperature.

- L_{var} is the **variance** (also called **estimation** or **generalisation**) **loss**. This is the "overhead" of training $h_{\boldsymbol{\theta}}$ by minimising the empirical risk on only m specific data points in $\mathbf{D}_{train}$. Usually L_{var} increases with the size of the hypothesis class $|\mathcal{H}|$, and decreases with the number of training samples m, see the red dotted line in Figure 3. In the temperature forecast example, a degree 10 polynomial has high variance, since a tiny change in the training data can flip the extrapolation beyond 16 months from going downwards to going upwards and vice versa. We can think of the size of $\mathcal{H}$ as a measure of its *complexity*, or, equivalently, memory capacity of the learning method – how detailed a characterization of the training set it can remember and then apply to new data.

Since the bias decreases with $|\mathcal{H}|$, while the variance increases, the total loss $L(\boldsymbol{\theta})$, shown as the black line in Figure 3, exhibits a minimum at an *intermediate* model complexity $|\mathcal{H}^*|$, shown by the vertical green line in Figure 3. This is called the **bias-complexity**, or **bias-variance tradeoff**. On the left of $|\mathcal{H}^*|$ we face underfitting, since a large L_{bias} will make both training and test losses large. On the right of $|\mathcal{H}^*|$ we face overfitting, since one can find a $\boldsymbol{\theta}^*$ which for *one particular* $\mathbf{D}_{train}$ will give a small training loss close to a small L_{bias}, but it may predict poorly for a different dataset, leading to a large test loss. Applied machine learning research concerns designing good hypothesis classes for the given problem. For example, we can expect that the temperature in the same city changes smoothly in time, so it should lend itself well to a low-order polynomial approximation.

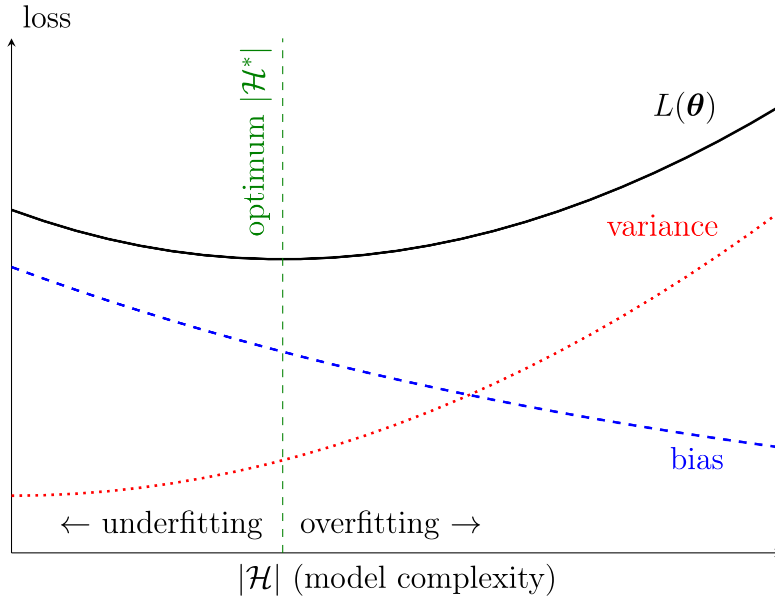


Figure 3: Bias-Variance tradeoff. This schematic shows the typical behaviour of the bias and variance losses as we increase the size of the hypothesis class. Notice how the bias always decreases with model complexity, but the variance, i.e. fluctuation in performance due to finite size sampling effects, increases with model complexity. Thus, optimal performance is achieved at intermediate levels of model complexity.

1.3.4 The No-Free-Lunch Theorem

The previous consideration may imply that it might actually make sense to take as large $\mathcal{H}$ as possible (ideally having a zero bias), and then try to find some "universal" learning algorithm to decrease the variance as well. However, we are limited by the number of training data points m we can realistically observe. In practice, we will almost always have continuous $\mathbf{x} \in \mathbb{R}^n$, and, if the distribution $\mathbb{P}$ has no restrictions at all, its full characterisation would need infinite amount of information. More rigorously, we can formulate a theorem (a counterexample, rather) that any learning algorithm that *does not see all possible outcomes* of the data-generating distribution will fail at some predictions.

Theorem 1.24 (No-Free-Lunch)**.** *Consider the binary classification task with $y \in \{-1, 1\}$ and the 0–1 loss*

$$\ell(y, h_{\boldsymbol{\theta}}(\mathbf{x})) = \begin{cases} 0, & h_{\boldsymbol{\theta}}(\mathbf{x}) = y, \\ 1, & \textit{otherwise.} \end{cases}$$

Consider a finite set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_{2m}\}$ with some $m > 0$. Let $h_{\boldsymbol{\theta}}(\mathbf{x})$ be any prediction rule trained by an algorithm that uses a training dataset with only m points. Then there exists a distribution $\mathbb{P}$ over $(\mathbf{x}(\omega), y(\omega))$, $\mathbf{x}(\omega) \in \mathbf{X}$, $y(\omega) \in \{-1, 1\}$, such that with probability of at least $1/7$, the total loss $L(h_{\boldsymbol{\theta}}) \geq 1/8$.

The formal proof is a bit long, so we skip it at this point. The intuition is that any learning algorithm that observes only half of the instances in $\mathbf{X}$ has no information on what should be the labels for the rest of $\mathbf{X}$. Therefore, there exist some "true" function $y(\mathbf{x})$ that can be learned on $\mathbf{X}_{train}$ of m points with a zero training loss, but will contradict the labels that $h_{\boldsymbol{\theta}}$ predicts on the remaining points in $\mathbf{X}$. The precise constants appear from the fact that the classification is binary, so there is only a finite amount (2^{2m}) of all possible labelling functions from $\mathbf{X}$ to $\{-1, 1\}$, and we can work with explicitly computable uniform distributions.

1.3.5 Fundamental theorem of statistical learning

For a particular class of machine learning problems a rigorous estimate of the expected risk is possible.

Definition 1.25. *A machine learning problem is called* ***binary classification*** *if the data is labelled, and* $y \in \mathcal{Y} = \{-1, 1\}$.

Definition 1.26. *A set* $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$ *is said to be* ***shattered*** *by a hypothesis class* $\mathcal{H}$ *if for any labels* $y_1, \dots, y_m \in \{-1, 1\}$ *there exists a prediction rule,* $h \in \mathcal{H}$*, that reconstructs these labels exactly,* $h(\mathbf{x}_i) = y_i$ *for all* $i = 1, \dots, m$.

Example 1.27. *Consider* $x, \theta \in \mathbb{R}$*, and* $\mathcal{H}_{s1} = \{h_\theta(x) = \operatorname{sign}(x - \theta)\}$*, the hypothesis class of one-dimensional sign functions. Consider a set of one point* $\mathbf{X} = \{x_1\}$*, with arbitrary* x_1*. This set is shattered by* $\mathcal{H}_{s1}$*. Indeed, we have only two possibilities:* $y_1 = 1$ *or* $y_1 = -1$*. In the first case, take any* $\theta < x_1$*, then* $\operatorname{sign}(x_1 - \theta) = 1$ *is* y_1 *exactly. In the second case, take any* $\theta > x_1$*, then* $\operatorname{sign}(x_1 - \theta) = -1$*, which is also the exact prediction.*

Example 1.28. *Consider the same* $\mathcal{H}_{s1}$ *as in the previous example, but now take a set of two distinct points,* $\mathbf{X} = \{x_1, x_2\}$*,* $x_1 < x_2$*. This set is* ***not*** *shattered by* $\mathcal{H}_{s1}$*. Indeed, labels* $y_1 = 1, y_2 = -1$ *can never be predicted exactly by* $\operatorname{sign}(x - \theta)$*, since any* $\theta < x_2$ *will produce a wrong prediction at* x_2*,* $\operatorname{sign}(x_2 - \theta) = 1 \neq y_2$*, whereas any* $\theta \geq x_2$ *will produce a wrong prediction at* x_1*,* $\operatorname{sign}(x_1 - \theta) = -1 \neq y_1$*.*

Now we are ready to formulate the measure of the complexity of the class $\mathcal{H}$.

Definition 1.29. *The* ***Vapnik-Chervonenkis (VC) dimension*** *of a hypothesis class* $\mathcal{H}$ *is the cardinality of the largest set* $\mathbf{X}$ *that is shattered by* $\mathcal{H}$*.*

Recalling the sign function, Example 1.27 indicates that $\operatorname{VCdim}(\mathcal{H}_{s1}) \geq 1$, since there is at least one set of size 1 that is shattered. Example 1.28 indicates that $\operatorname{VCdim}(\mathcal{H}_{s1}) < 2$, since no size-2 set can be shattered by $\mathcal{H}_{s1}$. Therefore, $\operatorname{VCdim}(\mathcal{H}_{s1}) = 1$. Although not generally the case, in many practical examples the VC dimension is proportional to the number of parameters in $\boldsymbol{\theta}$. Now we can use the VC dimension to estimate the minimal number of training points needed to learn a binary classifier to a satisfactory accuracy.

Theorem 1.30 (The Fundamental Theorem of Statistical Learning)**.** *Let the pointwise loss be* $\ell(y, \hat{y}) = |\hat{y} - y|$*. Fix* $\varepsilon, \delta > 0$*, and suppose there exists a probability space* $(\Omega, \mathcal{F}, \mathbb{P})$*, defining the binary classification problem, such that the training set* $\mathbf{D}_{train} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$ *consists of* m *independent samples from* $\mathbb{P}$*. Suppose the hypothesis class* $\mathcal{H}$ *has the VC-dimension* $k \in \mathbb{N}$*.*

Then there exists a constant $C > 0$ *independent of* ε, δ *and* k *such that if*

$$m \geq \frac{C}{\varepsilon^2}(k \log(\varepsilon^{-1}) + \log(\delta^{-1})),$$

then with probability at least $1 - \delta$ *the expected and empirical risks are close,*

$$|L(\boldsymbol{\theta}) - L_{\mathbf{D}_{train}}(\boldsymbol{\theta})| \leq \varepsilon,$$

where $L_{\mathbf{D}}(h) = \frac{1}{m}\sum_{i=1}^{m} \ell(h(\mathbf{x}_i), y_i)$ *and* $L(h) = \mathbb{E}[\ell(h(\mathbf{x}), y)]$*.*

This theorem implies that the empirical risk (training loss) is a reliable estimation of the true loss if the size of the training dataset is inversely proportional to ε^2 (up to a much more slowly increasing logarithmic term). This is similar to the estimate of the number of independent Monte Carlo samples needed to estimate an expectation of a random variable with an error of ε. On the other hand, the required number of training samples is also proportional to the VC-dimension of the chosen class of prediction rules, which is natural to prevent overfitting. For example, it is hopeless to train a prediction rule of VC-dimension k with less than k training points – it will overfit almost surely! The behaviour of training and test losses depending on the number of training points (for fixed class $\mathcal{H}$!) can be seen in Figure 4.

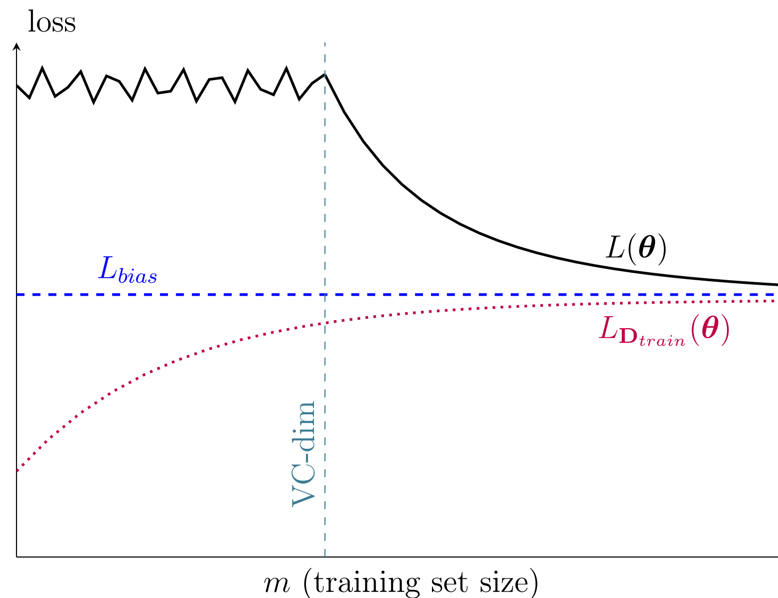


Figure 4: Bias-Variance tradeoff depending on the number of training samples. The training loss usually increases towards the bias of the given $\mathcal{H}$. Expected (and test) loss can be essentially arbitrary until the number of training points exceeds the VC-dimension of $\mathcal{H}$, after which it starts decreasing towards the bias too.

End of lecture 3

Summary

- Labelled and unlabelled data comes hand in hand with the type of machine learning problem (supervised or unsupervised), which needs to be chosen accordingly.
- "Learning" can be often formalised as an optimisation of a loss function with respect to the parameters of the prediction rule. However, ...
- ... relying only on the loss value on the training data can be misleading and overfit.
- Cross validation can be used to find sweet-spot parameters preventing this. Mathematically, this is backed up by ...
- ... the statistical learning theory which considers data as samples from some probability distribution, and links together the expected loss on any new data, the complexity (VC-dimension) of a class of prediction rules, and the amount of training data needed to learn an accurate prediction rule in that class.

2 Data preparation and retrieval

Before any learning can occur, we need to prepare the data in a computer-readable form, suitable for our prediction rule and learning algorithm. The initial source of data may be rather lacking of this. For example, the data may be not numerical (so it's not even obvious how to do mathematics on it), some samples may be missing or (worse!) wrong.

In this section, we will first review data formats and ways to operate with them in Python. Then we will consider information retrieval from "rough" data, such as natural text, which allows one to compare and score such data. Scoring can be also seen as prediction (of the relevance of a document to a query, for example), although this is not usually called "learning", since we do not optimise any parameters.

2.1 Numerical and non-numerical data

2.1.1 Explicit numerical data in vectors and matrices

The simplest scenario is when the data is already given in numbers. In this case, the domain data points $\mathbf{x} \in \mathbf{X}$ can be always written as vectors of some dimension n, $\mathbf{x} \in \mathbb{R}^n$ or $\mathbf{x} \in \mathbb{C}^n$.

Note that the original data points can be given in a different structure, for example, as matrices. However, any finite amount of numbers can be written as a vector. Indeed, let us assume that the original data is given as a matrix $X = [X_{i,j}] \in \mathbb{R}^{m \times L}$. But we can also stack the rows of the matrix horizontally instead of vertically:

$$\begin{bmatrix} \begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix} \\ \begin{pmatrix} 4 & 5 & 6 & 7 \end{pmatrix} \end{bmatrix} \to \begin{bmatrix} \begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix} & \begin{pmatrix} 4 & 5 & 6 & 7 \end{pmatrix} \end{bmatrix}.$$

Note that we have obtained a vector of size $n = mL$. In general, the *vectorisation* of a matrix $X \in \mathbb{R}^{m \times L}$ is a vector $\mathbf{x} \in \mathbb{R}^{mL}$ with elements defined by the following formula:

$$\mathbf{x}_{iL+j} = X_{i,j}, \qquad i = 0, \ldots, m-1, \quad j = 0, \ldots, L-1. \tag{2.1}$$

Note that this formula is reversible. One can think of the vector index $iL + j$ as a number composed from digits ij, where L is the range of j. For example, the number 37 is equal to $3 \cdot 10 + 7$, since 10 is the range of a decimal digit. Similarly to how two decimal digits can be chosen independently to compose any number from 0 to 99, the formula (2.1) guarantees that the vector index $iL + j$ encodes all possible combinations of the matrix indices i and j, and hence the vector $\mathbf{x}$ preserves all the elements of the matrix X. `Numpy` library has three functions suitable for reshaping of arrays and corresponding index conversion.

`np.reshape`

is, well, reshaping an array to a different shape (with the same total number of elements of course). We can use it in both directions, calling

`x = np.reshape(X, m*L)` and `X = np.reshape(x, (m,L))`

to convert a matrix into a vector, and a vector into a matrix, respectively. Note that if the output is a matrix, its desired shape needs to be given as a tuple. If we ever need to access individual elements by their indices, we can use

`np.unravel_index` and `np.ravel_multi_index`

to extract individual i, j from $iL+j$, or to combine i, j into the single index $iL+j$, respectively.

2.1.2 One-dimensional data: time series and discretised functions

It is quite natural to collect data at different points in time, such as in the temperature forecast example from Section 1.2.2. More generally, we assume that we have a function that depends on time, $f(t) : [0, \infty) \to \mathbb{R}$.

Note that we may also have a vector-function $\mathbf{f}(t)$ if we treat multiple data streams as different components of this vector-function. The motivation for considering a continuous variable t first stems from *physical models* which are often associated with the observed data. The variation of air temperature, for example, is due to the physical model of the atmosphere. These models are often posed as (partial) differential equations, which are the Cauchy problems with respect to the time, which makes t a continuous variable.

However, we cannot observe *all* infinitely many values of t of course. In practice, we have to sample data at a **finite set** of time points $0 \le t_1 < t_2 < \ldots < t_m$. Moreover, the observation may contain a **noise** (e.g. imperfection of the thermometer) which is independent of the underlying function $f(t)$. The actual data we observe is therefore a vector $\mathbf{y} \in \mathbb{R}^m$ with elements

$$y_i = f(t_i) + \xi_i, \quad i = 1, \ldots, m,$$

where ξ_i is (typically) a realisation of a random variable Ξ modelling the observation noise. A similar **observation model** was assumed in Problem Sheet 2. We see that y_i are labels of the regression problem, whereas the domain points $x_i = t_i$ are the time points.

Even in the assumption of no noise (almost impossible in practice, but we can assume that the noise is very small compared to data) we can still raise the question of Statistical Learning: how large m we need to learn an accurate model? This can be addressed by the concept of **interpolation** (a classical topic within Numerical Analysis): if we were to interpolate f at an arbitrary new point $x \neq x_i$, what can we say about the error of an interpolant reconstructable from $\mathbf{y}$ with respect to the exact $f(x)$? Without any better knowledge, we can start with the **piecewise linear interpolation**:

$$h_{\boldsymbol{\theta}^*}(x) = \theta_i^* + \left(\frac{\theta_{i+1}^* - \theta_i^*}{x_{i+1} - x_i}\right)(x - x_i), \quad \text{where} \quad i : x \in [x_i, x_{i+1}], \quad \text{and} \quad \boldsymbol{\theta}^* = \mathbf{y}.$$

The **expected risk** can be estimated using Corollary 2.2 of MA20222 (Numerical Analysis):

$$L(\boldsymbol{\theta}^*) = \int_{x_1}^{x_m} |f(x) - h(x)| \frac{1}{x_m - x_1} dx \le \frac{\max_i (x_{i+1} - x_i)^2}{8} \max_{x \in [x_1, x_m]} |f''(x)|.$$

To make sure we can interpolate (**predict**) the data accurately to an arbitrary point on the entire observed time interval, we need to sample the time points with a small interval $(x_{i+1} - x_i)$.

One slightly more specific example of time series data is **audio**. The sound itself can be described with $f(t)$ being the pressure of the air, oscillations of which produce the perception of a sound in the ear. Recorded (and played) audio is a similar variation of electrical voltage on the microphone (speaker). Storing audio on a computer requires a similar sampling of f at finite time points x_i. However, an audio signal is necessarily an oscillating function. If the time points x_i and x_{i+1} are too far from each other, the exact function $f(t)$ may oscillate many times between these points, but this information is irrevocably lost if one has access to only the discrete vector $\mathbf{y}$.

To prevent this from happening, the interval $x_{i+1} - x_i$ should be not greater than the half-period of the fastest oscillation in $f(t)$. Humans can hear sounds of frequencies between 20 and 20 000 Hz, which corresponds to oscillation periods between $2\pi/20$ and $2\pi/20000$. To satisfy the half-period rule of thumb, the most commonly used frequency for audio recording is 44 100 Hz. This equates to the interval between time points $x_{i+1} - x_i = 1/44100 \approx 0.000023$ seconds. This means that audio data is rather large – $\mathbf{y}$ is a vector of millions of elements for a typical song lasting a few hundred seconds. Keeping such data "as is" would consume a considerable amount of storage. Old music Compact Disks, for example, were able to carry only 10-20 songs.

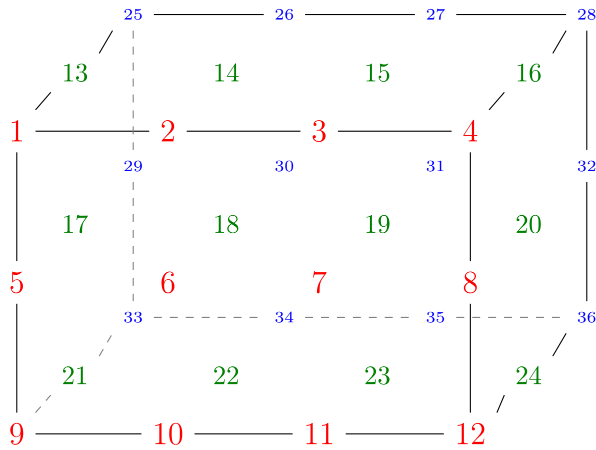
Data compression has become an important remedy. The general idea is to never keep each element of $\mathbf{y}$ explicitly, but to store only a much smaller amount of *generating* data, together with a recipe (algorithm) that reconstructs the original entries on demand. Data compression can be *lossy*, meaning that the reconstruction is only approximate with some (unimportant for e.g. human ear) information being lost, and *lossless*, which reconstructs the original $\mathbf{y}$ exactly. For example, MP3 and AAC (Advanced Audio Codec) are lossy audio compression algorithms, whereas the ZIP archiving is a lossless compression.

The drawback of data compression is that we may need to **convert** the data for use in e.g. Python. Getting back to audio, the `SciPy` library has a module `io` (`https://docs.scipy.org/doc/scipy/reference/io.html`) which can load, among others, wav sound files as `numpy` arrays. Note that the sampling frequency is not always 44 100 Hz, and thus it is stored in a wav file, and returned as the first output by `scipy.io.wavfile.read`. The second output is the actual audio data. It is directly a vector if the audio is *mono*, or a $m \times 2$ matrix if the audio is *stereo*. If we are working in a Jupyter notebook, we can play an audio from a `numpy` vector by using a built-in player `IPython.display.Audio`. It takes two inputs: `data`, a vector of mono audio (or a $2 \times m$ matrix of stereo audio), and `rate`, the sampling frequency.

Some data compression techniques are more convenient for mathematical computations. We will consider one of such (the linear dimensionality reduction) in Section 3.2 later.

2.1.3 Images: 2- and 3-dimensional data

Images are just matrices of pixels. For a grayscale image, each pixel is essentially a single number, taking integer values between 0 and 255, which denote the 256 shades of gray (with 0 being full black, and 255 being full white). Colour images are slightly more complicated: each pixel consists of three colours (red, green and blue), a combination of which with different intensities can make a perception of any visible colour on a display. This means that each pixel is actually a vector of size 3, each element of which takes values between 0 and 255 denoting the shades of red, green and blue, respectively. The entire image is thus a *three-dimensional array*, $X \in \mathbb{R}^{m \times L \times 3}$, and each element is addressed by three indices, $X_{i,j,k}$, $i = 1, \ldots, m$, $j = 1, \ldots, L$ and $k = 1, 2, 3$. This can be visualised as a three-dimensional matrix (often referred to as a 3-tensor), or as a collection of 3 matrices, carrying intensities of each of the three colours.



$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \quad \begin{bmatrix} 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix} \quad \begin{bmatrix} 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \end{bmatrix}$$

In `numpy`, a three-dimensional array can be indexed by the same square brackets constructions, but with three indices or levels, for example,

`X[i,j,k]` or `X[i][j][k]`

There can be several ways of converting an image into a vector $\mathbf{x}$, suitable to become a domain point in a machine learning dataset. If we are only interested in *objects* in the image, but not their colours, we can *average* the colour values to obtain a grayscale image $\tilde{X} \in \mathbb{R}^{m \times L}$, where $\tilde{X}_{i,j} = \frac{1}{3}\sum_{k=1}^{3} X_{i,j,k}$. `Numpy` has a convenient parameter `axis` in statistical functions that indicates over which index the function should be applied. The averaged colour, for example, is the mean over the 2nd axis (in Python numbering):

```
tildeX = np.mean(X, axis=2)
```

Note that this is a *lossy* data compression though. To keep all information, we can extend the index conversion formula (2.1):

$$\mathbf{x}_{i \cdot 3L + j \cdot 3 + k} = X_{i,j,k}.$$

Again, we can use `numpy` functions

`np.reshape(X, m*L*3)` or simply `X.flatten()`

to stretch an array data into a vector.

Similar to audio, modern images are coded in sophisticated formats. In Python, we can load them as `numpy` arrays using `matplotlib.pyplot.imread`. To plot a `numpy` array as an *image* (and not just a collection of colourful dots), it's better to use `matplotlib.pyplot.imshow`.

2.1.4 Non-numerical data: text and categories

However, the original data may not contain any single number. For example, it can be a piece of text, which we want to complete, classify, or simply compare to other text. Another example is categorical data, when domain data points look more like labels, in the sense that they can take only one of a few predefined values, such as class numbers, 0 or 1. The usual procedure in this case is to convert the given data into some numerical form. The next section discusses a few ways of doing this for words.

2.2 Information retrieval from text data (non-examinable)

All of us engage frequently with plenty of text data: the Internet. Fast and relevant search on the Internet requires an appropriate preprocessing of text documents and queries to enable application of efficient mathematical search algorithms. This process in general is called information retrieval.

For the rest of this section, you may also look at the textbook of Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, `https://nlp.stanford.edu/IR-book/`.

2.2.1 Vector space model of text

Definition 2.1. *The **Boolean retrieval** produces a binary answer (Yes=1/No=0) for each document in the given text dataset.*

Example 2.2. *Find which plays of Shakespeare contain the words Romans and Caesar but not Calpurnia.*

Possible solution: scan the entire Shakespeare's Collected Works, answer Yes to plays containing words Romans and Caesar, and change the answer to No for plays containing Calpurnia. *Problems:*

- scanning large documents word by word for every query is slow.
- Verbatim scan is too inflexible. For example, we would like to take into account occurrences of both "Romans", as well as "romans" or "roman".
- The method should be extensible beyond the boolean retrieval to allow ranked, probabilistic and learnable retrieval.

Alternative solution: scan the documents in advance, record all unique terms, and replace documents by lists of terms and their importance.

Definition 2.3. *A **term** is the smallest distinct unit of text considered.*

Terms can be any units we are interested to index and distinguish. They are usually words, but the information retrieval speaks of terms because some of them, such as "I-9" or "Hong Kong", are not "normal" words, but good units to index.

Definition 2.4. *A **document** is a fixed block of text (terms) in the given dataset. A **query** is a new block of text (usually not in the dataset), but containing a **subset of the same terms** as in the dataset.*

Each document can be a Shakespeare's play, and the query can be for example Romans AND Caesar AND NOT Calpurnia.

The requirement of both documents and queries to be formed from the same global dictionary of terms allows us to treat them as vectors, and hence enable mathematical processing.

Definition 2.5. *Let the dataset contain n terms indexed in order 1 to n. The **vector space model** associates each document or query to a **term-to-document vector** $\mathbf{q} = (q_1, \ldots, q_n) \in \mathbb{R}^n$ such that*

$$q_j = \begin{cases} \text{nonzero}, & \text{if the document contains the } j\text{-th term}, \\ 0, & \text{otherwise}, \end{cases} \qquad j = 1, \ldots, n.$$

The simplest example suitable for the Boolean retrieval is the binary vector space model.

Definition 2.6. *Let the dataset of m documents contain n terms. The* ***binary term-to-document vector*** *associates each document or query to a binary vector* $\mathbf{b} \in \mathbb{R}^n$ *with* $b_j = 1$ *if the document contains the j-th term, and* $b_j = 0$ *otherwise. The* ***binary term-to-document matrix*** *of the dataset* $B \in \mathbb{R}^{m \times n}$ *has the elements* $B_{i,j} = 1$ *if the i-th document contains the j-th term, and* $B_{i,j} = 0$*, otherwise.*

Example 2.2 (b). A part of the term-to-document matrix for Shakespeare's Collected Works is shown in Table 1. To answer the query Romans AND Caesar AND NOT Calpurnia, we

Table 1: A sample of the binary term-to-document matrix for selected Shakespeare's plays and terms.

	Anthony	Brutus	Caesar	Calpurnia	Cleopatra	romans	worser	$\cdots$
Anthony and Cleopatra	1	1	1	0	1	1	1	
Julius Caesar	1	1	1	1	0	1	0	
The Tempest	0	0	0	0	0	0	1	
Hamlet	0	1	1	0	0	1	1	
Othello	0	0	1	0	0	0	1	
Macbeth	1	0	1	0	0	0	0	
$\vdots$								

need to perform the Boolean operations AND and NOT elementwise on the columns of the term-to-document matrix, corresponding to each term. That is, we take the column vectors for Romans, Caesar and Calpurnia, complement the last, and then do an elementwise AND:

$$\begin{pmatrix}1\\1\\0\\1\\0\\0\end{pmatrix} \text{ AND } \begin{pmatrix}1\\1\\0\\1\\1\\1\end{pmatrix} \text{ AND } \begin{pmatrix}1\\0\\1\\1\\1\\1\end{pmatrix} = \begin{pmatrix}1\\0\\0\\1\\0\\0\end{pmatrix}.$$

We see that the result contains 1 (Boolean Yes) in positions 1 and 4, which are the indices of the documents Antony and Cleopatra and Hamlet, respectively.

The boolean retrieval operation simply checks whether each document matches or does not match a query. In the case of large document collections, the resulting number of matching documents can far exceed the number a human user could possibly sift through. Thus, a search engine should *rank-order* the documents matching a query. To do this, the search engine computes, for each matching document, a *score* with respect to the query at hand. A document that mentions a query term more often has more to do with that query and therefore should receive a higher score.

Definition 2.7. *Let the dataset of m documents contain n terms. The* ***term-frequency (TF)*** *term-to-document matrix* $F \in \mathbb{R}^{m \times n}$ *contains elements* $F_{i,j}$ *equal to the number of occurrences of the term j in the document i.*

Note that the i-th row of F satisfies Definition 2.5 since $F_{i,j} = 0$ means zero occurrences, i.e. the term is not contained in the document.

Remark 2.8. *We are still ignoring the exact ordering of terms in a document. This can be partially compensated by taking several words into one term (so words in different order are different terms) or giving fractional values of $F_{i,j}$ depending on the word's place in the document, so for simplicity we proceed with such unordered **bag of words** model of documents.*

The Python package `scikit-learn` (or shortly `sklearn`) contains a class `CountVectorizer` that can index a list of documents, collect the terms, and produce either binary or TF term-to-document matrix. Since `CountVectorizer` is a class (as most models available in `sklearn`), the usual workflow starts as follows:

```
model = CountVectorizer(param1=value1,...)  # Construct an object
F = model.fit_transform(data)  # Index documents and output the Term-to-document matrix
terms = model.get_feature_names_out() # A list of all terms
query_vector = model.transform(query_text) # Convert a query into the same vector space
```

2.2.2 Inverse document frequency weighting

One simple improvement upon Remark 2.8 we consider here is to discount terms that occur **too** often in many documents, and may thus be of limited use for scoring the relevance. For instance, a collection of documents on the auto industry is likely to have the term auto in almost every document.

Definition 2.9. *Let the dataset of m documents contain n terms.*

- *The **inverse document frequency (IDF)** vector $\mathbf{f}^{(IDF)} \in \mathbb{R}^n$ is defined elementwise as*

$$f_j^{(IDF)} = \log \frac{m}{f_j}, \quad j = 1, \dots, n, \tag{2.2}$$

  where $f_j = \sum_{i=1}^m B_{i,j}$ is the frequency of the term in all documents (and B is the binary term-to-document matrix).

- *The **TF·IDF** term-to-document matrix $F^{(TF\cdot IDF)} \in \mathbb{R}^{m \times n}$ is defined elementwise as*

$$F_{i,j}^{(TF\cdot IDF)} = F_{i,j} \cdot f_j^{(IDF)} = F_{i,j} \log \frac{m}{f_j}.$$

Note how the columns of the previous TF matrix are **weighted** with the inverse document frequency. In particular, a term contained in all m documents has $f_j = m$, and hence $f_j^{(IDF)} = \log(m/m) = 0$, which zeroes also the corresponding column of the TF·IDF matrix. In contrast, rare terms have high IDF.

The `sklearn` class `TfidfVectorizer` can be used to index documents and produce directly the TF·IDF term-to-document matrix. The workflow is identical to that of `CountVectorizer` except the name of the class. However, to prevent divisions by zero if some terms are not contained in any document, `TfidfVectorizer` adds ones in the IDF, as if an extra document was seen containing every term in the collection exactly once:

$$f_j^{(IDF_{a1})} = \log \frac{m+1}{f_j+1} + 1.$$

We call this definition the **add-one** smoothed IDF to distinguish it from the **traditional IDF** in Definition 2.9. In addition, `TfidfVectorizer` **normalises** all term-to-document vectors to make their norm 1. The corresponding term-to-document matrix reads

$$F_{i,j}^{(TF\cdot IDF_{a1})} = \frac{F_{i,j} \cdot f_j^{(IDF_{a1})}}{z_i}, \quad \text{where} \quad z_i^2 = \sum_{j=1}^{n} \left(F_{i,j} \cdot f_j^{(IDF_{a1})} \right)^2.$$

Since log() is a monotone function, the two IDF definitions have similar weighting effects.

Here are example values (using the base-10 logarithm) of the term frequency, document frequency, IDF and TF·IDF for words "try" and "insurance" in the Reuters collection of $m = 806791$ documents on the auto industry.

	$\sum_i F_{i,j}$	f_j	$f_j^{(IDF)}$	$\sum_i F_{i,j}^{(TF\cdot IDF)}$
try	10422	8760	1.96	20471
insurance	10440	3997	2.31	24064

It is intuitive that the word "try" is not very helpful for a query, while "insurance" may be. We see that while TF values are almost identical, IDF and TF·IDF assign a *significantly* higher score to "insurance".

This example shows that **metrics** on data are important, and we cannot just always take the listing index for a numerical encoding of non-numerical data.

2.3 Metrics and scores on data (examinable)

2.3.1 Vector distance

The most simple scenario is when the data is already numerical. In this case any data can be reshaped into a vector as we have demonstrated in Section 2.1.1, and measured using classical vector norms. Recall the standard norms for a vector $\mathbf{x} \in \mathbb{R}^n$,

- 2-norm, also known as Euclidean norm, reads $\|\mathbf{x}\|_2 := (x_1^2 + \cdots + x_n^2)^{1/2}$;
- 1-norm reads $\|\mathbf{x}\|_1 := |x_1| + \cdots + |x_n|$;
- ∞-norm, or supremum norm reads $\|\mathbf{x}\|_\infty := \max_{i=1,\ldots,n} |x_i|$.

A natural vector *distance* function can be defined as a norm of the difference between two vectors, $\|\mathbf{x}-\mathbf{y}\|$. On the other hand, matrix norms of the form $\|A\| = \sup_{\|x\|=1} \|A\mathbf{x}\|$, traditionally defined in numerical linear algebra, are less common in machine learning. An Occam's razor reason for this is that the main distinguishing property of matrix norms is the multiplication inequality, $\|AB\| \le \|A\|\|B\|$, which is not utilized much in machine learning.

Another reason for preferring simple norms and distance functions is the computational simplicity. For example, computation of the spectral matrix norm requires the solution of an eigenvalue problem – which can be slow.

Moreover, *separable* distance functions such as

$$\|\mathbf{x} - \mathbf{y}\|_2^2 \qquad \text{or} \qquad \|\mathbf{x} - \mathbf{y}\|_1$$

are especially preferred since they break down to pointwise loss functions $(x_j - y_j)^2$ or $|x_j - y_j|$, and thus critical for the possibility of empirical risk minimisation and cross validation.

Note that the loss function in the polynomial regression example presented in Section 1.2.2 can be also written using the vector norm,

$$L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m}\|h_{\boldsymbol{\theta}}(\mathbf{X}) - \mathbf{y}\|_2^2,$$

if by $h_{\boldsymbol{\theta}}(\mathbf{X})$ we mean the vector of values of all $h_{\boldsymbol{\theta}}(\mathbf{x}_i)$.

2.3.2 Angle distance and cosine similarity score

Recall that in two- or three-dimensional space, a cosine of the angle between two vectors can be computed as

$$\cos\angle(\mathbf{x}, \mathbf{y}) = \frac{x_1 y_1 + \cdots + x_n y_n}{\sqrt{x_1^2 + \cdots x_n^2} \cdot \sqrt{y_1^2 + \cdots + y_n^2}},$$

see Figure 5. However, this formula can be extended to arbitrary dimension n, and the cosine

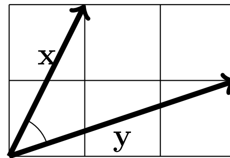


Figure 5: Cosine of angle between vectors is their relative inner product.

of the angle between vectors is defined as

$$\cos\angle(\mathbf{x}, \mathbf{y}) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|_2 \cdot \|\mathbf{y}\|_2} = \left\langle \frac{\mathbf{x}}{\|\mathbf{x}\|_2}, \frac{\mathbf{y}}{\|\mathbf{y}\|_2} \right\rangle, \tag{2.3}$$

for any valid definition of the inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ and the corresponding Euclidean norm $\|\mathbf{x}\|_2 = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$. In principle, we can take the absolute angle $|\angle(\mathbf{x}, \mathbf{y})|$ as a distance function between any vector data. However, its *insensitivity to scale* of the data makes it especially convenient in information retrieval problems.

End of lecture 4

2.3.3 Cosine similarity scoring of documents (non-examinable)

In addition to a term-to-document matrix, we can also encode *queries* as vectors. This uses the same model: a query is associated to a vector $\mathbf{q} \in \mathbb{R}^m$ with elements $q_j = 1$ if the jth term is contained in the query, and 0, otherwise. We can notice that in this model a query is just another document, which lives in the same vector space as the original document collection. Indeed, each document is associated with a column of the term-to-document matrix, defined in a chosen way, be it $B_i \in \mathbb{R}^m$, $F_i \in \mathbb{R}^m$ or $F_i^{(TF \cdot IDF)} \in \mathbb{R}^m$. The only assumption we make is that the dictionary (indexation) of the m terms is the same for all documents and queries.

So, given that a text query is just another term-to-document vector, $\mathbf{q} \in \mathbb{R}^n$, how do we quantify the similarity between two documents?

A first attempt might consider the norm of distance between two document vectors. This measure suffers from a drawback: two documents with very similar content can have a significant norm of distance simply because one document is much longer than the other. Thus the relative distributions of terms may be identical in the two documents, but the absolute term frequencies of one may be far larger.

It is the angle distance that allows one to compensate for the effect of document length. Note from the last term of (2.3) that the cosine of the angle between vectors depends only on the *normalised* vectors. Since the information retrieval concerns only a comparison of document scores (instead of their particular values), we can save on computing the arccos to get the actual angle between two vectors, and compare directly cosines of the angles, higher values of which correspond to more similarity between the vectors. This procedure is summarized in Algorithm 2.

In a certain sense, the vector space model of the information retrieval is similar to supervised learning: given a collection of documents (training data), we would like to predict the index of a query (test data). A significant difference though is that the information retrieval requires only a fixed number of explicit computations of boolean or vector operations, whereas the training in the supervised learning involves solving a potentially complicated optimisation problem.

Example 2.2 (c). To apply the vector space scoring to Shakespeare's plays with the binary term-to-document matrix shown in Table 1, and a query "Romans, Caesar, but NOT Calpurnia", we can assemble the query vector

$$\mathbf{q} = (0, 0, 1, -1, 0, 1, 0),$$

which gives the following cosine scores:

Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	$\cdots$
0.47	0.26	0	0.58	0.41	0.41	

This points to "Hamlet" as the best match, and "Anthony and Cleopatra" as the second best match. Note that this difference is solely due to a larger norm of $\mathbf{f}_1$ in the denominator of the cosine score due to more indexed words appearing in "Anthony and Cleopatra". The inner products $\langle \mathbf{q}, \mathbf{f}_1 \rangle = \langle \mathbf{q}, \mathbf{f}_4 \rangle = 2$ are the same. This may sound artificial, but recall that we did want to prioritise documents, in which the query terms are somewhat special, compared to documents that just contain everything.

Algorithm 2 Vector space document scoring

1: Index all n terms appearing in all given documents and possible queries.

2: Choose the form (weighting) of the term-to-document matrix: B, F or $F^{(TF \cdot IDF)}$, and let $\mathbf{f}_i \in \mathbb{R}^n$ be the i-th row of the chosen term-to-document matrix, $i = 1, \ldots, m$.

3: Given a query, encode it in the same vector space by assembling the vector $\mathbf{q} \in \mathbb{R}^n$ with a chosen nonzero weight q_j if the j-th term is contained in the query, and 0, otherwise.

4: Compute *cosine scores* of all documents,

$$\cos \angle(\mathbf{q}, \mathbf{f}_i) = \frac{\langle \mathbf{q}, \mathbf{f}_i \rangle}{\|\mathbf{q}\|_2 \cdot \|\mathbf{f}_i\|_2}.$$

5: Select the document with the highest score as the best match:

$$i_* = \arg\min_{i=1,\ldots,m} |\cos \angle(\mathbf{q}, \mathbf{f}_i)|.$$

The `TfidfVectorizer` provides a convenient way to compute cosine similarity scores, since it produces both the term-to-document matrix output of `fit_transform()` and the query

vector output of `transform()` already normalised. It remains to multiply the term-to-document matrix by the query vector to get a vector of all document cosine scores at once.

Summary

- Data can come in various forms, but it should usually be converted to numerical vectors to do any machine learning.
- Text or categorical data require information retrieval: creating a dictionary of terms, and encoding any document or query as a vector of occurrences of these terms.
- All such occurrence vectors form a term-to-document matrix which admits different weightings such as term frequency or inverse document frequency.
- Relevance between documents (queries) can be quantified via a cosine similarity score, which is invariant to the lengths of the documents. The cosine score is directly computable from term-to-document information, without "learning" (i.e. optimisation) as such.

3 Unsupervised learning

Recall that unsupervised learning deals with unlabelled datasets, $\mathbf{D} = \mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. One is interested in deriving some useful information from the domain data points $\mathbf{X}$ themselves, without any training labels or other "true" information provided by a "supervisor", hence the name. Most famous unsupervised learning problems are **clustering** and **dimensionality reduction** (and more general feature selection). Clustering concerns a meaningful grouping of the data points in $\mathbf{X}$ that are similar in some sense. Dimensionality reduction tries to replace the given n-dimensional vectors[3] $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^n$ by lower-dimensional vectors $\mathbf{z}_1, \ldots, \mathbf{z}_m \in \mathbb{R}^r$ one-to-one, with $r < n$ (ideally, $r \ll n$). This can be useful to speed up the computations and take less computer memory, but also because the smaller vectors $\mathbf{z}_1, \ldots, \mathbf{z}_m$ can be more meaningful for humans, and contain less noise.

One can think that these unsupervised learning tasks involve no prediction as such, and are therefore not machine learning tasks at all. However, this boundary is rather vague. Firstly, unsupervised learning does also involve a distance or loss function to measure similarity between data points, or accuracy of their dimension-reduced versions. Instead of labels, such a function is applied to the domain data points, but it may be subject to optimisation quite similarly to supervised learning. Secondly, once the clustering or dimensionality reduction is completed for a given dataset $\mathbf{X}$, a user may receive a new domain vector $\mathbf{x}_{m+1}$ and need to *predict* the best cluster to host $\mathbf{x}_{m+1}$, or an accurate reduced version $\mathbf{z}_{m+1}$ thereof.

3.1 Clustering of data

Clustering is an important practical task to find common groups of genes in biology, customers in marketing, or distribution of materials in a specimen. Yet it is one of the least precise areas of machine learning. One can say that clustering is a grouping of similar objects into the same group and dissimilar objects into different groups. However, these two objectives may contradict, because similarity (or proximity) is not a transitive relation, while cluster sharing is an equivalence relation and, in particular, it is a transitive relation. More concretely, it may be the case that each $\mathbf{x}_i$ is very similar to its two neighbours, $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$, but $\mathbf{x}_1$ and $\mathbf{x}_m$ are very dissimilar. If we wish to make sure that whenever two elements are similar they share the same cluster, then we must put all of the elements of the sequence in the same cluster. However, in that case, we end up with dissimilar elements ($\mathbf{x}_1$ and $\mathbf{x}_m$) sharing a cluster, thus violating the second requirement.

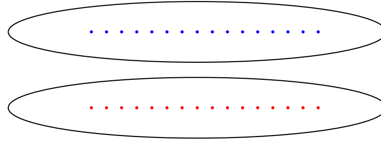
For example, consider clustering the following dots into two clusters.

. . . .

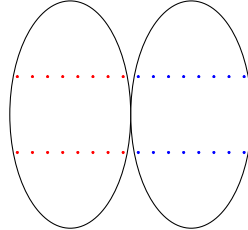
. . . .

Each point is characterised by a vector $\mathbf{x} = (h, v) \in \mathbb{R}^2$ of its horizontal and vertical coordinates, and the natural distance function is $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2$. A clustering algorithm that emphasizes not separating close-by points (e.g., the Single Linkage algorithm shown in Section 3.1.2) will cluster this input by separating it horizontally according to the two lines:

[3] In the previous sections we have seen several ways how non-vector data can be unambiguously turned into vectors. Therefore, we can always assume that domain data points are vectors from now on.



In contrast, a clustering method that emphasizes not having far-away points share the same cluster (e.g., the K-means algorithm shown in Section 3.1.3) will cluster the same input by dividing it vertically into the right-hand half and the left-hand half:



As we mentioned above, the problem of clustering (and unsupervised learning in general) is that there is no labels, which means no "ground truth" that the model can be trained to reproduce. As a result, there is no clear success evaluation procedure for clustering. In fact, even assuming full knowledge of the process generating data, it is not clear what is the "correct" clustering for that data or how to evaluate a proposed clustering. As a result, there is a wide variety of clustering algorithms that, on some input data, will output very different clusterings.

3.1.1 Clustering model

Clustering is characterised by its

Inputs:
- a dataset $\mathbf{X}$;
- a distance function $d : \mathbf{X} \times \mathbf{X} \to \mathbb{R}_+$ that is symmetric, nonnegative, satisfies $d(\mathbf{x}, \mathbf{x}) = 0$ for all $\mathbf{x} \in \mathbf{X}$, and often also satisfies the triangle inequality;
- (optionally) the number of required clusters K;

as well as

Outputs:
- non-overlapping clusters in the form of sets $C_1, \ldots, C_K$ such that $\cup_{k=1}^{K} C_k = \mathbf{X}$ and for all $k \neq j$, $C_k \cap C_j = \emptyset$. Equivalently,
- a vector of indices $\mathbf{k} = (k_1, \ldots, k_m)$, where k_i denotes which cluster the data point $\mathbf{x}_i$ belongs to, $\mathbf{x}_i \in C_{k_i}$, $i = 1, \ldots, m$.
- A clustering *dendrogram*: a tree of set inclusions with the entire dataset as the root, $C_K = \mathbf{X}$, non-overlapping splitting along the branches, $C_i = C_j \cup C_k$ if $\mathrm{Level}(C_j) = \mathrm{Level}(C_k) = \mathrm{Level}(C_i) - 1$, $C_j \cap C_k = \emptyset$ at the same level, and individual data points in the leaves, $C_i = \{\mathbf{x}_i\}$, $\mathrm{Level}(C_i) = 0$, $i = 1, \ldots, m$.

3.1.2 Linkage clustering algorithms

This family of algorithms build a tree of cluster embeddings in a sequential manner. Starting from the trivial clustering that has each data point as a single-point cluster, the algorithm keeps merging the "closest" clusters of the previous clustering until all data points are merged into a single cluster. Recording which clusters were merged in each step allows one to build a clustering dendrogram. However, to produce a meaningful clustering instead of the full dendrogram, we

need to **stop** the algorithm before reaching the root of the tree containing all data points. The precise level in the tree where the algorithm is stopped is a tuneable parameter. Moreover, the "closeness" of clusters need to be measured via a distance function between *clusters*. This function can be produced from the distance function between *points* in several ways.

Definition 3.1. *The* ***single linkage distance*** *between two clusters A and B is the minimal distance between their members,*

$$d(A, B) := \min_{\mathbf{x} \in A, \mathbf{y} \in B} d(\mathbf{x}, \mathbf{y}).$$

Alternatively, *average linkage* clustering defines the cluster distance as the average distance between a point in one of the clusters and a point in the other, and *max linkage* clustering introduces the distance between two clusters as the maximum distance between their elements. The single linkage clustering can be written as shown in Algorithm 3.

Algorithm 3 Single linkage clustering

1: Start with each data point in its own cluster, $C_1 = \{\mathbf{x}_1\}, \dots, C_m = \{\mathbf{x}_m\}$, number of clusters $K = m$, and tree levels $\text{Level}(C_1) = \dots = \text{Level}(C_m) = 0$.
2: **while** $C_K \neq \mathbf{X}$ **do**
3: Find $i_* \neq j_* \in \{1, \dots, K\} : d(C_{i_*}, C_{j_*}) = \min_{i \neq j} d(C_i, C_j)$, $C_i \cap C_j = \emptyset$. ▷ closest non-overlapping clusters
4: Add $C_{K+1} = C_{i_*} \cup C_{j_*}$ to the tree at $\text{Level}(C_{K+1}) = \max(\text{Level}(C_{i_*}), \text{Level}(C_{j_*})) + 1$.
5: Increase $K := K + 1$.
6: **end while**

Python libraries provide two most common linkage clustering implementations: Scipy's `linkage` and sklearn's `AgglomerativeClustering`.

An example set of 5 points and the corresponding single linkage dendrogram are shown in Figure 6.

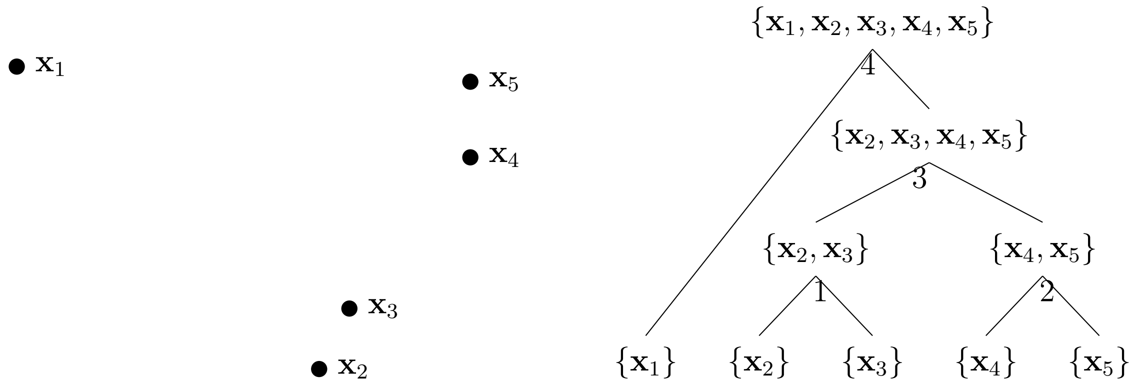


Figure 6: Example dataset $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_5\}$ (left) and its single linkage clustering dendrogram using the natural point distance $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2$ (right). Numbers near branches show the steps (order of merging) in the algorithm.

If one wishes to turn a dendrogram into a partition of the space (a clustering), one needs to employ a stopping criterion. Common stopping criteria include:

- Fixed number of clusters: fix some number $K_* \in \mathbb{N}$ and stop merging clusters as soon as the number of clusters (either in total or in the highest level) is K_*.

- Distance upper bound: fix some $r \in \mathbb{R}$, $r > 0$. Stop merging as soon as all the distances between clusters are larger than r.

These criteria can be selected by optional parameters to `AgglomerativeClustering` directly, or a separate function `fcluster` in the Scipy implementation.

End of lecture 5

3.1.3 K-means loss and K-means algorithm

An alternative approach is to define a *loss* function over a parametrized family of possibly clusterings, and find the optimal clustering by minimizing this loss. Given inputs $\mathbf{X}$ and d and (predicted) clustering outputs $C_1, \ldots, C_K$ (as introduced in Section 3.1.1), the clustering loss is some real-valued function $L(\mathbf{X}, d; C_1, \ldots, C_K)$. One of the most popular functions is the **K-means loss**. Given the clusters $C_1, \ldots, C_K$, each C_i is associated with a *centroid* $\boldsymbol{\mu}_i$, which has a meaning of a *centre of mass*. Note that $\boldsymbol{\mu}_i$ may *not* belong to the original dataset $\mathbf{X}$. It is assumed that the data points $\mathbf{x}_1, \ldots, \mathbf{x}_m$ belong to some larger metric space $\mathcal{X}$ with the same distance function $d(\mathbf{x}, \mathbf{y})$, that is $\mathbf{X} \subset \mathcal{X}$, and centroids belong to $\mathcal{X}$.

Definition 3.2. *Given $\mathbf{X} \subset \mathcal{X}$ and distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_+$, the **K-means loss** is defined as*

$$L(\mathbf{X}, d; C_1, \ldots, C_K) = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} d(\mathbf{x}, \boldsymbol{\mu}_i)^2, \tag{3.1}$$

where $\boldsymbol{\mu}_i$ is the centroid of C_i,

$$\boldsymbol{\mu}_i = \arg\min_{\boldsymbol{\mu} \in \mathcal{X}} \sum_{\mathbf{x} \in C_i} d(\mathbf{x}, \boldsymbol{\mu})^2, \qquad i = 1, \ldots, K. \tag{3.2}$$

Example 3.3. *For real-valued vectors $\mathbf{x}$ of n elements, the natural embedding space is $\mathcal{X} = \mathbb{R}^n$. Moreover, the centroid of a cluster with respect to the natural Euclidean distance function $d(\mathbf{x}, \boldsymbol{\mu}) = \|\mathbf{x} - \boldsymbol{\mu}\|_2$ is indeed the geometric centre (i.e. the mean):*

$$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}, \tag{3.3}$$

where $|C_i|$ is the number of elements in C_i.

Now the optimal clustering can be found from minimisation.

Definition 3.4. *The optimal K-means clustering $C_1^*, \ldots, C_K^*$ is that minimising the K-means loss,*

$$L(\mathbf{X}, d; C_1^*, \ldots, C_K^*) = \min_{C_1, \ldots, C_K} L(\mathbf{X}, d; C_1, \ldots, C_K).$$

A practical situation where such an objective makes sense is the facility location problem. Consider the task of locating K fire stations in a city. One can model houses as data points and aim to place the stations so as to minimise the average squared distance between a house and its closest fire station.

One may think that the clustering result is just the minimiser of (3.1), and should be independent of a particular optimisation algorithm. However, it turns out that finding the optimal K-means clustering for large data is computationally infeasible. Problems of such kind are called *NP-hard*, since the guaranteed optimal solution requires a Non-Polynomial number of operations, for example, comparison of loss values of all possible clusterings. The number of those is exponential in K, hence the name Non-Polynomial (NP). In practice, this problem is bypassed by taking a *sub-optimal* clustering, which can be produced by a feasible algorithm, as a satisfactory result.

The mostly used practical **K-means algorithm** is shown in Algorithm 4. It performs an *alternating iteration* over the optimisation problems (3.2) and (3.1): firstly the cluster assignments are fixed, and centroids are computed from (3.2), followed by fixing the centroids and computing (new) cluster assignments from (3.1), and repeating until convergence. This algorithm is implemented in sklearn's `KMeans` class.

Algorithm 4 K-means algorithm

Require: $\mathbf{X}$, number of clusters K, distance function d, maximal number of iterations T.
Ensure: Partitioning of $\mathbf{X}$ into cluster sets $C_1, \ldots, C_K$.

1: Randomly choose initial centroids $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K \in \mathcal{X}$, iteration number $t = 0$.
2: **while** not converged or $t < T$ **do**
3:     **for** $i = 1, \ldots, K$ **do**
4:         Put points near ith centroid into ith cluster: $C_i = \{\mathbf{x} \in \mathbf{X} : \arg\min_{j=1,\ldots,K} d(\mathbf{x}, \boldsymbol{\mu}_j) = i\}$
5:     **end for**
6:     **for** $i = 1, \ldots, K$ **do**
7:         Recompute the centroids as shown in (3.2). ▷ Or (3.3) if $d(\mathbf{x}, \mathbf{y})$ is Euclidean.
8:     **end for**
9:     $t = t + 1$.
10: **end while**

Although it is not too difficult to find a counterexample when the K-means algorithm is not optimal, in many practical scenarios this algorithm gives a meaningful clustering, to an extent that often the term *K-means clustering* refers to the outcome of this algorithm rather than to the clustering that minimises the K-means loss. A more rigorous reasoning for this is that, despite that the K-means algorithm may not give a *globally* optimal clustering, it does converge to a *locally* optimal clustering.

Theorem 3.5. *Each iteration of Algorithm 4 does not increase the K-means loss* (3.1).

Proof. Consider the update at iteration t of the K-means algorithm. Let $C_1^{(t-1)}, \ldots, C_K^{(t-1)}$ be the clustering on the previous iteration, and let $C_1^{(t)}, \ldots, C_K^{(t)}$ be the clustering in the end of the current iteration, and similarly let $\boldsymbol{\mu}_i^{(t-1)}$ and $\boldsymbol{\mu}_i^{(t)}$ denote ith centroids on the previous and current iterations, respectively. Consider the last step of the algorithm (update of the centroids). Since

$$\boldsymbol{\mu}_i^{(t)} = \arg\min_{\boldsymbol{\mu} \in \mathcal{X}} \sum_{\mathbf{x} \in C_i^{(t)}} d(\mathbf{x}, \boldsymbol{\mu})^2$$

for each i, we get for the total loss function that

$$L(\mathbf{X}, d; C_1^{(t)}, \ldots, C_K^{(t)}) = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i^{(t)}} d(\mathbf{x}, \boldsymbol{\mu}_i^{(t)})^2 \leq \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i^{(t)}} d(\mathbf{x}, \boldsymbol{\mu}_i^{(t-1)})^2. \tag{3.4}$$

Minimisations in the first step of the algorithm (population of the clusters with the points closest to their centroids) implies also that total expression

$$\sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} d(\mathbf{x}, \boldsymbol{\mu}_i^{(t-1)})^2$$

is minimised over all possible partitions $C_1, \ldots, C_K$. Hence,

$$\sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i^{(t)}} d(\mathbf{x}, \boldsymbol{\mu}_i^{(t-1)})^2 \leq \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i^{(t-1)}} d(\mathbf{x}, \boldsymbol{\mu}_i^{(t-1)})^2. \tag{3.5}$$

Note that the right hand side in (3.5) is the loss in the previous iteration, $L(\mathbf{X}, d; C_1^{(t-1)}, \ldots, C_K^{(t-1)})$. Plugging (3.5) into (3.4), we obtain that

$$L(\mathbf{X}, d; C_1^{(t)}, \ldots, C_K^{(t)}) \leq L(\mathbf{X}, d; C_1^{(t-1)}, \ldots, C_K^{(t-1)}),$$

which is the claim of the theorem. □

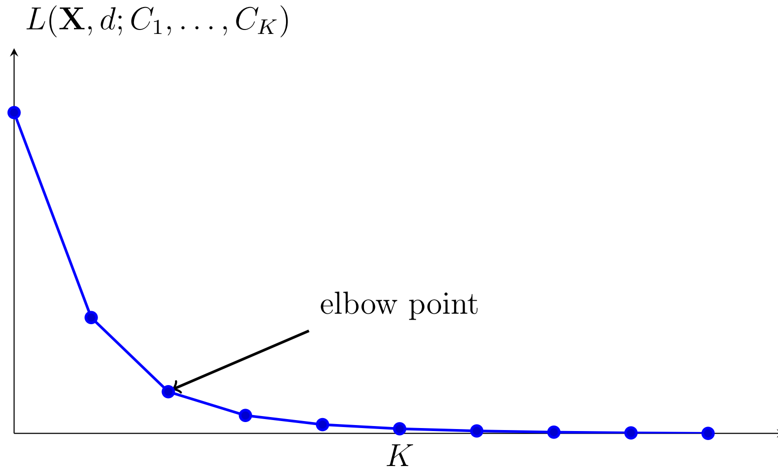
Recall that a bounded monotone sequence is convergent. Thus, Theorem 3.5 implies that the sequence $L(\mathbf{X}, d; C_1^{(t)}, \ldots, C_K^{(t)})$, $t = 0, 1, \ldots$ converges (the loss is always bounded from below by 0). This allows us to stop the **while** loop in Algorithm 4 when the difference between the K-means loss values in several consecutive iterations is below a chosen threshold.

While this theorem tells us that the K-means loss is monotonically nonincreasing in the K-means algorithm, there is no guarantee on the number of iterations needed to reach convergence. Furthermore, there is no nontrivial lower bound on the gap between the value of the K-means loss of the algorithm's output and the global minimum of the K-means loss. To improve the results of the K-means algorithm it is often recommended to repeat the procedure several times with different randomly chosen initial centroids. For example, we can choose the initial centroids to be random points from the data.

3.1.4 Choosing the number of clusters

Clustering is perhaps one of the least precise tasks in machine learning. The two examples in the beginning of Section 3.1 show that there may be different equally valid clusterings, depending on our assumptions. Some clustering algorithms (e.g. K-means), have the total loss (empirical risk) function such as (3.1). However, the cross validation technique is not applicable for clustering: if we exclude some data into a test set, the meaning of clusters may change, and hence there is no way to assess the correctness of clusters computed from the training dataset.

One approach to choose the number of clusters (the parameter not directly optimisable) is to select the so-called **elbow point** of the loss, plotted as a function of the number of clusters.



Conceptually, the elbow point indicates where the loss value *stabilises*, and does not decrease *much* further with increasing the number of clusters. Note though that the elbow point of a *discrete*[4] function may be difficult to identify.

A somewhat more reliable measure is the stabilisation of clusters themselves with respect to some **clustering score function**. Although we cannot do the cross validation, recall that the K-means algorithm is nonetheless randomised due to the random selection of the initial centroids. We can perform several **restarts** of the algorithm from different initial centroids and at different numbers of clusters, and choose the clustering (and the number of clusters) that gives the best score.

3.1.5 Silhouette Coefficient: a score of clustering outliers

This is a score function to indicate how well separated each cluster is from the other clusters. We introduce:

- $d_{in}(\mathbf{z}) = \frac{1}{|C_i|-1} \sum_{\mathbf{x} \in C_i, \mathbf{x} \neq \mathbf{z}} d(\mathbf{z}, \mathbf{x})$, where $i : \mathbf{z} \in C_i$: the average distance between a sample $\mathbf{z}$ and all other points in the **same** cluster C_i.
- $d_{out}(\mathbf{z}) = \frac{1}{|C_j|} \sum_{\mathbf{x} \in C_j} d(\mathbf{z}, \mathbf{x})$, where $j = \underset{k=1,\ldots,K,\, k \neq i}{\arg\min} \frac{1}{|C_k|} \sum_{\mathbf{x} \in C_k} d(\mathbf{z}, \mathbf{x})$: the average distance between a sample $\mathbf{z}$ and all other points in the **next nearest** cluster C_j.

Definition 3.6. *The Silhouette Coefficient of a* ***point*** $\mathbf{z} \in C_i$ *is defined as follows:*

$$s(\mathbf{z}) = \frac{d_{out}(\mathbf{z}) - d_{in}(\mathbf{z})}{\max\{d_{out}(\mathbf{z}), d_{in}(\mathbf{z})\}},$$

and the Silhouette Coefficient of the entire ***clustering*** *is the average Silhouette Coefficient over all points in the dataset,* $s(C_1, \ldots, C_K) = \frac{1}{|\mathbf{X}|} \sum_{\mathbf{z} \in \mathbf{X}} s(\mathbf{z})$.

The Silhouette Coefficients of points can be computed by `sklearn.metrics.silhouette_samples` in sklearn, and the average of those (the Silhouette Coefficient of the clustering) can be computed directly by `sklearn.metrics.silhouette_score`.

[4]For continuous parameters, one can use the second derivative of the loss function to find the point of the maximum curvature. However, there is no such analogue of a "derivative" for a function in a discrete parameter, such as the number of clusters.

If the clusters are well separated from each other, the distance between different clusters is much larger than the distance between points within the same cluster, $d_{out}(\mathbf{z}) \gg d_{in}(\mathbf{z})$, and hence $s(\mathbf{z}) \approx 1$. In contrast, if $\mathbf{z}$ is assigned a wrong cluster, it may have $d_{out}(\mathbf{z}) < d_{in}(\mathbf{z})$, and hence a negative Silhouette Coefficient.

To select the most reliable clustering we can run the K-means algorithm several times from different random initial centroids, and choose the clustering (and the number of clusters) with the largest Silhouette Coefficients. We can then compare the outputs using some score function $S(C_1, \ldots, C_K; C_1^*, \ldots, C_K^*)$ between the clusterings as a whole. The number of clusters can then be selected as the largest that gives an average score between all pairs of outputs above a chosen threshold. The pseudocode of this procedure is shown in Algorithm 5, and an example of a clustering score function is shown in the next subsection.

Algorithm 5 Restarted K-means algorithm with the number of clusters adaptation

Require: Data $\mathbf{X}$, pointwise distance function d, maximal number of K-means iterations T, number of restarts N, clustering score function S, score threshold $s_{\min} > 0$, maximal K.

Ensure: Clustering $C_1, \ldots, C_K$.

1: Start with maximal K, average score $s = 0$.
2: **while** $s < s_{\min}$ **do**
3: Decrease the number of clusters, $K = K - 1$
4: **for** $i = 1, \ldots, N$ **do** ▷ restarts
5: Run K-means (Algorithm 4) to get $C_1^{(i)}, \ldots, C_K^{(i)} = \texttt{KMeans}(\mathbf{X}, K, d, T)$.
6: **end for**
7: Compute the pairwise average score $s = \frac{2}{N(N-1)} \sum_{i<j} S(C_1^{(i)}, \ldots, C_K^{(i)}; C_1^{(j)}, \ldots, C_K^{(j)})$.
8: **end while**
9: **return** $C_1^{(1)}, \ldots, C_K^{(1)}$.

3.1.6 Rand index: a similarity score of two clusterings

This score function measures the similarity between two clusterings, $C_1, \ldots, C_K$ and $C_1^*, \ldots, C_K^*$. It is most useful when some *reference* $C_1^*, \ldots, C_K^*$ is known, but one can take a clustering with the maximum silhouette as such.

We begin by introducing two numbers:

- N_{same}: the number of pairs $i, j = 1, \ldots, m$, $i < j$, such that $\mathbf{x}_i$ and $\mathbf{x}_j$ share the same cluster in both clusterings:
$$\{\mathbf{x}_i, \mathbf{x}_j\} \in C_k \qquad \Leftrightarrow \qquad \{\mathbf{x}_i, \mathbf{x}_j\} \subset C_{k_*}^*.$$
- N_{diff}: the number of pairs $i, j = 1, \ldots, m$, $i < j$, such that $\mathbf{x}_i$ and $\mathbf{x}_j$ are in different clusters in both clusterings:
$$\mathbf{x}_i \in C_p,\ \mathbf{x}_j \in C_q \quad \text{with} \quad p \neq q \qquad \Leftrightarrow \qquad \mathbf{x}_i \in C_{p_*}^*,\ \mathbf{x}_j \in C_{q_*}^* \quad \text{with} \quad p_* \neq q_*.$$

In other words, N_{same} and N_{diff} are the numbers of pairs clustered consistently in the two clusterings. Now we can introduce

Definition 3.7. *The* ***Rand index*** *of clusterings $C_1, \ldots, C_K$ and $C_1^*, \ldots, C_K^*$ is defined as the ratio of the number of agreeing pairs and the total number of pairs,*
$$\mathrm{RI}(C_1, \ldots, C_K; C_1^*, \ldots, C_K^*) = \frac{N_{same} + N_{diff}}{N_{total}}, \qquad N_{total} = \binom{m}{2} = \frac{m(m-1)}{2}.$$

The Rand index is bounded, $0 \leq \mathrm{RI}(C_1, \ldots, C_K; C_1^*, \ldots, C_K^*) \leq 1$, since in addition to N_{same} and N_{diff}, there can also be pairs that are in the *same* cluster C_k, but in *different* clusters $C_{p_*}^*, C_{q_*}^*$ (and vice versa). Perfectly matching clusterings have $\mathrm{RI} = 1$.

Note that cluster labels in definitions of N_{same} and N_{diff} can be different, for example, $k \neq k_*$. Indeed, the ordering of cluster labels is artificial, and different algorithms (or even different restarts of the same algorithm) may end up with different labels for the same data points. Therefore, it is crucial to consider pairs of points to make the Rand index score insensitive to label swaps.

To compute the Rand index in Python, we can use the function `sklearn.metrics.rand_score` from the `sklearn` library.

End of lecture 6

3.2 Principal Component Analysis for dimensionality reduction

Dimensionality reduction is the process of **lossy compression** of data in a high dimensional space by mapping it into a new space whose dimensionality is much smaller. There are several reasons to reduce the dimensionality of the data.

- Reduce computing time and memory requirement.
- Learning in high dimension might lead to poor prediction.
  For instance, recall taking an unnecessarily high polynomial degree in the example in Section 1.2.2 which leads to overfitting and actually worse prediction accuracy.
- Interpretability of the data and visualisation.

One of the simplest options for dimensionality reduction is a *linear* transformation to the original data. That is, if the original data $\mathbf{X} \subset \mathbb{R}^n$, we aim at compressing it into a lower-dimensional space $\mathbb{R}^r$ with $r < n$ (ideally $r \ll n$) by finding a matrix $W \in \mathbb{R}^{r \times n}$ such that for any $\mathbf{x} \in \mathbf{X}$ its reduced version reads $\mathbf{z} = W\mathbf{x} \in \mathbb{R}^r$. Many computations can be performed with the reduced vectors $\mathbf{z}$ directly. However, to *recover* (more precisely, approximate) the original data $\tilde{\mathbf{x}} \approx \mathbf{x}$, we need another matrix $U \in \mathbb{R}^{n \times r}$ such that $\tilde{\mathbf{x}} = U\mathbf{z}$.

Definition 3.8. *Given a vector dataset $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \subset \mathbb{R}^n$, the* ***Principal Component Analysis (PCA)*** *finds the compression matrix $W_* \in \mathbb{R}^{r \times n}$ and the recovery matrix $U_* \in \mathbb{R}^{n \times r}$ such that the Mean Squared Error between the recovered and original vectors in $\mathbf{X}$ is minimal,*

$$W_*, U_* = \operatorname*{arg\,min}_{W \in \mathbb{R}^{r \times n}, U \in \mathbb{R}^{n \times r}} \frac{1}{m} \sum_{i=1}^{m} \|\mathbf{x}_i - UW\mathbf{x}_i\|_2^2. \tag{3.6}$$

Columns of U_* are called ***Principal Components****.*

To describe the solution more comprehensively, let us first show that we actually need to find only one matrix instead of two.

Lemma 3.9. *Let W_*, U_* be a solution to (3.6). Then the columns of U_* can be chosen orthonormal (that is, $U_*^\top U_* = I$, the $r \times r$ identity matrix), and $W_* = U_*^\top$.*

Proof. Note that $\mathcal{R} = \{UW\mathbf{x} : \mathbf{x} \in \mathbb{R}^n\}$ is a linear subspace. Let $V \in \mathbb{R}^{n \times r}$ be a matrix whose columns form an orthonormal basis of this subspace, that is, the range of V is $\mathcal{R}$ and $V^\top V = I$. Note that the dimension of $\mathcal{R}$ is at most r, since it belongs to a (possibly larger) r-dimensional

subspace $\hat{\mathcal{R}} = \{U\mathbf{z} : \mathbf{z} \in \mathbb{R}^r\}$, so the size of the matrix V is enough to span $\mathcal{R}$. Now, each vector in $\mathcal{R}$ can be written as $V\mathbf{q}$ with some $\mathbf{q} \in \mathbb{R}^r$. For any $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{q} \in \mathbb{R}^r$ we have

$$\|\mathbf{x}-V\mathbf{q}\|_2^2 = \|\mathbf{x}\|_2^2 - 2\langle V\mathbf{q}, \mathbf{x}\rangle + \langle V\mathbf{q}, V\mathbf{q}\rangle = \|\mathbf{x}\|_2^2 - 2\langle \mathbf{q}, V^\top \mathbf{x}\rangle + \langle \mathbf{q}, \underbrace{V^\top V}_{I}\mathbf{q}\rangle = \|\mathbf{x}\|_2^2 + \|\mathbf{q}\|_2^2 - 2\langle \mathbf{q}, V^\top \mathbf{x}\rangle.$$

Differentiating this expression in $\mathbf{q}$ and taking the gradient to zero, we obtain the minimiser

$$\mathbf{q}_* = \arg\min_{\mathbf{q}\in\mathbb{R}^r} \|\mathbf{x} - V\mathbf{q}\|_2^2 = V^\top \mathbf{x}.$$

The recovery vector belongs to $\mathcal{R}$, hence it can also be written as $\tilde{\mathbf{x}} = V\mathbf{q}$, and thus

$$\arg\min_{\tilde{\mathbf{x}}\in\mathcal{R}} \|\mathbf{x} - \tilde{\mathbf{x}}\|_2^2 = VV^\top \mathbf{x}.$$

This hold in particular for our dataset $\mathbf{X}$, so for any $\mathbf{x}_i$,

$$\|\mathbf{x}_i - \tilde{\mathbf{x}}_i\|_2^2 \geq \|\mathbf{x}_i - VV^\top \mathbf{x}_i\|_2^2,$$

for any $\tilde{\mathbf{x}}_i \in \mathcal{R}$ including $\tilde{\mathbf{x}}_i = UW\mathbf{x}_i$. Summing the inequality above over i and dividing by m, we obtain

$$\frac{1}{m}\sum_{i=1}^m \|\mathbf{x}_i - UW\mathbf{x}_i\|_2^2 \geq \frac{1}{m}\sum_{i=1}^m \|\mathbf{x}_i - VV^\top \mathbf{x}_i\|_2^2$$

for any $W \in \mathbb{R}^{r\times n}, U \in \mathbb{R}^{n\times r}$, hence $V^\top, V$ is the solution to (3.6). □

Equipped with this lemma, we can turn the original PCA (3.6) into a simpler problem

$$\min_{U\in\mathbb{R}^{n\times r}, U^\top U = I} L_{\mathbf{X}}(U), \qquad L_{\mathbf{X}}(U) := \frac{1}{m}\sum_{i=1}^m \|\mathbf{x}_i - UU^\top \mathbf{x}_i\|_2^2. \tag{3.7}$$

In turn, the empirical risk $L_{\mathbf{X}}(U)$ can be expanded using the *trace* of a matrix.

Definition 3.10. *The* trace *of $A \in \mathbb{R}^{n\times n}$ is defined as the sum of diagonal elements,*

$$\text{trace}(A) = \sum_{i=1}^n A_{i,i}.$$

See also the `numpy.trace` *function in Python.*

Theorem 3.11. *The PCA problem* (3.7) *is equivalent to*

$$\arg\max_{U\in\mathbb{R}^{n\times r}, U^\top U = I} \text{trace}\left(U^\top A U\right), \quad \textit{where} \quad A = \frac{1}{m}\sum_{i=1}^m \mathbf{x}_i \mathbf{x}_i^\top. \tag{3.8}$$

Proof. For any $\mathbf{x} \in \mathbb{R}^n$,

$$\begin{aligned}
\|\mathbf{x} - UU^\top \mathbf{x}\|_2^2 &= \|\mathbf{x}\|_2^2 - 2\langle \mathbf{x}, UU^\top \mathbf{x}\rangle + \langle \mathbf{x}, U\underbrace{U^\top U}_{I} U^\top \mathbf{x}\rangle \\
&= \|\mathbf{x}\|_2^2 - \sum_{i,j=1}^n \sum_{k=1}^r x_i u_{ik} u_{jk} x_j \\
&= \|\mathbf{x}\|_2^2 - \sum_{k=1}^r \left(\sum_{i=1}^n u_{ik} x_i\right)\left(\sum_{j=1}^n x_j u_{jk}\right) \\
&= \|\mathbf{x}\|_2^2 - \sum_{k=1}^r \mathbf{u}_k^\top \mathbf{x}\mathbf{x}^\top \mathbf{u}_k \\
&= \|\mathbf{x}\|_2^2 - \text{trace}(U^\top \mathbf{x}\mathbf{x}^\top U).
\end{aligned}$$

Since the trace is linear in the matrix elements,

$$\frac{1}{m}\sum_{i=1}^{m} \|\mathbf{x}_i - UU^\top \mathbf{x}_i\|_2^2 = \frac{1}{m}\sum_{i=1}^{m} \|\mathbf{x}_i\|_2^2 - \mathrm{trace}(U^\top A U),$$

so the minimisation of the left hand side is equivalent to the maximisation of $\mathrm{trace}(U^\top A U)$. □

The problem (3.8) is solved in Numerical Linear Algebra by the so-called *variational characterisation* of eigenvalues of a symmetric matrix.

Definition 3.12. *A number $\lambda \in \mathbb{C}$ is called an* eigenvalue *of a square matrix $A \in \mathbb{C}^{n\times n}$, and a vector $\mathbf{u} \in \mathbb{C}^n$, $\|\mathbf{u}\|_2 \neq 0$, is called an* eigenvector *corresponding to λ if*

$$A\mathbf{u} = \lambda \mathbf{u}.$$

For a symmetric real matrix $A = A^\top \in \mathbb{R}^{n\times n}$, all eigenvalues and eigenvectors are real-valued. This allows us to define **extreme** eigenvalues $\lambda_{\min}(A)$ and $\lambda_{\max}(A)$ as the minimal and maximal eigenvalues of A, respectively. Numerical Linear Algebra proves the following result.

Theorem 3.13 (Variational characterisation of extreme eigenvalues)**.** *For a symmetric real matrix $A = A^\top \in \mathbb{R}^{n\times n}$, the extreme eigenvalues read*

$$\lambda_{\max}(A) = \max_{\mathbf{x}\in\mathbb{R}^n, \|\mathbf{x}\|_2=1} \langle \mathbf{x}, A\mathbf{x}\rangle = \langle \mathbf{u}_{\max}, A\mathbf{u}_{\max}\rangle, \quad \lambda_{\min}(A) = \min_{\mathbf{x}\in\mathbb{R}^n, \|\mathbf{x}\|_2=1} \langle \mathbf{x}, A\mathbf{x}\rangle = \langle \mathbf{u}_{\min}, A\mathbf{u}_{\min}\rangle,$$

where $\mathbf{u}_{\max}, \mathbf{u}_{\min} \in \mathbb{R}^n$, $\|\mathbf{u}_{\max}\|_2 = \|\mathbf{u}_{\min}\|_2 = 1$ are the eigenvectors corresponding to $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$, respectively.

In addition, if the eigenvalues of the matrix are enumerated in descending order,

$$\lambda_1(A) \geq \lambda_2(A) \geq \cdots \geq \lambda_n(A),$$

and their eigenvectors are $\mathbf{u}_1, \ldots, \mathbf{u}_n$, a more general variational characterisation holds:

$$\lambda_1(A) + \cdots + \lambda_r(A) = \max_{U\in\mathbb{R}^{n\times r}, U^\top U = I} \mathrm{trace}(U^\top A U) = \mathrm{trace}(U_r^\top A U_r),$$

where $U_r = [\mathbf{u}_1, \ldots, \mathbf{u}_r]$, for any $r = 1, \ldots, n$. Therefore, to solve the PCA problem (3.8), it is sufficient to collect r leading eigenvectors of the matrix A as the recovery matrix U, and the compression matrix $U^\top$. In Python, eigenvalues and eigenvectors can be computed using functions `numpy.linalg.eig` (for general matrices), or better specifically `numpy.linalg.eigh` for Hermitian (in the real case, symmetric) matrices. The latter function guarantees that the eigenvalues are real, and the eigenvectors are orthonormal.

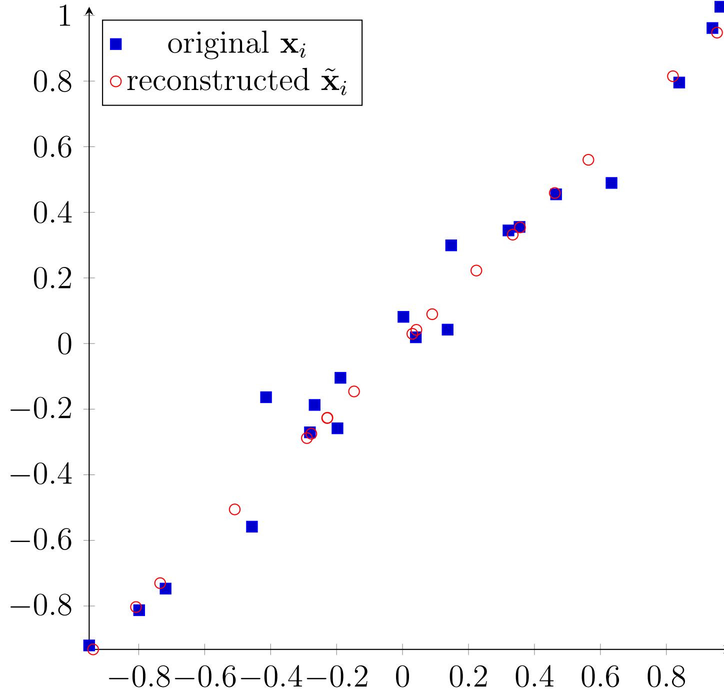
Remark 3.14. *It is a common practice to "center" the data points before applying PCA. That is, we first calculate the average vector $\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^m \mathbf{x}_i$, and then compute the eigenvectors of the matrix*

$$\hat{A} = \frac{1}{m}\sum_{i=1}^{m} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\top$$

instead.

To illustrate how PCA works, let us generate vectors in $\mathbf{R}^2$ that are close to a line, that is a 1-dimensional subspace. Namely, we choose each $\mathbf{x}_i$ in the form $(x, x + y)$, where x is chosen uniformly at random from $[-1, 1]$, and y is sampled from the normal distribution with mean zero and variance 0.01.

If we apply PCA to this data, the leading eigenvector will be close to the vector $(1/\sqrt{2}, 1/\sqrt{2})$. The reduced-dimension data is now a scalar $z = U^\top \mathbf{x}$ which is close to $(2x + y)/\sqrt{2}$, and the reconstructed vector is close to $(x+y/2, x+y/2)$. Due to the small variance of the perturbation y, the reconstructed vectors are close to the original vectors, as can be seen below.



3.3 Example: spectromicroscopy

One real-life example of using both dimensionality reduction and clustering is the X-ray spectromicroscopy. This technique is used in chemistry and engineering to find which materials are contained in a specimen. A spectromicroscopy experiment involves measuring the absorption of the X-ray beam at each of $m = L \times L$ pixels in space at n values of energy ("colour" if this was a visible light) of the incident beam. The data is a matrix of absorption values $X \in \mathbb{R}^{n\times m}$. The X-ray beam of energy $j = 1, \dots, n$, passing through each of r materials, is absorbed proportionally to the thickness of this material. Therefore, each element of the matrix X can be modelled as a noisy linear combination of absorption spectra of pure materials:

$$x_{j,i} = \sum_{k=1}^{r} \hat{u}_{j,k} \hat{z}_{i,k} + \xi_{j,i} \,,$$

where $\hat{u}_{j,k}$ is the absorption of the beam of energy j by 1 nm of the pure material k, $\hat{z}_{i,k}$ is the total thickness (in nm) of the material k at the pixel i, and $\xi_{j,i}$ are independent identically distributed random variables with zero mean, modelling the measurement inaccuracy.

We are interested in identifying which material is located at each pixel by recognising its energy spectrum, that is, the column $\mathbf{x}_i \in \mathbb{R}^n$, $i = 1, \dots, m$. However, we don't know the pure spectra $(\mathbf{u}_k)$ beforehand. We can use clustering (such as K-means) to identify the most representative spectra in each pixel from the given data. However, this may be unreliable due to the perturbations $\xi_{j,i}$. Ideally, we would like to get a matrix

$$\tilde{x}_{j,i} = \sum_{k=1}^{r} u_{j,k} z_{i,k} \,,$$

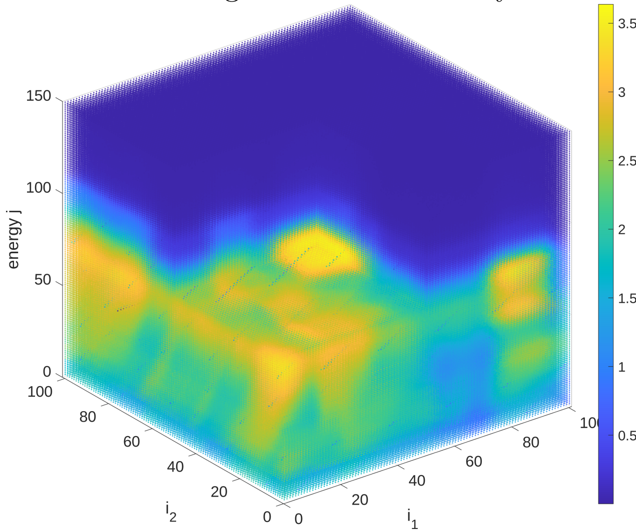
without the noise, but as close as possible to $x_{j,i}$, thus minimizing $\sum_{j,i}(x_{j,i} - \tilde{x}_{j,i})^2$. This is precisely the task solved by PCA, where $\mathbf{u}_1, \dots, \mathbf{u}_r$ are the r leading eigenvectors of $XX^\top$.

We can now apply the K-means algorithm directly to the compressed absorptions at each pixel $\mathbf{z}_i = U_r^\top \mathbf{x}_i$, where $U_r \in \mathbb{R}^{n \times r}$ is a matrix of r leading eigenvectors of $XX^\top$. This is usually both faster (due to a smaller size of $\mathbf{z}_i \in \mathbb{R}^r$ compared to $\mathbf{x}_i \in \mathbb{R}^n$) and more accurate due to suppression of $\xi_{j,i}$ after the projection of X onto the leading eigenvectors U_r.

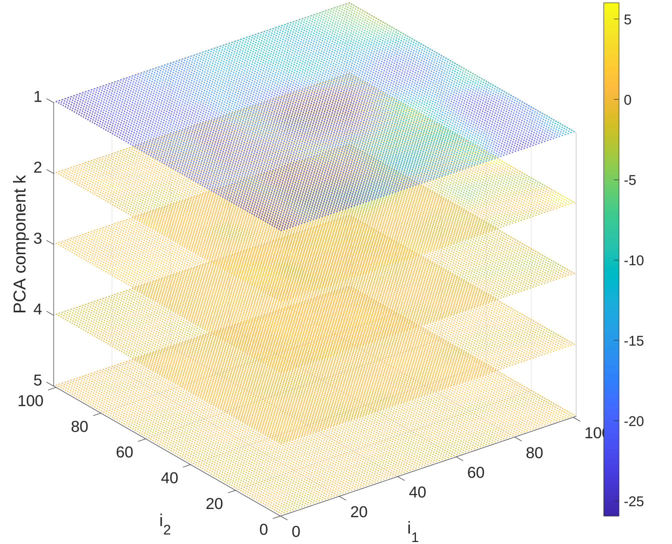
The number of sought clusters should also ideally be the number of materials, $K = N$, although this may vary: we can take $K > N$ if we want to isolate noisy artefacts into separate clusters, or vice versa, $K < N$ can be used to filter out contamination materials.

An example of absorptions and clustering of different iron oxides is shown below. Recall that each pixel i belongs to a $L \times L$ image with the horizontal position $i_1 = 1, \dots, L$, and vertical position $i_2 = 1, \dots, L$, such that $i = (i_1 - 1)L + i_2$, similarly to Eq. (2.1). Here, $L = 101$, the number of energies $n = 149$, the reduced dimension $r = 5$, and we seek $K = 4$ clusters. We can observe that clustering directly the matrix X may miss fine details, for example, in the left and right islands of Cluster 3, whereas clustering the image denoised by PCA is more reliable.

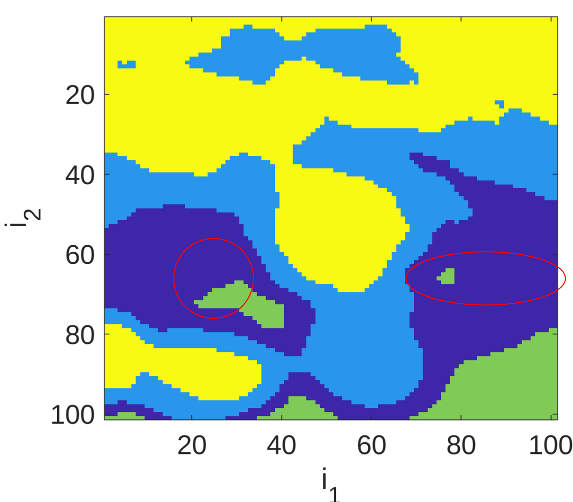
Rows of the absorption matrix X, drawn as 2D images stacked vertically.



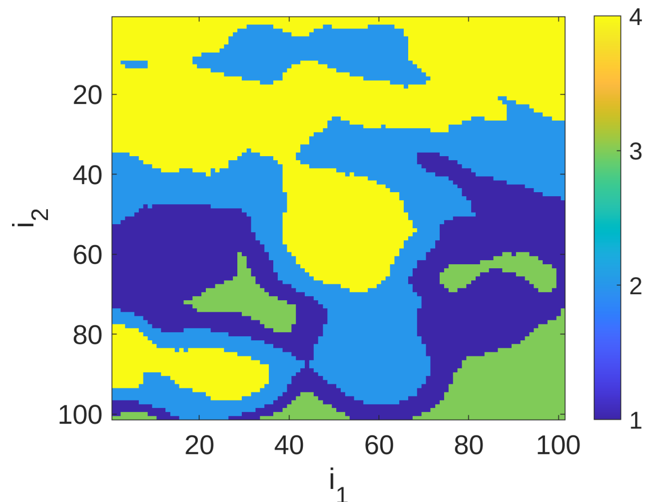
Rows of the PCA matrix $Z = [\mathbf{z}_1, \dots, \mathbf{z}_m]$, drawn as 2D images stacked vertically.



Clustering of columns of X. Colours denote different cluster labels.



Clustering of columns of Z. Colours denote different cluster labels.



End of lecture 7

Summary

- Clustering groups domain data points based on their similarity with respect to the chosen distance function. It can be implemented either by hierarchical grouping (linkage clustering algorithms) or by optimisation of some loss function (K-means algorithm).
- Principal Component Analysis is a linear dimensionality reduction technique that can approximately encode large data vectors by smaller vectors.

4 Supervised learning

In this chapter we are dealing with labelled datasets $\mathbf{D} = (\mathbf{X}, \mathbf{y})$. We are concerned with finding a prediction rule $h(\mathbf{x})$, supervised by the known labels $\mathbf{y}$, such that it can return accurate labels also for $\mathbf{x} \notin \mathbf{X}$. There are two broad subclasses of supervised learning problems: classification and regression.

A classification rule is designed to output one of the few class labels of a data point. Most frequently used is the *binary* classification, where the labels can take only one of two values, 1 or -1. In contrast, a regression rule outputs a prediction that can be any number. We can thus formulate the taxonomy between classification and regression based on the *output domain* as follows.

1. Classification: $y \in \mathcal{Y} = S_K$, where S_K is a set of K values.
   - Binary classification: $y \in \mathcal{Y} = \{-1, 1\}$.
2. Regression (univariate): $y \in \mathcal{Y} = \mathbb{R}$.
   - Regression (multivariate): $\mathbf{y} \in \mathcal{Y} = \mathbb{R}^{n'}$.

Remark 4.1. *Classification may look similar to clustering in the sense that each data point* $\mathbf{x}$ *is assigned a label* y *from a finite set, which can always be an index* $1, \ldots, K$*. The crucial difference is that classification is supervised by a set of known labels, while clustering is not.*

Next, regression and classification may differ in the loss function. For classification, the definition of the loss $\ell(h(\mathbf{x}), y)$ can simply be stating 0 if $h(\mathbf{x})$ predicts y correctly, and 1 otherwise. In regression problems, we would prefer a more accurate definition of the loss. For example, if a baby weights 3 kg, both predictions 3.00001 kg and 4 kg are "wrong", but intuitively we would definitely prefer the former. Therefore, regression loss relies typically on a distance between $h(\mathbf{x})$ and y, for example $\ell(h(\mathbf{x}), y) = \|h(\mathbf{x}) - y\|_2^2$.

4.1 Simple prediction models

Perhaps one of the simplest learnable models is a linear prediction rule, or simply *linear predictor*. They are easy to interpret and analyse, and they are often good enough at generalising simple datasets – or not even so simple ones when using feature mapping, which we will discuss later. We will introduce several subclasses of the linear models, distinguished by regression or classification tasks, as well as the relevant learning algorithms.

4.1.1 Linear functions as prediction rules

The entire family of linear predictors starts with the class of *affine functions.*

Definition 4.2. *Functions of* $\hat{\mathbf{x}} \in \mathbb{R}^n$ *of the form*

$$h_{\hat{\boldsymbol{\theta}},b}(\hat{\mathbf{x}}) = \langle \hat{\boldsymbol{\theta}}, \hat{\mathbf{x}} \rangle + b = \sum_{i=1}^{n} \hat{\theta}_i \hat{x}_i + b,$$

where $\hat{\boldsymbol{\theta}} \in \mathbb{R}^n$ *and* $b \in \mathbb{R}$ *are called* affine*. The set*

$$\mathcal{H}_n^{lin} = \{h_{\hat{\boldsymbol{\theta}},b}(\hat{\mathbf{x}}) : \hat{\boldsymbol{\theta}} \in \mathbb{R}^n, b \in \mathbb{R}\}$$

is called the class of affine functions.

Sometimes it is convenient to incorporate b, called the **bias**, into $\boldsymbol{\theta}$ as an extra coordinate, and expand the domain data points $\hat{\mathbf{x}}$ accordingly,

$$\boldsymbol{\theta} = (b, \hat{\theta}_1, \ldots, \hat{\theta}_n) \in \mathbb{R}^{n+1}, \qquad \mathbf{x} = (1, \hat{x}_1, \ldots, \hat{x}_n) \in \mathbb{R}^{n+1}, \tag{4.1}$$

which allows us to write

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = \langle \boldsymbol{\theta}, \mathbf{x} \rangle. \tag{4.2}$$

Definition 4.3. *Functions of the form* (4.2) *are called* homogeneous linear functions.

4.1.2 Linear regression

It turns out that the simplest supervised learning task is the linear regression, since both parameters and labels consist of real numbers. Both characterisation and optimisation of real-valued predictors can be carried out using elementary Numerical Analysis.

Linear regression is a common statistical tool for modelling the relationship between some "explanatory" variables and some real valued outcome. The hypothesis class of linear regression predictors is simply the set of linear functions $\mathcal{H}_n^{lin}$.

Next, we need to define a loss function for regression. Most often one takes the squared-distance function,

$$\ell(h_{\boldsymbol{\theta}}(\mathbf{x}), y) = (\langle \boldsymbol{\theta}, \mathbf{x} \rangle - y)^2,$$

and the total loss function (empirical risk) is called the Mean Squared Error (MSE):

$$L_{\mathbf{D}}(h_{\boldsymbol{\theta}}) = L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \ell(h_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = \frac{1}{m} \sum_{i=1}^{m} \left[(\langle \boldsymbol{\theta}, \mathbf{x}_i \rangle - y_i)^2 \right].$$

Of course, there are a variety of other loss functions that one can use, for example, the absolute value loss function $\ell(h_{\boldsymbol{\theta}}(\mathbf{x}), y) = |\langle \boldsymbol{\theta}, \mathbf{x} \rangle - y|$. The Mean Absolute Error (MAE) can be minimised by linear programming algorithms. The more simple case of the Mean Squared Error loss will be solved in Section 4.2.3. The minimiser $\boldsymbol{\theta}^*$ of $L_{\mathbf{D}}(\boldsymbol{\theta})$ is also called *Least Squares* solution to the linear regression, since it yields the least squares of the prediction error on average over the training dataset.

4.1.3 Linear regression for Polynomial features

The simple linear prediction rule may be too restrictive for some complicated data. However, we can attempt to change this data in a way that is more suitable for the linear regression.

Recall the polynomial fitting example from Section 1.2.2. The polynomial can be written as a linear prediction rule if we replace x by a vector of monomials of x,

$$h_{\boldsymbol{\theta}}(x) = \langle \boldsymbol{\theta}, \boldsymbol{\psi}(x) \rangle,$$

where

$$\boldsymbol{\psi}(x) = (1, x, \ldots, x^n).$$

Note that the function $\boldsymbol{\psi} : \mathbb{R} \to \mathbb{R}^{n+1}$ is a *transformation* of the original domain point x into a *feature space* $\mathbb{R}^{n+1}$. *Feature design* is used frequently in machine learning. In general, we may be able to find a feature transformation function $\boldsymbol{\psi}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^{n'}$ that yields a simple but accurate prediction rule (for example, $h_{\boldsymbol{\theta}}(\boldsymbol{\psi}(\mathbf{x}))$) in the feature space. It can be very effective if

some initial knowledge of the data is known (for example, we assume that it is a polynomial). The same MSE loss can be used for optimising the coefficients $\boldsymbol{\theta}$, with the obvious modification

$$L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left[(\langle \boldsymbol{\theta}, \boldsymbol{\psi}(x_i) \rangle - y_i)^2 \right].$$

4.1.4 Halfspaces binary classifier

Different supervised learning tasks may involve different label spaces, and therefore we typically not use the linear predictors directly, but pass their output through another function $\phi : \mathbb{R} \to \mathcal{Y}$, where $\mathcal{Y}$ is a set (or space) of possible labels. We can call ϕ a feature transformation as well, but now it acts on the output instead of the input. The first class of linear prediction rules we consider is the class of **halfspaces**, designed for binary classification problems, where $\hat{\mathbf{X}} \in \mathbb{R}^n$, and $\mathcal{Y} = \{-1, +1\}$.

Definition 4.4. *The set of binary classification prediction rules of the form*

$$\mathcal{H}_n^{hs} = \{\text{sign}(h_{\hat{\boldsymbol{\theta}},b}(\hat{\mathbf{x}})) : h_{\hat{\boldsymbol{\theta}},b} \in \mathcal{H}_n^{lin}\}$$

*is called the class of **halfspaces**.*

In other terms, here $\phi(z) = \text{sign}(z)$, which turns any real output of the linear predictor into an unambiguous label -1 or $+1$.

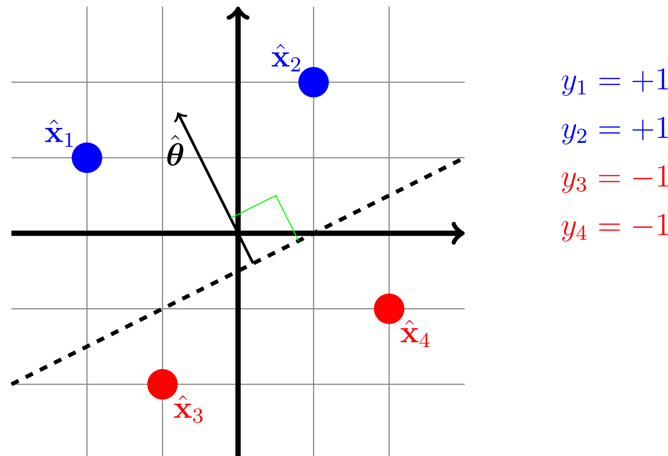
Definition 4.5. *A dataset* $\mathbf{D} = (\hat{\mathbf{X}}, \mathbf{y})$ *labelled with two classes* -1 *and* $+1$ *is called **separable** (in halfspaces) if there exists a hyperplane separating all the positive examples from all the negative examples:*

$$\exists \hat{\boldsymbol{\theta}}^* \in \mathbb{R}^n, b^* \in \mathbb{R} : \ \langle \hat{\boldsymbol{\theta}}^*, \hat{\mathbf{x}}_i \rangle + b^* \begin{cases} < 0, & \text{if } y_i = -1, \\ > 0, & \text{if } y_i = +1, \end{cases} \qquad \forall i = 1, \ldots, m.$$

Equivalently, we can combine the two cases of y_i into one condition

$$y_i(\langle \hat{\boldsymbol{\theta}}^*, \hat{\mathbf{x}}_i \rangle + b^*) > 0 \qquad \forall i = 1, \ldots, m. \tag{4.3}$$

Recall that $\{\hat{\mathbf{x}} : \langle \hat{\boldsymbol{\theta}}, \hat{\mathbf{x}} \rangle + b = 0\}$ is a hyperplane in $\mathbb{R}^n$ with a normal vector $\hat{\boldsymbol{\theta}} \neq 0$, and passing through the point $-b\frac{\hat{\boldsymbol{\theta}}}{\|\hat{\boldsymbol{\theta}}\|_2^2}$. An example of a separable dataset and one suitable hyperplane is shown below.



Note that there may exist (infinitely) many parameters $\hat{\boldsymbol{\theta}}^*$ and b^* satisfying the halfspaces condition (4.3). In the figure above, for example, we can move the dashed line up and down, or tilt it without violating (4.3). Moreover, we have no idea if another (test) point $\hat{\mathbf{x}}_5$ will end up correctly above or below the separating hyperplane. Therefore, for a well-generalisable halfspaces classifier, we need to impose extra *penalties* on unwanted behaviour of $\hat{\boldsymbol{\theta}}$ and b.

For the moment, we can just recall that our goal is to have $y_i\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle > 0$ for all i (here we use the homogeneous expansion (4.1)), so we can minimise the empirical risk constructed as a sum of magnitudes of $y_i\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle$ whenever this value is negative. However, the norm of the parameter vector does not matter: if $\langle\boldsymbol{\theta}, \mathbf{x}\rangle > 0$, the same holds also for $\langle(c\boldsymbol{\theta}), \mathbf{x}\rangle > 0$ with any $c > 0$. Thus, we can formulate the training of the halfspaces classifier as the following constrained minimisation problem:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^{n+1}} L_{\mathbf{D}}(\boldsymbol{\theta}) \quad \text{such that} \quad \|\boldsymbol{\theta}\|_2 = 1, \quad L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{-\min(y_i\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle,\ 0)}_{\ell(h_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i)}. \tag{4.4}$$

One solution to (4.4) can be obtained by minimizing instead the norm of the parameter under the constraint of *significantly* correct classification:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^{n+1}} \|\boldsymbol{\theta}\|_2^2 \quad \text{such that} \quad y_i\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle \geq 1 \quad \forall i. \tag{4.5}$$

This problem is more computationally convenient since it is of the *quadratic* type, for which many efficient algorithms have been developed. One implementation of the Empirical Risk Minimisation (ERM) (4.4) is the Perceptron algorithm of Rosenblatt. We will discuss it later.

In addition to being easier to compute, the solution to (4.5) has another useful property of separating the two classes of domain points with the *largest margin*. The margin of a hyperplane is defined as the minimal distance between a point in the training dataset and the hyperplane, $\min_{i=1,\dots,m} \min_{\mathbf{v}\in\mathbb{R}^{n+1}:\langle\boldsymbol{\theta},\mathbf{v}\rangle=0} \|\mathbf{x}_i - \mathbf{v}\|$.

The largest-margin halfspaces, being as far from all training points as possible, can be seen as the "safest" solution. This is simply because it can also correctly classify test points which are closer to the separating hyperplane than any training point. In the terminology of statistical learning, the largest-margin halfspaces are well generalisable. The halfspaces classifier built upon the solution of (4.5) is the simplest (*Hard-Margin*) version of the so-called Support Vector Machines. It can be generalised to an even more powerful machine learning tool that can ignore *slightly* incorrect predictions, and handle datasets that are not separable as per Definition 4.5. The name "Support Vector Machine" stems from the fact that the optimal parameter $\boldsymbol{\theta}^*$ is actually a linear combination of certain training domain points $\mathbf{x}_i \in \mathbf{X}_{train}$, called *support vectors*. These vectors are exactly at the distance $1/\|\boldsymbol{\theta}^*\|$ from the separating hyperplane, that is, they "support" the fence of the margin around the separating hyperplane.

End of lecture 8

4.1.5 Logistic regression and maximum likelihood estimators

Alternatively, we can transform the output of a prediction rule.

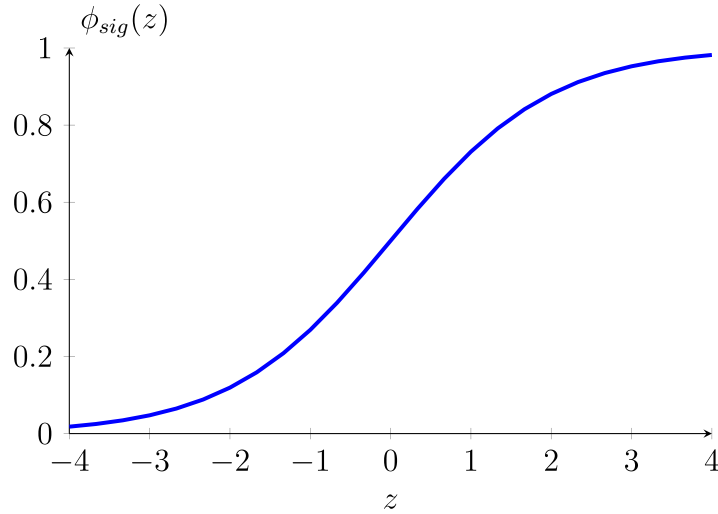
Definition 4.6. *The set*

$$\mathcal{H}_n^{log} = \{\phi_{sig}(\langle\boldsymbol{\theta}, \mathbf{x}\rangle) : \boldsymbol{\theta} \in \mathbb{R}^n\}$$

is called the class of ***logistic regression****, where the* ***sigmoid function*** $\phi_{sig} : \mathbb{R} \to [0, 1]$ *reads*

$$\phi_{sig}(z) = \frac{1}{1 + \exp(-z)}.$$

The name "sigmoid" means "S-shaped", referring to the plot of this function, shown below.



We can interpret $h_{\boldsymbol{\theta}}(\mathbf{x}) = \phi_{sig}(\langle \boldsymbol{\theta}, \mathbf{x} \rangle)$ as the *probability* that the output of the linear prediction $\langle \boldsymbol{\theta}, \mathbf{x} \rangle$ is positive, that is, that the label of $\mathbf{x}$ is $+1$. Recall that the prediction of the halfspaces classifier is essentially whether $\langle \boldsymbol{\theta}, \mathbf{x} \rangle$ is positive or not. Therefore, we can recover the halfspaces prediction by checking whether the logistic prediction is above $\phi_{sig}(0) = 1/2$ or not.

However, the logistic prediction rule offers a more **uncertainty quantification**: if $\langle \boldsymbol{\theta}, \mathbf{x} \rangle$ is close to 0, the logistic rule indicates that it is not sure about the halfspaces label $\text{sign}(\langle \boldsymbol{\theta}, \mathbf{x} \rangle)$ by returning a probability that is about 50%. In contrast, when $\langle \boldsymbol{\theta}, \mathbf{x} \rangle$ is significantly far from 0 (on either side), the probability is closer to 1 or 0, implying a more confident classification.

Now, again, we need to specify the loss function. Here, we can use the fact that $\phi_{sig}(\langle \boldsymbol{\theta}, \mathbf{x} \rangle)$ is a probability, and find the optimal $\boldsymbol{\theta}$ as a so-called *Maximum Likelihood Estimator* (MLE).

Definition 4.7. *Assume a dataset $\mathbf{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ contains independent realisations of a random variable $(X, Y) \sim \mathbb{P}_{\boldsymbol{\theta}}$, whose distribution is tuneable through the parameter $\boldsymbol{\theta} \in \mathbb{R}^n$. The* ***likelihood of D*** *is a probability of observing the labels $\mathbf{y} = \{y_1, \ldots, y_m\}$ given domain points $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$,*

$$p(\mathbf{y}|\mathbf{X}; \boldsymbol{\theta}) = \prod_{i=1}^{m} \mathbb{P}_{\boldsymbol{\theta}}(Y = y_i | X = \mathbf{x}_i).$$

Definition 4.8. *The* ***Maximum Likelihood Estimator (MLE)*** *is the parameter that maximizes the likelihood,*

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta} \in \mathbb{R}^n} p(\mathbf{y}|\mathbf{X}; \boldsymbol{\theta}).$$

We can define the logistic regression likelihood of one point as

$$\mathbb{P}_{\boldsymbol{\theta}}(Y = +1 | X = \mathbf{x}) = \phi_{sig}(\langle \boldsymbol{\theta}, \mathbf{x} \rangle) = \frac{1}{1 + \exp(-\langle \boldsymbol{\theta}, \mathbf{x} \rangle)} = \frac{1}{1 + \exp(-y\langle \boldsymbol{\theta}, \mathbf{x} \rangle)}.$$

The probability of observing $y = -1$ is the complement of $\phi_{sig}(\langle \boldsymbol{\theta}, \mathbf{x} \rangle)$ to 1,

$$\mathbb{P}_{\boldsymbol{\theta}}(Y = -1 | X = \mathbf{x}) = 1 - \phi_{sig}(\langle \boldsymbol{\theta}, \mathbf{x} \rangle) = \frac{\exp(-\langle \boldsymbol{\theta}, \mathbf{x} \rangle)}{1 + \exp(-\langle \boldsymbol{\theta}, \mathbf{x} \rangle)} = \frac{1}{\exp(\langle \boldsymbol{\theta}, \mathbf{x} \rangle) + 1} = \frac{1}{1 + \exp(-y\langle \boldsymbol{\theta}, \mathbf{x} \rangle)}.$$

We see that for both classes the probability of observing y can be written using the same formula

$$\mathbb{P}_{\boldsymbol{\theta}}(Y = y|X = \mathbf{x}) = \frac{1}{1 + \exp(-y\langle\boldsymbol{\theta}, \mathbf{x}\rangle)}.$$

Now for the entire dataset, following Definition 4.7,

$$p(\mathbf{y}|\mathbf{X}; \boldsymbol{\theta}) = \mathbb{P}_{\boldsymbol{\theta}}(Y = y_1|X = \mathbf{x}_1) \cdots \mathbb{P}_{\boldsymbol{\theta}}(Y = y_m|X = \mathbf{x}_m) = \prod_{i=1}^{m} \frac{1}{1 + \exp(-y_i\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle)}.$$

We can search for the maximum likelihood estimator directly as

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}\in\mathbb{R}^n} p(\mathbf{y}|\mathbf{X}; \boldsymbol{\theta}).$$

However, this might be difficult on a practical computer since this product of many numbers, each of which is below 1, may become smaller than the smallest number that can be stored in computer memory (the so-called underflow problem). Moreover, the joint probability lacks the ERM form that is beneficial for some optimisation algorithms. To circumvent this issue, we can notice that $\log(\cdot)$ is a monotone function, and

$$f(\boldsymbol{\theta}^*) = \max_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \quad \Longleftrightarrow \quad -\log(f(\boldsymbol{\theta}^*)) = \min_{\boldsymbol{\theta}} [-\log(f(\boldsymbol{\theta}))].$$

So instead of maximising the likelihood $p(\mathbf{y}|\mathbf{X}; \boldsymbol{\theta})$, we can minimise the **negative log-likelihood**

$$L_{\mathbf{D}}(\boldsymbol{\theta}) = \sum_{i=1}^{m} [-\log(p(y_i|\mathbf{x}_i; \boldsymbol{\theta}))] = \sum_{i=1}^{m} \log(1 + \exp(-y_i\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle)) = \frac{1}{m}\sum_{i=1}^{m} \ell(h_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i),$$

which has the usual form of an empirical risk with the pointwise loss

$$\ell(h_{\boldsymbol{\theta}}(\mathbf{x}), y) = m\log(1 + \exp(-y\langle\boldsymbol{\theta}, \mathbf{x}\rangle)).$$

The log-function appearing in the loss gives the name of "logistic regression" to this kind of prediction rule. Its advantage compared to the halfspaces loss (4.4) for example (in addition to uncertainty quantification) is that the logistic loss function is *convex* and *smooth*. It can thus be minimised using standard methods, some of which we consider next.

Note that we diverted from optimising $\boldsymbol{\theta}$ in a prediction rule $h_{\boldsymbol{\theta}}(\mathbf{x})$ to optimising $\boldsymbol{\theta}$ in the data-generating probability function. However, an optimal $h(\mathbf{x})$ can be recovered simply as another maximum likelihood estimator, but over the **label** y instead of the parameter $\boldsymbol{\theta}$.

Definition 4.9. *Given the likelihood function $p(y|\mathbf{x}; \boldsymbol{\theta})$, the* ***Bayes-optimal*** *prediction rule*

$$h_{Bayes}(\mathbf{x}) = \arg\max_{y\in\mathcal{Y}} p(y|\mathbf{x}; \boldsymbol{\theta}). \tag{4.6}$$

The name Bayes stems from the Bayes theorem, that we can see $p(y|\mathbf{x}; \boldsymbol{\theta}) = \mathbb{P}_{\boldsymbol{\theta}}(Y = y|X = \mathbf{x})$ as the *posterior* probability function, and $\mathbb{P}_{\boldsymbol{\theta}}(X = \mathbf{x}|Y = y)$ as a *likelihood* function of the domain point $\mathbf{x}$ to take a particular value given the label y instead. The Bayes theorem states

$$\mathbb{P}_{\boldsymbol{\theta}}(Y = y|X = \mathbf{x}) = \frac{\mathbb{P}_{\boldsymbol{\theta}}(X = \mathbf{x}|Y = y)\mathbb{P}_{\boldsymbol{\theta}}(Y = y)}{\mathbb{P}_{\boldsymbol{\theta}}(X = \mathbf{x})}, \tag{4.7}$$

where $\mathbb{P}_{\boldsymbol{\theta}}(Y = y)$ is the *prior* probability of observing the label y in absence of domain data.

4.1.6 Naive Bayes

Another frequently used Maximum Likelihood Estimator is the Naive Bayes classifier. The Naive Bayes method aims to reduce the number of trainable parameters in $\boldsymbol{\theta}$ (thus simplifying learning) by assuming extra properties of the data-generating distribution.

Example 4.10. *Assume that each component of* $\mathbf{x} = (x_1, \ldots, x_n)$ *and* y *can take only two values,* $x_j \in \{0, 1\}$, $j = 1, \ldots, n$, *and* $y \in \mathcal{Y} = \{-1, 1\}$. *Let*

$$k(\mathbf{x}, y) = 2^n x_1 + \cdots + 2x_n + \frac{y+1}{2} \in \{0, \ldots, 2^{n+1} - 1\}$$

index all possible values of $\mathbf{x}$ *and* y *similarly to* (2.1). *We can let* $\boldsymbol{\theta} = (\theta_0, \ldots, \theta_{2^{n+1}-1})$ *store the values of* $p(y|\mathbf{x}; \boldsymbol{\theta})$ *for all* $\mathbf{x}$ *and* y,

$$p(y|\mathbf{x}; \boldsymbol{\theta}) = \theta_{k(\mathbf{x},y)}.$$

Given data $\mathbf{x}_1, \ldots, \mathbf{x}_m$ *and* $y_1, \ldots, y_m$, *the maximum likelihood estimator can be learned as usual,*

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in [0,1]^{2^{n+1}}}{\arg\max} \prod_{i=1}^{m} p(y_i|\mathbf{x}_i; \boldsymbol{\theta}) = \underset{\boldsymbol{\theta} \in [0,1]^{2^{n+1}}}{\arg\max} \prod_{i=1}^{m} \theta_{k(\mathbf{x}_i, y_i)}.$$

This gives us the entire information about the data-generating distribution (the "Holy Grail" of statistical learning), but the number of parameters grows exponentially in n, which will quickly exceed the computing capacity and also overfit.

In the Naive Bayes approach we make the (rather naive) assumption that given the label y, the components of the domain point, $\mathbf{x} = (x_1, \ldots, x_n)$, are independent of each other.

Assumption 4.11 (Naive Bayes)**.**

$$\mathbb{P}_{\boldsymbol{\theta}}(X = \mathbf{x}|Y = y) = p(x_1|y; \boldsymbol{\theta}) \cdots p(x_n|y; \boldsymbol{\theta}).$$

Since $p(x_1|y; \boldsymbol{\theta}), \ldots, p(x_n|y; \boldsymbol{\theta})$ and the prior $\mathbb{P}_{\boldsymbol{\theta}}(Y = y)$ are separate functions, we can parametrise each of them by its own parameter vector, $p(x_1|y; \boldsymbol{\theta}_1), \ldots, p(x_n|y; \boldsymbol{\theta}_n), \mathbb{P}_{\boldsymbol{\theta}_{n+1}}(Y = y)$. The entire parameter vector is then a concatenation $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n, \boldsymbol{\theta}_{n+1})$.

Theorem 4.12. *Under the Naive Bayes assumption,*

$$h_{Bayes}(\mathbf{x}) = \underset{y \in \mathcal{Y}}{\arg\max} \left[p(x_1|y; \boldsymbol{\theta}_1) \cdots p(x_n|y; \boldsymbol{\theta}_n) \mathbb{P}_{\boldsymbol{\theta}_{n+1}}(Y = y) \right]. \tag{4.8}$$

Proof.

$$\begin{aligned}
h_{Bayes}(\mathbf{x}) &= \underset{y \in \mathcal{Y}}{\arg\max}\, \mathbb{P}_{\boldsymbol{\theta}}(Y = y|X = \mathbf{x}) \\
&= \underset{y \in \mathcal{Y}}{\arg\max} \frac{\mathbb{P}_{\boldsymbol{\theta}}(X = \mathbf{x}|Y = y)\mathbb{P}_{\boldsymbol{\theta}}(Y = y)}{\mathbb{P}_{\boldsymbol{\theta}}(X = \mathbf{x})} \\
&= \underset{y \in \mathcal{Y}}{\arg\max} \left[\mathbb{P}_{\boldsymbol{\theta}}(X = \mathbf{x}|Y = y)\mathbb{P}_{\boldsymbol{\theta}}(Y = y)\right] \\
&= \underset{y \in \mathcal{Y}}{\arg\max} \left[p(x_1|y; \boldsymbol{\theta}_1) \cdots p(x_n|y; \boldsymbol{\theta}_n) \mathbb{P}_{\boldsymbol{\theta}_{n+1}}(Y = y) \right].
\end{aligned}$$

□

Similarly, the maximum likelihood estimator can be learned as

$$(\boldsymbol{\theta}_1^*, \ldots, \boldsymbol{\theta}_{n+1}^*) = \arg\max_{\boldsymbol{\theta}_1,\ldots,\boldsymbol{\theta}_{n+1}} \prod_{i=1}^{m} \left[p(x_{i,1}|y_i;\boldsymbol{\theta}_1)\cdots p(x_{i,n}|y_i;\boldsymbol{\theta}_n)\mathbb{P}_{\boldsymbol{\theta}_{n+1}}(Y=y_i)\right].$$

In the setup of Example 4.10, note that $\mathbb{P}_{\boldsymbol{\theta}_{n+1}}(Y=y)$ depends on only two values of y, and can be stored in $\boldsymbol{\theta}_{n+1} \in \mathbb{R}^2$, whereas $p(x_k|y;\boldsymbol{\theta}_k)$ depends on only four values of the arguments, and can be stored in $\boldsymbol{\theta}_k \in \mathbb{R}^4$. In total, we need to learn only $2+4n$ parameters instead of 2^{n+1}. A massive reduction in memory consumption and computing time!

4.1.7 Multiclass classification

Suppose we want to predict a label that can take more than two values, $y \in \mathcal{Y} = \{1,\ldots,K\}$. One option is called One-versus-All: we stick as close as possible to the previous developments (e.g. halfspaces), and solve K binary classification problems instead, by separating $y=k$ (which we turn into $y=1$) and $y \neq k$ (which we label as -1), for each $k=1,\ldots,K$. However, maximum likelihood estimators in general (and the Naive Bayes in particular) are especially convenient for solving the multiclass problem directly. Both the optimal Bayes (4.6) and the Naive Bayes (4.8) prediction rules can select the $y \in \mathcal{Y}$ of the maximum likelihood for a fixed $\boldsymbol{\theta}$, for any integer number of classes K. Vice versa, the training is performed as the maximum likelihood estimator $\boldsymbol{\theta}^*$, where the likelihood is evaluated on the fixed training dataset $\mathbf{D}$.

End of lecture 9

4.2 Optimization algorithms

Recall that the goal of supervised learning is often to minimise a loss function, $L_{\mathbf{D}}(h_{\boldsymbol{\theta}})$. Writing up all minimizers $\boldsymbol{\theta}^*$ analytically is impossible for practical problems. Therefore, we need to consider numerical optimization algorithms, which can approximate $\boldsymbol{\theta}^*$ with a controllable accuracy (and ideally with a low amount of computing time).

Let us denote a function subject to minimisation $L:\mathbb{R}^n \to \mathbb{R}$ (i.e. a real valued function of several variables) and we have to seek a $\boldsymbol{\theta}^*$ such that

$$L(\boldsymbol{\theta}^*) = \min_{\boldsymbol{\theta}\in\mathbb{R}^n} L(\boldsymbol{\theta}) . \tag{4.9}$$

The minimisation is over all $\boldsymbol{\theta}$ in $\mathbb{R}^n$ without any constraints. Ideally, we aim to find a *global minimiser.*

Definition 4.13. *The* ***global minimiser*** *of $L:\mathbb{R}^n \to \mathbb{R}$ is a point $\boldsymbol{\theta}^* \in \mathbb{R}^n$ such that*

$$L(\boldsymbol{\theta}^*) \le L(\boldsymbol{\theta}) \qquad \forall \boldsymbol{\theta} \in \mathbb{R}^n. \tag{4.10}$$

However, unless the function $L(\boldsymbol{\theta})$ is globally convex, it may contain multiple local minima, and ensuring that any particular local minimiser $\boldsymbol{\theta}^*$ is also a global minimiser is a much harder problem, and no good generally available algorithms exist to do it.

Definition 4.14. *A point $\boldsymbol{\theta}^* \in \mathbb{R}^n$ is a* ***local minimiser*** *of L in $\mathbb{R}^n$ if there exists $r>0$ such that*

$$L(\boldsymbol{\theta}^*) \le L(\boldsymbol{\theta}) \qquad \forall \boldsymbol{\theta} \in \mathbb{R}^n \quad \text{such that} \quad \|\boldsymbol{\theta}-\boldsymbol{\theta}^*\|_2 \le r. \tag{4.11}$$

Most methods only find local minima. Moreover, the definition above is easy to check analytically, but unclear how to compute numerically. Practical algorithms seek *candidate* local minimisers by satisfying the first-order *necessary optimality condition*: if $\boldsymbol{\theta}^*$ is a local minimizer of a differentiable at $\boldsymbol{\theta}^*$ function L, then

$$\nabla L(\boldsymbol{\theta}^*) = \begin{bmatrix} \partial L(\boldsymbol{\theta}^*)/\partial\theta_1 \\ \partial L(\boldsymbol{\theta}^*)/\partial\theta_2 \\ \vdots \\ \partial L(\boldsymbol{\theta}^*)/\partial\theta_n \end{bmatrix} = 0, \tag{4.12}$$

and if the function is twice differentiable at $\boldsymbol{\theta}^*$, also the second-order condition $\nabla^2 L(\boldsymbol{\theta}^*) \geq 0$. Many good algorithms exist to do this.

4.2.1 First-order methods: gradient descent (GD)

The simplest method for finding a local minimum of a differentiable function is that of **gradient decent (GD)**. This method starts from the realisation that, at any starting point $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, L decreases most rapidly in the direction of $-\nabla L(\boldsymbol{\theta}_0)$. To see why this is the case, let us look for a unit direction $\hat{\mathbf{v}}$ (that is, $\|\hat{\mathbf{v}}\|_2 = 1$) such that

$$\frac{d}{dt}\left\{L(\boldsymbol{\theta}_0 + t\hat{\mathbf{v}})\right\}|_{t=0} \quad \text{is minimised} .$$

Denoting by $\langle\cdot,\cdot\rangle$ the inner product, this implies (via the chain rule) that

$$\langle(\nabla L)(\boldsymbol{\theta}_0 + t\hat{\mathbf{v}}), \hat{\mathbf{v}}\rangle|_{t=0} \quad \text{is minimised} ,$$

which in turn implies that $\langle(\nabla L)(\boldsymbol{\theta}_0), \hat{\mathbf{v}}\rangle$ should be as negative as possible. Since, by the Cauchy-Schwarz inequality,

$$\langle\nabla L(\boldsymbol{\theta}_0), \hat{\mathbf{v}}\rangle \;\leq\; \|\nabla L(\boldsymbol{\theta}_0)\|_2\,\|\hat{\mathbf{v}}\|_2 \;=\; \|\nabla L(\boldsymbol{\theta}_0)\|_2 \;, \tag{4.13}$$

the direction which gives the steepest descent is

$$\hat{\mathbf{v}} = -\frac{(\nabla L)(\boldsymbol{\theta}_0)}{\|(\nabla L)(\boldsymbol{\theta}_0)\|_2} \;,$$

and the quantity on the left hand side of (4.13) is $-\|\nabla L(\boldsymbol{\theta}_0)\|_2$.

In the method of gradient descent, a series of steps is chosen, with each step taken in the direction $-\nabla L$. The iteration proceeds as shown in Algorithm 6.

Algorithm 6 Gradient Descent (GD)

1: Start with a point $\boldsymbol{\theta}_0 \in \mathbb{R}^n$.
2: **for** $k = 0, 1, \ldots,$ until $L(\boldsymbol{\theta}_k)$ cannot be reduced further **do**
3: Choose a *learning rate* (length of the current step) $t_k > 0$.
4: Set $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_k \nabla L(\boldsymbol{\theta}_k)$.
5: **end for**
6: **return** $\boldsymbol{\theta}_k \approx \boldsymbol{\theta}^*$.

There are different ways to select the learning rate t_k. The simplest one is just to fix it once and for all iterations. However, this fixed step may be too large in the vicinity of the exact minimizer, and the method may jump around the exact solution, but never actually converge to it. If the learning rate is too large, the iterations may simply escape a sufficient vicinity of the local minimum and diverge.

One way to circumvent this problem is the **line search**. Note that t_k is just a *single* variable, and minimising some function over it is a relatively easy thing to do. Specifically, we consider all possible vectors of the form $\boldsymbol{\eta}_t = \boldsymbol{\theta}_k - t\nabla L(\boldsymbol{\theta}_k)$, and find the value of $t > 0$ which minimises the same loss function $L(\boldsymbol{\eta}_t)$. In other words, we search for the minimizer of $L(\boldsymbol{\theta})$ along the *line* passing through $\boldsymbol{\theta}_k$ in the direction of $-\nabla L(\boldsymbol{\theta}_k)$, hence the name of this variant of the method. There are many methods to solve the one-dimensional minimisation problem over t including the method of bisection and the (faster) golden search method. In Python, the one-dimensional minimisation can be solved by the function `fminbound` in the `optimize` module of `scipy`.

Even with line search, the GD method may be too slow. Note that the line-searched t_k is the minimiser of $\tilde{L}(t) := L(\boldsymbol{\eta}_t)$, hence,

$$\frac{d\tilde{L}}{dt}(t_k) = \frac{d}{dt}\left\{L(\boldsymbol{\theta}_k - t\nabla L(\boldsymbol{\theta}_k))\right\}|_{t=t_k} = 0.$$

By the chain rule,

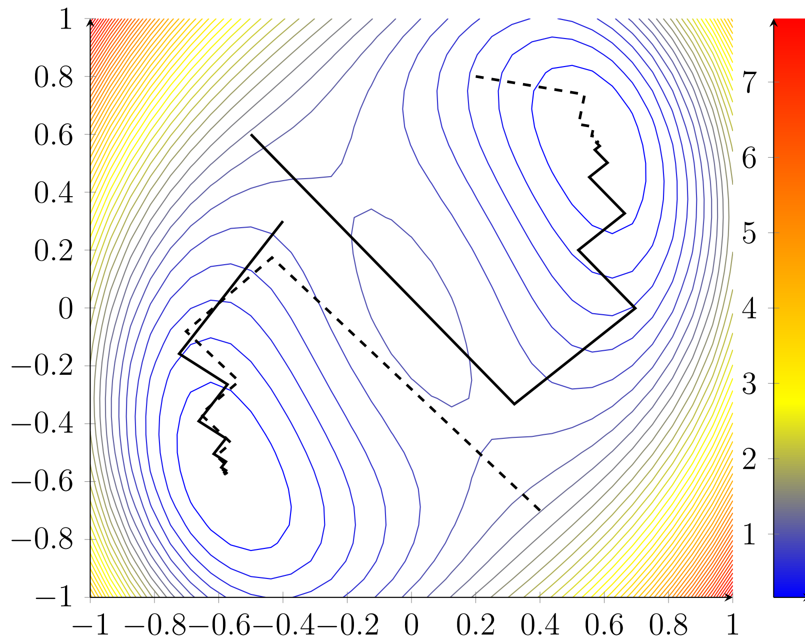
$$\langle -\nabla L(\boldsymbol{\theta}_k - t_k\nabla L(\boldsymbol{\theta}_k)), \nabla L(\boldsymbol{\theta}_k)\rangle = 0,$$

and so $\langle \nabla L(\boldsymbol{\theta}_{k+1}), \nabla L(\boldsymbol{\theta}_k)\rangle = 0$. That is, the consecutive search directions are always orthogonal to each other. If the contours of the loss function are long and thin this can lead to a very large number of iterations, making repeating zig-zag searches in very different directions.

We can see this in an application of GD to the problem of computing a minimum of the function

$$L(\theta_1, \theta_2) = (\theta_1 - \theta_2)^2 + (2\theta_1^2 + \theta_2^2 - 1)^2,$$

in which four different calculations from different starting points are depicted below. This function has two local minima, and the GD method can converge to either of the two depending on the starting point.



4.2.2 Convergence of gradient descent

To guarantee convergence of GD, we must assume certain properties of the loss function $L(\boldsymbol{\theta})$. Firstly, it makes no sense to follow the gradient if it does not even exist. However, to obtain a rigorous upper bound on the *rate* of convergence we need more.

Definition 4.15. *A domain* $\Omega \subset \mathbb{R}^n$ *is called* ***convex*** *if*

$$\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta} \in \Omega \qquad \forall \boldsymbol{\theta}, \boldsymbol{\eta} \in \Omega, \quad \forall \alpha \in [0,1].$$

This definition fulfils trivially if the domain contains only one point, $\Omega = \{\boldsymbol{\theta}\}$. However, such domains are obviously not interesting for learning. To exclude such pathological cases, we introduce also the following definition.

Definition 4.16. *A domain* $\Omega \subset \mathbb{R}^n$ *is called* ***nontrivial*** *if its measure is positive,*

$$|\Omega| = \int_\Omega 1 \; d\boldsymbol{\theta} > 0.$$

Now we are ready to state a useful property for loss functions.

Definition 4.17. *A function* $L : \Omega \to \mathbb{R}$ *is called* β***-smooth*** *on a nontrivial convex domain* $\Omega \subset \mathbb{R}^n$ *for some* $\beta > 0$ *if it is continuously differentiable, and its gradient has Lipschitz constant* β*,*

$$\|\nabla L(\boldsymbol{\eta}) - \nabla L(\boldsymbol{\theta})\|_2 \le \beta \|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2 \qquad \forall \boldsymbol{\eta}, \boldsymbol{\theta} \in \Omega.$$

In many cases we will take $\Omega = \mathbb{R}^n$, i.e. consider the function on the whole space. However, this doesn't have to be the case and a function can be β-smooth on a finite domain, but not β-smooth on the whole space $\mathbb{R}^n$.

End of lecture 10

The β-smoothness property implies the following inequality which is central to the analysis of GD.

Lemma 4.18. *If a function* $L : \Omega \to \mathbb{R}$ *is* β*-smooth on a nontrivial convex domain* $\Omega \subset \mathbb{R}^n$*, then*

$$L(\boldsymbol{\eta}) \le L(\boldsymbol{\theta}) + \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle + \frac{\beta}{2}\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2 \qquad \forall \boldsymbol{\eta}, \boldsymbol{\theta} \in \Omega. \tag{4.14}$$

Proof. Using the fundamental theorem of calculus and the chain rule, we can write

$$L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}) = \int_0^1 \frac{d}{dt} L(\boldsymbol{\theta} + t(\boldsymbol{\eta} - \boldsymbol{\theta}))dt = \int_0^1 \langle \nabla L(\boldsymbol{\theta} + t(\boldsymbol{\eta} - \boldsymbol{\theta})), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle dt.$$

Note that since Ω is convex, $\boldsymbol{\theta} + t(\boldsymbol{\eta} - \boldsymbol{\theta}) \in \Omega$. Adding and subtracting $\langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle$, and using the modulus inequality, we get

$$\begin{aligned} L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}) &= \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle + \int_0^1 \left(\langle \nabla L(\boldsymbol{\theta} + t(\boldsymbol{\eta} - \boldsymbol{\theta})), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle - \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle \right) dt \\ &\le \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle + \int_0^1 \left| \langle \nabla L(\boldsymbol{\theta} + t(\boldsymbol{\eta} - \boldsymbol{\theta})) - \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle \right| dt. \end{aligned}$$

Using the β-smoothness assumption, and the Cauchy-Schwarz inequality under the integral, we get

$$\begin{aligned} L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}) &\le \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle + \int_0^1 \beta \|t(\boldsymbol{\eta} - \boldsymbol{\theta})\|_2 \|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2 dt \\ &= \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle + \frac{\beta}{2}\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2. \end{aligned}$$

Now simply adding $L(\boldsymbol{\theta})$ to both sides concludes the proof. □

This Lemma already allows us to analyse the decrease of the loss function after one GD iteration. Namely, we can prove the following.

Lemma 4.19. *Let L be β-smooth on $\mathbb{R}^n$, $\boldsymbol{\theta}_k$ is the k-th iterate of GD, and $t_k > 0$ is the learning rate at the k-th iteration. Then*

$$L(\boldsymbol{\theta}_{k+1}) \leq L(\boldsymbol{\theta}_k) - t_k \left(1 - t_k \frac{\beta}{2}\right) \|\nabla L(\boldsymbol{\theta}_k)\|_2^2. \tag{4.15}$$

Proof. See Problem Sheet 7, Task (a). □

Furthermore, Task (b) of Problem Sheet 7 is concerned with finding the **optimal constant learning rate** $t_k = \hat{t} = 1/\beta$, such that the magnitude of the loss decrement $\hat{t}(1 - \hat{t}\beta/2)\|\nabla L(\boldsymbol{\theta}_k)\|_2^2$ is maximized.

Now we are ready to prove the first convergence result of GD.

Theorem 4.20. *Let L be β-smooth on $\mathbb{R}^n$ and bounded from below. Let $t_k = \hat{t} = 1/\beta$ for all $k \in \mathbb{N}_0 := \{0, 1, 2, \ldots\}$. Then for every $k \in \mathbb{N}_0$,*

$$\min_{i \leq k} \|\nabla L(\boldsymbol{\theta}_i)\|_2 \leq \left(\frac{2\beta}{k+1}(L(\boldsymbol{\theta}_0) - L(\boldsymbol{\theta}_{k+1}))\right)^{1/2} = \mathcal{O}(k^{-1/2}). \tag{4.16}$$

Proof. Using (4.15) at the ith iteration with $t_i = 1/\beta$ and the last term cast into the left hand side, we get

$$\sum_{i=0}^{k} \frac{1}{2\beta}\|\nabla L(\boldsymbol{\theta}_i)\|_2^2 \leq \sum_{i=0}^{k}(L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1})) = L(\boldsymbol{\theta}_0) - L(\boldsymbol{\theta}_{k+1}).$$

Therefore

$$\min_{i \leq k} \|\nabla L(\boldsymbol{\theta}_i)\|_2^2 \leq \frac{1}{k+1}\sum_{i=0}^{k} \|\nabla L(\boldsymbol{\theta}_i)\|_2^2 \leq \frac{2\beta}{k+1}(L(\boldsymbol{\theta}_0) - L(\boldsymbol{\theta}_{k+1})).$$

Moreover, since L is lower bounded, we have $\inf_{\boldsymbol{\theta} \in \mathbb{R}^n} L(\boldsymbol{\theta}) = L_{low} > -\infty$, and

$$\min_{i \leq k} \|\nabla L(\boldsymbol{\theta}_i)\|_2^2 \leq \frac{2\beta}{k+1}(L(\boldsymbol{\theta}_0) - L_{low}) \leq [2\beta(L(\boldsymbol{\theta}_0) - L_{low})]\,\frac{1}{k} = \mathcal{O}(k^{-1}).$$

Taking the square root of both sides of those inequalities gives the claim of the theorem. □

Recall that GD aims at finding a local minimiser $\boldsymbol{\theta}^*$ such that $\nabla L(\boldsymbol{\theta}^*) = 0$. This theorem gives an upper bound on the *residual* of a given approximate minimiser, $\|\nabla L(\boldsymbol{\theta}_i) - \nabla_L(\boldsymbol{\theta}^*)\|_2 = \|\nabla L(\boldsymbol{\theta}_i)\|_2$, which converges to zero as $k \to \infty$. However, in general we cannot claim that $\|\nabla L(\boldsymbol{\theta}_k)\|_2 \to 0$; this holds only for a *subsequence* of iterations, $\|\nabla L(\boldsymbol{\theta}_{i_k})\|_2 \to 0$, where i_k is the i realising the $\min_{i \leq k}$ in (4.16).

Remark 4.21. *Calculation of the β constant, and hence of the learning rate $t_k = 1/\beta$, can be difficult for a complicated function L. However, the convergence rate $\mathcal{O}(k^{-1/2})$ holds for any $t_k \in (0, \frac{2}{\beta})$, although with a different constant in the right hand side of (4.16) instead of 2β. In practice, a trial-and-error method can be used, when one tries smaller and smaller constant learning rates $t_k = t$ (e.g. $1/2, 1/4, 1/8$ and so on) until the GD method starts converging.*

Theorem 4.20 already gives us reassurance that the GD method can be used, but not a most fascinating one: the convergence rate of $k^{-1/2}$ is *very* slow. For example, to achieve the residual of 0.01 (1%), one needs about $1/0.01^2 = 10\,000$ iterations. Decreasing the residual by a factor of 10 (that is, obtaining one more significant decimal digit in the result) requires increasing the number of iterations by a factor of 100, which can quickly end up impractical. Fortunately, GD converges much faster if we assume more properties of the loss function.

Definition 4.22. *A continuous function $L : \Omega \to \mathbb{R}$ is called **convex** on a nontrivial convex domain $\Omega \subset \mathbb{R}^n$ if*

$$L(\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta}) \le \alpha L(\boldsymbol{\theta}) + (1-\alpha)L(\boldsymbol{\eta}) \qquad \forall \alpha \in [0,1], \quad \forall \boldsymbol{\theta}, \boldsymbol{\eta} \in \Omega. \tag{4.17}$$

Definition 4.23. *A continuous function $L : \Omega \to \mathbb{R}$ is called **λ-strongly convex** on a nontrivial convex domain $\Omega \subset \mathbb{R}^n$ if there exists $\lambda > 0$ such that the function*

$$R(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) - \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2$$

is convex on Ω.

Note that 0-strong convexity is just standard convexity.

End of lecture 11

To check the convexity of a function in practice, the following two results are convenient.

Lemma 4.24. *For a continuously differentiable function L, convexity is equivalent to*

$$\langle \nabla L(\boldsymbol{\eta}) - \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} \rangle \ge 0 \qquad \forall \boldsymbol{\theta}, \boldsymbol{\eta} \in \Omega. \tag{4.18}$$

Proof. First, note that $\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta} = \boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})$. Now if $L(\boldsymbol{\theta})$ is differentiable, we can write the directional derivative by definition as

$$\langle \nabla L(\boldsymbol{\theta}), (\boldsymbol{\eta} - \boldsymbol{\theta}) \rangle = \lim_{\alpha \to 1^-} \frac{L(\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})) - L(\boldsymbol{\theta})}{1-\alpha},$$

and plugging in (4.17), we get

$$\langle \nabla L(\boldsymbol{\theta}), (\boldsymbol{\eta} - \boldsymbol{\theta}) \rangle \le \lim_{\alpha \to 1^-} \frac{(\alpha - 1)L(\boldsymbol{\theta}) + (1-\alpha)L(\boldsymbol{\eta})}{1-\alpha} = -L(\boldsymbol{\theta}) + L(\boldsymbol{\eta}). \tag{4.19}$$

Similarly, $\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta} = \boldsymbol{\eta} - \alpha(\boldsymbol{\eta} - \boldsymbol{\theta})$, and hence

$$\langle \nabla L(\boldsymbol{\eta}), (\boldsymbol{\eta} - \boldsymbol{\theta}) \rangle = \lim_{\alpha \to 0^+} \frac{L(\boldsymbol{\eta}) - L(\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta})}{\alpha} \ge \lim_{\alpha \to 0^+} \frac{\alpha(L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}))}{\alpha} = L(\boldsymbol{\eta}) - L(\boldsymbol{\theta})$$

Subtracting (4.19), we obtain equation (4.18).

For the converse direction, using the first order Taylor expansion with the mean value form of the remainder, we can write

$$\begin{aligned} L(\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta}) &= L(\boldsymbol{\theta}) + \langle \nabla L(\mathbf{z}), \alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta} - \boldsymbol{\theta} \rangle \\ &= L(\boldsymbol{\theta}) + \langle \nabla L(\mathbf{z}), (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta}) \rangle, \end{aligned} \tag{4.20}$$

$$\begin{aligned} L(\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta}) &= L(\boldsymbol{\eta}) + \langle \nabla L(\mathbf{w}), \alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta} - \boldsymbol{\eta} \rangle \\ &= L(\boldsymbol{\eta}) + \langle \nabla L(\mathbf{w}), -\alpha(\boldsymbol{\eta} - \boldsymbol{\theta}) \rangle, \end{aligned} \tag{4.21}$$

where $\mathbf{z} = \boldsymbol{\theta} + t_*(1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})$, $\mathbf{w} = \boldsymbol{\eta} - s_*\alpha(\boldsymbol{\eta} - \boldsymbol{\theta})$ for some $t_*, s_* \in [0,1]$.

Adding together (4.20) multiplied by α and (4.21) multiplied by $(1-\alpha)$, we obtain

$$L(\alpha\boldsymbol{\theta} + (1-\alpha)\boldsymbol{\eta}) = \alpha L(\boldsymbol{\theta}) + (1-\alpha)L(\boldsymbol{\eta}) + \alpha(1-\alpha)\langle \nabla L(\mathbf{z}) - \nabla L(\mathbf{w}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle.$$

Note that $\mathbf{z} - \mathbf{w} = (\boldsymbol{\eta} - \boldsymbol{\theta})(t_*(1-\alpha) + s_*\alpha - 1)$, and $t_*(1-\alpha) + s_*\alpha \in [0,1]$ due to the convex combination of t_* and s_*. If $t_*(1-\alpha)+s_*\alpha-1 = 0$, this means $\mathbf{z} = \mathbf{w}$, hence $\nabla L(\mathbf{z}) - \nabla L(\mathbf{w}) = 0$, and the convexity inequality (4.17) holds as equality. If $t_*(1-\alpha) + s_*\alpha - 1 < 0$,

$$\langle \nabla L(\mathbf{z}) - \nabla L(\mathbf{w}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle = \langle \nabla L(\mathbf{z}) - \nabla L(\mathbf{w}), \mathbf{z} - \mathbf{w}\rangle \frac{1}{t_*(1-\alpha) + s_*\alpha - 1} \leq 0$$

using (4.18) for $\mathbf{z}$ and $\mathbf{w}$, and we obtain (4.17) again. □

Theorem 4.25. *If L is continuously differentiable, the λ-strong convexity is equivalent to*

$$\langle \nabla L(\boldsymbol{\eta}) - \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle \geq \lambda \|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2 \qquad \forall \boldsymbol{\theta}, \boldsymbol{\eta} \in \Omega. \tag{4.22}$$

Proof. Letting $R(\boldsymbol{\theta}) := L(\boldsymbol{\theta}) - \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2$, we have that

$$\begin{aligned} \langle \nabla L(\boldsymbol{\eta}) - \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta} - \geq \lambda \|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2 &= \langle \nabla L(\boldsymbol{\eta}) - \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle - \lambda\langle \boldsymbol{\eta} - \boldsymbol{\theta}, \boldsymbol{\eta} - \boldsymbol{\theta}\rangle \\ &= \langle \nabla R(\boldsymbol{\eta}) - \nabla R(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle. \end{aligned}$$

Therefore, the inequality (4.22) is equivalent to

$$\langle \nabla R(\boldsymbol{\eta}) - \nabla R(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle \geq 0. \tag{4.23}$$

as required. The result now follows from Lemma 4.24 and the definition of strong convexity (Definition 4.23). □

Another strong convexity property that will be useful is as follows.

Lemma 4.26. *If $L : \Omega \to \mathbb{R}$ is λ-strongly convex and differentiable, then*

$$L(\boldsymbol{\eta}) \geq L(\boldsymbol{\theta}) + \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle + \frac{\lambda}{2}\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2. \tag{4.24}$$

Proof. We start also with the directional derivative

$$\langle \nabla L(\boldsymbol{\theta}), (\boldsymbol{\eta} - \boldsymbol{\theta})\rangle = \lim_{\alpha \to 1^-} \frac{L(\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})) - L(\boldsymbol{\theta})}{1-\alpha},$$

but now express

$$L(\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})) = R(\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})) + \frac{\lambda}{2}\|\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})\|_2^2,$$

and plugging in (4.17) for $R(\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta}))$ gives

$$\begin{aligned} \langle \nabla L(\boldsymbol{\theta}), (\boldsymbol{\eta} - \boldsymbol{\theta})\rangle &\leq \lim_{\alpha \to 1^-} \frac{\alpha R(\boldsymbol{\theta}) + (1-\alpha)R(\boldsymbol{\eta}) + \frac{\lambda}{2}\|\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})\|_2^2 - L(\boldsymbol{\theta})}{1-\alpha} \\ &= L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}) - \frac{\lambda}{2} \lim_{\alpha \to 1^-} \frac{\alpha\|\boldsymbol{\theta}\|_2^2 + (1-\alpha)\|\boldsymbol{\eta}\|_2^2 - \|\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})\|_2^2}{1-\alpha}. \end{aligned}$$

Expanding

$$\|\boldsymbol{\theta} + (1-\alpha)(\boldsymbol{\eta} - \boldsymbol{\theta})\|_2^2 = \|\boldsymbol{\theta}\|_2^2 + 2(1-\alpha)\langle \boldsymbol{\theta}, \boldsymbol{\eta} - \boldsymbol{\theta}\rangle + (1-\alpha)^2\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2,$$

we get

$$\begin{aligned}\langle \nabla L(\boldsymbol{\theta}), (\boldsymbol{\eta}-\boldsymbol{\theta})\rangle &\leq L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}) - \frac{\lambda}{2}(\langle \boldsymbol{\eta},\boldsymbol{\eta}\rangle - \langle \boldsymbol{\theta},\boldsymbol{\theta}\rangle - 2\langle \boldsymbol{\theta},\boldsymbol{\eta}-\boldsymbol{\theta}\rangle)\\ &= L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}) - \frac{\lambda}{2}(\langle \boldsymbol{\eta},\boldsymbol{\eta}\rangle - 2\langle \boldsymbol{\theta},\boldsymbol{\eta}\rangle + \langle \boldsymbol{\theta},\boldsymbol{\theta}\rangle)\\ &= L(\boldsymbol{\eta}) - L(\boldsymbol{\theta}) - \frac{\lambda}{2}\|\boldsymbol{\eta}-\boldsymbol{\theta}\|_2^2 .\end{aligned}$$

Adding $L(\boldsymbol{\theta})$ and $\frac{\lambda}{2}\|\boldsymbol{\eta}-\boldsymbol{\theta}\|_2^2$ to both sides, we obtain (4.24). $\square$

Comparing (4.24) with (4.14), we note that for a function which is both β-smooth and λ-strongly convex it holds that $\lambda \leq \beta$.

For a λ-strongly convex loss function, the distance to its minimal value can be upper bounded by its gradient, which is known as the Polyak-Lojasiewicz (PL) inequality.

Lemma 4.27 (PL-inequality)**.** *Let L be λ-strongly convex on $\mathbb{R}^n$. Then*

$$L(\boldsymbol{\theta}) - \inf_{\boldsymbol{\eta}\in\mathbb{R}^n} L(\boldsymbol{\eta}) \leq \frac{1}{2\lambda}\|\nabla L(\boldsymbol{\theta})\|_2^2 \qquad \forall \boldsymbol{\theta}\in\mathbb{R}^n.$$

Proof. Firstly, we can bound $\inf_{\boldsymbol{\eta}\in\mathbb{R}^n} L(\boldsymbol{\eta})$ by taking the infimum of both sides of (4.24):

$$\inf_{\boldsymbol{\eta}\in\mathbb{R}^n} L(\boldsymbol{\eta}) \geq \inf_{\boldsymbol{\eta}\in\mathbb{R}^n}\left(L(\boldsymbol{\theta}) + \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta}-\boldsymbol{\theta}\rangle + \frac{\lambda}{2}\|\boldsymbol{\eta}-\boldsymbol{\theta}\|_2^2\right),$$

for any $\boldsymbol{\theta}\in\mathbb{R}^n$. To find the right hand side we can use the necessary optimality condition:

$$\nabla_{\boldsymbol{\eta}}\left(L(\boldsymbol{\theta}) + \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta}-\boldsymbol{\theta}\rangle + \frac{\lambda}{2}\|\boldsymbol{\eta}-\boldsymbol{\theta}\|_2^2\right)\Bigg|_{\boldsymbol{\eta}=\boldsymbol{\eta}^*} = \nabla L(\boldsymbol{\theta}) + \frac{2\lambda}{2}(\boldsymbol{\eta}^*-\boldsymbol{\theta}) = 0.$$

Solving this equation for $\boldsymbol{\eta}^*$, we get that the minimiser is $\boldsymbol{\eta}^* = \boldsymbol{\theta} - \frac{1}{\lambda}\nabla L(\boldsymbol{\theta})$, and the minimum of the function is

$$L(\boldsymbol{\theta}) + \langle \nabla L(\boldsymbol{\theta}), \boldsymbol{\eta}^*-\boldsymbol{\theta}\rangle + \frac{\lambda}{2}\|\boldsymbol{\eta}^*-\boldsymbol{\theta}\|_2^2 = L(\boldsymbol{\theta}) - \frac{1}{2\lambda}\|\nabla L(\boldsymbol{\theta})\|_2^2 \leq \inf_{\boldsymbol{\eta}\in\mathbb{R}^n} L(\boldsymbol{\eta}).$$

Rearranging the terms concludes the proof. $\square$

For loss functions that are both β-smooth and λ-strongly convex, we obtain an **exponential rate** of convergence – not only for the residual, but also for the iterates themselves.

Theorem 4.28. *Let $L : \mathbb{R}^n \to \mathbb{R}$ be bounded from below, λ-strongly convex and β-smooth on $\mathbb{R}^n$, and let $t_k = 1/\beta$ for all $k \in \mathbb{N}_0$. Then*

$$L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}^*) \leq (L(\boldsymbol{\theta}_0) - L(\boldsymbol{\theta}^*))\exp\left(-k\frac{\lambda}{\beta}\right),$$

and

$$\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 \leq \|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\|_2^2 \exp\left(-k\frac{\lambda}{\beta}\right),$$

where $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^n} L(\boldsymbol{\theta})$.

End of lecture 12

Proof. From Lemma 4.19, we obtain the loss decreasing property $L(\boldsymbol{\theta}_{k+1}) \leq L(\boldsymbol{\theta}_k) - \frac{1}{2\beta}\|\nabla L(\boldsymbol{\theta}_k)\|_2^2$, and the λ-strong convexity implies the PL-inequality with $\inf_{\boldsymbol{\eta}\in\mathbb{R}^n} L(\boldsymbol{\eta}) = L(\boldsymbol{\theta}^*)$. Combining the two, we get

$$\begin{aligned} L(\boldsymbol{\theta}_{k+1}) - L(\boldsymbol{\theta}^*) &\leq L(\boldsymbol{\theta}_k) - \frac{1}{2\beta}\|\nabla L(\boldsymbol{\theta}_k)\|_2^2 - L(\boldsymbol{\theta}^*) && \text{(subtract } L(\boldsymbol{\theta}^*) \text{ from both sides)} \\ &\leq L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}^*) - \frac{\lambda}{\beta}(L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}^*)) && \text{(replace } \|\nabla L(\boldsymbol{\theta}_k)\|_2^2 \text{ using PL)} \\ &= (L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}^*))\left(1 - \frac{\lambda}{\beta}\right). \end{aligned}$$

By induction, we get that $L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}^*) \leq (1 - \frac{\lambda}{\beta})^k (L(\boldsymbol{\theta}_0) - L(\boldsymbol{\theta}^*))$. Using that $\log(1+x) \leq x$ for all $x > -1$, we can bound $\log((1-\frac{\lambda}{\beta})^k) = k \cdot \log(1 - \frac{\lambda}{\beta}) \leq -k\frac{\lambda}{\beta}$, and thus

$$\left(1 - \frac{\lambda}{\beta}\right)^k \leq \exp\left(-k\frac{\lambda}{\beta}\right), \tag{4.25}$$

which gives the first claim of the theorem.

For the second claim we can first plug in $\boldsymbol{\theta}_{k+1}$ and expand the square norm,

$$\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2^2 = \left\|\boldsymbol{\theta}_k - \frac{1}{\beta}\nabla L(\boldsymbol{\theta}_k) - \boldsymbol{\theta}^*\right\|_2^2 = \|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 - \frac{2}{\beta}\langle \nabla L(\boldsymbol{\theta}_k), \boldsymbol{\theta}_k - \boldsymbol{\theta}^*\rangle + \frac{1}{\beta^2}\|\nabla L(\boldsymbol{\theta}_k)\|_2^2.$$

Replacing the middle term with the λ-strong convexity property (4.24), and the last term with the loss decreasing property gives

$$\begin{aligned} \|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2^2 &\leq \|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 + \frac{2}{\beta}\left(L(\boldsymbol{\theta}^*) - L(\boldsymbol{\theta}_k) - \frac{\lambda}{2}\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2\right) + \frac{2\beta}{\beta^2}(L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}_{k+1})) \\ &= \left(1 - \frac{\lambda}{\beta}\right)\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 + \frac{2}{\beta}(L(\boldsymbol{\theta}^*) - L(\boldsymbol{\theta}_{k+1})) \qquad (4.26) \\ &\leq \left(1 - \frac{\lambda}{\beta}\right)\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2. \end{aligned}$$

where for the last inequality we use that $\boldsymbol{\theta}^*$ is a global minimiser, so $L(\boldsymbol{\theta}^*) - L(\boldsymbol{\theta}_{k+1}) \leq 0$. Using induction and the bound (4.25) again, we complete the proof. □

Definition 4.29. *A method producing a sequence of iterates $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \ldots$ is called α-**linearly convergent** to $\boldsymbol{\theta}^*$ if there exists $\alpha \in (0,1)$ such that $\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2 \leq \alpha\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2$ for all k.*

Theorem 4.28 shows that if the loss function is β-smooth and λ-strongly convex, the GD method is α-linearly convergent with $\alpha = \sqrt{1 - \lambda/\beta}$. Each iteration of a α-linearly convergent method reveals $|\log_{10}\alpha|$ decimal digits of the result. The number of iterations is therefore proportional to the number of significant digits of the result. That's a massive improvement!

4.2.3 GD for empirical risk minimisation and linear regression (non-examinable)

Consider specifically the problem of empirical risk minimisation,

$$L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \ell(h_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i),$$

where $\mathbf{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ is a labelled training dataset, and $\ell : \mathbb{R}\times\mathbb{R} \to \mathbb{R}_+$ is a given pointwise loss function. To minimize the empirical risk with the GD method, it is enough to compute the gradient of the pointwise loss

$$\boxed{\tilde{\ell}_{\mathbf{x},y}(\boldsymbol{\theta}) := \ell(h_{\boldsymbol{\theta}}(\mathbf{x}), y),}$$

since, by linearity of the gradient,

$$\nabla L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}).$$

Let us look more closely at the linear regression problem. We assume that $h_{\boldsymbol{\theta}}(\mathbf{x}) = \langle\boldsymbol{\theta}, \mathbf{x}\rangle$ with $\boldsymbol{\theta}, \mathbf{x} \in \mathbb{R}^n$, and $\tilde{\ell}_{\mathbf{x},y}(\boldsymbol{\theta}) = (\langle\boldsymbol{\theta}, \mathbf{x}\rangle - y)^2$. It is easy enough to compute the gradient,

$$\nabla\tilde{\ell}_{\mathbf{x},y} = 2(\langle\boldsymbol{\theta}, \mathbf{x}\rangle - y)\mathbf{x},$$

and hence

$$\nabla L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m} \mathbf{x}_i \sum_{j=1}^{n} \theta_j x_{i,j} - \mathbf{x}_i y_i,$$

where $x_{i,j}$ is the jth element of $\mathbf{x}_i$. Collecting all those elements into a matrix

$$X = \begin{bmatrix}\mathbf{x}_1 & \cdots & \mathbf{x}_m\end{bmatrix} = \begin{bmatrix} x_{1,1} & \cdots & x_{m,1} \\ \vdots & & \vdots \\ x_{1,n} & \cdots & x_{m,n}\end{bmatrix} \in \mathbb{R}^{n\times m},$$

we can write the gradient of the loss function in a matrix-vector form,

$$\nabla L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{2}{m}(XX^\top\boldsymbol{\theta} - X\mathbf{y}) = A\boldsymbol{\theta} - \mathbf{b},$$

where $A = \frac{2}{m}XX^\top \in \mathbb{R}^{n\times n}$ is the so-called *Gram* matrix of X, and $\mathbf{b} = \frac{2}{m}X\mathbf{y} \in \mathbb{R}^n$.

In principle, we can try to resolve the necessary optimality condition $\nabla L_{\mathbf{D}}(\boldsymbol{\theta}) = 0$ directly by solving the system of linear equations $A\boldsymbol{\theta} = \mathbf{b}$. However, let us apply the GD method and analyse its convergence using Theorems 4.20 and 4.28. For this, we need to verify the definitions of β-smoothness and λ-strong convexity for $L_{\mathbf{D}}(\boldsymbol{\theta})$ of the linear regression.

Theorem 4.30. *The loss function $L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}(\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle - y_i)^2$ is β-smooth on $\mathbb{R}^n$ with $\beta = \frac{2}{m}\lambda_{\max}(XX^\top)$, where $\lambda_{\max}$ is the largest eigenvalue of a matrix.*

Proof. For any $\boldsymbol{\eta}, \boldsymbol{\theta} \in \mathbb{R}^n$, we can write

$$\begin{aligned}\|\nabla L_{\mathbf{D}}(\boldsymbol{\eta}) - \nabla L_{\mathbf{D}}(\boldsymbol{\theta})\|_2 &= \frac{2}{m}\|(XX^\top\boldsymbol{\eta} - X\mathbf{y}) - (XX^\top\boldsymbol{\theta} - X\mathbf{y})\|_2 \\ &= \frac{2}{m}\|(XX^\top)(\boldsymbol{\eta} - \boldsymbol{\theta})\|_2 \\ &\le \frac{2}{m}\|XX^\top\|_2\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2\,,\end{aligned}$$

where in the last inequality, we used the property of the matrix norm. Moreover, since $XX^\top$ is symmetric and positive semi-definite, its matrix 2-norm is equal to its largest eigenvalue (try to prove this using your knowledge from Numerical Analysis). Matching this to Definition 4.17, we obtain the claim of the lemma. □

Remark 4.31. *The definition of β-smoothness holds in fact with* any *β larger than $\frac{2}{m}\lambda_{\max}(XX^\top)$. However, taking β unnecessarily large will make GD unnecessarily slow, since this will increase the constant in front of $k^{-1/2}$ in Theorem 4.20, and the α-factor* $\exp(-\lambda/(2\beta))$ *in the α-linear convergence of Theorem 4.28. Therefore, we aim at the* minimal *β in the definition of β-smoothness. That being said, try to prove that it cannot be smaller than $\frac{2}{m}\lambda_{\max}(XX^\top)$.*

Theorem 4.32. *The loss function $L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}(\langle\boldsymbol{\theta}, \mathbf{x}_i\rangle - y_i)^2$ is λ-strongly convex on $\mathbb{R}^n$ with $\lambda = \frac{2}{m}\lambda_{\min}(XX^\top)$, where $\lambda_{\min}$ is the smallest eigenvalue of a matrix.*

Proof. We will check the condition (4.22). Plugging in $\nabla L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{2}{m}(XX^\top\boldsymbol{\theta} - X\mathbf{y})$, we get

$$\langle \nabla L_{\mathbf{D}}(\boldsymbol{\eta}) - \nabla L_{\mathbf{D}}(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle = \frac{2}{m}\langle XX^\top\boldsymbol{\eta} - XX^\top\boldsymbol{\theta}, \boldsymbol{\eta} - \boldsymbol{\theta}\rangle = \langle \boldsymbol{\eta} - \boldsymbol{\theta}, \frac{2}{m}XX^\top(\boldsymbol{\eta} - \boldsymbol{\theta})\rangle. \tag{4.27}$$

Using the variational characterisation of the minimal eigenvalue (Theorem 3.13) of the matrix $\frac{2}{m}XX^\top$, we get

$$\lambda_{\min}\left(\frac{2}{m}XX^\top\right) = \frac{2}{m}\lambda_{\min}\left(XX^\top\right) = \min_{\substack{\mathbf{x}\in\mathbb{R}^n \\ \|\mathbf{x}\|_2=1}} \left\langle \mathbf{x}, \frac{2}{m}XX^\top\mathbf{x}\right\rangle \le \left\langle \frac{\boldsymbol{\eta} - \boldsymbol{\theta}}{\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2}, \frac{2}{m}XX^\top\frac{\boldsymbol{\eta} - \boldsymbol{\theta}}{\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2}\right\rangle$$

since $\boldsymbol{\eta} - \boldsymbol{\theta}$ is arbitrary. Multiplying both sides by $\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2$ and plugging in (4.27), we get

$$\lambda_{\min}\left(\frac{2}{m}XX^\top\right)\|\boldsymbol{\eta} - \boldsymbol{\theta}\|_2^2 \le \langle \nabla L_{\mathbf{D}}(\boldsymbol{\eta}) - \nabla L_{\mathbf{D}}(\boldsymbol{\theta}), \boldsymbol{\eta} - \boldsymbol{\theta}\rangle,$$

as we need. □

Looking back at Theorem 4.28, we can rewrite the α-factor of the α-linear convergence as

$$\sqrt{1 - \frac{\lambda}{\beta}} = \sqrt{1 - \frac{\lambda_{\min}(XX^\top)}{\lambda_{\max}(XX^\top)}}.$$

Note that to guarantee the convergence of GD we need $\alpha < 1$, which requires $\lambda_{\max}(XX^\top) < \infty$ and $\lambda_{\min}(XX^\top) > 0$. While $\lambda_{\max}(XX^\top) < \infty$ for any finite X, $\lambda_{\min}(XX^\top) > 0$ if and only if $\operatorname{rank}(X) = n$. A necessary condition for this is $m \ge n$, so similar to the statistical learning theory, we need the data size at least as large as the size of the parameter $\boldsymbol{\theta}$.

However, even if $m \ge n$, some data points may be duplicate or very close to each other. In this case $\lambda_{\min}(XX^\top)$ is exactly or almost 0, and the α-linear convergence of GD falls apart. Nevertheless, the weaker convergence result established in Theorem 4.20 holds for any linear regression with finite data and labels, since $\beta = \frac{2}{m}\lambda_{\max}(XX^\top) < \infty$, and $L_{\mathbf{D}}(\boldsymbol{\theta})$ is bounded from below by 0.

This makes GD a reliable method for a huge range of optimisation problems, be it regression or classification, since GD also can be extended to functions that are only sub-differentiable, such as $\max(x, 0)$ or $\operatorname{sign}(x)$. From the bias-complexity tradeoff we know that a large training dataset, i.e. $m \gg 1$, is preferential for accurate learning. Now we see that this also yields a large (enough) λ parameter in the λ-strong convexity of the loss function, which gives GD a fast α-linear convergence.

However, there comes a computational bottleneck. If m is large, even if a single sample of $\nabla\tilde{\ell}_{\mathbf{x},y}(\boldsymbol{\theta})$ is fast to compute, the gradient of the total loss can become too slow, since the number of elementary computing operations is proportional to m. This problem is exacerbated by the fact that it is exactly the large dataset scenario when we want to increase the dimension of the parameter $\boldsymbol{\theta}$ too in order to train a richer prediction model. For example, the VGG16 neural network that was used to win the ImageNet competition in 2014 contains about 138 million parameters, and was trained on more than a million images, each image in turn containing thousands of pixels.

End of lecture 13

- Advantages of GD : Easy to use.
- Disadvantages : Needs to calculate the derivative ∇L explicitly. Also it can be very slow.

4.2.4 Stochastic gradient descent (SGD)

The **stochastic gradient descent (SGD)** method bypasses the slow computation of the exact gradient of the total loss function by allowing the descent direction to be a random vector, and requiring only that its **expected value** at each iteration is equal to the gradient direction. The pseudocode can be written as shown in Algorithm 7. The fundamental assumption is that $\mathbf{v}_k$ can be computed at a fraction of the cost required for the computation of the exact $\nabla L(\boldsymbol{\theta}_k)$.

Algorithm 7 Stochastic Gradient Descent (SGD)

1: Start with a point $\boldsymbol{\theta}_0 \in \mathbb{R}^n$.
2: **for** $k = 0, 1, \ldots,$ until $L(\boldsymbol{\theta}_k)$ cannot be reduced further **do**
3: Choose a random $\mathbf{v}_k$ such that $\mathbb{E}[\mathbf{v}_k] = \nabla L(\boldsymbol{\theta}_k)$.
4: Choose a *learning rate* (length of the current step) $t_k > 0$.
5: Set $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_k \mathbf{v}_k$.
6: **end for**
7: **return** $\boldsymbol{\theta}_k \approx \boldsymbol{\theta}^*$.

4.2.5 Convergence of SGD

To pitch the main idea without getting buried in too cumbersome calculus, let us focus solely on λ-strongly convex and β-smooth loss functions. The loss at the next SGD iteration can be written using the corollary of the β-smoothness (4.14) similarly to GD,

$$L(\boldsymbol{\theta}_{k+1}) \leq L(\boldsymbol{\theta}_k) - t_k \langle \nabla L(\boldsymbol{\theta}_k), \mathbf{v}_k \rangle + t_k^2 \frac{\beta}{2} \|\mathbf{v}_k\|_2^2.$$

Unfortunately, in contrast to GD, we cannot say anything about the sign of the middle term. Thus, $L(\boldsymbol{\theta}_{k+1})$ is *not* necessarily smaller than $L(\boldsymbol{\theta}_k)$. However, *in expectation* the loss is still decreased under certain assumptions. Firstly, taking the expectation (with respect to the distribution of $\mathbf{v}_k$) on both sides of the inequality above gives

$$\begin{aligned} \mathbb{E}_{\mathbf{v}_k}[L(\boldsymbol{\theta}_{k+1})] &\leq L(\boldsymbol{\theta}_k) - t_k \langle \nabla L(\boldsymbol{\theta}_k), \mathbb{E}_{\mathbf{v}_k}[\mathbf{v}_k] \rangle + t_k^2 \frac{\beta}{2} \mathbb{E}_{\mathbf{v}_k}[\|\mathbf{v}_k\|_2^2] \\ &= L(\boldsymbol{\theta}_k) - t_k \left(\|\nabla L(\boldsymbol{\theta}_k)\|_2^2 - t_k \frac{\beta}{2} \mathbb{E}_{\mathbf{v}_k}[\|\mathbf{v}_k\|_2^2] \right). \end{aligned}$$

Assuming a uniformly bounded *second moment* of the direction vector, $\mathbb{E}_{\mathbf{v}_k}[\|\mathbf{v}_k\|_2^2] \leq \gamma$, and that $\|\nabla L(\boldsymbol{\theta}_k)\|_2 \neq 0$ (unless $\boldsymbol{\theta}_k$ is the minimiser and we are done), we can always choose a positive learning rate $t_k < \frac{2\|\nabla L(\boldsymbol{\theta}_k)\|_2^2}{\beta\gamma}$ such that the expectation of the loss decreases. Since $\nabla L(\boldsymbol{\theta}_k)$ tends to 0 as we approach the minimiser, we can notice also that the learning rate t_k should vanish as $k \to \infty$ too. More formally, we can write the following.

Theorem 4.33. *Let $L : \mathbb{R}^n \to \mathbb{R}$ be bounded from below, λ-strongly convex and β-smooth on $\mathbb{R}^n$. Assume that $\mathbb{E}[\|\mathbf{v}_k\|_2^2] \leq \gamma < \infty$ for all $k \in \mathbb{N}_0$. Choose*

$$t_k = \frac{1}{\lambda}\frac{(k+1)^2 - k^2}{(k+1)^2}. \tag{4.28}$$

Then

$$\begin{aligned} \mathbb{E}[L(\boldsymbol{\theta}_k)] - L(\boldsymbol{\theta}^*) &\leq \frac{2\beta\gamma}{\lambda^2 k} = \mathcal{O}(k^{-1}), \\ \mathbb{E}\big[\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2\big] &\leq \frac{4\gamma}{\lambda^2 k} = \mathcal{O}(k^{-1}), \end{aligned} \tag{4.29}$$

where $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^n} L(\boldsymbol{\theta})$, and the expectations are over all $\mathbf{v}_i$, $i = 0, \ldots, k-1$.

Proof. As previously, expanding the squared norm of the new error $\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2^2$ when the SGD $\boldsymbol{\theta}_{k+1}$ is plugged in gives

$$\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2^2 = \|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|^2 - 2t_k\langle \mathbf{v}_k, \boldsymbol{\theta}_k - \boldsymbol{\theta}^*\rangle + t_k^2\|\mathbf{v}_k\|_2^2 .$$

Taking the expectation over $\mathbf{v}_k$ and replacing the term with ∇L using the λ-strong convexity property (4.24) similarly to (4.26), we obtain

$$\begin{aligned} \mathbb{E}_{\mathbf{v}_k}\big[\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2^2\big] &\leq \|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 - 2t_k\langle \nabla L(\boldsymbol{\theta}_k), \boldsymbol{\theta}_k - \boldsymbol{\theta}^*\rangle + t_k^2\gamma \\ &\leq \|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 + 2t_k\left(L(\boldsymbol{\theta}^*) - L(\boldsymbol{\theta}_k) - \frac{\lambda}{2}\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2\right) + t_k^2\gamma \\ &\leq (1 - \lambda t_k)\,\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 + t_k^2\gamma. \end{aligned} \tag{4.30}$$

Note that $\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2$ still depends on $\mathbf{v}_0, \ldots, \mathbf{v}_{k-1}$ and so is random. However, $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_k\mathbf{v}_k$ depends *explicitly* only on $\boldsymbol{\theta}_k$ and $\mathbf{v}_k$, but not on $\mathbf{v}_i$ or $\boldsymbol{\theta}_i$ for $i < k$. So the expectation in the left hand side of (4.30) should be understood in the *conditional* sense,

$$\mathbb{E}_{\mathbf{v}_k}\big[\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2^2 \,|\, \boldsymbol{\theta}_k\big] \leq (1 - \lambda t_k)\,\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 + t_k^2\gamma.$$

Now we can take the expectation of both sides over all $\mathbf{v}_1, \ldots, \mathbf{v}_{k-1}$, which can be written just with $\mathbb{E}$, since it acts upon all random vectors appearing up to the current step. This gives a recursion similar to the GD method,

$$\mathbb{E}\big[\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2^2\big] \leq (1 - \lambda t_k)\mathbb{E}[\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2] + t_k^2\gamma,$$

except the extra term $t_k^2\gamma$. This term can be taken care of by the choice of t_k. To shorten the calculations, let us define $e_0 = \|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\|_2^2$ and $e_k = \mathbb{E}[\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2]$ for $k \geq 1$. By induction, we can prove

$$\begin{aligned} e_{k+1} &\leq (1 - \lambda t_k)e_k + t_k^2\gamma \\ &\leq (1 - \lambda t_k)((1 - \lambda t_{k-1})e_{k-1} + t_{k-1}^2\gamma) + t_k^2\gamma \\ &\leq \cdots \leq e_0\prod_{j=0}^{k}(1 - \lambda t_j) + \gamma\sum_{j=0}^{k} t_j^2 \prod_{i=j+1}^{k}(1 - \lambda t_i). \end{aligned}$$

Choosing t_k as in the assumption of the theorem, we obtain

$$\prod_{i=j}^{k}(1-\lambda t_i) = \prod_{i=j}^{k} \frac{i^2}{(i+1)^2} = \frac{j^2}{(j+1)^2}\frac{(j+1)^2}{(j+2)^2}\cdots\frac{k^2}{(k+1)^2} = \frac{j^2}{(k+1)^2}.$$

Therefore,

$$\begin{aligned}
e_{k+1} &\le e_0\frac{0}{(k+1)^2} + \frac{\gamma}{\lambda^2}\sum_{j=0}^{k}\left(\frac{(j+1)^2-j^2}{(j+1)^2}\right)^2\frac{(j+1)^2}{(k+1)^2} \\
&= \frac{\gamma}{\lambda^2}\frac{1}{(k+1)^2}\sum_{j=0}^{k}\frac{(2j+1)^2}{(j+1)^2} && \left(\frac{(2j+1)^2}{(j+1)^2}\le 4\right) \\
&\le \frac{\gamma}{\lambda^2}\frac{4(k+1)}{(k+1)^2} \\
&= \frac{4\gamma}{\lambda^2(k+1)},
\end{aligned}$$

which is the second claim of the theorem – up to replacing $k+1$ by k. Finally, using the β-smoothness property (4.14),

$$L(\boldsymbol{\theta}_k) - L(\boldsymbol{\theta}^*) \le \langle\underbrace{\nabla L(\boldsymbol{\theta}^*)}_{=0}, \boldsymbol{\theta}_k - \boldsymbol{\theta}^*\rangle + \frac{\beta}{2}\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 = \frac{\beta}{2}\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2,$$

which, taking expectations, gives

$$\mathbb{E}[L(\boldsymbol{\theta}_k)] - L(\boldsymbol{\theta}^*) \le \mathbb{E}\left[\frac{\beta}{2}\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2\right] \le \frac{\beta\cdot 4\gamma}{2\lambda^2 k}.$$

□

Remark 4.34. *The specific choice of the learning rate* (4.28) *simplifies the calculations in the proof, but it may be rather impractical: in some* early *iterations it may happen that* $t_k > \frac{2\|\nabla L(\boldsymbol{\theta}_k)\|_2^2}{\beta\gamma}$*, and the error increases above the largest number representable on the computer, before it would start decreasing again. Therefore, we may need to take* t_k *smaller than* (4.28)*. Note that*

$$\frac{(k+1)^2-k^2}{\lambda(k+1)^2} = \frac{2(k+1)-1}{\lambda(k+1)^2} = \frac{2}{\lambda(k+1)} + \mathcal{O}((k+1)^{-2}) = \mathcal{O}(k^{-1}),$$

and one can choose instead $t_k = t_0/(k+1)$ *with a sufficiently small* t_0*. Similarly to GD, one can try decreasing values of* t_0 *(e.g.* $1/2, 1/4, 1/8$ *and so on) until the SGD method starts converging.*

Taking the square root of (4.29), we see that the SGD error converges with the same slow rate $\mathcal{O}(k^{-1/2})$ as the GD residual (4.16) with a much weaker assumption of β-smoothness only. This can be expected, as we lift the requirement of the exact gradient. Since the approximate gradient $\mathbf{v}_k$ can be computed much faster than the exact gradient $\nabla L_{\mathbf{D}}$ for large m, we can carry out more iterations in the same computing time. Moreover, variance reduction methods considered next allow one to recover the α-linear convergence.

End of lecture 14

4.2.6 SGD for empirical risk minimisation

For the empirical risk

$$L_{\mathbf{D}}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}),$$

we can construct the direction vector $\mathbf{v}_k$ by simply sampling $i_k = 1, \dots, m$ uniformly at random, and taking

$$\mathbf{v}_k = \nabla \tilde{\ell}_{\mathbf{x}_{i_k},y_{i_k}}(\boldsymbol{\theta}_k), \qquad i_k \sim \mathcal{U}(1, \dots, m).$$

Theorem 4.35. $\mathbb{E}[\mathbf{v}_k] = \nabla L_{\mathbf{D}}(\boldsymbol{\theta}_k)$ *as required in the SGD algorithm.*

Proof. Since the expectation over a discrete uniform distribution is simply the average, using also the linearity of the gradient, we get

$$\mathbb{E}[\mathbf{v}_k] = \mathbb{E}[\nabla \tilde{\ell}_{\mathbf{x}_{i_k},y_{i_k}}(\boldsymbol{\theta}_k)] = \nabla \mathbb{E}[\tilde{\ell}_{\mathbf{x}_{i_k},y_{i_k}}(\boldsymbol{\theta}_k)] = \nabla \left(\frac{1}{m}\sum_{i=1}^{m} \tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}_k)\right) = \nabla L_{\mathbf{D}}(\boldsymbol{\theta}_k).$$

□

Note that each iteration of SGD requires the computation of the gradient of the point loss function on **only one** data point, in contrast to GD where the complexity of the full gradient computation is at least proportional to m. In fact, for very large data, even the computation of the total loss $L_{\mathbf{D}}(\boldsymbol{\theta}_k)$ can be still slower than the computation of $\nabla \tilde{\ell}_{\mathbf{x}_{i_k},y_{i_k}}(\boldsymbol{\theta}_k)$ in **all** iterations! For this reason, it is sometimes said that SGD can optimise a function in less than a single evaluation of it.

This situation is rather an exception though. When the training data is limited (which is a much more often scenario), it is reasonable to make sure the optimisation method has used all of it. In this case, we replace the standard uniform distribution (with replacement) by a *shuffling* distribution. Namely, we let $\pi_1, \dots, \pi_m$ be distinct integer numbers, sampled from $\{1, \dots, m\}$ uniformly and randomly, but without replacement. Now the direction vector can be chosen as $\mathbf{v}_k = \nabla \tilde{\ell}_{\mathbf{x}_{\pi_k},y_{\pi_k}}(\boldsymbol{\theta}_k)$. However, what if we want to carry out more than m iterations? In this case, we can resample the shuffling indices $\pi_1, \dots, \pi_m$ and select the training data point with index π_{k-m} when $m < k \le 2m$, an index π_{k-2m} when $2m < k \le 3m$, and so on.

Algorithm 8 Stochastic Gradient Descent for ERM iterated over epochs

1: Start with a point $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, set $k = 0$.
2: **for** $\tau = 0, 1, \dots, n_{epochs}$ or until $L(\boldsymbol{\theta}_k)$ cannot be reduced further **do**
3: $\quad \pi_1, \dots, \pi_m$ = random_shuffle$(1, \dots, m)$.
4: $\quad$ **for** $i = 1, 2, \dots, m$ **do**
5: $\quad\quad$ Compute the direction vector $\mathbf{v}_k = \nabla \tilde{\ell}_{\mathbf{x}_{\pi_i},y_{\pi_i}}(\boldsymbol{\theta}_k)$.
6: $\quad\quad$ Choose a *learning rate* (length of the current step) $t_k > 0$.
7: $\quad\quad$ Set $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_k \mathbf{v}_k$.
8: $\quad\quad$ Increment the global iteration number $k := k + 1$.
9: $\quad$ **end for**
10: **end for**
11: **return** $\boldsymbol{\theta}_k \approx \boldsymbol{\theta}^*$.

Definition 4.36. *One complete pass of the training data through an optimisation algorithm is called an* ***epoch****.*

Let SGD sample i_k and $\mathbf{v}_k = \nabla\tilde{\ell}_{\mathbf{x}_{i_k},y_{i_k}}(\boldsymbol{\theta}_k)$ without replacement such that in m consecutive iterations each of $i_k = 1, \ldots, m$ is sampled exactly once. These m iterations make one epoch. When more than m iterations are conducted, SGD repeats the consideration of the same data points, but in a different "epoch" of the parameter values $\boldsymbol{\theta}_k$. This "epochal" SGD is formalised in Algorithm 8. Note also that the computing time of one epoch of SGD is about the computing time of one iteration of GD.

4.2.7 Early stopping of GD and SGD based on test loss

The default $L(\boldsymbol{\theta}_k)$ to monitor for convergence in Algorithm 8 is the training loss – the one that is being minimised. However, this may lead to the following problems:

- The loss may fluctuate near the minimum due to the stochastic gradient (Theorem 4.33 guarantees only the convergence in expectation).
- Minimising a training loss too much may be prone to overfitting.
- If $\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) > 0$, a threshold $L(\boldsymbol{\theta}_k) < \varepsilon$ may never be reached.

The **early stopping** of iterative optimisation algorithms monitors the ratio of the test losses instead, aiming at *significant decrease* of the test loss at *significantly different* iteration numbers that are insensitive to the stochastic fluctuations.

1. Choose the iteration gap $p \in \mathbb{N}$ and loss reduction threshold $q > 0$.
2. At each iteration with the global iteration number k, as soon as $k \geq p$,
   - if $\frac{L_{\mathbf{D}_{test}}(\boldsymbol{\theta}_k)}{L_{\mathbf{D}_{test}}(\boldsymbol{\theta}_{k-p})} > q$, stop and return $\boldsymbol{\theta}_{k_*}$, where $k_* = \underset{i=0,\ldots,k}{\arg\min}\, L_{\mathbf{D}_{test}}(\boldsymbol{\theta}_i)$.

4.2.8 Variance reduction methods: mini-batching and stochastic average gradient

The $\mathcal{O}(k^{-1/2})$ rate of convergence of the error established in Thm. 4.33 is again slow. In fact, even this rate is due to the learning rate t_k decreasing as $\mathcal{O}(k^{-1})$. With a constant learning rate (such as $t_k = 1/\beta$), the SGD Algorithm 7 never stops. Indeed, even at the exact solution $\boldsymbol{\theta}_k = \boldsymbol{\theta}^*$ where $\nabla L(\boldsymbol{\theta}^*) = 0$, the *variance* of the descent direction vector

$$\mathrm{Var}[\mathbf{v}_k] := \mathbb{E}\big[\|\mathbf{v}_k - \mathbb{E}\big[\mathbf{v}_k]\|_2^2\big] = \mathbb{E}\big[\|\mathbf{v}_k - \nabla L(\boldsymbol{\theta}^*)\|_2^2\big] = \mathbb{E}\big[\|\mathbf{v}_k\|_2^2\big],$$

is nonzero, and the algorithm will keep changing $\boldsymbol{\theta}_k$, and drift away from the exact solution. This motivates the development of *variance reduction* methods that can ensure the following.

Definition 4.37. *The increment vectors $\mathbf{v}_k \in \mathbb{R}^n$ are said to satisfy the **Variance Reduction (VR) property** if*

$$\mathbb{E}[\|\mathbf{v}_k - \nabla L(\boldsymbol{\theta}_k)\|_2^2] \to 0 \quad as \quad k \to \infty. \tag{4.31}$$

In addition to just decreasing t_k, another popular simple method is **mini-batching**. Instead of using just the single pointwise loss, $\mathbf{v}_k = \nabla\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}_k)$, this method uses the average of several $\nabla\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}_k)$ in each iteration to get a better estimate of the full gradient $\nabla L(\boldsymbol{\theta}_k)$:

$$\mathbf{v}_k = \frac{1}{|B_k|} \sum_{i \in B_k} \nabla\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}_k), \tag{4.32}$$

where $B_k \subset \{1, \dots, m\}$ is a set of random indices, and $|B_k|$ is the size of B_k. This method is especially useful when multiple gradients can be evaluated in parallel. When B_k is sampled uniformly with replacement, the variance of this gradient estimator is inversely proportional to the "batch size" $|B_k|$, so we can decrease the variance by increasing the batch size. However, satisfying (4.31) requires taking $|B_k| \to \infty$, which increases the computational cost towards the last iterations, potentially approaching that of GD.

To circumvent this problem, modern variance reduction methods maintain some estimate $\mathbf{v}_k$ of the full gradient $\nabla L(\boldsymbol{\theta}_k)$ **between** the iterations, and use the pointwise loss gradient $\nabla \tilde{\ell}_{\mathbf{x}_i, y_i}(\boldsymbol{\theta}_k)$ only to update the estimate $\mathbf{v}_k$ instead of recomputing it from scratch in each iteration. The so-called **Stochastic Average Gradient (SAG)** method uses

$$\mathbf{v}_k = \frac{1}{m} \sum_{i=1}^{m} \mathbf{g}_k^i, \quad \text{where} \quad \mathbf{g}_k^i = \nabla \tilde{\ell}_{\mathbf{x}_i, y_i}(\boldsymbol{\theta}_p) \quad \text{for some } p \leq k.$$

Initialising $\mathbf{g}_0^i$ with zero, SAG samples $i_k \in \{1, \dots, m\}$, and updates

$$\mathbf{g}_k^i = \begin{cases} \nabla \tilde{\ell}_{\mathbf{x}_i, y_i}(\boldsymbol{\theta}_k), & \text{if } i = i_k, \\ \mathbf{g}_{k-1}^i, & \text{otherwise.} \end{cases} \tag{4.33}$$

To get rid of the summation of m terms in every iteration, we can notice that

$$\begin{aligned} \mathbf{v}_k &= \frac{1}{m} \sum_{i=1, i \neq i_k}^{m} \mathbf{g}_k^i + \frac{1}{m} \mathbf{g}_k^{i_k} = \frac{1}{m} \sum_{i=1, i \neq i_k}^{m} \mathbf{g}_{k-1}^i + \frac{1}{m} \mathbf{g}_k^{i_k} \qquad (\mathbf{g}_{k-1}^i \text{ did not change for } i \neq i_k) \\ &= \mathbf{v}_{k-1} - \frac{1}{m} \mathbf{g}_{k-1}^{i_k} + \frac{1}{m} \mathbf{g}_k^{i_k}. \end{aligned} \tag{4.34}$$

The final SAG pseudocode can now be written as in Algorithm 9. It can be shown that in expectation SAG converges α-linearly if $L(\boldsymbol{\theta})$ is both β-smooth and λ-strongly convex, similarly to Thm. 4.28. However, in contrast to GD the time of one iteration does not grow with m.

Algorithm 9 Stochastic Average Gradient (SAG)

1: Start with a point $\boldsymbol{\theta}_0 \in \mathbb{R}^n$, set $\mathbf{v}_0 = \mathbf{g}_0^i = 0$, $i = 1, \dots, m$, and $k = 0$.
2: **for** $k = 0, 1, \dots,$ until $L(\boldsymbol{\theta}_k)$ cannot be reduced further **do**
3: Sample $i_k \in \{1, \dots, m\}$ at random.
4: Subtract the previous gradient $\tilde{\mathbf{v}} = \mathbf{v}_k - \frac{1}{m} \mathbf{g}_k^{i_k}$.
5: Compute the new gradient $\mathbf{g}_{k+1}^{i_k} = \nabla \tilde{\ell}_{\mathbf{x}_{i_k}, y_{i_k}}(\boldsymbol{\theta}_k)$, $\quad \mathbf{g}_{k+1}^i = \mathbf{g}_k^i$ for $i \neq i_k$.
6: Add the new gradient $\mathbf{v}_{k+1} = \tilde{\mathbf{v}} + \frac{1}{m} \mathbf{g}_{k+1}^{i_k}$.
7: Choose a *learning rate* (length of the current step) $t_k > 0$.
8: Set $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_k \mathbf{v}_{k+1}$.
9: **end for**
10: **return** $\boldsymbol{\theta}_k \approx \boldsymbol{\theta}^*$.

End of lecture 15

4.2.9 Derivative-free methods. Perceptron algorithm for halfspaces

Numerical differentiation. For complicated functions, the analytical derivation of the gradient can be tedious. Recall, for instance, the degree-4 polynomial in the GD notebook. Fortunately, for many loss functions we can *approximate* its partial derivatives by using finite differences similar to those used for the solution of differential equations in Numerical Analysis.

Let $\mathbf{e}_j = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^n$ be the so-called *jth unit* vector ($j = 1, \ldots, n$), where 1 is located in position j, and the other elements are zero. Given a differentiable loss function $L : \mathbb{R}^n \to \mathbb{R}$, we can write the Taylor series for the function $\tilde{L}(\tau) := L(\boldsymbol{\theta} + \tau\mathbf{e}_j)$, which gives

$$L(\boldsymbol{\theta} + \tau\mathbf{e}_j) = L(\boldsymbol{\theta}) + \langle \nabla L(\boldsymbol{\theta}), \tau\mathbf{e}_j \rangle + \mathcal{O}(\tau^2) = L(\boldsymbol{\theta}) + \tau \frac{\partial L}{\partial \theta_j}(\boldsymbol{\theta}) + \mathcal{O}(\tau^2), \qquad \tau \in \mathbb{R}.$$

Truncating $\mathcal{O}(\tau^2)$ terms for sufficiently small τ, we can approximate

$$(\nabla L(\boldsymbol{\theta}))_j = \frac{\partial L}{\partial \theta_j}(\boldsymbol{\theta}) \approx \frac{L(\boldsymbol{\theta} + \tau\mathbf{e}_j) - L(\boldsymbol{\theta})}{\tau}, \qquad j = 1, \ldots, n. \tag{4.35}$$

Pros

- We do not need to compute derivatives ourselves, only the loss function.

Cons

- One gradient evaluation needs $(n+1)$ function evaluations: $L(\boldsymbol{\theta})$ and all of $L(\boldsymbol{\theta} + \tau\mathbf{e}_j)$, which can be slow.
- The error in (4.35) due to truncating the Taylor series is $\mathcal{O}(\tau)$, which warrants taking τ as small as possible. However, ...
- ... if τ is too small, then $L(\boldsymbol{\theta})$ and $L(\boldsymbol{\theta} + \tau\mathbf{e}_j)$ indistinguishable in the computer that rounds the numbers off to only a finite number of digits.

To balance the Taylor truncation and the round-off errors, a rule of thumb suggestion for the jth partial derivative is

$$\tau = \sqrt{\epsilon} \cdot \max(|\theta_j|, \sqrt{\epsilon}) \cdot \mathrm{sign}(\theta_j),$$

where ϵ is the round-off error, which is typically about $2 \cdot 10^{-16}$ on modern computers and programming languages including Python. The max term prevents τ from vanishing at zero θ_j.

Perceptron algorithm. This algorithm was originally proposed by Rosenblatt in 1958 as a heuristic method that construct a sequence of iterates $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \ldots,$ that make the halfspaces classifier more and more correct with respect to the condition (4.3). The pseudocode is written in Algorithm 10. We see that Algorithm 10 is also free from gradients.

Algorithm 10 Perceptron algorithm of Rosenblatt for learning halfspaces

Require: A separable training set $\mathbf{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ (in homogeneous form)
Ensure: Halfspaces parameter vector $\boldsymbol{\theta}_k$ such that $y_i\langle \boldsymbol{\theta}_k, \mathbf{x}_i \rangle > 0$ for all $i = 1, \ldots, m$.

1: Initialise $\boldsymbol{\theta}_0 = (0, \ldots, 0) \in \mathbb{R}^{n+1}$.
2: **for** $k = 0, 1, 2, \ldots$ **do**
3:     **if** $\exists i_k$ such that $y_{i_k}\langle \boldsymbol{\theta}_k, \mathbf{x}_{i_k} \rangle \leq 0$ **then**
4:         Compute $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + y_{i_k}\mathbf{x}_{i_k}$.
5:     **else**
6:         Stop and return $\boldsymbol{\theta}_k$.
7:     **end if**
8: **end for**

However, the Perceptron algorithm is in fact similar to SGD! The pointwise loss of the ERM (4.4) $\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}) = -\min(y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle,\ 0)$ is continuously differentiable everywhere except $y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle = 0$, namely[5],

$$\nabla\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}) = \begin{cases} 0, & y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle > 0, \\ -y_i\mathbf{x}_i, & y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle < 0, \\ \text{undefined}, & y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle = 0. \end{cases}$$

Now, as long as $y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle \neq 0$, Line 4 of the Perceptron algorithm implements exactly the SGD update

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_k\nabla\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta}_k),$$

with the learning rate $t_k = 1$. The selection of i_k in Line 3 of Algorithm 10 is tuneable similarly to the general SGD: it can be sampled every time at random from the entire range $1,\dots,m$ as in the baseline Algorithm 7, or looped over within an epoch as in Algorithm 8.

Since the loss (4.4) is not even β-smooth, we cannot use Theorems 4.20, 4.28 or 4.33 to analyse the convergence of the Perceptron algorithm. However, we can prove a specialised theorem that the Perceptron algorithm converges actually always in a *finite* number of iterations.

Theorem 4.38. *Assume that $\mathbf{D} = \{(\mathbf{x}_1,y_1),\dots,(\mathbf{x}_m,y_m)\}$ is separable as per Def. 4.5, let*

$$B = \min_{\boldsymbol{\theta}\in\mathbb{R}^{n+1}} \|\boldsymbol{\theta}\|_2 \quad \text{such that} \quad y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle \geq 1, \quad \text{and} \quad \rho = \max_i \|\mathbf{x}_i\|_2 \quad \forall i = 1,\dots,m. \tag{4.36}$$

Then the Perceptron algorithm stops after at most $(\rho B)^2$ iterations, and returns $y_i\langle\boldsymbol{\theta}_k,\mathbf{x}_i\rangle > 0$.

Proof. Let $\boldsymbol{\theta}^*$ be the vector realising the min in (4.36). The idea of the proof is to show that after performing k iterations, the cosine of the angle between $\boldsymbol{\theta}_k$ and $\boldsymbol{\theta}^*$ is at least $\sqrt{k}/(\rho B)$. Since the cosine is at most 1, as soon as $\sqrt{k}/(\rho B)$ exceeds 1, or equivalently, $k \geq (\rho B)^2$, the method should converge to a vector collinear to $\boldsymbol{\theta}^*$.

Firstly, we show by induction that $\langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_k\rangle \geq k$. Indeed, at $k = 0$ we have $\boldsymbol{\theta}_0 = (0,\dots,0)$, and $\langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_0\rangle = 0 \geq 0$. For a given k, we can write

$$\begin{aligned}\langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_{k+1}\rangle - \langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_k\rangle &= \langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_{k+1}-\boldsymbol{\theta}_k\rangle \\ &= \langle\boldsymbol{\theta}^*,y_{i_k}\mathbf{x}_{i_k}\rangle = y_{i_k}\langle\boldsymbol{\theta}^*,\mathbf{x}_{i_k}\rangle \\ &\geq 1.\end{aligned}$$

Plugging in the induction assumption $\langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_k\rangle \geq k$, we obtain $\langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_{k+1}\rangle \geq k+1$.

Next, expanding the squares, we get

$$\begin{aligned}\|\boldsymbol{\theta}_{k+1}\|_2^2 = \|\boldsymbol{\theta}_k + y_{i_k}\mathbf{x}_{i_k}\|_2^2 &= \|\boldsymbol{\theta}_k\|_2^2 + 2y_{i_k}\langle\boldsymbol{\theta}_k,\mathbf{x}_{i_k}\rangle + y_{i_k}^2\|\mathbf{x}_{i_k}\|_2^2 \\ &\leq \|\boldsymbol{\theta}_k\|_2^2 + \rho^2,\end{aligned}$$

where the last inequality is due to the fact that for the chosen i_k it holds $y_{i_k}\langle\boldsymbol{\theta}_k,\mathbf{x}_{i_k}\rangle \leq 0$, and the norm of $\mathbf{x}_{i_k}$ is at most ρ. Now, since $\|\boldsymbol{\theta}_0\|_2^2 = 0$, we can again use the induction to get

$$\|\boldsymbol{\theta}_{k+1}\|_2^2 \leq (k+1)\rho^2.$$

Combining this with $\langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_k\rangle \geq k$ and $\|\boldsymbol{\theta}^*\|_2 = B$, we obtain

$$\cos\angle(\boldsymbol{\theta}^*,\boldsymbol{\theta}_k) = \frac{\langle\boldsymbol{\theta}^*,\boldsymbol{\theta}_k\rangle}{\|\boldsymbol{\theta}^*\|_2\|\boldsymbol{\theta}_k\|_2} \geq \frac{k}{B\sqrt{k}\rho} = \frac{\sqrt{k}}{B\rho}.$$

□

[5] In fact, a subdifferential of $\tilde{\ell}_{\mathbf{x}_i,y_i}(\boldsymbol{\theta})$ exists also at $y_i\langle\boldsymbol{\theta},\mathbf{x}_i\rangle = 0$, but this is beyond the scope of this course.

Remark 4.39. *The parameter vector* $\boldsymbol{\theta}^*$ *realising the* min *in* (4.36) *is the solution to* (4.4). *Clearly, if* $y_i\langle\boldsymbol{\theta}^*, \mathbf{x}_i\rangle \geq 1$*, then it holds that* $y_i\langle\boldsymbol{\theta}^*, \mathbf{x}_i\rangle > 0$ *and* $L_{\mathbf{D}}(\boldsymbol{\theta}^*/B) = 0$*, and* $\|\boldsymbol{\theta}^*/B\|_2 = 1$. *In contrast, a suitably rescaled solution of* (4.4) *satisfies only the constraints of* (4.36). *Indeed, any* $\boldsymbol{\theta}^*$ *satisfying* $y_i\langle\boldsymbol{\theta}^*, \mathbf{x}_i\rangle > 0$ *can be divided by* $\min_i [y_i\langle\boldsymbol{\theta}^*, \mathbf{x}_i\rangle] > 0$*, and makes* $y_i\langle\boldsymbol{\theta}^*, \mathbf{x}_i\rangle \geq 1$ *for all* i. *However, this is not necessarily a minimal-norm solution!*

If $(\rho B)^2 < n+1 = \mathrm{VCdim}(\mathcal{H}_n^{hs})$, one can think that the Perceptron algorithm is able to bypass the fundamental theorem of statistical learning by converging after using fewer than $\mathrm{VCdim}(\mathcal{H}_n^{hs})$ data points. However, $\mathrm{VCdim}(\mathcal{H}_n^{hs}) = n+1$ holds only for *general* halfspaces. Assuming $\|\mathbf{x}_i\|_2^2\|\boldsymbol{\theta}^*\|_2^2 \leq (\rho B)^2 < n+1$ changes the hypothesis class, which may now have a smaller VC-dimension.

Example 4.40. *Classify emails represented by the binary term-to-document vectors as "spam" and "not spam". Consider the following dataset.*

		"and"	"offer"	"the"	"of"	"sale"	y_i
	$\hat{\mathbf{x}}_1$	1	1	0	1	1	$+1$ (spam)
$\mathbf{D}_{train}$	$\hat{\mathbf{x}}_2$	0	0	1	1	0	-1 (not spam)
	$\hat{\mathbf{x}}_3$	0	1	1	0	0	$+1$ (spam)
	$\hat{\mathbf{x}}_4$	1	0	0	1	0	-1 (not spam)
$\mathbf{D}_{test}$	$\hat{\mathbf{x}}_5$	1	0	1	0	1	$+1$ (spam)
	$\hat{\mathbf{x}}_6$	1	0	1	1	0	-1 (not spam)

The Perceptron algorithm, using $\mathbf{x}_1, \ldots, \mathbf{x}_4$ *as the training set (recall that* $\mathbf{x}_i = [1, \hat{\mathbf{x}}_i]$*), converges in* 4 *iterations to*

$$\boldsymbol{\theta}^* = (0,\ 0, 2, 0, -1, 1).$$

This can be interpreted that the words "offer" and "sale", which have positive coefficients, are indicative of "spam", the word "of" with the negative coefficient is indicative of "not spam", and the other words are neutral. The resulting halfspaces rule classifies the test dataset correctly:

$$\langle\boldsymbol{\theta}^*, \mathbf{x}_5\rangle = 1, \qquad \langle\boldsymbol{\theta}^*, \mathbf{x}_6\rangle = -1.$$

End of lecture 16

4.2.10 Second-order methods: Newton's method (non-examinable)

The gradient descent was derived by minimising the loss $L(\boldsymbol{\theta}_k + t\hat{\mathbf{v}})$ with respect to the one-dimensional parameter t, stepping along the line in the direction $\hat{\mathbf{v}}$. As a by-product, this required us to compute the first derivatives in ∇L. For this reason, GD-based methods are called first-order. *Second-order* methods build a quadratic model around the loss to seek for the minimum, and require second derivatives for that.

The most significant second-order algorithm is the Newton's method, which was originally developed for single-variable root finding problems – but it can also be applied to optimisation (in n-dimensional space).

Assuming that the loss function $L : \mathbb{R}^n \to \mathbb{R}$ that is subject to minimisation is twice differentiable, we can define both the gradient

$$\nabla L(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\boldsymbol{\theta})}{\partial \theta_n} \end{bmatrix} \in \mathbb{R}^n,$$

and the **Hessian**

$$\nabla^2 L(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 L(\boldsymbol{\theta})}{\partial\theta_1\partial\theta_1} & \cdots & \frac{\partial^2 L(\boldsymbol{\theta})}{\partial\theta_1\partial\theta_n} \\ \vdots & & \vdots \\ \frac{\partial^2 L(\boldsymbol{\theta})}{\partial\theta_n\partial\theta_1} & \cdots & \frac{\partial^2 L(\boldsymbol{\theta})}{\partial\theta_n\partial\theta_n}. \end{bmatrix} \in \mathbb{R}^{n\times n}. \tag{4.37}$$

If $L(\boldsymbol{\theta})$ is twice continuously differentiable, the order of differentiation does not matter, and the Hessian is symmetric. Now, given the current iterate $\boldsymbol{\theta}_k$, we can write a multivariate Taylor series around $\boldsymbol{\theta}_k$ truncated at the second-order term, and express

$$L(\boldsymbol{\theta}_k + \mathbf{v}_k) = \underbrace{L(\boldsymbol{\theta}_k) + \langle \nabla L(\boldsymbol{\theta}_k), \mathbf{v}_k\rangle + \frac{1}{2}\langle \mathbf{v}_k, \nabla^2 L(\boldsymbol{\theta}_k)\mathbf{v}_k\rangle}_{\tilde{L}(\mathbf{v}_k)} + \text{h.o.t}, \tag{4.38}$$

where "h.o.t" stands for "higher-order terms". Now we derive $\mathbf{v}_k$ by solving necessary optimality conditions for the approximate quadratic model $\tilde{L}(\mathbf{v}_k)$:

$$\nabla \tilde{L}(\mathbf{v}_k) = \nabla L(\boldsymbol{\theta}_k) + \nabla^2 L(\boldsymbol{\theta}_k)\mathbf{v}_k = 0. \tag{4.39}$$

Assuming that the Hessian is invertible, we obtain Algorithm 11.

Algorithm 11 Newton's method for optimisation

1: Initialise $\boldsymbol{\theta}_0 \in \mathbb{R}^n$.
2: **for** $k = 0, 1, \ldots,$ until $L(\boldsymbol{\theta}_k)$ cannot be reduced further **do**
3: Solve linear equations $(\nabla^2 L(\boldsymbol{\theta}_k))\mathbf{v}_k = \nabla L(\boldsymbol{\theta}_k)$ on the *direction vector* $\mathbf{v}_k$.
4: Choose a *learning rate* $t_k > 0$ ▷ the default value solving (4.39) is $t_k = 1$.
5: Set $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_k\mathbf{v}_k$.
6: **end for**

Under sufficiently strong assumptions the Newton's method converges much faster than GD algorithms.

Definition 4.41. *A method producing a sequence of iterates $\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \ldots$ is called α-**quadratically convergent** to $\boldsymbol{\theta}^*$ if $\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2 \to 0$ as $k \to \infty$, and there exists $q > 0$ such that for all k, $\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2 \le \alpha\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2$.*

Theorem 4.42 (Newton's convergence)**.** *Let $L(\boldsymbol{\theta}) : \Omega_R \to \mathbb{R}$ be bounded from below and twice continuously differentiable with β-smooth $\nabla L(\boldsymbol{\theta})$ on $\Omega_R := \{\boldsymbol{\theta} : \|\boldsymbol{\theta} - \boldsymbol{\theta}^*\|_2 \le R\}$, where $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^n} L(\boldsymbol{\theta})$, and invertible Hessian $\nabla^2 L(\boldsymbol{\theta})$ on Ω_R. Let $t_k = 1$. Then there exists $r > 0$ such that for any $\boldsymbol{\theta}_0 : \|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\|_2 \le r$ Newton's method converges α-quadratically to $\boldsymbol{\theta}^*$.*

Proof. We start with rewriting the Newton's iteration as

$$\nabla^2 L(\boldsymbol{\theta}_k)(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*) = \nabla^2 L(\boldsymbol{\theta}_k)\big(\boldsymbol{\theta}_k - (\nabla^2 L(\boldsymbol{\theta}_k))^{-1}\nabla L(\boldsymbol{\theta}_k) - \boldsymbol{\theta}^*\big) = \nabla^2 L(\boldsymbol{\theta}_k)(\boldsymbol{\theta}_k - \boldsymbol{\theta}^*) - \nabla L(\boldsymbol{\theta}_k) + \nabla L(\boldsymbol{\theta}^*),$$

where $\nabla L(\boldsymbol{\theta}^*)$ is added since it is zero. Similarly to Lemma 4.18, we can use the fundamental theorem of calculus to write

$$\nabla L(\boldsymbol{\theta}_k) - \nabla L(\boldsymbol{\theta}^*) = \int_0^1 \frac{d}{dt}\nabla L(\boldsymbol{\theta}^* + t(\boldsymbol{\theta}_k - \boldsymbol{\theta}^*))dt = \int_0^1 \nabla^2 L(\boldsymbol{\theta}^* + t(\boldsymbol{\theta}_k - \boldsymbol{\theta}^*))(\boldsymbol{\theta}_k - \boldsymbol{\theta}^*)dt.$$

Plugging this in the previous equation gives

$$\nabla^2 L(\boldsymbol{\theta}_k)(\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*) = \int_0^1 \left[\nabla^2 L(\boldsymbol{\theta}_k) - \nabla^2 L(\boldsymbol{\theta}^* + t(\boldsymbol{\theta}_k - \boldsymbol{\theta}^*))\right] dt \cdot (\boldsymbol{\theta}_k - \boldsymbol{\theta}^*). \tag{4.40}$$

For the term under the integral we use the β-smoothness of ∇L, that is

$$\|\nabla^2 L(\boldsymbol{\theta}_k) - \nabla^2 L(\boldsymbol{\theta}^* + t(\boldsymbol{\theta}_k - \boldsymbol{\theta}^*))\|_2 \le \beta \|\boldsymbol{\theta}_k - (\boldsymbol{\theta}^* + t(\boldsymbol{\theta}_k - \boldsymbol{\theta}^*))\|_2,$$

where in the left hand side we use the matrix norm. Multiplying Equation (4.40) by $\nabla^2 L(\boldsymbol{\theta}_k)^{-1}$, and using the matrix norm property, we get

$$\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2 \le \|\nabla^2 L(\boldsymbol{\theta}_k)^{-1}\|_2 \beta \int_0^1 (1-t)dt \|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2 \le \frac{\sigma\beta}{2}\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2^2, \tag{4.41}$$

where $\sigma = \max_{\boldsymbol{\theta}\in\Omega_R} \|\nabla^2 L(\boldsymbol{\theta})^{-1}\|_2 < \infty$ by the assumption of invertibility of the Hessian. Now if we take any $r = \min(2q/(\sigma\beta), R)$ with $q < 1$, and assume that $\|\boldsymbol{\theta}_0 - \boldsymbol{\theta}^*\|_2 \le r$, we get

$$\|\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}^*\|_2 < \alpha \|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2,$$

which establishes:

1. the induction that $\|\boldsymbol{\theta}_k - \boldsymbol{\theta}^*\|_2 \le r \le R$ holds for any $k \ge 0$, and
2. the α-linear convergence needed for the first part of the α-quadratic convergence. The second part comes from (4.41).

□

Advantages of Newton's method.

- (Locally) quadratic convergence: doubling the number of correct digits each iteration.
- Tuning-free: within Ω_r, the best learning rate is $t_k = 1$ for any problem.

Disadvantages of Newton's method.

- No global convergence: Newton's method may diverge far from $\boldsymbol{\theta}^*$ where GD converges.
- The need for $\nabla^2 L$, which may be hard to derive and compute.

Newton's method for Empirical Risk Minimisation. In the ERM framework

$$L(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \tilde{\ell}_{\mathbf{x}_i, y_i}(\boldsymbol{\theta}),$$

the Hessian can be computed by averaging pointwise loss Hessians similarly to the gradient,

$$\nabla^2 L(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla^2 \tilde{\ell}_{\mathbf{x}_i, y_i}(\boldsymbol{\theta}).$$

Note that in general we **cannot** make a Stochastic Newton's method similarly to SGD: the Hessian of each individual $\tilde{\ell}_{\mathbf{x}_i, y_i}$ may be (and is actually likely) **not** invertible – only the full Hessian $\nabla^2 L(\boldsymbol{\theta})$ is.

For the Squared Error loss

$$\tilde{\ell}_{\mathbf{x},y}(\boldsymbol{\theta}) = (h_{\boldsymbol{\theta}}(\mathbf{x}) - y)^2,$$

where $h_{\boldsymbol{\theta}}(\mathbf{x})$ is the prediction rule of a known class, the Hessian of L can be computed from the gradient and Hessian of $h_{\boldsymbol{\theta}}$. Using the chain rule, we obtain

$$\nabla \tilde{\ell}_{\mathbf{x},y}(\boldsymbol{\theta}) = 2(h_{\boldsymbol{\theta}}(\mathbf{x}) - y)\nabla_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}}(\mathbf{x}),$$

and

$$\nabla^2 \tilde{\ell}_{\mathbf{x},y}(\boldsymbol{\theta}) = 2\nabla_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}}(\mathbf{x})\nabla_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}}(\mathbf{x})^\top + 2(h_{\boldsymbol{\theta}}(\mathbf{x}) - y)\nabla^2_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}}(\mathbf{x}),$$

where $\nabla_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}}(\mathbf{x})$ is understood as a column vector, such that the first term is a rank-1 matrix.

In many cases, we can omit the second term and modify Algorithm 11 into the so-called **Gauss-Newton** method: instead of solving $(\nabla^2 L(\boldsymbol{\theta}_k))\mathbf{v}_k = \nabla L(\boldsymbol{\theta}_k)$ in each iteration, we solve

$$H_k \mathbf{v}_k = \nabla L(\boldsymbol{\theta}_k), \quad \text{where} \quad H_k = \frac{2}{m}\sum_{i=1}^m \nabla_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}_k}(\mathbf{x}_i)\nabla_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}_k}(\mathbf{x}_i)^\top.$$

Another family of methods that allow one to compute only the gradients (and are thus similar to GD in terms of implementation complexity) is *Quasi-Newton* methods. These methods keep and update an approximation to the Hessian by using only the gradients at previous iterations.

Consider specifically the linear regression problem with $\tilde{\ell}_{\mathbf{x},y} = (\langle \boldsymbol{\theta}, \mathbf{x}\rangle - y)^2$. From Section 4.2.3 we know that the gradient is linear, $\nabla L_{\mathbf{D}}(\boldsymbol{\theta}) = A\boldsymbol{\theta} - \mathbf{b}$, where $A = \frac{2}{m}XX^\top$ and $\mathbf{b} = \frac{2}{m}X\mathbf{y}$. Hence, the Hessian $\nabla^2 L_{\mathbf{D}}(\boldsymbol{\theta}) = A$ is just the matrix A, for any $\boldsymbol{\theta}$. This gives the following observations:

- $\lambda_{\min}(A) > 0 \Rightarrow \lambda$-strong convexity of $L_{\mathbf{D}} \Rightarrow$ invertibility of $\nabla^2 L_{\mathbf{D}}$ for a well-defined Newton's method.
- In this case, Newton's method is equivalent to solving the linear equations, and converges in 1 iteration to the exact solution.

4.3 Non-parametric prediction methods

Historically, machine learning was developed as a largely heuristic area – without continuous optimisation of parameters $\boldsymbol{\theta}$. However, the simplicity of methods presented in this section makes them still popular when the computing speed is the primary criterion, and one needs at least some "reasonable" solution, fast. Moreover, non-parametric (also known as *memorising*) methods may actually give more intuitive prediction rules on categorical data (such as text).

4.3.1 Decision trees

A decision tree constructs a prediction $h : \mathcal{X} \to \mathcal{Y}$ by following a sequence of queries about the given domain data point $\mathbf{x} \in \mathcal{X}$, similarly to a flowchart. Traditional decision trees require **no training** (= optimisation) as such, only **counting** of data points satisfying a certain criterion.

Definition 4.43 (Non-examinable recap on graphs)**.**

- *A **graph** G is a pair of finite sets (V, E) where V is a set of vertices, and E is a set of edges. Each edge connects one or two vertices.*

- *A* ***path*** *from* $v_1 \in V$ *to* $v_i \in V$ *is a subset of vertices* $\{v_1, \dots, v_i\} \subset V$ *and edges* $\{e_1, \dots, e_{i-1}\} \subset E$ *connected consecutively without repeating edges,* $v_1 e_1 v_2 \cdots v_{i-1} e_{i-1} v_i$.

- *A* ***circuit*** *is a path with* $v_1 = v_i$.

- G *is called* ***connected*** *if there exists a path from any* $v_i \in V$ *to any* $v_j \in V$.

- *A connected graph* G *without circuits is called a* ***tree****. A vertex* $v_\ell \in V$ *is called a* ***leaf*** *of the tree if* v_ℓ *is connected to only one edge.*

- G *is called* ***directed*** *if edges are oriented, i.e. existence of a path* $v_i e_k v_j$ *does not imply existence of the path* $v_j e_k v_i$*. In this case we can say that* e_k ***emerges*** *from* v_i.

Now we can formulate a formal definition of a decision tree.

Definition 4.44. *Let* $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$*, consider a directed tree* $G = (V, E)$*, where:*

- *each non-leaf vertex* $v_b \in V$ *corresponds to some* ***branching coordinate*** x_j, $j \in \{1, \dots, n\}$;

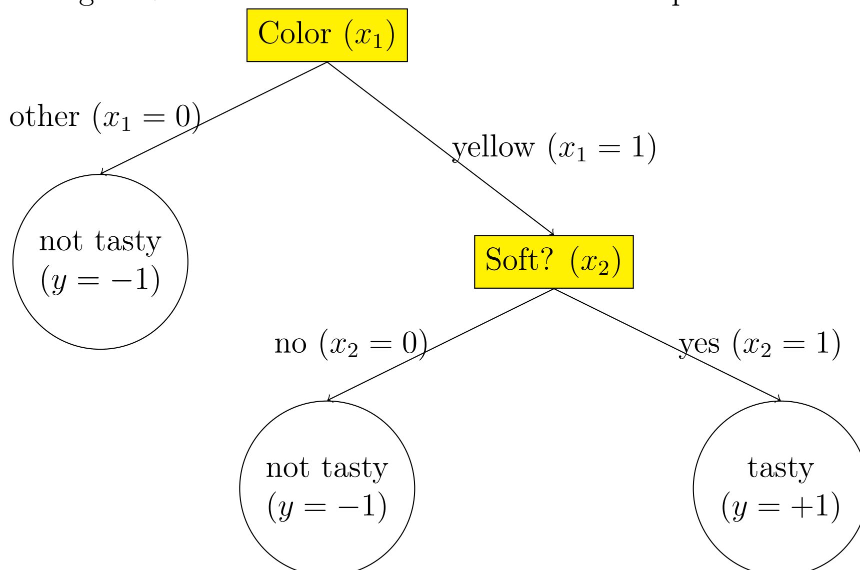
- *edges* e_k *emerging from a non-leaf vertex* v_b *correspond to non-overlapping exhaustive* ***conditions*** *on* x_j *associated with* v_b, $e_k \cap e_{k'} = \emptyset$ *for* $k \neq k'$, $\cup e_k = \mathbb{R}$;

- *each leaf* $v_\ell \in V$ *corresponds to a label* $y \in \mathcal{Y}$.

The ***decision tree prediction rule*** $h : \mathbb{R}^n \to \mathcal{Y}$ *maps* $\mathbf{x}$ *to a label* y *such that* $x_1, \dots, x_n, y$ *belong to a unique path in* G.

Example. Consider a decision tree for predicting whether a given fruit is tasty ($y = 1$) or not ($y = -1$). We can use two queries: whether the colour of the fruit is yellow, and whether it is soft. We can thus introduce the data point as $\mathbf{x} = (x_1, x_2) \in \{0, 1\}^2$, where x_1 denotes whether the colour is yellow ($x_1 = 1$) or not ($x_1 = 0$), and x_2 denotes whether the fruit is soft ($x_2 = 1$) or not ($x_2 = 0$). Now the decision tree can be drawn as shown in Figure 7.

Figure 7: Decision tree for the fruit tastiness prediction.



Another fun example of decision trees is the Twenty Questions game (`http://20q.net/`).

Note that the decisions are irreversible. If the colour is not yellow, the tree immediately predicts that the fruit is not tasty without additional queries. The main advantage of decision trees is their simplicity – in terms of both interpretation and computational speed. The main disadvantage is the irreversibility, leading to error intolerance and potentially biased selection.

For simplicity, in the rest of this lecture we consider $\mathbf{x} \in \{0,1\}^n$ and $y \in \{-1,1\}$.

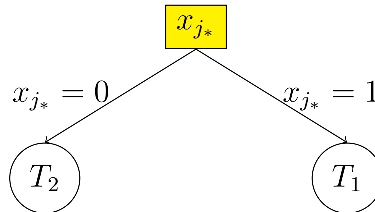
Automatic selection of the branching coordinate. To alleviate the irreversibility problem, we can now use some training data $\mathbf{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$, and find the order in which $x_1, \ldots, x_n$ are queried by maximizing some gain in the decay of the empirical risk.

One popular decision tree algorithm is known as "ID3" (short for "Iterative Dichotomizer 3"). This algorithm starts with the initial call $\mathtt{ID3}(\mathbf{D}, \{1, \ldots, n\})$, and builds a tree as in Def. 4.44 by descending into each branch (edge) recursively. The pseudocode is shown in Algorithm 12. It features a function $\mathtt{Gain}(\mathbf{D}, j)$ computing the gain in the loss decay due to selecting x_j into the current non-leaf vertex.

Algorithm 12 $\mathtt{ID3}(\mathbf{D}, \mathcal{J})$

Require: Training set $\mathbf{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$, set $\mathcal{J} \subset \{1, \ldots, n\}$ of coordinates to consider, gain function $\mathtt{Gain}(\mathbf{D}, j)$.

1: **if** $\mathcal{J} = \emptyset$ or $y_1 = \cdots = y_m$ **then**
2:     **return** a leaf with the most frequent label y among $y_1, \ldots, y_m$.
3: **end if**
4: Let $j_* = \arg\max_{j \in \mathcal{J}} \mathtt{Gain}(\mathbf{D}, j)$.
5: Let T_1 be the tree returned by $\mathtt{ID3}(\{(\mathbf{x}, y) \in \mathbf{D} : x_{j_*} = 1\}, \mathcal{J} \backslash j_*)$.
6: Let T_2 be the tree returned by $\mathtt{ID3}(\{(\mathbf{x}, y) \in \mathbf{D} : x_{j_*} = 0\}, \mathcal{J} \backslash j_*)$.
7: **return** the following tree:



Gain function. To decrease the loss on the training dataset $\mathbf{D}$ as fast as possible, we define the gain as the difference of training losses Before and After branching (i.e. selecting into a non-leaf vertex) of the coordinate x_j,

$$\mathtt{Gain}(\mathbf{D}, j) = L_{before}(\mathbf{D}) - L_{after}(\mathbf{D}, j), \tag{4.42}$$

respectively. Assuming that $(\mathbf{x}_i, y_i) \in \mathbf{D}$ are independent identically distributed samples of $(X, Y) \sim \mathbb{P}$, a suitable loss is the probability of sampling a wrong label. Before branching, we are to return the most frequent label within $\mathbf{D}$ at Line 2 of Alg. 12. The probability of sampling within a discrete set $\mathbf{D}$ is just the frequency

$$\mathbb{P}_{\mathbf{D}}[Y = 1] = \frac{|\{i \in \{1, \ldots, m\} : y_i = 1\}|}{|\mathbf{D}|}, \qquad \mathbb{P}_{\mathbf{D}}[Y = -1] = \frac{|\{i : y_i = -1\}|}{|\mathbf{D}|} = 1 - \mathbb{P}_{\mathbf{D}}[Y = 1],$$

where $|S|$ is the number of elements in a set S. Since we choose the dominant label, the probability that it is wrong is just the minimum of the two probabilities above,

$$L_{before}(\mathbf{D}) = C(\mathbb{P}_{\mathbf{D}}[Y=1]), \quad \text{where} \quad C(p) := \min\{p, 1-p\}, \quad p \in [0,1]. \tag{4.43}$$

After branching, for each choice of $x_j = 1$ or $x_j = 0$, the loss is the probability of returning the wrong label, *conditional* on the choice of x_j. The total probability of the wrong prediction is the sum of those, weighted by the marginal probabilities of observing $x_j = 1$ and $x_j = 0$. Therefore, after branching the training loss is

$$L_{after}(\mathbf{D}, j) = \mathbb{P}_{\mathbf{D}}[X_j = 1] C(\mathbb{P}_{\mathbf{D}}[Y=1|X_j=1]) + \mathbb{P}_{\mathbf{D}}[X_j = 0] C(\mathbb{P}_{\mathbf{D}}[Y=1|X_j=0]), \tag{4.44}$$

where

$$\mathbb{P}_{\mathbf{D}}[X_j = c] = \frac{|\{i : x_{i,j} = c\}|}{|\mathbf{D}|}, \qquad \mathbb{P}_{\mathbf{D}}[Y=1|X_j=c] = \frac{|\{i : y_i = 1,\ x_{i,j} = c\}|}{|\{i : x_{i,j} = c\}|}, \qquad c = 0, 1,$$

are the probabilities of the corresponding events, and $x_{i,j}$ is the jth element of the vector $\mathbf{x}_i$.

Note that if the data dimension n is large, the decision tree can consist of up to 2^n vertices, which can be infeasibly huge and lead to overfitting. Therefore, usually decision trees need a more or less manual pruning of features or early stopping. For example, if $\mathbf{x}$ is a landscape image, letting x_1, x_2 and so on be colours of individual pixels is nearly surely hopeless: the first hundred of pixels can be just slightly different shades of the blue sky. In contrast, if x_1 is the average colour of the entire image, x_2, x_3 are average colours over top and bottom half of the image, and so on, a useful decision tree can be built upon only a few features.

End of lecture 17

4.3.2 K-nearest neighbours

The idea of the method of nearest neighbours is to memorise the training set and then to predict the label of any new (test) data on the basis of the labels of its closest neighbours in the training set.

Often the nearest neighbour is fast to compute, for example, in the Web search where distances are based on links. In contrast to the linear regression or halfspaces classifier (even including features such as the logistic regression), where the prediction rule lives in some class of functions, the nearest neighbour method computes a label without searching for a predictor within some predefined class of functions.

We assume that the domain points $\mathbf{x} \subset \mathcal{X}$ belong to a metric space $\mathcal{X}$ with some distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_+$. Let $\mathbf{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ be the given dataset. For any given test point $\mathbf{x}$, the K-Nearest Neighbours algorithm returns a vector of $K \geq 1$ labels $(y_{\pi_1}, \ldots, y_{\pi_K})$, where $\pi_1, \ldots, \pi_K \in \{1, \ldots, m\}$ are the unique indices of the points $\mathbf{x}_1, \ldots, \mathbf{x}_m$ that are **nearest** to $\mathbf{x}$ with respect to the distance d,

$$d(\mathbf{x}, \mathbf{x}_{\pi_1}) \leq d(\mathbf{x}, \mathbf{x}_{\pi_2}) \leq \cdots \leq d(\mathbf{x}, \mathbf{x}_{\pi_K}) \leq \cdots \leq d(\mathbf{x}, \mathbf{x}_{\pi_m}).$$

The pseudocode can be formalised as shown in Algorithm 13. A particularly simple version is the **1-Nearest Neighbour**, which returns $h_{\mathbf{D}}(\mathbf{x}) = y_{\pi_1}$.

Algorithm 13 K-Nearest Neighbours

Require: Dataset $\mathbf{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$, distance function $d(\mathbf{x}, \mathbf{x}')$, test data point $\mathbf{x}$, number of labels to predict $K \geq 1$.

Ensure: Labels of the K nearest neighbours to $\mathbf{x}$.

1: Compute the vector of distances $\mathbf{r} = (d(\mathbf{x}, \mathbf{x}_1), \dots, d(\mathbf{x}, \mathbf{x}_m)) \in \mathbb{R}^m$.
2: Sort $\mathbf{r}$ ascending and record the permutation index $\boldsymbol{\pi} \in \mathbb{N}^m$ such that $r_{\pi_1} \leq \cdots \leq r_{\pi_m}$.
3: **return** the top K labels $(y_{\pi_1}, \dots, y_{\pi_K})$.

Alternatively, if one is still interested in a single output, one can compute the prediction as the average of the K labels of the nearest neighbours, $h_{\mathbf{D}}(\mathbf{x}) = \frac{1}{K}\sum_{i=1}^{K} y_{\pi_i}$. In the latter case, the output is not guaranteed to be in the admissible label set $\mathcal{Y}$, for example, it may not be integer. Therefore, the averaged K-nearest neighbours method is mainly used for regression instead of classification, since the regression prediction can be any real number.

Another popular predictor is the geometric center of mass of the K nearest neighbours,

$$h_{\mathbf{D}}(\mathbf{x}) = \frac{\sum_{i=1}^{K} d(\mathbf{x}, \mathbf{x}_{\pi_i}) y_{\pi_i}}{\sum_{i=1}^{K} d(\mathbf{x}, \mathbf{x}_{\pi_i})}.$$

Note that the 1-Nearest Neighbour algorithm can be implemented without sorting the vector of distances $\mathbf{r}$, by simply taking the index π_1 of the minimal element, $r_{\pi_1} = \min_{i=1,\dots,m} r_i$. This method needs $\mathcal{O}(m)$ comparison operations in addition to computing the m distances. In the K-Nearest Neighbours version, one can also compute the minimal elements K times, resulting in $\mathcal{O}(Km)$ comparison operations. However, algorithms such as Intro Sort[6] can sort a vector in $\mathcal{O}(m \log m)$ comparisons, which is faster if K is large.

Expected risk of 1-Nearest Neighbour (1-NN) classifier. We analyse the convergence of the 1-NN method in a simplified scenario.

Assumption 4.45. *(a) We consider the classification problem with* $y \in \{-1, 1\}$.

(b) Any data pair $(\mathbf{x}, y)$ *is an independent sample from* $(X, Y) \sim \mathbb{P}$, *where* X *is distributed uniformly on* $[0, 1]^n$.

(c) There exists a conditional probability function $\eta(\mathbf{x}) := \mathbb{P}[Y = 1 | X = \mathbf{x}]$ *that is Lipschitz,* $|\eta(\mathbf{x}) - \eta(\mathbf{x}')| \leq c\|\mathbf{x} - \mathbf{x}'\|_\infty$ *with* $c > 0$ *for any* $\mathbf{x}, \mathbf{x}' \in [0, 1]^n$, *where* $\|\mathbf{x}\|_\infty := \max_{j=1,\dots,n} |x_j|$.

For the pointwise loss function we can take the probability of sampling the wrong label, $\ell(h(\mathbf{x})) := \mathbb{P}[Y \neq h(\mathbf{x}) | X = \mathbf{x}]$. If $\eta(\mathbf{x})$ is known, the Bayes-optimal prediction rule is

$$h_{Bayes}(\mathbf{x}) = \begin{cases} 1, & \eta(\mathbf{x}) > 1/2, \\ -1, & \text{otherwise}, \end{cases}$$

with $\ell(h_{Bayes}(\mathbf{x})) = \min\{\eta(\mathbf{x}), 1 - \eta(\mathbf{x})\}$. The expected risk of h_{Bayes} (which is the bias loss) is $L_{bias} = \mathbb{E}[\min\{\eta(X), 1 - \eta(X)\}]$. For the practical 1-NN, we get the following result.

Theorem 4.46 (non-examinable)**.** *Under Assumption 4.45 with dataset* $\mathbf{D} = \{(\mathbf{x}_i, y_i)\}_{1 \leq i \leq m}$, *the probability of a wrong prediction for the 1-NN classifier with* $d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_\infty$ *is*

$$\mathbb{E}[\ell(h_{\mathbf{D}}(X))] = \mathbb{P}\left[h_{\mathbf{D}}(X) \neq Y\right] \leq 2L_{bias} + cm^{-\frac{1}{n+1}} + c\exp(-m^{\frac{1}{n+1}}).$$

[6]Called by default in the recent `numpy.argsort` function.

Proof (non-examinable). By construction of 1-NN, the probability of a wrong prediction is

$$\ell(h_{\mathbf{D}}(\mathbf{x})) = \mathbb{P}\left[Y \neq h_{\mathbf{D}}(\mathbf{x}) | X = \mathbf{x}\right] = \mathbb{P}\left[Y \neq y_{\pi_1} | X = \mathbf{x}\right] = \mathbb{P}\left[Y \neq Y' | X = \mathbf{x}, X' = \mathbf{x}_{\pi_1}\right],$$

where (X', Y') is an independent copy of (X, Y). The above probability expands as follows:

$$\begin{aligned}
\mathbb{P}\left[Y \neq Y' | X = \mathbf{x}, X' = \mathbf{x}_{\pi_1}\right] &= \mathbb{P}\left[Y = 1 | X = \mathbf{x}\right] \mathbb{P}\left[Y' = -1 | X' = \mathbf{x}_{\pi_1}\right] \\
&\quad + \mathbb{P}\left[Y = -1 | X = \mathbf{x}\right] \mathbb{P}\left[Y' = 1 | X' = \mathbf{x}_{\pi_1}\right] \\
&= \eta(\mathbf{x})(1 - \eta(\mathbf{x}_{\pi_1})) + (1 - \eta(\mathbf{x}))\eta(\mathbf{x}_{\pi_1}) \\
&= \eta(\mathbf{x})(1 + \eta(\mathbf{x}) - \eta(\mathbf{x}) - \eta(\mathbf{x}_{\pi_1})) + (1 - \eta(\mathbf{x}))(\eta(\mathbf{x}_{\pi_1}) + \eta(\mathbf{x}) - \eta(\mathbf{x})) \\
&= 2\eta(\mathbf{x})(1 - \eta(\mathbf{x})) + (\eta(\mathbf{x}) - \eta(\mathbf{x}_{\pi_1}))(2\eta(\mathbf{x}) - 1).
\end{aligned}$$

Taking the modulus of the right hand side and using the triangle inequality, we get

$$\begin{aligned}
\mathbb{P}\left[Y \neq Y' | X = \mathbf{x}, X' = \mathbf{x}_{\pi_1}\right] &\leq 2\eta(\mathbf{x})(1 - \eta(\mathbf{x})) + |\eta(\mathbf{x}) - \eta(\mathbf{x}_{\pi_1})| \cdot |2\eta(\mathbf{x}) - 1| \\
&\leq 2\min\{\eta(\mathbf{x}), 1 - \eta(\mathbf{x})\} + c\|\mathbf{x} - \mathbf{x}_{\pi_1}\|_\infty \cdot 1.
\end{aligned}$$

Taking the expectation over X, we get

$$\mathbb{E}[\ell(h_{\mathbf{D}}(X))] = \mathbb{P}\left[Y \neq Y' | X' = \mathbf{x}_{\pi_1}\right] \leq 2L_{bias} + c\mathbb{E}[\|X - \mathbf{x}_{\pi_1}\|_\infty]. \tag{4.45}$$

The last expectation can be split into two contributions: that over a hypercube around $\mathbf{x}_{\pi_1}$ stepping at most some $\varepsilon < 1$ away from $\mathbf{x}_{\pi_1}$ in any direction, and that over the rest of $[0, 1]^n$.

$$\mathbb{E}[\|X - \mathbf{x}_{\pi_1}\|_\infty] = \mathbb{E}[\|X - \mathbf{x}_{\pi_1}\|_\infty | \|X - \mathbf{x}_{\pi_1}\|_\infty \leq \varepsilon] + \mathbb{E}[\|X - \mathbf{x}_{\pi_1}\|_\infty | \|X - \mathbf{x}_{\pi_1}\|_\infty > \varepsilon].$$

Now $\|\mathbf{x} - \mathbf{x}_{\pi_1}\|_\infty$ can be upper-bounded by ε in the first term, and by just 1 in the second term,

$$\mathbb{E}[\|X - \mathbf{x}_{\pi_1}\|_\infty] \leq \varepsilon \cdot \underbrace{\mathbb{P}[\|X - \mathbf{x}_{\pi_1}\|_\infty \leq \varepsilon]}_{\leq 1} + 1 \cdot \mathbb{P}[\|X - \mathbf{x}_{\pi_1}\|_\infty > \varepsilon].$$

However, $\mathbf{x}_{\pi_1}$ was selected by minimising the distance to X over all data points. That is, $\|X - \mathbf{x}_{\pi_1}\|_\infty > \varepsilon$ means that *all* $\mathbf{x}_1, \ldots, \mathbf{x}_m$ should be at least ε away from X, and since they are independent,

$$\mathbb{P}[\|X - \mathbf{x}_{\pi_1}\|_\infty > \varepsilon] = \prod_{i=1}^m \mathbb{P}[\|X - \mathbf{x}_i\|_\infty > \varepsilon] = \prod_{i=1}^m \left(1 - \mathbb{P}[\|X - \mathbf{x}_i\|_\infty \leq \varepsilon]\right).$$

In the last term, since X is uniformly distributed,

$$\mathbb{P}[\|X - \mathbf{x}_i\|_\infty \leq \varepsilon] = \int_{\|\mathbf{x} - \mathbf{x}_i\|_\infty \leq \varepsilon} 1 d\mathbf{x} = \begin{cases} (2\varepsilon)^n, & \mathbf{x}_i \text{ is at least } \varepsilon \text{ away from all walls of } [0,1]^n, \\ \cdots & \\ \varepsilon^n, & \mathbf{x}_i \text{ is at a corner of } [0,1]^n. \end{cases}$$

Altogether,

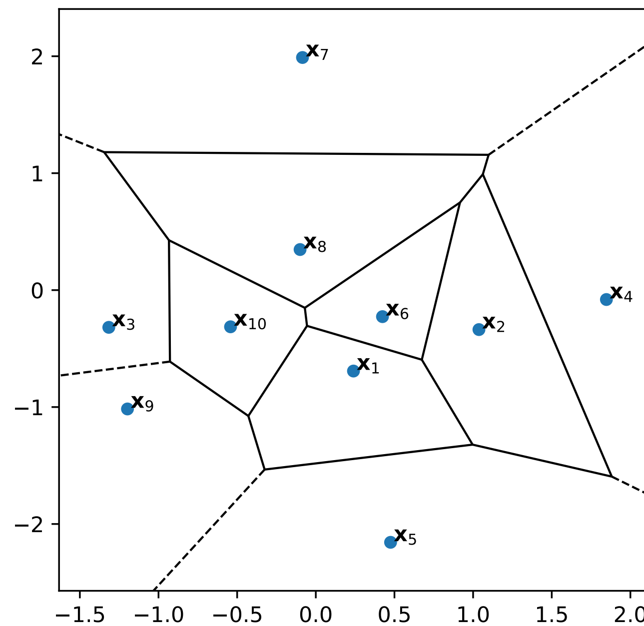
$$\mathbb{E}[\|X - \mathbf{x}_{\pi_1}\|_\infty] \leq \varepsilon + \prod_{i=1}^m (1 - \varepsilon^n) \leq \varepsilon + \exp(-m\varepsilon^n).$$

Choosing $\varepsilon = m^{-1/(n+1)}$, and plugging the previous estimate into (4.45) gives the result. □

We see that the expected risk of the 1-NN classifier converges to at most 2 bias losses with the amount of data $m \to \infty$. However, when the data dimension n is large, this convergence can be rather slow.

When $\mathbf{x} \in \mathbb{R}^n$, the 1-NN method returns the same label (that of the closest $\mathbf{x}_{\pi_1}$) for any $\mathbf{x}' \in \mathbb{R}^n$ which are sufficiently close to $\mathbf{x}$, such that they are still closer to $\mathbf{x}_{\pi_1}$ than to other points in the training set. Hence, one can ask for a critical $\mathbf{x}' \in \mathbb{R}^n$ which are exactly the same distance from two training points. These $\mathbf{x}'$ collectively form the so-called *decision boundaries*.

In turn, all of the $\mathbf{x} \in \mathbb{R}^n$ which are labelled with the same y_{π_1} constitute the so-called *Voronoi cell* around $\mathbf{x}_{\pi_1}$. The union of all Voronoi cells and decision boundaries performs the *Voronoi Tessellation* of the space. An example for $n = 2$ is shown below: blue circles are training data points, black lines are decision boundaries, and the white areas are Voronoi cells.



End of lecture 18

Summary

- Supervised learning aims to find a generalisable prediction rule by making the prediction accurate on known (*supervised*) labels in the data.

- Many prediction rules (linear, halfspaces, Bayes) for classification and regression depend on continuous parameters $\boldsymbol{\theta} \in \mathbb{R}^n$. These require a continuous optimisation (also called *training*) using appropriate algorithms, for example, from the Gradient Descent family. These algorithms may require plenty of computing time, but continuously parametrised prediction rules allow for more general data and better generalisation accuracy.

- Other prediction rules (decision trees and nearest neighbours) are free from continuous parameters, and require only counting and discrete optimisation. These methods are usually faster since no training is needed, but may be limited to a particular class of data (e.g. 0 or 1 values only), and are often more difficult to tune the accuracy.